Release 9.2
LassoGuide

Contents

List of Tables

Foreword

Lasso has been around for decades, and will be around for decades to come.

Although other languages have become known as the more "popular" languages, the criteria used to measure popularity have rarely been challenged. Lasso came out of the Mac world—a world filled with artists, educators, and medical practitioners—in the nascent days of the Internet. It has been used by a wide variety of people, whether they be occasional programmers, hard-core developers or anything in between. It has been used to create systems which support and manage the Internet, to drive billions of dollars in sales, and to monitor simple events of everyday life.

Lasso presented the world with a novel and logical way to script, develop, and write code poetry. Tens of thousands of people owe their livelihoods to Lasso, and almost all live their lives unaware of Lasso's importance. Lasso has become a foundation block of the world's digital future, steadily inspiring joy and value to those who use it regularly.

This latest version of Lasso, Lasso 9, makes a monumental leap in its underlying theory and architecture to secure the future of the platform. With countless hours of effort, a group of dedicated contributors have worked together to ensure that a strong central manual—a canon—would exist for this new language. These individuals include:

Kyle Jessup
Fletcher Sandbeck
Jonathan Guthrie
Paul Higgins
Brad Lindsay
Steve Piercy
Michael Collins
Aaron Smalser
Eric Knibbe

It is with great pleasure that I thank this group for their concerted and tireless efforts over several years to bring this endeavour to bear. May this be the book which shines a light for the Lasso development community, allowing it to be seen from around the world and shared with others.

Long Live Lasso!

Sean Stephens
CEO
LassoSoft Inc.

Welcome to **Lasso**Guide

Lasso is a powerful programming language used to drive millions of web pages from servers around the world. It has an easy-to-master syntax and allows fast, flexible development and scripting. Lasso can be used in many ways, and as a language, provides a virtually infinite set of shortcuts for achieving development goals.

This guide is meant to serve as both an introduction and comprehensive manual to Lasso and Lasso Server 9.2. The material in this guide will evolve and improve along with Lasso. The most up-to-date version of this documentation containing all improvements can be found at http://lassoguide.com/.

Organization of This Guide

This guide is divided up into eight parts covering all aspects of the Lasso programming language.

- *Getting Started with Lasso* introduces the basic features of the Lasso language and server, as well as instructions for installing and configuring Lasso.

- *Language Elements* covers the syntax and features of the Lasso language. Read this thoroughly for a complete understanding of how Lasso code is structured.

- *Data Handling*, *System Input and Output*, *Application Development*, and *External Communication* detail the capabilities of the libraries that ship with Lasso, divided into appropriate categories. Method definitions and examples of common use cases are included.

- *Database Operations* describes Lasso's database connection interface. Basic database operations as well as pointers about specific database types are covered.

- *Extending Lasso* includes tutorials and references for adding your own functionality using Lasso's C and Java APIs.

Explanations, method definitions, and code examples are arranged within the text to teach you the Lasso platform step-by-step. An index is also available to help find information about a particular language element.

Conventions Used in This Guide

There are many code samples used throughout this guide. References to methods, types, or traits and small snippets of code inlined with other text are set in a monospace typeface, e.g. `sample_method` or `short code snippet`. References to variable names or to values will be in double quotes "like this".

Longer blocks of sample code will be slightly offset from the surrounding text and will have syntax highlighting applied to them. The result produced by running the code will be displayed using line comments. If the result fits on one line, then a line comment in the form of `// => Value Produced` will be used. If multiple lines are needed, then the first line will just have `// =>` while all subsequent lines will start with a line comment and space, followed by the value for that line. For example:

```
// Single-line value produced
2 + 3
// => 5

// Multi-line value produced
'Line one.' + '\n' + 'Line two.'

// =>
// Line one.
// Line two.
```

For examples involving running commands from the command line, a shell prompt (**$>**) will be used. Any output to standard out that is generated from the command will be shown below the command as you would see it in your terminal. For examples of issuing Lasso commands from the interactive interpreter, a Lasso prompt (**>:**) will be used, and any values produced from running those commands will be shown using the line comment convention as outlined above for sample code blocks.

Additional Resources

Here are some additional resources you may find useful:

Lasso Reference[1]
: Reference to the built-in types, methods, and traits.

LassoTalk[2]
: The online Lasso community/email list is a great place to ask questions and get answers.

TagSwap[3]
: Methods, types, and traits created by members of the Lasso community to solve common problems.

LassoSoft Website[4]
: The latest information about Lasso.

Lasso source code repository[5]
: An SVN repository containing source code for a number of Lasso components.

LassoGuide PDF[6]
: The current version of LassoGuide in PDF format.

LassoGuide source[7]
: The Git repository containing the full LassoGuide source.

---

[1] http://www.lassosoft.com/lassoDocs/languageReference
[2] http://www.lassotalk.com/
[3] http://www.lassosoft.com/tagswap
[4] http://www.lassosoft.com/
[5] http://source.lassosoft.com/svn/lasso/lasso9_source/trunk/
[6] http://lassoguide.com/LassoGuide9.2.pdf
[7] https://github.com/LassoSoft/LassoGuide

Part I

Getting Started with Lasso

A Taste of Lasso

1.1 Lasso Basics

Welcome to Lasso! This guide is meant to assist you in diving into the language. It assumes you have prior programming experience and that you have properly installed Lasso and configured Lasso to work with your web server. (See the appropriate installation instructions for your operating system in the *index_server*.)

The examples in this particular chapter can all be run inside the **Lasso Quick Code** area in the Lasso Server Admin web application. (Be sure to leave the "<?lasso" and "Auto-collect" checkboxes checked.)

1.1.1 Hello World

The obligatory "Hello World" example is extremely simple in Lasso:

```
'Hello, world.'
// => Hello, world.
```

As you can see, the code just declares the string **'Hello,world.'**, and that value is produced when you run the code.

Here is what's going on behind the scenes: Lasso is an object-oriented language, so everything is an object. Every object has a member method named **asString** that it can implement. (Left unimplemented, it just returns the name of the object's type.) Any statement that is just an object by itself or produces an object will have that object's **asString** method implicitly called, and that value will therefore be produced. (For a counter-example, assignment statements don't produce an object and so will *not* cause any value to be produced.)

In our example, we used a string literal to create a string object. Since that statement produces an object, Lasso then calls the **string->asString** method on that object which has been implemented to return the value of the string.

1.1.2 Captures

This implicit call of the **asString** method is important for code inside a capture. Captures form the backbone of Lasso: all Lasso code executes inside a capture. (See the **Captures** chapter for more information.) They can also be used for code within a conditional. However, not all captures output the values generated by calls to **asString**. In the example below, the values in the **if** statement won't be displayed, but the values in the **loop** statement will be:

```
if(false) => {
    'will never get to me'
else(false)
    "will never get to me either"
else
    `got to me, but I won't be displayed`
}

loop(5) => {^
    loop_count + ', '
```

```
  ^}

  // => 1, 2, 3, 4, 5,
```

Here we have a basic **if** statement using a normal capture (`{ ... }`) and a **loop** statement using an auto-collected capture (`{^ ... ^}`). The auto-collect capture concatenates all the values produced using **asString** and returns them while the normal capture just ignores them.

1.1.3 Variables

There are two different types of variables in Lasso: local variables and thread variables. Local variables are lexically scoped while thread variables are available anywhere in the execution of the thread they are defined in. Here is an example of creating and using both types of variables:

```
local(localVar) = 30    // Creates a local variable
var(threadVar)  = 12    // Creates a thread variable
$threadVar + #localVar

// => 42
```

One handy feature of local variables is decompositional assignment. This makes it easy to assign values from array-like types into locals:

```
local(a, b, c) = array('rhino', 'runs', 'rapidly')

#a
// => rhino
#b
// => runs
#c
// => rapidly
```

For more information on variables, see the *Variables* chapter.

1.2 Lasso Language Features

The Lasso programming language has a number of great features that make coding in it enjoyable. This tutorial will scratch the surface of some of the best features of Lasso while also giving an introduction to defining methods, types, and traits.

1.2.1 Type Constraints

Lasso allows programmers to specify that a variable they create can only store objects of a specific type or trait. The following example creates a local variable that can only store integer values:

```
local(myInt::integer) = 5
#myInt = 8
#myInt = '44'
// => // Throws an error since we are trying to assign a string
```

This syntax also works for type-constraining thread variables.

1.2.2 Methods

Defining your own methods in Lasso is extremely easy. The following example returns the time of day ("morning", "afternoon", or "evening") given a specified hour:

```
define time_of_day(hour::integer) => {
    // Check to make sure the hour value is valid
    fail_if(#hour < 0 or #hour > 23,
        error_code_invalidParameter,
        error_msg_invalidParameter + ': hour must be between 0 and 23'
    )

    if(#hour >= 5 and #hour < 12) => {
        return 'morning'
    else(#hour >= 12 && #hour < 17)
        return 'afternoon'
    else
        return 'evening'
    }
}
```

The first line contains the **define** keyword, followed by the name for the method and its the parameter list in parentheses (the method signature), followed by the association operator (**=>**) and an open brace. All the code between that open brace and its matching closing brace is the capture associated with the method, which is executed when the method is called.

The method starts by making sure that the hour passed to it is valid. If it is, then the code that determines the time of day will run and return the proper value.

Notice that the type constraint in the method definition's signature constrains **hour** to be an integer object. This enables a handy feature in Lasso called "multiple dispatch". Let's say we want a similar function that accepts a date object. No need for a different method name; instead we can define that method like this:

```
define time_of_day(datetime::date=date) => time_of_day(#datetime->hour)
```

This defines a second method that also has the name "time_of_day", but accepts a date object and returns the value of calling the **time_of_day** method that takes an integer, passing it the hour of the date object. This method definition doesn't have a capture associated with it. If your method is going to just return the value of an expression, you can put that expression to the right of the association operator. It's equivalent to this code:

```
define time_of_day(datetime::date=date) => {
    return time_of_day(#datetime->hour)
}
```

Besides multiple dispatch, methods can also have optional parameters and named parameters. In the **time_of_day** example method that takes a date object, the **datetime** parameter is actually optional: the current date and time will be used if no value is passed. See the *Methods* chapter for more information on parameter definition and use.

1.2.3 Types

Lasso is an object-oriented language that comes with a number of core types already defined, but you can also create your own types. Below is a simple type definition to demonstrate how:

```
define person => type {
    data public nameFirst::string
    data
        public nameMiddle::string,
        public nameLast::string
```

```
    public onCreate(first::string, last::string, middle::string='') => {
        .'nameFirst'    = #first
        .'nameMiddle'   = #middle
        self->'nameLast' = #last
    }

    public nameFirstLast => self->nameFirst + ' ' + .nameLast
}
```

The type definition starts off with the **define** keyword followed by the type name, the association operator, the **type** keyword, and finally the braces for the capture containing the type definition code. The definition starts with two data sections that define three data members for the type. Two member methods are then defined using the access level keyword **public** instead of the **define** keyword. The **onCreate** methods are special for types: they define type creator methods that are automatically called when you create instances of your type. The following code would use the **person->onCreate** method to create an object of type "person" and then output their first and last name:

```
local(cool_dude) = person('Bill', 'Doerrfeld')  // "middle" is defined as an optional parameter
#cool_dude->nameFirstLast

// => Bill Doerrfeld
```

Types in Lasso also have single inheritance and can implement and import traits, described next. For more information on types, see the *Types* chapter.

1.2.4 Traits

Traits are a great way to package up and make available reusable code for types. If there is functionality that needs to be shared between different types, it can be packaged up as a trait instead of creating a different implementation for each type or forcing a complex inheritance scheme.

Defining traits is similar to defining types. The following example is a slightly modified version of the definition for **trait_positionallyKeyed**:

```
define ex_trait_positionallyKeyed => trait {
    import trait_doubleEnded

    require size()::integer, get(key::integer)

    provide
        first()  => (.size > 0 ? .get(1) | null),
        second() => (.size > 1 ? .get(2) | null),
        last()   => (.size > 0 ? .get(.size) | null)
}
```

The definition starts with the **define** keyword followed by the name of the trait, the association operator, the **trait** keyword, and then a set of braces enclosing the trait definition. There are then three sections that start with their own keyword:

import
 This section can contain a comma-separated list of traits that the current trait implements. In this case, because our trait implements a **first** and **last** method, it can import **trait_doubleEnded** which allows for types that use this trait to also get the methods that **trait_doubleEnded** provides. (Alternatively, if trait A imports trait B but doesn't implement trait B's required traits, then any type that imports trait A must also meet the requirements for trait B by implementing the missing methods.)

require
 This section can contain a comma-separated list of method signatures that must be implemented by any type wanting to

import this trait. In this case it requires a `size` method that returns an integer and a `get` method that takes a single integer parameter.

provide
   This section can contain a comma-separated list of method definitions. This is where the reusable code is defined that types importing this trait will be able to access.

The result of this trait definition is that types defining a `size` method and a `get` method can import this trait and have the following methods available as member methods: `first`, `second`, `last`. For more information on defining and using traits, see the *Traits* chapter.

1.2.5 Query Expressions

Query expressions allow programmers to create highly readable code that can do complex manipulation of data sets. Here is a quick example:

```
local(data_set) = (: 42, 11, 72, 13, 14, 88, 92, 35)

with number in #data_set
where #number % 2 == 0
skip 1
take 3
sum #number

// => 174
```

Every query expression starts as `with` *newLocalName* `in` *trait_queriable*, where `newLocalName` becomes the name of a local variable only accessible in the query expression, and `trait_queriable` is an object whose type implements and imports `trait_queriable`, such as the `staticarray` in the example.

After this initial `with` clause, a query expression can have zero or more operation clauses that each start with their own keyword. The example above uses three: `where` which filters the input using an expression, `skip` which skips a set number of elements, and `take` which returns a set number of elements. Order does matter.

Every query expression ends with one action clause that specifies what should be done for each iteration. In this case, we're using the `sum` action to add each value in the iteration together. Other actions are `min`, `max`, `average`, and `select`, which returns a new set of values rather than a single value; and `do`, which runs a block of code for each value.

The example above iterates over each element in the staticarray and first tests to see if it is an even number. It then skips the first even number it finds and only executes the `sum` action on the next three. The end result is that it adds 72, 14, and 88 together.

The best part about query expressions is that most of the actions are lazily executed. This means you can store a query expression in a variable, and it will wait to be executed until the value for the variable is expected. For a more thorough description, see the *Query Expressions* chapter.

1.3 Serving Lasso

There are lots of ways to create websites using Lasso. There are a number of frameworks available[8], plus other ones not listed on that page, that can help. You could even easily create your own framework. In this chapter, we will look at how easy it is to use files that embed Lasso in HTML code, and examine a simple packaging architecture that Lasso provides called LassoApps.

---

[8] http://www.lassosoft.com/Lasso-frameworks

1.3.1 Embedding Lasso Code

Lasso is designed to make it easy to intermix HTML and Lasso code in a single file. Just create a normal HTML file with the ".lasso" extension and you can add Lasso code between the following delimiters: `[ ... ]`, `<?lasso ... ?>`, or `<?= ... ?>`.

For example, you could place the following code in a file named "test.lasso" in the server's web root:

```
<?lasso
    local(now) = date
?>
<!DOCTYPE html>
<html>
<head>
    <title>Test Lasso</title>
</head>
<body>
    <p>
        This page was loaded on [#now->format(`E, MMMM d, YYYY`)] at <?= #now->format(`h:mm:ss a`) ?>.
    </p>
    It is currently
    [if(date->hour >= 5 and date->hour < 12) => {^]
        morning!
    [else(date->hour >= 12 && date->hour < 17)]
        afternoon!
    [else]
        evening!
    [^}]
</body>
```

Now all you need to do is use a web browser to request the URL from the server (e.g. *http://example.com/test.lasso*) and it will use Lasso to return an HTML page with something like the following content:

```
This page was loaded on Wed, July 31, 2013 at 10:36:42 AM
It is currently morning!
```

1.3.2 Creating LassoApps

A LassoApp is a bundle of Lasso source files, HTML files, images, and other media into a single deployable unit. While developing, this deployable unit is a folder with the above contents, but you can also choose to compile the bundle and have a binary file to distribute.

To create a LassoApp, create a directory in the "LassoApps" directory of your instance's home directory. By default, URLs for the LassoApp will start with **/lasso9/*AppName*/**. The discussion that follows will assume an app named "AddressBook" with URLs that look like *http://example.com/lasso9/AddressBook*.

Special Files

_install Files

The first time an instance loads a LassoApp, it will execute any files with a file name beginning with "_install" and ending with ".lasso" or ".inc". For example, an install file that performs a specific task, such as creating a database required by the app, could be named "_install.create_dbs.lasso".

_init File

Another special file is the "_init" file. While the "_install" files will only ever execute once at installation, a file such as "_init.lasso" will be executed every time the instance starts. Initialization files are used to define all of the types, traits,

and methods used within the application; along with any code set by **define_atBegin**. (Defining methods, types, etc. is best done at startup on a production system, since redefining a method can have an impact on system resources.)

Matching URLs to Code Files

LassoApps match the code files they process based on the type of content requested as represented by the extension in the URL path. The default type is HTML if no extension is used or if the ".lasso" extension is used. That means the following example URLs will all match the same code:

```
http://example.com/lasso9/AddressBook/people
http://example.com/lasso9/AddressBook/people.html
http://example.com/lasso9/AddressBook/people.lasso
```

Lasso matches those URLs to a file named "people.lasso" in the root of the AddressBook directory. It processes that file and then it checks for any secondary files to process. These secondary files are based on the content extension, so in the case of the above URLs, it will execute a file named "people[html].lasso". The primary file can return a value that can be used by the secondary file. This allows you to easily separate code for logic from code for display. (Note that if you use the URL ending in "people.lasso", Lasso won't look for a secondary file to run based on content; only that code file will be run.)

For example, your "people.lasso" file could contain code to create an array of people objects and then return that array at the end:

```
local(found_people) = array

// ... populate the array ...

return #found_people
```

Your "people[html].lasso" file might look something like this:

```
<?lasso
    // Store the value returned from people.lasso
    local(contacts) = #1
?>
<!DOCTYPE html>
<html>
<head>
    <title>Your Contacts</title>
</head>
<body>
    <table>
    <thead>
        <tr><th>First Name</th><th>Middle Name</th><th>Last Name</th></tr>
    </thead>
    <tbody>
    [with person in #contacts do {^]
        <tr>
            <td>[#person->firstName]</td>
            <td>[#person->middleName]</td>
            <td>[#person->lastName]</td>
        </tr>
    [^}]
    </tbody>
    </table>
</body>
</html>
```

This separation of logic and presentation allows for some rather powerful features. For example, let's say we wanted our application to return a JSON representation of the array of people when accessed via the URL *http://example.com/lasso9/AddressBook/people.json*. We already have the logic that finds the people and creates the array, so all that's required is add a file named "people[xhr].lasso" to create and display the array of maps:

```lasso
<?lasso
   local(people) = #1
   json_serialize(
      with person in #people
      select map(
         'firstName'  = #person->firstName,
         'middleName' = #person->middleName,
         'lastName'   = #person->lastName
      )
   )
?>
```

For more information on creating and compiling LassoApps, see the **LassoApps** chapter.

1.4 Next Steps

This has been just a taste of Lasso programming. Hopefully this chapter has whet your appetite, and you can now begin to use it for your own projects, which is always the best way to learn any language.

As you start to use Lasso, we recommend reading through all of Part II, **Language Elements**, as it will go into detail about the grammar, syntax, and features only outlined in this chapter. Also, take a look at the section and chapter titles in the rest of this book to familiarize yourself with its contents. When you find yourself needing to know more about those features, capabilities, or types, you'll then know where to find the chapter for it.

Lasso Installation

2.1 Lasso Platform Overview

Lasso Server is a powerful and comprehensive tool for building and hosting data-driven web applications. This chapter introduces important concepts and information you should know before starting to use the Lasso Platform or Lasso Server.

Lasso Server (and *Lasso Developer*) is server-side software that runs along with any FastCGI-capable web server. It adds a suite of dynamic functionality and database connectivity to your website. This functionality empowers you to build and serve just about any dynamic web application that you need with maximum productivity and ease.

Lasso Server works by receiving requests from the web server and routing those requests to specific server-side resources. Generally, these local resources are files written in the Lasso language. Lasso can be easily embedded in templates with HTML, XML, or other data types. Lasso Server manages processing and executing these files and responding to the client with the resulting data. The details of programming in the Lasso language are covered in this book starting with Part II, *Language Elements*.

2.1.1 Lasso Runtime

The Lasso runtime is a language platform designed from the ground up with the single purpose of hosting online applications. It was engineered with a focus on scalability and performance. Lasso's threading model is integrated with its transparent event-driven I/O subsystem to make efficient use of hardware with multiple CPUs and to ensure that network requests are served promptly under heavy load.

The Lasso runtime system manages the dynamic compilation of Lasso language code down to native machine code with integrated caching schemes and automatic optimization of the most frequently used methods and object types. It also supports a flexible array of options for ahead-of-time compilation of code into dynamically loaded libraries as well as stand-alone executables for OS X and Linux platforms.

Although the Lasso runtime can operate in the role of a quick command-line utility, it is designed with features to support the needs of an "always-on" application process. Internally, such an application consists of a set of subprocesses, or threads, each waiting to receive messages and sleeping until they arrive. Each thread maintains its own set of variables, and a thread cannot directly access the variables of another thread.

2.1.2 Lasso Language

This version of the Lasso language builds upon many years of success with providing a versatile and full-featured programming language. Lasso is a dynamically typed, object-oriented language with close ties to its database abstraction layer. The language integrates closely to the Lasso runtime to give access to the lower level operating system in a uniform and performance-minded manner. It improves upon this by already including many of the most useful third-party libraries, data sources, technologies, and protocols.

Lasso's type system combines the dynamic nature of scripting languages with some of the safety and features normally only available in statically typed languages. Types can be extended through a "traits" system, promoting code re-use, and methods are open to multiple dispatch which uses type annotations to provide pattern matching over method calls.

New types and methods can be added at runtime, which means that a running system is not required to be brought down for most changes. This is helpful not only during development but also while a system is in service and urgent, well-tested patches need to be applied.

Object types and methods can be compiled ahead of time using the Lasso compiler (`lassoc`). The result of this compilation is a dynamic library that can be automatically loaded on demand when the type or method is first required. This helps keep unused data out of memory, improves startup time, and lets an application be built and deployed in a modular manner.

2.1.3 Lasso Server

Lasso Server offers a suite of tools and methodologies for developing, serving, and administering data-driven web sites written in the Lasso language. Lasso Server operates with any web server supporting FastCGI, such as Apache, IIS, lighttpd, and nginx. A full complement of methods and objects are included for accessing web server parameters and variables as well as for encoding and decoding data and for responding to requests.

Lasso Server provides an object model for programmatically building response documents in addition to a simple template mode for creating or customizing HTML, XML, PDF, or any other sort of data on the fly. This system is designed to make it easy to separate logic from presentation.

Lasso Server provides easy to use, yet powerful, control over content type and character encoding. Combined with the Lasso language's highly integrated support for Unicode strings, Lasso Server can readily serve content for any language using just a single string object and API.

Also provided are built-in support for logging, bulk email sending, users and groups security, sessions, and more; including integration with many third-party libraries such as curl, OpenSSL, and SQLite. Lasso Server brings a rich set of tools together into one package.

The Lasso Instance Manager and Lasso Server Admin applications are included with Lasso Server. These applications provide administrative access to a running system via a web browser. Lasso Instance Manager handles creation, licensing, and status of individual Lasso Server instances, while Lasso Server Admin gives access to database configuration, users and groups, sessions, email queues, error logs, and more. Lasso Instance Manager and Lasso Server Admin provide an accessible access point for the server administrator to monitor and configure the operations of the server.

2.1.4 Lasso Developer

Lasso Developer is a free of charge, single-user edition of Lasso Server that can be used by a single developer to create and test interactive web sites on their own machine. Lasso Developer has a client IP addresses limitation and per-minute transaction limit. Lasso Developer is designed for authoring and demonstrating web sites and is the perfect way to get started with Lasso Server.

Any installation of Lasso Server will default to Lasso Developer functionality when run without a valid serial number.

2.2 OS X Installation

These instructions are for installing Lasso Server on OS X. An Intel-based Mac running OS X 10.5 or later is required.

2.2.1 OS X Prerequisites

- Lasso's *PDF functions* require a Java runtime, which is not included with OS X 10.7 and later. The latest version of Apple's "Java for OS X" can be downloaded from the Apple support site[9].

---

[9] https://support.apple.com/kb/DL1572

- Lasso's **image functions** require X11, which is not included with OS X 10.8 and later. For those systems, install XQuartz[10] instead.

2.2.2 Installation

1. Download and expand the Lasso Server for OS X[11] installer from the LassoSoft website.

2. Run the installer to perform a standard installation, which will install Lasso Server and start or restart the system's Apache. By default the following files and folders will be installed:

   Apache config

   - `/etc/apache2/other/mod_lasso9.10.5.conf` (OS X 10.5)
   - `/etc/apache2/sites/mod_lasso9.10.5.conf` (OS X Server 10.5)
   - `/etc/apache2/other/mod_lasso9.conf` (OS X 10.6 and later)
   - `/etc/apache2/sites/mod_lasso9.conf` (OS X Server 10.6 and Server 10.7)
   - `/Library/Server/Web/Config/apache2/sites/mod_lasso9.conf` (OS X Server 10.8 and later)

   Apache plugin

   - `/usr/libexec/apache2/mod_lasso9.10.5.so` (OS X 10.5)
   - `/usr/libexec/apache2/mod_lasso9.so` (OS X 10.6 and later)

   shared library

   - `/Library/Frameworks/Lasso9.framework`

   launchd item

   - `/Library/LaunchDaemons/com.lassosoft.lassoinstancemanager.plist`

   binaries

   - `/usr/bin/lasso9`
   - `/usr/bin/lassoc`
   - `/usr/sbin/lassoim`
   - `/usr/sbin/lassoserver`

   user data

   - `/var/lasso`

3. When the installer has finished, click on the link on the web page that appears in order to load the initialization form (found on your own machine at *http://localhost/lasso9/instancemanager*) and complete your Lasso installation.

From here on, you can read up on using the **Lasso Instance Manager** and **Instance Administration and Configuration** interfaces.

Note: On OS X Server, verify that the Web or Websites service is running in Server Preferences or Server.app.

Important: If you upgrade your OS X installation or install OS X Server after installing Lasso Server, you will need to either manually move the Apache conf file and plugin for Lasso to continue running, or simply reinstall Lasso to place the files in the correct locations.

---

[10] https://www.xquartz.org
[11] http://www.lassosoft.com/Lasso-9-Server-Download#Mac

2.3 CentOS 5/6 Installation

These instructions are for installing Lasso Server on 32-bit/64-bit CentOS 5 or 64-bit CentOS 6.

Note: This version of Lasso requires that SELinux be disabled or set to permissive mode.

2.3.1 Installation with yum

To install Lasso Server via **yum**, the LassoSoft yum repository must be configured on the server.

CentOS 5 32-bit/64-bit
Import the LassoSoft public key:

```
$> rpm --import http://centosyum.lassosoft.com/RPM-GPG-KEY-lassosoft
```

Add the LassoSoft repository to **/etc/yum.repos.d**:

```
$> rpm -ivh http://centosyum.lassosoft.com/Lasso-CentOS-repo-1.0-1.el5.noarch.rpm
```

To install Lasso Server on a **32-bit CentOS 5 system**, as root run:

```
$> yum install Lasso-Instance-Manager-9.2.7-7.el5
```

To install Lasso Server on a **64-bit CentOS 5 system**, as root run:

```
$> yum install Lasso-Instance-Manager-9.2.7-7.el5.x86_64
```

CentOS 6 64-bit
Import the LassoSoft public key:

```
$> rpm --import http://centos6yum.lassosoft.com/RPM-GPG-KEY-lassosoft
```

Add the LassoSoft repository to **/etc/yum.repos.d**:

```
$> rpm -ivh http://centos6yum.lassosoft.com/Lasso-CentOS-repo-1.0-1.el6.noarch.rpm
```

To install Lasso Server on a **64-bit CentOS 6 system**, as root run:

```
$> yum install Lasso-Instance-Manager-9.2.7-7.el6
```

When done, open *http://your-server-domain.name/lasso9/instancemanager* to load the initialization form and complete your Lasso installation. From here on, you can read up on using the *Lasso Instance Manager* and *Instance Administration and Configuration* interfaces.

2.3.2 RPM Installation

To install Lasso Server directly from an RPM, download the latest release for CentOS 5[12] or CentOS 6[13], then as root run:

```
$> yum --nogpgcheck localinstall Lasso-Instance-Manager-9.2*.rpm
```

---

[12] http://centosyum.lassosoft.com/Lasso_Server_9.2/
[13] http://centos6yum.lassosoft.com/Lasso_Server_9.2/

2.4 Ubuntu Installation

These instructions are for installing Lasso Server on 32-bit/64-bit Ubuntu 12.

2.4.1 Installation with apt

To install Lasso Server via **apt**, the LassoSoft repository must be configured on the server. Add the repository by appending this to **/etc/apt/sources.list**:

```
deb http://debianrepo.lassosoft.com/ legacy main
```

Import the LassoSoft public key:

```
$> curl http://debianrepo.lassosoft.com/lassosoft-public.gpg.key | sudo apt-key add -
```

Then run the following to install Lasso Server:

```
$> sudo apt-get update
$> sudo apt-get install lasso-instance-manager
```

Lasso's Java support (which includes methods for PDF manipulation) and ImageMagick support are provided as separate packages. If you need the functionality these packages provide, they can be installed with the following commands:

```
$> sudo apt-get install lasso-java-api
$> sudo apt-get install lasso-imagemagick
```

When done, open *http://your-server-domain.name/lasso9/instancemanager* to load the initialization form and complete your Lasso installation. From here on, you can read up on using the *Lasso Instance Manager* and *Instance Administration and Configuration* interfaces.

2.4.2 Package Installation

To install Lasso directly from the Debian packages, download the latest releases from the repository archive[14], then run these commands (ignore errors when running **dpkg**):

```
$> sudo apt-get update
$> sudo apt-get install apache2
$> sudo dpkg -i lasso-instance-manager_9.2*.deb lasso-imagemagick_9.2*.deb lasso-java-api_9.2*.deb
$> sudo apt-get install -f
```

2.5 Windows Installation

These instructions are for installing Lasso Server on 64-bit Windows Server 2012, Windows Server 2008 R2, Windows 8, and Windows 7. Supported web servers are IIS 7, IIS 8, and Apache 2.2.

2.5.1 Windows Prerequisites

Lasso Server requires the following Microsoft updates:

---

[14] http://debianrepo.lassosoft.com/9.2/

- Microsoft Visual C++ 2012 Redistributable[15] (auto-installed if required)

- Servers running IIS 7 or 8 need ISAPI enabled:

    – For Windows Server, use the Roles Wizard to add "ISAPI Extensions" and "ISAPI Filters" under *Web Server → Application Development*.

    – For other Windows versions, open *Control Panel → Programs and Features* and click *Turn Windows features on or off*, then under *Internet Information Services → World Wide Web Services → Application Development Features*, enable "ISAPI Extensions" and "ISAPI Filters".

These installs are optional, but recommended:

- Lasso's **PDF functions** require a Java runtime. The latest version of Oracle's Java Runtime Environment can be downloaded from the Java support site[16].

- Lasso's **image functions** require ImageMagick. Download and install "ImageMagick-6.7.7-7-Q16-windows-x64-dll.exe" or later from an ImageMagick installers archive[17].

- Running Lasso under Apache requires a 64-bit installation of Apache 2.2. Only 32-bit installers of Apache 2.2 are officially available from http://httpd.apache.org/, but unofficial 64-bit installers[18] can be found elsewhere online.

If either is installed after Lasso, restart Lasso Instance Manager by opening the built-in Services application, selecting the "Lasso Instance Manager" service, and clicking the "Restart Service" icon.

2.5.2 Installation

1. Download the Lasso Server for Windows[19] installer for your preferred web server from the LassoSoft website and run the installer package. (Their contents are identical, but only the IIS installer configures IIS for you automatically.) By default the following files and folders will be installed:

    Lasso folder
    ```
    C:\Program Files\LassoSoft\Lasso Instance Manager\
    ```

    IIS plugin
    ```
    C:\Windows\System32\isapi_lasso9.dll
    ```

2. When the installer has finished and your web server has been configured, open *http://your-server-domain.name/lasso9/instancemanager* to load the initialization form and complete your Lasso installation. From here on, you can read up on using the **Lasso Instance Manager** and **Instance Administration and Configuration** interfaces.

2.5.3 Configuring IIS 7 or 8

These steps will replicate the configuration commands run by the IIS installer.

To add Lasso Connector for IIS to the list of allowed ISAPI extensions:

- Open IIS Manager[20] and select your computer name from the nodes on the left.

- In the main panel, double-click on *ISAPI and CGI Restrictions*.

- Click *Add...* to add a new entry:

    – ISAPI or CGI path: `C:\Windows\System32\isapi_lasso9.dll`

---

[15] http://www.microsoft.com/en-us/download/details.aspx?id=30679
[16] http://www.java.com/
[17] http://ftp.icm.edu.pl/packages/ImageMagick/binaries/
[18] http://www.anindya.com/apache-http-server-2-4-4-and-2-2-24-x86-32-bit-and-x64-64-bit-windows-installers/
[19] http://www.lassosoft.com/Lasso-9-Server-Download#Win
[20] https://technet.microsoft.com/en-us/library/cc770472(v=ws.10).aspx

- Description: `Lasso9`
- Check *Allow extension path to execute*
- Back in the main panel, double-click on *ISAPI Filters*.
- Click *Add…* to add a new entry:
  - Filter name: `isapi_lasso9.dll`
  - Executable: `C:\Windows\System32\isapi_lasso9.dll`

To pass requests for "*.lasso" files to Lasso Server:

- Open IIS Manager[21] and select your computer name from the nodes on the left, or a site within it.
- In the main panel, double-click on *Handler Mappings*.
- Click *Add Script Map…* to add a new script map:
  - Request path: `*.lasso`
  - Executable: `C:\Windows\System32\isapi_lasso9.dll`
  - Name: `Lasso9Handler`
  - Request Restrictions…: under *Mapping*, uncheck "Invoke handler only if request is mapped" (leave other settings at "All verbs" and "Script")

To configure access to Lasso Instance Manager and Lasso Server Admin:

- Open IIS Manager[22] and expand your computer name from the nodes on the left.
- Right-click a web site under your computer name, e.g. "Default Web Site".
- Select *Add Application…* to add a new application:
  - Alias: `lasso9`
  - Application pool: select an appropriate pool (generally DefaultAppPool is acceptable)
  - Physical path: `C:\Program Files\LassoSoft\Lasso Instance Manager\www\`
- Select the newly created application from the nodes on the left and double-click on *Handler Mappings*.
- Click *Add Script Map…* to add a new script map:
  - Request path: `*`
  - Executable: `C:\Windows\System32\isapi_lasso9.dll`
  - Name: `LassoAdmin`
  - Request Restrictions…: under *Mapping*, uncheck "Invoke handler only if request is mapped" (leave other settings at "All verbs" and "Script")

Restart IIS when finished to apply the new configuration.

2.5.4 Configuring Apache 2.2

These steps must be run manually for Apache to serve Lasso pages.

- Open `C:\Program Files\LassoSoft\Lasso Instance Manager\home\LassoExecutables\` and copy these files:

---

[21] https://technet.microsoft.com/en-us/library/cc770472(v=ws.10).aspx
[22] https://technet.microsoft.com/en-us/library/cc770472(v=ws.10).aspx

- `mod_lasso9.dll` into the Apache `modules\` folder

- `mod_lasso9.conf` into the Apache `conf\` folder

- In the `conf\` folder, open the Apache `httpd.conf` file for editing and add the following line: `Include conf/mod_lasso9.conf`

- Restart Apache.

- In a browser, open *http://localhost/lasso9/instancemanager* to load the initialization form and complete your Lasso installation.

2.5.5 Troubleshooting

ISAPI and CGI Restrictions or ISAPI Filters options for IIS are missing.

- If you cannot find either ISAPI option, it is most likely not installed. To install the ISAPI options on IIS 7 or 8:

    Windows Server

    1. Open `Server Manager`

    2. Navigate to the list of currently installed Web Server roles

    3. Expand *Web Server → Application Development*

    4. Check "ISAPI Extensions" and "ISAPI Filters"

    5. Continue through installation wizard

    Windows 7 or 8

    1. Open "Control Panel"

    2. Open `Programs and Features`

    3. Click *Turn Windows features on or off"*

    4. Expand *Internet Information Services → World Wide Web Services → Application Development Features*

    5. Check "ISAPI Extensions" and "ISAPI Filters"

    6. Click *OK*

    7. Continue through installation wizard

IIS gives the error `Handler "Lasso9Handler" has a bad module "IsapiModule" in its module list` when loading "*.lasso" files.

- IIS's ISAPI options are not installed, or were installed after Lasso Server. Follow the steps above to ensure ISAPI is installed and manually add Lasso Connector for IIS to the list of allowed ISAPI extensions.

Lasso pages are not loading.

- The Application Pool for the site may be set to run 32-bit applications. To disable:

    1. Open `IIS Manager`

    2. Select the site's "Application Pool"

    3. Click *Advanced Settings*

4. Set "Enable 32-bit Applications" to "False"

- IIS may be missing required features. To check:

   Windows Server

   1. Open `Server Manager`

   2. Navigate to the list of currently installed Web Server roles

   3. Expand *Web Server → Common HTTP Features*

   4. Check "Default Document" and "Static Content"

   5. Continue through installation wizard

   Windows 7 or 8

   1. Open "Control Panel"

   2. Open `Programs and Features`

   3. Click *Turn Windows features on or off"*

   4. Expand *Internet Information Services → World Wide Web Services → Common HTTP Features*

   5. Check "Default Document" and "Static Content"

   6. Click *OK*

   7. Continue through installation wizard

Standard 404 error page is returned instead of Lasso's default not found page.

- IIS's handler for "*.lasso" files may have a request restriction set. To disable:

   1. Open `IIS Manager`

   2. Select your computer name from the nodes on the left or a site within it, depending where the handler was first defined

   3. In the main panel, double-click on *Handler Mappings*

   4. Edit the script map for "*.lasso" files

   5. Click *Request Restrictions...*

   6. Under *Mapping*, uncheck "Invoke handler only if request is mapped"

   7. Click *OK* twice, then *Yes* to apply the change

Standard 500 error page is returned instead of Lasso's default error page.

- IIS's "HTTP Errors" feature may be enabled. To disable:

   Windows Server

   1. Open `Server Manager`

   2. Navigate to the list of currently installed Web Server roles

   3. Expand *Web Server → Common HTTP Features*

   4. Uncheck "HTTP Errors"

   5. Continue through installation wizard

Windows 7 or 8

1. Open "Control Panel"

2. Open **Programs and Features**

3. Click *Turn Windows features on or off"*

4. Expand *Internet Information Services* → *World Wide Web Services* → *Common HTTP Features*

5. Uncheck "HTTP Errors"

6. Click *OK*

7. Continue through installation wizard

Lasso Server Management

3.1 Lasso Instance Manager

Lasso Instance Manager is a companion tool integrated into Lasso Server that permits the management of one or more Lasso Server *instances*, which are isolated **lassoserver** processes each running within their own environment. This tool can install and uninstall new Lasso Server instances and allows those instances to be configured in one convenient location.

Lasso Instance Manager stands as an intermediary between the individual Lasso Server instances and the web server. Each Lasso Server instance is configured with a host name pattern. By default, Lasso Instance Manager catches all requests with the file extension ".lasso" and all requests where the URI path begins with "/lasso9/", and it uses the instance's host name patterns to determine which running instance should serve each request.

3.1.1 Initialization

After the initial installation, you will be presented with a simple install acknowledgment and a process to initialize and create the first instance. Under OS X this will be a web page; under Linux you will see output in the terminal giving the URL *http://your-server-domain.name/lasso9/instancemanager*.

When Lasso Instance Manager is run for the first time, it will look for an existing Lasso Server installation that was installed through the Lasso Server 9.0 installer and will import that instance. No files will be removed. The Admin LassoApp will be updated and any existing Lasso Server 9.0 Apache 2 configuration file will be disabled.

If no existing Lasso Server 9.0 instance is located, a new Lasso Server instance will be created named "default". This default instance will be configured with the same administrative username and password that was selected when first initializing Lasso Instance Manager, and is set up to respond to all host names.

3.1.2 User Interface

The main interface for Lasso Instance Manager consists of a list of all known Lasso Server instances on that machine. Below this list is a button for creating new instances.
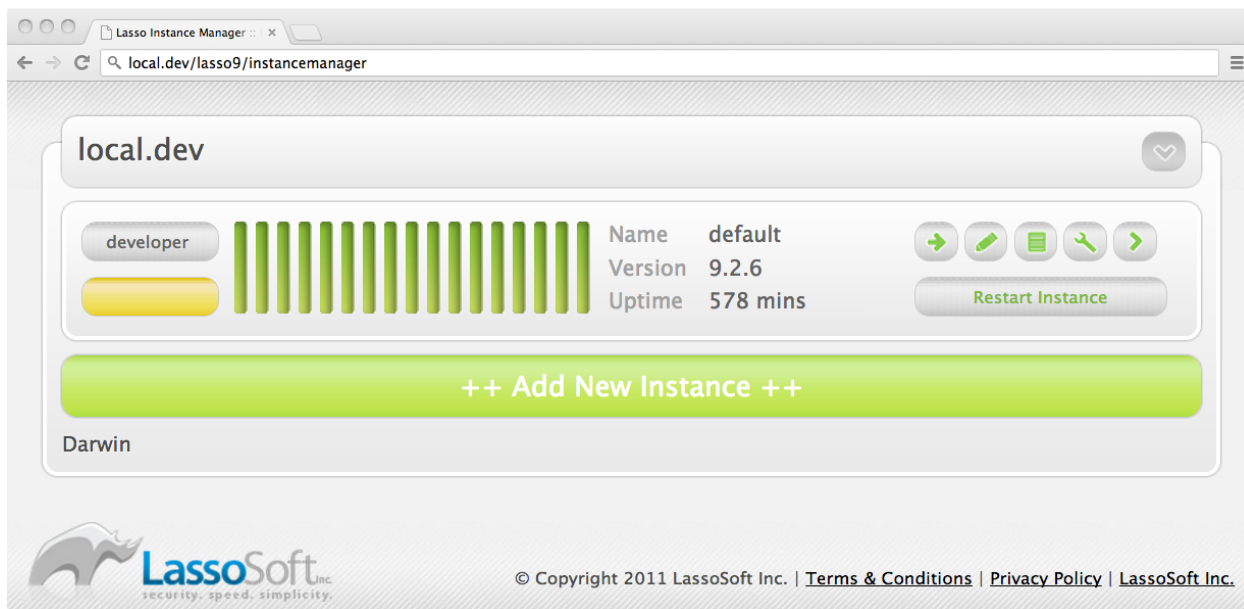
Fig. 3.1: Lasso Instance Manager – Main Screen

Each instance in the list shows:

- Current licensing mode, based on the instance serial number, if any.
- Current run status: red is not running, yellow is running but with a developer license, green is running and fully licensed.
- Instance name
- Instance version (as of 9.2.5)
- Instance uptime

Additionally, a row of buttons on the right permit access to the following:

- Instance configuration: instance name, description, admin URL, home path, OS user, host pattern, *Disable Instance* button, *Delete Instance* button
- Instance notes
- Instance output log
- Instance environment variables
- Instance admin link
- *Restart Instance* button

Note that Lasso Instance Manager periodically updates the interface automatically to show each instance's status.

Add a New Instance

Click the *++ Add New Instance ++* button at the bottom of the interface. This will present a form allowing you to customize the Lasso instance to be created. When done, click the *+ Add* button at the bottom to create your new instance.

Fig. 3.2: Lasso Instance Manager – Adding an Instance

Instance Creation Fields

Instance Name
This is the name of the new instance. It serves to uniquely identify the instance among the others.

Host Patterns
A *host pattern* is a string of characters that are matched against the incoming HOST field of the HTTP request. An instance may have several of these patterns. These patterns serve to direct individual requests to specific Lasso instances. A host pattern resembles a domain name, and can contain the wildcard character "%". The pattern `%foo.com` would match "www.foo.com" and "foo.com". The pattern `localhost` would match only "localhost". If no instance is configured to respond to a given host, an error will be returned to the client.

Note that the host pattern uses globbing rather than regular expressions for pattern matching.

Instance URL
This URL is used only within the Instance Manager. This URL provides a convenient way to link from the Instance Manager to a page served by that Lasso instance.

Description
An optional description of this Lasso instance.

Serial Number

If you already have a license serial number for the new instance, enter it here.

OS User

The new instance will consist of a process that runs as this specific operating system user. Additionally, the Lasso home directory will have its permissions adjusted to restrict access to users other than this one. The default user name is "_lasso".

OS Group

This group name will be used for adjusting the Lasso home directory permissions. If left blank, the OS user's default group will be used. The default group is "_lasso", which is used with the OS user name of "_lasso".

Home Parent Directory

Every Lasso Instance has a home directory. This directory is automatically created *within the path indicated by this form field* when the instance is created. This field *only* indicates the path up to the directory *in which* the new instance's home directory will be created. The name of the new instance's home directory will be the name of the instance (specified in the first form field). Note that illegal characters and spaces will be stripped from the new directory name.

- Unless manually edited, new instance home directories are created in a specific location within the directory housing the Lasso Instance Manager. This location will differ based on the platform on which Instance Manager is running. This location is shown at the time a new instance is being created. If this path is edited, the new value will become the default for subsequently created instances (though the path can still be edited).

- By default, ownership of the new instance's home directory will be set to what is specified in the "OS User" and "OS Group" fields.

The most important bits of information to fill in are the instance name and the host pattern. The rest can be left as they are unless you have a specific need to tailor this instance.

By default, all new instances run as the "_lasso" operating system user. For enhanced security between different Lasso instances, use a different OS user and group for each. As an OS user is permitted to read files owned by that user, Lasso instances with the same OS user are able to share files among themselves. In many cases this is not a problem, or may not even be desired, but if a single computer is hosting many instances for many different users, differing usernames may be required.

Change Instance Configuration

After an instance has been created, the instance's host patterns and URL can still be modified. Click the instance's "Configuration" button (the first of the five in the top right) to reveal the configuration for the instance. A button is shown next to the editable items. Click the button and follow the directions to edit these items.

Disable an Instance

If an instance needs to be temporarily disabled, first click the instance's "Configuration" button to reveal the configuration for the instance. Then, click the *Disable Instance* button. This will terminate the instance's process. The instance will no longer be automatically started and can no longer serve requests. To re-enable the instance, click the *Start Instance* button near the top right of the instance view.

Delete an Instance

When an instance is no longer required, it can be removed. First, click the instance's "Configuration" button to reveal the configuration for the instance. This view contains a *Delete Instance* button and a checkbox for indicating that the instance's home directory should be deleted as well. If this checkbox is not checked when the *Delete Instance* button is clicked, the instance's home directory will be left in place. Deleting an instance will terminate that instance's process, remove the instance from the list, and optionally delete the instance's home directory.

Add Instance Notes

Instance notes are for your own reference. They permit reminders or important details to be associated with an instance. To add an instance note, click the instance's "Notes" button to reveal the instance notes view. Type your note in the provided text area and then click the *+ Add Note* button.

View Instance Logs

Lasso Instance Manager captures the last number of lines of console output generated by a Lasso instance. Click the *Logs* button to show the log for an instance. While the view is shown, the log data will automatically refresh. The data can be manually refreshed by clicking the *Refresh Log File View* button. Each log can be cleared or downloaded by clicking the appropriate button in this view.
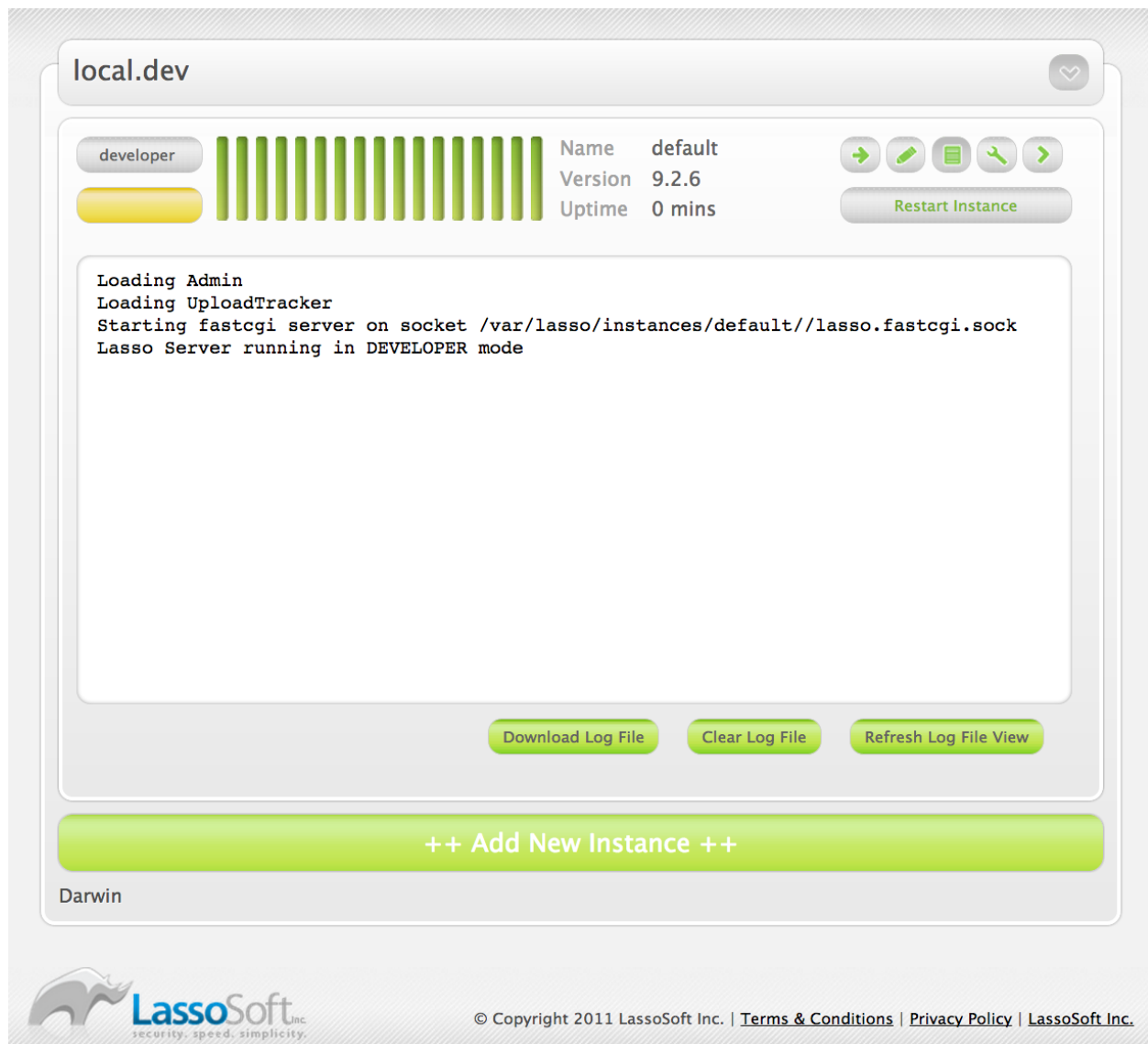
Fig. 3.3: Lasso Instance Manager – Viewing Instance Logs

Modify Instance Environment Variables

Environment variables control how an instance runs or how the software that an instance is using (e.g. ImageMagick or Java) operates. Click the instance's "Variables" button to reveal the environment variables view. All current variables are shown in this view. Existing variables can be removed, and new variables can be added. Any variable modifications will not take effect until the instance is restarted.

New instances are automatically configured with the **LASSO9_HOME** and **LASSO9_MASTER_HOME** variables. It is recommended that these not be modified or removed unless a highly customized instance is required.
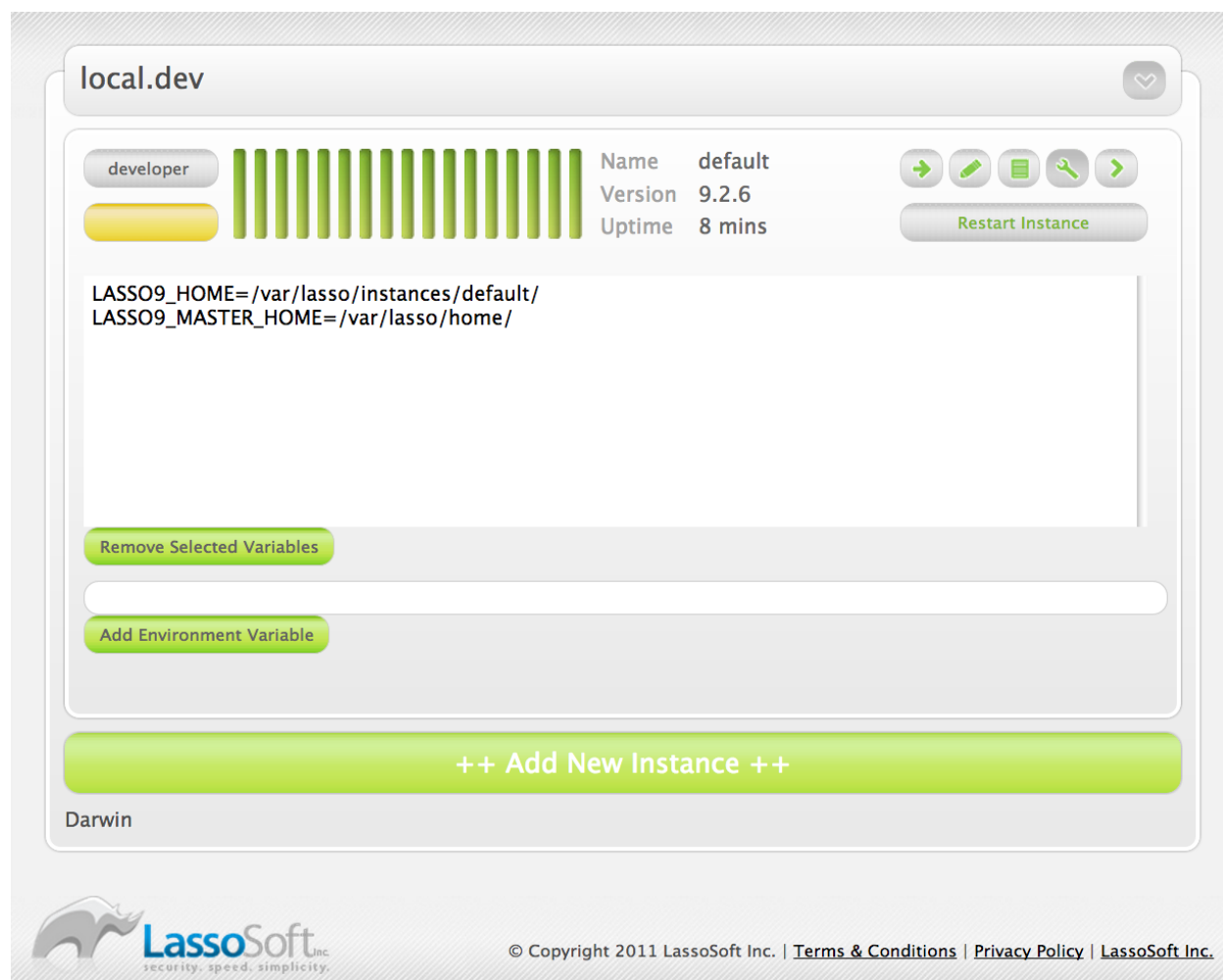
Fig. 3.4: Lasso Instance Manager – Adding Instance Environment Variables

Restart an Instance

Clicking the *Restart Instance* button will open an alert, confirming your intention to restart the instance. Pressing *Cancel* will clear the alert window and nothing further will happen. Pressing *OK* will terminate the instance's process and then restart that process. The instance's running light will switch to green or yellow once the instance is fully running again.

If the instance isn't currently running, the button will say *Start Instance*. Pressing the button will not result in a prompt, and will instead immediately start the instance.

Click the button in the instance's main view which shows the instance's licensing status (upper left corner button). A dialog will appear, permitting a new serial number to be entered. Setting an instance's serial number will restart that instance.

3.1.3 Instance Home Directory Contents

A Lasso instance's home directory can contain several folders and files that can be used to tailor the instance. Specifically, these are the "LassoApps", "LassoLibraries", "LassoModules", and "LassoStartup" directories. However, by default, an instance will also look for the first three directories in the Instance Manager's home directory. Files can be placed in the appropriate location inside of the instance's home directory in order to override the files provided by the Instance Manager.

LassoApps
> The *LassoApps* directory contains applications that are loaded when an instance starts up. At startup, the instance finds all the applications in the Instance Manager's "LassoApps" directory and compares it with the applications in its own "LassoApps" directory. Any applications in the Instance Manager's "LassoApps" directory with the same name as those in the instance's home directory are skipped. This allows an instance to install its own version of a Lasso application with the same name without ever loading the Instance Manager's version.

LassoLibraries
> The *LassoLibraries* directory contains all available on-demand libraries. These libraries are loaded as required as the instance runs. Whenever an attempt is made to use a non-existent method or type, the "LassoLibraries" directory is searched for a suitable implementation. An instance will first look in its own home directory for such a library. If not found, the Instance Manager's home directory is searched. This permits an instance to override a library that would have been loaded from the Instance Manager's home directory with its own version or to have its own instance-specific library.

LassoModules
> The *LassoModules* directory contains all Lasso C API (LCAPI) modules. These are all loaded when an instance is first started. The instance will first load all modules located in the Instance Manager's home, and then all modules located in the instance's home. This permits an instance to replace an LCAPI module with its own version, if required, or to have an instance-specific LCAPI module.

LassoStartup
> The *LassoStartup* directory contains plain-text Lasso files which are read when the instance starts. Any uncompiled custom types or methods can be placed in files ending in either ".lasso" or ".inc" and will be available across the instance.

> *Note:* Lasso only searches for "LassoStartup" in each instance's home directory, and not in the Instance Manager's home directory.

3.1.4 Starting and Stopping Lasso Instance Manager

Stopping the Lasso Instance Manager process differs on each platform.

> OS X
> > The OS X installer creates a launchd service that manages the Instance Manager process. To stop this service, execute the following command from the terminal:
> >
> > ```
> > $> sudo launchctl unload /Library/LaunchDaemons/com.lassosoft.lassoinstancemanager.plist
> > ```
>
> Linux
> > The CentOS and Ubuntu installers create a service "lassoimd" for the Instance Manager executable, which loads at startup. To stop this service, execute the following command from the terminal:

```
$> sudo service lassoimd stop
```

Windows

The Windows installer creates a service that can be stopped using Windows' built-in Services application by selecting the "Lasso Instance Manager" service and clicking the square "Stop Service" button.

Stopping the Instance Manager will also stop all Lasso instances. No Lasso instance will be able to serve any requests while the Instance Manager is not running.

When installed, Lasso Instance Manager is configured to automatically start when the computer boots up. If the Instance Manager has been manually stopped, it can be manually started again.

OS X

The OS X installer creates a launchd service that manages the Instance Manager process. To start this service, execute the following command from the terminal:

```
$> sudo launchctl load /Library/LaunchDaemons/com.lassosoft.lassoinstancemanager.plist
```

You can then verify that Lasso Instance Manager is running:

```
$> ps -ax | grep lassoim
62 ?? 7:10.95 /usr/sbin/lassoim
```

Linux

The CentOS and Ubuntu installers create a service "lassoimd" for the Instance Manager executable, which loads at startup. To start this service, execute the following command from the terminal:

```
$> sudo service lassoimd start
```

You can then verify that Lasso Instance Manager is running:

```
$> sudo service lassoimd status
lassoimd (pid 4653) is running...
```

Windows

The Windows installer creates a service that can be started using Windows' built-in Services application by the "Lasso Instance Manager" service and clicking the triangle "Start Service" button. You can then verify that Lasso Instance Manager is running by checking if the "Status" column for the "Lasso Instance Manager" service says "Started".

3.1.5 Uninstallation

OS X

An uninstaller is provided in the same package as the original installer. Run this to uninstall Lasso Instance Manager. This action will remove any Lasso instance home directories that were created in the default location (`/var/lasso`). This will not remove any home directories that were created in custom locations.

Linux

Use the standard package manager (yum or apt) to uninstall Lasso Instance Manager.

Windows

Use the system's built-in uninstall utility via the `Programs and Features` control panel.

3.2 Instance Administration and Configuration

Lasso Server provides a convenient, web-based interface for configuring a Lasso instance's settings, managing and maintaining databases, and much more. This interface is referred to as the *Lasso Server Admin*.

3.2.1 Accessing Lasso Server Admin

Lasso Server can have multiple independent instances defined within **Lasso Instance Manager**. Requests are distributed to each host based on the value of each request's **Host** header.

The default instance will catch all incoming web requests. If no additional instances have been created, then Lasso Server Admin for the sole instance can be accessed using any domain name pointing at the server that's enabled for Lasso Server. (Use "localhost" if accessing Lasso Server Admin from a browser on the same machine Lasso Server is installed on. Otherwise, replace "www.example.com" with the server's actual domain name or IP address.)

```
http://localhost/lasso9/admin
http://www.example.com/lasso9/admin
```

To access Lasso Server Admin for a particular instance, it is necessary to construct a URL that meets the criteria for the instance. For example, if an instance only serves requests for the domain "secure.example.com" then the following URL would load the Lasso Server Admin for that instance:

```
http://secure.example.com/lasso9/admin
```

The web browser should prompt for the administrator username and password using a standard HTTP authentication prompt. The web browser will not prompt if Lasso Server Admin has already been accessed using the browser in the current session or if the authentication information has been stored in a keychain, passport, or browser preferences.

If an error is displayed, make sure Lasso Server and the web server are running as described in the installation instructions for your operating system elsewhere in this guide.

3.2.2 User Interface

All settings for a particular instance are configured here.

System Status

The *System Status* tab, located in the top right portion of the page, contains information regarding the Lasso Server instance.

- **Uptime** – length of time the instance has been running
- **CPU Time** – how much CPU time has been consumed by the instance
- **Threads** – number of threads in use by the instance
- **Memory** – amount of memory consumed by the instance
- **Free** – memory Lasso has been allocated but is not currently using
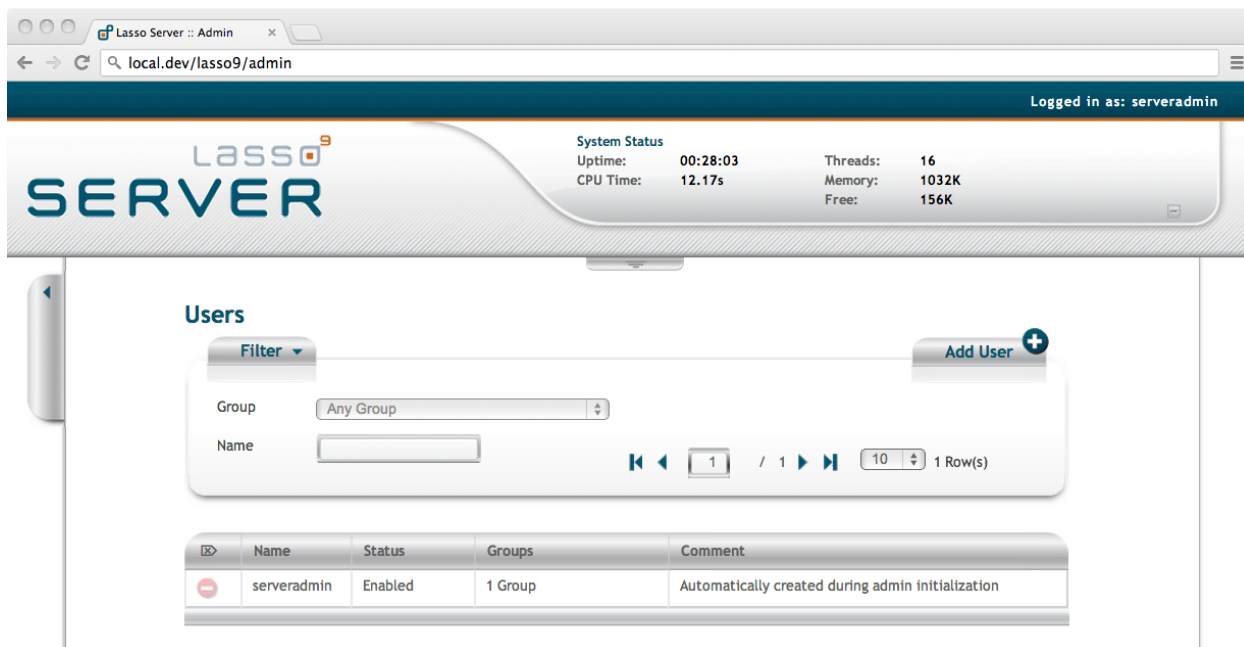
Fig. 3.5: Lasso Server Admin – Landing Page and System Status

Lasso Quick Code

Lasso Quick Code allows you to run code snippets within the Lasso Server Admin web interface. The Lasso Quick Code console can be opened by clicking the arrow tab in the top center of the page.
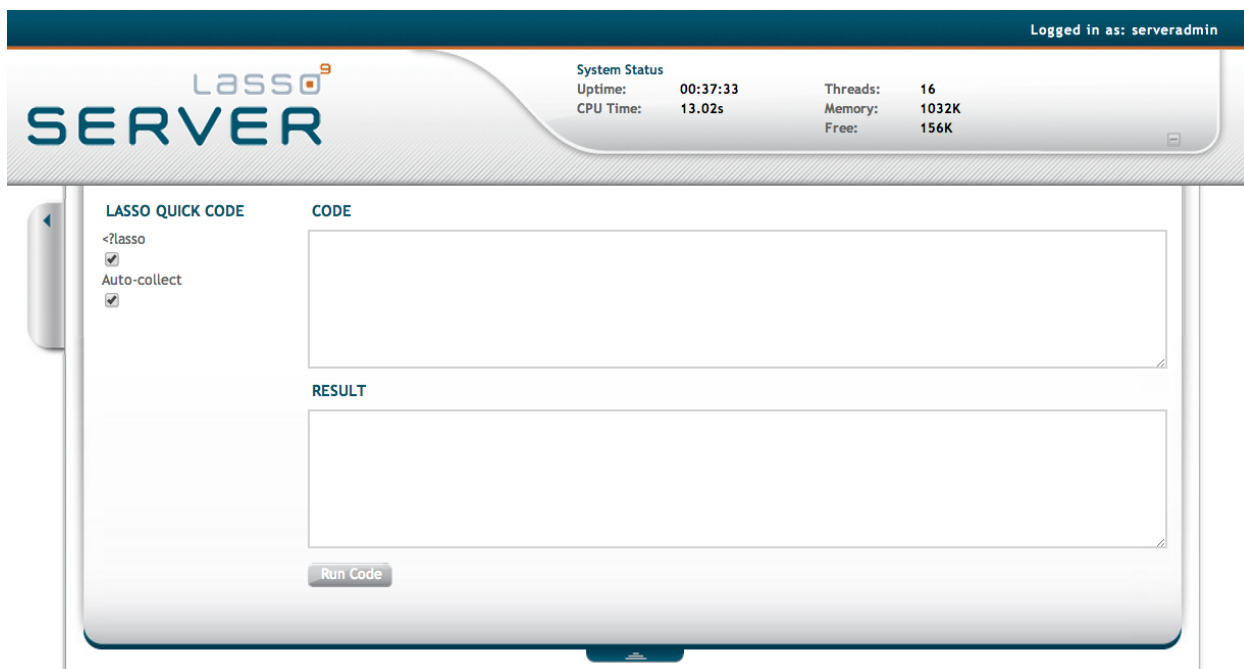
Fig. 3.6: Lasso Server Admin – Lasso Quick Code

The Lasso Quick Code console has two configurable options:

- **<?lasso** – When this option is checked, anything entered in the "Code" field is treated as having been wrapped in `<?lasso ... ?>` delimiters.

- **Auto-collect** – When this option is checked, the Lasso Quick Code console will display the results of auto-collecting the code in the "Result" field.

Main Menu

The Lasso Server Admin is primarily navigated by using the pop-up menu on the upper left portion of the page. This is the "Main Menu" which contains the following divisions and options:

Lasso

- Datasources – Configure database connections.

- License – Enter license serial number. (This can also be entered in Lasso Instance Manager.)

Auth

- Users – Configure Lasso users.

- Groups – Configure Lasso groups.

Monitors

- Log Book – View log messages and configure logging settings.

- Email Queue – View emails currently in queue.

- Sessions – View active sessions, delete expired sessions, and change session storage settings.

Utilities

- DB Browser – Manage and browse databases.
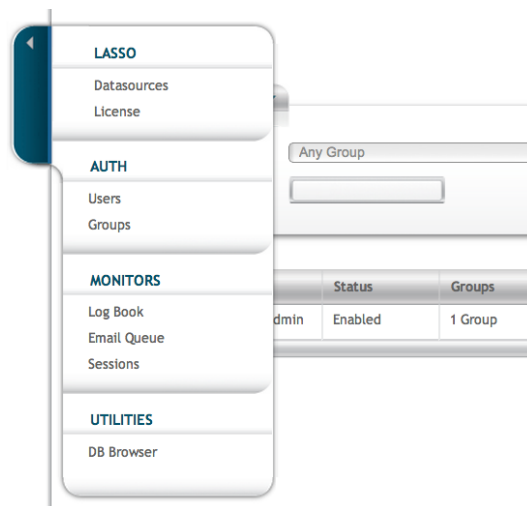
Fig. 3.7: Lasso Server Admin – Main Menu

3.2.3 Administrative Tasks

Each of the links in the Main Menu takes you to different sections in Lasso Server Admin to perform specific administrative tasks. These tasks are outlined in the sections that follow.

Configuring Datasources

Clicking on the "Datasources" link in the Main Menu will take you to a web page that lists the datasources your instance can access. Clicking on a data source will reveal a list of hosts that have already been configured as well as an *Add host* button. Clicking on that button reveals a form to enter the "Host", "Port", "Username", and "Password" information for a new host. Entering this information and clicking *Add Host* will add the new host to the list of hosts for that data source.

Clicking on a host in the list of hosts for a data source will reveal a form that allows you to edit the connection information for the host or to delete the host. Below that form is a list of databases that the credentials entered can access.

Clicking on a database reveals a form that allows the alias name for that database to be set. The alias name is what is matched when the `-database` parameter is used in an `inline` method. Below that form is a list of tables in the database that the entered credentials can access.

See the *Datasource Setup* chapter for detailed information on connecting Lasso Server to various data sources.

Entering a License Serial Number

Clicking on the "License" link in the Main Menu will take you to a web page that displays the current license of the instance. There is also an *Add Serial Number* button that allows you to add or change the serial number. This information can also be viewed and updated in Lasso Instance Manager.

Managing Lasso Users

Clicking on the "Users" link in the Main Menu will take you to an interface that allows you to add, remove, and search for Lasso users. Lasso users are stored in the instance's internal SQLite databases. Each user has a name, password, and a status (enabled or disabled). They can optionally have a comment and belong to one or more Lasso groups. These users can be used with the `auth_...` methods for HTTP authentication.

During the initial installation and setup of the instance, Lasso Server Admin creates a user in the "ADMINISTRATORS" group and authenticates anyone trying to access itself against those credentials. Be sure you don't delete this user without first creating another user in the "ADMINISTRATORS" group. In fact, you should always have at least one administrator that can log in to Lasso Server Admin.

Search for a User

You have two ways to filter the list of Lasso users to find the user(s) you are seeking. You can filter users based on their membership in a group by selecting a group in the *Group* drop-down list. Changing this selection will cause the list of users to update itself based on the criteria you have selected. You can also search for a user by name by typing part or all of the name in the "Name" text field. As you type, the list of users will filter itself based on the data you enter.

Add a User

To add a Lasso user, click the *Add User* button in the top right of the "Users" interface. A dialog will appear allowing you to enter the name, password, and a comment as well as allowing you to select the status and which groups they should be a member of. Once all the data is correctly entered, click the *Add* button to add the user.

Delete a User

To delete a user, first find the name in the listing of users. Once you have found the user, click the minus button in the column to the left of the name. You will receive a dialog box double-checking your intentions. Click the *OK* button in that dialog box to delete the user.

Managing Lasso Groups

Clicking on the "Groups" link in the Main Menu will take you to an interface that allows you to add, remove, and search for Lasso groups. Lasso groups are stored in the instance's internal SQLite databases. Each group has a name and a status (enabled or disabled). They can optionally have a comment. These groups can be used with Lasso users and with the `auth_...` methods for HTTP authentication.

During the initial installation and setup of the instance, Lasso Server Admin creates a group named "ADMINISTRATORS" and any user assigned to that group can authenticate into Lasso Server Admin. There is also a special group named "Any Group" that will appear in the "Users" interface of Lasso Server Admin. This is a way to reference every user since everyone is a member of this special group.

Search for a Group

You can filter groups based on their names by typing part or all of the name in the "Name" text field. As you type, you will notice the list updating itself based on the data you enter.

Add a Group

To add a Lasso group, click the *Add Group* button in the top right of the "Groups" interface. A dialog will appear allowing you to enter the name and a comment as well as selecting the status of the group. Once all the data is correctly entered, click the *Add* button to add the group.

Delete a Group

To delete a group, first find it in the listing of groups. Once you have found the group, click the minus button in the column to the left of its name. You will receive a dialog box double-checking your intentions. Click the *OK* button in that dialog box to delete the group.

Monitoring and Managing Lasso Logs

Clicking on the "Log Book" link in the Main Menu will take you to an interface for managing the instance's Log Book. The "Log Book" interface allows you to view and delete errors, warnings, detail messages, and deprecated functionality warnings that have been logged by Lasso Server via the `log_...` methods. In this interface you can also specify the site's logging settings.

Note: Configuring error logging in Lasso Server Admin is not the same as configuring page-level error handling, such as for syntax errors and security errors. Page-level error handling is described in the *Error Handling* chapter.

Filter Log Messages

You can filter log entries based on their message by typing part of the message in the "Message" text field. As you type, the list of log messages will filter itself based on the data you enter.

Delete Log Messages

There are two action buttons below the log entries table:

- **Delete All Messages** – delete all log entries stored in the SQLite database

- **Delete All Found** – delete all log entries that have been found based on the search term in the "Message" text field above

Log Book Settings

Click on the *Settings* button at the top right of the "Log Book" interface. A dialog will appear with a matrix of checkboxes that allows you to select where each type of log message is sent. Make your selections, then click the *Save* button to update the instance's logging settings.

Monitoring and Managing the Email Queue

Clicking on the "Email Queue" link in the Main Menu will take you to a web page that displays the instance's email queue. The email queue logs all email messages that are being sent from the instance. Messages remain in the queue while they are being sent to the SMTP mail server looked up by Lasso or specified in the `email_send` method by the developer. For more information, see the *Sending Email* chapter.

Filter Email Messages

You can filter the email messages being displayed in the queue by their status: "Any", "Queued", "Sending", or "Error". Simply choose one of those statuses from the *Queue Status* drop-down list and the queue entries will automatically update to reflect your selection.

Delete Email Messages

To remove an email message from the queue, first find it in the listing of entries. Once you have found the message, click the minus button in the column to the left of its ID. You will receive a dialog box double-checking your intentions. Click the *OK* button in that dialog box to remove the message from the queue.

Managing Lasso Sessions

Clicking on the "Sessions" link in the Main Menu will take you to an interface that allows you to browse and manage sessions in real time as well as configure the location for storing sessions.

View Sessions

Sessions can be stored in any of the available data sources for your instance of Lasso Server as well as in memory. The default is to use a SQLite database and table to store session information. You can view the session information you have stored in any of the data sources by selecting the data source from the *Driver* drop-down list and then selecting the appropriate values in the *Database* and *Sessions Table* drop-down lists if appropriate. (These last two lists will be disabled for the "SQLite" and "In Memory" drivers. Otherwise, they will show the databases/tables you have access to for the selected driver's data source.)

Delete Expired Sessions

Clicking the *Delete Expired Sessions* button beneath the *Driver* drop-down list will remove all expired session entries from the currently selected session data source's table. By default, Lasso Server periodically clears out expired sessions, so it is not usually necessary to run it manually.

Configure Session Storage Location

By default, Lasso Server is configured to store session information using the "SQLite" session driver. You can change this by following these steps:

1. Select the driver you wish to use from the *Driver* drop-down list.

2. If the driver is not "SQLite" or "In Memory", select a value from the *Database* drop-down list and the *Sessions Table* drop-down list. (You can click the *Create Sessions Table* button below the *Sessions Table* drop-down list to have Lasso Server create a table in the selected database with the correct schema for storing sessions. If you click this button, you will be given the chance to name the table whatever you desire, and then that new table will be selected in the *Sessions Table* drop-down list.)

3. Click the *Select As Default Driver* button to have the **session_start** method use your selection for storing session information.

Browsing Data Sources

Clicking on the "DB Browser" link in the Main Menu will take you to an interface that allows you to issue SQL queries to accessible SQL databases. This includes any SQLite, MySQL, or SQL-compliant ODBC database that has been set up in the "Datasources" interface of Lasso Server Admin.

Browsing data is as easy as selecting the appropriate values in the *Datasource*, *Host*, *Database*, and *Table* drop-down lists. Lasso Server Admin will automatically issue a **SELECT *** on the chosen table and display the results in the table below.

You can run your own SQL statements on the chosen host/database/table by entering them in the provided "Statement" text area and clicking the *Issue Statement* button below the text area. The results will be shown in the table below. If there are any errors in your SQL statement, an alert message will inform you of the error, and no results will be displayed.

3.3 Datasource Setup

Lasso Server communicates with *data sources*, which are any type of software mechanism for storing and retrieving data (including databases), using Lasso *data source connectors*, which are modular components that translate Lasso requests into commands specific to each data source. Connections to data sources, or *datasources*, are configured in the "Datasources" section of Lasso Server Admin. Lasso Server provides built-in connectors for all of the data sources listed below.

Most connectors can access data sources that are installed on the same machine as Lasso Server or on a remote machine. Some connectors can only access files on the local machine. Whether or not an ODBC data source can communicate with Lasso on a separate machine depends on whether or not the driver can communicate via TCP/IP.

Custom data source connectors for other data sources can also be created for use with Lasso Server using Lasso's C API (LCAPI), Java API (LJAPI) or Lasso itself using the **dsinfo** type. (Information about creating and using LCAPI third-party data source connectors can be found in the **Creating Lasso Data Sources** chapter of the LCAPI documentation.)

FileMaker Server Data Sources
 Supports FileMaker Server 9–15 and FileMaker Server 7–12 Advanced.

MySQL Data Sources
 Supports MySQL Server 3.x, 4.x, or 5.x data sources. The MySQL client libraries need to be installed when using this data source.

Oracle Data Sources
 Supports Oracle data sources. The Oracle "Instant Client" libraries must be installed in order to activate this data source.

PostgreSQL Data Sources
 Supports PostgreSQL data sources. The PostgreSQL client libraries must be installed in order to activate this data source.

ODBC Data Sources
> Support any data source with a compatible ODBC driver.

SQL Server Data Sources
> Supports Microsoft SQL Server. The SQL Server client libraries must be installed in order to activate this data source.

SQLite Data Sources
> SQLite is also the internal data source used for the storage of Lasso's preferences and security settings.

3.3.1 Lasso Data Source Connectors

Data source connectors allow database actions to be performed via Lasso code. Database actions can be used in Lasso to search for records in a database that match specific criteria; to navigate through the found set from a search; to add, update, or delete a record in a database; to fetch schema information about a database; and more. Additionally, database actions can be used to execute SQL statements in SQL-compliant databases.

Interacting with data sources via Lasso generally involves these steps:

1. Configuring the data source application or service to accept connections from Lasso. This is done in the data source itself, outside of Lasso. This chapter describes configuring each data source to accept connections from Lasso.

2. Configuring Lasso Server to communicate with a data source host. This involves adding the data source connection information in the "Datasources" section of Lasso Server Admin. This chapter details creating connections with the data sources described above.

3. Writing Lasso code to interact with the data source. This is covered in the **Database Interaction Fundamentals** chapter.

Alternatively, data sources can be connected to directly by specifying all the connection parameters within an inline. Using this method does not require setting up the data source host in Lasso Server Admin, and can be used when some security can be sacrificed for coding efficiency. In this case, the following steps need to be taken:

1. Configuring the data source application or service to accept connections from Lasso, as described above.

2. Writing Lasso code to interact with the data source and passing in the host parameters to the inline. Each of the data sources documented below will give examples of interfacing with a data source host in this manner.

3.3.2 FileMaker Server Data Sources

Lasso Server communicates with FileMaker Server 9–15 and FileMaker Server 7–12 Advanced through their built-in XML interface. Lasso cannot communicate with any other products in the FileMaker 7, 8, or 9 product line such as FileMaker Pro.

Requirements

One of the following:

- FileMaker Server 9–15 for Windows or OS X
- FileMaker Server 9–12 Advanced for Windows or OS X
- FileMaker Server Advanced 7 or 8 for Windows or OS X

Additionally, the Web Publishing Engine must be installed and each database must be configured according to the instructions in the following section.

Configuring FileMaker Server 9 or Higher

This section describes setting up FileMaker Server 9–15 or FileMaker Server 9–12 Advanced for use with Lasso. These versions will be referred to collectively as FileMaker Server.

Follow the instructions included with FileMaker Server carefully. Starting with version 9, setting up FileMaker Server is considerably easier than setting up earlier versions.

- Make sure that the "Web Serving" options are turned on and that the XML interface is enabled.

- The databases that are to be accessed by Lasso must be in the FileMaker Server Data/Databases folder and must be "Open" within FileMaker Server.

- Each database to be accessed by Lasso must have the "fmxml" keyword added to the "Extended Privileges" section of the "Accounts & Privileges" dialog box. The username and password entered into Lasso Server Admin must use a Privilege Set that has access to this extended privilege.

- FileMaker Server database security is only as secure as the Publishing Engine setup. It is possible for web browsers to communicate directly with the Publishing Engine. It is strongly recommended that the security features of FileMaker Server be used to secure web-accessible databases.

- It is strongly recommended that only a single IP address corresponding to the machine on which Lasso Server runs be permitted to access the Publishing Engine.

- For tips on optimizing performance for FileMaker databases, see the **FileMaker Data Sources** chapter.

Configuring FileMaker Server Advanced 7 or 8

This section describes setting up FileMaker Server Advanced for use with Lasso.

Follow the instructions included with FileMaker Server Advanced carefully. There are several steps in the process that are not obvious and require reading the documentation to set up properly. Configuring FileMaker Server Advanced is beyond the scope of this documentation, but some common pitfalls are listed below.

- Make sure both FileMaker Server and the FileMaker Server Advanced Publishing Engine are installed. The machine with the Publishing Engine must be running a supported web server.

- Configure FileMaker Server with a Client Services identifier and passcode. Enter this same identifier and passcode in the Web Publishing Administration Console.

- Verify XML Publishing is turned on in the Web Publishing Administration Console.

- The databases that are to be accessed by Lasso must be in the FileMaker Server Data/Databases folder and must be "Open" within FileMaker Server.

- Each database to be accessed by Lasso must have the "fmxml" keyword added to the "Extended Privileges" section of the "Accounts & Privileges" dialog box. The username and password entered into Lasso Server Admin must use a Privilege Set that has access to this extended privilege.

- FileMaker Server Advanced database security is only as secure as the Publishing Engine setup. It is possible for web browsers to communicate directly with the Publishing Engine. It is strongly recommended that the security features of FileMaker Server Advanced be used to secure web-accessible databases.

- It is strongly recommended that only a single IP address corresponding to the machine on which Lasso Server runs be permitted to access the Publishing Engine.

- For tips on optimizing performance for FileMaker databases, see the **FileMaker Data Sources** chapter.

Adding a FileMaker Server Data Source Host

For general information about navigating Lasso Server Admin and adding a host to a data source, see the section *Configuring Datasources* in the *Instance Administration and Configuration* chapter.

To add a new FileMaker Server host:

1. In the "Datasources" section of Lasso Server Admin, click the *filemakerds* item.

2. Click the *Add host* item to reveal the host connection form.

3. Enter the IP address or domain name where the FileMaker Server data sources are being hosted.

4. Enter the TCP port the FileMaker Server communicates on in the "Port" field. See the FileMaker Server documentation[23] for information on where to find or set this. It is commonly "80" for FileMaker Server, or "443" to connect over https.

5. Select "Yes" from the *Enabled* drop-down to enable the host.

6. Enter a username for the host in the "Username" field and a password for the host in the "Password" field. Lasso will connect to the data source and all databases therein using this username and password by default. If the host does not require a username or password, then leave either field blank.

7. Click the *Add host* button.

8. Once the host is added, the new host appears in the "Hosts" listing below.

Databases in newly created hosts are enabled by default. The administrator can disable databases by expanding the database listing and setting the *Enabled* drop-down to "No". With the FileMaker Server data source added here, `inline` methods can use the `-database` parameter to specify the name of the FileMaker database to perform an action on.

Specifying FileMaker Server Hosts in Inlines

Setting up a data source host in Lasso Server Admin is the best way to ensure that access to the data source is centrally controlled. However, it can sometimes be beneficial to access a data source host without a lot of configuration. This section describes how to construct an `inline` method to access a FileMaker Server data source host. See the section *Inline Connection Options* for full details about specifying hosts in inlines.

To access a FileMaker Server host directly in an `inline` method, the `-host` parameter can be used to specify all of the connection parameters. The `-host` parameter takes an array that should contain the following elements:

- `-datasource` should be specified as "filemakerds".

- `-name` should be specified as the IP address or domain name of the machine hosting FileMaker Server.

- `-port` is optional and defaults to "80" if no port is specified.

- `-username` set to the user to authenticate as.

- `-password` set to the specified user's password to authenticate the connection.

The following code shows how a connection to a FileMaker Server data source hosted on the same machine as Lasso might appear:

```
inline(
    -host=(:
        -datasource='filemakerds',
        -name='localhost',
        -port='80',
        -username='username',
        -password='secret'
    ),
```

---

[23] http://www.filemaker.com/support/product/documentation.html

```
   -findAll,
   -database='database',
   -table='table'
) => {^
   found_count
^}
```

If there are no databases or tables listed, check the following links in a web browser to verify that the Web Publishing Engine is working correctly. Replace "filemaker_host" and "database_name" with values for your particular situation.

- `http://`*`filemaker_host`*`/fmi/xml/FMPXMLRESULT.xml?-dbnames`

- `http://`*`filemaker_host`*`/fmi/xml/FMPXMLRESULT.xml?-db=`*`database_name`*`&-layoutnames`

If either URL returns an error code other than 0 or fails in any way, Lasso will be unable to submit requests to FileMaker Server. Verify that XML Publishing is enabled or consult the FileMaker Server documentation[24] on how to proceed.

3.3.3 MySQL Data Sources

Lasso Server can communicate with MySQL servers configured to accept TCP/IP client connections. For more information on MySQL, visit http://www.mysql.com/.

Requirements

- MySQL Server 3.23 or MySQL Server 4.x or MySQL Server 5.x

- The MySQL service must be running and accepting TCP/IP connections on a port with no conflicts. This is port 3306 by default.

- MySQL access privileges must be properly assigned for the machine running Lasso Server to be allowed to authenticate.

- The MySQL client libraries are included with the connector, or automatically installed by the package manager.

Configuring MySQL Server

MySQL is operated via a command-line interface application which is normally located in the "bin" directory of the MySQL installation on the server machine. For information on how to use this, consult the MySQL documentation. Various installers for MySQL may have the service automatically start when the machine boots up, so also check the installation instructions for the installation method you are using.

Security for MySQL data sources can be set at any level (server-level, database-level, table-level, etc.). For unrestricted operation, all permissions for all levels of security need to be given to the user Lasso Server uses to connect. This involves setting a new user and password for Lasso Server in MySQL with the appropriate permissions, and then entering the username and password in Lasso Server Admin. Follow the procedure below for granting all permissions to Lasso Server in MySQL using the MySQL command-line utility.

1. From the command line, log in to MySQL as your root user by entering the following command:

   ```
   $> mysql -u root -p
   ```

   You will be prompted for the MySQL root user's password specified during the MySQL installation.

2. After entering the password, you'll see the MySQL command prompt (`mysql>`). Enter the following to create a new user with a username and password and access to all levels of security in MySQL:

---

[24] http://www.filemaker.com/support/product/documentation.html

```
mysql> GRANT ALL ON *.* TO Username@Hostname IDENTIFIED BY "Password";
```

Replace "Username" and "Password" with the username and password values you wish for the user to have, and replace "Hostname" with the IP address or domain name that Lasso Server will be connecting from.

Now there is a user with all permissions that can communicate with MySQL from the machine Lasso Server is running on. This user can now be used when configuring the MySQL host in the "Datasources" section of Lasso Server Admin.

Important: You may, of course, wish to tighten security and restrict the user Lasso Server uses. It is possible to assign limited privileges to the user Lasso Server uses one at a time by replacing "ALL" in the "GRANT" statement with an individual permission (e.g. INSERT, SELECT, DELETE), and replacing "*.*" with a specific database or database.table name. This will restrict the functionality of Lasso Server to the privileges that are assigned to it. For example, giving Lasso Server only the "SELECT" privilege will allow a MySQL database to be searched using Lasso, but records cannot be added, updated, or deleted using Lasso.

Adding a MySQL Data Source Host

For general information about navigating Lasso Server Admin and adding a host to a data source, see the section **Configuring Datasources** in the **Instance Administration and Configuration** chapter.

To add a new MySQL host:

1. In the "Datasources" section of Lasso Server Admin, click the *MySQLDS* item.

2. Click the *Add host* item to reveal the host connection form.

3. Enter the IP address or domain name where the MySQL databases are being hosted in the "Host" field.

4. Enter the TCP port the MySQL service communicates on in the "Port" field. This is commonly "3306" for MySQL.

5. Select "Yes" from the *Enabled* drop-down to enable the host.

6. Enter a username for the host in the "Username" field and a password for the host in the "Password" field. Lasso will connect to the data source and all databases therein using this username and password by default.

7. Click the *Add host* button.

8. Once the host is added, the new host appears in the "Hosts" listing below.

Databases in newly created hosts are enabled by default. The administrator can disable databases by expanding the database listing and setting the *Enabled* drop-down to "No". With the MySQL data source added here, `inline` methods can use the `-database` parameter to specify the name of the MySQL database to perform an action on.

Specifying MySQL Hosts in Inlines

Setting up a data source host in Lasso Server Admin is the best way to ensure that access to the data source is centrally controlled. However, it can sometimes be beneficial to access a data source host without a lot of configuration. This section describes how to construct an `inline` method that accesses a MySQL data source host. See the section **Inline Connection Options** for full details about specifying hosts in inlines.

To access a MySQL host directly in an `inline` method, the `-host` parameter can be used to specify all of the connection parameters. The `-host` parameter takes an array that should contain the following elements:

- `-datasource` should be specified as "mysqlds".

- `-name` should be specified as the IP address or domain name of the machine hosting MySQL.

- `-port` is optional and defaults to "3306" if no port is specified.

- `-username` set to the user to authenticate as.

- `-password` set to the specified user's password to authenticate the connection.

The following code shows how a connection to a MySQL data source hosted on the same machine as Lasso might appear:

```
inline(
    -host=(:
        -datasource='mysqlds',
        -name='localhost',
        -port='3306',
        -username='username',
        -password='secret'
    ),
    -findAll,
    -database='database',
    -table='table'
) => {^
    found_count
^}
```

3.3.4 Oracle Data Sources

Lasso Server can communicate with an Oracle service running on a host machine via a TCP/IP connection. For more information on Oracle, visit https://www.oracle.com/.

Requirements

- Oracle Database 10g or later

- The Lasso Server machine must have the Oracle "Instant Client" installed if Lasso Server and Oracle are running on separate machines. The Instant Client download[25] can be found on the Oracle website. (Make sure to download just the basic Instant Client files rather than the complete Oracle 10g client or database installer.)

Installing Oracle Instant Client

Lasso must be restarted after installing the Instant Client. You can use the command line to verify that Lasso is loading the libraries correctly by checking the output of **lasso9 -s "database_initialize"**.

OS X

1. Ensure that the paths **/usr/local/oracle/** and **/usr/local/lib/** exist and are writable by the current user.

2. Download version 12.1.0.2 of the Instant Client Package - Basic for OS X.

3. Decompress the archive, which will create a folder named "instantclient_12_1".

4. Copy the entire folder into **/usr/local/oracle/**.

5. Execute the following commands to create symbolic links so that Lasso can find the Oracle libraries.

```
$> cd /usr/local/lib
$> ln -sf /usr/local/oracle/instantclient_12_1/libclntsh.dylib.12.1 libclntsh.dylib
$> ln -sf /usr/local/oracle/instantclient_12_1/libocci.dylib.12.1 libocci.dylib
```

Linux

[25] http://www.oracle.com/technetwork/database/features/instant-client/index-097480.html

1. Ensure that the path **/usr/local/oracle/** exists and is writable by the current user.

2. Download the zipfile package of version 12.1.0.2 of the Instant Client Package - Basic for Linux.

3. Decompress the archive, which will create a folder named "instantclient_12_1".

4. Copy the entire folder into **/usr/local/oracle/**.

5. Execute the following commands to create symbolic links and update ldconfig so that Lasso can find the Oracle libraries.

```
$> cd /usr/local/oracle/instantclient_12_1/
$> ln -sf libclntsh.so.12.1 libclntsh.so
$> ln -sf libocci.so.12.1 libocci.so
$> cd ..
$> ln -sf instantclient_12_1 lib
$> echo "/usr/local/oracle/lib" > /etc/ld.so.conf.d/oracle.conf
$> ldconfig
```

Windows

1. Download and install the Visual C++ 2010 x64 Redistributable[26] package from Microsoft.

2. Download version 12.1.0.2 of the Instant Client Package - Basic for Windows.

3. Extract the archive contents to an accessible location, such as **C:\instantclient_12_1**.

4. In *Control Panel → System → Advanced System Settings → Advanced → Environment Variables... → System Variables*, append the string **;C:\instantclient_12_1** to the **Path** environment variable.

Configuring Oracle

The Oracle database server must be configured with a user that has access to all of the databases, tables, and other resources that will be published through Lasso. Consult the Oracle documentation for help configuring Oracle's built-in security. The Oracle website has a "Getting Started" section which explains how to install and perform basic configuration of an Oracle database server[27].

Adding an Oracle Data Source Host

For general information about navigating Lasso Server Admin and adding a host to a data source, see the section *Configuring Datasources* in the *Instance Administration and Configuration* chapter.

To add a new Oracle host:

1. In the "Datasources" section of Lasso Server Admin, click the *Oracle* item. (Restart Lasso if necessary to make it appear.)

2. Click the *Add host* item to reveal the host connection form.

3. Enter the IP address or domain name where the Oracle data sources are being hosted, the port, and the database name using the "host:port/database" format in the "Host" field (e.g. "www.example.com:1521/MyDatabase").

4. Enter the TCP port of the Oracle service in the "Port" field. This is commonly "1521" for Oracle.

5. Select "Yes" from the *Enabled* drop-down to enable the host.

6. Enter a username for the host in the "Username" field and a password for the host in the "Password" field. Lasso will connect to the data source and all databases therein using this username and password by default.

7. Click the *Add host* button.

---

[26] https://www.microsoft.com/en-us/download/details.aspx?id=26999
[27] http://docs.oracle.com/cd/B28359_01/nav/portal_1.htm

8. Once the host is added, the new host appears in the "Hosts" listing below.

Databases in newly created hosts are enabled by default. The administrator can disable databases by expanding the database listing and setting the *Enabled* drop-down to "No". With the Oracle Server data source added here, **inline** methods can use the **-database** parameter to specify the name of the Oracle database to perform an action on.

Specifying Oracle Hosts in Inlines

Setting up a data source host in Lasso Server Admin is the best way to ensure that access to the data source is centrally controlled. However, it can sometimes be beneficial to access a data source host without a lot of configuration. This section describes how to construct an **inline** method that accesses an Oracle data source host. See the section *Inline Connection Options* for full details about specifying hosts in inlines.

To access an Oracle host directly in an **inline** method, the **-host** parameter can be used to specify all of the connection parameters. The **-host** parameter takes an array that should contain the following elements:

- **-datasource** should be specified as "oracle".
- **-name** should be specified as the IP address or domain name of the machine hosting Oracle, followed by a colon and the port to connect on, and ending with a slash and the database name (e.g. "www.example.com:1521/MyDatabase").
- **-port** is optional and defaults to "1521" if no port is specified.
- **-username** set to the user to authenticate as.
- **-password** set to the specified user's password to authenticate the connection.

The following code shows how a connection to an Oracle data source might appear:

```
inline(
    -host=(:
        -datasource='oracle',
        -name='oracle.example.com:1521/MyDatabase',
        -port='1521',
        -username='username',
        -password='secret'
    ),
    -findAll,
    -database='database',
    -table='table'
) => {^
    found_count
^}
```

3.3.5 PostgreSQL Data Sources

Lasso Server can communicate with PostgreSQL servers configured to accept TCP/IP client connections. For more information on PostgreSQL, visit https://www.postgresql.org/.

Requirements

- PostgreSQL 8.x
- The Lasso Server machine must have the PostgreSQL "libpq" library installed. This comes packaged with OS X and is installed automatically with Lasso on Linux. Windows users can find **libpq.dll** included with the PostgreSQL ODBC

driver[28] package; once installed, adding the containing directory to the `Path` environment variable will allow Lasso to load the native driver.

Configuring PostgreSQL

The PostgreSQL database server must be configured with a user that has access to all of the databases, tables, and other resources that will be published through Lasso. Consult the PostgreSQL documentation[29] for help configuring its built-in security.

Adding a PostgreSQL Data Source Host

For general information about navigating Lasso Server Admin and adding a host to a data source, see the section *Configuring Datasources* in the *Instance Administration and Configuration* chapter.

To add a new PostgreSQL server host:

1. In the "Datasources" section of Lasso Server Admin, click the *PostgreSQL* item.

2. Click the *Add host* item to reveal the host connection form.

3. Enter the IP address or domain name where the PostgreSQL data source is being hosted in the "Host" field.

4. Enter the TCP port the PostgreSQL service is listening on in the "Port" field. This is commonly "5432" for PostgreSQL.

5. Select "Yes" from the *Enabled* drop-down to enable the host.

6. Enter a username for the host in the "Username" field and a password for the host in the "Password" field. Lasso will connect to the data source and all databases therein using this username and password by default.

7. Click the *Add host* button.

8. Once the host is added, the new host appears in the "Hosts" listing below.

Databases in newly created hosts are enabled by default. The administrator can disable databases by expanding the database listing and setting the *Enabled* drop-down to "No". With the PostgreSQL data source added here, `inline` methods can use the `-database` parameter to specify the name of the PostgreSQL database to perform an action on.

Specifying PostgreSQL Hosts in Inlines

Setting up a data source host in Lasso Server Admin is the best way to ensure that access to the data source is centrally controlled. However, it can sometimes be beneficial to access a data source host without a lot of configuration. This section describes how to construct an `inline` method that accesses a PostgreSQL data source host. See the section *Inline Connection Options* for full details about specifying hosts in inlines.

To access a PostgreSQL host directly in an `inline` method, the `-host` parameter can be used to specify all of the connection parameters. The `-host` parameter takes an array that should contain the following elements:

- `-datasource` should be specified as "postgres".

- `-name` should be specified as the IP address or domain name of the machine hosting PostgreSQL.

- `-port` is optional and defaults to "5432" if no port is specified.

- `-username` set to the user to authenticate as.

- `-password` set to the specified user's password to authenticate the connection.

The following code shows how a connection to a PostgreSQL data source hosted on the same machine as Lasso might appear:

---

[28] https://www.postgresql.org/ftp/odbc/versions/
[29] https://www.postgresql.org/docs/manuals/

```
inline(
    -host=(:
        -datasource='postgres',
        -name='localhost',
        -port='5432',
        -username='username',
        -password='secret'
    ),
    -findAll,
    -database='database',
    -table='table'
) => {^
    found_count
^}
```

3.3.6 ODBC Data Sources

ODBC (Open Database Connectivity) is a generalized API for providing access to databases. Lasso Server can communicate with any ODBC-compliant data source as long as the operating system has a compatible ODBC driver properly installed. For more information on ODBC, see the **ODBC Data Sources** chapter and the documentation included with your operating system.

Requirements

- An ODBC driver that has been configured as a System DSN in the ODBC control panel.

    OS X
        ODBC data sources are configured using "ODBC Manager" which can be downloaded from http://www.odbcmanager.net/ and installed in the **/Applications/Utilities** folder. (Note that the folder **/Library/ODBC** must be created first.)

    Linux
        Consult the documentation of the ODBC drivers for information about how to set up data sources on Linux. Many ODBC drivers ship with a control panel that allows configuration of those drivers.

    Windows
        ODBC data sources are configured using "ODBC Data Source Administrator" which is normally accessed through the Windows Control Panel under **Administrative Tools**.

Configuring ODBC Connections

Consult the documentation for your data sources and ODBC drivers for details about how to secure access to the data made available through the driver. Most data sources will require the following steps:

1. Install your ODBC driver using the provided installer or instructions. This may involve creating an **odbcinst.ini** file.

2. Create a System DSN in the ODBC administration application, or edit the **odbc.ini** file. Note that the System DSN name, username, and password configured here will need to be entered in Lasso.

3. Locate and configure the **SQL.ini** file for your driver, if applicable. This file sets the options for your ODBC driver including the location of your data source. Consult your driver's documentation for details about where to find this file and what options can be configured.

4. Follow the steps below to add the data source to Lasso.

FreeTDS for SQL Server via ODBC Examples

Here's how to configure FreeTDS on OS X to allow Lasso to access an SQL Server data source via ODBC:

1. Install the Homebrew package manager using the instructions at http://brew.sh/.

2. Use Homebrew to first install a newer version of the iODBC libraries than what OS X ships with, and then the FreeTDS drivers with Unicode support.

   ```
   $> brew install libiodbc
   $> brew install freetds --with-odbc-wide
   ```

3. Use **tsql** and **iodbctestw** to verify that FreeTDS and iODBC are working.

   ```
   $> tsql -H hostname -p 1433 -U username -P password
   locale is "en_CA.UTF-8"
   locale charset is "UTF-8"
   using default charset "UTF-8"
   1> quit
   $> iodbctestw "DRIVER=/usr/local/lib/libtdsodbc.so;UID=username;PWD=password;SERVER=hostname;
    →DATABASE=databasename;PORT=1433"
   iODBC Unicode Demonstration program
   This program shows an interactive SQL processor
   Driver Manager: 03.52.1216.0712
   Driver: 01.00.0009 (libtdsodbc.so)
   SQL> quit
   ```

4. Create a folder for system-level ODBC configuration files.

   ```
   $> sudo mkdir -p /Library/ODBC
   ```

5. Use "ODBC Manager" from http://www.odbcmanager.net/ to add a new driver:

   Driver Name
       FreeTDS

   Driver File
       /usr/local/lib/libtdsodbc.so

   Setup File
       /usr/local/lib/libtdsodbc.so

   Define As
       System

   Then add a new System DSN:

   Driver
       FreeTDS

   DSN
       datasourcename

   Server
       hostname

   Database
       databasename

   Port
       1433

6. Use **iodbctestw** to verify that the DSN is working.

```
$> iodbctestw "DSN=datasourcename;UID=username;PWD=password"
```

These are the configuration steps for CentOS or Ubuntu Linux:

1. Install FreeTDS, which on CentOS requires the EPEL repository. The unixODBC package should have already been installed by the package manager.

   CentOS

   ```
   $> yum install epel-release
   $> yum install freetds unixODBC
   ```

   Ubuntu

   ```
   $> sudo apt-get install tdsodbc freetds-bin unixodbc
   ```

2. Use **tsql** to verify that FreeTDS is working.

   ```
   $> tsql -H hostname -p 1433 -U username -P password
   locale is "en_CA.UTF-8"
   locale charset is "UTF-8"
   using default charset "UTF-8"
   1> quit
   ```

3. Add the FreeTDS driver to **/etc/odbcinst.ini**, adjusting paths if necessary:

   CentOS

   ```
   [FreeTDS]
   Driver      = /usr/lib64/libtdsodbc.so.0
   Setup       = /usr/lib64/libtdsS.so.2
   ```

   Ubuntu

   ```
   [FreeTDS]
   Driver      = /usr/lib/x86_64-linux-gnu/odbc/libtdsodbc.so
   Setup       = /usr/lib/x86_64-linux-gnu/odbc/libtdsS.so
   ```

4. Add the DSN to **/etc/odbc.ini**:

   ```
   [datasourcename]
   Driver     = FreeTDS
   Server     = hostname
   Database   = databasename
   Port       = 1433
   ```

5. Use **isql** to verify that the DSN is working.

   ```
   $> isql datasourcename username password
   +---------------------------------------+
   | Connected!                            |
   |                                       |
   | sql-statement                         |
   ```

```
| help [tablename]                     |
| quit                                 |
|                                      |
+--------------------------------------+
SQL> quit
```

Adding an ODBC Data Source Host

For general information about navigating Lasso Server Admin and adding a host to a data source, see the section *Configuring Datasources* in the *Instance Administration and Configuration* chapter.

To add a new ODBC host:

1. In the "Datasources" section of Lasso Server Admin, click the *ODBC* item.

2. Click the *Add host* item to reveal the host connection form.

3. Enter the System DSN name of the ODBC connection in the "Host" field.

4. Enter the TCP port of the ODBC connection in the "Port" field.

5. Select "Yes" from the *Enabled* drop-down to enable the host.

6. Enter a username for the host in the "Username" field and a password for the host in the "Password" field. Lasso will connect to the data source and all databases therein using this username and password by default.

7. Click the *Add host* button.

8. Once the host is added, the new host appears in the "Hosts" listing below.

Databases in newly created hosts are enabled by default. The administrator can disable databases by expanding the database listing and setting the *Enabled* drop-down to "No". With the ODBC data source added here, **inline** methods can use the **-database** parameter to specify the name of the database to perform an action on.

Specifying ODBC Hosts in Inlines

Setting up a data source host in Lasso Server Admin is the best way to ensure that access to the data source is centrally controlled. However, it can sometimes be beneficial to access a data source host without a lot of configuration. This section describes how to construct an **inline** method that accesses an ODBC data source host. See the section *Inline Connection Options* for full details about specifying hosts in inlines.

To access an ODBC host directly in an **inline** method, the **-host** parameter can be used to specify all of the connection parameters. The **-host** parameter takes an array that should contain the following elements:

- **-datasource** should be specified as "odbc".

- **-name** should be specified as the System DSN.

- **-username** set to the user to authenticate as, if required.

- **-password** set to the specified user's password to authenticate the connection, if required.

The following code shows how a connection to an ODBC data source hosted on the same machine as Lasso might appear:

```
inline(
   -host=(:
      -datasource='odbc',
      -name='System_DSN_Name',
      -username='username',
      -password='secret'
   ),
```

```
    -findAll,
    -database='database',
    -table='table'
) => {^
    found_count
^}
```

3.3.7 SQL Server Data Sources

Lasso Server can communicate with Microsoft SQL Server databases configured to accept TCP/IP client connections. For more information on SQL Server, visit https://www.microsoft.com/en-us/cloud-platform/sql-server/.

Requirements

- Microsoft SQL Server 2005–2012

- The Lasso Server machine must have the SQL Server client libraries installed.

    OS X and Linux
    The FreeTDS libraries need to be compiled and installed, for which the source can be found at http://www.freetds.org/. (Instead of compiling from source, you may first want to look into installing via a package manager such as **apt**, **yum**, **macports**, or **homebrew**.) See *Configuring ODBC Connections* for an example configuration.

    Windows
    The required client libraries are pre-installed with the operating system.

Configuring SQL Server

The SQL Server database server must be configured with a user that has access to all of the databases, tables, and other resources that will be published through Lasso. Consult the SQL Server documentation[30] for help configuring its built-in security.

Adding a SQL Server Data Source Host

For general information about navigating Lasso Server Admin and adding a host to a data source, see the section *Configuring Datasources* in the *Instance Administration and Configuration* chapter.

To add a new SQL Server database host:

1. In the "Datasources" section of Lasso Server Admin, click the *SQLServer* item.

2. Click the *Add host* item to reveal the host connection form.

3. Enter the IP address or domain name where the SQL Server data source is being hosted followed by a backslash and the name of a database in the "Host" field. (e.g. "www.example.com\MyDatabase")

4. Enter the TCP port the SQL Server service is listening on in the "Port" field. This is commonly "1433" for SQL Server.

5. Select "Yes" from the *Enabled* drop-down to enable the host.

6. Enter a username for the host in the "Username" field and a password for the host in the "Password" field. Lasso will connect to the data source and all databases therein using this username and password by default.

7. Click the *Add host* button.

8. Once the host is added, the new host appears in the "Hosts" listing below.

---

[30] https://msdn.microsoft.com/library/mt590198(v=sql.1).aspx

Databases in newly created hosts are enabled by default. The administrator can disable databases by expanding the database listing and setting the *Enabled* drop-down to "No". With the SQL Server data source added here, **inline** methods can use the **-database** parameter to specify the name of the SQL Server database to perform an action on.

Specifying SQL Server Hosts in Inlines

Setting up a data source host in Lasso Server Admin is the best way to ensure that access to the data source is centrally controlled. However, it can sometimes be beneficial to access a data source host without a lot of configuration. This section describes how to construct an **inline** method that accesses a SQL Server data source host. See the section *Inline Connection Options* for full details about specifying hosts in inlines.

To access a SQL Server host directly in an **inline** method, the **-host** parameter can be used to specify all of the connection parameters. The **-host** parameter takes an array that should contain the following elements:

- **-datasource** should be specified as "sqlserver".

- **-name** should be specified as the IP address or domain name of the machine hosting SQL Server.

- **-port** is optional and defaults to "1433" if no port is specified.

- **-username** set to the user to authenticate as.

- **-password** set to the specified user's password to authenticate the connection.

The following code shows how a connection to a SQL Server data source hosted on the same machine as Lasso might appear:

```
inline(
   -host=(:
      -datasource='sqlserver',
      -name='(local)\MYDB',
      -username='username',
      -password='secret'
   ),
   -findAll,
   -database='database',
   -table='table'
) => {^
   found_count
^}
```

3.3.8 SQLite Data Sources

Lasso Server comes with an embedded high-performance data source called SQLite. This data source is used to store Lasso's internal site preferences and security settings. SQLite is installed, enabled, and preconfigured within Lasso Server by default. No further set up or installation of SQLite is required.

SQLite databases are stored in the "SQLiteDBs" folder within each instance's home directory. By default this folder contains databases that are required for Lasso Server to function. Custom databases may be created and added to this folder and Lasso **inline** methods will automatically have access to them using the **-database** parameter.

Part II

Language Elements

Calling Lasso

This chapter describes two different methods of calling Lasso: either using Lasso as a script processor on the command line or using Lasso as a web application server through the web browser.

This information is presented at the start of this part as it is vital to understanding the rest of the topics and examples given in this guide.

4.1 Calling Lasso Web Pages

Lasso is most often used to serve web applications. Lasso code can be embedded in HTML pages and executed before they are served to web visitors. A page that includes Lasso code within it is referred to as a *Lasso page*.

Lasso code is embedded within a regular HTML file by inserting the code between a certain set of delimiters. These delimiters consist of an opening and a closing element. Outside of these delimiting elements, all text is treated as if it were plain text string literals. Such text is not interpreted by Lasso. Between the delimiters, all text is parsed and executed as Lasso code. In this manner, an HTML file becomes a template, with the final resulting data being the combination of whatever plain text the file contained, plus whatever text was generated via any contained Lasso code.

The available delimiting elements are described below. The "..." shown between the delimiters illustrates where Lasso code would be inserted by the developer.

```
<?lasso ... ?>
```

```
<?= ... ?>
```

```
[ ... ]
```

All three delimiters will produce identical results. Multiple expressions can be contained between these delimiters. The result from each contained expression is converted to a string and then concatenated together along with any plain text existing outside of the delimiters.

Although square brackets (`[ ... ]`) are enabled by default, they can be disabled by placing `[no_square_brackets]`, usually at the top of the page, outside the delimiters. Once the Lasso parser encounters `[no_square_brackets]`, square brackets are be turned "off" and any subsequently encountered square brackets will be treated as plain text. Turning square brackets off works on a per-file basis, and cannot be turned back on once they are off. To illustrate how Lasso code is embedded within a Lasso page, the following code may be stored in a file named "test.lasso" contained within the web server root.

```
<!DOCTYPE html>
<html>
<head>
    <title>My Lasso Page</title>
</head>
<body>
    <p><?= 'The current date is ' + date ?>.</p>
</body>
</html>
```

The above begins with plain HTML markup, then adds two Lasso code expressions into the document using a delimiter pair. When this file is loaded through a browser, the code shown above is executed and the result is returned to the web browser.

If the embedded message is not visible in the web browser or an error occurs, then you should make sure that Lasso Server has been properly installed on your machine. (See the appropriate installation instructions for your operating system in the *index_server*.)

4.2 Calling Lasso from the CLI

Lasso code can be saved in a file and then executed on the command line. This style of execution happens directly and does not require a web server or web browser. Additionally, since a web server or web request is not in effect during such execution, none of the web serving–specific functionality is available in this context. (For more information on the command-line tools that come as part of the Lasso platform, see the *Command-Line Tools* chapter.)

4.2.1 Using the lasso9 Tool

The **lasso9** executable is a tool included with Lasso that handles the parsing and execution of Lasso code from the command line. For example, the following text could be placed into a file "test.lasso":

```
'The current date is ' + date
```

The file can be executed from the terminal using lasso9. If the reader has created such a test file and has done a **cd** to the location of the file, then the file can be executed like so:

```
$> lasso9 ./test.lasso
The current date is 2012-08-08 15:07:25
```

If the terminal reports the command was not found, or you receive some other error, then you should make sure that Lasso has been installed properly on your machine. See the appropriate installation instructions for your operating system in the *index_server*.

When running Lasso code on the command line, delimiters are not required, though they can be used. By default, text is assumed to consist of Lasso code only, unless the file's text begins with an open angle bracket (**<**), in which case it is assumed to start out as plain text. For example, the *test file shown in "Calling Lasso Web Pages"* could be run on the command line and would generate the expected HTML result, including the embedded message.

4.2.2 Associating Files with the lasso9 Tool

Files containing Lasso code can be directly associated with the **lasso9** tool by inserting a standard "hashbang" or "shebang" line *at the very top of the file* and making the file executable (usually accomplished by running **chmod +x test.lasso**).

The hashbang line for a standard installation looks like this:

```
#!/usr/bin/env lasso9
```

Using the same "test.lasso" file as before, but placing the hashbang line at the top, the complete example would look as follows:

```
#!/usr/bin/env lasso9
'The current date is ' + date
```

Once it has been made executable, the file can be directly executed on the command line.

```
$> ./test.lasso
The current date is 2012-08-08 15:07:25
```

The result, regardless of the execution method, is identical. Also, note that the file's extension (".lasso" in this case) is irrelevant when executing Lasso code on the command line. The example file could just have easily been named "test", with no extension, and the results would have been the same.

4.2.3 Executing Code Directly

The **lasso9** tool includes the **-s** option for passing a string of Lasso code to execute. This method bypasses the need to first place the code in a file. Instead, the source code can be given directly to lasso9 when it is invoked.

```
$> lasso9 -s "'The current date is ' + date"
The current date is 2012-08-08 15:07:25
```

Running the above example will produce the same output as the previous examples. Care must be exercised when using this method because the shell will interpret some characters for itself, therefore distorting the source code given to the command. Because of this, it is generally recommended that such source code be surrounded between double quotes and that single quotes be used for any contained string literals, as illustrated in the example above.

4.2.4 Executing Code from STDIN

The **lasso9** tool can also accept code to execute from STDIN. This is useful when piping results from one command to lasso9 in order for it to execute the given code. In order to have lasso9 receive its code from STDIN, the **--** argument is used. The following example uses the standard **echo** command to pipe code from STDIN for lasso9 to read and execute:

```
$> echo "'The current date is ' + date" | lasso9 --
The current date is 2012-08-08 15:07:25
```

Literals

A *literal* is an object with its own special syntax that allows it to be inserted directly into code. Lasso supports **string**, **boolean**, **integer**, **decimal**, **tag**, **staticarray**, and **generateSeries** literals, the words **null** and **void**, and comments.

The method for expressing these literals is largely similar to other scripting languages. For example, an integer literal is expressed, as one would expect, by simply using the numeral in the source text. **23** is an example of an integer literal.

5.1 String Literals

Lasso supports two kinds of string literals: quoted and ticked. Quoted strings can contain escape sequences, while ticked strings cannot. Both quoted and ticked string literals can contain line breaks, and produce the same type of string objects. The differences between the two types of literals are handled entirely during parsing. All strings in Lasso are Unicode strings, which means that a string can contain any of the characters available in Unicode.

5.1.1 Quoted Strings

The first kind of string literal is a *quoted string*, which is a series of zero or more characters surrounded by either single or double quotes. If a string literal begins with a single quote, then it must end with a single quote. The same holds for a string literal that begins with a double quote; it must end with a double quote.

```
'This is a string literal'
"This is also a string literal"
```

Within this type of string literal, the backslash character (**\x5C**) is interpreted as an escape character. This means that when a backslash is encountered in a string literal, it changes the meaning of the immediately following character(s). For example, a backslash is required in order to create a string literal that contains the quote character that surrounds the string.

```
'This is a \'string literal\' with quotes'
"This is also a \"string literal\" with quotes"
```

Note that a backslash is not required in order to insert the alternate quote type into a string literal. For example, a double-quoted string can contain a single quote without having to escape it.

```
"Escaping this single quote isn't required"
```

A backslash is also required in order to insert a literal backslash into a string. In order to embed a backslash into a string, two backslashes must be used.

```
'This string literal has a backslash \\ in it'
```

A backslash followed by an end-of-line (a literal line feed or carriage return or carriage return/line feed pair) will cause that end-of-line and all following literal whitespace to be removed from the resulting string. The string resumes starting with the first encountered non-whitespace character. This sort of escape sequence can be useful for preserving the visual formatting of a string literal while removing the characters used to achieve that formatting from the resulting string.

```
'This string \
        had a break in it'
// => This string had a break in it
```

The backslash can also be used to insert Unicode characters represented either by hex code, or by character name. Where the Unicode character name is used, the name must be the official Unicode name for that character, enclosed between a set of colons. Additionally, it is an error to use an unrecognized character name.

Also supported are a series of commonly used escape sequences. The following table shows all of the permissible escape sequences.

Table 5.1: Supported String Escape Sequences

Escape Sequence	Value	Description
\x*dd*	Unicode character	1–2 hex digits
\u*dddd*	Unicode character	4 hex digits
\U*dddddddd*	Unicode character	8 hex digits
ddd	Unicode character	1–3 octal digits
\:*NAME*:	Unicode character	Unicode character name
\a	0x07	bell
\b	0x08	backspace
\e	0x1B	escape
\f	0x0C	form feed
\n	0x0A	line feed
\r	0x0D	carriage return
\t	0x09	tab
\v	0x0B	vertical tab
\"	0x22	double quote
\'	0x27	single quote
\?	0x3F	question mark
\\	0x5C	backslash
\<end-of-line>	none	escaped whitespace

5.1.2 Ticked Strings

A *ticked string* is a series of zero or more characters surrounded by a pair of backticks (`\x60`). Within a ticked string, the backslash character holds no special meaning. Ticked strings do not recognize any escape sequences, and this can make them particularly useful when using regular expressions which often require many backslashes. (Using regular quoted strings, the backslashes would themselves have to be doubled.) The caveat for this is that a literal backtick character cannot appear within a ticked string.

```
`This is a ticked string`
`A ticked string can contain 'single quotes', "double quotes",
\backslash characters\ and more - anything except backticks!`
```

5.2 Boolean Literals

A *boolean* is an object that is either "true" or "false". Lasso supports the creation of these objects by using the word **true** or **false** directly in the source code.

```
true
false
```

type **boolean**

boolean()

boolean(*obj::any***)**

> Casts a value to a boolean value. Only the following objects and values evaluate to "false"; all others are "true":
>
> - **integer** zero: **0**
>
> - **decimal** zero: **0.0**
>
> - **null** and **void**
>
> - calling **boolean** with no parameter
>
> - empty **string**: **'', "", ``**

Note: Although the empty string evaluates to "false", this functionality is deprecated. Instead, call **string->size** to check for empty strings.

5.3 Integer Literals

An *integer* is a whole number. Integers can be positive or negative, and Lasso puts no limit on the size of an integer. Integers consist of the digits 0 through 9 and can be written directly into the source code.

```
1
-4
+937
11801705635790
```

Integers can also be written using hexadecimal notation. Hexadecimal integers begin with a zero followed by an upper or lowercase "x" followed by one or more hexadecimal digits (0–9 and A–F). Either upper or lowercase letters are permitted. A hexadecimal integer literal is always interpreted as a positive integer.

```
0x1
0x04
0x3A9
0x11F018BE6
```

Both numeric and hexadecimal integer literals produce the same **integer** type with the same set of member methods.

See the *Math* chapter for more information on the **integer** type.

5.4 Decimal Literals

A *decimal* is a fractional number. Decimal numbers contain a decimal point and therefore are called "decimals". Lasso supports 64-bit decimals. This gives Lasso's decimal numbers a range from approximately negative to positive 2x10^300 and with precision down to 2x10^-300. A decimal literal begins with an optional "-" or "+" character followed by zero or more digits, a decimal point, one or more additional digits, and ending with an optional exponent. A decimal exponent begins with an upper or lowercase "E", followed by an optional "-" or "+" character followed by one or more digits. Lasso also supports decimal literals for NaN (not a number) as well and positive and negative infinity. (Note that case is irrelevant when using the `NaN` and `infinity` literals.)

```
.1
-.89
1.0
-93.42e-4
+93.42e4
NaN
infinity
-infinity
```

See the *Math* chapter for more information on the `decimal` type.

5.5 Tag Literals

A *tag* is an object that uniquely represents a particular string of characters. Unlike strings, tags cannot be modified. Tags are used to represent type and method names as well as variable names. A tag should begin with a letter or underscore, followed by zero or more letters, numbers, underscores, or period characters. Tags cannot contain spaces.

Tags are commonly used when applying type constraints to methods, data members, and variables; though they have other purposes as well, such as *type and object introspection*.

A tag literal consists of two colons followed by the tag's characters.

```
// Creates a tag object representing "name"
::name
```

In Lasso, the `tag` type is used in many different locations. For example, when asking an object what type it is with `type`, it will reply with a tag object representing its name. Since there will be only one tag object for every individual name, comparing tags for equality is very fast.

tag_exists(*value::string*)
> Checks if a tag matching the given string value exists without attempting to create a tag. Returns the tag if it exists or "void" if it does not.

5.6 Staticarray Literals

Lasso's *staticarray* type is an efficient, non-resizable collection for holding any series of object types which is used in many places in Lasso. Staticarrays are created in the same way as any object, but Lasso supports a shortcut syntax to produce staticarrays. This expression begins with an open parenthesis immediately followed by a colon, then zero or more comma-delimited expressions, ending with the closing parenthesis.

```
// Creates a staticarray containing 1, 2, and "hello"
(: 1, 2, 'hello')
```

See the *Collections* chapter for more information on the `staticarray` type.

5.7 Series Literals

Lasso's *generateSeries* type is a quick and efficient way to create a *series* or *range* of integers for use with query expressions. The shortcut syntax for creating a series consists of a starting integer and ending integer separated by the word "to". An optional integer specifying the step size, which defaults to 1, can be added after the word "by".

```
0 to 10 by 2
// => 0, 2, 4, 6, 8, 10
```

See the *Query Expressions* chapter for more information on the `generateSeries` type.

5.8 Null and Void

type **null**

> A *null* represents a blank or empty value, indicated in code by the word **null**. All types inherit from **null**, so its member methods (listed under *Type/Object Introspection Methods* in the *Types* chapter) can be used by any type.

type **void**

> A type indicated in code by **void** which is similar to **null** in that it is only ever equal to itself, but indicates a non-existent rather than an empty value.

5.9 Comments

Lasso supports three types of comments: single line comments, block comments, and doc comments. Single line and block comments are ignored, having no effect on the execution of any nearby code. Doc comments are saved with the adjacent method, type, or trait, as explained below.

5.9.1 Single Line Comments

A *single line comment* begins with two forward slashes (**//**). The comment runs until the end of the line, which is either a carriage return, line feed, or a carriage return/line feed pair.

```
local(n) = 123 // This is the first comment
// This is another comment
#n += 456
```

Note that when embedding Lasso code between a set of delimiters, a closing delimiter on the same line as a single line comment will be skipped by the Lasso parser.

5.9.2 Block Comments

A *block comment* permits a large section of code to be commented. Any characters, as well as multiple lines, are permitted between the opening delimiter (**/***) and closing delimiter (***/**). Block comments cannot be nested.

```
local(n) = 123
/* this is a block comment
it has multiple lines */
#n += 456
```

5.9.3 Doc Comments

A *doc comment* permits a block of documentation to be associated with either a type, trait, or method. This comment is not processed by Lasso in any way, but is saved as-is with the object. A doc comment begins with the opening doc comment delimiter (**/**!**) and runs until a closing delimiter (***/**). Any characters can appear within a doc comment, and a doc comment can consist of multiple lines.

Doc comments can only appear in the following locations:

- Immediately before a type definition
- Immediately before a trait definition
- Immediately before a member or unbound method definition
- Immediately before a trait's provide or require section

```
/**!
    This doc comment is associated with this method
*/
define foo->xyz() => { ... }

/**!
    This doc comment is associated with this type definition
*/
define foo => type {
    /**!
        Doc comment for the type's xyz() method
    */
    public xyz() => { ... }
}

/**!
    This doc comment is associated with this trait
*/
define tBar => trait {
    /**!
        Doc comment for the trait's doIt() method
    */
    provide doIt() => { ... }
}
```

Doc comments for a type can be set and retrieved programatically using the **tag->docComment** method, as long as Lasso is run with the **LASSO9_RETAIN_COMMENTS** variable enabled.

```
$> env LASSO9_RETAIN_COMMENTS=1 lasso9 -s "::array->docComment"
/**!
An array is an object that can hold multiple values...

$> env LASSO9_RETAIN_COMMENTS=1 lasso9 -s "
::boolean->docComment = 'Boolean objects are either true or false.'
::boolean->docComment
```

```
    "
```
Boolean objects are either true or false.

Variables

A *variable* is a construct for saving and referencing the result of an expression. A variable points to an object and permits that object to be saved and used repeatedly later.

There are two types of variables: local variables and thread variables. The type of the variable defines its scope and the rules about using it. Each variable is given a name, and that name is used to access the variable's value. An object that a variable points to can be changed, or reassigned, as described in the *Operators* chapter.

6.1 Variable Names

Lasso variable names should begin with a letter or an underscore followed by a letter, then zero or more letters, numbers, underscores, or period characters. Variable names are case-insensitive, so a variable named "rhino" can also be accessed with "RhINo" as well.

6.2 Local Variables

Each capture runs with its own set of variables. These are called *local variables* or *locals*, and they are the most commonly used type of variable. Locals begin and end within the capture in which they are defined, though the objects they point to may exist beyond that point. Nested captures also have access to any locals defined in their parent capture before their own definition.

A local must be defined before it can be used. When a variable is defined, it is generally done so along with an initial value to be assigned to that variable. If an initial value is omitted, the variable will have the default value of "null". Multiple locals can be defined at one time, either with or without default values, using the following syntax examples:

```
// Defines local "name" set to the value of the expression
local(name = expression)
local(name) = expression

// Defines locals "name" without a value and "b" set to 1
local(name, b = 1)
```

A local can be accessed using two different methods. In the first method, the local variable may or may not have previously been defined. If the local has not been defined, then it is defined and assigned a value of "null". Regardless, the value of the variable is produced as the result. This is only the case when one variable name is used and when it is not accompanied by an initial value.

```
local(name)
// => // The value of "name", potentially creating "name"
```

Local variables can also be accessed using the "#" character before the name. This is the preferred method for accessing local variables.

```
#name
// => // The value of "name"
```

When using this method, the local variable must have already been defined or it is considered an error. This error-checking is done at the time the code is parsed, meaning that the local definition must *physically* precede the "#" access point within the source code.

The set of local variables for each capture is determined as the code is compiled and cannot be modified at runtime, unlike thread variables which can be given names dynamically.

6.2.1 Parameter Pseudo-locals

Lasso permits the parameter values given to a method to be accessed by position, using the local variable symbol "#" followed by an integer value. The integer value corresponds to the position of the desired parameter value, beginning with "1". For example, in a method given two parameters, the first would be available using **#1** and the second would be available using **#2**.

See the **Methods** chapter for information on methods and method parameters.

6.3 Thread Variables

Thread variables, or *vars*, are variables that are shared and accessible outside of any particular capture, yet are restricted to the currently executing thread. Each thread maintains its own set of vars. Vars are useful for maintaining program states that go beyond the operation of any one method.

Vars are created in a manner similar to locals, but instead use the **var** declaration.

```
// Defines var "name" set to the value of the expression
var(name = expression)
var(name) = expression

// Defines vars "name" without a value and "b" set to 1
var(name, b = 1)
```

A var created without an initial value will be given the default value of "null".

Vars can be created using an expression value for a name, unlike locals which require a fixed literal name. This expression must result in a string or a tag object. That value is used as the variable's name.

```
// Defines var with name of nameExpr
var(nameExpr = expression)
```

Because a literal variable name can resemble a method call with no parameters, if the variable name is intended to be the result of a method call, then that call must be given empty parentheses **()** to disambiguate.

```
// Defines var with the name of what nameCall() returns
var(nameCall() = expression)
```

A var can be accessed using two methods, similar to that of local variables. First, the var may simply be referenced using the **var** syntax along with the var's name. The var may or may not have previously been defined. If the var has not been defined, then it is defined and assigned a value of "null". The value of the variable is produced as the result. This is only the case when one variable name is used and when it is not accompanied by an initial value.

```
var(name)
// => // The value of "name", potentially creating "name"
```

Vars can also be accessed using the "$" character before the name. When using this method, an error is returned if the var has not been previously defined.

```
$name
// => // The value of "name"
```

6.4 Type Constraints

A *type constraint* can be applied to a local or thread variable in order to ensure that the value of the variable is always an object of a particular type or trait. For example, a local variable could be constrained to always hold a string object. If an attempt was made to assign to that variable a non-string object, such as an integer, the assignment would fail.

Lasso is a dynamically typed language, and, by default, variables can hold any type of object. Type constraints permit a developer to restrict variables to hold only particular object types or containing a particular trait in order to ensure that the code operating on those variables is given valid inputs.

Type constraints are applied when a local or thread variable is first defined. This is done by supplying a **tag literal**, which consists of two colons (`::`) and then the name of the type or trait to which the variable will be constrained, immediately following the variable name. The following example applies constraints to a local and a var:

```
local(lname::integer) = 0
var(vname::trait_forEach) = array
```

In the above example, "lname" is constrained to hold only integers, and "vname" is constrained to hold only types supporting **trait_forEach**. The next example shows valid and invalid usage of the two variables:

```
#lname = 400
// => // Valid: 400 is an integer

#lname = 'hello'
// => // FAILURE: #lname can only hold integers

$vname = (: 1, 2, 'hello')
// => // Valid: staticarrays support trait_forEach

$vname = 940
// => // FAILURE: $vname can only hold types that support trait_forEach

local(lname) = 'hello'
// => // FAILURE: #lname can still only hold integers
```

When applying a type constraint in a variable declaration, a provided default value is required.

```
local(lname::integer, x, y, z)
// => // FAILURE: #lname requires default value
```

6.5 Decompositional Assignment

Lasso will "decompose" the right-hand side value (RHS) of an assignment when the left-hand side value (LHS) is a local declaration containing just a list of variable names. This supports wildcards (the _ character) as well as nested name lists. Any type that supports **trait_forEach** can be used like this on the RHS.

The following examples should help clarify:

```
local(one, two, three, four) = (: 1, 2, 3, 4, 5, 6)
```

```
#one
// => 1
#two
// => 2
#three
// => 3
#four
// => 4

local(_, two, _, four) = (: 1, 2, 3, 4, 5, 6)

#two
// => 2
#four
// => 4

local(_, two, _, four) = 1 to 100 by 3

#two
// => 4
#four
// => 10

local(one, _, three, (_, four)) = array('a', 'b', 'c', array('d', 'e'))

#one #three #four
// => ace

local(wanted, _, w2) = 'ABCDEFGH'

#wanted
// => A
#w2
// => C
```

Note that the local must include more than one element, and none of the elements can be assigned values.

```
local(x) = #foo
// => // Unchanged, works as expected

local(x, _) = #foo
// => // Fine, grabs first #foo

local(x = 1, _) = #foo
// => // FAILURE: x cannot have value
```

Also note that assign-produce (`:=`) cannot be used with decompositional assignment, and that quoted variable names are not permitted.

Operators

An *operator* is a special symbol that, combined with one or more operands, performs an operation using those operands and, generally, produces a value.

Lasso supports the standard arithmetical operators and logical operators as well as numerous other useful operations. Operators can be *unary*, taking only one operand, *binary* requiring two operands, or *ternary*, in the case of the conditional operator, requiring up to three operands.

Lasso permits the behavior of some operators to be controlled by the operand objects themselves. This is accomplished in an object by having it implement a method whose name matches the symbol for that operator. For example, a type that needed to support addition would implement a method named **+** which accepts one parameter and returns the resulting value.

7.1 Assignment Operations

Assignment places the result of an expression into a destination. The destination must be a local or thread variable, or it must be an appropriately named method call. Lasso supports two types of assignment operators: assign-produce (**:=**) which produces the assigned value, and standard assignment (**=**) which does not.

```
// "dest" assigned value of expression
dest = expression

// "dest" assigned value of expression, "dest" produced
dest := expression
// => // Produces a reference to "dest"
```

An assign-produce operation, which produces the left-hand operand, is right-associative so that multiple assignments can be lined up. The following assigns "1" to "dst1", "dst2" and "dst3" and also produces "1":

```
dst1 := dst2 := dst3 := 1
// => 1
```

Locals and vars can both be assigned using assignment syntax.

```
// local "l" assigned expression
#l = expression

// local "l" assigned expression
local(l) = expression

// var "v" assigned expression
$v = expression

// var "v" assigned expression
var(v) = expression
```

Variables and data members are the only elements to which values can truly be assigned, but Lasso permits methods to be created that mimic the act of assignment. This is done by naming the method with a "=" character at the end. For example, a method that wanted to accept assignment for **foo** would be named **foo=**. Such a method must accept at least one parameter and must return the assigned value as if it were being called in the role of assign-produce (**:=**). Methods that permit such assignment are useful as "setters" and let an object control how the assignment is ultimately made. See the *Types* chapter for more detail on creating *setter methods*.

7.2 Arithmetical Operations

"Arithmetic" typically refers to mathematical operations using integer or decimal numbers, as explained in the *Math* chapter. However, an arithmetical operator can be applied to any object that supports the operation.

7.2.1 Basic Operators

These operators are all binary, requiring two operands. All of these operators can be implemented by a type containing the properly named method. Only the left-hand operand's method is called. None of these operators should modify either operand, but must return a new object. The examples that follow show the use of each operator:

```
op1 + op2
// => // Returns the value of adding op2 to op1

op1 - op2
// => // Returns the value of subtracting opt2 from op1

op1 * op2
// => // Returns the value of multiplying op1 by op2

op1 / op2
// => // Returns the value of dividing op1 by op2

op1 % op2
// => // Returns the remainder of dividing op1 by op2 (modulo operation)

(: 10 + 3, 10 - 3, 10 * 3, 10 / 3, 10 % 3)
// => staticarray(13, 7, 30, 3, 1)
```

7.2.2 Assignment Operators

While the basic arithmetical operators use their operands to produce a new value, Lasso supports syntax for applying the operator *to* one of the operands. The following operators perform their operation and assign the result to the left-hand side operand. Only the left-hand operand can be assigned to and not every expression is capable of being assigned to, as described in the section on *assignment operations*. These assignment expressions do not produce a value.

```
// Equivalent to op1 = op1 + op2
op1 += op2

// Equivalent to op1 = op1 - op2
op1 -= op2

// Equivalent to op1 = op1 * op2
op1 *= op2

// Equivalent to op1 = op1 / op2
```

```
op1 /= op2

// Equivalent to op1 = op1 % op2
op1 %= op2
```

During parsing, these operators are expanded to their regular arithmetical and assignment operations, so a type does not need to do anything to support them aside from implementing the assignment operator method and the appropriate arithmetical operator method.

7.2.3 Pre-/Post-Increment and Decrement Operators

There is a common need to "advance" an object in a bidirectional manner. Usually this is done using integers as counters, though the concept can be applied elsewhere. Lasso supports the increment and decrement operators (**++** and **--**) in both pre and post modes.

Pre-incrementing and pre-decrementing an object will add or subtract 1 to or from the object and then produce that object as a result. Post-incrementing and post-decrementing an object first copies that object, then adds or subtracts 1 to or from the original operand, then produces the copied object as a result.

```
// Pre-increment "op"
++op
// => // Produces the newly incremented "op"

// Pre-decrement "op"
--op
// => // Produces the newly decremented "op"

// Post-increment "op"
op++
// => // Produces a copy of "op" before incrementing

// Post-decrement "op"
op--
// => // Produces a copy of "op" before decrementing
```

These increment/decrement operators are translated into regular arithmetical method calls with "1" as the method parameter. This means that if a type is intended to be used with the increment (**++**) and decrement (**--**) operators, all that's necessary is to implement **+** and **-** which will be called with "1" as the parameter.

7.2.4 Positive and Negative Operators

Lasso supports the unary operators which are typically intended to change the sign of an integer or decimal number. These operators can be applied to any object that supports them. When applied, these operators will produce a new object, leaving the single operand unchanged.

```
+op1
// => // Produces a new object whose value is positive op1

-op1
// => // Produces a new object whose value is negative op1
```

Types can implement this operator by defining a method named **+** or **-** that accepts zero parameters. When unary **+** or **-** is applied to **integer** or **decimal** literals, no method call is generated. Instead, the positive or negative number is created from the beginning.

7.3 Boolean Operations

Boolean describes the values "true" and "false". Lasso supports several operators that either treat their operands as boolean values and/or produce boolean values. These operators are broken down into several categories. (See the definition of **boolean** for how other other values are cast to boolean types.)

7.3.1 Logical Operators

There are three *logical operators*. The first is the unary operator "not". This operator treats its single operand as a boolean value and produces the opposite of that value. The "not" operator turns a "true" into a "false" and a "false" into a "true". Although the operand can be of any type, this operator always produces a "true" or "false" value. The "not" operator can take one of two forms: an exclamation mark (**!**) or the **not** keyword.

```
!true
// => false
```

```
not false
// => true
```

The other two logical operators are logical "and" and logical "or", and they also can take two forms: double ampersands (**&&**) or the **and** keyword for logical "and", and double pipes (**||**) or the **or** keyword for logical "or".

These binary operators treat their first operand as a boolean value and perform their operation based on that value. Logical "and" inspects its first operand, and if it is "true", produces its second operand. If the first operand is "false", logical "and" will produce the value "false". Logical "or" inspects its first operand, and if it is "true", produces that first operand. If the first operand is "false", logical "or" will produce the second operand.

```
op1 && op2
// => // Returns "false" if either op1 or op2 evaluates to "false" else opt2
```

```
op1 || op2
// => // Returns op1 if it evaluates to "true" else op2
```

These operators perform shortcut evaluation, meaning that if the result of the operation is determined before the second operand is evaluated, then the second operand will not be evaluated. Also note that the behavior of the logical operators cannot be defined by the operand objects.

7.3.2 Equality Operators

The *equality operators* are used to determine if one object is logically equivalent to another. These operators are split into positive and negative equality tests as well as strict and non-strict equality tests. A positive equality test checks if one object *is equal to* another object while a negative equality test checks if an object *is not equal to* another. Strict equality testing further tests the types of the operand objects. If the right-hand operand is not an instance of the type of the left-hand operand, then the equality test fails. These operators all produce either a "true" or "false" value.

```
op1 == op2
// => // Produces "true" if op1 is equal to op2 else false
```

```
op1 != op2
// => // Produces "true" if op1 is not equal to op2 else false
```

```
op1 === op2
// => // Produces "true" if op1 is both equal to and the same type as op2 else false
```

```
op1 !== op2
// => // Produces "true" if op1 is not equal to or not the same type as op2 else false

(: 3 == 3.0, 3 != 3.0, 3 === 3.0, 3 !== 3.0)
// => staticarray(true, false, false, true)
```

If an object is to be tested for equality against another, its type must implement a method named **onCompare**. The **onCompare** method is automatically called at runtime to perform equality checks. It is only called on the left-hand operand, and this method must accept one parameter, which is the right-hand operand. When called, it indicates whether the left-hand operand is less than, equal to, or greater than the right-hand operand by returning either an integer less than zero, zero, or greater than zero, respectively. The act of checking the object types in the case of strict equality testing is automatically performed by the runtime, so a type need not account for that scenario in its own implementation of **onCompare**.

7.3.3 Relative Equality Operators

The *relative equality operators* indicate whether an object is less than, greater than, or possibly equal to another object. These operators all produce either a "true" or "false" value.

```
op1 < op2
// => // Produces "true" if op1 less than op2 else "false"

op1 > op2
// => // Produces "true" if op1 greater than op2 else "false"

op1 <= op2
// => // Produces "true" if op1 less than or equal to op2 else "false"

op1 >= op2
// => // Produces "true" if op1 greater than or equal to op2 else "false"
```

Types control how equality checks behave by implementing the **onCompare** method as described above in the section on *equality operators*. Because **onCompare** is required to return an integer value (either zero, less than zero, or greater than zero), it can handle all possible types of equality tests.

7.3.4 Containment Operators

There are two *containment operators* used to test if an object "contains" another object. One checks for positive containment (**>>**) and the other for negative containment (**!>>**). Both are binary operators and both produce either a "true" or "false" value.

```
op1 >> op2
// => // Produces "true" if op2 is contained within op1 else false

op1 !>> op2
// => // Produces "true" if op2 is not contained within op1 else false
```

In order to support containment testing, a type must implement a method named `contains`. This method must accept one parameter, which is the right-hand operand, and must return a boolean "true" or "false". Only the left-hand operand will have its `contains` method called.

Containment testing only logically applies to certain types of objects. For example, it makes no sense to ask what an integer object contains, because it is scalar, consisting of only one value. Containment testing is primarily done on collection types such as **array** or **map**. Objects of those types can contain any number of other arbitrary objects, so it makes sense to query them for their contents.

7.3.5 Conditional Operator

The *conditional operator* allows for concisely implementing if/then/else logic in which an expression is tested and depending on its boolean value, either the "then" or the "else" expressions will be executed and their values produced as the result of the operator. The "then" and "else" can consist of only one expression. The "else" portion of a conditional operator may be omitted. In such a case, if the condition is "false", a "void" object will be produced.

The conditional operator is a ternary operator consisting of the two "?" and "|" characters. The "?" follows the test condition and the "|" delimits the "then" and "else" expressions. A conditional operator with no "else" condition will have no delimiting "|" character.

```
test ? expression1 | expression2
// => // Produces expression1 if test is "true" else expression2

test ? expression
// => // Produces expression if test is "true" else void
```

7.4 Grouping

Sub-expressions can be grouped together by surrounding them with parentheses. This can be used to alter the normal precedence of some operations. All subexpressions in parentheses are evaluated before the expressions surrounding them. The first example below shows how multiplication normally occurs before addition. The second example applies parentheses to have the addition take precedence.

```
2 * 5 + 7
// => 17

2 * (5 + 7)
// => 24
```

7.5 Invocation

Parentheses can be applied to some expressions in order to *invoke* the value. Invoking can have different results for different objects. By default, most objects return a copy of themselves when they are invoked. Methods, when invoked, execute the method, returning its value.

Invoking an object by applying parentheses is always equivalent to directly calling the method named **invoke**. The following examples invoke a local variable and a thread variable with no parameters:

```
#lv()
// => // Produces the value of invoking the object stored in the local "lv"

$tv->invoke
// => // Produces the value of invoking the object stored in the var "tv"
```

Parameters may be given to an **invoke**. The following invokes **#lv** with three parameters:

```
#lv(1, 'two', 3)
// => // Produces the value of invoking the object stored in the local "lv" with those parameters
```

See the *Types* chapter for more information on the *invoke* callback.

It is also possible to dynamically generate parameters and programmatically pass them into an invocation. By first adding the parameters to an array named "my_params" and including a colon after the opening parenthesis of the invocation statement, the following example results in an equivalent invocation as the previous.

```
local(my_params) = array(1, 'two', 3)
#lv(: #my_params)
// => // Produces the value of invoking the object stored in the local "lv" with those parameters
```

This form is useful for passing a set of values from an object of any type supporting **trait_forEach** to a method that accepts a rest parameter.

```
define printArgs(...) => with i in #rest do stdoutnl(#i)
printArgs(: #my_params)

// =>
// 1
// two
// 3
```

The concept behind invocation is somewhat abstract, but it permits objects and methods to operate as *function objects*. This is an object that can be called upon to do an operation with zero or more parameters and produce a value. For example, a sorting routine could employ such an object to handle the actual comparisons between two objects, invoking the object each time it is required, while the routine handles only the shifting of the objects during the sort.

This technique would permit the sorting routine to be customized for a wide variety of object types as well as ascending and descending directions by just switching out the objects designated to handle each permutation while keeping the internal operations identical.

7.6 Target Operation

To *target* means to access a particular member method or data member from an object. The target operator (**->**) is a binary operator accepting the target object as the left-hand operand and the method name as the right-hand operand. Targeting a member method always executes that method, passing along any given parameters.

```
#lv->meth()
// => // Produces the value of calling meth() on the object stored in #lv with no parameters

#lv->meth
// => // Same as the first example, showing parentheses are optional

#lv->meth(40)
// => // Produces the value of calling meth() on the object stored in #lv with 1 parameter

#lv->meth(40, 'sample')
// => // Produces the value of calling meth() on the object stored in #lv with 2 parameters
```

Accessing a data member is accomplished through a similar syntax, but by surrounding the name in single quotes. A data member can only be accessed from within the type in which the data member is defined. When accessing a data member, it is an error to have any value except for **self** as the left-hand operand, and the right-hand operand must be single-quoted.

```
self->'dMem'
// => // Produces the value stored in the "dMem" data member
```

As it is very common to access data and methods using the current "self", Lasso provides a shortcut syntax for accessing members within "self" or inherited members. Using a period (**.**) before the member name will target the current "self". Using two periods (**..**) before the member name will target inherited members, skipping the current "self" and searching for the

member starting from the parent of the type that defined the currently executing member method. Two periods can only be used for methods, as only "self" can access data members.

```
.'dMem'
// => // Produces the value stored in the "dMem" data member (same as self->'dMem')

.meth(1, 2)
// => // Produces the value of calling self->meth(1, 2)

..meth(3, 4)
// => // Produces the value of calling inherited->meth(3, 4)
```

7.6.1 Retarget Operation

The *retarget* operation allows the same target object to be used for multiple method calls. The retarget operator (**&**) is placed between the individual method calls. Retarget is only ever used in the context of a member method call using the target operator (**->**). The target object of the last target operator is used as the object for the retargeted member call. For each method call, the **&** is placed following the method's name, parameters, and capture block (if present).

The retarget operator can be used to string two or more methods together. The return value of the final method will be produced by this type of retarget.

```
object->meth & meth2
// => // Execute meth on the object then execute meth2 and produce its value

object->meth(1, 2) & meth2()
// => // Execute meth on the object then execute meth2 and produce its value
```

Retarget can also be used to change the produced value of a member method call to be that of the target object. This is done by having a trailing **&** at the end of a method call.

```
targetObject->meth(1, 2) &;
// => // Execute meth, but produce targetObject
```

Formatting Retarget

When stringing several method calls together, formatting over multiple lines can help with readability. It is important, however, to keep the **&** on the same line as the *next* method call, and to follow any trailing retarget operators with a semicolon to ensure the expression is ended. This holds only for cases that have a next method and for method call expressions that are not ultimately parenthesized.

The following example illustrates this formatting principle:

```
targetObject->meth(5, 7)
& meth2()
& meth3(90) &;
// => // Execute meth, meth2, meth3, and then produce targetObject
```

7.7 Method Escaping

To *escape* a method is to allow a method to be searched for by name and returned to the caller. The caller can later use that method, executing it by applying parentheses as described in the section on **invocation**. This makes it easy for methods to be treated as regular values and to be used as callbacks. It is an error if the method that is being escaped is not defined.

Both member methods and unbound methods can be escaped. There are two escape method operators, one for member methods and one for unbound methods. Escaping a member method uses the binary escape operator (`->\`), while escaping an unbound method uses the unary escape operator (`\`).

```
#lv->\meth
// => // Produces a reference to the member method "meth" of the object in local "lv"
```

```
\meth
// => // Produces a reference to the unbound method "meth"
```

When a member method is escaped, the resulting value is bound to that target object. This ensures that when the resulting value/method is invoked, that the current "self" will be the object from which the method was escaped. Additionally, if there is more than one method defined under the given name, all of the methods are retrieved. This permits multiple dispatch to be used with an escaped method.

The right-hand method name operand can come from the result of any expression. When using such a dynamic method name, the expression must be surrounded in parentheses to disambiguate.

```
#lv->\(meth + 'name')
// => // Produces a reference to the member method whose name matches the
//        result of appending "name" to the value returned by "meth"
```

Although the escape operators are used to find methods by name, the object produced by the operators is a *memberstream*. This object manages the finding of the desired method, the potential bundling of the target object (in the case of `->\`), and the execution of the method when the `memberstream` is invoked.

Control Flow

Control Flow makes a program tick. With it, sections of code can be skipped or repeated multiple times. Code can be executed in every repetition of a loop or every several repetitions. Complex decision trees can be created that execute code only under very specific conditions. Lasso supports a variety of constructs for performing conditional logic.

8.1 Conditional Constructs

Lasso offers two types of conditional constructs, one for general conditionals and another which trades flexibility for speed and readability.

8.1.1 If/Else Conditional

An *if/else conditional* is a construct that allows code to be executed only if a particular expression evaluates as "true". The if/else conditional differs from the conditional operator in that it permits multiple conditional tests as well as multiple expressions within the conditional bodies (the conditional operator allows only a single expression). The if/else conditional supports one default "else" which will execute if none of the conditional expressions are "true".

The if/else conditional can take two forms. The following example shows the first form. The "`// ...`" in the example shows where the body expressions for that particular condition would occur.

```
if(expression1)
    // Code here executed if expression1 evaluates to "true"
    // ...
else(expression2)
    // Code here executed if expression2 evaluates to "true"
    // ...
else
    // Code here executed if neither expression1 or expression2 evaluates to "true"
    // ...
/if
```

Each expression is evaluated in order, and the first value evaluating to "true" will have its corresponding conditional body executed. Once completed, no further conditions will be tested and execution will resume at the end of the if/else conditional.

The second form operates like the first, but permits the if/else to be used with the association/code block syntax.

```
if(expression1) => {
    // ...
else(expression2)
    // ...
else
    // ...
}
```

Either form is accepted. Although an if/else conditional produces no value, the first form does auto-collection, as will the second if associated with an auto-collect block (`=> {^ ... ^}`). See the *Captures* chapter for more information about these different types of code blocks.

There is also a shortcut syntax for the if/else conditional in the form `test ? expression1 | expression2`, where the first expression is run if the test is "true" and the second if the test is "false", described further in the section *Conditional Operator* of the *Operators* chapter.

8.1.2 Match/Case Conditional

A *match/case conditional* allows code to be selectively executed based upon the logical equivalence of two or more objects. The match/case conditional is given an initial test value and a series of case values and conditional bodies. The initial value is tested against each case value using the initial value's **onCompare** method. The first case value that matches the initial test value will have its conditional body executed. Each case can have more than one value to test against. If no case values match, then the default case, if present, has its conditional body executed. Using a match/case conditional when possible allows the compiler to perform optimizations not available with an if/else conditional, potentially leading to better performance.

Like the if/else conditional, a match/case conditional has two forms. The following example shows the first form with several case values and a default case:

```
match(expression)
case(c1, c2)
   // Code here executed if c1 or c2 matches expression
   // ...
case(c3)
   // Code here executed if c3 matches expression
   // ...
case
   // Code here executed if neither c1, c2, or c3 matches expression
   // ...
/match
```

The second form uses the association/code block syntax:

```
match(expression) => {
case(c1, c2)
   // ...
case(c3)
   // ...
case
   // ...
}
```

Either form is accepted. Although a match/case conditional produces no value, the first form does auto-collection, as will the second if associated with an auto-collect block (`=> {^ ... ^}`). See the *Captures* chapter for more information about these different types of code blocks.

8.2 Loop Constructs

Lasso offers several constructs that execute a body of code repeatedly, or *loop*, based upon some criteria. This criteria can be a boolean expression, a number counting to a predefined point, or the count of the number of elements in a composite object. Each method of looping supports skipping to the top of the next iteration, aborting the loop process entirely, and retrieving the current count of the number of loops that have occurred.

Each of these loop constructs support the two forms shown for if/else and match/case. Most examples are shown in both forms. Also, like if/else and match/case conditionals, loop constructs do not produce a value, but the first form does auto-collection, as will the second if associated with an auto-collect block (`=> {^ ... ^}`). See the *Captures* chapter for more information about these different types of code blocks.

8.2.1 While Loop

A *while loop* executes its body as long as its test expression is "true". The test expression is evaluated before the beginning of each loop.

```
// Form 1
while(expression)
    // Code here executes for as long as "expression" is true
    // ...
/while

// Form 2
while(expression) => {
    // ...
}
```

8.2.2 Counting Loop

A *counting loop* steps from one integer number to another, either counting up or down each iteration, until the counter reaches the end value. The most common usage of a counting loop is to give it a number indicating how many times it is to execute its body. Other usages involve giving the counting loop a specific starting number, a specific ending number, and an increment value by which the counter will be incremented for each iteration.

In the following example, the body will be executed 5 times:

```
// Form 1
loop(5)
    // Code here executed 5 times in a row
    // ...
/loop

// Form 2
loop(5) => {
    // ...
}
```

To specify the starting number, ending number, and increment, you can use one of the following two forms of the counting loop:

```
// Loop to 5 starting from -10 incrementing by 10
loop(5, -10, 10)
    // Code here executed each pass through the loop
    // ...
/loop

// Loop to 5 starting from -10 incrementing by 10
loop(-to=5, -from= -10, -by=10)
    // ...
/loop
```

In the case of using unnamed parameters, the order of the integers is significant. In the case of using keyword parameters, either the **-from** or **-by** may be omitted, and all keyword parameters may be supplied in any order.

8.2.3 Iterate Loop

An *iterate loop* is applied to objects that contain other objects, such as arrays, maps, or any type that supports **trait_forEach**. Iterate will execute the body once for each element contained in such an object. Iterate makes the individual elements available through the **loop_value** method. When iterating objects that store their elements associatively as keys and values, the key is also made available through the **loop_key** method.

The following example creates a staticarray and iterates its contents:

```
local(lv) = staticarray(2, 4, 6, 8, 10)

// Form 1
iterate(#lv)
    loop_value   // The current value from #lv
/iterate

// => 246810

// Form 2
iterate(#lv) => {
    // ...
}
```

8.2.4 Loop Methods

loop_abort()
> Can be used within the body of any of the loop constructs mentioned in this chapter. When called, the current loop construct will cease and execution will continue at the code following it.

loop_continue()
> Can be used within the body of a loop construct to cause the current loop to cease executing. Looping begins again at the top with the testing of the loop condition if present, and continues with the next iteration if applicable.

loop_count()
> All of the loop constructs keep track of the current loop number. The **loop_count** method can be called to retrieve this number. For while and iterate loops, the loop number always begins with "1" on the first loop and advances by "1" on each additional iteration. In a counting loop, the loop number begins with the loop's "from" value and advances either forward or backward depending on how the loop was constructed.

Note: *Query expressions* do not support **loop_abort**, **loop_continue**, or **loop_count**.

---

loop_key()
> When called within an iterate loop that's iterating a map, returns the key of the current map element. Returns "void" if the iterated object is any other type.

loop_value()
> When called within an iterate loop, returns the current element from the object being iterated. Returns the element's value if the iterated object is a map.

Captures

Captures are the basic frame of execution in Lasso. All code that executes does so within a capture. When a method is invoked, a capture is first automatically created for that method to execute in. When executing code in a source file, a capture is again automatically created for that code to execute in.

Captures are everywhere in Lasso, and learning how to use them will give you a powerful tool to use for solving some complex problems. This chapter provides in-depth information about captures and examples of their use.

9.1 Capture Structure

A *capture* is a representation of the control state of a section of code. While methods' code blocks are stateless (once they have had their code established), captures maintain state, some of which may change frequently during execution. This state consists of:

- The current method's code

- The current "self" and "inherited"

- The current "params" staticarray

- The current set of local variables, and their values

- The current *program counter*, or "PC". This value is the offset within that capture's code at which execution is currently happening.

- The name of the current method call

- The current *continuation*, which is the element to be executed after the current capture completes

- The set of handlers that must be executed before the capture completes

- A *home capture*, which is the capture in which this capture was created

When a capture is invoked, it will in turn execute its associated code which will execute within the context of that capture's state. The currently executing capture is known as the *current capture* and is made available through the **currentCapture** method. (See the **Operators** chapter for more information about **invocation**.)

9.2 Creating Captures

As previously mentioned, captures are automatically created when a method is executed. Captures can also be manually created by using curly braces as an expression. When using the association operator (**=>**) to invoke an object by passing it a capture, the capture is known as the object's *associated block* or *capture block*.

```
#ary->forEach => {
    // ... a capture of the surrounding code ...
}
```

In the code above, **forEach** is associated with a capture object. This results in **forEach** being invoked with the capture as its capture block, which it may execute as needed.

Captures can also be assigned to variables like any other object. The following example creates a capture and assigns it to the variable "cap":

```
local(cap) = { /* ... the capture's code ... */ }
```

There are two types of captures supported in Lasso: regular captures, like the examples above, and auto-collect captures. An auto-collect capture concatenates the result of calling the **asString** method on every value produced inside the capture when the capture is executed, and produces that value. The following example creates an auto-collect capture and assigns it to the variable "cap":

```
local(cap) = {^ /* ... the capture's code ... */ ^}
```

Because all executing code occurs within a capture, every capture that is manually created (as in the two examples above) is done so within the context of another capture. This surrounding capture is known as the new capture's *home capture*. Not all captures will have a home. Captures created automatically based on the invocation of a method will not have a home. A capture that is created within a capture that does have a home will have its home set to its parent capture's home. This means that nested captures will all have the same home.

A capture with a home will always take the following environment values from its home: self, locals, params, and current call name. A capture without a home will have state values based on the circumstances of the call. All other capture state is unique to each capture. As described below, the home capture is important for determining the behavior of **return** and **yield**.

9.3 Executing Captures

Captures are executed by calling their **invoke** method:

```
local(cap) = { /* ... the capture's code ... */ }
#cap->invoke  // Invoke the capture
#cap()        // Shorthand invocation
```

You can pass parameters to the **capture->invoke** method, and these are available with the special parameter local variables (**#1**, **#2**, etc.):

```
local(dist) = {
    local(x1) = #1
    local(y1) = #2
    local(x2) = #3
    local(y2) = #4
}
#dist(8, 2, 10, 5) // Sets #x1, #y1, #x2, #y2 to 8, 2, 10, 5, respectively
```

When you invoke an auto-collect capture, the auto-collected value will be returned and can be accessed using **capture->autoCollectBuffer**:

```
local(distance) = {^
    local(x1) = #1
    local(y1) = #2
    local(x2) = #3
    local(y2) = #4

    math_sqrt(math_pow(math_abs(#x2-#x1), 2) + math_pow(math_abs(#y2-#y1), 2))
^}
```

```
#distance(8, 2, 10, 5)
// => 3.605551

#distance->autoCollectBuffer
// => 3.605551
```

Stored captures can be executed at any point and the code contained within will operate as if it had been executed in the context in which it was created. This means that it will have access to the surrounding local variables where the capture was created even when the capture is being executed in code that has a different scope. The example below illustrates this by creating a capture in the **method1** method whose code is set to update the local variable "my_local" in **method1**. We then invoke that capture in "method2" which changes the value for "my_local" in **method1**. Returning "my_local" confirms that the value has been updated by **method2**.

```
define method1 => {
    local(my_local)
    local(my_cap) = {
        #my_local->append(#1)
    }

    #my_local = 'Hello'
    method2(#my_cap)

    return #my_local
}

define method2(cap::capture) => {
    #cap(', world.')
}

method1

// => Hello, world.
```

9.4 Producing Values and Detaching

Captures can produce values by using **yield** or **return**. Both **yield** and **return** halt the execution of any of the capture's remaining code and produce the specified value. Yielding from a capture differs from returning in how it leaves the capture. A **return** will reset the capture's PC to the top while a **yield** will not modify the PC. This affects how the capture behaves if it is executed a second time. A capture that has been returned from will begin executing from the start of the capture. A capture that has been yielded from will begin executing immediately after the expression that caused it to yield in the first place. A capture may yield many times.

```
local(cap) = {
    yield 1
    yield 2
    yield 3
    yield 4
}->detach

#cap()
// => 1
#cap()
// => 2
#cap()
// => 3
```

```
#cap()
// => 4
#cap()
// => 1    // Capture reached the end and reset
```

Note that once a capture reaches its end, the PC will automatically be reset back to the top. (Read on for a discussion of why we use **capture->detach** here.)

Even though a capture has yielded, it can still elect to return later in the code, thus resetting itself:

```
#cap = {
    yield  1
    yield  2
    return 3 // Subsequent calls will start from beginning
    yield  4 // This is unreachable
}
```

The current home capture is very important for determining the behavior of **return** and **yield**. Because captures are intended to execute as if they had been invoked directly within their home, **return** and **yield** will both behave by exiting from the current home as well as itself. This is known as a *non-local return*, and is illustrated in the following example which implements a potential **contains** method:

```
define contains(a::array, val) => {
    #a->forEach => {
        #val == #1 ?
            return true // This return is non-local
    }
    return false
}
```

Even though the **return true** occurs within a nested capture that is potentially several levels deep, it causes all intervening captures to halt their execution (with all their handlers executing in the process) up to and including the capture's home.

A capture can be detached from its home in order to escape from this behavior. The easiest way to accomplish this is to call the capture's **capture->detach** method. This method detaches the capture from its home and returns itself as the method's result. (This is what we did in the first **yield** example above.)

The following example creates a capture and detaches it from its home. Returning from within the capture no longer exits the surrounding capture.

```
local(cap) = {
    return self->type
}->detach

#cap()
// => // Produces result of self->type
```

Note that because the capture above is detached, it returns as normal and simply produces its value to the caller and allows the caller to continue its execution. It is not a non-local return.

Captures provide two other forms of **yield** and **return**: **yieldHome** and **returnHome**. These are only valid when the capture has a home and can be used to return from a capture *to* its home, instead of returning *from* its home. These special-purpose forms are used to accomplish some implementation details such as certain looping constructs or other control flow structures. (For example, **loop_continue** and **loop_abort** both rely on using these forms.)

9.5 Capture Methods

currentCapture()
> Returns a reference to the capture that is currently executing.

type **capture**
> A capture is a block of Lasso code that can be passed to another method or invoked locally. Captures are context-aware and retain state during execution.

capture->invoke(…)
> Executes the capture object and the code that is associated with it.

capture->detach()
> Detaches the capture so that it no longer has a home capture and then returns itself. After this, calling **capture->home** will return "void".

capture->restart()
> Resets the program counter (PC) for the capture and begins executing the capture's code again.

capture->continuation()
> Returns the capture that will be executed after this capture completes.

capture->home()
> Returns the home capture of the current capture object.

capture->callSite_file()
> Returns the file name where the capture object was defined.

capture->callSite_line()
> Returns the current line of code that is being executed in the capture object based on the file where the capture was defined.

capture->callSite_col()
> Returns the current column of code that is being executed in the capture object based on the file where the capture was defined.

capture->callStack()
> Returns the current call stack of the code that is being executed based on where the capture was called. Each line of the call stack consists of a line number, column number, and file name for the capture invocations leading up to the current one. The top of the stack has the most recent capture call and the list works its way back through each call.

capture->givenBlock()
> Returns the capture block associated with the current capture object, if any.

capture->autoCollectBuffer()
> If the capture is an auto-collect capture, then this will store the current auto-collect value created by invoking the capture.

capture->autoCollectBuffer=(*value*)
> If the capture is an auto-collect capture, this method allows for setting the auto-collect value.

capture->calledName()

capture->methodName()
> If the capture was created to run a method, this will return the method's name.

capture->invokeAutoCollect(…)
> This invokes the capture. If it is an auto-collect capture, it will return the auto-collect value, but it will not update **capture->autoCollectBuffer**.

Query Expressions

Query expressions allow the elements in arrays and other types of sequences to be easily iterated, filtered, and manipulated using a natural language syntax which is reminiscent of SQL.

A query expression can take each element in a sequence, manipulate it, and produce a new sequence. Query expressions let a developer drill down into nested sequences. For example, a query expression could iterate over each line in a block of text, then each word, and then each character; all in one expression. Query expressions provide a variety of useful operations, such as `order by`, `sum`, `average` and `group by`.

10.1 Query Expression Structure

Every query expression consists of three parts.

- The *with clause* specifies the variable name used to hold each element during evaluation, as well as the source of the data for the expression. One or more with clauses are required for every query expression. Multiple with clauses are used to dig down into nested sequences.

- A series of optional operations allow the elements to be filtered, sorted, skipped, etc. Operators include `where`, `let`, `skip`, `take`, `order by` and `group by`.

- An *action* tells Lasso what to do with the elements selected by the expression. Actions include `select`, `do`, `sum`, `average`, `min`, and `max`.

Whitespace, including line breaks, is insignificant within the clauses of a query expression. Syntactically, a query expression will begin with the word `with` and will end when terminated by an action.

Query expressions can be treated as objects. This means they can be assigned to variables and used repeatedly, and they can be passed as parameters. Unless otherwise noted, query expressions are evaluated in a lazy manner. This means that creating the query expression does not execute it. It is only when something else attempts to draw elements from the query expression that it begins to generate results.

All local variables available at the location of a query expression's creation are available within the query expression itself. However, new variables introduced by a query expression clause will not be available outside of the query expression that introduces them.

10.1.1 The With Clause

The with clause always begins with the word `with` followed by a variable name which is created as a local variable available only within the current query expression. Next follows the word `in` and then the source data element, which is any object whose type supports the `trait_queriable` trait, such as an **array** or a **list**. Note that when declaring the variable at the beginning of the with clause, the variable name is given by itself, without the "#" character, just as if the local were being defined using the standard `local` syntax.

```
with variable_name in source
```

Multiple subsequent with clauses can follow the first. When this occurs, the second `with` word can optionally be replaced by a comma. Multiple with clauses indicate a nesting of iterations. The following two example snippets are equivalent:

```
with variable_name in source
with another_name in #variable_name
```

```
with variable_name in source,
another_name in #variable_name
```

10.2 Actions

An *action* defines the result of a query expression. Actions permit a sequence to be transformed into a new sequence, or permit sequence elements to be used to compute an aggregate, or permit an arbitrary block of code to be executed for each resulting element.

10.2.1 Select

A *select* clause permits a new sequence to be generated based upon the source sequence. A select clause consists of the word `select` followed by a single expression. The expression is evaluated once for each element from the source sequence that makes its way through the query expression. The result of the select's expression will be an element going into the new sequence.

The following example computes the square of each element in the source array. The expression in the select clause performs the math to compute the square, the result of which becomes an element in the resulting sequence.

```
with n in array(1, 2, 3, 4, 5, 6, 7, 8, 9)
select #n * #n

// => 1, 4, 9, 16, 25, 36, 49, 64, 81
```

One query expression can be nested within another. In the next example, the query expression is assigned to a variable. That variable is used in a subsequent query expression. The first query expression is not evaluated until the second query expression is evaluated.

```
local(qe =
    with n in array(1, 2, 3, 4, 5, 6, 7, 8, 9)
    select #n * #n
)

with newN in #qe
select #newN * #newN

// => 1, 16, 81, 256, 625, 1296, 2401, 4096, 6561
```

10.2.2 Do

A *do* clause permits a block of code to be executed for each element that makes its way through the query expression. A do clause consists of the word **do** followed by either a single expression or a capture using either the regular curly brace form (`{ ... }`) or the auto-collect curly brace form (`{^ ... ^}`). If the code associated with a do clause consists of more than one expression, the code must be contained in a capture.

The following examples show how the query expression do clause can be used to manipulate the elements in the source array. Both query expressions operate identically.

```
local(ary) = array('the', 'quick', 'brown', 'fox', 'jumped', 'the', 'shark')

with n in #ary
do #n->upperCase

with n in #ary
do {
    #n->upperCase
}
```

It is important to note that when using **do** the query is immediately evaluated and that the query expression produces no result value. All other query expression actions are evaluated lazily, only as needed, and produce a result value dependent on the action in question.

The block of code given to a **do** remains attached to the surrounding method context, such that one could **return** or **yield** or access and create local variables.

10.2.3 Sum

A *sum* clause is useful when adding all of the resulting query expression elements together. A sum clause consists of the word **sum** followed by a single expression. The result of the expression will be the value used in the summation. The summation is performed using the **+** operator, so each element in the sequence must support the addition operator for the sum to succeed. The result of a query expression using a sum clause will be a single value.

The following example uses a sum clause to add together each element from the initial sequence:

```
with n in array(1, 2, 3, 4, 5, 6, 7, 8, 9)
sum #n

// => 45
```

10.2.4 Average

An *average* clause produces the average of each element that makes its way through the query expression. As expected, using **average** will take the sum of each element and then divide that value by the number of elements. As with **sum**, **average** produces a single result value.

```
with n in array(1, 2, 3, 4, 5, 6, 7, 8, 9)
average #n

// => 5
```

10.2.5 Min and Max

The *min* and *max* clauses produce the smallest or largest value from the sequence, respectively. The standard less than (**<**) and greater than (**>**) operators are used to find the result value.

```
with n in array(1, 2, 3, 4, 5, 6, 7, 8, 9)
min #n

// => 1

with n in array(1, 2, 3, 4, 5, 6, 7, 8, 9)
```

```
   max #n

   // => 9
```

10.3 Operations

In a query expression, an *operation* is an optional clause that affects how the query expression behaves by removing elements from the sequence, ordering the elements in a certain manner, or introducing new variables.

10.3.1 Where

A *where* operation lets elements be included or excluded from further consideration based upon a boolean expression. A where operation will generally run a test involving the current element. If the test expression results in "false", the element is discarded and the next element is selected and operated upon. If the test expression results in "true", the query expression proceeds with the next operation or action in the expression.

A where operation is composed of the word **where** followed by a single expression. The result of the expression should be boolean "true" or "false".

The following example performs a query expression using the numbers in an array. The where operation filters out all even numbers, leaving only odd numbers for the rest of the query expression. The local variable "n" holds each number in turn as the expression is evaluated.

```
with n in array(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
   where #n % 2 != 0 // Ignore even numbers
select #n

// => 1, 3, 5, 7, 9
```

Multiple where operations can be used in a query expression. Using multiple where operations is essentially the same as combining the expressions using the logical "and" operator (**&&** or **and**). The following two snippets are equivalent, while the third is not.

```
with n in array(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
   where #n % 2 != 0 // Ignore even numbers
   where #n % 3 != 0 // Ignore numbers evenly divisible by 3
select #n

// => 1, 5, 7


with n in array(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
   where #n % 2 != 0 && #n % 3 != 0
select #n

// => 1, 5, 7


with n in array(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
   where #n % 2 != 0 || #n % 3 != 0
select #n

// => 1, 2, 3, 4, 5, 7, 8, 9
```

10.3.2 Let

A *let* operation introduces a new variable into the query expression. Usually, this is done when evaluating an expression whose value will be used repeatedly further on throughout the query expression. For example, a let operation may evaluate an expression based upon the current iteration variable, assigning the result to a new variable, and then using both further within the query.

Variables introduced with a let operation have the same scope as those introduced in a with clause. That is, they only exist within the query expression.

A let operation consists of the word **let** followed by a new variable name, the assignment operator (**=**), and then an expression, the result of which will be assigned to the new variable.

The following example snippet assigns the square of the current iteration value to a new variable using a let operation:

```
with n in array(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
   let n2 = #n * #n
select #n2

// => 0, 1, 4, 9, 16, 25, 36, 49, 64, 81
```

The next example snippet uses both **where** and **let** together:

```
with n in array(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
   let n2 = #n * #n    // Square the current value
   where #n2 % 2 != 0  // Discard even values using the new variable
select #n2

// => 1, 9, 25, 49, 81
```

10.3.3 Skip

A *skip* operation permits a specified number of values from the source sequence to be skipped. A skip operation consists of the word **skip** followed by either a literal integer, or an expression that will evaluate to an integer.

The following example snippet skips the first 5 elements from the source array. Only the 6th element and beyond are sent to the remaining portion of the query expression.

```
with n in array(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
   skip 5
select #n

// => 5, 6, 7, 8, 9
```

10.3.4 Take

A *take* operation permits only a certain number of elements to be iterated upon. Elements beyond the specified value are ignored and not sent to the remainder of the query expression. A take operation consists of the word **take** followed by a literal integer or an expression that will evaluate to an integer.

The following example snippet takes only the first 5 elements from the data source. The remaining elements are ignored.

```
with n in array(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
   take 5
select #n
```

```
// => 0, 1, 2, 3, 4
```

The **skip** and **take** can be used together to limit which elements a query expression will operate over to a specific range. The order in which **skip** and **take** are specified is significant. (Generally, **skip** is specified before **take**, though this is not a requirement.)

The following example snippet skips the first 3 elements, takes only the next 4 and leaves the rest ignored. This results in only the numbers 3, 4, 5, and 6 for the rest of the query expression.

```
with n in array(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
    skip 3
    take 4
select #n

// => 3, 4, 5, 6
```

The next example snippets show how the ordering of **skip** and **take** is important. This first query expression takes only the first 4 elements of the series, though the first 3 of them are skipped. The second query produces the same result, but uses **skip** and **take** in the reverse order.

```
with n in array(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
    take 4
    skip 3
select #n

// => 3
```

```
with n in array(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
    skip 3
    take 1
select #n

// => 3
```

10.3.5 Order By

Query expressions permit the elements of a series to be ordered in an arbitrary manner by using an *order by* operation. This is done by using the words **order by** and then an expression, the result of which is used as the value by which the particular element will be ordered. This can be followed optionally by a direction indicator, which is the word **descending** or **ascending**. When a direction is not indicated, ascending order is assumed. Further ordering criteria can be specified by following the initial order by expression with a comma, and then the next ordering expression and optional direction indicator.

The following example orders the elements in the array using the default ascending order, and the next, in descending order:

```
with n in array(9, 2, 1, 3, 5, 4, 6, 7, 0, 8)
    order by #n
select #n

// => 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
```

```
with n in array(9, 2, 1, 3, 5, 4, 6, 7, 0, 8)
    order by #n descending
select #n

// => 9, 8, 7, 6, 5, 4, 3, 2, 1, 0
```

The expression provided to an order by can be any arbitrary expression. This permits elements to be ordered in any manner as desired by the developer. For example, a series of string objects could be ordered based upon their lengths, or elements could be randomly ordered based upon a random number generated for this purpose.

```
with n in array('the', 'quick', 'brown', 'fox', 'jumped', 'the', 'shark')
    order by #n->size
select #n

// => the, fox, the, quick, brown, shark, jumped


with n in array(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
    order by integer_random(0, 99)
select #n

// => 9, 8, 6, 5, 2, 1, 7, 0, 4, 3
```

In the next example snippet, a series of user objects, represented by their first and last names, could be ordered in an alphabetical manner:

```
with n in array('Krinn'='Jones', 'Ármarinn'='Hammershaimb',
        'Kjarni'='Jones', 'Halbjörg'='Skywalker',
        'Björg'='Riley', 'Hjörtur'='Hammershaimb')
    order by #n->second, #n->first
select #n

// => (Hjörtur = Hammershaimb), (Ármarinn = Hammershaimb), (Kjarni = Jones), \
//    (Krinn = Jones), (Björg = Riley), (Halbjörg = Skywalker)
```

10.3.6 Group By

A *group by* operation permits similar elements to be grouped together by a particular key expression and represented as a single object called a *queriable_grouping*. This new object can be further used throughout the query expression. A **queriable_grouping** object maintains a reference to each of the original elements within the group. It also possesses a **key** method which produces the value by which the particular elements were mutually grouped.

A group by consists of three elements: the object going into the group, the key by which the objects are grouped, and a new local variable name. This new variable name will be introduced into the query expression for further use and will be a **queriable_grouping** object. It has the following form:

```
group new_object_expression by key_expression into new_local_name
```

A group by operation makes the most sense when used with other operations and actions. The following example takes a series of users, represented by a pair with their last and first name, and performs a query expression over them.

```
with n in array('Jones'='Krinn', 'Hammershaimb'='Ármarinn',
        'Jones'='Kjarni', 'Skywalker'='Halbjörg',
        'Riley'='Björg', 'Hammershaimb'='Hjörtur')
    let swapped = pair(#n->second, #n->first)
    group #swapped by #n->first into g
    let key = #g->key
    order by #key
select pair(#key, #g)

// => // Line breaks added for readability
// (Hammershaimb = (Ármarinn = Hammershaimb), (Hjörtur = Hammershaimb)),
// (Jones = (Krinn = Jones), (Kjarni = Jones)),
```

```
// (Riley = (Björg = Riley)),
// (Skywalker = (Halbjörg = Skywalker))
```

The example above example breaks down into six steps:

1. Begin the query expression using "n" as the variable to hold each initial element from the source array. There are six elements in the source array, so "n" will start off pointing to the first element. Once the query expression completes its first iteration, "n" will point to the second element and the query will perform another iteration, and so on, until the end of the array is reached.

   ```
   with n in array('Jones'='Krinn', 'Hammershaimb'='Ármarinn',
           'Jones'='Kjarni', 'Skywalker'='Halbjörg', 'Riley'='Björg',
           'Hammershaimb'='Hjörtur')
   ```

2. Create a new pair containing the swapped last and first names. Name this "swapped".

   ```
   let swapped = pair(#n->second, #n->first)
   ```

3. Group each of the new user pairs by last name: **#n->first** is used as the key as it still contains the original last name. From this point forward, no previously introduced variables are available. Only "g" exists now. It will contain each **queriable_grouping** object generated by the group by operation at this step.

   ```
   group #swapped by #n->first into g
   ```

4. Access the grouping key for the current value of "g". Save it into "key".

   ```
   let key = #g->key
   ```

5. Sort the resulting grouping objects by "key", which contains the last name, using **order by**. Therefore, all of the resulting group objects will come out of the query expression ordered alphabetically by last name.

   ```
   order by #key
   ```

6. Finally, create a new pair containing "key" and the grouping object and select that, making the new pair one of the new elements in the result of the query expression.

   ```
   select pair(#key, #g)
   ```

The result of the example query expression looks as follows. Notice how the results for **'Hammershaimb'** and **'Jones'** each contain both of the users in those groups.

```
// => // Line breaks added for readability
// (Hammershaimb = (Ármarinn = Hammershaimb), (Hjörtur = Hammershaimb)),
// (Jones = (Krinn = Jones), (Kjarni = Jones)),
// (Riley = (Björg = Riley)),
// (Skywalker = (Halbjörg = Skywalker))
```

10.4 GenerateSeries Type

The **generateSeries** method generates a series of integer values, and is great for use in query expression **with** clauses.

type **generateSeries**

generateSeries(*from*, *to*, *by=1*)
> This method creates an integer series. The first parameter specifies the first number in the series. The second parameter specifies the maximum value of the last number in the series, and an optional third parameter specifies the step to use

for going through the series (defaults to 1). Note that the second parameter will not be included in the series if the step value causes it to be skipped.

The following example uses a query expression to sum the even numbers starting with 2 and ending with 10:

```
// Note that 11 is not part of the generated series
with num in generateSeries(2, 11, 2)
sum #num

// => 30
```

There is also a **generateSeries** literal syntax that can be used. The following is equivalent to the preceding example:

```
with num in 2 to 11 by 2
sum #num

// => 30
```

A **generateSeries** object can also be converted to a staticarray for later use.

```
generateSeries(2, 11, 2)->asStaticArray
// => staticarray(2, 4, 6, 8, 10)
```

10.5 Making an Object Queriable

An object can be used as the source of a with clause in a query expression if its type has implemented and imported the **trait_queriable** trait. For this, a type must implement the **forEach** member method. This method is always called with a capture block. Within the **forEach** member method, the object being queried should invoke the capture block, passing it each available element in turn.

The following example implements a user list type. Objects of this type can be used in query expressions. For the sake of this example, it permits iteration over a fixed list of users, which it provides to the query one by one.

```
// Define the user_list type
define user_list => type {
    trait { import trait_queriable }

    public forEach() => {
        local(gb) = givenBlock

        // Provide the 6 users one at a time
        #gb->invoke('Krinn'='Jones')
        #gb->invoke('Ármarinn'='Hammershaimb')
        #gb->invoke('Kjarni'='Jones')
        #gb->invoke('Halbjörg'='Skywalker')
        #gb->invoke('Björg'='Riley')
        #gb->invoke('Hjörtur'='Hammershaimb')

    }
}

// Create a user_list object
local(ul) = user_list

// Use it in a query
with user in #ul
select #user->first
```

```
// => Krinn, Ármarinn, Kjarni, Halbjörg, Björg, Hjörtur
```

Types with one or more iterator methods can be used in a query expression by exposing each iterator with an *eacher*, which is a method that takes an escaped iterator method and an optional set of initial parameters, and uses the **eacher** method to return a generator for the iterator.

For example, while a string cannot be iterated upon directly, it has an iterator **string->forEachCharacter**, which is implemented as an eacher below:

```
define string->eachCharacter()::trait_forEach => eacher(self->\forEachCharacter)
```

A string can then run a query expression on each character by using **string->eachCharacter**:

```
with i in 'Hammershaimb'->eachCharacter
select #i

// => H, a, m, m, e, r, s, h, a, i, m, b
```

Methods

Methods are the fundamental process abstraction in many languages, including Lasso. *Methods* provide a means for encapsulating a series of expressions so that they can be called repeatedly as a group. Complex, multi-step tasks are best expressed as a group of related methods. A method is defined under a specific name and is associated with a signature and a code block.

11.1 Signatures

Before method definitions can be understood, it is important to understand signatures. A *signature* is a description of a method and includes its name, parameter names, and types, and the method's return value type. Signatures are used when defining methods, and simplified signatures are used when defining types and traits. This chapter will concentrate on signatures for defining methods only.

Method names in Lasso consist of letters, numbers, and underscores. A method name must start with a letter, or one or two leading underscores followed by a letter. Letter case is not considered when comparing method names.

Method names beginning with an underscore are generally intended to only be used internally, as they represent methods that could change in the future and are therefore considered unstable.

Some valid examples of method names are shown below:

```
field
_date_msec
Encode_Base64
String_ReplaceRegexp
```

There are several other characters that are valid in specific circumstances. The mathematical operators **+**, **-**, *****, **/**, and **%** are used in method names when supplying implementations for these operations for types. See the section *Operator Overloading* in the *Types* chapter for more information.

Most signatures consist of a method name followed by parentheses which surround a list of parameters for the method, and an optional return type.

A signature for the **loop** method is shown below. The parameter list includes three parameters: a **to** parameter which must be an integer, and two more integer parameters each with default values of "1".

```
loop(to::integer, from::integer=1, by::integer=1)
```

When a method is called, the parameter names given in the method's signature become the variable names for those parameters within the method's body. The **loop** method above would have access to the local variables "to", "from", and "by".

A signature's parameter list allows the specification of required and optional parameters, type constraints, keyword parameters, and rest parameters.

11.1.1 Empty Signature

A signature specified with an empty set of parentheses indicates that the method will not require or accept any parameters. Giving parameters to a method defined with an empty signature will result in a failure. The following is an example of a signature without any parameters:

```
method_name()
```

11.1.2 Rest-Only Signature

A signature whose parameter list is just three periods (`...`) indicates that the method will accept any number and type of parameters. A method defined with a rest-only signature can be called with no parameters or any combination of values and keyword parameters.

```
method_name(...)
```

Note that these are three periods rather than a Unicode ellipsis character.

Within the method, the parameters that were passed in can be accessed through a local variable named "rest". If any parameters were given, the rest variable will be a staticarray. If no parameters are given, it will remain "void". This signature can be useful for methods that have highly variable parameters needing special processing, such as **inline**.

11.1.3 Required Parameters

Required parameters can be specified within a signature by naming them in order. All required parameters must be listed before any other parameter types. When calling a method with required parameters, all parameter values must be provided in the proper order according to the method's signature.

The name of each required parameter must be a valid variable name. Each name should begin with a letter or an underscore followed by a letter, then zero or more letters, numbers, underscores, or period characters.

The following signature defines two required parameters named **firstname** and **lastname**. Within the method these parameters can be accessed through the local variables "firstname" and "lastname".

```
method_name(firstname, lastname)
```

When calling this method, both parameters must be given in order.

```
method_name('Henry', 'Gibbons')
```

The parameter names are only used within the method so the choice of parameter names need only make sense to the implementer of the method. However, the parameter names may be used in documentation or reported in error messages so they should be made descriptive when possible. Knowing a method requires the parameter **firstname** is more descriptive than a method that requires the parameter **fn**.

11.1.4 Optional Parameters

Optional parameters are those which are listed in a method's signature but are not required to be given values when the method is called. Optional parameters are specified within a signature by providing default values along with the parameter names. A default value is specified after a parameter name by using an equal sign (`=`) followed by an expression. The expression's value will be used to assign the default value to the parameter's local variable should that value be omitted by the caller.

The default value expression will be evaluated independently with each call as required from within the associated method's body, so any state valid at the beginning of the associated method is valid during the evaluation of all optional parameter default values.

Although optional parameters may be omitted when calling a method, when optional parameter values are provided, they must be provided in order. That is, when the method is called, once an optional parameter is omitted, all subsequent optional parameters must also be omitted.

The parameters in the following signature are both optional. If the **host** parameter is not specified the local variable "host" within the method will have the default value **'localhost'**. If the **port** parameter is not specified, it will have the default value of "80".

```
connect(host='localhost', port=80)
```

When the method is called the parameters that are passed to it will be assigned to each of the optional parameters in turn. The method called as **connect('www.lassosoft.com')** will have a default port value of "80". The method called as **connect()** will have both default values. And, the method called as **connect('www.lassosoft.com',443)** will use the specified values, overriding both defaults. In this example, there is no way to only specify a value for **port**.

Mixing Required and Optional Parameters

When calling a method that accepts both required and optional parameters, all required parameter values must be specified before any optional parameter values. The values that are passed will be assigned to the required parameters first. While there are sufficient remaining values, the optional parameters will be assigned in order.

For example, the following signature has one required parameter **host** and two optional parameters **port** and **timeout**:

```
connect(host, port=80, timeout=15)
```

The **host** parameter must be provided before **port** can be provided with a value, and both **host** and **port** must be provided before **timeout** can be provided with a value.

11.1.5 Keyword Parameters

Keyword parameters are named parameters that can be specified in any order. When keyword parameter values are passed to a method, they are given with the associated parameter name, using the following syntax:

```
-parameterName = expression
```

If a method has any required or optional parameters, they must be specified before the keyword parameters in both the method signature and when calling the method.

Keyword parameters are specified by preceding the parameter name with a hyphen (**-**). Within the method body, the keyword parameter's associated local variable will not have the hyphen.

Keyword parameters can be either required or optional. Optional keyword parameters are indicated in the same manner as regular optional parameters, by following the parameter name with an equals (**=**) and a default value expression.

For example, a hypothetical **find_in_string** method could have the following signature. The required input is followed by two keyword parameters: the required **-find** and the optional **-ignoreCase**.

```
find_in_string(input, -find::string, -ignoreCase::boolean=false)
```

When this method is called the input must always be given first. However, the two keyword parameters can be given in either order, provided they follow all non-keyword parameters. It is valid to call the method in any of the following ways:

```
find_in_string('the fox', -find='x', -ignoreCase=true)
find_in_string('the fox', -ignoreCase=true, -find='x')
find_in_string('the fox', -find='x')
```

Within the method's body, three predefined local variables will be created for these parameters including **input**, **find**, and **-ignoreCase**.

Note that calling the method as **find_in_string('the fox')** will generate a failure because the **-find** keyword parameter is required (since it has no default value). Calling the method as **find_in_string(-find='x','the fox')** will also generate a failure because the input is being specified after a keyword parameter. All required parameters and any optional parameters being passed must be specified before the first keyword parameter.

Boolean Keyword Parameters

Often, keyword parameters will be used to indicate simple boolean values. For example, as a set of options or flags given to a method to control the details of its behavior. When calling a method, a keyword parameter can be passed without an associated value. Doing so is implicitly the same as passing a boolean "true" value for that parameter. Boolean keyword parameters are normally specified with a default value of "false" so if the keyword parameter is not specified the predefined variable will have a value of "false".

The following signature defines the method **server_date** as accepting either a **-short** keyword parameter, a **-long** keyword parameter, or neither:

```
server_date(-short=false, -long=false)
```

If the method is called as **server_date(-short)** then the predefined local variable "short" will have a value of "true" and the predefined local variable "long" will have a value of "false". If the method is called as **server_date()** then both variables will have a value of "false".

11.1.6 Rest Parameters

The list of parameters may end with three periods (**...**) in order to specify that the method should accept a variable number of additional parameters after any specified required and optional parameters. The additional parameters are known as *rest parameters*. When the method is called, any additional parameters are placed into a predefined local variable named "rest". If there are no rest parameters, the "rest" local will be "void"; otherwise, it will be a staticarray holding the remaining parameter values passed to the method.

The signature below specifies that the **string_concatenate** method requires one parameter named **value**, but will accept any number of additional parameters. Within the method, the first parameter will be placed into the predefined local variable "value", and the remaining parameters, if any, will be placed into the predefined local variable "rest":

```
string_concatenate(value, ...)
```

Note that these are three periods rather than a Unicode ellipsis character.

By default, the rest parameter local variable is always named "rest", but an alternate variable name can be specified in the signature by placing the desired name immediately after the three periods. The following signature would rename the rest variable to "other":

```
string_concatenate(value, ...other)
```

11.1.7 Parameter Type Constraints

In a signature, all parameter types, with the exception of the rest parameter, can be specified with an optional type constraint. While parameter count and ordering ensure that the caller is passing the right number of parameters in the right order, type constraints ensure that the parameter values are of the right type. For example, if a method that expects to receive two string parameters is given two integers, it is being used incorrectly. If a caller passes a parameter value that does not fit the type constraint set for that parameter, then a failure will be generated. Any type or trait name can be used as a constraint, and all parameter values must pass the "isA" test for their constraint before the method body begins to execute. (The "isA" test involves calling the object's **isA** method with the constraint; if a non-zero value is returned, then it passes. See **isA** for details about this member method.) Additionally, all parameter default values must produce results of a type matching the type constraint set for their respective parameters.

A type constraint is specified by following the parameter name with two colons (**::**) and a type name. Whitespace is permitted on either side of the double colon (examples herein will not include whitespace). The signature below has both of its required parameters constrained to only accept values that are of type **string**.

```
method_name(firstname::string, lastname::string)
```

If the parameter has a default value, it should be placed after the type constraint.

```
method_name(firstname::string, lastname::string = '')
```

A parameter with no type constraint will accept any type of value. Constrained and unconstrained parameters can be mixed.

```
method_name(firstname::string, lastname)
method_name(firstname, lastname::string)
method_name(firstname::string, lastname::string, -age::decimal=0.0, -dept='')
```

Within a method body, parameters with type constraints translate into local variables with type constraints. A parameter that is constrained to accept a particular object type becomes a local variable that can hold only that type of object. See the *Variables* chapter for more information on *type-constrained variables*.

11.1.8 Return Type

Specifying a return type for a signature enforces that the value returned by its code block is of a specific type. If a method returns a value having a type that does not pass the "isA" test for the specified return type, then a failure is generated. (The "isA" test involves calling the object's **isA** method with the constraint; if a non-zero value is returned, then it passes. See **isA** for details about this member method.) Specifying a return type provides knowledge to the caller of the method about the method's resulting value. It also ensures the method's developer that their programming is correct, at least with respect to the method returning the proper value type. Specifying a return type is optional, and a method without a specified return type may return values of any type, or may return no value at all (in which case the value returned to the caller is "void").

The return type for a signature is specified at the end of the signature, following the parameter list parentheses, by including two colons (**::**) and a type or trait name.

The following signature specifies that the method will always return a value of type **string**.

```
string_concatenate(value, ...other)::string
```

11.1.9 Type Binding

Signatures are also used to denote that the method belongs to a particular type. This is referred to as the *type binding* for the signature. A signature with no bound type is referred to as being *unbound*. All example signatures given up to this point were unbound signatures. A type binding occurs at the beginning of the signature, before the signature's name. It consists of a type name followed by the target operator (**->**) followed by the rest of the signature.

```
type_name->method_name(...)
method_name(...)
```

In the above example, the first signature is bound to the type **type_name** while the second signature is unbound. A method using the first signature cannot be called except with a target instance of **type_name**. The second signature can be called at any point without a target type instance.

11.1.10 Signature Syntax

These are the syntax diagrams for signatures and their related elements.
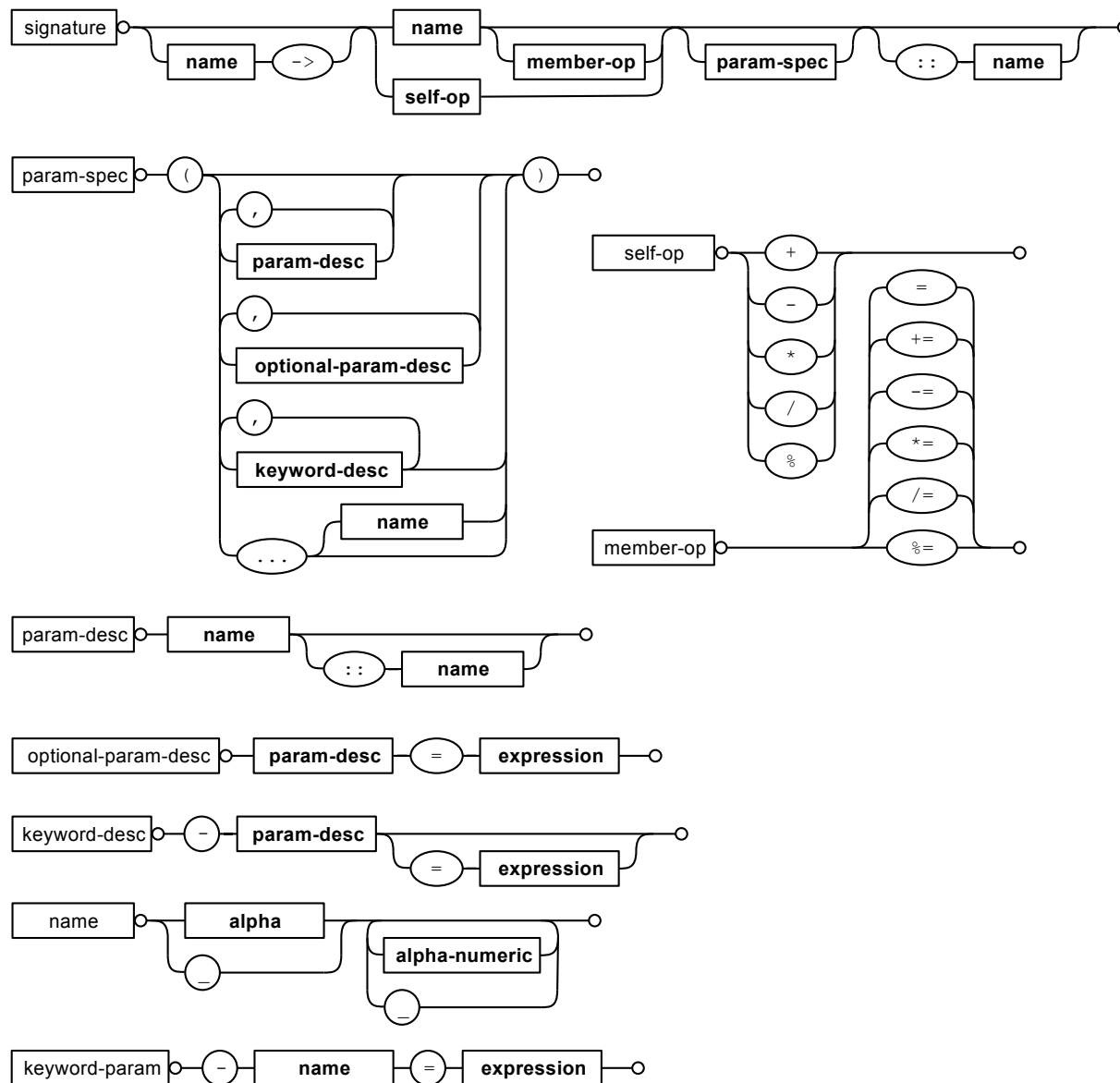
11.2 Defining Methods

Before a method can be used, it must first be defined. Defining a method combines a signature with a method body, and allows it to be called by name from within other methods.

The **define** keyword is used to define new methods, types, and traits. When defining a method, the word **define** is followed by a signature, the association operator (**=>**), and then an expression that provides the body for the new method.

```
define signature => expression
```

If a method is defined that has a signature equivalent to an already-defined method, the new definition will replace the old and the old definition will no longer be available. Keyword parameters cannot be used to uniquely identify a method. A method taking, for example, two required parameters and a certain set of keyword parameters will be overwritten by a new method that requires the same two parameters and an entirely different set of keyword parameters.

11.2.1 Methods Returning Simple Expressions

A simple method definition is shown below. The signature **hello()** describes what and how the method will be called, in this case **hello** with no parameters. After the association operator, the expression **'Hello,world!'** provides the method's return value. The method below simply returns a string:

```
define hello() => 'Hello, world!'
```

Any single expression, including the ternary conditional operator or mathematical expressions, can be used as the method's return value. Assignments, local or thread variable declarations, or any other expression known at compilation time not to produce a value may not be used as a method's return value expression.

```
define pi() => math_acos(-1)
define times_twenty(n) => #n * 20
define is_blank(s::string) => #s->size ? false | true
```

11.2.2 Code Blocks

Many methods do more than return a single easily calculated value. A method body can be composed of multiple expressions enclosed by a pair of curly braces (**{ ... }**). This type of method body is referred to as a *code block*.

Code blocks provide the most flexibility when defining methods. They allow a series of expressions to be encapsulated as the implementation of the method. One or more **return** statements may be used to end execution of the method body and to optionally return a value to the caller.

The methods used as examples above may be written using code blocks as follows:

```
define pi() => {
    return math_acos(-1)
}
define times_twenty(n) => {
    return #n * 20
}
define is_blank(s::string) => {
    return #s->size ? false | true
}
```

The expressions within a code block method body are generally formatted so that they each appear on a separate line. Some expressions are terminated by an end-of-line, and expressions may be explicitly terminated by using a semicolon at the end of the expression.

The following definition for the hypothetical **strings_combine** method uses a series of instructions within the method body to generate the return value for the method:

```
define strings_combine(value::string, with, alsoWith='') => {
    local(result) = string(#value)
    #result->append(#with->asString)
    #result->append(#alsoWith->asString)
    return #result
}
```

11.2.3 frozen Methods

To prevent a method's definition from being modified, the **frozen** keyword can be used. When inserted after the association operator, it prevents the method from being added to with multiple dispatch or overridden.

```
define phi => frozen (1 + math_sqrt(5)) / 2
define phi => 500
phi
// => 1.618034
```

11.2.4 define Syntax

This is the syntax diagram for **define**.
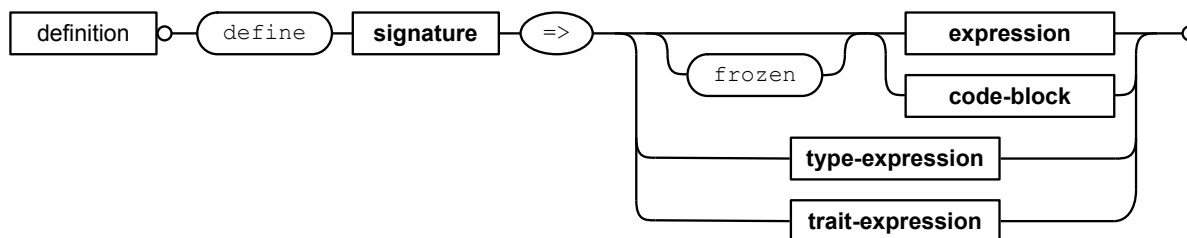
11.3 Multiple Dispatch

Multiple dispatch is a technique that permits more than one method body and signature to be defined under a given method name. The various signatures will differ in the number or types of parameters they are stated to receive. When the method name is called, the parameters given by the caller (or the lack thereof) will determine which method body will actually be executed. The process of determining which method body to call is referred to as "dispatch".

11.3.1 The Dispatch Process

The process of method dispatch first involves taking the name the caller has used and matching it to one or more methods defined under that name. These methods are the set of methods potentially valid for that call. Methods are removed from this set as each parameter value is checked against each valid method's type constraint for that parameter. If the parameter value is acceptable according to this constraint (a lack of a type constraint on a parameter means that any type is valid for that position), then the method remains in the set of valid methods, otherwise it is removed. For each parameter position, methods that accept, at most, fewer than that number of parameters are also removed from the valid set.

In many cases, when the final parameter value is checked there will remain only one valid method. In cases where there are multiple remaining valid methods, the methods are sorted and the top-most method is selected as the method to be executed for that call. The methods are sorted according to how closely related each given parameter value is to each method's stated type constraint for that parameter position, with each subsequent parameter having a lower priority than the previous.

- Methods with a type constraint for a parameter position will sort higher than methods that do not have a type constraint.

- Methods having a required parameter for a position will sort higher than methods with an optional parameter.

- Methods that are valid only because they accept rest parameters will sort lower than methods that accept an actual declared parameter.

In the case where the result of the sorting leads to two or more equally valid methods, then the call is ambiguous and a failure will be generated. In practice, ambiguous methods are usually handled when the conflicting method is first defined, leading to the second definition overwriting the first, which removes the first from future consideration during dispatch.

Keyword parameters are never considered during the method selection process until the end where the single remaining method's keyword parameters (if any) are validated. Two methods cannot differentiate themselves based on accepting a different set of keyword parameters. Methods must be distinguished based solely on their required or optional parameters.

11.3.2 Using Multiple Dispatch

Constraints Example

Multiple dispatch comes into play any time more than one method is defined under a single name. As example, consider the scenario where special diagnostic information needs to be created for a variety of possible types: **array**, **string**, **bytes** and a default **any** type. In the example below, the **log_object** method is defined multiple times, each accepting a different possible type. Each of the four methods is written to handle only their input value types.

```
define log_object(a::array) => {
   return '[log] array with ' + #a->size + ' elements\n'
}
define log_object(s::string) => {
   return '[log] string with value "' + #s + '"\n'
}
define log_object(b::bytes) => {
   return '[log] bytes with hex value 0x' + #b->encodeHex + '\n'
}
define log_object(any) => {
   return '[log] unhandled object type: ' + #any->type + '\n'
}
log_object('Hello!')
log_object(bytes('ABCD'))
log_object(array(1, 2, 3, 4, 5))
log_object(pair(1, 2))

// =>
// [log] string with value "Hello!"
// [log] bytes with hex value 0x41424344
// [log] array with 5 elements
// [log] unhandled object type: pair
```

Multiple dispatch allows several related methods to be grouped under a single name. This permits method bodies to be more succinct and tailored directly to the input types. This promotes maintainability in a code base, as shorter methods are easier to understand and maintain.

If the above example was instead written to have a single **log_object** method that accepted any value type (we'll call it a mega-method), and within that mega-method, inspected the parameter value type to decide what action to take, then the

method would need to be modified each time a new log object type was added. If a log implementation needed to be added for objects of type **pair**, then a new case would need to be placed within that mega-method.

Problems arise if a user wishes to add logging implementations for their own object types. If **log_object** were only this single mega-method, then the user would likely have to resort to writing their own set of log methods, falling back to using **log_object** only for object types that it is known to handle. However, with multiple dispatch, the user may directly add their own **log_object** method with its own unique signature. The new method is incorporated automatically into the system and none of the other methods need to be modified.

```
define log_object(p::pair) => {
    return '[log] pair with: ' + #p->first + ', ' + #p->second + '\n'
}
log_object('Hello!')
log_object(bytes('ABCD'))
log_object(array(1, 2, 3, 4, 5))
log_object(pair(1, 2))

// =>
// [log] string with value "Hello!"
// [log] bytes with hex value 0x41424344
// [log] array with 5 elements
// [log] pair with: 1, 2
```

Number of Parameters Example

The number of parameters that a set of methods accepts can be used to determine method dispatch. For example, one method may require a single parameter while a second method requires two parameters, such as in the example that follows:

```
define log_object(a::array) => {
    return '[log] array with ' + #a->size + ' elements'
}
define log_object(a::array, extra::boolean) => {
    local(result) = log_object(#a)
    #extra ?
        return #result + '. Elements: ' + #a->join(', ')
    return #result
}

log_object(array(1, 2, 3, 4, 5))
// => [log] array with 5 elements

log_object(array(1, 2, 3, 4, 5), true)
// => [log] array with 5 elements. Elements: 1, 2, 3, 4, 5
```

Note how the body of the second method calls the first method to get the initial result string before augmenting it and returning that value.

Types

Types are the fundamental data abstraction concept in Lasso. Since Lasso is an object-oriented language, every piece of data is an object and every object is of a particular type. A *type* is a predefined layout of data combined with a particular set of methods. Types provide a means for encapsulating data with the collection of methods designed to modify *objects* representing that data in predetermined ways.

12.1 Defining Types

Before a type can be used, it must first be defined. Defining a type is done in the same manner as other entities (traits, methods). The word **define** is used, followed by the name for the type, the association operator (**=>**), and a type expression that provides the description of the type's methods and data members.

```
define typeName => type expression
```

12.1.1 Type Expressions

A *type expression* consists of the word **type** followed by a set of curly braces (**{ ... }**). Between those curly braces reside a series of sections; each describing a different aspect of the type. These sections include: "parent"; "data"; "trait"; and "public", "protected", and "private" member methods. Each section begins with one of those words and ends at the beginning of the next section or the end of the type expression (which would be a close curly brace). Each section is optional. Sections can occur in any order. The sections "trait" and "parent" can occur only once.

The most simple type definition is shown below. It defines a type named "person" and contains no sections. Therefore, the **person** type contains no methods or data members of its own. It is a completely valid, if somewhat useless, type.

```
define person => type { }
```

12.1.2 Data Members

Each data section defines one or more *data members* for the type, which are other objects contained by the type. In a data member section, the word **data** is followed by one or more data member names. Data member names follow the same rules as variable and method names. They can begin with an underscore or the characters A–Z and then can be followed by zero or more underscores, letters, numbers, or period characters. Character case is irrelevant for data member names.

Like variables, data members store values. Three values are unique to each instance of the type. If a person type was created then it could contain data members for the first and last name of the person, his/her birthdate, social security number, address, etc. Just as every individual has his own values for these items, so would every instantiated object.

The following example type implementation shows several different methods for defining data members. These methods can be mixed and matched in a single type to provide the best readability. Data sections can also be interspersed with the other sections in the type expression if necessary.

```
define person => type {
   data firstName, lastName
   data age
   data
      birthdate
   data ssn
   data address1, address2, city,
      state, zip, country
}
```

Type Constraints

Data member values can be constrained to hold only particular types of objects. To do this, follow the data member name with two colons (`::`) and then a type or trait name. When a data member is constrained, it cannot be assigned any value that does not fit the constraint. The following type constrains "firstName" and "lastName" to be string objects and "age" to be an integer value:

```
define person => type {
   data firstName::string, lastName::string
   data age::integer
}
```

Default Values

Data members can be given default values. When a type instance is first created, before it can be otherwise used, its data members are assigned their default values. A default value can be any single expression. The following type definition uses both type constraints and default values for "firstName" and "lastName", but just a default value for "age":

```
define person => type {
   data firstName::string = '', lastName::string = ''
   data age = 0
}
```

Accessing Data Members

Data members can be accessed from within the methods of a type by targeting the current type instance using the keyword **self** and the target operator (`->`) followed by the name of the data member between single quotes. The following expression would set the value of the data member "age" to "36":

```
self->'age' = 36
```

The following expression produces the value of the "age" data member:

```
self->'age'
// => 36
```

Equivalently, Lasso supports a shortcut syntax for targeting "self" by using a single period. The examples above could be rewritten using a period in place of **self->**.

```
.'age' = 36
```

```
.'age'
// => 36
```

All of the data members in a type are private. This means that a data member can *only* be directly accessed using either of the above syntaxes; only when "self" is the target object. Optionally, data members can be exposed to the outside world. The following section describes how getters and setters can be used to access data member values from outside of the owning type.

12.1.3 Getters and Setters

A *getter* is a member method that produces the value of a data member, while a *setter* is a member method that permits the value of a data member to be assigned. If the value of a data member should be accessible from outside of the owning type, then it is necessary to create a getter and/or a setter method for that data member.

If the word `public`, `protected`, or `private` is given in front of a data member name, Lasso will automatically create a getter method and a setter method with the appropriate access level as described in the section on *member methods*. The following code defines three publicly accessible data members:

```
define person => type {
    data public firstName, public lastName
    data public age::integer=0
}
```

The automatically created getter method has the same name as the data member. Parentheses are optional after the getter (as they are with all methods accepting zero parameters). The current value for the data member can be returned as follows:

```
#person->firstName
// => // Produces the value stored in the "firstName" data member


#person->lastName()
// => // Produces the value stored in the "lastName" data member
```

The automatically created setter permits the assignment (`=`) or the assign-produce (`:=`) operators to assign a new value to the data member. As with the getter, parentheses are optional.

```
// Sets "firstName" to a new value
#person->firstName = 'John'

// Sets "lastName" to a new value
#person->lastName() := 'Doe'
// => Doe
```

Exposing a data member in this manner always creates both the getter and setter. However, getters and setters can also be added manually without automatically exposing both get and set behaviors. One hypothetical use for this is a type that wants to provide to the outside world read-only access to one of its data members. Additionally, a getter or a setter can be added manually in order to override or replace the automatically provided behavior; perhaps to validate the values in a particular manner.

The following example defines a `person` type that manually exposes its "firstName" data member by defining two member methods, one for the getter and another for the setter. (See the section on *member methods* for more information on creating member methods.)

```
define person => type {

    // The firstName data member
    data firstName

    // The firstName getter
    public firstName() => {
        return .'firstName'
```

```
    }

    // The firstName setter
    public firstName=(value) => {
        .'firstName' = #value
    }
}
```

The type definition above would operate identically if it instead omitted the manual getter and setter methods and made its "firstName" data member public.

Implementing getter and setter methods for a data member allows **assignment operators** to be used with it. For example, since the **+**, **-**, and ***** operators are implemented for the string type (see the section on *Operator Overloading* below), they can be used to modify the "firstName" data member:

```
local(someone = person)
#someone->firstName = "Bob"
#someone->firstName += "by"    // Bobby
#someone->firstName -= "y"     // Bobb
#someone->firstName *= 2       // BobbBobb
```

Setters can be defined to accept more than one parameter. When called, the additional parameters are given in the method call's parentheses, just as with a regular method. When defining such a setter method, the first parameter is always the new value for the assignent. All additional parameters follow. For example, with a "firstName" setter that includes an optional nickname:

```
public firstName=(value, nick) => {
    .'firstName' = `"` + #nick + `" ` + #value
}
```

it would be called like this:

```
#someone->firstName("Big Wheels") = "Bob"    // "Big Wheels" Bob
```

For another multi-parameter setter example, see **security_registry->userComment=**.

Within a manual getter or setter, it is vital to refer to the data member using the single-quoted name syntax. Otherwise, an infinite recursion situation may arise as the getter/setter continually re-calls itself.

12.1.4 Member Methods

A *member method* is a method that belongs to a particular type, as opposed to an *unbound method* which does not, thus acting as a standalone function. A member method can operate on the data members of its owning type in addition to any parameters the method may receive.

Member methods are created in sections of a type expression beginning with the word **public**, **private**, or **protected**, followed by a method signature, the association operator (**=>**), and the implementation of the method. Each section can define one or more methods separated by commas. The choice of word used to begin a member methods section influences how the methods are permitted to be accessed. There are three such access levels.

public
> Public member methods can be called without any restrictions. They represent the public interface of the type. When the type is documented for others to use, only the public methods are described.

private
> Private member methods can only be called from methods defined within the owning type. Private methods are to be used for low-level implementation details that shouldn't be exposed to the end user or to inheriting types.

protected

Protected member methods can be called from within the owning type implementation or any type that inherits from that type. Protected methods represent functionality that is not intended to be exposed to the public, but which may be overridden, modified, or used from within types inheriting from the owning type.

The following type expression defines three data members and three member methods. The method `describe` returns a description of the person and is intended to be called by users of the type. The methods `describeName` and `describeAge` are private and protected methods, not intended to be used by the outside world.

```
define person => type {
   data
      public firstName,
      public lastName,
      public age

   public describe() => {
      return .describeName + ', ' + .describeAge
   }
   private describeName() => .firstName + ' ' + .lastName
   protected describeAge() => 'age ' + .age
}
```

Given the definition above, the following example illustrates valid and invalid use of a **person** object:

```
local(p) = person

#p->describe
// =>   , age

#p->describeAge
// => // FAILURE: access not permitted
```

The second usage fails because the `describeAge` method is protected. A type that inherits from person can access `describeAge`, but it cannot access `describeName` because that method is marked as private.

12.1.5 Inheritance

Every type inherits from one or more parent types. To *inherit* from another type means that every instance of the type will automatically possess all of the data members and methods of the parent type, plus those defined in the type expression itself. The concept of inheritance is used to build more complex types out of more generalized types.

A more general type may have several different more specific types inheriting from it as it provides a basic set of functionality that each inheriting type will also possess. Lasso only supports single-inheritance, that is, each type has only one immediate parent and that parent has only one immediate parent. All types can eventually trace down to a **null** parent. If a parent is not explicitly specified when a type is defined then the parent of the type is **null**.

All of the public or protected member methods belonging to a parent type will be made available to the types that inherit from it. Any method defined in a parent type that conflicts with those of an inheriting type will be replaced by the inheriting type's method, unless the parent's method was declared as `frozen`. This permits inheriting types to override or replace functionality provided by a parent.

Parent Section

The *parent* section names the parent that the type being defined is to inherit from. For example, the **person** type can inherit from the **entity** type by naming it in its parent section. Each person object that gets created will then possess all of the data members and methods found in the **entity** type, whatever those might be.

```
define person => type {
   parent entity
}
```

Only one parent type can be listed. The parent section can appear only once in a type expression. While you can place it anywhere in the type expression, it is recommended that you place it at the top.

The following code defines two simple types: **one** and **two**. Type **two** inherits from type **one**. Notice that the **second** method is overridden by the second type, but the **first** method is not.

```
define one => type {
   public first() => 'alpha'
   public second() => 'beta'
   public last() => frozen 'omega'
}

define two => type {
   parent one
   public second() => 'gamma'
   public last() => 'zeta'
}
```

When the **first** method of a **two** object is called, the value "alpha" is returned since it is automatically calling the method from the parent type. The **second** method returns "gamma" since it is calling the overridden method from type **two**. The **last** method always returns "omega" because the parent type defined it with the **frozen** keyword.

```
two->first
// => alpha
two->second
// => gamma
two->last
// => omega
```

Accessing Inherited Methods

Sometimes it is necessary to call "down" to an inherited method. A method inherited from an ancestor (any of the parents down the chain to **null**) can be accessed by using the **inherited** keyword followed by the target operator (**->**) followed by the method call (name and any parameters).

In the following example, the method **third** is defined to call the inherited method **second**. The method from type **two** will be bypassed in favor of the corresponding method from type **one**.

```
define one => type {
   public first() => 'alpha'
   public second() => 'beta'
}

define two => type {
   parent one
   public second() => 'gamma'
   public third() => inherited->second
}

two->third
// => beta
```

Equivalently, Lasso supports a shortcut syntax for targeting "inherited" by using two periods, which can be used to access the methods of a parent type. The example above can be rewritten using `..` in place of `inherited->`.

```
define two => type {
   parent one
   public second() => 'gamma'
   public third() => ..second
}
```

Trait Section

Every type has a single trait which may be composed of other subtraits. A type inherits all of the methods that its trait defines, provided that the type implements the requirements for the trait. For example, a type must be serializable for it to be stored in a session, which means importing the **trait_serializable** trait. (See the *Traits* chapter for a complete description of how traits are created.)

The trait section of a type expression can import one or more other traits. These traits are combined to form the trait for the type. The following code shows a type definition that imports the **trait_array** and **trait_map** traits:

```
define mytype => type {
   trait {
      import trait_array, trait_map
   }
}
```

A trait section can appear anywhere within a type expression, but can appear only once.

12.1.6 Type Creators

A *type creator* is a method that returns a new instance of a type. For example, calling the method named **string** produces a new string object. By default each type has a creator method that corresponds to the name of the type and requires no parameters.

The example type **person** would automatically have a creator method **person** that returns a new instance of the type.

```
// Assigns a new person object to #myperson
local(myperson) = person()
```

If a type does not define its own creator method(s), then it is provided with a default zero-parameter type creator. Attempting to provide parameters to a type creator that does not accept any parameters will fail.

```
local(myperson) = person(264)
// => // FAILURE: person() accepts no parameters
```

onCreate

Many types allow one or more parameters to be provided when a new object is created in order to customize the object before it is used. A type can specify its own type creators by defining one or more methods named **onCreate**. When a new object is created, the **onCreate** method corresponding to the given parameters is immediately called before the new object is returned to the user. Each **onCreate** must be a public member method.

To illustrate, the following type definition defines an **onCreate** method that requires three parameters: "firstName", "lastName", and "birthdate". These parameters correspond to the data members of the type and allow them to be set when the object is first created. The creator simply assigns the parameter values to the data members.

```
define person => type {
    data firstName::string, lastName::string
    data birthdate::date

    public onCreate(firstName::string, lastName::string, birthdate::date) => {
        .'firstName' = #firstName
        .'lastName' = #lastName
        .'birthdate' = #birthdate
    }
}
```

To create an instance of this type, the creator must be called with the required parameters. The following code will create a new instance of the **person** type:

```
local(myperson) = person('Cathy', 'Cunningham', date('1/1/1974'))
```

Note that when a creator has been specified, the default creator, which requires no parameters, is not automatically provided. Lasso will not supply a default type creator when the author has included their own. Also note that if a type overrides its parent's creator, it needs to include a call to the parent's creator method, passing on any arguments as required.

```
public onCreate(...) => ..onCreate(: #rest)
```

Many type creators can be defined by specifying multiple **onCreate** methods. The following type defines three type creators. The first permits **person** objects to be created with no parameters; the second, with first and last names; and the third, with first and last names and a birthdate.

```
define person => type {
    data firstName::string, lastName::string
    data birthdate::date

    public onCreate() => {}
    public onCreate(firstName, lastName) => {
        .'firstName' = string(#firstName)
        .'lastName' = string(#lastName)
    }
    public onCreate(
            firstName::string,
            lastName::string,
            birthdate::date) => {
        .'firstName' = #firstName
        .'lastName' = #lastName
        .'birthdate' = #birthdate
    }
}
```

12.1.7 Callback Methods

In addition to the **onCreate** method, Lasso reserves a number of other method names as callbacks which are automatically used in different situations. Lasso provides default behavior so all callbacks are optional, but by defining a callback a type can customize its behavior.

asString

The **asString** method is called when an object is expressed as a string. By default, a type instance will simply output the name of the object's type. Overriding this method allows a type to control how it is output. The following code defines a simple type

that outputs a greeting when its **asString** method is called:

```
define mytype => type {
    public asString() => 'Hello World!'
}

mytype
// => Hello World!
```

onCompare

The **onCompare** method is called whenever an object is compared against another object. This includes when using the equality (**==**), and inequality (**!=**) operators, and when objects are compared for ordinality using any of the relative equality operators (**<**, **<=**, **>**, **>=**). It's also called via **null->onCompareStrict**, which first verifies that the two objects are the same type, when using the strict equality (**===**) and inequality (**!==**) operators.

An **onCompare** method must accept one parameter and must return an integer value.

```
public onCompare(rhs)::integer
```

If the parameter is equal to the current type instance then a value of "0" should be returned. If the current type instance is less than the parameter then an integer less than zero should be returned (e.g. "-1"). If the current type instance is greater than the parameter then an integer greater than zero should be returned (e.g. "1").

For example, the following **person** type has an **onCompare** method that gives **person** objects the ability to compare themselves with each other:

```
define person => type {
    data public firstName::string,
         public lastName::string

    public onCompare(other::person) => {
        .firstName != #other->firstName ?
            return .firstName < #other->firstName ? -1 | 1
        .lastName != #other->lastName ?
            return .lastName < #other->lastName ? -1 | 1
        return 0
    }

    public onCreate(firstName::string, lastName::string) => {
        .firstName = string(#firstName)
        .lastName = string(#lastName)
    }
}
```

Given the above type definition, the following examples use the **onCompare** method behind the scenes to provide the ability to compare persons:

```
person('Bob', 'Barker') == person('Bob', 'Barker')
// => true

person('Bob', 'Barker') == person('Bob', 'Parker')
// => false
```

Multiple **onCompare** methods can be provided, each specialized to compare against particular object types. For example, an **integer** type would want to permit itself to be compared against other integer objects, but it should also want to be comparable to decimal objects. Such an **integer** type would have one **onCompare** method for integer objects and another

for decimal objects. This example also shows how the **onCompare** method can be manually called on objects. In this case, the "value" data member is responsible for doing the actual comparisons, so its **onCompare** method is called and the value returned.

```
define myint => type {
   data private value

   public onCompare(i::integer) => .value->onCompare(#i)
   public onCompare(d::decimal) => .value->onCompare(integer(#d))
}
```

contains

The **contains** method is called whenever an object is compared using the contains (**>>**) or not contains (**!>>**) operators. A **contains** method definition should accept one parameter and must return a boolean value, either "true" or "false".

```
public contains(rhs)::boolean
```

If the parameter is contained within the current type instance (using whatever logic makes sense for the type) then a value of "true" should be returned; otherwise, a value of "false" should be returned.

For example, the type **odds** below overrides the contains operators so that **odds >> 3** will return "true" and **odds >> 4** will return "false".

```
define odds => type {
   public contains(rhs::integer)::boolean => {
      return #rhs % 2 == 1
   }
}
```

Other types that implement their own **contains** methods include **array** and **map**, which search their contained objects for a match before returning "true" or "false".

invoke

The **invoke** method is called whenever an object is invoked by applying parentheses to it. By default, invoking an object produces a copy of the invoked object. However, objects can add their own **invoke** methods to alter this behavior. The following code shows how an instance of the **person** type might be invoked:

```
define person => type {
   data
      public firstName::string,
      public lastName::string

   public invoke() => .firstName + ' ' + .lastName + ' was invoked!'
   public onCreate(firstName::string, lastName::string) => {
      .firstName = string(#firstName)
      .lastName = string(#lastName)
   }
}
```

The following shows how a **person** object would be invoked, by either directly calling the **invoke** method or by applying parentheses:

```
local(per) = person('Bob', 'Parker')

#per()
// => Bob Parker was invoked!

#per->invoke
// => Bob Parker was invoked!
```

_unknowntag

Implementing the **_unknowntag** method allows a type to handle requests for methods that it does not have. When a search for a member method fails, the system will call the **_unknowntag** method if it is defined. The originally sought method name is available by calling **method_name**.

The following example creates a type whose only member method is **_unknowntag**, which returns the name of the called method:

```
define echo_method => type {
    public _unknowntag => method_name->asString
}

echo_method->rhino
// => rhino
```

12.1.8 Operator Overloading

Types can provide their own routines to be called when the standard arithmetical operators (**+ -* / %**) are used with an instance of the type on the left-hand side of the expression.

If the standard operators are overloaded they should be mapped as closely as possible to the standard arithmetical meanings of the operators. For example, the addition operator (**+**) is also used for string concatenation.

Overloading Arithmetical Operators

An arithmetical operator is overloaded by defining a member method whose name is the same as the operator symbol. The method must accept one parameter and return an appropriate value. The type instance should not be modified by these operations.

```
public +(rhs)
public -(rhs)
public *(rhs)
public /(rhs)
public %(rhs)
```

The following example provides a full set of arithmetical operators for the **myint** type:

```
define myint => type {
    data private value

    public onCreate(value = 0) => { .value = #value }
    public asString() => string(.value)
    public +(rhs::integer) => myint(.value + #rhs)
    public -(rhs::integer) => myint(.value - #rhs)
    public *(rhs::integer) => myint(.value * #rhs)
```

```
    public /(rhs::integer) => myint(.value / #rhs)
    public %(rhs::integer) => myint(.value % #rhs)
}

myint(9) + 5 * 40
// => 209
```

Overloading Equality Operators

See the section on the *onCompare method* for information about how to overload the equality operators (**==**, **!=**, **<**, **<=**, **>**, **>=**, **===**, **!==**).

Overloading Containment Operators

See the section on the *contains method* for information about how to overload the containment operators (**>>**, **!>>**).

12.2 Modifying Types

Lasso permits types to have methods added to them outside of the original defining type expression. This is done by defining the method using the word **define** followed by the name of the type, the target operator (**->**), and then the rest of the method signature and body. The following example adds the method **speak** to the **person** type:

```
define person->speak() => 'Hello, world!'
```

12.3 Type/Object Introspection Methods

Lasso provides a number of methods that can be used to gain information about a type or object. These methods are summarized below, and can be called by any type.

null->type()
> Returns the type name for any type instance. The value is the name that was used when the type was defined.

null->isA(name::tag**)**
> Checks whether an instance of an object is of the given type, returning an integer indicating the result.

> **0**
>> The given type has no relation to the object.

> **1**
>> The **name** parameter matches the type of the instance. (The method call **null->isA(::null)** will only return "1" for the **null** type instance itself.

> **2**
>> The **name** parameter matches a trait implemented by the type of the instance, or one of its parents.

> **3**
>> The **name** parameter matches the parent type of the instance.

null->isNotA(name::tag**)**
> The opposite of **null->isA**.

null->listMethods()
> Returns a staticarray of **signature** objects for all of the methods that are available for the type.

`null->`**hasMethod(**_name::tag_**)**
> Returns "true" if the type implements a method with the given name.

`null->`**parent()**
> Returns the name of the parent of the target object. If the method returns "null" then the final parent has been reached.

`null->`**trait()**
> Returns the trait for the target object. Returns "null" if the object does not have a trait.

> _See also:_

> **setTrait** and **addTrait** methods in the _Traits_ chapter

type **tag**
> An immutable object that represents a unique string of characters. Since Lasso uses tags internally to keep track of names, this type has member methods that can query them.

`tag->`**istype()** → boolean
> Check if a type with the same name as the given tag exists.

```
::string->istype
// => true
```

`tag->`**gettype()**
> Create an instance of a type matching the given tag. This is useful for calling **listMethods** on a type that has no literal syntax or simple type creator.

```
::regexp->gettype->listmethods
// => ... regexp->input(), regexp->replacepattern(), regexp->findpattern(), ...
```

`tag->`**doccomment()**

`tag->`**doccomment=(**_value::string_**)**
> Retrieve and set doc comments for a type matching the given tag. Requires that Lasso be run with the **LASSO9_RETAIN_COMMENTS** variable enabled.

> _See also:_

> _Tag Literals_ and _Doc Comments_ in the _Literals_ chapter

Traits

Traits provide a way to define type functionality in a modular fashion. Each *trait* includes a set of reusable method implementations along with a set of requirements that must be satisfied in order for the included methods to function properly.

13.1 Trait Logic

Traits allow a hierarchy of types that share common functionality to be created without relying on single or multiple inheritance. Traits are similar to mixins and abstract classes found in other languages.

Each trait encapsulates a set of requirements and provides a set of member methods. When a trait is applied to a type, the requirements are checked. If they are satisfied, then the provided member methods are added to the type as if they had been implemented directly in the type. Traits can only define public member methods.

Lasso includes many types that have common member methods. For example, the **pair**, **array**, **string**, and other types implement **first**, **second**, and **last** methods which return the named element.

```
array(1, 2, 3, 4)->last
// => 4

'Quick brown fox'->second
// => u

pair('name'='John')->first
// => name
```

The **first** method can be implemented by calling the **get(x)** member method of each type with a parameter of "1". The **second** method calls it with a parameter of "2". The **last** method calls the **get(x)** method with a parameter defined by the size of the type (usually found by calling the **size** member method).

The requirements for implementing the **first**, **second**, and **last** methods are that the type has to have **get(x)** and **size** member methods. In a trait this requirement would be specified as follows:

```
require get(x::integer)
require size()::integer
```

The requirements take the form of a list of member method signatures. If the type that the trait is applied to defines all of the trait's required member method signatures, then the methods provided by the trait will work.

The methods provided by the trait are specified similar to how methods are defined in custom types. (However, instead of using the **public** keyword, the method definition starts with the **provide** keyword.) The implementation for the **first**, **second**, and **last** methods would appear as follows:

```
provide first() => .get(1)
provide second() => .get(2)
provide last() => .get(.size)
```

Note that the period notation is used to call the member methods of the current object; the same as it would be used within a custom type implementation. The implementation of the provided methods can make use of the **get** and **size** member methods because the requirements ensure that they will be available.

The full trait definition for **trait_firstLast** would be as follows:

```
define trait_firstLast => trait {
    require get(x::integer)
    require size()::integer
    provide first()  => .get(1)
    provide second() => .get(2)
    provide last()   => .get(.size)
}
```

If we define a new type (e.g. **month**) that supports **get** and **size**, then we can import this trait to automatically get an implementation of **first**, **second**, and **last**.

```
define month => type {
    trait {
        import trait_firstlast
    }
    data y, m

    public onCreate(year::integer, month::integer) => {
        .'y' = #year
        .'m' = #month
    }

    public get(x::integer) => {
        return date(-year=.'y', -month=.'m', -day=#x)
    }

    public size()::integer => {
        local(temp) = date(-year=.'y', -month=.'m'+1, -day=1)
        #temp->subtract(-day=1)
        return #temp->dayofmonth
    }
}
```

13.2 Defining Traits

A trait is defined using a *trait expression* consisting of the **define** keyword followed by the trait name, the association operator (**=>**), the keyword **trait**, and a code block containing the definition of the trait.

```
define myTrait => trait {
    // ...
}
```

The code block contains one or more sections which are each identified by a label. Method implementations that are provided by the trait are specified in a provide section. Requirements for the trait are specified in a require section. Other traits can be imported in an import section.

13.2.1 provide

The member methods that a trait provides are specified similarly to the public section of a type definition. The *provide* section begins with the keyword **provide**, which is followed by a comma-separated list of member method definitions. The member list has the same form as custom method definitions. Each method is defined using a signature, the association operator (**=>**), and an expression or code block that defines the implementation of the method.

The following trait would provide two member methods for getting and setting a data member:

```
define myTrait => trait {
   provide getFirstName() => {
      return .firstName
   }
   provide setFirstName(value::string) => {
      .firstName = #value
   }
}
```

13.2.2 require

The *require* section allows specifying a list of method signatures that are required for the trait to operate properly. The signatures may be simple method names, or they may be complete signatures with parameter specifications. As many require sections as are necessary can be specified.

The section begins with the keyword **require** followed by a comma-separated list of method signatures. The following trait requires a getter and setter for the "firstName" data member:

```
define myTrait => trait {
   require firstName, firstName=
   provide getFirstName() => {
      return .firstName
   }
   provide setFirstName(value::string) => {
      .firstName = #value
   }
}
```

13.2.3 import

The *import* section allows the characteristics of other traits to be imported into this trait definition, thus allowing a hierarchy of traits to be defined. As many import sections as are necessary can be specified.

The section begins with the keyword **import** followed by a comma-separated list of trait names. The following trait simply imports the characteristics of the built-in **trait_array** trait:

```
define myTrait => trait {
   import trait_array
}
```

All of the requirements and provided member methods of the imported trait will be added to the trait being defined. The requirements of one of the traits may be satisfied by the methods provided by another trait.

However, if two traits provide the same member method, there will be a conflict. The conflict is resolved by eliminating both implementations of that member method and adding a requirement for it to the trait. The type that the trait is ultimately applied to must implement that member method in order for the trait to be applied.

13.3 Trait Composition

Traits can be combined together into new traits using the **+** operator. This is called "composing" a new trait. The result of this expression will be a trait that has all the requirements and provides all the member methods of the traits that have been combined.

The same rules that are used for importing traits apply to composed traits, such as traits satisfying each others' requirements and resolving conflicting method names.

An alternate method of defining the trait example from the start of this chapter would be to define three subtraits and then use the composition operator (**+**) to compose them into a single trait.

```
define trait_first => trait {
    require get
    provide first() => .get(1)
}
define trait_second => trait {
    require get
    provide second() => .get(2)
}
define trait_last => trait {
    require get, size
    provide last() => .get(.size)
}
define trait_firstLast => trait_first + trait_second + trait_last
```

Replacing the last line with the trait definition below would produce exactly the same result. In general, the latter method is preferred for trait definitions, while the trait composition is preferred for runtime changes.

```
define trait_firstlast => trait {
    import trait_first
    import trait_second
    import trait_last
}
```

13.4 Checking Traits

Since traits provide member methods for a type it is often useful to check whether a given type instance has a trait applied. The **isA** method can be used for this check. This member method can be used on any type instance and will return a positive integer if the instance is the provided type or has the provided trait name applied to it.

In this code the **isA** method returns "2" since the **month** type does have the **trait_firstLast** trait applied to it:

```
local(mymonth) = month(2008, 12)

#mymonth->isA(::trait_firstlast)
// => 2
```

13.5 Applying Traits

Traits can be applied to types as part of the type definition. This makes the trait an integral part of the type definition. The provided member methods are indistinguishable to the user of the type from member methods that are implemented directly in the type.

Each type definition can include a single trait section. The trait can import as many traits as are needed.

```
define myType => type {
    trait {
        import ...
    }
    data ...
    public ...
}
```

When an instance of the type is created, the instance has the specified trait applied to it automatically.

The trait of any object in Lasso can be programmatically manipulated using the **trait**, **setTrait**, and **addTrait** methods described in the next section.

13.6 Trait Manipulation Methods

null->**trait**(*t::trait*)
> Returns the trait for the target object. Returns "null" if the object does not have a trait.

null->**setTrait**(*t::trait*)
> Sets the trait of the target object to the parameter, replacing the existing trait.

null->**addTrait**(*t::trait*)
> Combines the target object's trait with the parameter.

In general, traits will be added to a type instance to provide additional functionality rather than resetting the entire trait for a given object. The two examples below are equivalent:

```
#myinstance->addtrait(trait_firstlast)
#myinstance->settrait(#myinstance->trait + trait_firstlast)
```

Caution: The **setTrait** method should be used with care since resetting the trait of a type instance may result in many of its member methods becoming unavailable or ceasing to function.

Error Handling

Responding to errors gracefully is a hallmark of good programming. Errors in Lasso run the gamut from expected errors, such as a database search that returns no records, to syntax errors that require fixing before a page will even process. Lasso provides tools to manage errors at several different levels, which can act redundantly to ensure that no errors will be missed.

14.1 Error Types

The following lists the types of errors that can occur in Lasso:

Web Server Errors
These include "file not found" errors and access violations in realms. These will be reported with standard HTTP response codes (e.g. 404 for "File Not Found").

Syntax Errors
These include misspellings of type or method names, missing delimiters, and mismatched types. Lasso will return an error message rather than the processed Lasso page if it encounters a syntax error.

Action Errors
These include missing or misspelled database names, table names, or field names, and other problems specifying database actions. The database action cannot be performed until the error is corrected.

Database Errors
These are generated by the data source application and include type mismatches, missing required field values, and others. Lasso will report the returned error from the data source application without modification.

Logical Errors
These are problems that cause a page to process unexpectedly even though the syntax of the code is correct. These include infinite loops, missing cases, and assumptions about the size or composition of a found set.

Security Violations
These are not strictly errors, but are attempts to perform database actions or file accesses that are not allowed by the permissions set for the current user.

Operating System Errors
These are errors reported by the operating system Lasso is running on. One example of these errors is trying to perform file operations on a directory.

Some errors are more serious than others. Pages will not be processed at all if they contain syntax errors or if there are operational problems that prevent Lasso Server from being accessed. Other errors are commonly encountered in the normal use of a website. Most database errors and security violations are handled by simply showing a "No Records Found" message or displaying a security dialog box to prompt the user for a username and password.

The following mechanisms for handling errors can be used individually or in concert to provide comprehensive error handling:

- Automatic error reporting is performed by Lasso in response to unhandled errors.

- A custom error page allows the automatic error report to be replaced by a custom page. Custom error pages are usually created for each site on a server.

- Error handling methods allow action and logical errors and security violations to be handled within a Lasso page.

- Error handling methods allow advanced error handling to be built into Lasso pages. These techniques allow error handling routines to be built into a page without disrupting the normal processing of a page if no errors occur.

14.2 Error Reporting

Lasso Server delivers an error report in response to an error that prevents processing of the page. This error report contains an error code, message, and stack trace which can be used to identify the cause and location of an error. The various parts of the stack can be accessed using the **error_...** methods.

While the standard error report is great for developers, it is meaningless for visitors to your website. A custom error page can be defined to be displayed to a site visitor rather than Lasso's built-in error report. The error message displayed on a custom error page will depend on the Lasso code used on the custom page.

To define a custom error page, create a file named **error.lasso** and place it in the root of the web serving folder. Each distinct web serving folder on a host can have a custom error page.

Custom error pages can be further fine-tuned by placing the **error.lasso** file in the web serving folder's subdirectories. Lasso Server will process the first **error.lasso** it encounters on the file path, starting with the current directory and continuing upward until it reaches the root of the web serving folder. If none are found, Lasso Server will use the default error report.

14.2.1 Error Reporting Methods

The **error_...** methods in Lasso allow custom errors to be reported and provide access to the most recently reported error by the code executing in the current Lasso page. This allows a developer to check for specific errors and respond, if necessary, with an error message or with code to correct the error.

Lasso maintains a single error code and error message, which is set by any method that reports an error. The error code and error message should be checked immediately after a method that may report an error. If any intervening methods or expressions report errors, then the original error code and error message will be lost.

Custom errors can be created using the **error_setErrorMessage** and **error_setErrorCode** methods. Once set, the **error_currentError** method or **error_code** and **error_msg** methods will return the custom error code and message. A developer can use these methods to incorporate both built-in and custom error codes into the error recovery mechanisms for a site.

error_currentError(*-errorCode=?***)**
> Returns the current error message. The optional **-errorCode** parameter causes the current error code to be returned instead.

error_code()
> Returns the current error code.

error_msg()
> Returns the current error message.

error_obj()
> Returns the current error name from the Lasso variable **$_err_obj**, or "null" if no error object is present.

error_push()
> Pushes the current error condition onto a stack and resets the current error code and error message.

error_pop()
> Restores the most recent error condition stored using **error_push**.

error_reset()
> Resets the current error code and error message.

error_setErrorCode(*code*)
>   Sets the current error code to a custom value.

error_setErrorMessage(*msg*)
>   Sets the current error message to a custom value.

error_stack()
>   Returns the stack trace for the current error.

Display the Current Error

The following code will display a short error message using the **error_msg** method and the **error_code** method. If the code on the page is executing normally and there is no current error to report then the code will return the result shown below:

```
'The current error is ' + error_code + ': ' + error_msg
// => The current error is 0: No Error
```

Alternatively, the **error_currentError** method could be used to create the same message with the following code:

```
'The current error is ' + error_currentError(-errorCode) + ': ' + error_currentError
// => The current error is 0: No Error
```


Set the Current Error

The current error code and message can be set using the **error_setErrorCode** and **error_setErrorMessage** methods. These methods will not affect the execution of the current Lasso page, but will simply set the current error so it will be returned by the **error_currentError** method or **error_code** and **error_msg** methods.

In the following example, the error message is set to "A custom error occurred" and the error code is set to "-1":

```
error_setErrorMessage('A custom error occurred')
error_setErrorCode(-1)
```

The **error_currentError** method now reports this custom error when it is called later in the page, unless any intervening code changed the error message again:

```
'The current error is ' + error_code + ': ' + error_msg
// => The current error is -1: A custom error occurred
```

The current error code and message can also be set using the **error_code** and **error_msg** methods:

```
error_msg = 'A custom error occurred'
error_code = -1
```


Store and Restore the Current Error

The following code uses the **error_push** and **error_pop** methods to store the current error code and message before the **protect** block is executed. This allows the **protect** block to execute without any previous error on the page bleeding into it and mistakenly triggering the **handle_failure** block. Then the error code and message are restored at the end of the block.

```
error_push  // Push error onto stack

protect => { // Protect from failure
   handle_failure => {
       // Handle any errors generated within the protect block
```

```
    }
    // ...
}

error_pop   // Retrieve error from stack
```

The **error_push** and **error_pop** methods can also be used to prevent custom methods from modifying the current error condition, while still using error-handling code within the method. The following code stores the current error code and message at the beginning of the custom method definition. The error code and message are restored just before the custom method returns a value.

```
define myMethod() => {
    // Push current error onto stack
    error_push

    // ... code that may generate an error ...

    // Retrieve error from stack
    error_pop

    return 'myValue'
}
```

Reset the Current Error

The following code demonstrates how to use the **error_reset** method to reset the error message to "No error" and the error code to "0":

```
error_code = -1
error_msg  = 'Too slow'
error_code + ': ' + error_msg

// => -1: Too slow

error_reset
error_code + ': ' + error_msg

// => 0: No error
```

14.2.2 Lasso Errors

The table below lists Lasso's standard error codes and values.

Table 14.1: Lasso Error Codes and Messages

Error Method	Value
error_code_noerror	0
error_msg_noerror	No error
error_code_fileNotFound	404
error_msg_fileNotFound	File not found
error_code_runtimeAssertion	-9945
error_msg_runtimeAssertion	Runtime assertion
error_code_aborted	-9946
error_msg_aborted	General Abort
error_code_methodNotFound	-9948
error_msg_methodNotFound	Method not found
error_code_divideByZero	-9950
error_msg_divideByZero	Divide by Zero
error_code_invalidParameter	-9956
error_msg_invalidParameter	Invalid parameter
error_code_networkError	-9965
error_msg_networkError	Network error
error_code_resNotFound	-9967
error_msg_resNotFound	Resource not found

14.3 Error Handling

Lasso includes powerful error handling methods that allow areas of a page to be protected and errors to be handled. Error-specific handlers are called if any errors occur in a protected area of a page. These methods allow comprehensive error handling to be built into a page without disturbing the code of the page with many conditionals and special cases.

14.3.1 Error Handling Methods

fail(*msg::string*)

fail(*code::integer*, *msg::string*, *stack::string=?*)
    Halts execution and generates the specified error. Can be called with just an error message, an error code and an error message, or an error code, message, and stack trace.

fail_if(*cond*, *msg::string*)

fail_if(*cond*, *code::integer*, *msg::string*)
    Conditionally halts execution and generates the specified error if the specified condition evaluates to "true". Takes two or three parameters: a conditional expression, an integer error code, and a string error message or just the conditional expression and the error message.

handle(*cond=?*)
    Conditionally executes a given capture block after the code in the current capture block or Lasso page has completed or a **fail** method is called. May take a conditional expression as a parameter that limits executing the capture block to when the conditional statement evaluates to "true". If an error occurs in the Lasso code before the handle block is defined, then the handle's capture block will not be executed.

handle_failure(*cond=?*)
> Functions the same as **handle** except that the contents are executed only if an error was reported in the surrounding capture block or Lasso page.

protect()
> Protects a portion of a page. If code inside the given capture block throws an error or a **fail** method is executed inside the capture block, then the error is not allowed to propagate outside the protected capture block. This means that a **fail** will only halt the execution of the rest of the code in the **protect** capture, and execution will resume starting with the code following that capture.

abort()
> Sets the current error code to **error_code_aborted** and stops Lasso from continuing execution. This *cannot* be stopped with **protect**.

14.3.2 handle and handle_failure

The **handle** method is used to specify a block of code that will be executed after the current code segment is completed. The **handle** method can take a single parameter that is a conditional expression (defaults to "true"). If the conditional expression evaluates as "true", then the code in the given capture block is executed.

All **handle** and **handle_failure** methods are processed sequentially, giving each a chance to be executed in the order they were specified and allowing for execution of multiple **handle** blocks. Therefore, it is necessary to define them before logic that could halt execution. Any **handle** methods that are defined after a script failure will not be executed. It is generally good practice to place **handle** and **handle_failure** methods at the start of the parent capture block, most commonly a **protect** capture block. (This is a change from previous versions of Lasso and increases the reliability of executing fault-condition fallbacks.)

The **handle** methods will not be executed if a syntax error occurs while Lasso is parsing a page. When Lasso encounters a syntax error it returns an error page instead of processing the code on the page.

The **handle** methods will be executed if a logical error occurs while Lasso is processing a page. However, the returned result will be an error message rather than the output of the page. Code within the **handle** capture block can redirect the user to another page using **redirect_url** or can replace the contents of the page being served.

There are two ways to use **handle** methods within a Lasso page:

1. When used on their own in a Lasso page, the code inside the **handle** methods will be conditionally executed after all the rest of the code in the Lasso page has completed. The **handle** methods can be used to provide post-processing code for a Lasso page.

2. When used within any Lasso capture block, the code inside the **handle** methods will be conditionally executed after the capture block is executed. The **handle** methods will most commonly be used within a **protect** block to provide error handling.

14.3.3 fail and fail_if

The **fail** method allows an error to be triggered from within Lasso code. Use of the **fail** method immediately halts execution of the current page and starts execution of any registered **handle** method contained within.

The **fail** method can be used in the following ways:

- To report an unrecoverable error. Just as Lasso automatically halts execution of a Lasso page when a syntax error or internal error is encountered, Lasso code can use the **fail** method to report an error that cannot be recovered from:

  ```
  fail(-1, 'An unrecoverable error occurred')
  ```

- To trigger immediate execution of the page's **handle** methods. If an error is handled by one of the **handle** methods specified in the Lasso page (outside of any other capture blocks), then the code within the **handle** capture block will be executed. The **handle** block can recover from the error and allow execution to continue by using the **error_reset** method.

- To trigger immediate execution of a **protect** capture block's **handle** block, which is described in the next section.

The **fail_if** method allows conditional execution of a **fail** without using a full if/else conditional. The first parameter to **fail_if** is a conditional expression. The last two parameters are the same integer error code and string error message as in the **fail** method. In the following example the **fail_if** method is only executed if the variable "x" does not equal "0":

```
fail_if(#x != 0, 100, "Value does not equal 0.")
```

14.3.4 protect

The **protect** method is used to catch any errors that occur within the code surrounded by the capture block. They create a protected environment from which errors cannot propagate to the page itself. Even if Lasso reports an internal error it will be caught by the **protect** method, allowing the rest of the page to execute successfully.

Any **fail** or **fail_if** methods called within **protect** capture blocks will halt execution only of the code contained within the **protect** capture block. Any **handle** capture blocks contained within the **protect** capture blocks will be conditionally executed. However, Lasso requires these **handle** capture blocks to be present before the error occurs, so put them at the top of the **protect** capture block. The Lasso page will continue executing normally after the closing of the **protect** capture block.

The **protect** capture blocks can be used for the following purposes:

- To protect a portion of a page so that any errors that would normally result in an error message being displayed to the user are instead handled in the internal **handle** capture blocks.

- To provide advanced flow control in a page. Code within the **protect** capture blocks is executed normally until a **fail** signal is encountered. The code then jumps immediately to the internal **handle** block.

Protect a Portion of a Page from Errors

Wrap the portion of the page that needs to be protected in a **protect** capture block. Any internal errors that Lasso reports will be caught by the **protect** capture block and not reported to the end user. A **handle** capture block should be included to handle the error if necessary.

In the following Lasso code an attempt is made to set a variable "myVar" to "null". However, if the variable has not been previously declared, an error would be reported, and the page would not continue processing. Since the code is executed within a **protect** capture block, no error is reported, and the **protect** capture block exits silently while the Lasso page resumes execution after the **protect** block.

```
protect => {
    $myVar = null
}
```

Use protect with Custom Errors

The following example shows a **protect** capture block that surrounds code containing two **fail_if** statements with custom error codes "-1" and "-2". A **handle** block at the start of the **protect** is set to intercept either of these custom error codes. This **handle** block will only execute if one of the **fail_if** methods executes successfully.

```
protect => {^
   handle => {^
      if(error_code == -1)
         '... Handle custom error -1 ...'
      else(error_code == -2)
         '... Handle custom error -2 ...'
      else
         '... Another error has occurred ...'
      /if
   ^}

   'Before the fail_if\n'

   local(
      condition_one = false,
      condition_two = true
   )
   fail_if(#condition_one, -1, 'Custom error -1')
   fail_if(#condition_two, -2, 'Custom error -2')

   '\nAfter the fail_if'
^}

// =>
// Before the fail_if
// ... Handle custom error -2 ...
```

Threading

In computing, a *thread* is a sequence of instructions being managed by an operating system. Lasso has integrated support for running multiple threads, allowing it to handle many application requests at the same time. Threading in Lasso is designed to be easy to use and safe. Lasso does not feature global variables, so all data is local to individual threads. Threads can communicate with one another by sending object messages back and forth. These objects are copied as they are transmitted to ensure that data structures remain consistent.

Lasso supports creating or splitting a new thread given a block of code. It also supports creating thread objects which run in their own thread.

15.1 Splitting Threads

A new thread can be created by calling the **split_thread** method. This method requires a capture block. The capture given to **split_thread** will be run in a new thread. This new thread will contain copies of the local variables that are active at the time the new thread is created. Changing the value of a variable in the new thread will not affect the variables that were active at the creation point. Additionally, the current self is cleared for the new thread.

split_thread() → pair
> Takes a capture assigned as a capture block and runs that capture in a separate thread. Any local variables that would normally be available to that capture are copied and available in the new thread. This method also returns a pair object with file descriptors for writing and reading messages to and from the newly created thread.

15.1.1 Thread with Capture

The following example shows a new thread being created. The new thread simply prints a message to the console. This illustrates how **split_thread** is used and how a new capture (between curly braces **{ ... }**) is given to **split_thread** which will be run in a new thread.

```
split_thread => {
    stdoutnl("I'm alive in a new thread!")
}
```

15.1.2 Thread Communication

When a new thread is created by calling **split_thread**, the return value of that method call is a pair of filedesc objects. Similarly, the parameter given to the new thread is a pair of filedesc objects. (This can be accessed in the new thread by the pseudo-local variable **#1**.) The **filedesc** type represents a file descriptor or pipe over which data can be sent or received. These objects provide the means for the new thread and the creator thread to communicate. Two filedesc objects are required for thread communication, one representing the *write* end of the pipe and the other representing the *read* end. Objects are written to the write filedesc and read from the read filedesc.

Within this context of the given pair of filedescs, the write filedesc is always the first member of the pair while the read filedesc is always the second member. The creator thread writes objects to the new thread using the write filedesc, and reads objects

from the new thread using the read. The newly created thread operates in the same manner, writing and reading objects to and from its creator thread.

Send and Receive Objects Between Threads

The next example creates a new thread and illustrates how objects can be sent and received:

```
// Create the new thread, saving the filedesc pair in #creatorPipes
local(creatorPipes) = split_thread => {

    // Save the filedescs sent to this new thread
    local(
        writePipe = #1->first,
        readPipe = #1->second
    )

    // Loop forever, reading messages and sending replies
    while(true) => {

        // Read an object
        local(o) = #readPipe->readObject

        // Print a message
        stdoutnl("I read an object: " + #o)

        // Write a reply object
        #writePipe->writeObject("Reply from the new thread")
    }
}

// Write an object to our new thread
#creatorPipes->first->writeObject("Sent from the creator!")

// Read the reply from the new thread
stdoutnl(#creatorPipes->second->readObject)

// Do it again
#creatorPipes->first->writeObject("Sent from the creator 2!")
stdoutnl(#creatorPipes->second->readObject)

// =>
// I read an object: Sent from the creator!
// Reply from the new thread
// I read an object: Sent from the creator 2!
// Reply from the new thread
```

Threads created with **split_thread** exit when they reach the end of their code body. If the example thread above did not loop reading/writing messages, it would read one message, write one reply, reach the end of its code, and then exit.

15.2 Thread Objects

Thread objects represent a second way to create new threads in Lasso. A *thread object* is an object that exists in its own thread. This means that any method calls to a thread object run serially in the object's thread. Thread objects exist as singletons, which means that only one instance of a particular thread type can exist. Thread objects permit data to be globally available, yet forces access to that data to be synchronized.

Thread objects are created and begin running at the point where they are defined. Thread types are defined similarly to how normal types are defined, except that in such a definition, the word **type** is replaced with the word **thread**.

15.2.1 Simple Counter Thread

The following example creates a simple thread object. This object maintains a counter that can be advanced and retrieve its current value. Because this is a thread object, it is globally available and other threads can safely advance the counter.

```
define counter_thread => thread {
   data private val = 0

   public advanceBy(value::integer) => {
      .val += #value
      return .val
   }
}
```

The above example defines a **counter_thread** object. This object exists and begins running as soon as it is defined. Clients can access the thread object by calling it by name; in this case by calling the **counter_thread** method:

```
counter_thread->advanceBy(40)
// => 40

counter_thread->advanceBy(10)
// => 50
```

Note that each time **counter_thread** is called, the same thread object is retrieved. Hence, after the second call to **counter_thread->advanceBy**, the "val" data member has a value of "50".

Thread objects can be composed of the same elements as a regular type, including public and private data members, and can have any other (non-thread) object type as a parent.

15.2.2 Simple Map Thread

This next example creates a thread type that inherits from type **map**. This results in creating a global map of values that can be safely accessed by other threads.

```
define map_thread => thread {
   parent map
   public onCreate() => ..onCreate
}

map_thread->insert('one'=1) & insert('two'=2)

map_thread->get('two')
// => 2
```

Thread objects cannot be copied. Additionally, thread objects will continue to run forever, though they can terminate themselves by calling **abort**. Also, all parameter values given to a thread object method are copied, as well as any return value of a thread object method. This ensures that no two threads are ever operating on the same data at the same time, a situation that can have catastrophic results.

15.2.3 Thread Objects and onCreate

Because thread objects are created as soon as they are defined, a thread object must have a zero parameter **onCreate** method, or no **onCreate** methods at all. If a thread object requires further configuration, as would normally be done at the point of object creation, it should be done immediately following the thread object's definition. For example, the **counter_thread** could be defined to permit its "val" data member to have an initial value set, as shown in the following code:

```
define counter_thread => thread {
   data private val = 0

   // Default zero-parameter onCreate
   public onCreate() => {}

   // Additional onCreate, letting val be initialized
   public onCreate(initValue::integer) => {
      .val = #initValue
   }

   public advanceBy(value::integer) => {
      .val += #value
      return .val
   }
}

// Initialize the counter
counter_thread->onCreate(900)

// Now it can be used
counter_thread->advanceBy(20)
// => 920
```

15.2.4 active_tick

Thread objects can define a method named **active_tick**. If defined, this method will be called periodically by the system. This lets a thread object carry out periodic activity regardless of any methods called by clients. The **active_tick** method should accept zero parameters, and should return an integer value. The integer value tells the system how many seconds *at the latest* the **active_tick** method should be called again. The **active_tick** method may be called sooner than the indicated time as it provides the timeout value for reading messages for that thread. Threads requiring precise timing for events should not rely on the **active_tick** calls only being called after the timeout value.

The next example defines a thread object that prints a message to the console every 2 seconds:

```
define lazy_ticker => thread {
   public active_tick() => {
      stdoutnl("Hello, from lazy ticker")
      return 2
   }
}
```

The **active_tick** method can be one of several member methods, can reference and call other member methods, and the tick timer (return value) can be programmatically manipulated so that it does not have to be a hard-coded value. In this way, a single **active_tick**-enabled thread can manage multiple tasks and conditionally perform additional tasks based on the results of its basic task, can put itself to sleep or adjust the sleep timer, and have methods that are called completely separately from the **active_tick** method. In short, any thread type can also contain an **active_tick** method to perform periodic maintenance or time-sensitive tasks.

Part III

Data Handling

Strings

Text in Lasso is stored and manipulated using the **string** type or the `string_...` methods. This chapter details the operators and methods that can be used to manipulate string values.

Tip: The **string** type is often used in conjunction with the **bytes** type to convert binary data between different character encodings (e.g. UTF-8, ISO-8859-1). See the *Byte Streams* chapter for more information about the **bytes** type.

16.1 String Objects

Text processing is a central function of Lasso. Many Lasso methods are dedicated to outputting and manipulating text. Lasso is used to format text-based HTML pages or XML data for output. Lasso is also used to process and manipulate text-based HTML form inputs and URLs.

As a result of this focus on text processing, the **string** type is the primary type of data in Lasso. The result of all expressions are converted to strings before they are output into the HTML page or XML data being served.

The following operations that can be performed directly on strings:

1. Operators can be used to perform string calculations:

    ```
    'The' + ' ' + 'String'
    // => The String
    ```

2. String member methods can be used to manipulate the string value:

    ```
    'the string'->titlecase&;
    // => The String
    ```

3. String member methods can be used to return new strings based on the value of the current string:

    ```
    'The String'->sub(5, 6)
    // => String
    ```

4. String member methods can be used to test the attributes of strings:

    ```
    'The String'->contains('the')
    // => true
    ```

Each of these methods is described in detail in the sections that follow. This chapter contains a description and examples of using operators and methods to manipulate strings.

16.1.1 Unicode Characters

Lasso supports the processing of Unicode characters in all **string** methods. The escape sequence **\u...** can be used with 4 hexadecimal digits (or **\U...** with 8 or **\x...** with 2) to embed a Unicode character in a string. For example **\u002F** represents a "/" character, **\u0020** represents a space, and **\u0042** represents a capital letter "B". The same type of escape sequence can be used to embed any Unicode character, e.g. **\u4E26** represents the Traditional Chinese character 並.

Lasso also supports common escape sequences including **"\r"** for a return character, **"\n"** for a newline character, **"\r\n"** for a Windows return/newline, **"\f"** for a form-feed character, **"\t"** for a tab, and **"\v"** for a vertical-tab. See the table *Supported String Escape Sequences* for the full list.

16.2 Converting Values to Strings

Expressions that produce a value will convert that value to the **string** type automatically, or they can be explicitly converted using the **string** creator method as well as the **asString** member method every object has.

string(*obj::any*)

string(*obj::bytes, enc::string=?*)
> Converts a value to type **string**. Requires one value which is the data to be converted. An optional second parameter can be used when converting byte streams in order to specify which character set should be used to translate the byte stream to a string (defaults to "UTF-8").

16.2.1 Automatic String Conversion

Integer and decimal values are converted to strings automatically if they are used as a parameter to a string operator. If either of the parameters to the operator is a string then the other parameter is converted to a string automatically. The following example shows how the integer **123** is automatically converted to a string because the other parameter of the **+** operator is the string **'String'**:

```
'String ' + 123
// => String 123
```

The following example shows how a variable that contains the integer **123** is automatically converted to a string for the expression:

```
local(number) = 123
'String ' + #number + '\n' + #number->type

// =>
// String 123
// integer
```

Array, map, and pair values are converted to strings automatically when they are output to a web page or included as part of an auto-collect block. The value they return is intended for the developer to be able to see the contents of the complex type and is not intended to be displayed to site visitors.

```
array('One', 'Two', 'Three')
// => array(One, Two, Three)

map('Key1'="Value1", 'Key2'="Value2")
// => map(Key1 = Value1, Key2 = Value2)

pair('name'='value')
// => (name = value)
```

The parameters sent to the `string_...` methods are automatically converted to strings. The following example shows the result of calling `string_length` on an integer:

```
string_length(21)
// => 2
```

16.2.2 Explicitly Convert a Value to a String Object

Integer and decimal values can be converted to string objects using the **string** creator method. The value of the new string is the same as the value of the integer or decimal value when it is output using the **toString** method.

The following example shows a math calculation and the integer result **579**. The next line shows the same calculation with string parameters and the result of **123456**.

```
123 + 456
// => 579

string(123) + string(456)
// => 123456
```

Boolean values can also be converted to a string object using the **string** creator method. The value will always either be the string "true" or the string "false". The following example shows a conditional result converted to type **string**:

```
string('dog' == 'cat')
// => false
```

String member methods can be used on any value by first converting that value to a string using either the **string** creator method or the **asString** member method every object has. The following example shows how to use the **string->size** member method on an integer by first converting it to a string object:

```
21->asString->size
// => 2

string(21)->size
// => 2
```

Byte streams that are converted to strings can include the character set to be used to export the data in the byte stream. By default byte streams are assumed to contain UTF-8 character data. The following example code would translate a byte stream contained in a variable named "myByteStream" using the ISO-8859-1 encoding to interpret the character data. This is analogous to using the **bytes->exportString** method which is described in more detail in the *Byte Streams* chapter:

```
string(#myByteStream, 'ISO-8859-1')
```

16.3 String Inspection Methods

The **string** type has many member methods that return information about the value of the string object. Many of these methods are documented below. (Information about regular expressions and the **regexp** type is found in the *Regular Expressions* chapter.)

type **string**

`string->`**`length()`**
> *Deprecated since version 9.0:* Use **string->size** instead.

string->**size**()
>   Returns the number of characters in the string.

string->**charName**(*position::integer*)
>   Takes a parameter that specifies the position of the character to inspect. It returns the Unicode name for the specified character.

string->**charType**(*position::integer*)
>   Takes a parameter that specifies the position of the character to inspect. It returns the Unicode type for the specified character.

string->**digit**(*position::integer*, *base::integer*)
>   Takes a parameter that specifies the position of the character to inspect and a parameter that specifies the base or radix. If the specified character is a digit for the specified radix, then it returns the integer value for that digit. (Remember that when integers are converted to strings, they default to displaying in base 10.) The radix or base can be any from "2" to "36".

string->**sub**(*position::integer*, *size::integer=?*)

string->**substring**(*start::integer*, *size::integer=?*)
>   Returns a portion of the string. The starting point is specified by the first parameter and the number of characters to return is specified by the second. If the second parameter is not specified, then all characters from the specified starting position to the end of the string are returned.

string->**integer**(*position::integer=?*)
>   Takes a parameter that specifies the position of the character to inspect, defaulting to the first character if no position is specified. It returns the Unicode integer value of that character.

string->**charDigitValue**(*position::integer*)
>   Takes a parameter that specifies the position of the character to inspect. If the specified character is a digit, then it will return an integer of the value of the digit. Otherwise it returns "-1".

string->**getNumericValue**(*position::integer*)
>   Takes a parameter that specifies the position of the character to inspect. If the specified character is a digit, then it will return a decimal of the value of the digit. Otherwise it returns the decimal "-123456789.0".

string->**isAlnum**(*position::integer=?*)
>   Takes a parameter that specifies the position of the character to inspect, defaulting to the first character. If the specified character is alphanumeric the method will return "true" otherwise it will return "false".

string->**isAlpha**(*position::integer=?*)
>   Takes a parameter that specifies the position of the character to inspect, defaulting to the first character. If the specified character is alphabetic the method will return "true" otherwise it will return "false".

string->**isUAlphabetic**(*position::integer=?*)
>   Takes a parameter that specifies the position of the character to inspect, defaulting to the first character. If the specified character has the Unicode alphabetic property then the method will return "true" otherwise it will return "false".

string->**isBase**(*position::integer=?*)
>   Takes a parameter that specifies the position of the character to inspect, defaulting to the first character. If the specified character is a base Unicode character the method will return "true" otherwise it will return "false".

string->**isBlank**(*position::integer=?*)
>   Takes a parameter that specifies the position of the character to inspect, defaulting to the first character. If the specified character is a space or tab the method will return "true" otherwise it will return "false".

string->**isCntrl**(*position::integer=?*)
>   Takes a parameter that specifies the position of the character to inspect, defaulting to the first character. If the specified character is a control character then the method will return "true" otherwise it will return "false".

string->**isDigit**(*position::integer=?*)
>   Takes a parameter that specifies the position of the character to inspect, defaulting to the first character. If the specified character is a base 10 digit then the method will return "true" otherwise it will return "false".

string->**isXDigit**(*position::integer=?*)
>   Takes a parameter that specifies the position of the character to inspect, defaulting to the first character. If the specified character is a hexadecimal digit then the method will return "true" otherwise it will return "false".

string->**isGraph**(*position::integer=?*)
>   Takes a parameter that specifies the position of the character to inspect, defaulting to the first character. If the specified character is printable and not whitespace then the method will return "true" otherwise it will return "false".

string->**isLower**(*position::integer=?*)
>   Takes a parameter that specifies the position of the character to inspect, defaulting to the first character. If the specified character is lowercase the method will return "true" otherwise it will return "false".

string->**isULowercase**(*position::integer=?*)
>   Takes a parameter that specifies the position of the character to inspect, defaulting to the first character. If the specified character has the Unicode lowercase property then the method will return "true" otherwise it will return "false".

string->**isPrint**(*position::integer=?*)
>   Takes a parameter that specifies the position of the character to inspect, defaulting to the first character. If the specified character is printable the method will return "true" otherwise it will return "false".

string->**isPunct**(*position::integer=?*)
>   Takes a parameter that specifies the position of the character to inspect, defaulting to the first character. If the specified character is punctuation the method will return "true" otherwise it will return "false".

string->**isSpace**(*position::integer=?*)
>   Takes a parameter that specifies the position of the character to inspect, defaulting to the first character. If the specified character is whitespace the method will return "true" otherwise it will return "false".

string->**isTitle**(*position::integer=?*)
>   Takes a parameter that specifies the position of the character to inspect, defaulting to the first character. If the specified character is in the Unicode category "Letter, Titlecase" then the method will return "true" otherwise it will return "false".

string->**isUpper**(*position::integer=?*)
>   Takes a parameter that specifies the position of the character to inspect, defaulting to the first character. If the specified character is uppercase the method will return "true" otherwise it will return "false".

string->**isUUppercase**(*position::integer=?*)
>   Takes a parameter that specifies the position of the character to inspect, defaulting to the first character. If the specified character has the Unicode uppercase property then the method will return "true" otherwise it will return "false".

string->**isWhitespace**(*position::integer=?*)
>   Takes a parameter that specifies the position of the character to inspect, defaulting to the first character. If the specified character is whitespace the method will return "true" otherwise it will return "false".

string->**isUWhitespace**(*position::integer=?*)
>   Takes a parameter that specifies the position of the character to inspect, defaulting to the first character. If the specified character has the Unicode whitespace property then the method will return "true" otherwise it will return "false".

string->**find**(*find::string, offset::integer, -case::boolean=?*)

string->**find**(*find::string, offset::integer, length::integer*)

string->**find**(*find::string, offset::integer, length::integer, patOffset::integer, patLength::integer, case::boolean*)

string->**find**(*find::string,   -offset::integer=?,   -length::integer=?,   -patOffset::integer=?,   -patLength::integer=?, -case::boolean=?*)
>   Searches the value of the string object for the specified string pattern, returning the position of where the pattern first begins in the string object value or zero if the pattern cannot be found.

An optional **-case** parameter can be used to specify case-sensitive pattern matching. The **-offset** and **-length** parameters can be used to specify a portion of the string within which to look for the match, with the former specifying the position to begin the search and the latter specifying the number of characters to search. (If **-length** is not specified, the method will search to the end of the string.) The **-patOffset** and **-patLength** parameters can be used to specify that only a portion of the pattern should be used for matching; they behave similarly for the pattern string as the **-offset** and **-length** parameters do for the base string.

string->**findLast**(*find::string, offset::integer=?, -length::integer=?, -patOffset::integer=?, -patLength::integer=?, -case::boolean=?*)
  This method is similar to **string->find** except that it returns the starting position of the *last* match found in the string object.

string->**contains**(*find::string, -case::boolean=?*)

string->**contains**(*find::regexp, -ignoreCase::boolean=?*)
  Takes a parameter that specifies a string or regular expression to match within the string object. It returns "true" if it finds a match, otherwise it will return "false".

  By default, string matching is not case-sensitive unless the optional **-case** parameter is passed to the method, but regular expression matching is case-sensitive unless the optional **-ignoreCase** parameter is passed to the method.

string->**get**(*position::integer*)
  Takes a parameter that specifies the position of the character to return.

string->**equals**(*find::string, case::boolean*)

string->**equals**(*find::string, -case::boolean=?*)
  This method is similar to the **==** equality operator. It returns "true" if the specified string is equivalent to the base string. This matching will not be case-sensitive unless passed the **-case** parameter.

string->**compare**(*find::string, -case::boolean=?*)

string->**compare**(*find::string, offset::integer, length::integer=?, patOffset::integer=?, patLength::integer=?, -case::boolean=?*)
  Takes a string pattern to compare with the string object and returns "0" if they are equal, "1" if the characters in the string are bitwise greater than the parameter, and "-1" if the characters in the string are bitwise less than the parameter. Comparisons are not case-sensitive unless passed the optional **-case** parameter.

  Optionally, the comparison can be made on smaller portions of the string object by passing the **offset** and **length** parameters, and smaller portions of the pattern by passing the **patOffset** and **patLength** parameters.

string->**beginsWith**(*find::string, case::boolean*)

string->**beginsWith**(*find::string, -case::boolean=?*)
  Takes a parameter that specifies a string to compare with the beginning of the string object value. It returns "true" if it matches the beginning, otherwise it will return "false".

  By default, string matching is not case-sensitive unless the optional **-case** parameter is passed to the method.

string->**endsWith**(*find::string, case::boolean*)

string->**endsWith**(*find::string, -case::boolean=?*)
  Takes a parameter that specifies a string to compare with the end of the string object value. It returns "true" if it matches the end, otherwise it will return "false".

  By default, string matching is not case-sensitive unless the optional **-case** parameter is passed to the method.

string->**getPropertyValue**(*position::integer, property::integer*)
  Takes a parameter that specifies the position of the character to inspect and a second parameter that specifies a Unicode property. It returns the Unicode property value for the indicated character and property. Unicode properties are defined in the Unicode Character Database[31] (UCD) and Unicode Technical Reports[32] (UTR).

---

[31] http://www.unicode.org/ucd/
[32] http://www.unicode.org/reports/

Lasso defines many methods that return values for these Unicode property names. All of these values have the **UCHAR_** prefix.

*string->***hasBinaryProperty**(*position::integer*, *property::integer*)
> Takes a parameter that specifies the position of the character to inspect and a second parameter that specifies a Unicode property. It returns "true" if the specified character has the specified property, otherwise it returns "false".

16.3.1 Find the Size of a String

The following example returns the number of characters of the string:

```
'Ralph is a red rhinoceros'->size
// => 25
```

16.3.2 Check for Lowercase Characters

The following example inspects each character in a string and counts the number of lowercase letters it contains:

```
local(num_lcase) = 0
local(my_string) = 'Ralph is a red rhinoceros'

loop(#my_string->size) => {
   #my_string->isLower(loop_count) ? #num_lcase++
}
#num_lcase

// => 20
```

16.3.3 Check the Beginning of a String

The following example checks to see if a string begins with "https:". If so, it displays "secure", otherwise it displays "insecure":

```
local(url) = 'https://secure.example.com'
#url->beginsWith('https:') ? 'secure' | 'insecure'

// => secure
```

16.3.4 Find a Substring

This example uses the **string->find** method to find and output each position in a string where there is an apostrophe:

```
local(my_string) = "Don't, it's not worth it!"
local(position)  = 0

while(#position < #my_string->size) => {^
   #position = #my_string->find(`'`, #position + 1)
   if(0 == #position) => {
      loop_abort
   }
   #position + '\n'
^}

// =>
```

```
// 4
// 10
```

16.3.5 Extract a Substring

The following example will pull the substring "red" out of the base string:

```
local(my_string) = 'Ralph is a red rhinoceros'
#my_string->sub(12, 3)

// => red
```

16.3.6 Extract a Specified Character Position

The following example uses **string->get** to return the last character in a string:

```
local(my_string) = 'Ralph is a red rhinoceros'
#my_string->get(#my_string->size)

// => s
```

16.4 String Manipulation Methods

The **string** type includes many member methods that can be used to modify or manipulate a string object in-place. These methods do not return a value, and instead modify the value of the string object. Many of these member methods are documented below.

string->**append**(*s::string*)

string->**append**(*obj::any*)
> Takes a single parameter that will be converted to a string and then concatenated to the end of the string object. It modifies the string object in-place, not returning any value.

string->**appendChar**(*i::integer*)
> Takes an integer that is the Unicode integer value in base 10 of a character. This character is then concatenated with the end of the string object. It modifies the string object in-place, not returning any value.

string->**remove**(*position::integer=?*)

string->**remove**(*position::integer*, *num::integer*)
> Takes a parameter that specifies the position of the first character to remove, defaulting to the first character. A second parameter can specify the number of characters to remove and defaults to removing all the characters from the starting position. It modifies the string object in-place, not returning any value.

string->**normalize()**
> Transforms a string object into its normalized form. It modifies the string object in-place, not returning any value. For more information on normalizing Unicode strings, see the Unicode Normalization FAQ[33] and Unicode Standard Annex #15[34].

---

[33] http://www.unicode.org/faq/normalization.html
[34] http://www.unicode.org/reports/tr15/

string->**foldCase**()
    Converts the characters in the string object to allow for case-insensitive comparisons. It modifies the string object in-place, not returning any value.

string->**trim**()
    Removes any whitespace from the beginning and end of a string. It modifies the string object in-place, not returning any value.

string->**reverse**()
    Changes the string object to the value of the base string in reverse order. It modifies the string object in-place, not returning any value.

string->**toLower**(*position::integer*)
    Takes a parameter that specifies the position of the character to modify. That character is converted to lowercase if possible. It modifies the string object in-place, not returning any value.

string->**toUpper**(*position::integer*)
    Takes a parameter that specifies the position of the character to modify. That character is converted to uppercase if possible. It modifies the string object in-place, not returning any value.

string->**toTitle**(*position::integer*)
    Takes a parameter that specifies the position of the character to modify. That character is converted to title case if possible. It modifies the string object in-place, not returning any value.

string->**lowercase**()
    Changes every possible character in a string to lowercase. It modifies the string object in-place, not returning any value.

string->**uppercase**()
    Changes every possible character in a string to uppercase. It modifies the string object in-place, not returning any value.

string->**titlecase**()

string->**titlecase**(*language::string, country::string*)
    Changes every possible word in a string to title case. It can optionally take a language code for the first parameter and a country code for the second to specify a locale to be used when performing this operation. It modifies the string object in-place, not returning any value.

string->**padLeading**(*tosize::integer, with::string=?*)
    Takes a parameter that specifies the target size of the string. If the base string object is smaller in size, then it changes the string by prepending a character to the start of the string until the string is the specified size. The character used for prepending defaults to a space, but can be set with the optional second parameter. It modifies the string object in-place, not returning any value.

string->**padTrailing**(*tosize::integer, with::string=?*)
    Takes a parameter that specifies the target size of the string. If the base string object is smaller in size, then it changes the string by appending a character to the end of the string until the string is the specified size. The character used for appending defaults to a space, but can be set with the optional second parameter. It modifies the string object in-place, not returning any value.

string->**removeLeading**(*find::string*)

string->**removeLeading**(*find::regexp*)
    Takes either a string or a regular expression and removes all specified matches from the beginning of the string. It keeps removing until the beginning of the string no longer matches the specified pattern. It modifies the string object in-place, not returning any value.

string->**removeTrailing**(*find::string*)
    Takes a string and removes all matches specified from the end of the string. It keeps removing until the end of the string no longer matches the specified parameter. It modifies the string object in-place, not returning any value.

string->merge(*where::integer, what::string, offset::integer=?, length::integer=?***)**
> Merges a specified string into the base string. It requires the first parameter to specify the position in the base string for the merge to take place and a second parameter that specifies the string to merge into the base string. It modifies the string object in-place, not returning any value.

> Optionally, a third parameter can specify the starting position of the passed string to be used in the merge and a fourth can specify the number of characters to after the offset to be merged from the passed string.

string->replace(*find::string, replace::string, -case::boolean=?***)**

string->replace(*find::regexp, replace=?, ignoreCase=?***)**
> Takes either a string or a regular expression and replaces all matches found in the string object value with the specified replacement. For regular expression matches, the replacement string can be specified for this method, or it will use the replacement string of the **regexp** object. It modifies the string object in-place, not returning any value.

> When using a regular expression, the method defaults to a case-sensitive matching unless otherwise specified by the third parameter. When using a string for matching, the default is the reverse: it uses case-insensitive matching unless otherwise specified by the third parameter.

16.4.1 Append Data to a String

This example uses the **string->append** method to add a trailing slash to a directory path if one does not already exist:

```
local(dir_path) = '/var/lasso/home'

if(not #dir_path->endsWith('/')) => {
    #dir_path->append('/')
}
#dir_path

// => /var/lasso/home/
```

16.4.2 Remove Whitespace Around a String

This example uses the **string->trim** method to remove whitespace from the beginning and end of the string and then outputs the string:

```
local(my_string) = '\n    Ralph the Ringed Rhino   \n\n'
#my_string->trim
#my_string

// => Ralph the Ringed Rhino
```

16.4.3 Ensure All Characters are Lowercase

This example takes a string and converts all the characters to lowercase and then outputs the changed string:

```
local(my_string) = 'Ralph the Ringed Rhino races red radishes in THE RINK.'
#my_string->lowercase
#my_string

// => ralph the ringed rhino races red radishes in the rink.
```

16.4.4 Remove a Pattern from the End of a String

This example removes all the trailing commas from the string:

```
local(my_string) = 'First, Second, Fifth,,,'
#my_string->removeTrailing(',')
#my_string

// => First, Second, Fifth
```

16.5 String Encoding Methods

string->**hash()**
> Returns a simple hash of the string object.

string->**unescape()**
> Returns a string with any escape sequences (a sequence beginning with a backslash) in the base string object replaced with their literal Unicode equivalents. This is the same escape process Lasso does for non-ticked string literals.

string->**encodeHtml()**

string->**encodeHtml(**_linebreaks::boolean, ignorechars::boolean_**)**
> Returns a string with any reserved, illegal, or extended ASCII characters in the base string object converted to their equivalent HTML entity. This replacement can be modified by passing two boolean parameters. If the first parameter is set to "true", then line breaks are encoded. If the second parameter is set to "true", then the following characters are not encoded: **" & ' < >** (double quotation mark, ampersand, single quotation mark, less than or left angle bracket, and greater than or right angle bracket, respectively).

string->**decodeHtml()**
> Returns a string with any HTML entities in the base string object converted to their Unicode equivalent. This is the opposite of the **string->encodeHtml** method.

string->**encodeXml()**
> Returns a new string of the base string object with any reserved or illegal XML characters encoded into their equivalent XML entity.

string->**decodeXml()**
> Returns a string from the base string object with any XML entities converted to their Unicode equivalent. This is the opposite of the **string->encodeXml** method.

string->**encodeHtmlToXml()**
> Returns a string from the base string object with any HTML encoded entities converted to XML encoding.

string->**asBytes(**_encoding::string=?_**)**
> Returns the value of the base string as a bytes object. By default, UTF-8 encoding is used for this conversion, but any encoding can be specified as a string parameter to this method.

string->**encodeSql()**
> Returns the value of the base string with any illegal characters for MySQL data sources properly escaped.

string->**encodeSql92()**
> Returns the value of the base string with any illegal characters for SQL-92–compliant databases properly escaped. Not for use with MySQL.

16.5.1 Convert Escape Sequences

The following example creates a string with escape sequences using a ticked string literal so that Lasso won't automatically unescape them. It then outputs the string before calling **string->unescape** and then shows the result of calling **string->unescape**:

```
local(my_string) = `Chinese Character: \u4E26`
#my_string + '\n'
#my_string->unescape

// =>
// Chinese Character: \u4E26
// Chinese Character: 並
```

16.5.2 Encode HTML Entities

The following example uses **string->encodeHtml** to return a string with the special HTML entities encoded:

```
local(my_string) = '<>&'
#my_string->encodeHtml

// => &lt;&gt;&amp;
```

16.5.3 Encode for Use in MySQL

The following example returns a string whose quotes have been encoded for use in a MySQL SQL statement:

```
local(my_string) = "Don't forget to encode"
#my_string->encodeSql

// => Don\'t forget to encode
```

16.6 String Iteration Methods

string->**forEachCharacter()**
> Takes a capture block and executes that block once for every character in the base string. The character can be accessed in the capture block through the special local variable **#1**.

string->**forEachWordBreak()**
> Takes a capture block and executes that block once for every word in the base string. The word can be accessed in the capture block through the special local variable **#1**.

string->**forEachLineBreak()**
> Takes a capture block and executes that block once for every substring that would be generated by splitting the base string object on a line break. Every line break character is recognized: **"\r"**, **"\n"**, and **"\r\n"**. Each of the substrings can be accessed in the capture block through the special local variable **#1**.

string->**forEachMatch**(*exp::string*)

string->**forEachMatch**(*exp::regexp*)
> Takes a capture block and executes that block once for every specified match in the base string object. Matches can be specified as either **string** or **regexp** objects. The match can be accessed in the capture block through the special local variable **#1**.

string->**eachCharacter**()

> Returns an **eacher** that can be used in conjunction with query expressions to inspect and perform complex operations on every character in the base string object.

string->**eachWordBreak**()

> Returns an **eacher** that can be used in conjunction with query expressions to inspect and perform complex operations on every word in the base string object.

string->**eachLineBreak**()

> Returns an **eacher** that can be used in conjunction with query expressions to inspect and perform complex operations on every line in the base string object.

string->**eachMatch**(*exp::string*)

string->**eachMatch**(*exp::regexp*)

> Returns an **eacher** that can be used in conjunction with query expressions to inspect and perform complex operations on every specified match in the base string object. Matches can be specified as either **string** or **regexp** objects.

16.6.1 Iterate Over Lines

The following example takes a string with multiple lines and runs the lines of the string together with slashes, storing the result in the variable "quoted_poem". It removes the trailing slash at the end and then displays the variable "quoted_poem" in quotes.

```
local(poem) = '\
An old silent pond...
A frog jumps into the pond,
Splash! Silence again.'

local(quoted_poem) = ''
#poem->forEachLineBreak => {
    #quoted_poem->append(#1 + '/')
}
#quoted_poem->removeTrailing('/')
'"' + #quoted_poem + '"'

// => "An old silent pond.../A frog jumps into the pond,/Splash! Silence again."
```

16.6.2 Iterate Over Words

The following example takes a string and inspects each word using a query expression. If the word starts with the letter "r" then it will transform it to uppercase. The query expression selects each word, allowing us to create a staticarray of words.

```
local(my_string) = 'Ralph is a red rhinoceros.'
(
    with word in #my_string->eachWordBreak
    select (#word->beginsWith('r') ? #word->uppercase& | #word)
)->asStaticArray

// => staticarray(RALPH, is, a, RED, RHINOCEROS.)
```

16.6.3 Iterate Over a Specified Regular Expression Match

The following example uses **string->eachMatch** with a **regexp** object to find every vowel in a string, where the local variable "vowels" is used to count the number of each vowel in the string.

```
local(my_string) = 'ralph is a red rhinoceros.'
local(vowels)    = map('a'=0, 'e'=0, 'i'=0, 'o'=0, 'u'=0)

with letter in #my_string->eachMatch(regexp(`[aeiouAEIOU]`))
do #vowels->find(#letter)++
#vowels

// => map(a = 2, e = 2, i = 2, o = 2, u = 0)
```

16.7 String Export Methods

string->**split**(*find::string*)
> Returns an array with elements created by breaking up the string on the specified string. If an empty string is specified, each element of the array is a single character of the string.

string->**values**()
> Returns an array, each element of which is one character of the string.

string->**keys**()
> Returns a **generateSeries** from 1 to the number of characters in the string, or an empty **generateSeries** if the string is empty.

16.7.1 Split a String Into an Array

The following example creates an array by splitting a string on a comma:

```
local(my_string) = '1,3,9,f,g'
#my_string->split(',')

// => array(1, 3, 9, f, g)
```

Byte Streams

Binary data in Lasso is stored and manipulated using the **bytes** type. This chapter details the operators and methods that can be used to manipulate binary data.

Tip: The **bytes** type is often used in conjunction with the **string** type to convert binary data between different character encodings (e.g. UTF-8, ISO-8859-1). See the *Strings* chapter for more information about the **string** type.

17.1 Creating Bytes Objects

While string data in Lasso is processed as one- to four-byte Unicode characters, the **bytes** type can be used to represent raw strings of single bytes, which is often referred to as a *byte stream* or *binary data*.

Lasso's methods return a byte stream in the following situations:

- The **field** method returns a byte stream from MySQL "BLOB" fields.

- The **bytes** creator method can be used to allocate a new byte stream.

- The **web_request->param** methods return a byte stream.

- Other methods that return or require binary data as outlined in their documentation.

type **bytes**

bytes()

bytes(*initial::integer*)

bytes(*copy::bytes*)

bytes(*import::string, encoding::string=?*)

bytes(*doc::pdf_doc*)
> Allocates a byte stream. Can be used to convert a **string** or **pdf_doc** type to a **bytes** type, or to instantiate a new **bytes** object. Accepts one optional parameter that can specify the initial size in bytes for the stream; or specify the **string**, **pdf_doc**, or **bytes** object to convert to a new **bytes** object. If converting a **string** object, it can accept an optional second parameter to specify the encoding of the string.

17.1.1 Instantiate a New Bytes Object

Use the **bytes** creator method. The example below creates an empty bytes object with a size of 1024 bytes:

```
local(obj) = bytes(1024)
```

17.1.2 Convert String Data to a Bytes Object

Use the **bytes** creator method. The following example converts a string to a bytes object:

```
local(obj) = bytes('This is some text')
```

17.2 Bytes Inspection Methods

Byte streams are similar to strings and support many of the same member methods. Additionally, byte streams support a number of member methods that make it easier to deal with binary data. The most common methods are outlined below.

bytes->**length()**
> *Deprecated since version 9.0:* Use **bytes->size** instead.

bytes->**size()**
> Returns the number of bytes contained in the bytes object.

bytes->**get**(*position::integer*)
> Returns a single byte from the stream. Requires a parameter specifying which byte to fetch.

bytes->**getRange**(*position::integer*, *num::integer*)
> Returns a range of bytes from the byte stream. Requires two parameters: the first specifies the byte position to start from, and the second specifies how many bytes to return.

bytes->**find**(*find::bytes*, *position::integer=?*, *length::integer=?*, *patPosition::integer=?*, *patLength::integer=?*)

bytes->**find**(*find::string*, *position::integer=?*, *length::integer=?*, *patPosition::integer=?*, *patLength::integer=?*)
> Requires either a byte stream or string sequence as the first parameter. Returns the position of the beginning of the sequence being searched for within the bytes object, or "0" if the sequence is not contained within the object. Four optional integer parameters (position, length, parameter position, parameter length) indicate position and length limits that can be applied to the instance and the parameter sequence.

bytes->**contains**(*find*)
> Returns "true" if the byte stream contains the specified sequence.

bytes->**beginsWith**(*find::string*)

bytes->**beginsWith**(*find::bytes*)
> Returns "true" if the byte stream begins with the specified sequence.

bytes->**endsWith**(*find::string*)

bytes->**endsWith**(*find::bytes*)
> Returns "true" if the byte stream ends with the specified sequence.

bytes->**sub**(*position::integer*, *num::integer=?*)
> Returns a specified slice of the byte stream. Requires an integer parameter that specifies the index into the byte stream to start taking the slice from. An optional second integer parameter can specify the number of bytes to slice out of the byte stream. If the second parameter is not specified, then all of the rest of the byte stream is taken.

bytes->**marker()**

bytes->**position()**
> Returns the current position at which imports will occur in the byte stream.

bytes->**split**(*find::string*)

bytes->**split**(*find::bytes*)
> Returns an array of bytes objects using the specified sequence as the delimiter to split the byte stream. If the delimiter

provided is an empty byte stream or string, the byte stream is split on each byte, so the returned array will have each byte as one of its elements.

bytes->**exportString**(*encoding::string*)
> Returns a string representing the byte stream. Accepts a single parameter specifying the character encoding (e.g. "ISO-8859-1", "UTF-8") for the export.

bytes->**export8bits()**
> Returns the first byte as an integer.

bytes->**export16bits()**
> Returns the first 2 bytes as an integer.

bytes->**export32bits()**
> Returns the first 4 bytes as an integer.

bytes->**export64bits()**
> Returns the first 8 bytes as an integer.

17.2.1 Return the Size of a Byte Stream

Use the **bytes->size** method. The example below returns the size of a bytes object:

```
local(obj) = bytes('abc...')
#obj->size

// => 6
```

17.2.2 Return a Single Byte from a Byte Stream

Use the **bytes->get** method. An integer parameter specifies the index of the byte to return. Note that this method returns an integer, not a fragment of the original data (such as a string character):

```
local(obj) = bytes('hello world')
#obj->get(2)

// => 101
```

17.2.3 Find a Value Within a Byte Stream

Use the **bytes->find** method. The example below returns the starting byte number of the value **'rhino'**, which is contained within the byte stream:

```
bytes('running rhinos risk rampage')->find('rhino')
// => 9
```

17.2.4 Determine If a Byte Stream Contains a Value

Use the **bytes->contains** method. The example below returns "true" if the value **'Rhino'** is contained within the byte stream. Note that in this example it returns "false" because the bytes of **'rhino'** are a different sequence then the bytes of **'Rhino'**.

```
bytes('running rhinos risk rampage')->find('Rhino')
// => false
```

17.2.5 Export a String from a Byte Stream

Use the **bytes->exportString** method. The following example exports a string using UTF-8 encoding:

```
local(obj) = bytes('This is a string')
#obj->exportString('UTF-8')

// => This is a string
```

17.3 Bytes Manipulation Methods

Calling the following methods will modify the bytes object.

bytes->**setSize**(*num::integer*)
> Sets the byte stream size to the specified number of bytes.

bytes->**setRange**(*what::bytes*, *where::integer=?*, *whatStart::integer=?*, *whatLen::integer=?*)
> Sets a range of characters within a byte stream. Requires one parameter for the binary data to be inserted. Optional second, third, and fourth parameters specify the integer offset into the byte stream to insert the new data, and the offset and length of the new data to be inserted, respectively.

bytes->**replace**(*find::bytes*, *replace::bytes*)
> Replaces all instances of a value within a byte stream with a new value. Requires two parameters: the first parameter is the value to find, and the second parameter is the value with which to replace the first parameter.

bytes->**remove**()

bytes->**remove**(*position::integer*, *num::integer*)
> Removes bytes form a byte stream. When passed without a parameter, it removes all bytes, setting the object to an empty bytes object. In its second form, it requires an offset into the byte stream and the number of bytes to remove starting from there.

bytes->**removeLeading**(*find::bytes*)
> Removes all occurrences of the specified sequence from the beginning of the byte stream. Requires one parameter specifying the data to be removed.

bytes->**removeTrailing**(*find::bytes*)
> Removes all occurrences of the parameter sequence from the end of the byte stream. Requires one parameter specifying the data to be removed.

bytes->**append**(*rhs::bytes*)

bytes->**append**(*rhs::string*)
> Appends the specified data to the end of the byte stream. Requires one parameter specifying the data to append.

bytes->**trim**()
> Removes all whitespace ASCII characters from the beginning and the end of the byte stream.

bytes->**setPosition**(*i::integer*)
> Sets the current position within the byte stream. Requires a single integer parameter.

bytes->**importString**(*s::string*, *enc::string=?*)
> Imports a string parameter. A second parameter can specify the character encoding (e.g. "ISO-8859-1", "UTF-8") to use for the import.

bytes->**import8bits**(*i::integer*)
> Imports the first byte of an integer parameter.

bytes->**import16bits**(*i::integer*)
> Imports the first 2 bytes of an integer parameter.

bytes->**import32bits**(*i::integer*)
> Imports the first 4 bytes of an integer parameter.

bytes->**import64bits**(*i::integer*)
> Imports the first 8 bytes of an integer parameter.

bytes->**swapBytes()**
> Swaps the position of every pair of bytes (e.g. a byte stream of `'father'` becomes `'afhtre'`).

17.3.1 Add a String to a Byte Stream

Use the **bytes->append** method. The following example adds the string `'I am'` to the end of a byte stream:

```
local(obj) = bytes
#obj->append('I am')
```

17.3.2 Find and Replace Values in a Byte Stream

Use the **bytes->replace** method. The following example finds the string `'Blue'` and replaces it with the string `'Green'` within the byte stream:

```
local(colors) = bytes('Blue Red Yellow')
#colors->replace('Blue', 'Green')
```

17.3.3 Import a String Into a Byte Stream

Use the **bytes->importString** method. The following example imports a string using ISO-8859-1 encoding:

```
local(obj) = bytes('This is a string')
#obj->importString('This is another string', 'ISO-8859-1')
```

Math

Numbers in Lasso are stored and manipulated using the **decimal** and **integer** types. This chapter details the operators and methods that can be used to manipulate decimal and integer values and to perform mathematical operations. Each of these methods is described in detail in the sections that follow; however, the Lasso Reference is the primary documentation source for Lasso operators and methods.

18.1 Creating Integer Objects

The **integer** type represents whole number values. Basically, zero and any positive or negative number that does not contain a decimal point is an integer value in Lasso. Examples include **-123** or **456**. Integer objects may also be expressed in hexadecimal notation such as **0x1A** or **0xff**.

type **integer**

integer()

integer(*obj::any***)**
> The creator method for integer converts any object to an integer. If the type for the object being converted does not easily represent an integer, then the integer zero will be returned.

18.1.1 Explicit Integer Conversion

Strings that contain numeric data can be converted to integer objects using the **integer** creator method. The string must start with a numeric value. In the following examples the integer **123** is the result of each explicit conversion. Only the first integer found in the string **'123 and then 456'** is recognized:

```
integer('123')
// => 123

integer('123 and then 456')
// => 123
```

Decimals that are converted to an integer are rounded to the nearest integer:

```
integer(123.0)
// => 123

integer(123.999)
// => 124
```

18.2 Formatting Integer Objects

Integer objects can be formatted for display using the **integer->asString** method detailed below.

Note: Integers and decimals have no state, so they cannot carry around formatting information. The **integer->asString** method replaces the functionality of the **integer->setFormat** method from previous versions of Lasso.

---

integer->asString(*-hexadecimal::boolean=?,   -padding::integer=?,   -padChar::string=?,   -padRight::boolean=?,*
                          *-groupChar::string=?***)**
>   Returns a string representation of the integer value formatted as specified by the parameters passed to the method. If no parameters are passed to the method, the string will be the integer value output in base 10.

>   Parameters

>   - **-hexadecimal** (***boolean***) – If set to "true", the integer will output in hexadecimal notation.

>   - **-padding** (***integer***) – Specifies the desired length for the output. If the formatted number is less than this length then the number is padded.

>   - **-padChar** (***string***) – Specifies the character to insert if padding is required. Defaults to a space.

>   - **-padRight** (***boolean***) – Set to "true" to pad the right side of the output. By default, padding is appended to the left side of the output.

>   - **-groupChar** (***string***) – Specifies the character to use for thousands grouping. Defaults to empty.

18.2.1 Format an Integer as a Hexadecimal Value

The following example will create a variable with an integer value and then output that value in base 16:

```
local(my_int) = 255
#my_int->asString(-hexadecimal)

// => ff
```

18.3 Integer Bitwise Methods

Bitwise operations can be performed with Lasso's integer objects. These operations can be used to examine and manipulate binary data. They can also be used for general purpose binary set operations.

Integer literals in Lasso can be specified using hexadecimal notation. This can greatly aid in constructing literals for use with the bitwise operation. For example, **0xff** is the integer literal **255**.

integer->bitAnd(*i::integer***)**
>   Performs a bitwise "and" operation between each bit in the base integer and the integer parameter, returning the result.

integer->bitOr(*i::integer***)**
>   Performs a bitwise "or" operation between each bit in the base integer and the integer parameter, returning the result.

integer->bitXOr(*i::integer***)**
>   Performs a bitwise "exclusive or" operation between each bit in the base integer and the integer parameter, returning the result.

integer->bitNot()
>   Returns the result of flipping every bit in the base integer.

integer->bitShiftLeft(*i::integer***)**
>   Returns the result of shifting the bits in the base integer left by the number specified in the integer parameter.

integer->bitShiftRight(*i::integer***)**
>   Returns the result of shifting the bits in the base integer right by the number specified in the integer parameter.

integer->**bitClear**(*i::integer*)
>   Returns the result of clearing the bit specified in the integer parameter.

integer->**bitFlip**(*i::integer*)
>   Returns the result of flipping the bit specified in the integer parameter.

integer->**bitSet**(*i::integer*)
>   Returns the result of setting the bit specified in the integer parameter.

integer->**bitTest**(*i::integer*)
>   Returns "true" if the bit specified in the integer parameter is 1, otherwise returns "false".

Note: Integers are by-value objects and are immutable, so it is not possible to change their value. This is in contrast to previous versions of Lasso, where these bit methods modified the integer in-place.

18.3.1 Perform a Bitwise OR

In the following example the boolean "or" of **0x02** and **0x04** is calculated and returned in hexadecimal notation:

```
local(bit_set) = 0x02
#bit_set->bitOr(0x04)->asString(-hexadecimal)

// => 6
```

18.3.2 Shift Bits to the Left

In the following example, **0x02** is shifted left by three places and output in hexadecimal notation:

```
local(bit_set) = 0x02
#bit_set = #bit_set->bitShiftLeft(3)
#bit_set->asString(-hexadecimal)

// => 10
```

18.3.3 Set and Test a Specified Bit

In the following example, the second bit of an integer is set and then tested:

```
local(bit_set) = 0
#bit_set = #bit_set->bitSet(2)
#bit_set->bitTest(2)

// => true
```

18.4 Creating Decimal Objects

The **decimal** type represents real or floating point numbers. Basically, 0.0 or any positive or negative number that contains a decimal point is a decimal object in Lasso. Examples include **-123.0** and **456.789**. Decimal values can also be written in exponential notation such as **1.23e2** which is equivalent to **1.23** times **10^2** or **123.0**.

type **decimal**

```
decimal()
```

```
decimal(i::integer)
```

```
decimal(d::decimal)
```

```
decimal(s::string)
```

```
decimal(b::bytes)
```

```
decimal(n::null)
```

```
decimal(n::void)
```

> The creator methods for the **decimal** type converts **integer**, **string**, **bytes**, **null**, and **void** objects to a decimal value.
>
> The precision of a decimal value when converted to a string is always displayed as six decimal places even though the actual precision of the number may vary based on the size of the number and its internal representation. The output precision of decimal numbers can be controlled using the **decimal->asString** method described later in this chapter.

18.4.1 Implicit Decimal Conversion

Integer values are converted to decimal values automatically if they are used as a parameter to an arithmetical operator in conjunction with a decimal value. The following example shows how the integer **123** is automatically converted to a decimal value because the other parameter of the **+** operator is the decimal value **456.0**:

```
456.0 + 123
// => 579.0
```

The following example shows how a variable with a value of "123" is automatically converted to a decimal value:

```
local(number) = 123
456.0 + #number

// => 579.0
```

18.4.2 Explicit Decimal Conversion

Strings containing numeric data can be converted to the **decimal** type using the **decimal** creator method. The string must start with a numeric value. In the following examples the number **123.0** is the result of each explicit conversion. Only the first decimal value found in the string **'123 and then 456'** is recognized:

```
decimal('123')
// => 123.0
```

```
decimal('123.0')
// => 123.0
```

```
decimal('123 and then 456')
// => 123.0
```

Integers that are converted to decimals simply have a decimal point appended. The value of the number does not change.

```
decimal(123)
// => 123.0
```

18.5 Formatting Decimal Objects

Decimal objects can be formatted for display using the **decimal->asString** method detailed below.

Note: Integers and decimals have no state, so they cannot carry around formatting information. The **decimal->asString** method replaces the functionality of the **decimal->setFormat** method from previous versions of Lasso.

decimal->asString(-*decimalChar::string=?*,   -*groupChar::string=?*,   -*precision::integer=?*,   -*scientific::boolean=?*,
       -*padding::integer=?*, -*padChar::string=?*, -*padRight::boolean=?***)**
    Returns a string representation of the decimal value formatted as specified by the parameters passed to the method. If
    no parameters are passed to the method, the string will be the decimal value with six places of precision.

    Parameters

    - **-decimalChar** (*string*) – The character that should be used for the decimal point. It defaults to a period.

    - **-groupChar** (*string*) – The character that should be used for thousands grouping. Defaults to an empty
      string.

    - **-precision** (*integer*) – The number of places after the decimal point that should be output. The default
      is 6.

    - **-scientific** (*boolean*) – Set to "true" to force output in exponential notation. The default is "false", so
      decimals are only output in exponential notation if required.

    - **-padding** (*integer*) – Specifies the desired length for the output. If the formatted number is less than
      this length then the number is padded.

    - **-padChar** (*string*) – Specifies the character that will be inserted if padding is required. Defaults to a
      space.

    - **-padRight** (*boolean*) – Set to "true" to pad the right side of the output. By default, padding is prepended
      to the left side of the output.

18.5.1 Format a Decimal Number as U.S. Currency

The following example outputs a decimal value as if it were U.S. currency by setting the precision to "2". For readability, it also
sets a comma as the grouping character.

```
local(dollar_amt) = 1234.56
#dollar_amt->asString(-precision=2, -groupChar=',')

// => 1,234.56
```

18.6 Arithmetical Operations

The easiest way to manipulate integer and decimal objects is to use arithmetical operators. The sections below detail all the
operators that can be used with integer and decimal values. See the *Operators* chapter for further documentation of how
these operators are used.

18.6.1 Basic Arithmetical Operators

Each basic operator takes two parameters, one to its left and the other to its right. If either of the parameters is a decimal
then the result will be a decimal value. Some of the operators can also be used to perform string operations. If either of the

parameters is a string value then the string operation defined by the operator will be performed rather than the arithmetical operation.

Table 18.1: Arithmetical Operators

Operator	Name	Description
+	Addition	Adds two parameters.
-	Subtraction	Subtracts the right parameter from the left parameter.
*	Multiplication	Multiplies two parameters.
/	Division	Divides the left parameter by the right parameter.
%	Modulo	Produces the remainder of dividing the left parameter by the right parameter.

Using Arithmetical Operators

Two numbers can be added using the **+** operator. The output will be a decimal value if either of the parameters are a decimal value.

```
100 + 50
// => 150
```

```
100 + -12.5
// => 87.500000
```

The difference between two numbers can be calculated using the **-** operator. The output will be a decimal value if either of the parameters are a decimal value. Note that in the second instance, when subtracting a negative number, the two **-** operators must be separated by a space so as not to be confused with the **--** operator.

```
100 - 50
// => 50
```

```
100 - -12.5
// => 112.500000
```

Two numbers can be multiplied using the ***** operator. The output will be a decimal value if either of the parameters are a decimal value.

```
100 * 50
// => 5000
```

```
100 * -12.5
// => -1250.000000
```

18.6.2 Arithmetical Assignment Operators

Each of the operators takes two parameters, one to its left and the other to its right. The first parameter must be a variable that holds an integer, decimal, or string. The second parameter can be an integer, decimal, or string literal. The result of the operation is calculated and then stored back in the variable specified as the left-hand parameter.

Table 18.2: Arithmetical Assignment Operators

Operator	Name	Description
=	Assign	Assigns the right parameter to the variable designated by the left parameter.
+=	Add-assign	Adds the right parameter to the value of the left parameter and assigns the result to the variable designated by the left parameter.
-=	Subtract-assign	Subtracts the right parameter from the value of the left parameter and assigns the result to the variable designated by the left parameter.
*=	Multiply-assign	Multiplies the value of the left parameter by the value of the right parameter and assigns the result to the variable designated by the left parameter.
/=	Divide-assign	Divides the value of the left parameter by the value of the right parameter and assigns the result to the variable designated by the left parameter.
%=	Modulo-assign	Assigns the value of the left parameter modulo the right parameter to the variable designated by the left parameter.

Using Arithmetical Assignment Operators

A variable can be assigned a new value using the assignment operator (=). The following example shows how to define an integer variable and then set it to a new value, which is then output:

```
local(my_variable) = 100
#my_variable = 123456
#my_variable

// => 123456
```

A variable can be used as a collector by adding new values using the += operator. The following example shows how to define an integer variable and then add several values to it, then output the final value:

```
local(my_variable) = 100
#my_variable += 123
#my_variable += -456
#my_variable

// => -233
```

18.6.3 Arithmetical Equality Operators

Each of the arithmetical equality operators takes two parameters, one on its left and one on its right.

Table 18.3: Arithmetical Equality Operators

Operator	Name	Description
==	Equal	Returns "true" if the parameters are equal.
!=	Not equal	Returns "true" if the parameters are not equal.
<	Less	Returns "true" if the left parameter is less than the right parameter.
<=	Less or equal	Returns "true" if the left parameter is less than or equal to the right parameter.
>	Greater	Returns "true" if the left parameter is greater than the right parameter.
>=	Greater or equal	Returns "true" if the left parameter is greater than or equal to the right parameter.

Using Arithmetical Equality Operators

Two numbers can be compared for equality using the equality (`==`) and inequality (`!=`) operators. The result is a boolean "true" or "false". Integers are automatically converted to decimal values when compared with decimals.

```
100 == 123
// => false

100.0 != -123.0
// => true

100 == 100.0
// => true

100.0 != -123
// => true
```

Numbers can be compared using the relative equality operators (`<`, `<=`, `>`, `>=`). The result is a boolean "true" or "false".

```
-37 > 0
// => false

100 < 1000.0
// => true
```

18.7 Basic Math Methods

Lasso contains many methods that can be used to perform mathematical functions. The functionality of some of these methods overlaps the functionality of the mathematical operators. It is recommended that you use the equivalent operator when one is available.

math_abs(*value*)
> Returns the absolute value of the parameter.

math_add(*value*, ...)
> Returns the sum of all parameters.

math_ceil(*value*)
> Returns the next integer greater than the parameter.

math_convertEuro(*value*, *euroTo::string*)
> Converts between the Euro and other European Union currencies.

math_div(*value*, ...)
> Divides each of the parameters in order from left to right.

math_floor(*value*)
> Returns the next integer less than the parameter.

math_max(*value*, ...)
> Returns the maximum of all parameters.

math_min(*value*, ...)
> Returns the minimum of all parameters.

math_mod(*value*, *factor*)
> Returns the value of the first parameter modulo the second parameter.

math_mult(*value, …*)
> Returns the product of multiplying each of the parameters together.

math_random() → decimal

math_random(*upper::integer, lower=0*) → integer

math_random(*upper::decimal, lower=0.0*) → decimal

math_random(*-upper, -lower*) → integer
> If called with no parameters, returns a random number between 0.0 and 1.0. This method can also take two parameters, with the first as the upper bound for the random number, and the second as the lower bound. If the first parameter is an integer, an integer will be returned, and if it is a decimal, then a decimal will be returned.
>
> This method can also be called with **-upper** and **-lower** keyword parameters and will then return an integer value regardless of the types of the objects passed as parameters.
>
> When returning integer values **math_random** will return a maximum 32-bit value. The range of returned integers is approximately between +/- 2,000,000,000.

math_rint(*value*)
> Returns a decimal value rounded to the nearest integer.

math_roman(*value*)
> Returns a string representing the number passed in as a Roman numeral.

math_round(*value, factor*)
> Rounds the first parameter to the precision specified by the second parameter.

18.7.1 Using Basic Math Methods

The following are all examples of using basic math methods to calculate the results of various mathematical operations:

```
math_add(1, 2, 3, 4, 5)
// => 15

math_add(1.0, 100.0)
// => 101.000000

math_sub(10, 5)
// => 5

math_div(10, 9)
// => 1

math_div(10, 8.0)
// => 1.250000

math_max(100, 200)
// => 200
```

18.7.2 Round to an Integer

Decimals can be rounded to an integer using the **integer** creator method, the **math_floor** method to round to the next lowest integer, or the **math_ceil** method to round to the next highest integer:

```
integer(37.6)
// => 38

math_floor(37.6)
// => 37

math_ceil(37.6)
// => 38
```

18.7.3 Round to Nearest Integer

Decimals can be rounded to the nearest integer using the **math_rint** method. This method rounds the decimal, but does not convert it to an integer:

```
math_rint(37.6)
// => 38.000000
```

18.7.4 Round to a Specified Precision

Numbers can be rounded to arbitrary precision using the **math_round** method with a decimal parameter. The second parameter should be of the form **0.01**, **0.0001**, **0.000001**, etc.

```
math_round(3.1415926, 0.0001)
// => 3.141600

math_round(3.1415926, 0.001)
// => 3.142000

math_round(3.1415926, 0.01)
// => 3.140000

math_round(3.1415926, 0.1)
// => 3.100000
```

Numbers can be rounded to an even multiple of another number using the **math_round** method with an integer parameter. The integer parameter should be a power of 10.

```
math_round(1463, 1000)
// => 1000.000000

math_round(1463, 100)
// => 1500.000000

math_round(1463, 10)
// => 1460.000000
```

Note: If a rounded result needs to be shown to the user but the actual value stored in a variable does not need to be rounded, then either the **integer->asString** or **decimal->asString** method can be used to alter how the number is displayed. See the documentation of these methods earlier in this chapter for more information.

18.7.5 Return a Random Integer Value

In the following example a random number between **1** and **100** is returned. The random number will be different each time the page is loaded.

```
math_random(100, 1)
// => 55
```

18.7.6 Return a Random Decimal Value

In the following example a random decimal number between **0.0** and **1.0** is returned. The random number will be different each time the page is loaded.

```
math_random(1.0, 0.0)
// => 0.532773
```

18.7.7 Return a Random Hex Color Value

In the following example a random hexadecimal color code is returned. The random number will be different each time the page is loaded. The range is from **0** to **255** to return two-digit hexadecimal values between **00** and **FF**.

```
local(color) = "#" +
    math_random(255,0)->asString(-hexadecimal, -padding=2, -padChar='0') +
    math_random(255,0)->asString(-hexadecimal, -padding=2, -padChar='0') +
    math_random(255,0)->asString(-hexadecimal, -padding=2, -padChar='0')
'<span style="color: ' + #color + ';">Color</span>'

// => <span style="color: #e64b32;">Color</span>
```

18.8 Trigonometry and Advanced Math Methods

Lasso provides a number of methods for calculating square roots, logarithms, and exponents, and performing trigonometric functions.

math_acos(*value*)
> Arc Cosine. Returns the value of taking the arc cosine of the passed parameter. The return value is in radians between "0" and "π".

math_asin(*value*)
> Arc Sine. Returns the value of taking the arc sine of the passed parameter. The return value is in radians between "-π/2" and "π/2".

math_atan(*value*)
> Arc Tangent. Returns the value of taking the arc tangent of the passed parameter. The return value is in radians between "-π/2" and "π/2".

math_atan2(*value*, *factor*)
> Arc Tangent of a Quotient. Returns the value of taking the angle in radians between the x-axis and coordinates passed to it. The return value is in radians between "-π" and "π".

math_cos(*value*)
> Cosine. Returns the value of taking the cosine of the passed parameter.

math_sin(*value*)
> Sine. Returns the value of taking the sine of the passed parameter.

math_tan(*value*)
> Tangent. Returns the value of taking the tangent of the passed parameter.

math_exp(*value*)
> Natural Exponent. Returns the value of taking *e* raised to the specified power.

math_ln(*value*)

math_log(*value*)
> Natural Logarithm. Returns the value of taking the natural log of the passed parameter.

math_log10(*value*)
> Base 10 Logarithm. Returns the value of taking the base 10 log of the passed parameter.

math_pow(*value*, *factor*)
> Exponent. Returns the value of taking the first parameter and raising it to the value of the second parameter.

math_sqrt(*value*)
> Square Root. Returns the positive square root of the passed parameter. The parameter passed to this method must be positive.

18.8.1 Using Advanced Math Methods

The following are examples of using some of these advanced math methods to calculate various mathematical operations:

```
math_pow(3, 3)
// => 27

math_sqrt(100.0)
// => 10.000000

math_acos(-1.0)
// => 3.141593

math_exp(math_log(5))
// => 5.000000
```

Date and Duration

This chapter introduces the **date** and **duration** types in Lasso. *Dates* are objects that represent a calendar date and/or clock time. *Durations* are objects that represents a length of time in hours, minutes, and seconds. Date and duration objects can be manipulated using operators, and methods can be used to determine date differences, time differences, and more. Date objects may also be formatted and converted to a number of predefined or custom formats, and specific information may be extrapolated from a date object (day of week, name of month, etc.).

19.1 Date Objects

Since dates and durations can take many forms, values that represent a date or a duration must be explicitly converted to a date or duration object using the **date** and **duration** creator methods. For example, a value of "01/01/2002 12:30:00" will be treated as a string until converted to a date object using the **date** method:

```
date('01/01/2002 12:30:00')
```

Once a value is converted to a date or duration object, special member methods, accessors, conversion operations, and math operations may then be used.

When performing date operations, Lasso uses its internal date libraries to automatically adjust for leap years and daylight saving time for the local time zone in all applicable regions of the world (as not all regions recognize daylight saving time). The current time and time zone are based on that of the computer or web server Lasso is running on.

Note: Lasso extracts daylight saving time information from the operating system. For information on special exceptions with date calculations during daylight saving time, see the section *Date and Duration Math*.

19.2 Date Type

For Lasso to recognize a string as a date, the string must be explicitly converted to a **date** type using the **date** creator method:

```
date('5/22/2002 12:30:00')
```

When converting to a **date** type using the **date** creator method, the date formats shown below are automatically recognized as valid date strings by Lasso. These automatically recognized date formats are U.S. or MySQL dates with a four-digit year followed by an optional 24-hour time with seconds. The slash (**/**), hyphen (**-**), and colon (**:**) characters are the only punctuation marks recognized in valid date strings by Lasso when used in the formats shown below.

```
1/25/2002
1/25/2002 12:34
1/25/2002 12:34:56
1/25/2002 12:34:56 GMT
2002-01-25
```

```
2002-01-25 12:34:56
2002-01-25 12:34:56 GMT
```

Lasso also recognizes a number of special-purpose date formats which are shown below. These are useful when working with HTTP headers or email message headers.

```
20020125123456
20020125T12:34:56
Tue, Dec 17 2002 12:34:56 -0800
Tue Dec 17 12:34:56 PST 2002
```

The date formats containing time zone information (e.g. "-0800" or "PST") will be recognized as GMT dates. The time zone will be used to automatically adjust the date/time to the equivalent GMT date/time.

If using a date format not listed above, custom date formats can be defined using the `-format` parameter of the **date** creator method.

The following variations of the automatically recognized date formats are valid without using the `-format` parameter:

- If the **date** creator method is used without a parameter then the current date and time is returned. Milliseconds are rounded to the nearest second.

- If the time is not specified then it is set to be the current hour when the object is created. For example, "22:00:00" if the object was created at 10:48:59 PM:

    ```
    mm/dd/yyyy -> mm/dd/yyyy 22:00:00
    ```

- If the seconds are not specified then the time is assumed to be even on the minute:

    ```
    mm/dd/yyyy hh:mm -> mm/dd/yyyy hh:mm:00
    ```

- An optional "GMT" designator can be used to specify Greenwich Mean Time rather than local time:

    ```
    mm/dd/yyyy hh:mm:ss GMT
    ```

- Two-digit years are assumed to be in the 1st century. For best results, always use four-digit years:

    ```
    mm/dd/00 -> mm/dd/0001
    mm/dd/39 -> mm/dd/0039
    mm/dd/40 -> mm/dd/0040
    mm/dd/99 -> mm/dd/0099
    ```

- Days and months can be specified with or without leading "0"s. The following are all valid Lasso date strings:

    ```
    1/1/2002
    01/1/2002
    1/01/2002
    01/01/2002
    01/01/2002 16:35
    01/01/2002 16:35:45
    GMT 01/01/2002 12:35:45 GMT
    ```

19.2.1 Converting Values to Dates

If the value is in a recognized string format described previously, simply use the **date** creator method:

```
date('05/22/2002')
// => 05/22/2002

date('05/22/2002 12:30:00')
// => 05/22/2002 12:30:00

date('2002-05-22')
// => 2002-05-22
```

If the value is not in a string format described previously, use the **date** creator method with the **-format** parameter. For information on how to use the **-format** parameter, see the section *Formatting Dates*.

```
date('5.22.02 12:30', -format='%m.%d.%y %H:%M')
// => 5.22.02 12:30

date('20020522123000', -format='%Y%m%d%H%M')
// => 200205221230
```

Date values stored in database fields or variables can be converted to a date object using the **date** creator method. Either the format of the date stored in the field or variable should be in one of the formats described above or the **-format** parameter must be used to explicitly specify the format.

```
date(#myDate)
date(field('modified_date'))
date(web_request->param('birth_date'))
```

19.2.2 Date Methods

type **date**

date()

date(*-year=?, -month=?, -day=?, -hour=?, -minute=?, -second=?, -dateGMT=?, -locale::locale=?*)

date(*date::string, -format::string=?, -locale::locale=?*)

date(*date::integer, -locale::locale=?*)

date(*date::decimal, -locale::locale=?*)

date(*date::date, -locale::locale=?*)
> All the various creator methods that can be used to create a date object. When called without parameters, it returns a date object with the current date and time. A date object can be created from properly formatted strings, integers, decimals, and dates. A date object can also be created by passing valid values to the keyword parameters **-second**, **-minute**, **-hour**, **-day**, **-month**, **-year**, and **-dateGMT**. Each creator method also allows for specifying a **locale** object to use with the **-locale** keyword parameter. (By default this is set to what the **locale_default** method returns.)

date_format(*value, format::string*)

date_format(*value, -format::string*)
> Returns the passed-in date parameter in the specified format. Requires a date object or any valid objects that can be converted to a date (it automatically recognizes the same formats as the **date** creator methods). The format can be specified as the second parameter or as the value part of a **-format** keyword parameter and defines the format for the return value. See the section *Formatting Dates* below for more details on format strings.

date_setFormat(*format::string*)
> Sets the date format for date objects to use for output for an entire Lasso thread. The required parameter is a format string.

date_gmtToLocal(_value_**)**
> Converts the date/time of any object that can be converted to a date object from Greenwich Mean Time to the local time of the machine running Lasso Server.

date_localToGMT(_value_**)**
> Converts the date/time of any object that can be converted to a date object from local time to Greenwich Mean Time.

date_getLocalTimeZone()
> Returns the current time zone of the machine running Lasso Server as a standard GMT offset string (e.g. "-0700"). Optional **-long** parameter shows the name of the time zone (e.g. "America/New_York").

date_minimum()
> Returns the minimum possible date recognized by a date object in Lasso.

date_maximum()
> Returns the maximum possible date recognized by a date object in Lasso.

date_msec()
> Returns an integer representing the number of milliseconds recorded on the machine's internal clock. Can be used for general timing of code execution.

Display Date Values

The current date/time can be displayed with **date**. The example below assumes a current date and time of "5/22/2002 14:02:05":

```
date
// => 2002-05-22 14:02:05
```

The **date** type can be used to assemble a date from individual parameters. The following method assembles a valid Lasso date by specifying each part of the date separately. Since the time is not specified it is assumed to be the current time the date object is created in the example below assumes the current date and time of "5/7/2013 15:45:04":

```
date(-year=2002, -month=5, -day=22)
// => 2002-05-22 15:45:04
```

Convert Date Values To and From GMT

Any date object can be converted to and from Greenwich Mean Time (GMT) using the methods **date_gmtToLocal** and **date_localToGMT**. These methods will only convert to and from the current time zone of the machine running Lasso. The following example uses Eastern Daylight Time (EDT) as the current time zone:

```
date_gmtToLocal(date('5/22/2002 14:02:05 GMT'))
// => 05/22/2002 10:02:05 EDT

date_localToGMT(date('5/22/2002 14:02:05 EDT'))
// => 05/22/2002 18:02:05 GMT+00:00
```

Display the Current Time Zone of the Server

The **date_getLocalTimeZone** method displays the current time zone of the machine running Lasso. The following example uses Eastern Standard Time (EST) as the current time zone:

```
date_getLocalTimeZone
// => -0500

date_getLocalTimeZone(-long)
// => America/New_York
```

Time a Section of Lasso Code

Call the **date_msec** method to get a clock value before and after the code has executed. The difference in times represents the number of milliseconds that have elapsed. Note that the **date_msec** value may occasionally roll back around to zero so any negative times reported by this code should be disregarded.

```
local(start) = date_msec

// ... the code to time ...

'The code took ' + (date_msec - #start) + ' milliseconds to process.'
// => The code took 5 milliseconds to process.
```

19.2.3 Formatting Dates

Various methods take a format string for one of their parameters. A *format string* is a compilation of symbols that define the format of the string to be output or parsed. There are two different sets of formatting strings. Detailed in the following table are the classic formatting symbols, first introduced in earlier versions of Lasso:

Table 19.1: Classic Date Formatting Symbols

Symbol	Description
%D	U.S. Date Format (mm/dd/yyyy)
%Q	MySQL date format (yyyy-mm-dd)
%q	MySQL timestamp format (yyyymmddhhmmss)
%r	12-hour time format (hh:mm:ss [AM/PM])
%T	24-hour time format (hh:mm:ss)
%Y	4-digit year
%y	2-digit year
%m	month number (01=January, 12=December)
%B	full English month name (e.g. "January")
%b	abbreviated English month name (e.g. "Jan")
%d	day of month (01–31)
%w	day of week (1=Sunday, 7=Saturday)
%W	week of year
%A	full English weekday name (e.g. "Wednesday")
%a	abbreviated English weekday name (e.g. "Wed")
%H	24-hour time hour (0–23)
%h	12-hour time hour (1–12)
%M	minute (0–59)
%S	second (0–59)
%p	AM/PM for 12-hour time
%G	GMT time zone indicator (e.g. GMT-05:00)
%z	time zone offset in relation to GMT (e.g. +0100, -0800)
%Z	time zone designator (e.g. PST, EDT)
%%	percent character

Each of the date format symbols that returns a number (except **%w**) automatically pads that number with "0" so all values returned by the method are the same length.

- An optional underscore (**_**) between the percent sign (**%**) and the letter designating the symbol specifies that a space should be used instead of "0" for the padding character (e.g. **%_m** returns " 1" for January).

- An optional hyphen (**-**) between the percent sign (**%**) and the letter designating the symbol specifies that no padding should be performed (e.g. **%-m** returns "1" for January).

Note: For **%W**, previous versions of Lasso count weeks of the year starting at 0; starting week 1 on the next Monday, and week 52 on the last Monday of the year. The current version starts week 1 on the last Sunday of the previous year if the first day of the year falls on a Sunday through Wednesday, or the first Sunday of the current year otherwise, in which case the days before are part of the last week of the previous year.

Note: A date value parsed with the **%z** or **%Z** symbols will result in a date object for the equivalent GMT date/time.

As of version 9, Lasso also recognizes ICU date formatting symbols[35] for both creating and displaying dates. These format strings simply use letters to specify the format without any flags (such as the **%** character). For example, the ICU format string

---

[35] http://userguide.icu-project.org/formatparse/datetime#TOC-Date-Time-Format-Syntax

to output a two-digit year is **yy**, and to output a four-digit year is **yyyy**. Because of this, characters that are not symbols need to be escaped if they are in the format string. To escape characters in an ICU format string, wrap them in single quotes. Use two consecutive single quotes for a literal single quote.

Table 19.2: ICU Date Formatting Symbols

Symbol	Description	Example	Result
G	era designator	G, GG, or GGG	AD
		GGGG	Anno Domini
		GGGGG	A
y	year	yy	96
		y or yyyy	1996
Y	year of "Week of Year"	Y	1997
u	extended year	u	4601
U	cyclic year name (Chinese lunar calendar)	U	甲子
Q	quarter	Q or QQ	2
		QQQ	Q2
		QQQQ	2nd quarter
q	stand alone quarter	q or qq	2
		qqq	Q2
		qqqq	2nd quarter
M	month in year	M or MM	9
		MMM	Sept
		MMMM	September
		MMMMM	S
L	stand alone month in year	L or LL	9
		LLL	Sept
		LLLL	September
		LLLLL	S
w	week of year	w or ww	27
W	week of month	W	2
d	day of month	d	2
		dd	2
D	day of year	D	189
F	day of week in month	F	2 (2nd Wed in July)
g	modified Julian day	g	2451334
E	day of week	E, EE, or EEE	Tues
		EEEE	Tuesday
		EEEEE	T
		EEEEEE	Tu
e	local day of week	e or ee	2
		eee	Tues
		eeee	Tuesday
		eeeee	T
		eeeeee	Tu
c	stand alone local day of week	c or cc	2

Table 19.2 – continued from previous page

Symbol	Description	Example	Result
		ccc	Tues
		cccc	Tuesday
		ccccc	T
		cccccc	Tu
a	am/pm marker	a	pm
h	hour in am/pm (1–12)	h	7
		hh	7
H	hour of day (0–23)	H	0
		HH	0
k	hour of day (1–24)	k	24
		kk	24
K	hour in am/pm (0–11)	K	0
		KK	0
m	minute of hour	m	4
		mm	4
s	second of minute	s	5
		ss	5
S	millisecond (maximum of 3 significant digits)	S	2
	for S or SS, truncates to the number of letters	SS	23
		SSS	235
	for SSSS or longer, fills additional places with 0	SSSS	2350
A	milliseconds in day	A	61201235
z	Time Zone: specific non-location	z, zz, or zzz	PDT
		zzzz	Pacific Daylight Time
Z	Time Zone: ISO 8601[36] basic hms? / RFC 822[37]	Z, ZZ, or ZZZ	-800
	Time Zone: long localized GMT (=OOOO)	ZZZZ	GMT-08:00
	TIme Zone: ISO 8601 extended hms? (=XXXXX)	ZZZZZ	-08:00, -07:52:58, Z
O	Time Zone: short localized GMT	O	GMT-8
	Time Zone: long localized GMT (=ZZZZ)	OOOO	GMT-08:00
v	Time Zone: generic non-location	v	PT
	(falls back first to VVVV)	vvvv	Pacific Time or Los Angeles Time
V	Time Zone: short time zone ID	V	uslax
	Time Zone: long time zone ID	VV	America/Los_Angeles
	Time Zone: time zone exemplar city	VVV	Los Angeles
	Time Zone: generic location (falls back to OOOO)	VVVV	Los Angeles Time
X	Time Zone: ISO 8601 basic hm?, with Z for 0	X	-08, +0530, Z
	Time Zone: ISO 8601 basic hm, with Z	XX	-0800, Z
	Time Zone: ISO 8601 extended hm, with Z	XXX	-08:00, Z
	Time Zone: ISO 8601 basic hms?, with Z	XXXX	-0800, -075258, Z
	Time Zone: ISO 8601 extended hms?, with Z	XXXXX	-08:00, -07:52:58, Z
z	Time Zone: ISO 8601 basic hm?, without Z for 0	x	-08, +0530
	Time Zone: ISO 8601 basic hm, without Z	xx	-800

Table 19.2 – continued from previous page

Symbol	Description	Example	Result
	Time Zone: ISO 8601 extended hm, without Z	xxx	-08:00
	Time Zone: ISO 8601 basic hms?, without Z	xxxx	-0800, -075258
	Time Zone: ISO 8601 extended hms?, without Z	xxxxx	-08:00, -07:52:58
'	begin/end text string	`'text'`	text
''	literal single quote	`''`	'

Note: Format strings in Lasso can contain both percent-based formatting as well as ICU formatting in the same string. Because of this, be sure you properly escape any characters you don't want treated as format delimiters in your format string. For example, if the current date was "2013-03-09 20:15:30", the following code: `date->format("day: %A")` would produce "9PM2013: Saturday" as the "day" portion of the format string would be treated as part of ICU formatting. Wrapping in single quotes mitigates this: `date->format("'day: '%A")`.

Convert Date Strings to Various Formats

The following examples show how to use **date_format** to output either Lasso date objects or valid Lasso date strings to alternate formats:

```
date_format('06/14/2001', -format='%A, %B %d')
// => Thursday, June 14

date_format('06/14/2001', '%a, %b %d')
// => Thu, Jun 14

date_format('2001-06-14', -format='%Y%m%d%H%M')
// => 200106141600

date_format(date('1/4/2002'), '%m.%d.%y')
// => 01.04.02

date_format(date('1/4/2002 02:30:00'), -format='%B, %Y')
// => January, 2002

date_format(date('1/4/2002 02:30:00'), '%r')
// => 02:30:00 AM

date_format(date, -format='y-MM-dd')
// => 2013-02-24
```

Import and Export Dates from MySQL

A common conversion in Lasso is converting MySQL dates to and from U.S. dates. Dates are stored in MySQL in the format "yyyy-mm-dd". The following example shows how to import a date in this format and then output it to U.S. date format using the **date_format** method:

```
date_format('2001-05-22', -format='%-D')
// => 5/22/2001
```

---

[36] https://en.wikipedia.org/wiki/ISO_8601
[37] https://tools.ietf.org/html/rfc822.html

```
date_format('5/22/2001', -format='%Q')
// => 2001-05-22

date_format(date('2001-05-22'), '%D')
// => 05/22/2001

date_format(date('5/22/2001'), '%Q')
// => 2001-05-22
```

Set a Custom Date Format for a Thread

Use the **date_setFormat** method. This allows all date objects in a thread to be output in a custom format without the use of the **date_format** or **date->format** methods. The format specified is only valid for the currently executing thread after the **date_setFormat** method has been called:

```
date_setFormat('%m%d%y')
```

The example above means that from now on in the currently executing thread, all dates converted to strings will use that format.

```
date('01/01/2002')
// => 010102
```

19.2.4 Date Formatting Methods

In addition to **date_format** and **date_setFormat**, Lasso also offers the **date->format** and **date->setFormat** member methods for performing formatting adjustments on date objects.

date->**format**()

date->**format**(*format::string, -locale::locale=?*)

date->**format**(*-format::string, -locale::locale=?*)
> Outputs the date object in the specified format. If no format is passed, the current format stored with the object will be used. Optionally takes a **locale** object to set its locale.

date->**setFormat**(*format::string*)
> Sets a date output format for a particular date object. Requires a format string as a parameter.

date->**getFormat**()
> Returns the current format string set for the current date object. This will always return an ICU format string.

date->**clear**()
> Resets the specified fields to their default values. The following fields can be specified as keyword parameters: **-second**, **-minute**, **-hour**, **-day**, **-week**, **-month**, **-year**. If no parameters are specified, then the entire date is reset to default values.

date->**set**(…)
> Sets one or more elements of the date to a new value. If a field overflows then other fields may be modified as well. For example, if you have the date "3/31/2008" and you set the month to "2" then the day will be adjusted to "29" automatically resulting in "2/29/2008".
>
> Elements must be specified as *keyword=value* parameters. See the table *Date Field Element Parameters for get and set* for the full list of parameters that this method can set.

`date->get(…)`
> Returns the current value for the specified field of the current date object. Only one field value can be fetched at a time. Note that many of the more common fields can also be retrieved through individual member methods.
>
> See the table *Date Field Element Parameters for get and set* for the full list of value types that this method can retrieve.

Table 19.3: Date Field Element Parameters for **get** and **set**

Parameter	Description
-year	Specifies the year field for the date.
-month	Specifies the month field for the date.
-week	Specifies the week field for the date.
-day	Specifies the day field for the date.
-hour	Specifies the hour field for the date.
-minute	Specifies the minute field for the date.
-second	Specifies the second field for the date.
-weekofyear	Specifies the week of year field for the date.
-weekofmonth	Specifies the week of month field for the date.
-dayofmonth	Specifies the day of month field for the date.
-dayofyear	Specifies the day of year field for the date.
-dayofweek	Specifies the day of week field for the date.
-dayofweekinmonth	Specifies the day of week in month field for the date.
-ampm	Specifies the am/pm field for the date.
-hourofday	Specifies the hour of day field for the date.
-zoneoffset	Specifies the time zone offset field for the date.
-dstoffset	Specifies the DST offset field for the date.
-yearwoy	Specifies the year week of year field for the date.
-dowlocal	Specifies the local day of week field for the date.
-extendedyear	Specifies the extended year field for the date.
-julianday	Specifies the julian day field for the date.
-millisecondsinday	Specifies the milliseconds in day field for the date.

Convert Date Objects to Various Formats

The following examples show how to output date objects in alternate formats using the **date->format** method:

```
local(my_date) = date('2002-06-14 00:00:00')
#my_date->format('%A, %B %d')
// => Friday, June 14

local(my_date) = date('06/14/2002 09:00:00')
#my_date->format('%Y%m%d%H%M')
// => 200206140900

local(my_date) = date('01/31/2002')
#my_date->format('%d.%m.%y')
// => 31.01.02

local(my_date) = date('09/01/2002')
#my_date->format('%B, %Y')
```

```
// => September, 2002
```

Set an Output Format for a Specific Date Object

Use the **date->setFormat** method. This causes all instances of a particular date object to be output in a specified format:

```
local(my_date) = date('01/01/2002')
#my_date->setFormat('%m%d%y')
```

The example above causes all instances of **#my_date** in the current code to be output in a custom format without the **date_format** or **date->format** methods:

```
#my_date
// => 010102
```

Use Locales to Format Dates

The **locale** type that allows for automatically formatting data such as dates and currency based on known standards for various locations. You can use **locale** objects to output dates in these standard formats.

type **locale**

locale(*language::string, country::string=?, variant::string=?*)
> Creates a **locale** object which may be used to format the output of various data in the manner specified by the locale.
>
> The method requires one parameter which is the 2-letter ISO 639[38] code of the language, and accepts optional parameters for the 2-letter ISO 3166[39] country code and a variant code which allows further refinement to the locale.

locale->format(*as::date, style::integer=?, andTime::integer=?, addlFlag::integer=?*)
> Display a date in the format of the given locale. The method requires one parameter which is the date value to be formatted. When formatting dates, the method accepts up to three additional integer flags which specify different date/time formatting types.

The following example creates two locale objects (one for the U.S. and one for Canada) and uses them to output the date in the format for each locale:

```
local(my_date) = date('01/01/2005 08:40:33 AM')
local(en_us)   = locale('en', 'US')
local(en_ca)   = locale('en', 'CA')

#en_us->format(#my_date, 1)
// => January 1, 2005

#en_ca->format(#my_date, 1)
// => 1 January, 2005
```

19.2.5 Date Accessor Methods

A date accessor method returns a specific integer or string value from a date object, such as the name of the current month or the seconds of the time.

date->year()
> Returns a four-digit integer representing the year for the date object (defaults to current date).

---

[38] http://www.loc.gov/standards/iso639-2/
[39] http://www.iso.org/iso/prods-services/iso3166ma/02iso-3166-code-lists/country_names_and_code_elements

`date->`**`month(`**`-long::boolean=?, -short::boolean=?`**`)`**
>   Returns the numerical month (1=January, 12=December) for the date object. Optional **`-long`** parameter returns the full month name (e.g. "January") or an optional **`-short`** parameter returns an abbreviated month name (e.g. "Jan").

`date->`**`week()`**

`date->`**`weekOfYear()`**
>   Returns the numerical week of the year (out of 52) for the date object.

`date->`**`weekOfMonth()`**
>   Returns the numerical week of the month for the date object.

`date->`**`dayOfWeekInMonth()`**
>   Returns the numerical day of week in month for the date object.

`date->`**`dayOfYear()`**
>   Returns the numerical day of the year (out of 365) for the date object. Will work for leap years as well (out of 366).

`date->`**`day()`**

`date->`**`dayOfMonth()`**
>   Returns the numerical day of the month (e.g. 15) for the date object.

`date->`**`dayOfWeek()`**
>   Returns the numerical day of the week (1=Sunday, 7=Saturday) for the date object.

`date->`**`hour()`**

`date->`**`hourOfDay()`**
>   Returns the hour (0–23) for the date object.

`date->`**`hourOfAMPM()`**
>   Returns the relative hour (1–12) for the date object.

`date->`**`minute()`**
>   Returns the minute (0–59) for the date object.

`date->`**`second()`**
>   Returns the second (0–59) for the date object.

`date->`**`millisecond()`**
>   Returns the millisecond (0–59) for the date object.

`date->`**`time()`**
>   Returns the time for the date object.

`date->`**`ampm()`**
>   Returns "0" if the time is before noon and "1" if the time is noon or later.

`date->`**`am()`**
>   Returns "true" if the time is in the morning (before noon), otherwise returns "false".

`date->`**`pm()`**
>   Returns "true" if the time is in the evening (noon or after), otherwise returns "false".

`date->`**`timezone()`**
>   Returns the set time zone for the date object. Defaults to the current time zone of the server.

`date->`**`zoneOffset()`**
>   Returns the time zone offset field for the date object.

`date->`**`gmt()`**
>   Returns "true" if the date object is in GMT time and "false" if it is in local time.

date->**dst()**
> Returns "true" if the date object is in daylight saving time and "false" if it is not.

date->**dstOffset()**
> Returns the daylight saving time (DST) offset field for the date object. Returns "0" if the date for the time zone is not experiencing daylight savings.

date->**asInteger()**
> Returns "epoch time", the number of seconds from 1/1/1970 to the time of the date object.

Using Date Accessors

The individual parts of a date object can be displayed using the **date** type member methods:

```
date('5/22/2002 14:02:05')->year
// => 2002

date('5/22/2002 14:02:05')->month
// => 5

date('2/22/2002 14:02:05')->month(-long)
// => February

date('5/22/2002 14:02:05')->day
// => 22

date('5/22/2002 14:02:05')->dayOfWeek
// => 4

date('5/22/2002 14:02:05')->time
// => 14:02:05

date('5/22/2002 14:02:05')->hour
// => 14

date('5/22/2002 14:02:05')->minute
// => 2

date('5/22/2002 14:02:05')->second
// => 5
```

The **date->millisecond** method can only return the current number of milliseconds (as related to the clock time) for the machine running Lasso:

```
date->millisecond
// => 957
```

19.3 Duration Type

A **duration** is a special type that represents a length of time. A duration is not a 24-hour clock time, and may represent any number of hours, minutes, or seconds.

Similar to dates, durations must be created using **duration** creator methods before they can be manipulated. Durations may be converted from a "hours:minutes:seconds"-formatted string, or just as seconds.

```
duration('1:00:00')
// => 1:00:00

duration(3600)
// => 1:00:00
```

Once an object has been created as a **duration** type, duration calculations and accessors may then be used. Durations are especially useful for calculating lengths of time under 24 hours, though they can be used for any lengths of time. Durations are based on start and end date/time objects. These start and end date/times are either specified when creating the duration or the current date/time is used as the start date/time while the end date/time is calculated based on the specified length for the duration.

19.3.1 Duration Methods

type **duration**

duration(*time::string*)

duration(*time::integer*)

duration(*start::date*, *end::date*)

duration(*start::string*, *end::string*)

duration(*-year=?*, *-week=?*, *-day=?*, *-hour=?*, *-minute=?*, *-second=?*)
> Creates a duration object. Accepts a duration string for **'hours:minutes:seconds'**, an integer number of seconds, or a start and end date specified as either dates or strings that can be converted to dates. Or by specifying one or more of the following keyword parameters to add the amount of time indicated by the name of the keyword parameter: **-year**, **-week**, **-day**, **-hour**, **-minute**, or **-second**.

duration->**year()**
> Returns the integer number of years in a duration (based on the specified start and end date or based on a start date of when the duration object was created with an end date dependant on the specified length of time).

duration->**month()**
> Returns the integer number of months in a duration (based on the specified start and end date or based on a start date of when the duration object was created with an end date dependant on the specified length of time).

duration->**week()**
> Returns the integer number of weeks in the duration.

duration->**day()**
> Returns the integer number of days in the duration.

duration->**hour()**
> Returns the integer number of hours in the duration.

duration->**minute()**
> Returns the integer number of minutes in the duration.

duration->**second()**
> Returns the integer number of seconds in the duration.

Create and Display Durations

Durations can be created using the **duration** creator method with the **-week**, **-day**, **-hour**, **-minute**, and **-second** parameters. This always returns a duration object whose **duration->asString** method returns a string in "hours:minutes:seconds" format.

```
duration(-week=5, -day=3, -hour=12)
// => 924:00:00

duration(-day=4, -hour=2, -minute=30)
// => 98:30:00

duration(-hour=12, -minute=45, -second=50)
// => 12:45:50

duration(-hour=3, -minute=30)
// => 03:30:00

duration(-minute=15, -second=30)
// => 00:15:30

duration(-second=30)
// => 00:00:30
```

Specific elements of time can be returned from a duration using the accessor member methods.

```
duration('8766:30:45')->year
// => 1

duration('8766:30:45')->month
// => 12

duration('8766:30:45')->week
// => 52

duration('8766:30:45')->day
// => 365

duration('8766:30:45')->hour
// => 8766

duration('8766:30:45')->minute
// => 525990

duration('8766:30:45')->second
// => 31559445
```

19.4 Date and Duration Math

Date calculations can be performed by using special **date_...** methods, **date** member methods, and operators. Date calculations that can be performed include adding or subtracting year, month, week, day, and time increments to and from dates, and calculations with durations.

Important: Lasso does not account for changes to and from daylight saving time when performing date math and duration calculations. One should take this into consideration when performing a date or duration calculation across dates that encompass a change to or from daylight saving time, as the resulting date may be off by an hour.

19.4.1 Date Math Methods

Lasso provides a few top-level methods for performing date calculations. These methods are summarized below.

date_add(*value*, *-millisecond::integer=?*, *-second::integer=?*, *-minute::integer=?*, *-hour::integer=?*, *-day::integer=?*, *-week::integer=?*, *-month::integer=?*, *-year::integer=?*)

> Returns a date value generated by adding a specified amount of time to a specified date object or valid date string. The first parameter is a date object or valid value that can be converted to a date. Keyword/value parameters define what should be added to the first parameter.

date_subtract(*value*, *-millisecond::integer=?*, *-second::integer=?*, *-minute::integer=?*, *-hour::integer=?*, *-day::integer=?*, *-week::integer=?*, *-month::integer=?*, *-year::integer=?*)

> Returns a date value generated by subtracting a specified amount of time from a specified date value. The first parameter is a Lasso date object or valid value that can be converted to a date. Keyword/value parameters define what should be subtracted from the first parameter.

date_difference(*value*, *when*, *...*)

> Returns the time difference between two specified dates. A duration is the default return value. Optional parameters may be used to output a specific integer time value instead of a duration: **-millisecond**, **-second**, **-minute**, **-hour**, **-day**, **-week**, **-month**, or **-year**.

Add Time to a Date

Using the **date_add** method, a specified number of hours, minutes, seconds, days, or weeks can be added to a date object or valid value that can be converted to a date. The following examples show the result of adding different values to the current date of "5/22/2002 14:02:05":

```
date_add(date, -second=15)
// => 2002-05-22 14:02:20

date_add(date, -minute=15)
// => 2002-05-22 14:17:05

date_add(date, -hour=15)
// => 2002-05-23 05:02:05

date_add(date, -day=15)
// => 2002-06-06 14:02:05

date_add(date, -week=15)
// => 2002-09-04 14:02:05

date_add(date, -month=6)
// => 2002-11-22 14:02:05

date_add(date, -year=1)
// => 2003-05-22 14:02:05
```

Subtract Time from a Date

Using the **date_subtract** method, a specified number of hours, minutes, seconds, days, or weeks can be subtracted a date object or valid value that can be converted to a date. The following examples show the result of subtracting different values from the date "5/22/2001 14:02:05":

```
date_subtract(date('5/22/2001 14:02:05'), -second=15)
// => 05/22/2001 14:01:50
```

```
date_subtract(date('5/22/2001 14:02:05'), -minute=15)
// => 05/22/2001 13:47:05

date_subtract(date('5/22/2001 14:02:05'), -hour=15)
// => 05/21/2001 23:02:05

date_subtract('5/22/2001 14:02:05', -day=15)
// => 05/7/2001 14:02:05

date_subtract('5/22/2001 14:02:05', -week=15)
// => 02/6/2001 14:02:05
```

Determine the Duration Between Two Dates

Use the **date_difference** method. The following examples show how to calculate the time difference between two date object or valid values that can be converted to a date:

```
date_difference(date('5/23/2002'), date('5/22/2002'))
// => 24:00:00

date_difference(date('5/23/2002'), date('5/22/2002'), -second)
// => 86400

date_difference(date('5/23/2002'), '5/22/2002', -minute)
// => 1440

date_difference(date('5/23/2002'), '5/22/2002', -hour)
// => 24

date_difference('5/23/2002', date('5/22/2002'), -day)
// => 1

date_difference('5/23/2002', date('5/30/2002'), -week)
// => -1

date_difference('5/23/2002', '6/23/2002', -month)
// => -1

date_difference('5/23/2002', '5/23/2001', -year)
// => 1
```

19.4.2 Date Math Member Methods

A number of member methods are used for performing date math operations on date objects, such as adding durations to dates, subtracting a duration from a date, and determining a duration between two dates. These methods are summarized below.

date->**add**(…)
>   Adds a specified amount of time to a date object. Can pass a duration object or specify the amount of time by passing keyword parameters to define what values should be added to the object: **-second**, **-minute**, **-hour**, **-day**, **-week**, **-month**, or **-year**.

date->**roll**(…)
>   Like **date->add**, this method adds the specified amount of time to the current date object. However, unlike **date->add**,

only the specified field is adjusted. For example, rolling 60 minutes doesn't change the date at all since the minute field will roll back to its original value and the hour field will not be modified. Valid fields to roll are `-second`, `-minute`, `-hour`, `-day`, `-week`, `-month`, or `-year`.

date->subtract(…)
> Subtracts a specified amount of time from a date object. Can pass a duration object or specify the amount of time by passing keyword/value parameters to define what should be subtracted from the object: `-millisecond`, `-second`, `-minute`, `-hour`, `-day`, or `-week`.

date->difference(*when*, …)
> Calculates the duration between two date objects. The first parameter must be a valid value that can be converted to a date. By default, this method returns a duration object, but will return an integer time value if one of the following optional parameter is specified: `-millisecond`, `-second`, `-minute`, `-hour`, `-day`, `-week`, `-month`, or `-year`.

date->daysBetween(*other::date*)
> Requires another date object as a parameter and returns the number of days between the current date object and the specified date object.

date->businessDaysBetween(*other::date*)
> Requires another date object as a parameter and returns the number of business days between the current date object and the specified date object.

date->hoursBetween(*other::date*)
> Requires another date object as a parameter and returns the number of hours between the current date object and the specified date object.

date->minutesBetween(*other::date*)
> Requires another date object as a parameter and returns the number of minutes between the current date object and the specified date object.

date->secondsBetween(*other::date*)
> Requires another date object as a parameter and returns the number of seconds between the current date object and the specified date object.

Note: The **date->add**, **date->roll**, and **date->subtract** methods do not return any values, but are instead used to change the value of the object calling them.

Add Time to a Date

Use the **date->add** method. The following examples show how to add a duration to a date and display that date:

```
local(my_date) = date('5/22/2002')
#my_date->add(duration('24:00:00'))
#my_date
// => 05/23/2002

local(my_date) = date('5/22/2002 00:00:00')
#my_date->add(duration(3600))
#my_date
// => 05/22/2002 01:00:00

local(my_date) = date('5/22/2002')
#my_date->add(-week=1)
#my_date
// => 05/29/2002
```

Subtract Time from a Date

Use the **date->subtract** method. The following examples show how to subtract a duration from a date object and display
that date:

```
local(my_date) = date('5/22/2002')
#my_date->subtract(duration('24:00:00'))
#my_date
// => 05/21/2002


local(my_date) = date('5/22/2002 00:00:00')
#my_date->subtract(duration(7200))
#my_date
// => 05/21/2002 22:00:00


local(my_date) = date('5/22/2002')
#my_date->subtract(-day=3)
#my_date
// => 05/19/2002
```

Determine the Duration Between Two Dates

Use the **date->difference** method. The following examples show how to calculate the time difference between two dates
and display as a duration:

```
local(my_date) = date('5/15/2002 00:00:00')
#my_date->difference(date('5/22/2002 01:30:00'))
// => 169:30:00


local(my_date) = date('5/15/2002')
#my_date->difference(date('5/22/2002'), -day)
// => 7
```

19.4.3 Date Math Operators

Date and duration math can also be performed using math operators in a manner similar to integer objects. If a date or duration
appears to the left of a math operator then the appropriate math operation will be performed and the result will be a date or
duration as appropriate.

date->+(*rhs*)

> A duration can be added to a date or two durations summed using the **+** operator.

date->-(*rhs*)

> A duration can be subtracted from a date or duration or the duration between two dates can be determined using the
> **-** operator.

Add or Subtract Dates and Durations

The following examples show addition and subtraction operations using dates and durations:

```
date('5/22/2002') + duration('24:00:00')
// => 05/23/2002


date('5/22/2002') - duration('48:00:00')
// => 05/20/2002
```

Determine the Duration Between Two Dates

The following calculates the duration between two dates using the subtraction operator (-):

```
date('5/22/2002') - date('5/15/2002')
// => 168:00:00
```

Add One Day to the Current Date

The following example adds one day to the current date:

```
date + duration(-day=1)
// => 2007-10-30 18:03:27
```

Return the Duration Between Now and a Future Date

The following example returns the duration between the current date and 12/31/2250:

```
date('12/31/2250') - date
// => 2079000:56:08
```

Regular Expressions

The regular expression type in Lasso allows for powerful search and replace operations on strings and byte streams. This chapter details how the regular expression type works and other Lasso methods that use regular expressions.

20.1 Regular Expression Structure

A *regular expression* is a pattern that describes a sequence of characters that you want to search for in a target (or input) string. Regular expressions consist of letters or digits that simply match themselves, wildcards that match any character in a class such as whitespace or digits, and combining symbols that expand wildcards to match several characters rather than just one. Lasso uses the ICU Regular Expressions package[40] for its support of regular expressions.

Note: Full documentation of regular expression methodology is outside the scope of this guide. Consult a standard reference on regular expressions for more information about how to use this flexible technology.

20.1.1 Basic Matchers

The simplest regular expression is just a pattern containing letters or digits. The regular expression **bird** is said to match the string "bird". The regular expression **123** matches the string "123". The regular expression is matched against an input string by comparing each character in the regular expression to each character in the input string, one after another. Regular expressions are normally case-sensitive so the regular expression **John** would not match the string "john".

Unicode characters within a regular expression work the same as any other character. The escape sequence **\u2FB0** with the four-digit hex value for a Unicode character can also be used in place of any actual character (within regular expressions or any Lasso strings). The escape sequence **\u2FB0** represents a Chinese character.

Regular expressions can also match part of a string. The regular expression **bird** is found starting at position 3 in the string "A bird in the hand".

A regular expression can contain wildcards that match one of a set of characters. **[Jj]** is a wildcard which matches either an uppercase "J" or a lowercase "j". The regular expression **[Jj]ohn** will match either the string "John" or the string "john". The wildcard **[aeiou]** matches any lowercase vowel. The wildcard **[a-z]** matches any lowercase roman letter. The wildcard **[0-9]** matches any arabic digit. The wildcard **[a-zA-Z]** matches any uppercase or lowercase roman letter. If a Unicode character is used in a character range then any characters between the hex value for the two characters are matched. The wildcard **[\u2FB0-\u2FBF]** will match 16 different Chinese characters.

The period (**.**) is a special wildcard that matches any single character. The regular expression **..** would match any two-character string including "be", "12", or even " " (two spaces). The period will match any ASCII or Unicode character including punctuation or most whitespace characters. It will not match return or newline characters.

A number of other predefined wildcards are available. The predefined wildcards are all preceded by a backslash (****).

---

[40] http://userguide.icu-project.org/strings/regexp

Many of the predefined wildcards come in pairs. The wildcard `\s` matches any whitespace character including tabs, spaces, returns, or newlines. The wildcard `\S` matches any non-whitespace character. The wildcard `\w` matches any alphanumeric character or underscore. The "w" is said to stand for "word" since these are all characters that may appear within a word. The wildcard `\W` matches non-word characters. The wildcard `\d` matches any arabic digit and the wildcard `\D` matches any non-digit. For example, the regular expression `\w\w\w` would match any three-character word such as "cat" or "dog". The regular expression `\d\d\d-\d\d\d\d-\d\d\d\d` would match a standard North American phone number in the form "360-555-1212".

The predefined wildcards only work on standard ASCII strings. There is a special pair of wildcards `\p` and `\P` that allow different characters in a Unicode string to be matched. The wildcard is specified as `\p{Property}`. A list of properties can be found in the table below. For example the wildcard `\p{L}` matches any Unicode letter character, the wildcard `\p{N}` matches any Unicode digit, and the wildcard `\p{P}` matches any Unicode punctuation characters. The `\P{Property}` wildcard is the opposite. `\P{L}` matches any Unicode character that is not a letter.

Many characters have special meanings in regular expressions including `[ ] ( ) { } . * + ? ^ $ \ |`. In order to match one of these characters literally it is necessary to use a backslash in front of it. For example `\[` matches a literal opening square bracket rather than starting a character range.

It is important to remember that double- or single-quoted string literals use a backslash for escape sequences, so you must use a double backslash to use the predefined wildcards and to escape special characters. You can avoid having to use a double backslash by specifying the regular expression using ticked string literals. However, the use of ticked string literals makes it difficult to match common escape sequences such as returns (`\r`) or newlines (`\n`). It is recommended that you use ticked string literals for all of your regular expressions until you need one of these escape sequences, and then that you concatenate in a non-ticked string literal for these sequences. For example, the following string concatenation would create a regular expression that matches a letter followed by a tab followed by a digit:

```
local(my_regexp) = `\w` + "\t" + `\d`
```

Basic Matching Strings

Below is a listing of basic matchers and a brief definition. Matches are case-sensitive by default. Be sure to note whether quoted or ticked string literals are being used.

`` `.` ``
    Period matches any single character except a line break.

`` `[ ]` ``
    Character class. Matches any character contained between the square brackets.

`` `[^ ]` ``
    Character exception class. Matches any character that is not contained between the square brackets.

`` `[a-z]` ``
    Lowercase character range. Matches any character between the two specified.

`` `[A-Z]` ``
    Uppercase character range.

`` `[a-zA-Z]` ``
    Combination character range matching any letter.

`` `[0-9]` ``
    Numeric character range.

`"\t"`
    Matches a tab character.

`"\r"`
    Matches a return character.

"\n"
    Matches a newline character.

`"`
    Matches a double quote.

`'`
    Matches a single quote.

`\x##`
    Matches a single ISO-8859-1 character. The number signs should be replaced with the 2-digit hex value for the character.

`\u####`
    Matches a single Unicode character. The number signs should be replaced with the 4-digit hex value (code point) for the Unicode character.

`\p{ }`
    Matches a single Unicode character with the stated property. (The available properties are listed next.)

`\P{ }`
    Matches a single Unicode character that does not have the stated property. (The available properties are listed next.)

`\w`
    Matches an alphanumeric "word" character, including underscores.

`\W`
    Matches a non-alphanumeric character (whitespace or punctuation).

`\s`
    Matches a blank, whitespace character. Equivalent to `[\t\n\f\r\p{Z}]`.

`\S`
    Matches a non-blank, non-whitespace character.

`\d`
    Matches a digit character. Equivalent to `[0-9]`.

`\D`
    Matches a non-digit character.

`\`
    Escapes the next character. This allows any symbol to be specified as a matching character including the reserved characters `[ ] ( ) { } . * + ? ^ $ \ |`.

The following table lists the property symbols that can be used with the `\p` and `\P` wildcards. The main symbol (e.g. `\p{L}`) will match all of the characters that are matched by each of the variants.

Table 20.1: Unicode Property Symbols

Symbol	Property	Variants	Description
L	letter	Lu	Uppercase Letter
		Ll	Lowercase Letter
		Lt	Titlecase Letter
		Lm	Modifier Letter
		Lo	Other Letter
N	number	Nd	Decimal Digit Number
		Nl	Letter Number
		No	Other Number

<div align="center">Continued on next page</div>

Table 20.1 – continued from previous page

Symbol	Property	Variants	Description
P	punctuation character	Pc	Connector Punctuation
		Pd	Dash Punctuation
		Ps	Open Punctuation
		Pe	Close Punctuation
		Pi	Initial Punctuation
		Pf	Final Punctuation
		Po	Other Punctuation
S	symbol	Sm	Math Symbol
		Sc	Currency Symbol
		Sk	Modifier Symbol
		So	Other Symbol
Z	separator (typically whitespace)	Zs	Space Separator
		Zl	Line Separator
		Zp	Paragraph Separator
M	mark	Mn	Non-Spacing Mark
		Mc	Spacing Combining Mark
		Me	Enclosing Mark
C	"other" character	Cc	Control
		Cf	Format
		Cs	Surrogate
		Co	Private Use
		Cn	Not Assigned

20.1.2 Combining Symbols

Combining symbols allow wildcards to be expanded to match entire substrings rather than individual characters. For example, the wildcard `[a-z]` matches one lowercase letter and needs to be repeated three times to match a three letter word `[a-z][a-z][a-z]`. Instead, the combining symbol `{3}` can be used to specify that the preceding wildcard should be repeated three times `[a-z]{3}`.

The combining symbol `+` matches one or more repetitions of the preceding matcher. The expression `[a-z]+` matches any string of lowercase letters. This expression matches the strings "a", "green", or "international". It does not match "$1,544,897.00" because that string does not contain any lowercase letters.

The combining symbol `+` can be used with the `.` wildcard to match any string of one or more characters (`.+`), with the wildcard `\w` to match any word (`\w+`), or with the wildcard `\s` to match one or more whitespace characters (`\s+`). The `+` symbol can also be used with a simple letter to match one or more repetitions of the letter. The regular expression `Me+t` matches both the string "Met" and the string "Meet", not to mention "Meeeeeet".

The combining symbol `*` matches zero or more repetitions of the preceding matcher. The `*` symbol can be used with the generic wildcard `.` to match any string of characters (`.*`). The `*` symbol can be used with the whitespace wildcard `\s` to match a string of whitespace characters. For example, the expression `\s*cat\s*` will match the string "cat", but also the string " cat ".

Braces are used to designate a specific number of repetitions of the preceding wildcard. When the braces contain a single number they designate that the preceding wildcard should be matched exactly that number of times. For example, `[a-z]{3}` matches any three lowercase letters. When the braces contain two numbers they allow for any number of repetitions from the lower number to the upper number. The pattern `[a-z]{3,5}` matches any three to five lowercase letters. If the second

number is omitted then the braces function similarly to a `+`, e.g. `[a-z]{3,}` matches any string of lowercase letters with a length of 3 or longer.

The symbol `?` on its own makes the preceding matcher optional. For example, the expression `mee?t` will match either the string "met" or "meet" since the second "e" is optional, but it won't match "meeeet".

When used after a `+`, `*`, or braces the `?` makes the match non-greedy. Normally, a subexpression will match as much of the input string as possible. The expression `<.*>` will match a string that begins and ends with angle brackets. It will match the entire string `"<b>Bold Text</b>"`. With the non-greedy option the expression `<.*?>` will match the shortest string possible. It will now match just the first part of the string `"<b>"` and a second application of the expression will match the last part of the string `"</b>"`.

+
> Matches 1 or more repetitions of the preceding symbol.

*
> Matches 0 or more repetitions of the preceding symbol.

?
> Makes the preceding symbol optional.

{n}
> Braces. Matches "n" repetitions of the preceding symbol.

{n,}
> Matches at least "n" repetitions of the preceding symbol.

{n,m}
> Matches at least "n", but no more than "m" repetitions of the preceding symbol.

+?
> Non-greedy variant of the plus sign; matches the shortest string possible.

*?
> Non-greedy variant of the asterisk; matches the shortest string possible.

{ }?
> Non-greedy variant of braces; matches the shortest string possible.

20.1.3 Groupings

Groupings have two purposes in regular expressions: they allow portions of a regular expression to be designated as groups that can be used in a replacement pattern, and they allow more complex regular expressions to be built up from simple regular expressions.

Parentheses are used to designate a portion of a regular expression as a replacement group. Most regular expressions are used to perform find/replace operations so this is an essential part of designing a pattern. Note that if parentheses are meant to be a literal part of the pattern then they need to be escaped as `\(` and `\)`. The regular expression `<b>(.*?)</b>` matches an HTML bold tag. The contents of the tag are designated as a group. If this regular expression is applied to the string `"<b>Bold Text</b>"` then the pattern matches the entire string and "Bold Text" is designated as the first group.

Similarly, a phone number could be matched by the regular expression `((d{3})) (d{3})-(d{4})` with three groups. The first group represents the area code (note that the parentheses appear in both escaped form `\( \)` to match literal opening and closing parentheses and normal form `( )` to designated a grouping). The second group represents the prefix and the third group the subscriber number. When the regular expression is applied to the string "(360) 555-1212" then the pattern matches the entire string and generates the groups "360", "555", and "1212".

Parentheses can also be used to create a subexpression that does not generate a replacement group using `(?:)`. This form can be used to create subexpressions that function much like very complex wildcards. For example, the expression `(?:blue)+` will match one or more repetitions of the subexpression "blue". It will match the strings "blue", "blueblue" or "blueblueblueblue".

The **|** symbol can be used to specify alternation. It is most useful when used with subexpressions. The expression `(?:blue)|(?:red)` will match either the word "blue" or the word "red".

( )

> Grouping for output. Defines a numbered group for output. Up to nine groups can be defined.

(?:)

> Grouping without output. Can be used to create a logical grouping that should not be assigned to an output.

|

> Alternation. Matches either the characters before or the characters after the symbol. May appear within a group to limit the alternation boundary.

20.1.4 Replacement Expressions

When regular expressions are used for find/replace operations the replacement expression can contain placeholders into which the defined groups from the search expression are substituted. The placeholder **$0** represents the entire matched string. The placeholders **$1** through **$9** represent the first nine groupings as defined by parentheses in the regular expression.

The regular expression `<b>(.*?)</b>` from above matches an HTML bold tag with the contents of the tag designated as a group. The replacement expression `<em>$1</em>` will essentially replace the bold tags with emphasis tags, without disrupting the contents of the tags. For example the string `"<b>Bold Text</b>"` would result in `"<em>Bold Text</em>"` after a find/replace operation.

The phone number expression `((d{3})) (d{3})-(d{4})` from above matches a phone number and creates three groups for the parts of the phone number. The replacement expression `$1-$2-$3` would rewrite the phone number to be in a more standard format. For example, the string "(360) 555-1212" would result in "360-555-1212" after a find/replace operation.

$0–$9

> Names a group in the replace string. **$0** represents the entire matched string. Up to nine groups can be specified using the digits 1 through 9.

Tip: To place a literal **$** in a replacement string, escape it as **\$**.

20.1.5 Advanced Expressions

The ICU library also supports a number of more advanced symbols for special purposes. Some of these symbols are listed in the following table, but a reference on regular expressions should be consulted for full documentation of these symbols and other advanced concepts. A list of regular expression flags follows.

(?#)

> Regular expression comment. The contents are not interpreted as part of the regular expression.

(?i)

> Sets a flag to make the remainder of the regular expression case-insensitive. Similar to specifying `-ignoreCase`.

(?-i)

> Sets the remainder of the regular expression to be case-sensitive (the default).

(?i:)

> The contents of this group will be matched case-insensitive and the group will not be added to the output.

(?-i:)

> The contents of this group will be matched case-sensitive and the group will not be added to the output.

(?=)
> Positive lookahead assertion. The contents are matched following the current position, but not added to the output pattern.

(?!)
> Negative lookahead assertion. The same as above, but the content must not match following the current position.

(?<=)
> Positive lookbehind assertion. The contents are matched preceding the current position, but not added to the output pattern. The length of possible strings matched by lookbehinds cannot be unbounded (no * or + operators).

(?<!)
> Negative lookbehind assertion. The same as above, but the contents must not match preceding the current position.

`\b`
> Matches the boundary between a word and a space. Does not properly interpret Unicode characters. The transition between any regular ASCII character (matched by `\w`) and a Unicode character is seen as a word boundary.

`\B`
> Matches a boundary not between a word and a space.

`\A`
> Matches the beginning of the input.

`\Z`
> Matches the end of the input.

`^`
> Matches the beginning of the input, or the line if the **m** flag is set.

`$`
> Matches the end of the input, or the line if the **m** flag is set.

Regular Expression Flags

i
> Sets matching to be case-insensitive.

x
> Allows whitespace in comments and patterns.

s
> Allows the **.** character to also match line break characters.

m
> Allows the characters **^** and **$** to match the start and end of lines, respectively. By default these will only match at the start and end of the input.

w
> Changes the behavior of `\b` so that word boundaries are defined according to Unicode Standard Annex #29[41].

20.2 Regexp Type

The **regexp** type allows a regular expression to be defined once and then reused many times. It facilitates simple search operations, splitting strings, and interactive find/replace operations.

---

[41] http://www.unicode.org/reports/tr29/

The **regexp** type has some advantages over the `string_...` methods that perform regular expression operations. Performance can be increased by creating a regular expression once and then reusing it multiple times. The type has a number of member methods that allow access to the stored regular expressions and input and output of strings, performing find/replace operations, or acting as components in an interactive find/replace operation. These are described below.

20.2.1 Creating Regular Expression Objects

type **regexp**

regexp(*find::string*, *replace::string*, *input::string*, *ignorecase::boolean*)

regexp(*find::string*, *replace::string=?*, *input::string=?*, *-ignoreCase::boolean=?*)

regexp(*-find::string*, *-replace::string=?*, *-input::string=?*, *-ignoreCase::boolean=?*)
> The **regexp** creator method creates a reusable regular expression. A **regexp** object must be initialized with a string regular expression pattern as either the first parameter or as the argument of a **-find** keyword parameter. The type will also store a replacement pattern, and input string passed as either the second and third parameters or specified with the **-replace** or **-input** keyword parameter, respectively. These can be overridden with particular member methods. The type also has an **-ignoreCase** option which controls whether regular expressions are applied with case sensitivity or not.
>
> A regular expression can be created that explicitly specifies the find pattern, replacement pattern, input string, and optionally with the **-ignoreCase** option. Using a fully qualified regular expression that is output to the page (rather than being stored in a variable) is an easy way to perform a quick find/replace operation.

```
regexp(`[aeiou]`, 'x', 'The quick brown fox jumped over the lazy dog.')->replaceAll
// => Thx qxxck brxwn fxx jxmpxd xvxr thx lxzy dxg.
```

> However, a regular expression will usually be stored in a variable and then later run against an input string. The following code stores a regular expression with a find and replace pattern into the variable "my_regexp". The following section *Simple Find/Replace and Split Methods* will show how this regular expression can be applied to strings.

```
local(my_regexp) = regexp(-find=`[aeiou]`, -replace=`x`, -ignoreCase)
```

regexp->**findPattern()**
> Returns the find pattern.

regexp->**replacePattern()**
> Returns the replacement pattern.

regexp->**input()**
> Returns the input string.

regexp->**ignoreCase()**
> Returns "true" if the **-ignoreCase** flag has been set, otherwise returns "false".

regexp->**groupCount()**
> Returns an integer specifying how many groups were found in the find pattern.

regexp->**output()**
> Returns the output string.

For example, the regular expression above can be inspected by the following code. The group count is "0" since the find expression does not contain any groups (designated by parentheses):

```
'FindPattern: ' + #my_regexp->findPattern + '\n'
'ReplacePattern: ' + #my_regexp->replacePattern + '\n'
'IgnoreCase: ' + #my_regexp->ignoreCase + '\n'
'GroupCount: ' + #my_regexp->groupCount + '\n'
```

```
// =>
// FindPattern: [aeiou]
// ReplacePattern: x
// IgnoreCase: true
// GroupCount: 0
```

20.2.2 Simple Find/Replace and Split Methods

The **regexp** type provides two member methods that perform a find/replace on an input string and one method that splits an input string into an array. These methods are documented with examples below, and are shortcuts for longer operations that can be performed using the interactive methods described in the next section.

regexp->**replaceAll**(*replace::string*)

regexp->**replaceAll**(*-input=?, -find=?, -replace=?, -ignoreCase=?*)
> The first listed incarnation of this method allows you to change the replacement string. The second will replace all occurrences of the current find pattern with the current replacement pattern. The **-input** parameter specifies what string should be operated on. If no input is provided then the input stored in the regular expression object is used. If desired, new **-find** and **-replace** patterns can also be specified within this method along with the **-ignoreCase** flag.

regexp->**replaceFirst**(*-input=?, -find=?, -replace=?, -ignoreCase=?*)
> Replaces the first occurrence of the current find pattern with the current replacement pattern. The **-input** parameter specifies what string should be operated on. If no input is provided then the input stored in the regular expression object is used. If desired, new **-find** and **-replace** patterns can also be specified within this method along with the **-ignoreCase** flag.

regexp->**split**(*-input=?, -find=?, -replace=?, -ignoreCase=?*)
> Splits the string using the regular expression as a delimiter and returns a staticarray of substrings. The **-input** parameter specifies what string should be operated on. If no input is provided then the input stored in the regular expression object is used. If desired, new **-find** and **-replace** patterns can also be specified within this method along with the **-ignoreCase** flag.

Use the Same Regular Expression on Multiple Inputs

The same regular expression can be used on multiple inputs by first creating the regular expression using one of the **regexp** creator methods and then calling **regexp->replaceAll** with a new **-input** as many times as necessary. Since the regular expression is only created once this technique can be considerably faster than using the **string_replaceRegExp** method repeatedly.

```
local(my_regexp) = regexp(-find=`[aeiou]`, -replace=`x`, -ignoreCase)
#my_regexp->replaceAll(-input='The quick brown fox jumped over the lazy dog.')
#my_regexp->replaceAll(-input='Lasso Server')

// =>
// Thx qxxck brxwn fxx jxmpxd xvxr thx lxzy dxg.
// Lxssx Sxrvxr
```

The replace pattern can also be changed if necessary. The following code changes both the input and replace patterns each time the regular expression is used:

```
local(my_regexp) = regexp(-find=`[aeiou]`, -replace=`x`, -ignoreCase)
#my_regexp->replaceAll(-input='The quick brown fox jumped over the lazy dog.', -replace=`y`)
#my_regexp->replaceAll(-input='Lasso Server', -replace=`z`)
```

```
// =>
// Thy qyyck brywn fyx jympyd yvyr thy lyzy dyg.
// Lzssz Szrvzr
```

The replacement pattern can reference groups from the input using **$1** through **$9**. The following example uses a regular expression to clean up the formatting on a couple of telephone numbers:

```
local(my_regexp) = regexp(`\(((\d{3})\) (\d{3})-(\d{4})`, `$1-$2-$3`)
#my_regexp->replaceAll(-input='(360) 555-1212')
#my_regexp->replaceAll(-input='(800) 555-1212')

// =>
// 360-555-1212
// 800-555-1212
```

Split a String Using a Regular Expression

The **regexp->split** method can be used to split a string using a regular expression as the delimiter. This allows strings to be split into parts using sophisticated criteria. For example, rather than splitting a string on a comma, the "and" before the last item can be taken into account. Or, rather than splitting a string on space, the string can be split into words taking punctuation and other whitespace into account.

The same regular expression from the example above can be used to split a string into substrings. In this case the string will be split on vowels, generating a staticarray with elements containing only consonants or spaces:

```
local(my_regexp) = regexp(-find=`[aeiou]`, -replace=`x`, -ignoreCase)

#my_regexp->split(-input='The quick brown fox jumped over the lazy dog.')
// => staticarray(Th,  q, , ck br, wn f, x j, mp, d , v, r th,  l, zy d, g.)
```

The **-find** pattern can be modified in-place within the **regexp->split** method to split the string on a different regular expression. In this example the string is split on any one of one or more non-word characters. This splits the string into words not including any whitespace or punctuation.

```
#my_regexp->split(-find=`\W+`, -input='The quick brown fox jumped over the lazy dog.')
// => staticarray(The, quick, brown, fox, jumped, over, the, lazy, dog)
```

If the **-find** expression contains groups then they will be returned in the array in between the split elements. For example, surrounding the **-find** pattern above with parentheses will result in an array of alternating word elements and whitespace/punctuation elements.

```
#my_regexp->split(-find=`(\W+)`, -input='The quick brown fox jumped over the lazy dog.')
// => staticarray(The,  , quick,  , brown,  , fox,  , jumped,  , over,  , the,  , lazy,  , dog, .)
```

20.2.3 Interactive Find/Replace Methods

The **regexp** type provides a collection of member methods that make interactive find/replace operations possible. Interactive in this case means that Lasso code can intervene in each replacement as it happens. Rather than performing a simple one-shot find/replace like those shown in the last section, it is possible to programmatically determine the replacement strings using database searches or any logic.

The order of operations of an interactive find/replace operation is as follows:

1. The regular expression object is initialized with a **-find** pattern and **-input** string. In this example the find pattern will match each word in the input string in turn:

   ```
   local(my_regexp) = regexp(
       -find=`\w+`,
       -input='The quick brown fox jumped over the lazy dog.',
       -ignoreCase
   )
   ```

2. A **while** loop is used to advance the regular expression match with **regexp->find**. Each time through the loop the pattern is advanced one match forward. If there are no further matches then the method returns "false" and the loop is exited:

   ```
   while(#my_regexp->find) => {
       // ...
   }
   ```

3. Within the **while** loop the **regexp->matchString** method is used to inspect the current match. If the find pattern had groups then they could be inspected here by passing an integer parameter to **regexp->matchString**:

   ```
   local(match) = #my_regexp->matchString
   ```

4. The match is manipulated. For this example the match string will be reversed using the **string->reverse** method. This will reverse the word "lazy" to be "yzal":

   ```
   #match->reverse
   ```

5. The modified match string is now appended to the output string using the **regexp->appendReplacement** method. This method will automatically append any parts of the input string that weren't matched (the spaces between the words):

   ```
   #my_regexp->appendReplacement(#match)
   ```

6. After the **while** loop the **regexp->appendTail** method is used to append the unmatched end of the input string to the output (the period at the end of the example input):

   ```
   #my_regexp->appendTail
   ```

7. Finally, the output string from the regular expression object is displayed:

   ```
   #my_regexp->output
   // => ehT kciuq nworb xof depmuj revo eht yzal god.
   ```

This same basic order of operation is used for any interactive find/replace operation. The power of this methodology comes in the fourth step where the replacement string can be generated using any code necessary, rather than needing to be a simple replacement pattern.

regexp->**find**(*position::integer=?*)

   Advances the regular expression one match in the input string. Returns "true" if the regular expression was able to find another match, otherwise returns "false". Defaults to checking from the start of the input string (or from the end of the most recent match), but you can optionally pass an integer parameter to set the position in the input string at which to start the search.

regexp->**matchString**(*group::integer=?*)

   Returns a string containing the last pattern match. Optional integer parameter specifies a group from the find pattern to return (defaults to returning the entire pattern match).

regexp->**matchPosition**(*group::integer=?*)
>   Returns a pair containing the start position and length of the last pattern match. Optional integer parameter specifies a group from the find pattern to return (defaults to returning information about the entire pattern match).

regexp->**appendReplacement**(*pattern::string*)
>   Performs a replace operation on the current pattern match and appends the result onto the output string. Requires a single parameter specifying the replacement pattern including group placeholders **$0**–**$9**. Automatically appends any unmatched runs from the input string.

regexp->**appendTail()**
>   The final step in an interactive find/replace operation. This method appends the final unmatched run from the input string into the output string.

regexp->**reset**(*-input=?, -find=?, -replace=?, -ignoreCase=?*)
>   Resets the object. If called with no parameters, the input string is set to the output string. Accepts optional **-find**, **-replace**, **-input**, and **-ignoreCase** parameters.

regexp->**matches**(*position::integer=?*)
>   Returns "true" if the pattern matches the entire input string. Optional integer parameter sets the position in the input string at which to start the search.

regexp->**matchesStart**(*position::integer=?*)
>   Returns "true" if the pattern matches a substring of the input string. Defaults to checking the start of the input string. Optional integer parameter sets the position in the input string at which to start the search.

Perform an Interactive Find/Replace Operation

This example searches for variable names with a dollar sign in an input string and replaces them with variable values. An interactive find/replace operation is used so that the existence of each variable can be checked dynamically as the string is processed.

The string has several words replaced by variable references and each replacement is defined with a replacement word in a map.

```
local(my_string)    = 'The quick $color fox $verb over the lazy $animal.'
local(replacements) = map(
    'color'  = "red",
    'verb'   = "soared",
    'animal' = "ocelot"
)
```

A regular expression is initialized with the input string and a pattern that looks for words beginning with a dollar sign. The word itself is defined as a group within the find pattern. A **while** loop uses **regexp->find** to advance through all the matches in the input string. The method **regexp->matchString** with a parameter of "1" returns the map key for each match. If this key exists then its value is substituted back into output string using **regexp->appendReplacement**, otherwise, the full match is substituted back into the output string with the replacement pattern **$0**. Finally, any remaining unmatched input string is appended to the end of the output string using **regexp->appendTail**.

```
local(my_regexp) = regexp(-find=`\$(\w+)`, -input=#my_string, -ignoreCase)
while(#my_regexp->find) => {
    #my_regexp->appendReplacement(
        #replacements->find(#my_regexp->matchString(1)) or `$0`
    )
}
#my_regexp->appendTail
```

After the operation has completed the output string is displayed:

```
#my_regexp->output
// => The quick red fox soared over the lazy ocelot.
```

20.3 String Methods Taking Regular Expressions

The **string_findRegExp** and **string_replaceRegExp** methods can be used to perform regular expression find and replace routines on text strings.

string_findRegExp(*input, -find::string, -ignoreCase=?*)
> Takes two parameters: a string value and a **-find** keyword parameter. Returns an array with each instance of the **-find** regular expression in the string parameter. Optional **-ignoreCase** parameter uses case-insensitive patterns.

string_replaceRegExp(*input, -find::string, -replace::string, -ignoreCase=?, -replaceOnlyOne=?*)
> Takes three parameters: a string value, a **-find** keyword parameter, and a **-replace** keyword parameter. Returns an array with each instance of the **-find** regular expression replaced by the value of the **-replace** string parameter. Optional **-ignoreCase** parameter uses case-insensitive parameters. Optional **-replaceOnlyOne** parameter replaces only the first pattern match.

20.3.1 Matching Patterns Using string_findRegExp

The **string_findRegExp** method returns an array of items that match the specified regular expression within the string. The array contains the full matched string in the first element, followed by each of the matched subexpressions in subsequent elements.

In the following example, every email address in a string is returned in an array:

```
string_findRegExp(
    'Send email to address@example.com.',
    -find=`\w+@\w+\.\w+`
)

// => array(address@example.com)
```

In the following example, every email address in a string is returned in an array and subexpressions are used to divide the username and domain name portions of the email address. The result is an array with the entire match string, then each of the subexpressions.

```
string_findRegExp(
    'Send email to address@example.com.',
    -find=`(\w+)@(\w+\.\w+)`
)

// => array(address@example.com, address, example.com)
```

In the following example, every word in the source is returned in an array. The first character of each word is separated as a subexpression. The returned array contains 16 elements, one for each word in the source string and one for the first character subexpression of each word in the source string.

```
string_findRegExp(
    `The quick brown fox jumped over a lazy dog.`,
    -find=`(\w)\w*`
)

// => array(The, T, quick, q, brown, b, fox, f, jumped, j, over, o, a, a, lazy, l, dog, d)
```

The resulting array can be divided into two arrays using the following code. This code loops through the array (stored in **result_array**) and places the odd elements in the array **word_array** and the even elements in the array **char_array**.

```
local(word_array, char_array) = (: array, array)
local(result_array) = string_findRegExp(
    `The quick brown fox jumped over a lazy dog.`,
    -find=`(\w)\w*`
)
with key in #result_array->keys
let value = #result_array->get(#key)
do {
    if(#key % 2 == 0) => {
        #char_array->insert(#value)
    else
        #word_array->insert(#value)
    }
}

#word_array
// => array(The, quick, brown, fox, jumped, over, a, lazy, dog)

#char_array
// => array(T, q, b, f, j, o, a, l, d)
```

In the following example, every phone number in a string is returned in an array. The **\d** symbol is used to match individual digits and the **{3}** symbol is used to specify that three repetitions must be present. The parentheses are escaped **\(** and **\)** so they aren't treated as grouping characters.

```
string_findRegExp(
    'Phone (800) 555-1212 for information.',
    -find=`\(\d{3}\) \d{3}-\d{4}`
)

// => array((800) 555-1212)
```

In the following example, only words contained between HTML bold tags are returned. Positive lookahead and lookbehind assertions are used to find the contents of the tags without the tags themselves. Note that the pattern inside the assertions uses a non-greedy modifier.

```
string_findRegExp(
    'This is some <b>sample text</b>!',
    -find=`(?<=<b>).+?(?=</b>)`
)

// => array(sample text)
```

20.3.2 Replacing Values Using string_replaceRegExp

In the following example, every occurrence of the world "Blue" in the string is replaced by the HTML code **Blue** so that the word "Blue" appears in blue on the web page. The **-find** parameter is specified so either a lowercase or uppercase "b" will be matched. The **-replace** parameter references **$1** to insert the actual value matched into the output.

```
string_replaceRegExp(
    'Blue Lake sure is blue today.',
    -find=`([Bb]lue)`,
```

```
    -replace=`<span style="color: blue;">$1</span>`
)

// => <span style="color: blue;">Blue</span> Lake sure is <span style="color: blue;">blue</span> today.
```

In the following example, every email address is replaced by an HTML anchor tag that links to the same email address. The **\w** symbol is used to match any alphanumeric characters or underscores. The at sign (**@**) matches itself. The period must be escaped (**\.**) in order to match an actual period and not just any character. This pattern matches any email address of the format *name@example.com*:

```
string_replaceRegExp(
    'Send email to address@example.com.',
    -find=`(\w+@\w+\.\w+)`,
    -replace=`<a href="mailto:$1">$1</a>`
)

// => Send email to <a href="mailto:address@example.com">address@example.com</a>.
```

Collections

Lasso provides a variety of collection data types for storing data in an ordered and unordered manner. Objects of these types contain zero or more other arbitrary objects. Built-in support is provided for common collection types such as arrays, lists, maps, and others.

21.1 Ordered Collection Types

Ordered collections store their elements positioned by the order in which they are inserted. The element inserted first into an ordered collection will always be first unless subsequently repositioned. Lasso provides support for **pair**, **array**, **staticarray**, **list**, **queue**, and **stack** types.

21.1.1 Pair Type

type **pair**
> Pairs are one of the most basic of collections. A pair always contains two elements. These are referred to as the "first" and "second" elements and are accessed through methods of the same name.

Creating Pair Objects

pair()

pair(*p::pair*)

pair(*value, value*)

pair(*value=value*)
> A pair is created in one of four ways. First, a zero parameter call to the **pair** method will generate a pair with the first and second values set to "null". Second, a pair can be created by passing it another pair. This will set the first and second values to the first and second values from the passed pair. Third and fourth, a pair can be created by specifying the first and second values as parameters or a key and value when calling the **pair** method.

Using Pair Objects

pair->**first**()

pair->**second**()
> These methods return the first or second element of a pair.

pair->**first=**(*value*)

pair->**second=**(*value*)
> These methods set the first or second element of a pair to the passed value.

21.1.2 Array Type

type **array**

Array objects store zero or more elements and provide random access to those elements by position. Positions are 1-based integers. Arrays will grow as needed to accommodate new elements. Elements can be inserted and removed from arrays at any position. However, inserting an element in any position except at the end of an array results in all subsequent elements being moved down. Therefore, arrays are best used when inserting or removing only at the end of the array.

Creating Array Objects

array()

array(value, …**)**

An array can be created with zero or more parameters. All parameters passed to the **array** method will be inserted into the new array.

Using Array Objects

array->insert(value, position::integer=?**)**

This method adds a new element to the array. Elements are added to the end of the array by default, but a second parameter permits the position of the insertion to be specified. Position 1 is at the beginning of the array. Position zero and negative positions will cause the method to fail. A position larger than the size of the array will insert the element at the end.

array->remove(position::integer=?**)**

array->remove(position::integer, count::integer**)**

array->removeAll(matching=?**)**

These methods remove one or more elements from the array. Calling **remove** with no parameters removes the last element from the array, while **remove** with a **position** parameter will remove the element from that location. All subsequent elements must then be moved up to fill the slot. A second **count** parameter can be specified to indicate that more that one element should be removed, starting from the indicated position.

The **removeAll** method with no parameters will remove all elements from the array. The second form takes one parameter. All elements in the array to which the parameter compares equally will be removed.

array->get(position::integer**)**

array->get=(value, position::integer**)**

The **get** method returns the element located at the indicated position. The method will fail if the position is out of range. The setter version of this method allows the position to be assigned a new value, e.g.:

```
#array->get(2) = "I am the second element!"
```

array->sub(position::integer, count::integer=?**)**

Returns a range of elements from the array. The first parameter indicates the starting position and the second parameter indicates how many of the elements to return.

array->first()

array->second()

array->last()

These methods return the first, second, and last elements from the array, respectively. If the array does not have an element for that position, "null" will be returned.

array->**contains**(*matching*) → boolean

array->**count**(*matching*) → integer

array->**findPosition**(*matching*, *startPosition=1*)

array->**find**(*matching*)

These methods search the array for elements matching the parameter. The **contains** method returns "true" if the matching parameter compares equally to any contained elements. The **count** method returns the number of matching elements. The **findPosition** method returns the position at which the next matching element can be found with an optional second parameter indicating where the search should begin. The **find** method returns a new array containing all of the matched objects.

array->**size**() → integer

Returns the number of elements in the array.

array->**sort**(*ascending::boolean=true*)

Performs a sort on the elements. Elements are repositioned in either ascending or descending order depending on the given parameter.

array->**join**(*delimiter::string=""*) → string

Joins all the elements as strings with the **delimiter** parameter between each.

Example of joining an array of numbers:

```
array(1, 2, 3, 4, 5)->join(', ')
// => 1, 2, 3, 4, 5
```

array->**asStaticArray**() → staticarray

Returns the array's elements in a new staticarray.

array->**+**(*rhs::trait_forEach*) → array

Arrays can be combined with other collection types by using the **+** operator. A new array containing all the elements is returned.

Example of combining an array, staticarray, and pair into a new array:

```
array(1, 2, 3, 4, 5) + (: '6', '7', '8') + pair('nine', 'ten')
// => array(1, 2, 3, 4, 5, 6, 7, 8, nine, ten)
```

21.1.3 Staticarray Type

type **staticarray**

A staticarray is a collection object that is created with a fixed size and is not resizable. Positions within the staticarray can be reassigned different objects, but new positions cannot be added or removed. Staticarrays are designed to be as efficient as possible both in the time used to create a new object and in the memory used for the object itself. The elements of a staticarray are accessed randomly, like an array, with 1-based positions.

Lasso provides a shortcut for creating staticarray objects through the **(:)** syntax. This syntax begins with an open parenthesis followed by a colon, then zero or more elements, finalized by a closing parenthesis.

Creating Staticarray Objects

staticarray()

staticarray(*value*, *...*)

staticarray_join(*count::integer*, *value*)
> The first two methods create a new staticarray given zero or more elements. The last method, **staticarray_join**, creates a new staticarray of the given size with each element filled by the value given as the second parameter.
>
> Example of creating a few staticarrays:

```
// staticarray with no elements
(:)

// staticarray with variety of elements
(: 1, 2, 8, 'Hi!', pair(1, 2))

// staticarray with 12 elements set to void
staticarray_join(12, void)
```

Using Staticarray Objects

staticarray->get(*position::integer*)

staticarray->get=(*value*, *position::integer*)
> The **get** method returns the element at the indicated position. This method will fail if the position is out of range. The **get** method also permits a position to be reassigned with an assignment statement in the same manner as **array->get=**.

staticarray->first()

staticarray->second()

staticarray->last()
> The first, second, and last methods return the corresponding element or "null" if there is no element at the position.

staticarray->contains(*matching*) → boolean

staticarray->findPosition(*matching*, *startPosition=1*)

staticarray->find(*matching*)
> These methods search the staticarray for elements matching the parameter. The **contains** method returns "true" if the matching parameter compares equally to any contained elements. The **findPosition** method returns the position at which the next matching element can be found with an optional **startPosition** parameter indicating where the search should begin. The **find** method returns a new array containing all of the matched objects.

staticarray->join(*count::integer*, *value*) → staticarray

staticarray->join(*s::staticarray*) → staticarray
> These methods combine the staticarray with other elements to create a new staticarray. The first method adds the number of positions indicated by the first parameter and fills them with the value specified by the second parameter. The second method combines the staticarray with the passed staticarray to produce a new staticarray containing the elements from both.
>
> Example of joining new elements into a new staticarray:

```
(: 1, 2, 3)->join(5, 'Hi')
// => staticarray(1, 2, 3, Hi, Hi, Hi, Hi, Hi)

(: 1, 2, 3)->join((: 4, 5, 6))
// => staticarray(1, 2, 3, 4, 5, 6)
```

staticarray->sub(*position::integer*, *count::integer=?*) → staticarray
> The **sub** method returns a range of elements. The first parameter indicates the starting position and the optional second parameter indicates how many of the elements to return. The elements are returned as a new staticarray.

staticarray->+(*s::staticarray*) → staticarray

staticarray->+(*value*) → staticarray
> The **+** operator can be used with staticarrays to create a new staticarray with the additional elements. The first variant returns a new staticarray with all the elements from the two staticarrays, and the second returns a staticarray with all the elements of the first and the additional element on the right-hand side of the operator.

21.1.4 List Type

type **list**
> A list presents a series of objects stored in a linked manner. Elements can be efficiently added or removed from a list at the end or the beginning, but cannot be added into the middle. Lists do not support random access, so the only way to get particular elements from a list is through one of the iterative constructs such as *query expressions*.

Creating List Objects

list()

list(*value, …*)
> The **list** method creates a new list object using the parameters given as the elements for the list.

Using List Objects

list->**insertFirst**(*value*)

list->**insertLast**(*value*)

list->**insert**(*value*)
> These methods insert new elements into the list. Elements can be inserted at the beginning or the ending of the list. The **insert** method with no parameters inserts at the end of the list.

list->**removeFirst**()

list->**removeLast**()

list->**remove**()
> These methods remove an element from the list. Either the first or the last element can be removed. The **remove** method with no parameters removes the last element.

list->**removeAll**(*matching=?*)
> The first **removeAll** method with no parameters removes every element from the list. The second form accepts a parameter which is compared against the elements. All matching elements are removed from the list.

list->**first**()

list->**last**()
> These methods returns the first and last elements, respectively.

list->**contains**(*matching*) → boolean
> Takes one parameter and compares it against the elements in the list. It returns "true" if the list contains a match.

21.1.5 Queue Type

type **queue**
> Queue objects store data in a "first in, first out" (FIFO) manner. Elements can efficiently be inserted into the end of the queue (called "pushing") and removed from the front of the queue (called "popping"). Queues do not support random

access, so the only way to get particular elements from a queue is through one of the iterative constructs such as *query expressions*.

Creating Queue Objects

queue()

queue(*value, …***)**
>   Creates a queue object using the parameters passed to it as the elements of the queue.

Using Queue Objects

queue->**insert(***value***)**

queue->**insertLast(***value***)**

queue->**insertFrom(***value::trait_forEach***)**
>   These methods insert new elements into the queue. Elements will always be inserted at the end of the queue. The **insertFrom** method allows for multiple elements to be inserted into the queue by taking an object that implements `trait_forEach`.

queue->**first()**

queue->**get()**
>   These methods return the first element in the queue. (This is the least recently inserted element.) The **get** method additionally removes the element from the queue.

queue->**size()**
>   Returns the number of elements in the queue.

queue->**remove()**

queue->**removeFirst()**
>   These methods remove the first element in the queue. (This is the least recently inserted element.)

queue->**unspool(***i::integer=?***)**
>   This method returns a staticarray of the elements in the queue and removes them from the queue. The number of elements to return and remove can be specified as an integer parameter to this method.

21.1.6 Stack Type

type **stack**
>   *Deprecated since version 9.2:* Use **array** instead.

>   Stack objects store data in a "last in, first out" (LIFO) manner. Elements can efficiently be inserted into the beginning of the stack (called "pushing") and removed from the beginning of the stack (called "popping"). Stacks do not support random access, so the only way to get particular elements from a stack is through one of the iterative constructs such as *query expressions*.

Creating Stack Objects

stack()

stack(*value, …***)**
>   Creates a stack object using the parameters passed to it as the elements of the stack.

Using Stack Objects

stack->**insert**(*value*)

stack->**insertFirst**(*value*)
> These methods insert new elements into the stack. Elements will always be inserted at the beginning of the stack.

stack->**first**()

stack->**get**()
> These methods return the first element in the stack. (This is the most recently inserted element.) The **get** method additionally removes the element from the stack.

stack->**size**()
> Returns the number of elements in the stack.

stack->**remove**()

stack->**removeFirst**()
> These methods remove the first element in the stack. (This is the most recently inserted element.)

21.2 Unordered Collection Types

Unordered collections store their elements without position-based ordering. Lasso supports two unordered collection types: **map** and **set**. Maps provide access to the elements via separate keys. Sets store only the elements themselves.

21.2.1 Map Type

type **map**
> Maps are used to store values along with associated keys. An element can later be found given the key value with which it was inserted. New elements can be inserted or removed freely from a map. Only one element can be stored for any given key and inserting a duplicate key will replace any existing element.
>
> The keys used in a map can be of any type, provided that type has a suitable **onCompare** method. Keys must compare themselves consistently such that if **A < B** then always **B >= A**. Most built-in Lasso types, such as strings, integers, and decimals, fit this criteria.

Creating Map Objects

map()

map(*p::pair*, ...)

map(*key=value*, ...)
> A map is created with zero or more key/value pair parameters. Any non-pair parameters given are inserted as a key with a "null" value.
>
> Example of creating a map with a series of parameters using string-based keys:

```
local(myMap) = map(
    'C' = 247,
    'L' = "Hi!",
    'G' = 97.401,
    'N' = array(4, 5, 6)
)
```

Using Map Objects

map->**insert**(*p::pair*, …)

map->**insert**(*key=value*, …)
> Inserts a new key/value pair into the map. If the key specified already exists, it is replaced.

map->**remove**(*key*)

map->**removeAll**(*matching=?*)
> The first method, **remove**, removes the indicated key/value from the map. If the key does not exist in the map then no action is taken. The second method, **removeAll** with no parameters, removes all of the keys/values from the map. If called with a parameter, all keys matching that parameter are removed.

map->**get**(*key*)

map->**get=**(*value*, *key*)

map->**find**(*key*)

map->**contains**(*key*) → boolean
> These methods get particular elements from the map or test that a key is contained within the map. The **get** method finds the element within the map associated with the key and returns the value, or reassigns it if a new value is assigned. If the key is not found the method will fail. The **find** method will search for the key within the map and return the value if it exists. If the key is not found the method will return "void". The **contains** method returns "true" if the matching parameter compares equally to any contained elements.

map->**size()** → integer
> Returns the number of elements contained within the map.

21.2.2 Set Type

type **set**
> A set contains only unique elements. Each element is itself a key. Sets support quickly determining if an object is contained within it. Elements within a set must be able to **onCompare** themselves just as described for **map** keys.

Creating Set Objects

set()

set(*key*, …)
> A set is created with zero or more element parameters. The element values are inserted into the set.

Using Set Objects

set->**find**(*key*)

set->**get**(*key*)

set->**contains**(*key*) → boolean
> These methods find the given key within the set. The **find** method will return the key if it is found; it returns "void" if the key is not within the set. The **get** method will return the key, but will fail if the key is not contained within the set. The **contains** method returns "true" if the key is in the set.

set->**insert**(*key*)
> Inserts the key into the set. Any duplicate key value is replaced.

set->**remove**(*key*)

```
set->removeAll()
```
> The **remove** method removes the indicated key from the set. If the key is not contained within the set then no action
> is taken. The **removeAll** method removes all keys from the set.

Encryption

Lasso provides a set of data encryption methods which support the most common encryption and hash functions used on the Internet today. These encryption methods make it possible to interoperate with other systems that require encryption and to store data in a secure fashion within data sources or files.

Lasso has built-in methods for the BlowFish encryption algorithm and for the SHA1 and MD5 hash algorithms.

Lasso's cipher methods provide access to a wide range of industry-standard encryption algorithms. The **cipher_list** method lists which algorithms are available on your system and the **cipher_encrypt**, **cipher_decrypt**, and **cipher_digest** methods allow values to be encrypted, decrypted, or digest values to be generated, respectively.

22.1 Encryption Methods

Lasso provides a number of methods that allow data to be encrypted for secure storage or transmission. Three different types of encryption are supplied:

BlowFish
> This is a fast, popular encryption algorithm. Lasso provides tools to encrypt and decrypt string values using a developer-defined seed. This is the best method to use for data that needs to be stored in a database or transmitted securely and decrypted later.

MD5
> This is a one-way cryptographic hash algorithm that is often used to verify file integrity. MD5 is generally considered unsuitably weak for security purposes.

SHA1
> This is a one-way cryptographic hash algorithm that is often used for passwords. There is no way to decode data that has been hashed using SHA1.

encrypt_blowfish(*plaintext*, *-seed::string*)
> Encrypts a string using the industry-standard BlowFish algorithm. Accepts two parameters, a string to be encrypted and a **-seed** keyword parameter with the key or password for the encryption. It returns an encrypted bytes object.
>
> The BlowFish methods are not binary-safe, so the output of the method will be truncated after the first null character. It is necessary to use **encode_base64**, **encode_hex**, or **encode_utf8** prior to using this method to encrypt data that might contain binary characters.

decrypt_blowfish(*cipherText*, *-seed::string*)
> Decrypts a byte stream encrypted using the industry-standard BlowFish algorithm. Accepts two parameters, a byte stream to be decrypted and a **-seed** keyword parameter with the key or password for the decryption. Returns a decrypted bytes object.

encrypt_md5(*data::bytes*) → string

encrypt_md5(*data::any*) → string
> Hashes a string using the one-way MD5 hash algorithm. Accepts one parameter, the data to be hashed. Returns a fixed-size hash value in hexadecimal as a string.

encrypt_hmac(*-password, -token, -digest=?, -base64::boolean=?, -hex::boolean=?, -cram::boolean=?, ...* **)**

Generates a keyed hash message authentication code for a given input and password. The method requires a **-password** parameter to specify the key for the hash and a **-token** parameter to specify the text message that is to be hashed. These parameters should be specified as a string or as a byte stream. The digest algorithm used for the hash can be specified using an optional **-digest** parameter. The digest algorithm defaults to "MD5". "SHA1" is another common option. However, any of the digest algorithms returned by **cipher_list(-digest)** can be used.

The output is a byte stream by default. The **-base64** parameter specifies the output should be a Base64 encoded string. The **-hex** parameter specifies the output should be a hex format string like "0x0123456789abcdef". The **-cram** parameter specifies the output should be in a cram hex format like "0123456789ABCDEF".

22.1.1 BlowFish Seeds

BlowFish requires a seed in order to encrypt or decrypt a string. The same seed that was used to encrypt data using the **encrypt_blowfish** method must be passed to the **decrypt_blowfish** method to decrypt that data. If you lose the key used to encrypt data then the data will be essentially unrecoverable.

Seeds can be any string between 4 characters and 112 characters long. Pick the longest string possible to ensure a secure encryption. Ideal seeds contain a mix of letters, digits, and punctuation.

> *Caution:* The security considerations of storing, transmitting, and hard-coding seed values is beyond the scope of this book. The examples that follow present methodologies that are easy to use, but may not provide the highest level of security possible. You should consult a security expert if security is very important for your website.

22.1.2 Store Encrypted Data in a Database

Use the **encrypt_blowfish** and **decrypt_blowfish** methods to encrypt data that will be stored in a database and then decrypt the data when it is retrieved from the database.

In the example below, the data in the variable "plaintext" is encrypted and stored in the "ciphertext" variable. This is then used to store the data in the "ciphertext" field of the "people" table in the "contacts" database.

```
local(plaintext)  = 'The data to be encrypted.'
local(ciphertext) = encrypt_blowfish(#plaintext, -seed='My Insecure Seed')

inline(
   -add,
   -database='contacts',
   -table='people',
   -keyField='id',
   'first_name'='John',
   'last_name'='Doe',
   'ciphertext'=encode_base64(#ciphertext)
) => {}
```

The example below retrieves the record created above and places the Base64-decoded value of the "ciphertext" field in a variable of the same name. It then decrypts the data into the "plaintext" variable and displays that variable.

```
inline(
   -search,
   -database='contacts',
   -table='people',
   -keyField='id',
   'first_name'='John',
```

```
    'last_name'='Doe'
) => {
    local(ciphertext) = decode_base64(field('ciphertext'))
}

local(plaintext) = decrypt_blowfish(#ciphertext, -seed='My Insecure Seed')
#plaintext

// => The data to be encrypted.
```

22.1.3 Store and Check Hashed Passwords

The **encrypt_md5** method can be used to store a secure version of a password for a site visitor. On every subsequent visit, the password given by the visitor is hashed using the same method and compared to the stored value. If they match, then the visitor has supplied the same password they initially created.

The following example takes a visitor-supplied password from a form and stores it hashed using MD5 into the "people" table in the "contacts" database:

```
local(visitor_password) = web_request->param('password')
inline(
    -add,
    -database='contacts',
    -table='people',
    -keyField='id',
    'first_name'='John',
    'last_name'='Doe',
    'username'='dodo',
    'password'=encrypt_md5(#visitor_password)
) => {}
```

On subsequent visits, the visitor would be prompted for their username and password. The following example shows how to verify the credentials they supply via a form:

```
local(username) = web_request->param('username')
local(password) = web_request->param('password')

inline(
    -search,
    -database='contacts',
    -table='people',
    -keyField='id',
    'username'=#username,
    'password'=encrypt_md5(#password)
) => {
    local(is_authenticated) = (found_count > 0)
}
if(#is_authenticated) => {
    // ... login successful ...
else
    // ... credentials don't match ...
}
```

Important: For more security, most login solutions require both a username and a password. Also, many login solutions restrict the number of login attempts that they will accept from a client's IP address, use salts, and iterate over the hashing algorithm

thousands of times. Again, you should consult a security expert if security is very important for your website.

22.2 Cipher Methods

Lasso includes a set of methods that allow access to a wide variety of encryption algorithms. These cipher methods provide implementations of many industry-standard encryption methods and can be very useful when communicating using Internet protocols or communicating with legacy systems. The **cipher_list** method can be used to list which algorithms are supported on a particular Lasso installation.

Note: The actual list of supported algorithms may vary between Lasso installations depending on the platform and system version. The algorithms listed in this guide should be available on all systems, but other more esoteric algorithms may be available on some systems and not on others.

cipher_encrypt(*data*, *-cipher::string*, *-key*, *-seed=?*) → bytes
  Encrypts a string using a specified algorithm. Requires three parameters: the data to be encrypted, a **-cipher** keyword parameter specifying which algorithm to use, and a **-key** keyword parameter specifying the key for the algorithm. An optional **-seed** parameter can be used to seed some algorithms with a random component.

cipher_decrypt(*data*, *-cipher::string*, *-key*, *-seed=?*) → bytes
  Decrypts a string using a specified algorithm. Requires three parameters: the data to be decrypted, a **-cipher** keyword parameter specifying which algorithm to use, and a **-key** keyword parameter specifying the key for the algorithm. An optional **-seed** parameter can be used to seed some algorithms with a random component.

cipher_digest(*data*, *-digest*, *-hex::boolean=?*) → bytes
  Hashes data using a specified digest algorithm. Requires two parameters: The data to be encrypted and a **-digest** parameter that specifies the algorithm to be used. Optional **-hex** parameter encodes the result as a hexadecimal string.

cipher_list(*-digest::boolean=?*)
  Lists the algorithms that the cipher methods support. With the optional **-digest** parameter, it returns only digest algorithms.

The following list some of the cipher algorithms that can be used with **cipher_encrypt** and some of the digest algorithms that can be used with **cipher_digest**. Use **cipher_list** for a full list of supported algorithms.

AES
  Advanced Encryption Standard. A symmetric key encryption algorithm which is the replacement for DES. An implementation of the Rijndael algorithm.

DES
  Data Encryption Standard. A block cipher developed by IBM in 1977 and previously used as the government standard encryption algorithm for years.

3DES
  Triple DES. This algorithm uses the DES algorithm three times in succession with different keys.

RSA
  A public key algorithm named after Rivest, Shamir, and Adleman. One of the most commonly used encryption algorithms. (Note that Lasso does not generate public/private key pairs.)

DSA
  Digital Signature Algorithm. Part of the Digital Signature Standard. Can be used to sign messages, but not for general encryption.

SHA1
  Secure Hash Algorithm. Produces a 160-bit hash value. Used by DSA.

MD5

> Message Digest. A hash function that generates a 128-bit message digest. Replaces the MD4 and MD2 algorithms (which are also supported). Also implemented in Lasso as **encrypt_md5**.

22.2.1 List All Supported Algorithms

Use the **cipher_list** method. The following example will return a list of all the cipher algorithms supported by this installation of Lasso:

```
cipher_list
// => staticarray(DES-ECB, DES-EDE, DES-CFB, DES-OFB, DES-CBC, DES-EDE3-CBC, \
//                RC4, RC2-CBC, BF-CBC, CAST5-CBC, RC5-CBC)
```

With a **-digest** parameter the method will limit the returned list to all of the digest algorithms supported by this installation of Lasso:

```
cipher_list(-digest)
// => staticarray(MD2, MD4, MD5, SHA, SHA1, DSA-SHA, DSA, RIPEMD160)
```

22.2.2 Calculate a Digest Value

Use the **cipher_digest** method. The following example will return the DSA signature for the value of a database field "message":

```
cipher_digest(field('message'), -digest='DSA')
```

22.2.3 Encrypt a Value Using 3DES

Use the **cipher_encrypt** method. The following example will return the 3DES encryption for the value of a database field "message":

```
cipher_encrypt(field('message'), -cipher='DES-EDE3-CBC', -key='My Very Secret Key For 3DES')
```

Serialization and Compression

To *serialize* an object is to convert the object into a format that can be transmitted over the network or written to a file. The serialized object data can then later be used to *deserialize*—or re-create—the object.

Lasso uses XML for object serialization. An object whose type supports serialization can be converted to and from XML, or can be stored in a session. The object is given control over which of its data members will be written to the output.

Lasso also provides a set of methods to compress or decompress data for more efficient data transmission.

23.1 Serializing and Deserializing Objects

An object is serialized by calling its **serialize** method, which serializes the object and returns the resulting data as a string. This method is provided through **trait_serializable**, which is described below.

Serialized object data is converted back into an object by using a **serialization_reader** object. This object is created with the serialized data after which its **read** method is called. If the read is successful, then a new object of the same type and data as the original serialized object is returned.

type **serialization_reader**

serialization_reader(*s::string*)

serialization_reader(*x::xml_element*)
>   Creates a **serialization_reader** object. Can be instantiated with a string of XML or an **xml_element** object.

serialization_reader->read(*x::xml_element=?*)
>   Re-creates the serialized element.

This example code serializes an array of objects, then deserializes it back into a new array:

```
local(a)    = array(1, 2, 'three', pair(4='five'))
local(data) = #a->serialize
local(a2)   = serialization_reader(#data)->read

#a == #a2
// => true
```

23.2 Supporting Serialization

In order to be serializable, an object must meet a few requirements. When creating new object types, these requirements must be met or the objects will not be serializable. Additionally, any objects contained by a serializable type must themselves also be serializable in order to be properly handled.

Serializable objects must implement the following methods:

trait **trait_serializable**

require **trait_serializable->onCreate()**

  A serializable object must implement a zero parameter **onCreate** method. Note that if a type has no **onCreate** methods at all, then a suitable method is automatically added to the type to allow this requirement to be met. During deserialization, a new instance of the object is created. No parameters are passed at that point.

require **trait_serializable->serializationElements()** → trait_forEach

  This method is called during object serialization. It should return an array, staticarray, or some other suitable object containing each of the elements that should be serialized along with the target object.

  Each element in the return value should be a **serialization_element**. These objects contain a key and a value. The key and the value must both be serializable. The key and the value can be objects of any type. They are both given back to the object when it is deserialized in order to return it to the state it was in when it was serialized to begin with.

require **trait_serializable->acceptDeserializedElement(***d::serialization_element***)**

  As an object is deserialized by a **serialization_reader**, first a new instance is created, then this method is called once for each of the serialization elements that were originally included in the data. The **serialization_element** items contain the keys and values used to re-create the original object state.

Implementing the proper methods allows the object to import **trait_serializable**, which provides the **serialize** method. This trait should be added when the type is defined.

provide **trait_serializable->serialize()** → string

  Serializes the object and returns the resulting data. That data can then be deserialized, re-creating an object with the correct data.

serialization_element objects are used when both serializing and deserializing. This simple object must be created with a key and a value. The key and value are made available through methods named accordingly.

type **serialization_element**

serialization_element(*key, value***)**

  Create a new **serialization_element** object with a key and value.

serialization_element->key()

serialization_element->value()

  These methods respectively return the key and value that was set when the object was created. Both the key and value can be objects of any serializable type.

23.2.1 Serializable Type Example

This example illustrates how to create a new object type that is serializable. The example type has data members that are saved during serialization.

```
define example_obj => type {
    trait { import trait_serializable }

    data public dmem1 = 'Value for first member',
        public dmem2 = 'Second member\'s value'

    public serializationElements()::trait_forEach => {
        return (:
            serialization_element(1, .dmem1),
            serialization_element(2, .dmem2)
        )
    }

    public acceptDeserializedElement(d::serialization_element) => {
        match(#d->key) => {
```

```
            case(1)
                .dmem1 = #d->value
            case(2)
                .dmem2 = #d->value
        }
    }
}

local(
    obj = example_obj,
    data = #obj->serialize,
    new = serialization_reader(#data)->read
)
#new->dmem1

// => Value for first member
```

23.3 Compression Methods

Lasso provides two methods that allow data to be stored or transmitted more efficiently. The **compress** method can be used to compress any text string into an efficient byte stream that can be stored in a binary field in a database or transmitted to another server. The **decompress** method can then be used to restore a compressed byte stream into the original string.

compress(*b::bytes*)

compress(*s::string*)
> Compresses a string or bytes object.

uncompress(*b::bytes*)

decompress(*b::bytes*)
> Decompresses a byte stream.

The compression algorithm should only be used on large string values. For strings of less than one hundred characters the algorithm may actually result in a larger string than the source.

These methods can be used in concert with the **serialize** method which creates a string representation of a type that implements **trait_serializable**, and the **serialization_reader->read** method which returns the original value based on a string representation.

23.3.1 Compress and Decompress a String

The following example takes the string value stored in the variable "input" and compresses it and stores that information in "smaller". Finally, it decompresses the data into the variable "output" and then displays the value now stored in output.

```
local(input)   = 'This is the string to be compressed.'
local(smaller) = compress(#input)
local(output)  = decompress(#smaller)
#output

// => This is the string to be compressed.
```

23.3.2 Compress and Decompress an Array

The following example takes an array value stored in "my_array" and serializes the data into the "input" variable. It then compresses that data into the "smaller" variable. The "output" variable is then set to the decompressed and deserialized value stored in the "smaller" variable. The value in "output" is then displayed.

```
local(my_array) = array('one', 'two', 'three', 'four', 'five')
local(input)    = #my_array->serialize
local(smaller)  = compress(#input)
local(output)   = serialization_reader(xml(decompress(#smaller)))->read
#output

// => array(one, two, three, four, five)
```

Part IV

System Input and Output

File System

Lasso provides access to the local file system through the **file** and **dir** types. File objects are used to create, delete, read, and write file data. Dir objects are use to create and delete directories and to iterate through directory contents. Each are front ends for the **filedesc** and **dirdesc** types, which are internal interfaces used by these and other methods for communication with the filesystem and other processes (e.g. **net_tcp**, **split_thread**).

24.1 Paths

Individual files and directories are identified by their paths. Paths may include "." or "" components to indicate "parent" or "current" locations, respectively. Path components are generally separated by forward slashes, though backward slashes are acceptable as well and may be more natural on Windows operating systems. Regardless of which type of slash is used, Lasso will normalize all paths to match the conventions of the operating system before using the path in any system function.

Paths can be either relative or full. Full paths always start with at least one slash, or in the case of Windows, may start with a drive letter designation (e.g. "C:"). Full file paths are based from the file system root. When serving web requests under Lasso Server, the file system root defaults to the host document root as indicated by the web server for that request (IIS, Apache, etc.) or as set by the **LASSOSERVER_DOCUMENT_ROOT** web request variable. This applies to the current thread only. Any new threads will not inherit the request-specific file system root.

It is possible to escape the host document root and target the real file system root by using a full path with either a drive letter designation in the case of Windows, or by prefixing the path with two additional forward slashes. For example, "//foo/bar" and "C:\foo\bar" would both reference the same file on Windows, provided "C:" is the system drive.

When not serving a web request, such as when running items from "LassoStartup" or when running scripts through the **lasso9** command-line tool, the file system root is set to the system's natural root which is "/" for UNIX-based systems or "C:" (for example) on Windows-based systems.

Relative paths do not begin with a slash or drive designation and indicate a file or directory that is located based on the current working directory. During a web request, the current working directory is the directory location of the currently active source file. For example, when processing a request for the file "/foo/bar.lasso", "/foo/" is the current working directory and a file with a relative path of "baz.lasso" will be looked for as "/foo/baz.lasso". To illustrate, consider the following three example files. Within the first two are tests checking for the existence of the next file.

```
/test.lasso - file 'dir/test.lasso' exists
/dir/test.lasso - file 'dir2/test.lasso' exists
/dir/dir2/test.lasso
```

When not serving a web request or when running shell scripts via lasso9, the current working directory is as set by the operating system or shell. In this situation, the current working directory path can be retrieved with the **io_file_getcwd** method, and the current working directory can be set with the **io_file_chdir** method. Manipulating the working directory in this way changes it globally for all threads in the current process.

24.2 File Type

type **file**

file()

file(*path::string***)**
>   File objects can be instantiated with or without an initial path. Creating a file object does not open the file. If created
>   without a path, a path must be specified when later opening the file.

24.2.1 Opening Files

A file must be opened before it can be read from or written to. Once a file is opened, it should be closed when it is no longer
needed. While Lasso will close all files that become garbage-collected, it is recommended to immediately close files once their
tasks are completed. Many operating systems have limitations on the number of simultaneously opened files, and ensuring
that they are closed promptly will improve system performance.

file->openRead()

file->openWrite()

file->openWriteOnly()

file->openAppend()

file->openTruncate()
>   These methods open the file using the open mode indicated in the method name.
>
>   - **openRead** will open the file in read-only mode.
>
>   - **openWrite** will open the file in read/write mode.
>
>   - **openAppend** will open the file in read/write mode and will set the current write position to the end of the file.
>
>   - **openTruncate** will open the file in read/write mode and will set the file's size to zero.
>
>   Write, append, and truncate modes will create the file if it does not exist. Read-only mode will fail if the file does not
>   exist.
>
>   All the methods will fail if the process does not have access to the file in question. In this case the **error_code** and
>   **error_msg** will be set to the values generated by the operating system.

file->openRead(*path::string***)**

file->openWrite(*path::string, okCreate::boolean=?***)**

file->openWriteOnly(*path::string, okCreate::boolean=?***)**

file->openAppend(*path::string, okCreate::boolean=?***)**

file->openTruncate(*path::string, okCreate::boolean=?***)**
>   These methods will open the file in the same manner as the preceding methods, however these methods allow the file
>   path to be given at the time the file is opened. An optional second boolean parameter can be given indicating whether
>   the file should be created if it does not exist. If "false" is given for this parameter then the file will not be created and a
>   failure will be generated using the operating system's error code and message.

24.2.2 Closing Files

Once a file is opened, it must later be closed. Once a file is closed it can no longer be read from or written to until it is reopened.

```
file->doWithClose()
```
Requires a capture block when called. The capture block will be invoked and then the file will be closed. This is the safest method to use when working with files as it will ensure the file is closed even if a failure occurs within the capture block.

Example of writing to a file within a capture block:

```
local(f) = file('n.txt')
#f->doWithClose => {
    #f->openWrite
    // ... work with file ...
}
```

```
file->close()
```
This method simply closes the file.

24.2.3 Reading File Data

File data can be read as either bytes or string objects. By default, string objects, which are always Unicode, are created with the assumption that the file contains UTF-8 encoded data. This assumption can be changed by settings the file objects's character encoding value. When reading the data as a bytes object, the unaltered file data is returned.

Data can be read line by line or as individual bytes or in chunks of bytes. Each read will return the bytes immediately following the previously read bytes unless the file's read/write position is moved. Attempts to read past the end of the file will return a zero-sized bytes object.

```
file->readBytes(count::integer=?) → bytes
```
Reads and returns all the remaining data from the file, or reads up to the requested number of bytes. There may be fewer bytes available than requested.

```
file->readString(count::integer=?) → string
```
Reads and returns all the remaining data from the file, or reads up to the requested number of bytes and attempts to convert it into a string object. This method is generally not safe when dealing with multi-byte characters as the read end point may come in the middle of a character sequence, producing invalid Unicode data.

```
file->marker() → integer
```

```
file->marker=(m::integer)
```
These methods respectively get and set the file object's current read/write marker. This value controls where the next read or write will take place. The marker value is zero-based. Settings the marker to zero moves the marker to the beginning of the file.

```
file->encoding() → string
```

```
file->encoding=(e::string)
```
These methods respectively get and set the file object's character encoding value. This value controls how the **file->readString** method converts the data read from the file into a string object. This value defaults to UTF-8.

```
file->forEach()
```

```
file->forEachLine()
```
These methods provide iteration over the file's bytes either one at a time or line by line.

Example of performing an operation for each line of a file:

```
#f->forEachLine => {
    local(theLine) = #1
    // ...
}
```

24.2.4 Writing File Data

Data can be written to files using either bytes or string objects as the source. When writing Unicode string data to a file, the file's encoding value is used. Writing past the end of the file will increase the file's size. Manipulating the file's marker will adjust where the next write takes place.

file->**writeBytes**(*b::bytes*) → integer

file->**writeString**(*s::string*) → integer
>   These methods write bytes or string data to the file and return the number of bytes that were written.

file->**moveTo**(*path::string*, *overwrite::boolean=false*)

file->**copyTo**(*path::string*, *overwrite::boolean=false*)
>   These two methods attempt to move or copy the file to a new location or fail trying. The overwrite parameter indicates that if the destination file already exists the method should fail. Setting overwrite to "true" will have it replace the existing file with the file referenced by the file object.

file->**delete()**
>   This methods will delete the file from the system. The file is closed first.

24.2.5 File Manipulation Methods

file->**exists()** → boolean
>   Returns "true" if the file exists on the system.

file->**path()** → string
>   Returns the path to the file.

file->**parentDir()** → dir
>   Returns a **dir** object set to the file's parent directory.

file->**size()** → integer

file->**size=**(*s::integer*)
>   These methods get and set the file's size. Setting the size in this manner will change the file's size on disk.

file->**modificationTime()** → integer

file->**modificationDate()** → date
>   These methods return the raw file modification time as an integer and the modification time as a date object, respectively.

file->**lastAccessTime()** → integer

file->**lastAccessDate()** → date
>   These methods return the raw file last access time as an integer and the last access time as a date object, respectively.

file->**linkTo**(*path::string*, *hard::boolean=false*)
>   Attempts to create a hard or soft link of the file at the specified location. This method may not be available or may not operate consistently across all supported operating systems.

file->**chown**(*user::string*, *group::string=?*)

file->**chown**(*uid::integer*, *gid::integer*)

file->**chmod**(*to::integer*)

file->**perms()** → integer
>   These methods are used to set and get the permissions of the file. These operations are currently supported on UNIX-based systems only.

24.2.6 Standard File Objects

file_stdin() → file

file_stdout() → file

file_stderr() → file
> Lasso makes the standard in, out, and error files available using these methods. In general, these file objects should not be closed. The file objects returned from these methods will not close the underlying system file when they are garbage-collected.

24.3 Dir Type

type **dir**

dir(*path::string*, *-resolveLinks=false*)
> Dir objects are instantiated with a path and an optional **-resolveLinks** keyword parameter. This parameter defaults to "false". If set to "true", then the dir object will resolve symbolic links when iterating over its contents, when returning its own **file->perms** and when determining if it is indeed a directory through the **dir->isDir** method.

24.3.1 Creating Directories

dir->create(*perms::integer=integer_bitOr(io_file_s_irwxg, io_file_s_irwxu, io_file_s_irwxo)*)
> Attempts to create the directory at the path indicated when the dir object was created. The perms parameter indicates the permissions that the directory should be given. This defaults to the equivalent of "rwxrwxrwx".

> This method will attempt to create any non-existent intermediate directories along the path with the same permissions. It does not alter the permissions of any existing directories.

24.3.2 Iterating Directory Contents

The contents of a directory can be explored in a variety of ways. The contents can be returned as a series of string paths or as a series of file and dir objects. Sub-directory contents can be returned recursively.

The paths of subdirectories produced by these methods will have a trailing forward slash. A dir object never returns a path or object representing the "." or "" directory entries.

Each of the values returned by these methods can be used in query expressions or in **iterate**. A dir object itself can be used in a query expression or iterate. In this case, the behavior will be the same as with the **dir->eachPath** method, described below.

dir->**eachPath()**

dir->**eachFilePath()**

dir->**eachDirPath()**
> These methods are used to operate on the relative paths of the contents of the directory. The **eachPath** method will return both files and subdirectories, while **eachFilePath** and **eachDirPath** return only the file or subdirectory paths, respectively.

dir->**eachPathRecursive()**

dir->**eachFilePathRecursive()**

dir->**eachDirPathRecursive()**
> These methods are used to operate on the relative paths or the contents of the directory. When a subdirectory is encountered, its contents are also included, and so on as deep as the directory tree goes.

```
dir->each()

dir->eachFile()

dir->eachDir()
```
> These methods return the directory contents as file or dir objects. The **each** method returns both the files and directories within the directory. The **eachFile** and **eachDir** methods return only the files or directories, respectively.

List Directory Contents

Use a **dir** object in a query expression to list the contents of the current working directory:

```
with path in dir('.')
sum #path + '\n'

// =>
// A Folder/
// My_File.txt
// Sub_Directory/
```

Use a **dir** object to list a directory's contents as **file** objects:

```
with f in dir('foo/')->eachFile
// f is a file object
sum #f->size->asString(-padding=10) + ' ' + #f->name + '\n'

// =>
//     12779 An Example File.pdf
//         0 empty_file
//      1063 Rhino Habitats.txt
//    109572 Rhino Running.jpg
//      3270 Summary.txt
```

24.3.3 Directory Manipulation Methods

dir->**moveTo**(*path::string*)
> Attempts to rename, or "move", the directory. A failure is generated if the operation fails.

dir->**delete()**
> Attempts to delete the directory. A directory must be empty before it can be successfully deleted. A failure is generated if the operation fails.

dir->**exists()** → boolean
> Returns "true" if the directory exists on disk.

dir->**path()** → string
> Returns the directory's path as a string.

dir->**parentDir()** → dir
> Returns the directory's parent directory as a **dir** object.

Images and Media

Lasso includes features that allow you to manipulate and serve images and media files on the fly. The **image_...** methods allow you to do the following with image files in the supported image formats:

- Scaling and cropping images, facilitating the creation of thumbnail images on the fly.
- Rotating images and changing image orientation.
- Applying image effects such as modulation, blurring, and sharpening effects.
- Adjusting image color depth and opacity.
- Combining images, adding logos and watermarks.
- Image format conversion.
- Retrieval of image attributes, such as dimensions, bit depth, and format.
- Executing extended ImageMagick commands.

Note: The **image** type and features in Lasso are implemented using ImageMagick 6.6.6-10 (July 7, 2011 build), which is installed as part of Lasso Server on OS X. Windows and Linux require ImageMagick to be installed separately, which is covered with their respective installation instructions. For more information on ImageMagick, visit http://www.imagemagick.org/.

25.1 Image File Operations

Image files can be manipulated via Lasso by setting a variable to an instance of the **image** type, and then using various member methods to manipulate the variable. Instantiating an image object usually involves loading data from an image file on the server into memory as an image object. Once the image file is manipulated, it can either be served directly to the client browser, or it can be saved to disk on the server.

25.1.1 Dynamically Manipulate an Image File

The following shows an example of initializing, manipulating, saving, and serving an image file named "image.jpg" using the **image** type:

```
<?lasso
    local(myImage) = image('/images/image.tif')
    #myImage->scale(-height=35, -width=35, -thumbnail)
    #myImage->save('/images/image.jpg')
?>
<img src="/images/image.jpg" border="0">
```

In the example above, an image file named "image.tif" is referenced as a Lasso image object using the **image** type, then resized to 35 x 35 pixels using the **image->scale** method. (The optional **-thumbnail** parameter optimizes the image for the web.)

Then the image is converted to JPEG format and saved to disk using the **image->save** method. Finally, the new image is displayed on the current page using an HTML **<img>** tag.

This chapter explains in detail how these and other methods are used to manipulate image and media files. This chapter also shows how to output an image file to a client browser within the context of a Lasso page.

25.1.2 Supported Image Formats

Because the **image** member methods are based on ImageMagick, Lasso supports reading and manipulating over 88 major file formats (not including subformats). A comprehensive list of supported image formats[42] can be found at the ImageMagick website.

A list of commonly used image formats that are certified to work with Lasso out-of-the-box without requiring installation of additional components are shown in the table **Tested and Certified Image Formats**.

Table 25.1: Tested and Certified Image Formats

Format	Description
BMP	Microsoft Windows bitmap file.
CMYK	Raw cyan, magenta, yellow, and black samples.
GIF	CompuServe Graphics Interchange Format. LZW-compressed 8-bit RGB with up to 256 palette entries.
JPEG	Joint Photographic Experts Group format. Also known as *JPG*.
PNG	Portable Network Graphics format.
PSD	Adobe Photoshop bitmap file.
RGB	Raw red, green, and blue samples.
TIFF	Tagged Image File Format. Also known as *TIF*.

Note: Many of the formats listed on the ImageMagick site such as EPS and PDF may be used with the **image_...** methods, but require additional components such as Ghostscript to be installed before they will work. These formats may be used, but because they rely heavily on third-party components, they are not officially supported.

25.1.3 File Permissions

In order to successfully create, manipulate, and save image files using the **image_...** methods, the user running the Lasso process must be allowed by the operating system to write and execute files inside the folder. To check folder permissions in Windows, right-click on the folder and select *Properties → Security*. For OS X or Linux, use **ls -al** from the command line to check permissions and use the **chmod** and **chown** commands to adjust the permissions. (Refer to the *ls*, *chmod*, and *chown* man pages for more information on their use).

25.2 Referencing Images as Lasso Objects

For Lasso to be able to edit an image, an image file or image data must first be modelled as a Lasso image object using the **image** type. Once a variable has been set to an image object, various member methods can be used to manipulate the image. Once the image data is manipulated, it can either be served directly to the client browser, or it can be saved to disk on the server.

type **image**

---

[42] http://www.imagemagick.org/script/formats.php#supported

image()

image(*filePath::string, -info=?*)

image(*bytes::bytes, -info=?*)
>   Creates an image object. Requires either the path to an image file or a bytes object with an image's binary data to initialize the object. Once an image object is initialized, it may be edited and saved using the **image** member methods which are described throughout this chapter.
>
>   The optional **-info** parameter retrieves all the attributes of an image without reading the pixel data. This allows for better performance and less memory usage when initializing an image object.
>
>   Example of creating an image object from a file:

```
local(myImage1) = image('/images/image.jpg')
```

>   Example of creating an image object with just the attributes:

```
local(myImage2) = image('/images/largeimage.jpg', -info)
```

>   Example of creating an image object with bytes data:

```
local(binary)   = file('image.jpg')->readBytes
local(myImage3) = image(#binary)
```

25.3 Image Information Methods

Information about an image can be returned using special **image** member methods. These methods return specific values representing the attributes of an image such as size, resolution, format, and file comments. All the image information methods in Lasso are defined below.

image->width() → integer
>   Returns the image width in pixels.

image->height() → integer
>   Returns the image height in pixels.

image->resolutionH() → integer
>   Returns the horizontal resolution of the image in dpi.

image->resolutionV() → integer
>   Returns the vertical resolution of the image in dpi.

image->depth() → integer
>   Returns the color depth of the image in bits. Can be either 8 or 16.

image->format()
>   Returns the image format (GIF, JPEG, etc).

image->pixel(*x::integer, y::integer, -hex=?*)
>   Returns the color of the pixel located at the specified pixel coordinates (X, Y). The returned value is an array of RGB color integers (0–255) by default. An optional **-hex** parameter returns a hex color string ("#FFCCDD") instead of an RGB array.

image->comments()
>   Returns any comments included in the image file header.

image->describe(*-short=?*)
>   Lists various image attributes, mostly for debugging purposes. An optional **-short** parameter displays abbreviated information.

```
image->file()
```
> Returns the image file path and name, or "null" for in-memory images.

25.3.1 Return Height and Width of an Image

Use the **image->height** and **image->width** methods on an image object. This returns an integer value representing the height and width of the image in pixels:

```
local(myImage) = image('/images/image.jpg')
#myImage->width + ' x ' + #myImage->height

// => 400 x 300
```

25.3.2 Return Resolution of an Image

Use the **image->resolutionH** and **image->resolutionV** methods on an image object. This returns a decimal value representing the horizontal and vertical **DPI** (Dots Per Inch) of the image:

```
local(myImage) = image('/images/image.jpg')
#myImage->resolutionV + ' x ' + #myImage->resolutionH

// => 600 x 600
```

25.3.3 Return Color Depth of an Image

Use the **image->depth** method on an image object. This returns an integer value representing the color depth of an image in bits:

```
local(myImage) = image('/images/image.jpg')
#myImage->depth

// => 16
```

25.3.4 Return Format of an Image

Use the **image->format** method on an image object. This returns a string value representing the file format of the image:

```
image('/images/image.gif')->format

// => GIF
```

25.3.5 Return Pixel Information About an Image

Use the **image->pixel** method on an image object. This returns a string value representing the color of the pixel at the specified coordinates:

```
local(myImage) = image('/images/image.jpg')
#myImage->pixel(25, 125, -hex)

// => FF00FF
```

25.4 Converting and Saving Images

This section describes how image files can be converted from one format to another and saved to file. This is all accomplished using the **image->save** method, which is described below.

image->**convert**(*ext::string, -quality::integer=?*)
>   Converts an image object to a new format. Requires a file extension as a string parameter which represents the new format the image is being converted to (e.g. **'jpg'**, **'gif'**). A **-quality** parameter specifies the image compression ratio (integer value of 1–100) used when saving to JPEG or GIF format.

image->**save**(*path::string, -quality::integer=?*)
>   Saves the image to a file in a format defined by the file extension. Automatically converts images when the extension of the image to save as differs from that of the original image. A **-quality** parameter specifies the image compression ratio (integer value of 1–100) used when saving to JPEG or GIF format.

image->**addComment**(*comment*)
>   Adds a file header comment to the image before it is saved. Passing a "null" parameter removes any existing comments.

25.4.1 Convert an Image File from One Format to Another

Use the **image->convert** and **image->save** methods on an image object, specifying the new format as part of the **image->convert** method:

```
local(myImage) = image('/images/image.gif')
#myImage->convert('JPG', -quality=100)
#myImage->save('/images/image.jpg', -quality=100)
```

25.4.2 Automatically Convert the Format of an Image File

Use the **image->save** method on an image object, changing the image file extension to the desired image format. A **-quality** parameter value of "100" specifies that the resulting JPEG file will be saved at the highest quality resolution:

```
local(myImage) = image('/images/image.gif')
#myImage->save('/images/image.jpg', -quality=100)
```

25.4.3 Save an Image Object to a File

Use the **image->save** method on an image object, specifying the desired image name, path, and format:

```
local(myImage) = image('/folder/image.jpg')
#myImage->save('/images/image_copy.jpg')
```

25.4.4 Add a Comment to an Image File Header

Use the **image->addComment** method to add a comment to an image object before it is saved to file. This comment is not displayed, but stored with the image file information:

```
local(myImage) = image('/images/image.gif')
#myImage->addComment('This is a comment')
#myImage->save(/images/image.gif')
```

25.4.5 Remove All Comments from an Image File Header

Use the **image->addComment** method with a "null" parameter to remove all comments from an image object before it is saved to file. The following code adds a comment and then removes all comments. The result is an image with no comments:

```
local(myImage) = image('/images/image.gif')
#myImage->addComment('This is a comment')
#myImage->addComment(null)
#myImage->save('/images/image.gif')
```

25.5 Images Manipulation Methods

Images can be transformed and manipulated using special **image** member methods. These methods change the appearance of the image as it served to the client browser. This includes methods for changing image size and orientation, applying image effects, adding text to images, and merging images, which are described in the following subsections.

25.5.1 Changing Image Size and Orientation

Lasso provides methods that allow you to scale, rotate, crop, and invert images. These methods are defined below.

image->**scale**(…)
>   Scales an image to a specified size. Requires either a **-width** or **-height** parameter, which specify the new size of the image using either integer pixel values (e.g. "50") or string percentage values (e.g. "50%"). An optional **-sample** parameter indicates pixel sampling should be used so no additional colors will be added to the image. An optional **-thumbnail** parameter optimizes the image for display on the web. If only one of the **-width** or **-height** is specified then the other value is calculated proportionally.

image->**rotate**(*deg::integer, -bgColor=::string=?*)
>   Rotates an image counterclockwise by the specified amount in degrees (integer value of 0–360). An optional **-bgColor** parameter specifies the hex color to fill the blank areas of the resulting image.

image->**crop**(…)
>   Crops the original image by cutting off extra pixels beyond the boundaries specified by the parameters. Requires **-height** and **-width** parameters which specify the pixel size of the resulting image, and **-left** and **-right** parameters specify the offset of the resulting image within the initial image.

image->**flipV()**
>   Creates a vertical mirror image by reflecting the pixels around the central X-axis.

image->**flipH()**
>   Creates a horizontal mirror image by reflecting the pixels around the central Y-axis.

Enlarge an Image

Use the **image->scale** method on an image object. The following example enlarges "image.jpg" to 225 X 225 pixels. The optional **-sample** parameter specifies that pixel sampling should be used:

```
local(myImage) = image('/images/image.jpg')
#myImage->scale(-height=225, -width=225, -sample)
#myImage->save('/images/image.jpg')
```

Shrink an Image

Use the **image->scale** method on an image object. The following example shrinks "image.jpg" to 25 x 25 pixels. The optional **-thumbnail** parameter optimizes the image for the web:

```
local(myImage) = image('/images/image.jpg')
#myImage->scale(-height=25, -width=25, -thumbnail)
#myImage->save('/images/image.jpg')
```

Rotate an Image

Use the **image->rotate** method on an image object. The following example rotates the image 60 degrees counterclockwise on top of a white background:

```
local(myImage) = image('/images/image.jpg')
#myImage->rotate(60, -bgColor='FFFFFF')
#myImage->save('/images/image.jpg')
```

Crop an Image

Use the **image->crop** method on an image object. The example below crops 10 pixels off of each side of a 70 x 70 image:

```
local(myImage) = image('/images/image.jpg')
#myImage->crop(-left=10, -right=10, -width=50, -height=50)
#myImage->save('/images/image.jpg')
```

Mirror an Image

Use the **image->flipV** method on an image object. The following example mirrors the image vertically:

```
local(myImage) = image('/images/image.jpg')
#myImage->flipV
#myImage->save('/images/image.jpg')
```

25.5.2 Applying Image Effects

Lasso provides methods that allow you to add image effects by applying special image filters. This includes color modulation, image noise enhancement, sharpness controls, blur controls, contrast controls, and composite image merging. These methods are described below.

image->modulate(*bright::integer*, *saturation::integer*, *hue::integer*)
> Controls the brightness, saturation, and hue of an image. Brightness, saturation, and hue are controlled by three comma-delimited integer parameters, where 100 equals the original value.

image->contrast(*increase::boolean=true*)
> Enhances the intensity differences between the lighter and darker elements of the image. Specify "false" to reduce the image contrast, otherwise the contrast is increased.

image->blur(*-angle::decimal*)

image->blur(*-gaussian*, *-radius::decimal*, *-sigma::decimal*)
> Applies either a motion or Gaussian blur to an image. To apply a motion blur, an **-angle** parameter with a decimal degree value must be specified to indicate the direction of the motion. To apply a Gaussian blur, a **-gaussian** keyword

parameter must be specified in addition to `-radius` and `-sigma` parameters that require decimal values. The `-radius` parameter is the radius of the Gaussian in pixels, and `-sigma` is the standard deviation of the Gaussian in pixels. For reasonable results, the radius should be larger than the sigma.

image->**sharpen**(*-radius::integer, -sigma::integer, -amount::decimal=?, -threshold::decimal=?*)

Sharpens an image. Requires `-radius` and `-sigma` parameters that are integer values. The `-radius` parameter is the radius of the Gaussian sharp effect in pixels, and `-sigma` is the standard deviation of the Gaussian sharp effect in pixels. For reasonable results, the radius should be larger than the sigma. Optional `-amount` and `-threshold` parameters may be used to add an unsharp masking effect. `-amount` specifies the decimal percentage of the difference between the original and the blur image that is added back into the original, and `-threshold` specifies the threshold in decimal pixels needed to apply the difference amount.

image->**enhance**()

Applies a filter that improves the quality of a noisy, lower-quality image.

Adjust Brightness of an Image

Use the **image->modulate** method on an image object and adjust the first integer parameter, representing brightness. The following example increases the brightness of an image by a factor of two:

```
local(myImage) = image('/images/image.jpg')
#myImage->modulate(200, 100, 100)
#myImage->save('/images/image.jpg')
```

Adjust Color Saturation of an Image

Use the **image->modulate** method on an image object and adjust the second integer parameter, representing color saturation. The following example decreases the color saturation of an image by 25%:

```
local(myImage) = image('/images/image.jpg')
#myImage->modulate(100, 75, 100)
#myImage->save('/images/image.jpg')
```

Adjust Hue of an Image

Use the **image->modulate** method on an image object and adjust the third integer parameter, representing hue. The following example tints the image green by increasing the hue value. Decreasing the hue value tints the image red:

```
local(myImage) = image('/images/image.jpg')
#myImage->modulate(100, 100, 175)
#myImage->save('/images/image.jpg')
```

Adjust Contrast of an Image

Use the **image->contrast** method on an image object. The first example increases the contrast. The second example uses a "false" parameter value, which reduces the contrast instead:

```
local(myImage) = image('/images/image.jpg')
#myImage->contrast
#myImage->save('/images/image.jpg')

local(myImage) = image('/images/image.jpg')
```

```
#myImage->contrast(false)
#myImage->save('/images/image.jpg')
```

Apply a Motion Blur to an Image

Use the **image->blur** method on an image object. The following example applies a motion blur at 20 degrees:

```
local(myImage) = image('/images/image.jpg')
#myImage->blur(-angle=20)
#myImage->save('/images/image.jpg')
```

Apply a Gaussian Blur to an Image

Use the **image->blur** method with the **-gaussian** parameter on an image object. The following example applies a Gaussian blur with a radius of 15 pixels and a standard deviation of 10 pixels:

```
local(myImage) = image('/images/image.jpg')
#myImage->blur(-radius=15, -sigma=10, -gaussian)
#myImage->save('/images/image.jpg')
```

Sharpen an Image

Use the **image->sharpen** method on an image object. The following example applies a Gaussian sharp effect with a radius of 20 pixels and a standard deviation of 10 pixels:

```
local(myImage) = image('/images/image.jpg')
#myImage->sharpen(-radius=20, -sigma=10)
#myImage->save('/images/image.jpg')
```

Sharpen an Image with an Unsharp Mask Effect

Use the **image->sharpen** method with the **-amount** and **-threshold** parameters on an image object. The following example applies an unsharp mask effect with a radius of 20 pixels and a standard deviation of 10 pixels:

```
local(myImage) = image('/images/image.jpg')
#myImage->sharpen(-radius=20, -sigma=10, -amount=50, -threshold=20)
#myImage->save('/images/image.jpg')
```

Enhance a Low-Quality Image

Use the **image->enhance** method on an image object:

```
local(myImage) = image('/images/image.jpg')
#myImage->enhance
#myImage->save('/images/image.jpg')
```

25.5.3 Adding Text to Images

Lasso allows text to be overlaid on top of images using the **image->annotate** method as described below.

image->annotate(_annotation::string,   -left::integer,   -top::integer,   -font::string=?,   -size::integer=?,   -color::string=?, -aliased::boolean=?_**)**

> Overlays text onto an image. Requires a string value as a parameter, which is the text to be overlaid. Required **-left** and **-top** parameters specify the place of the text in pixel integers relative to the upper left corner of the image. An optional **-font** parameter specifies the name (with extension) and full path to a system font to be used for the text, and an optional **-size** parameter specifies the text size in integer pixels. An optional **-color** parameter specifies the text color as a hex string ("#FFCCDD"). An optional **-aliased** keyword parameter turns on text anti-aliasing.

Note: The full hard drive path to the font must be used (e.g. **-font='//Library/Fonts/Arial.ttf'**) when specifying a font. True Type ("*.ttf"), and Type One ("*.pfa", "*.pfb") font types are officially supported.

Add Text to an Image

Use the **image->annotate** method on an image object. The example below adds the text "(c) 2013 LassoSoft" to the specified image:

```
local(myImage) = image('/images/image.jpg')
#myImage->annotate(
   '(c) 2003 LassoSoft',
   -left=5,
   -top=300,
   -font='/Library/Fonts/Arial.ttf',
   -size=8,
   -color='#000000',
   -aliased
)
#myImage->save('/images/image.jpg')
```

25.5.4 Merging Images

Lasso allows images to be merged using the **image->composite** method. This method supports over 20 different composite methods, which are described in the table below.

image->composite(_second::image, -op::string=?, -left::integer=?, -top::integer=?_**)**

> Composites a second image onto the current image. Requires two Lasso image objects to be composited. An **-op** parameter specifies the composite method that affects how the second image is applied to the first image (a list of operators is shown below). Optional **-left** and **-top** parameters specify the horizontal and vertical offset of the second image over the first in integer pixels (defaults to the upper left corner). An optional **-opacity** parameter attenuates the opacity of the composited second image, where a value of "0" is fully opaque and "1.0" is fully transparent.

> The table below shows the various composite operators that can be specified by the **-op** parameter. The descriptions for each method are adapted from the ImageMagick web site.

Table 25.2: Composite Image Tag Operators

Composite Operator	Description
Over	The result is the union of the two image shapes with the composite image obscuring the image in the region of overlap.
In	The result is the first image cut by the shape of the second image. None of the second image data is included in the result.
Out	The result is the second image cut by the shape of the first image. None of the first image data is included in the result.
Plus	The result is the sum of the raw image data with output image color channels cropped to 255.
Minus	The result is the subtraction of the raw image data with color channel underflow cropped to zero.
Add	The result is the sum of the raw image data with color channel overflow channel wrapping around 255 to 0.
Subtract	The result is the subtraction of the raw image data with color channel underflow wrapping around 0 to 255.
Difference	Returns the difference between two images. This is useful for comparing two very similar images.
Bumpmap	The resulting image is shaded by the second image.
CopyRed	The resulting image is the red layer in the image replaced with the red layer in the second image.
CopyGreen	The resulting image is the green layer in the image replaced with the green layer in the second image.
CopyBlue	The resulting image is the blue layer in the image replaced with the blue layer in the second image.
CopyOpacity	The resulting image is the opaque layer in the image replaced with the opaque layer in the second image.
Displace	Displaces part of the first image where the second image is overlaid.
Threshold	Only colors in the second image that are darker than the colors in the first image are overlaid.
Darken	Only dark colors in the second image are overlaid.
Lighten	Only light colors in the second image are overlaid.
Colorize	Only base spectrum colors in the second image are overlaid.
Hue	Only the hue of the second image is overlaid.
Saturate	Only the saturation of the second image is overlaid.
Luminize	Only the luminosity of the second image is overlaid.
Modulate	Has the effect of the **Hue**, **Saturate**, and **Luminize** functions applied at the same time.

Overlay an Image On Top of Another Image

Use the **image->composite** method to add an image object to a second image object. The following example adds "image2.jpg" offset by five pixels in the upper left corner of "image1.jpg":

```
local(myImage1) = image('/images/image1.jpg')
local(myImage2) = image('/images/image2.jpg')
#myImage1->composite(#myImage2, -left=5, -top=5)
#myImage1->save('/images/image1.jpg')
```

Add a Watermark to an Image

Use the **image->composite** method with the **-opacity** parameter to add an image object to a second image object. The following example adds a mostly transparent version of "image2.jpg" to "image1.jpg":

```
local(myImage1) = image('/images/image1.jpg')
local(myImage2) = image('/images/image2.jpg')
#myImage1->composite(#myImage2, -opacity=0.75)
#myImage1->save('/images/image1.jpg')
```

Shade Image with a Second Image

Use the **image->composite** method with the "Bumpmap" operator to shade an image object over a second image object:

```
local(myImage1) = image('/images/image1.jpg')
local(myImage2) = image('/images/image2.jpg')
#myImage1->composite(#myImage2, -op='Bumpmap')
#myImage1->save('/images/image1.jpg')
```

Return the Pixel Difference Between Two Images

Use the **image->composite** method with the "Difference" operator to return the pixel difference between two defined image variables:

```
local(myImage1) = image('/images/image1.jpg')
local(myImage2) = image('/images/image2.jpg')
#myImage1->composite(#myImage2, -op='Difference')
#myImage1->save('/images/image1.jpg')
```

25.6 Extended ImageMagick Commands

For users who have experience using the ImageMagick command-line utility, Lasso provides the **image->execute** method to allow advanced users to take advantage of additional ImageMagick commands and functionality.

image->**execute()**
> Execute ImageMagick commands. Provides direct access to the ImageMagick command-line interface. Supports the "composite", "mogrify", and "montage" commands. See the ImageMagick Command-Line Tools documentation[43] for detailed descriptions of these commands and their corresponding parameters.

25.6.1 Execute an ImageMagick Command Using Lasso

Use the **image->execute** method on an image object, with the desired command as the parameter. The following example shows the "mogrify" command adding a distinctive blue border to an image:

```
local(myImage) = image('/images/image.gif')
#myImage->execute('mogrify -bordercolor blue -border=3x3')
#myImage->save('/images/image.gif')
```

---

[43] http://www.imagemagick.org/script/command-line-tools.php

25.7 Serving Image and Media Files

This section discusses how to serve image and media files, including referencing files within HTML pages and serving files separately via HTTP.

25.7.1 Referencing Within HTML Files

The easiest way to serve images and media files is by simply referencing files stored within the web server root using standard HTML tags such as `<img>` or `<embed>`. The path to the image file can be calculated in the Lasso page or stored within a database field. Since the specified file is ultimately served by the web server application that is optimized for serving images and media files, this is the most efficient way to serve images and media files.

Generate the Path to an Image or Media File

The following example shows a variable "company_name" that contains "LassoSoft". This variable is used to construct a path to an image file stored within the "images" folder named with the company name and "_logo.gif" to form the full file path "/images/LassoSoft_logo.gif":

```
[local(company_name) = 'LassoSoft']
<img src="/images/[#company_name]_logo.gif" />

// => <img src="/images/LassoSoft_logo.gif" />
```

Using the same image path described above, the path to the image file is stored within the variable "image_path" and then referenced in the HTML `<img>` tag:

```
[local(company_name) = 'LassoSoft']
[local(image_path)   = '/images/' + #company_name + '_logo.gif']
<img src="[#image_path]" />

// => <img src="/images/LassoSoft_logo.gif" />
```

The following example shows a variable "band_name" that contains "ArtOfNoise". This variable is used to construct a path to sound files stored within the "sounds" folder named with the band name and ".mp3" to form the full file path "/sounds/ArtOfNoise.mp3". The path to the sound file is stored within the variable "sound_path" and then referenced in the HTML `<a>` tag:

```
[local(band_name)  = 'ArtOfNoise']
[local(sound_path) = '/images/' + #band_name + '.mp3']
<a href="[#sound_path]">Download MP3</a>

// => <a href="/sounds/ArtOfNoise.mp3">Art of Noise Song</a>
```

25.7.2 Serving Files via HTTP

Lasso can also be used to serve image and media files rather than merely referencing them by path. Files are served through Lasso using the **web_response->sendFile** method or a combination of the **web_response->replaceHeader** method and **web_response->includeBytes** method. Lasso also includes an **image->data** method that automatically converts an image object to a bytes object, allowing an edited image object to be output using **web_response->sendFile** without it first being written to disk.

In order to serve an image or media file through Lasso the MIME type of the file must first be determined. Often, this can be discovered by looking at the configuration of the web server or web browser. The MIME type for a GIF is *image/gif* and the MIME type for a JPEG is *image/jpeg*.

Note: It is not recommended that you configure your web server application to process all "*.gif" and "*.jpg" files through Lasso. Lasso will attempt to interpret the binary data of the image file as Lasso code. Instead, use one of the procedures below to serve an image file from a path with a ".lasso" extension.

---

image->data()
> Converts an image object to a binary bytes object. This is useful for serving images to a browser without writing the image to file.

Serve an Image File

Use the **web_response->sendFile** method to set the MIME type of the image to be served, and use the **image->data** method to get the binary data from an image object. The **web_response->sendFile** method aborts the current response, so it will be the last line of code to be processed. The following example shows a GIF named "picture.gif" being served from an "images" folder:

```
local(image) = image('/images/picture.gif')
web_response->sendFile(#image->data, -type='image/gif')
```

Alternatively, you can use the **web_response->replaceHeader** method to set the MIME type of the image to be served and use the **web_response->includeBytes** method to include data from the image file. If using this method, verify that no stray data is inadvertently added into the outgoing data buffer as it will corrupt the output. This includes whitespace characters. The following example shows a GIF named "picture.gif" being served from an "images" folder. It is the only contents of this file being called by the client browser and calls abort to avoid any data corruption:

```
<?lasso
    web_response->replaceHeader('Content-Type'='image/gif')
    web_response->includeBytes('/images/picture.gif')
    abort
?>
```

If either of the code examples above is stored in a file named "image.lasso" at the root of the web serving folder then the image could be accessed with the following **<img>** tag:

```
<img src="/image.lasso" />
```

Serve a Media File

Use the **web_response->sendFile** method to set the MIME type of the file to be served and pass it a **file** object to include data from the media file. The following example shows a sound file named "ArtOfNoise.mp3" being served from a "sounds" folder:

```
web_response->sendFile(
    file('/sounds/ArtOfNoise.mp3'),
    'ArtOfNoise.mp3',
    -type='audio/mp3'
)
```

If the code above is stored in a file named "ArtOfNoise.lasso" at the root of the web serving folder then the sound file could be accessed with the following **<a>** tag:

```
<a href="/ArtOfNoise.lasso">Art of Noise Song</a>
```

This same technique can be used to serve media files of any type by designating the appropriate MIME type in the **-type** option passed to the **web_response->sendFile** method.

Limit Access to a File

Since the Lasso page can process any Lasso code before serving the image it is easy to create a file that generates an error if an unauthorized person tries to access a file. The following code checks the **client_username** for the name "John". If the current user is not named "John" then a file "error.gif" is served instead of the desired "picture.gif" file. To completely limit access to the files, they are being served from outside the web root of the web server so that the files can't be loaded directly by a URL. In this example, the files are being served from the "secret" folder which is at the root level of the file system:

```
if('John' == client_username) {
    web_response->sendFile(
        file('//secret/picture.gif'),
        'picture.gif',
        -type='image/gif'
    )
else
    web_response->sendFile(
        file('/images/error.gif'),
        'picture.gif',
        -type='image/gif'
    )
}
```

This same technique can be used to restrict access to any image or media file.

Portable Document Format

Lasso provides support for creating **PDF** (Portable Document Format) files. The PDF file format[44] is a widely accepted standard for electronic documentation, and facilitates superb printer-quality documents from simple graphs to complex forms such as tax forms, escrow documents, loan applications, stock reports, and user manuals.

26.1 Lasso and PDF Files

PDF files are created in Lasso by using the **pdf_doc** type, and calling various member methods and other **pdf_...** methods to add data to the object. The PDF is then written to file when the Lasso page containing all code is served by the web server. The **pdf_...** methods in Lasso are implemented in LJAPI, and use the iText Java library[45].

26.1.1 Create a Basic PDF File Using Lasso

The following shows an example of creating and outputting a PDF file named "MyFile.pdf" using the **pdf_...** methods:

```
local(my_file) = pdf_doc(
    -file='MyFile.pdf',
    -size='A4',
    -margin=(: 144.0, 144.0, 72.0, 72.0)
)
local(font) = pdf_font(-face='Helvetica', -size=36)
local(text) = pdf_text('I am a PDF document', -font=#font)
#my_file->add(#text)
#my_file->close
```

In the example above, a variable named "my_file" is set to a **pdf_doc** type with a file name of "MyFile.pdf". A single font type is defined for the document using the **pdf_font** type. Then, the text "I am a PDF document" is defined using the **pdf_text** type, and added using the **pdf_doc->add** member method. The PDF is then written to file upon execution of **#my_file->close**. Since no path information was specified along with the file name to the **-file** parameter, the file "MyFile.pdf" is created in the same folder as the page whose code created it.

This chapter explains in detail how these and other methods are used to create and edit PDF files. This chapter also shows how to output a PDF file to a client browser within the context of a Lasso page, which is described in the section *Serving PDF Files* below.

Note: When creating files, the user running the Lasso Server instance or command-line process must be allowed to write to the folder by the operating system. For more information, see the *File System* chapter.

---

[44] https://acrobat.adobe.com/us/en/why-adobe/about-adobe-pdf.html
[45] http://itextpdf.com/

26.2 Reading PDF Files

Lasso provides a type that allows existing PDF files to be read and manipulated. A PDF file is read using **pdf_read**. The file can then be inspected for page count, page size, and the values of any embedded form elements. Pages from the file can be placed within a new PDF file. A range of pages from the PDF file can be saved as a new PDF file and encryption options can be added to the new PDF file.

type **pdf_read**

pdf_read(*-file::string*, *-password::string=?*)
>   Reads an existing PDF file into an object. Requires one parameter **-file** which specifies the name of the PDF file to be read. Optional **-password** parameter specifies the owner's password for the file.

pdf_read->**pageCount()** → integer
>   Returns the number of pages in the file.

pdf_read->**pageSize**(*page::integer=?*) → staticarray
>   Returns the size of a page in the file as a staticarray of width and height. Optional integer parameter specifies which page in the PDF to return the size of and defaults to the first page.

pdf_read->**getHeaders()** → map

pdf_read->**getHeaders**(*name::string*)
>   Returns a map of header elements from the PDF file, or the value for a specified header name.

pdf_read->**fieldNames()** → array
>   Returns an array of form elements embedded in the PDF file.

pdf_read->**fieldType**(*name::string*)
>   Returns the type of a single form element. Requires one parameter which is the name of the field element to be inspected. Types include "Checkbox", "Combobox", "List", "PushButton", "RadioButton", "Text", and "Signature".

pdf_read->**fieldValue**(*name::string*)
>   Returns the value of a single form element. Requires one parameter which is the name of the field element to be inspected.

pdf_read->**setFieldValue**(*field::string*, *value::string*, *-display::string=?*)
>   Sets the value of a single form element. Requires two parameters: the name of a form element and a new value for the element. Optional **-display** parameter specifies a display string for the element.

pdf_read->**importFDF**(*file::string*, *-noFields=?*, *-noComments=?*)

pdf_read->**importFDF**(*data::bytes*, *-noFields=?*, *-noComments=?*)
>   Merges an FDF file into the current PDF file. Any form elements within the file will be populated with the values from the FDF file. Accepts a parameter that specifies the path to the FDF file. Alternately, accepts a bytes object containing the file data. Optional **-noFields** and **-noComments** parameters prevent either fields or comments from being merged.

pdf_read->**exportFDF**(*path::string=?*)
>   Exports an FDF file from the current PDF file. The FDF file will contain values for each of the form elements in the PDF file. If a parameter is specified then the FDF file will be written to that path. Otherwise, a byte object containing the data for the FDF file will be returned.

pdf_read->**javaScript()**
>   Returns the global document JavaScript action for the current PDF file.

pdf_read->**addJavaScript**(*script::string*)
>   Adds a JavaScript action to the current PDF file.

pdf_read->**save**(*file::string*, *-encryptStrong=false*, *-permissions=''*, *-userPassword=''*, *-ownerPassword=''*)
>   Saves a copy of the current PDF file. Requires one parameter which specifies the path to the file where the PDF file should

be saved. Also accepts `-userPassword`, `-ownerPassword`, `-encryptStrong`, and `-permissions` parameters. See the descriptions in the following documentation on the **pdf_doc** type for more information about these parameters.

pdf_read->**setPageRange**(*to::string*)
> Selects a range of pages to save into a new PDF file. Multiple ranges can be specified separated by commas. Ranges take the form "4-10" to specify a start and end page number. Optional "e" or "o" prefixes specify to only select even or odd pages. An optional "!" prefix specifies a range of pages that should not be included. For example, "o4-10" would select the pages 5, 7, and 9 while "1-10,!2-9" would select the pages 1 and 10.

Tip: A pdf_read object can be used in concert with the **pdf_doc->insertPage** method described below to insert pages from an existing PDF file into a new PDF file.

26.2.1 Read In an Existing PDF File

In order to work with an existing PDF file, it must first be read in as a pdf_read object.

```
local(old_pdf) = pdf_read('/documents/somepdf.pdf')
```

26.2.2 Determine Attributes of an Existing PDF File

The number of pages and the dimensions of an existing PDF file can be returned using the **pdf_read->pageCount** and **pdf_read->pageSize** methods.

```
local(old_pdf) = pdf_read('/documents/somepdf.pdf')
'Number of pages: ' + #old_pdf->pageCount + '<br />\n'
'Page size: ' + #old_pdf->pageSize(1)

// =>
// Number of pages: 12<br />
// Page size: staticarray(0.000000, 792.000000, 612.000000, 792.000000)
```

26.3 Creating PDF Files

PDF files are initialized and created using the **pdf_doc** type. This is the basic type used to create PDF documents with Lasso, and is used in concert with all methods described in this chapter.

type **pdf_doc**

pdf_doc(…)
> Initializes a PDF file. Uses optional parameters that set the basic specifications for the file being created. Data is added to the object using member methods, which are described throughout this chapter. The table below outlines the optional parameters that can be passed to a **pdf_doc** creator method.
>
> Parameters
> - **-file** – Defines the file name and path of the PDF file. If omitted, the PDF file is created in RAM (see the section *Serving PDF Files* for more information). If a file name is specified without a folder path, the file is created in the same location as the Lasso page containing the **pdf_...** methods.
> - **-size** – Define the page size of the file. Values for this parameter are standard print sizes, and can be "A0", "A1", "A2", "A3", "A4", "A5", "A6", "A7", "A8", "A9", "A10", "B0", "B1", "B2", "B3", "B4", "B5", "ARCH_A", "ARCH_B", "ARCH_C", "ARCH_D", "ARCH_E", "FLSA", "FLSE", "HALFLETTER", "LEDGER", "LEGAL", "LETTER", "NOTE", and "TABLOID". Defaults to "A4". Optional.

- **-height** – Defines a custom page height for the file. Accepts an integer value which represents the size in points. This can be used in combination with the **-width** parameter instead of the **-size** parameter. Optional.

- **-width** – Defines a custom page width for the file. Requires an integer value which represents the size in points. This can be used in combination with the **-height** parameter instead of the **-size** parameter. Optional.

- **-margins** – Defines the margin size for the page. Requires an array of four decimal values which define the left, right, top, and bottom margins for the page (*left,right,top,bottom*). Optional.

- **-color** – Defines the initial text color of the PDF file. Requires a hex color string. Defaults to "#000000" if not specified. Optional.

- **-useDate** – Adds the current date and time to the document header. Optional.

- **-noCompress** – Produces a PDF without compression to allow PDF code to be viewed. PDF files are compressed by default if not used. Optional.

- **-pageNo** – Sets the starting page number for the PDF file. Requires an integer value, which is the page number of the first page. Optional.

- **-pageHeader** – Sets text that will be displayed at the top of each page in the PDF. Requires a text string as a value. Optional.

- **'Header'='Content'** – Adds defined document headers to the PDF file. **'Header'** is replaced with the name of the document header (e.g. "Title", "Author"), and **'Content'** is replaced with the header value. Optional.

- **-userPassword** – Specifies a password that will be required to open the resulting PDF in a reader application including Adobe Reader, Preview, etc. The file will be encrypted if this parameter is specified. Optional.

- **-ownerPassword** – Specifies a password that will be required to open the resulting PDF in an editor including Acrobat Pro, Lasso's **pdf_read** type, etc. The file will be encrypted if this parameter is specified. Optional.

- **-encryptStrong** – If specified then strong 128-bit encryption is used rather than 40-bit encryption. Note that encryption will only be performed if either **-userPassword** or **-ownerPassword** is specified. Optional.

- **-permissions** – A comma-delimited list of permissions for the PDF file. Values include "Print", "Modify", "Copy", or "Annotate". Four additional options are available only if **-encryptStrong** is used: "FillIn", "Assemble", "ScreenReader", and "DegradedPrint". Optional.

The examples below show creating basic pdf_doc objects, though these objects contain little or no data. Calling **pdf_doc->close** on an object with no data will have no result, and no PDF file will be created. Various types of data can be added to these objects using the methods described in the remainder of this chapter.

26.3.1 Start a Basic PDF File

Use the **pdf_doc** type to create a PDF file which could eventually be saved to a hard drive location on the machine running Lasso. Use the **-file** parameter to define the location and file name, and the **-size** parameter to define a predefined standard size. This basic example creates a pdf_doc object that is ready to have data added to the first page:

```
local(my_file) = pdf_doc(-file='my_file.pdf', -size='A4')
```

26.3.2 Start a PDF File with a Custom Page Size

Use the **pdf_doc** type with the **-height** and **-width** parameters to define a custom page size in points. One inch is equal to 72 points.

```
local(my_file) = pdf_doc(-file='MyFile.pdf', -height='648.0', -width='468.0')
```

26.3.3 Start a PDF File with Custom Margins

Use the **pdf_doc** type with the **-margins** parameter to define custom page margins (in points). The following example adds a margin of 72 points (one inch) to the left and right sides of the page, but adds no margin to the top and bottom. This example also adds the date and time of creation to the document header using the **-useDate** parameter:

```
local(my_file) = pdf_doc(
    -file='MyFile.pdf',
    -size='A4',
    -margins=(: 72.0, 72.0, 0.0, 0.0),
    -useDate
)
```

26.3.4 Start an Uncompressed PDF File

Use the **pdf_doc** type with the **-noCompress** parameter.

```
local(my_file) = pdf_doc(-file='MyFile.pdf', -size='A4', -noCompress)
```

26.3.5 Start a PDF File with Custom Document Headers

Use the **pdf_doc** type with appropriate header.

```
local(my_file) = PDF_Doc(
    -file='MyFile.pdf',
    -size='A4',
    -title='My PDF File',
    -subject='How to create PDF files',
    -author='John Doe'
)
```

26.4 Adding Content to PDFs

There are several different types of data that can be added to a PDF file. Many of these types are first defined as objects using methods such as **pdf_text**, **pdf_list**, **pdf_image**, **pdf_table**, or **pdf_barcode** and then added to a pdf_doc object using the **pdf_doc->add** member method. Each type is described separately in subsequent sections of this chapter.

pdf_doc->add(_object_, … **)**

> Adds a PDF content object to a file. This can be used to add **pdf_text**, **pdf_list**, **pdf_image**, **pdf_table**, or **pdf_barcode** objects. If no position information is specified then the object is added to the flow of the page, otherwise it is drawn at the specified location. Requires one parameter, which is the object to be added. Optional parameters are described below.

> > Parameters

- **-align** – Sets the alignment of the object in the page (`'Left'`, `'Center'`, or `'Right'`). Defaults to "Left". Works only for pdf_image and pdf_barcode objects. Optional.

- **-wrap** – Keyword parameter specifies that text should flow around the embedded object. Works only for pdf_image and pdf_barcode objects. Optional.

- **-left** – Specifies the placement of the object relative to the left side of the document. Requires a decimal value, which is the placement offset in points. Works only for pdf_image and pdf_barcode objects. Optional.

- **-top** – Specifies the placement of the object relative to the top of the document. Requires a decimal value, which is the placement offset in points. Works only for pdf_image and pdf_barcode objects. Optional.

- **-height** – Scales the object to the specified height. Requires a decimal value which is the desired object height in points. Works only for pdf_image and pdf_barcode objects. Optional.

- **-width** – Scales the object to the specified width. Requires a decimal value which is the desired object width in points. Works only for pdf_image and pdf_barcode objects. Optional.

For examples of using the **pdf_doc->add** method to add text, image, table, and barcode PDF objects to a pdf_doc object, see the corresponding sections in this chapter.

pdf_doc->getVerticalPosition()
> Returns the current vertical position where text will next be inserted on the page.

26.4.1 Adding Pages

If the content of a PDF file will span more than one page, additional pages can be added using special **pdf_doc** member methods. These methods signal where pages start and stop within the flow of the Lasso PDF creation methods.

pdf_doc->addPage()
> Adds additional blank pages to the pdf_doc object. When used, this method ends in the current page and starts a new page. Note that a new page will not be added if there is no content on the current page.
>
> The following example ends a preceding page, and starts a new page:

```
#my_file->add('Thus, ends the discussion on page 1.')
#my_file->addPage
#my_file->add('On page 2, we will discuss something else.')
```

pdf_doc->addChapter(text::string, -number::integer, -hideNumber=?**)**

pdf_doc->addChapter(text::pdf_text, -number::integer, -hideNumber=?**)**
> Adds a page with a named chapter title (and bookmark) to a pdf_doc object. Requires a text string or pdf_text object as a parameter, which specifies the chapter title. An additional **-number** parameter sets an integer chapter number for the chapter. An optional **-hideNumber** parameter specifies that no number will be shown.
>
> The following example adds a page with the text "30. Important Chapter" to the pdf_doc object with a defined chapter number of 30:

```
#my_file->addChapter(pdf_text('Important Chapter'), -number=30)
```

pdf_doc->setPageNumber(page::integer**)**
> Sets a page number for a new page. Requires an integer value.
>
> The following example sets a page number of 5 for the current page:

```
#my_file->setPageNumber(5)
```

`pdf_doc->`**`getPageNumber()`** → integer
> Returns the current page number.

> The following example returns a page number of 1 when used within the first page of the file:

```
#my_file->getPageNumber
// => 1
```

26.4.2 Adding Pages from Existing PDFs

Pages in existing PDF files can be added to a pdf_doc object using the **pdf_read** type. This type makes it possible to use existing PDF files as templates.

Note: Lasso cannot change existing text or graphics that are contained within a PDF file read in using **pdf_read**. Instead, Lasso is able to overlay text, graphics, and other elements on the PDF.

Once an existing PDF file has been read in as a Lasso object using **pdf_read**, it may be added to a pdf_doc object using the **pdf-doc->insertPage** method.

`pdf_doc->`**`insertPage(`**`pdf::pdf_read`, `number::integer`, ...**`)`**
> Inserts a page from a pdf_read object into a pdf_doc object. Requires a reference to a pdf_read object, followed by a comma and the number of the page to insert. This method has many optional parameters for specifying how an existing page should be inserted into a pdf_doc object. These parameters are explained below.

> Parameters
>> - **-newPage** – Keyword parameter specifying that the new page should be appended at the end of the file. Otherwise the page is drawn over the first page in the pdf_doc object by default.
>>
>> - **-top** – If the page being inserted is shorter than the current pages in the pdf_doc object, this parameter may be used to specify the offset of the new page from the top of the current page frame in points.
>>
>> - **-left** – If the page being inserted is not as wide the current pages in the pdf_doc object, this parameter may be used to specify the offset of the new page from the left of the current page frame in points.
>>
>> - **-width** – Scales the inserted page by width. Requires either a point width value, or a percentage string (e.g. '50%').
>>
>> - **-height** – Scales the inserted page by height. Requires either a point height value, or a percentage string (e.g. '50%').

Insert an Existing Page Into a New PDF File

Use the **pdf_doc->insertPage** method with a defined pdf_read object. The example below makes the first page of "somepdf.pdf" the first page of the pdf_doc object. Content may then be overlaid on top of the new page using the methods described in the rest of this chapter:

```
local(new_pdf) = pdf_doc(-file='MyFile.pdf', -size='A4')
local(old_pdf) = pdf_read('/documents/somepdf.pdf')
#new_pdf->insertPage(#old_pdf, 1)
```

Insert an Existing Page at End of a New PDF File

Use the **pdf_doc->insertPage** method with the optional **-newPage** parameter. The example below adds the first page of the "somepdf.pdf" PDF after all existing pages in the pdf_doc object:

```
local(new_pdf) = pdf_doc(-file='MyFile.pdf', -size='A4')
local(old_pdf) = pdf_read('/documents/somepdf.pdf')
#new_pdf->insertPage(#old_pdf, 1, -newPage)
```

Position an Inserted Page

Use the **pdf_doc->insertPage** method with the optional **-top** and/or **-left** parameters. The example below places the inserted page 50 points away from the top and left sides of the new document page frame:

```
local(new_pdf) = pdf_doc(-file='MyFile.pdf', -size='A4')
local(old_pdf) = pdf_read('/documents/somepdf.pdf')
#new_pdf->insertPage(#old_pdf, 1, -top=50, -left=50)
```

26.5 Accessing PDF File Information

Parameter values of a pdf_doc object can be returned using special accessor methods. These methods return specific values such as the page size, margin size, or the value of any other pdf_doc data members described in the previous section. All PDF accessor methods are defined below.

pdf_doc->**getMargins()** → staticarray
> Returns the current page margins as a staticarray (: *left,right,top,bottom*).

pdf_doc->**getSize()** → staticarray
> Returns the current page size as a staticarray of width and height point values (: *width,height*).

pdf_doc->**getColor()** → string
> Returns the current color as a hex string.

pdf_doc->**getHeaders()**
> Returns all document headers as a map object in the form **map('header1' = 'content1','header2' = 'content2',...)**.

pdf_doc->**setFont**(*font::pdf_font*)
> Sets a font for all following text. The value is a pdf_font object.

26.5.1 Return PDF Page Margins

Use the **pdf_doc->getMargins** method. The following example returns the current margins of a defined pdf_doc object:

```
#my_file->getMargins
// => staticarray(72.0, 72.0, 72.0, 72.0)
```

26.5.2 Return PDF Page Size

Use the **pdf_doc->getSize** method. The following example returns the current sizes of a defined pdf_doc object:

```
#my_file->getSize
// => staticarray(595, 842)
```

26.5.3 Return PDF Base Font Color

Use the **pdf_doc->getColor** method. The following example returns the base font color of a defined pdf_doc object:

```
#my_file->getColor
// => #333333
```

26.6 Saving PDF Files

Once a pdf_doc object has been filled with the desired content, the **pdf_doc->close** method must be used to signal that the PDF file is finished and is ready to be written to file or served to a visitor's browser.

pdf_doc->close()
> Closes a pdf_doc object and commits it to file after all desired data has been added to it. Additional data may not be added to the specified object after this method is called.

26.6.1 Close a PDF File

Use the **pdf_doc->close** method after all desired modifications have been performed on the pdf_doc object.

```
local(my_file) = pdf_doc(
    -file='MyFile.pdf',
    -size='A4',
    -margins=(: 144.0, 144.0, 72.0, 72.0)
)
local(font) = pdf_font(-face='Helvetica', -size=36)
local(text) = pdf_text('I am a PDF document', -font=#font)
#my_file->add(#text)
#my_file->close
```

26.7 Creating Text Content

Text content is the most basic type of data within a PDF file. PDF text is first defined as a pdf_text object, and then added to a pdf_doc object using the **pdf_doc->add** method.

A pdf_text object may be positioned within the current PDF page using the **-left** and **-top** parameters of the **pdf_doc->add** method. Otherwise, if no positioning parameters are specified, the text will be added to the top left corner of the page by default.

26.7.1 Setting Fonts

Before adding text, it is important to first define the font and style for the text to determine how it will appear. This is done using the **pdf_font** type.

type **pdf_font**

pdf_font(*-face=?, -file=?, -size=?, -color=?, -encoding::string=?, -embed=?*)
> Stores all the specifications for a font style. This includes font family, size, style, and color. Parameters are used with the **pdf_font** creator method that define the font family, size, color, and specifications. The following parameters may be used with the **pdf_font** creator method.

>   Parameters

- **-face** – Specifies the font by its family name. Allowed font names are "Courier", "Courier-Bold", "Courier-BoldOblique", "Courier-Oblique", "Helvetica", "Helvetica-Bold", "Helvetica-BoldOblique", "Helvetica-Oblique", "Symbol", "Times-Roman", "Times-Bold", "Times-BoldItalic", "Times-Italic", and "ZapfDingbats". Optional.

- **-file** – Uses a font from a local font file. The file name and path to the font must be specified (e.g. "/Fonts/Courier.ttf"). This parameter may be used instead of the **-face** parameter. Optional.

- **-size** – Sets the font size in points. Requires an integer point value as a parameter (e.g. "14"). Optional.

- **-color** – Sets the font color. Requires a hex color string as a parameter (e.g. "#550000"). Defaults to "#000000" if not specified. Optional.

- **-encoding** – Sets the desired font encoding. The font encoding defaults to "CP1252" if not specified. TrueType fonts can be asked to return an array of supported encodings via the **pdf_font->getSupportedEncodings** method. Optional.

- **-embed** – Embeds the fonts used within the PDF file as opposed to relying on the client PDF reader for font information. Optional.

The following examples show how to set variables as pdf_font objects that define the font styles to be used with a pdf_text object.

Set a Basic Font Style

Set a variable as a pdf_font object. The following example sets a font style to be a standard "Helvetica" font with a size of 14 points. The font color is also set to green:

```
local(my_font) = pdf_font(-face='Helvetica', -size=14, -color='#005500')
```

Individual parameters may be viewed and changed in a pdf_font object using **pdf_font** member methods. These parameters are most useful for retrieving and setting information about a pdf_font object that was defined using the **-file** parameter, and are summarized below.

pdf_font->**setFace**(*face::string*)
> Changes the font face of the pdf_font object to one of the allowed font names.

pdf_font->**setColor**(*color::string*)

pdf_font->**setColor**(*color::pdf_color*)
> Changes the font color of the pdf_font object.

pdf_font->**setSize**(*size::integer*)
> Changes the font size of the pdf_font object.

pdf_font->**setEncoding**(*encoding::string*)
> Changes the encoding of the pdf_font object.

pdf_font->**setUnderline**(*on::boolean=true*)
> Sets or unsets the pdf_font object style to underlined.

pdf_font->**setBold**(*on::boolean=true*)
> Sets or unsets the pdf_font object style to bold.

pdf_font->**setItalic**(*on::boolean=true*)
> Sets or unsets the pdf_font object style to italic.

pdf_font->**getFace**()
> Returns the current font face of a pdf_font object.

pdf_font->**getColor**()
> Returns the current font color of a pdf_font object.

`pdf_font->`**`getSize()`**
> Returns the current font size of a pdf_font object.

`pdf_font->`**`getEncoding()`**
> Returns the current encoding of a pdf_font object.

`pdf_font->`**`getPSFontName()`**
> Returns the exact PostScript font name of the current font of a pdf_font object, e.g. "AdobeCorIDMinBd".

`pdf_font->`**`isTrueType()`**
> Returns "true" if the current font is a TrueType font.

`pdf_font->`**`getSupportedEncodings()`**
> Returns an array of all supported encodings for a current TrueType font face, e.g. "array('1252 Latin 1', '1253 Greek')".

`pdf_font->`**`getFullFontName()`**
> Returns the full TrueType name of the current font of a pdf_font object (e.g. "Comic Sans", "MS Negreta").

`pdf_font->`**`textWidth(`**_text::string_**`)`**
> Returns an integer value representing how wide (in pixels) the text would be using the current pdf_font object. Requires a string value that is the text for which the width is desired.

Change a Font Face

Use the **`pdf_font->setFace`** method. The following example sets a defined pdf_font object to a standard "Courier" font:

```
#my_font->setFace('Courier')
```

Change a Font Color

Use the **`pdf_font->setColor`** method. The following example sets a defined pdf_font object to the color red:

```
#my_font->setColor('#990000')
```

Underline a Font

Use the **`pdf_font->setUnderline`** method. The following example sets a predefined pdf_font object to use an underlined style:

```
#my_font->setUnderline
```

Return a Font Face

Use the **`pdf_font->getFace`** method. The following example returns the current font face of a defined pdf_font object:

```
#my_font->getFace
// => Courier
```

Return a Font Encoding

Use the **`pdf_font->getEncoding`** method. The following example returns the encoding of the current font face of a defined pdf_font object:

```
#my_font->getEncoding
// => Cp1252
```

26.7.2 Adding Text

PDF text content is constructed using the **pdf_text** type, which is then added to a pdf_doc object using the **pdf_doc->add** method. The **pdf_text** constructor method and parameters are described below.

type **pdf_text**

pdf_text(*text::string*, …)

> Creates a text object to be added to a pdf_doc object. The constructor method requires the text string to be added to the PDF file as the first parameter. Optional parameters are listed below.

> Parameters

>> • **-type** – Specifies the text type. This can be "Chunk", "Phrase", or "Paragraph". Different parameters are available for each of these types, as described below. Defaults to the "Paragraph" type if no **-type** parameter is specified. Optional.

>> • **-color** – Sets the font color. Requires a hex color string as a parameter (e.g. "#550000"). Defaults to "#000000" if not specified. Optional.

>> • **-backgroundColor** – Sets the text background color. Require a hex color string as a parameter (e.g. "#550000"). Optional.

>> • **-underline** – Keyword parameter underlines the text. Optional.

>> • **-textRise** – Sets the baseline shift for superscript. Requires a decimal value that specifies the text rise in points. Optional.

>> • **-font** – Sets the font for the specified text using a pdf_font object. The font defaults to the current inherited font if no **-font** parameter is specified. Optional.

>> • **-anchor** – Links the specified text to a URL. The value of the parameter is the URL string (e.g. *http://www.example.com*). Optional.

>> • **-name** – Sets the name of an anchor destination within a page. The value of the parameter is the anchor name (e.g. "Name"). Optional.

>> • **-goTo** – Links the specified text to a local anchor destination to go to. The value of the parameter is the local anchor name (e.g. "Name"). Optional.

>> • **-file** – Links the specified text to a PDF file. The value of the parameter is a PDF file name (e.g. "Somefile.pdf"). The **-goTo** parameter can be used concurrently to specify an anchor name within the destination file. Optional.

>> • **-leading** – Sets the leading space in points (the space above each line of text), requires a decimal value. For "Phrase" and "Paragraph" types only.

>> • **-align** – Sets the alignment of the text in the page (`'Left'`, `'Center'`, or `'Right'`). Optional.

>> • **-indentLeft** – Sets the left indent of the text object. Requires a decimal value which is the number of points to indent the text. Optional. Available for "Paragraph" types only.

>> • **-indentRight** – Sets the right indent of the text object. Requires a decimal value which is the number of points to indent the text. Optional. Available for "Paragraph" types only.

The following examples show how to add text to a defined PDF variable named "my_file" that has been initialized previously using the **pdf_doc** method.

Add a Chunk of Text

Use the **pdf_text** type with the **-type='Chunk'** parameter. The following example adds the text "LassoSoft" to the pdf_doc object with a predefined font. The text is positioned in the top left corner of the page by default:

```
local(text) = pdf_text('LassoSoft', -type='Chunk', -font=#my_font)
#my_file->add(#text)
```

Add a Paragraph of Text

Use the **pdf_text** type with the **-type='Paragraph'** parameter. The following example adds three sentences of text to the pdf_doc object with a predefined font:

```
local(text) = pdf_text(
    "The mysterious file cabinet in orbit has been successfully lassoed. The \
        file cabinet had been traveling at a velocity of 300 meters per \
        second. Top scientists suspect that the cabinet had been in orbit for \
        some time.",
    -type='Paragraph',
    -font=#my_font,
    -leading=10.0,
    -indentLeft=20.0
)
#my_file->add(#text)
```

Add a Linked Phrase

Use the **pdf_text** type with the **-anchor** parameter. The following example adds the text "Click here to go somewhere" to the pdf_doc object with a predefined font, and links the phrase to *http://www.example.com*:

```
local(text) = pdf_text(
    "Click here to go somewhere",
    -type='Chunk',
    -font=#my_font,
    -anchor='http://www.example.com',
    -underline
)
#my_file->add(#text, -left=100.0, -top=100.0)
```

26.7.3 Adding Floating Text

Instead of adding text to the flow of the page, text can also be positioned on a page using the **pdf_doc->drawText** method. The **pdf_doc->drawText** method accepts coordinates that allow the text to be placed at an absolute position on the page.

pdf_doc->drawText(*text::string, -font=?, -alignment=?, -leading::decimal=?, -rotate::decimal=?, -left::integer=?, -top::integer=?, -width::integer=?, -height::integer=?***)**
> Adds specified text that is positioned on a page using point coordinates. An optional **-leading** parameter (decimal value) sets the text leading space in points (the space above each line of the text). A **-left** parameter specifies the placement of the left side of the text from the left side of the page in points, and a **-top** parameter specifies the placement of the bottom of the image from the bottom of the page in points (decimal value).

Note: The **pdf_doc->drawText** method is a graphics operation. It relies on the fill color set using the **pdf_doc->setColor** method. The color of the **-font** parameter will not be recognized.

Add Floating Text

Use the **pdf_doc->drawText** method. The following example adds the text "Some floating text" to the pdf_doc object with a predefined font at the coordinates specified in the **-top** and **-left** parameters. The coordinates represent the distance in points from the lower and left sides of the page:

```
#my_file->drawText('Some floating text',
    -font=#my_font,
    -left=144.0,
    -top=480.0
)
```

26.7.4 Adding Lists

A list of items can be constructed using the **pdf_list** type, which can be added to a pdf_doc object. The **pdf_list** constructor method and parameters are described below.

type **pdf_list**

pdf_list(…)
> Creates a list object to be added to a pdf_doc object. Text list items are added to this object using the **pdf_list->add** method. Optional parameters for this object are described in the table below.

>> Parameters

>>> • **-format** – Specifies whether the list is numbered, lettered, or bulleted. Requires a value of **'Number'**, **'Letter'**, **'Bullet'**. Defaults to "Bullet" if no **-format** parameter is specified. Optional.

>>> • **-bullet** – Specifies a custom character to use as the bullet character. Requires a character as a parameter (e.g. **'x'**). Defaults to the empty string if not specified. Optional.

>>> • **-indent** – Sets the space between the bullet and the list item. Requires a decimal or integer parameter which is the width of the indentation in points. Optional.

>>> • **-font** – Sets the font for the specified text using a pdf_font object. The font defaults to the current inherited font if no **-font** parameter is specified.

>>> • **-align** – Sets the alignment of the list in the page (**'Left'**, **'Center'**, or **'Right'**). Optional.

>>> • **-color** – Sets the font color. Requires a hex color string as a parameter (e.g. **'#550000'**). Defaults to "#000000" if not used. Optional.

>>> • **-backgroundColor** – Sets the text background color. Require a hex color string as a parameter (e.g. **'#550000'**). Optional.

>>> • **-leading** – Sets the list leading space in points (the space above each line of text), requires a decimal value. Optional.

pdf_list->add(*text::string*)

pdf_list->add(*text::pdf_text*)
> Add objects to the list. Requires a text string or a pdf_text object as a parameter.

Add a Numbered List

Use the **pdf_list** type with the **-format='Number'** parameter to define the list, and the **pdf_list->add** method to add items to the list. The example below creates a numbered list with three items:

```
local(list) = pdf_list(-format='Number', -align='Center', -font=#my_font)
#list->add('This is item one')
#list->add('This is item two')
#list->add('This is item three')
#my_file->add(#list)
```

Add a Bulleted List

Use the **pdf_list** type with the **-format='Bullet'** parameter to define the list, and the **pdf_list->add** method to add items to the list. The example below adds a bulleted list with four items, where a hyphen (**-**) is used as the bullet character:

```
local(list) = pdf_list(-format='Bullet', -bullet='-', -font=#my_font)
#list->add('This is item one')
#list->add('This is item two')
#list->add('This is item three')
#list->add('This is item four')
#my_file->add(#list)
```

26.7.5 Special Characters

When adding text to a pdf_doc object, escape sequences can be used to insert special characters such as line breaks, tabs, and more. These characters are summarized in the table below.

Table 26.1: Supported PDF Escape Sequences

Escape Sequence	Description
\n	line break (OS X and Linux)
\r\n	line break (Windows)
\t	tab
\"	double quote
\'	single quote
\\	backslash

Use Special Characters in a Text String

The following example shows how to use special characters within a pdf_doc text object:

```
#my_file->add('\\ \t \'Single Quotes\', \"Double Quotes\" ')
```

26.8 Creating and Using Forms

Forms can be created in PDF files for submitting information to a website. PDF forms use the same attributes as HTML forms, making them useful for submitting information to a website in place of an HTML form. This section describes how to create form elements within a PDF file, and also how PDF forms can be used to submit data to a Lasso-enabled database.

Note: Due to the iText implementation of PDF support in Lasso, created PDF files may contain only one form.

26.8.1 Creating Forms

Form elements are created in pdf_doc objects using **pdf_doc** form member methods which are described below.

pdf_doc->addTextField(*name::string*, *value::string*, *-left*, *-top*, *-width*, *-height*, *-font=?*)
>   Adds a text field to a form. Requires the first parameter to specify the name of the text field, and the second parameter to specify the default value entered. An optional **-font** parameter can be used to specify a pdf_font object for the font of the text.

pdf_doc->addPasswordField(*name::string*, *value::string*, *-left*, *-top*, *-width*, *-height*, *-font=?*)
>   Adds a password field to a form. Requires the first parameter to specify the name of the password field, and the second parameter to specify the default value entered. An optional **-font** parameter can be used to specify a pdf_font object for the font of the text.

pdf_doc->addTextArea(*name::string*, *value::string*, *-left*, *-top*, *-width*, *-height*, *-font=?*)
>   Adds a text area to a form. Requires the first parameter to specify the name of the text area, and the second parameter to specify the default value entered. An optional **-font** parameter can be used to specify a pdf_font object for the font of the text.

pdf_doc->addCheckBox(*name::string*, *value::string*, *-left*, *-top*, *-width*, *-height*, *-checked::boolean=?*)
>   Adds a checkbox to a form. Requires the first parameter to specify the name of the checkbox, and the second parameter to specify the value for the checkbox. An optional **-checked** parameter specifies that the checkbox is checked by default.

pdf_doc->addRadioGroup(*name::string*)
>   Adds a radio button group to a form. Requires a parameter specifying the name of the radio button group. Radio buttons must be assigned to the group using the **pdf_doc->addRadioButton** method.

pdf_doc->addRadioButton(*group::string*, *value::string*, *-left*, *-top*, *-width*, *-height*)
>   Adds a radio button to a form. Requires the first parameter to specify the name of the radio button group, and the second parameter to specify the value of the radio button.

pdf_doc->addComboBox(*name::string*, *values::trait_forEach*, *-default::string=?*, *-editable::boolean=?*, *-left*, *-top*, *-width*, *-height*, *-font=?*)
>   Adds a drop-down menu to a form. Requires the first parameter to specify the name of the drop-down menu, and the second parameter to specify the array of values contained in the menu **(: 'Value1','Value2')**. Optionally, the array passed as the second parameter can contain a pair for each value. The first element in the pair is the value to be used upon form submission, and the second element is the human-readable label to be used for display only.
>
>   An optional **-default** parameter specifies the name of a default value to select. An optional **-editable** parameter specifies that the user may edit the values on the menu. An optional **-font** parameter can be used to specify a pdf_font object for the font of the text.

pdf_doc->addSelectList(*name::string*, *values::trait_forEach*, *-default=*", *-left*, *-top*, *-width*, *-height*, *-font=?*)
>   Adds a select list to a form. Requires the first parameter to specify the name of the select list, and the second parameter to specify the array of values contained in the select list **(: 'Value1','Value2')**. Optionally, the array passed as the second parameter can contain a pair for each value. The first element in the pair is the value to be used upon form submission, and the second element is the human-readable label to be used for display only.
>
>   An optional **-default** parameter specifies the name of a default value to select. An optional **-font** parameter can be used to specify a pdf_font object for the font of the text.

pdf_doc->addHiddenField(*name::string*, *value::string*)
>   Adds a hidden field to a form. Requires the first parameter to specify the name of the hidden field and the second parameter to specify the default value entered.

pdf_doc->addSubmitButton(*name::string*, *caption::string*, *value::string*, *url::string*, *-left*, *-top*, *-width*, *-height*, *-font=?***)**

Adds a submit button to a form. Also specifies the URL to which the form data will be submitted. Requires the first parameter to specify the name of the button. The second parameter specifies a caption (displayed name) for the button. The third parameter is the value for the submit button, and the fourth parameter specifies the URL of the response page. An optional **-font** parameter can be used to specify a pdf_font object for the font of the text.

pdf_doc->addResetButton(*name::string*, *caption::string*, *value::string*, *-left*, *-top*, *-width*, *-height*, *-font=?***)**

Adds a reset button to a form. Requires the first parameter to specify the name of the button, the second parameter specifies a caption (displayed name) for the button, and the third parameter specifies the value for the button. An optional **-font** parameter can be used to specify a pdf_font object for the font of the text.

Note: With the exception of the **pdf_doc->addSubmitButton** and **pdf_doc->addResetButton** methods, no form input element methods include captions or labels with the field elements. Field captions and labels can be applied using the **pdf_text** and **pdf_doc->add** methods to position text appropriately. See the section ***Creating Text Content*** for more information.

All **pdf_doc** form member methods, with the exception of **addHiddenField** and `addRadioButtonGroup`, require placement parameters for specifying the exact positioning of form elements within a page. These parameters are summarized in the table ***Form Placement Parameters***.

Table 26.2: Form Placement Parameters

Parameter	Description
`-left`	Specifies the placement of the left side of the form element from the left side of the current page in points. Requires a decimal value.
`-top`	Specifies the placement of the bottom of the form element from the bottom of the current page in points. Requires a decimal value.
`-width`	Specifies the width of the form element in points. Requires a decimal value.
`-height`	Specifies the height of the form element in points. Requires a decimal value.

Add a Text Field

Use the **pdf_doc->addTextField** method. The example below adds a field named "Field_Name" that has "Some Text" entered by default. The field size is 144.0 points (two inches) wide and 36.0 points high:

```
#my_file->addTextField(
    'Field_Name',
    'Some Text',
    -font=#my_font,
    -left=72.0, -top=350.0, -width=144.0, -height=36.0
)
```

Add a Text Area

Use the **pdf_doc->addTextArea** method. The example below adds a text area named "Field_Name" that has the text "Insert default text here" entered by default. The field size is 144.0 points wide and 288.0 points high:

```
#my_file->addTextArea(
    'Field_Name',
    'Insert default text here',
    -font=#my_font,
```

```
      -left=72.0, -top=300.0, -width=144.0, -height=288.0
)
```

Add a Checkbox

Use the **pdf_doc->addCheckbox** method. The example below adds a field named "Field_Name" with a checked value of "Checked_Value" that is checked by default. The checkbox is 4.0 points wide and 4.0 points high, and is positioned 272.0 points from the bottom and left sides of the page:

```
#my_file->addCheckBox(
    'Field_Name',
    'Checked_Value',
    -checked,
    -left=272.0, -top=272.0, -width=4.0, -height=4.0
)
```

Add a Group of Radio Buttons

Use the **pdf_doc->addRadioGroup** and **pdf_doc->addRadioButton** methods. The example below adds a radio button group named "Group_Name" and adds two radio buttons with the values of "Yes" and "No". The radio buttons are 6.0 points wide and 6.0 points high each:

```
#my_file->addRadioGroup('Group_Name')
#my_file->addRadioButton(
    'Group_Name',
    -value='Yes',
    -left=72.0, -top=372.0, -width=6.0, -height=6.0
)
#my_file->addRadioButton(
    'Group_Name',
    -value='No',
    -left=90.0, -top=372.0, -width=6.0, -height=6.0
)
```

Note: If the **pdf_doc->addRadioGroup** method is not used, then radio buttons will not appear in the form.

Add an Editable Drop-Down Menu

Use the **pdf_doc->addComboBox** method. The example below adds a drop-down menu named "Menu_Name" with the values "One", "Two", "Three", and "Four" as menu values. The value "One" is selected by default, and an **-editable** parameter allows the users to edit the values if desired. The drop-down menu size is 144.0 points wide and 36.0 points high:

```
#my_file->addComboBox(
    'List_Name',
    (: 'One', 'Two', 'Three', 'Four'),
    -default='One',
    -editable,
    -left=72.0, -top=272.0, -width=144.0, -height=36.0
)
```

Add a Drop-Down Menu with Different Displayed Values

Use the **pdf_doc->addComboBox** method whose values are each pairs. The example below adds a drop-down menu named "Menu_Name" with the values "1", "2", "3", and "4" as submittable menu values, but displays the names "One", "Two", "Three", and "Four" for each value. No value is selected by default:

```
#my_file->addComboBox(
    'List_Name',
    (: pair(1 = 'One'),
       pair(2 = 'Two'),
       pair(3 = 'Three'),
       pair(4 = 'Four')
    ),
    -left=72.0, -top=272.0, -width=144.0, -height=36.0
)
```

Add a Select List

Use the **pdf_doc->addSelectList** methods. The example below adds a select list named "List_Name" with the values "One", "Two", "Three", and "Four" as list items. The select list is 144.0 points wide and 288.0 points high, and is positioned 72.0 points from the bottom and left sides of the page:

```
#my_file->addSelectList(
    'List_Name',
    (: 'One', 'Two', 'Three', 'Four'),
    -default='One',
    -left=72.0, -top=72.0, -width=144.0, -height=288.0
)
```

Add a Hidden Field

Use the **pdf_doc->addHiddenField** method. The example below adds a hidden field named "Field_Name" with a value of "Hidden_Value" to a pdf_doc object named "my_file". No placement coordinates are needed because the field is not displayed on the page:

```
#my_file->addHiddenField('Field_Name', 'Some_Value')
```

Add a Submit Button

Use the **pdf_doc->addSubmitButton** method. The example below adds a submit button named "Button_Name" with a value of "Submitted_Value". A caption parameter specifies the displayed name of the button, which is "Submit This Form". The URL parameter specifies that the user will be taken to *http://www.example.com/response.lasso* when the button is selected in the form:

```
#my_file->addSubmitButton(
    'Button_Name',
    'Submit This Form',
    'Submitted_Value',
    'http://www.example.com/response.lasso',
    -left=72.0, -top=72.0, -width=144.0, -height=36.0
)
```

Add a Reset Button

Use the **pdf_doc->addResetButton** method. The example below adds a reset button named "Button_Name" with a value of "Reset_Value". The caption parameter specifies the displayed name of the button, which is "Reset This Form":

```
#my_file->addResetButton(
    'Button_Name',
    'Reset This Form',
    'Reset_Value',
    -left=72.0, -top=72.0, -width=144.0, -height=36.0
)
```

26.8.2 Submitting Form Data to Lasso-Enabled Databases

Using Lasso Server, one has the ability to submit data from a PDF form to a Lasso-enabled site for interaction with a database. PDF forms may be used in the same way as HTML forms to submit request parameters to a Lasso response page, where database actions can occur via an **inline** method.

Submit Information to a Database Using a PDF Form

1. In the "form.lasso" page, name the PDF form fields to correspond to the names of fields in the desired database. The names of these fields will be used in the **inline** method in the Lasso response page.

```
local(my_file) = pdf_doc(-file='form.pdf', -size='A4')
local(my_font) = pdf_font(-face='Helvetica', -size=12)
#my_file->drawText('First Name:', -font=#my_font, -left=80.0, -top=60.0)
#my_file->drawText('Last Name:',  -font=#my_font, -left=80.0, -top=60.0)
#my_file->addTextField(
    'First Name',
    'Enter First Name',
    -left=144.0, -top=72.0, -width=144.0, -height=36.0
)
#my_file->addTextField(
    'Last Name',
    'Enter Last Name',
    -left=144.0, -top=92.0, -width=144.0, -height=36.0
)
```

2. Create a submit button in the "form.lasso" page that contains the name and URL of the Lasso response page.

```
#my_file->addSubmitButton(
    'Search',
    'Click here to Search',
    'Search',
    'http://www.example.com/response.lasso',
    -font=#my_font,
    -left=144.0, -top=122.0, -width=80.0, -height=36.0
)
#my_file->close
```

After the pdf_doc object is closed and executed on the server, a "form.pdf" file will be created with the form.

3. In the "response.lasso" page, create an **inline** method that uses the action parameters passed from the PDF form to perform a database action. This example performs a search on the "Contacts" database using the values for "first_name" and "last_name" passed from the PDF form.

```
inline(
    -search,
    -database='contacts',
    -table='people',
    -keyField='id',
    'first_name'=web_request->param('first_name'),
    'last_name'=web_request->param('last_name')
) => {^
    'There were ' + found_count + ' record(s) found in the People table.\n'
    records => {^
        '<br />' + field('first_name') + ' ' + field('last_name') + '\n'
    ^}
^}
```

If the user of the PDF form entered "Jane" for the first name and "Doe" for the last name, then the following results would be returned:

```
// =>
// There were 1 record(s) found in the People table.
// <br />Jane Doe
```

You could also use this method to update data in a database.

26.9 Creating Tables

Tables can be created in PDF files for displaying data. These are created using the **pdf_table** type and added to a PDF object using **pdf_doc** member methods, which are described in this section.

26.9.1 Defining Tables

Tables for organizing data can be defined for use in a PDF file using the **pdf_table** type. Objects of this type are added to a pdf_doc object.

type **pdf_table**

pdf_table(*cols::integer, rows::integer, …*)

> Creates a table to be placed in a PDF. Uses parameters that set the basic specifications of the table to be created. The first parameter is required and specifies the number of columns in the table. The second parameter is also required and specifies the number of rows in the table. Below is a list of optional parameters for the **pdf_table** constructor method.

> Parameters
>> • **-spacing** – Specifies the spacing around a table cell. Defaults to "0" (no spacing) if not specified. Optional.
>>
>> • **-padding** – Specifies the padding within a table cell. Defaults to "0" (no padding) if not specified. Optional.
>>
>> • **-width** – Specifies the width of the table as a percentage of the current page width. Defaults to the width of the cell text plus spacing, padding, and borders if not specified. Optional.
>>
>> • **-borderWidth** – Specifies the border width of the table in points. Requires a decimal value. Optional.
>>
>> • **-borderColor** – Specifies the border color of the table. Requires a hex color string (e.g. **'#000000'**). Optional.
>>
>> • **-backgroundColor** – Specifies the background color of the table. Requires a hex color string (e.g. **'#CC-CCCC'**). Optional.

- **-colWidth** – Sets the column width for each column in the table. Requires an array of decimals representing the width percentage of each column. Optional.

Member methods can be used to set additional specifications for a pdf_table object, as well as access data member values from pdf_table objects. These methods are summarized below.

pdf_table->getColumnCount()
> Returns the number of columns in a pdf_table object.

pdf_table->getRowCount()
> Returns the number of rows in a pdf_table object.

pdf_table->getAbsWidth()
> Returns the total pdf_table object width in pixels.

Create a Basic Table

Use the **pdf_table** type. The example below creates a table with two columns and five rows, with table cell spacing of one point and cell padding of two points. The width of the table is set at 75% of the current page width:

```
local(my_table) = pdf_table(
    2,
    5,
    -spacing=1,
    -padding=2,
    -width=75,
    -backgroundColor='#CCCCCC'
)
```

Create a Table with a Border

Use the **pdf_table** type with the **-borderWidth** and **-borderColor** parameters. The example below creates a basic table, and then adds a black border with a width of 3 points to the table:

```
local(my_table) = pdf_table(
    2,
    5,
    -spacing=1,
    -padding=2,
    -borderWidth=3,
    -borderColor='#000000'
)
```

Rotate a Table

Use the **pdf_table** type with the **-rotate** parameter. The example below creates a basic table, and then rotates it by 90 degrees clockwise:

```
local(my_table) = pdf_table(
    2,
    5,
    -spacing=1,
    -padding=2,
    -rotate=90
)
```

Create a Table with Specific Column Widths

Use the **pdf_table** type with the **-colWidth** parameter. The example below creates a basic table with percentage widths for three columns:

```
local(my_table) = pdf_table(
    2,
    5,
    -spacing=1,
    -padding=2,
    -colWidth=(: '50.0', '25.0', '25.0')
)
```

26.9.2 Adding Content to Table Cells

Content is added to table cells using additional **pdf_table** member methods which are summarized below.

pdf_table->add(_str::string_, _col::integer_, _row::integer_, ...**)**

pdf_table->add(_text::pdf_text_, _col::integer_, _row::integer_, ...**)**

pdf_table->add(_table::pdf_table_, _col::integer_, _row::integer_, ...**)**

pdf_table->add(_image::pdf_image_, _col::integer_, _row::integer_, ...**)**

pdf_table->add(_barcode::pdf_barcode_, _col::integer_, _row::integer_, ...**)**
> Inserts text content, a new nested table, an image, or a barcode into a cell. Requires a string, **pdf_text**, **pdf_table**, **pdf_image**, or **pdf_barcode** object to be inserted as the first parameter. Also requires specifying the column number as the second parameter and row number as the third parameter. Row and columns numbers start from "0" with rows increasing from top to bottom and columns increasing from left to right. The table below lists the optional parameters that can also be specified.
>
>> Parameters
>>
>>> • **-colspan** – Specifies the number of columns a cell should span. If specified, requires an integer value "1" or greater. Optional.
>>>
>>> • **-rowspan** – Specifies the number of rows a cell should span. If specified, requires an integer value "1" or greater. Optional.
>>>
>>> • **-verticalAlignment** – Vertical alignment for text within a cell. Accepts a value of **'Top'**, **'Center'**, or **'Bottom'**. Defaults to "Center" if not specified. Optional.
>>>
>>> • **-horizontalAlignment** – Horizontal alignment for text within a cell. Accepts a value of **'Left'**, **'Center'**, or **'Right'**. Defaults to "Center" if not specified. Optional.
>>>
>>> • **-borderColor** – Specifies the border color for the cell (e.g. **'#440000'**). Defaults to "#000000" if not specified. Optional.
>>>
>>> • **-borderWidth** – Specifies the border width of the cell in points. Requires an integer value. Defaults to "0" if not specified. Optional.
>>>
>>> • **-header** – Specifies that the cell is a table header. This is typically used for cells in the first row. Optional.
>>>
>>> • **-noWrap** – Specifies that the text contained in a cell should not wrap to conform to the cell size specifications. If used, the cell will expand to the right to accommodate longer text strings. Optional.

Add a Cell to a Table

Use the **pdf_table->add** method. The example below adds a cell to the first row and column in a table. Note that the first row and column are numbered "0":

```
#my_table->add(
   'This is the first cell in my table',
   0,
   0,
   -colspan=1,
   -rowspan=1
)
```

Add a Multi-Column Cell to a Table

Use the **pdf_table->add** method with the number of columns to span for the **-column** parameter. The example below adds a cell to the first row that spans three columns. The **-noWrap** parameter is used to indicate that the added text will not be wrapped into multiple lines:

```
#my_table->add(
   'This text will only stay on one line regardless of the table size',
   0,
   0,
   -colspan=3,
   -rowspan=1,
   -noWrap
)
```

Add a Header Cell to a Table

Use the **pdf_table->add** method with the **-header** parameter. The example below adds the header "My Column Title" to the first column of the table:

```
#my_table->add(
   'My Column Title',
   0,
   0,
   -header
)
```

Add a Cell with a Border to a Table

Use the **pdf_table->add** method with the **-borderWidth** and **-borderColor** parameter. The example below adds a cell with a red border to the first column of the table:

```
#my_table->add(
   'This cell has a border',
   0,
   0,
   -borderWidth=45.0,
   -borderColor='#440000'
)
```

26.9.3 Adding Tables

Once a pdf_table object is completely defined and has cell content, it may then be added to a pdf_doc object using the **pdf_doc->add** method.

Add a Table to a pdf_doc Object

Use the **pdf_doc->add** method. The following example adds a predefined pdf_table object named "my_table" to a pdf_doc object named "my_file":

```
#my_file->add(#my_table)
```

26.10 Creating Graphics

This section describes how to draw custom graphic objects and insert image files within a PDF file.

26.10.1 Inserting Images

Image files can be placed within PDF pages using the **pdf_image** type in conjunction with the **pdf_doc->addImage** method as documented below.

type **pdf_image**

pdf_image(…)
> Reads an image file as a Lasso object so it can be placed into a PDF file. Requires either a **-file**, **-url**, or **-raw** parameter, as described in the list below. Only images in JPEG, GIF, PNG, and WMF formats may be used.

> > Parameters
> > > - **-file** – Specifies the local path to an image file. Required if the **-url** or **-raw** parameters are not used.
> > > - **-url** – Specifies a URL to an image file. Required if the **-file** or **-raw** parameters are not used.
> > > - **-raw** – Inputs a raw string of bits representing the image. Required if the **-url** or **-file** parameters are not used.
> > > - **-height** – Scales the image to the specified height. Requires a decimal value which is the desired image height in points. Optional.
> > > - **-width** – Scales the image to the specified width. Requires a decimal value which is the desired image width in points. Optional.
> > > - **-proportional** – Keyword parameter specifying that all scaling should preserve the aspect ratio of the inserted page. Optional.
> > > - **-rotate** – Rotates the image by the specified degrees clockwise. Optional.

Add an Image File to a pdf_doc Object

Use the **pdf_image** type. The following example adds a file named "Image.jpg" in a "/Documents/Images/" folder to a pdf_doc object named "my_file":

```
local(image) = pdf_image(-file='/Documents/Images/Image.jpg')
#my_file->add(#image, -left=144.0, -top=300.0)
```

Scale an Image File

Use the **pdf_image** type with the `-height` or `-width` parameter. The following example proportionally reduces the size of the added image by 50%:

```
local(image) = pdf_image(-file='/Documents/Images/Image.jpg', -height='50%')
#my_file->add(#image, -left=144.0, -top=300.0)
```

Rotate an Image File

Use the **pdf_image** type with the `-rotate` parameter. The following example rotates the added image by 90 degrees clockwise:

```
local(image) = pdf_image(-file='/Documents/Images/Image.jpg', -rotate=90.0)
#my_file->add(#image, -left=144.0, -top=300.0)
```

26.10.2 Drawing Graphics

To draw custom graphics, Lasso uses a coordinate system to determine the placement of each graphical object. This coordinate system is a standard coordinate plane with horizontal (X) vertical (Y) axis, where a point on a page is defined by an array containing horizontal and vertical position values "(X, Y)". The base point of the coordinate plane "(0, 0)" is located in the lower left corner for the current page. Increasing an X-Value moves a point to the right in the page, and increasing the Y-Value moves the point up in the page. The current width and height of the page in points defines the maximum X and Y values.

Custom graphics may be drawn in PDF pages using **pdf_doc** drawing member methods. These member methods operate by controlling a "virtual pen" which draws graphics similar to a true graphics editor. These member methods are summarized below.

pdf_doc->**setColor**(*type::string, color::pdf_color*)

pdf_doc->**setColor**(*type::string, color::string, …*)
> Sets the color and style for subsequent drawing operations on the page. Requires the first parameter to specify whether the drawing action is of type "Stroke", "Fill", or "Both". The second parameter is also required and is either a pdf_color object or a string that specifies a color type of "Gray", "RGB", or "CMYK". If "Gray" is specified, a decimal specifies a color strength value. If "RGB" is specified, three decimal values specify red, green, and blue values, respectively. If "CMYK" is specified, four decimal values specify cyan, magenta, yellow, and black values, respectively. Color values are specified as decimals ranging from "0" to "1.0".

pdf_doc->**setLineWidth**(*width::decimal*)
> Sets the line width for subsequent drawing actions on the page in points. Requires a decimal point value.

pdf_doc->**line**(*x1, y1, x2, y2*)
> Draws a line. Requires a set of integer points which specifies the starting point and ending point of the line.

pdf_doc->**curveTo**(*x1, y1, x2, y2, x3, y3*)
> Draws a curve. Requires a set of integer points as parameters which specifies the starting point, middle point, and ending point of the curve.

pdf_doc->**rect**(*x, y, width, height, -fill::boolean=?*)
> Draws a rectangle. Requires the first two parameters to be a set of "X" and "Y" integer points which specifies the lower right corner of the rectangle, and the next two parameters specify the height and width of the rectangle sides from that coordinate. An optional `-fill` parameter draws a filled rectangle.

pdf_doc->**circle**(*x, y, radius, -fill::boolean=?*)
> Draws a circle. Requires the first two parameters to be a set of integer points for the center coordinates of the circle and the third parameter to be the length of the radius. An optional `-fill` parameter draws a filled circle.

`pdf_doc->`**`arc`**`(`*x, y, radius, start, end, -fill::boolean=?*`)`

> Draws an arc. Requires the first two parameters to be a set of integer points for the center coordinates of the arc and the third parameter to be the radius of the invisible circle to which the arc belongs. The fourth parameter must be a starting degree which specifies the degrees of the circle at which the arc starts, and the fifth parameter must be an ending degree which specifies the circle degrees at which the arc ends. Angles start with "0" to the right of the center and increase counter-clockwise. An optional **`-fill`** parameter draws a filled arc.

Note: The color and line width must be set on each new page of the PDF prior to calling any drawing methods.

Set Color and Style for a Drawing Action

Use the **`pdf_doc->setColor`** method. The example below sets a color of red for all subsequent drawing action until another **`pdf_doc->setColor`** method is called:

```
#my_file->setColor('Stroke', 'RBG', 0.1, 0.9, 0.9)
```

The example below sets the fill color of red for all subsequent drawing action until another **`pdf_doc->setColor`** method is called. The methods to draw rectangles, circles, or arcs must be called with the optional **`-fill`** parameter for this color choice to be applied:

```
#my_file->setColor('Fill', 'RBG', 0.1, 0.9, 0.9)
```

Set Line Width of a Drawing Action

Use the **`pdf_doc->setLineWidth`** method. The example below sets a line width of 5 points for all subsequent drawing action until another **`pdf_doc->setLineWidth`** method is called:

```
#my_file->setLineWidth(5.0)
```

Draw a Line

Use the **`pdf_doc->line`** method. The example below draws a horizontal line from points "(8, 8)" to points "(32, 32)":

```
#my_file->line(8, 8, 32, 32)
```

Draw a Curve

Use the **`pdf_doc->curveTo`** method. The example below draws a curve starting from points "(8, 8)", peaking at points "(32, 32)", and ending at points "(56, 8)":

```
#my_file->curveTo(8, 8, 32, 32, 56, 8)
```

Draw a Filled Rectangle

Use the **`pdf_doc->rect`** method. The example below draws a rectangle whose lower left corner is at coordinates "(10, 60)", has left and right sides that are 50 points long, and has top and bottom sides that are 20 points long. The optional **`-fill`** parameter ensures this rectangle has the current fill color applied:

```
#my_file->rect(10, 60, 20, 50, -fill)
```

Draw a Circle

Use the **pdf_doc->circle** method. The example below draws a circle whose center is at coordinates "(50, 50)" and has a radius of 20 points:

```
#my_file->circle(50, 50, 20)
```

Draw an Arc

Use the **pdf_doc->arc** method. The example below draws an arc whose center is at coordinates (50, 50), has a radius of 20 points, and runs from 0 degrees to 90 degrees from the center:

```
#my_file->arc(50, 50, 20, 0, 90)
```

26.11 Creating Barcodes

Barcodes are special device-readable images that can be created in PDF files using the **pdf_barcode** type, and added to a pdf_doc using member methods, which are described in this section. Lasso can be used to create the following industry-standard barcodes:

- Code 39 (alphanumeric, ASCII subset)
- Code 39 Extended (alphanumeric, escaped text)
- Code 128
- Code 128 UCC/EAN
- Code 128 Raw
- EAN (8 digits)
- EAN (13 digits)
- POSTNET
- PLANET

Barcodes can be defined for use in a PDF file using the **pdf_barcode** type. Objects of this type can then be added to pdf_doc objects.

type **pdf_barcode**

pdf_barcode(…)

> Creates a barcode image to be placed in a PDF. Uses parameters which set the basic specifications of the barcode to be created.

> Parameters

>> - **-type** – Specifies the type of barcode to be created. Available parameters are **'CODE39'**, **'CODE39_EX'**, **'CODE128'**, **'CODE128_UCC'**, **'CODE128_RAW'**, **'EAN8'**, **'EAN13'**, **'POSTNET'**, and **'PLANET'**. Required.

- **-code** – Specifies the numeric or alphanumeric barcode data. Some formats require specific data strings: "EAN8" requires an 8-digit integer, "EAN13" requires a 13-digit integer, "POSTNET" requires a ZIP code, and "CODE39" requires uppercase characters. Required.

- **-color** – Specifies the color of the bars in the barcode. Requires a hex string color value. Defaults to "#000000" if not specified. Optional.

- **-supplemental** – Adds an additional two or five-digit supplemental barcode to "EAN8" or "EAN13" barcode types. Requires a two or five-digit integer as a parameter. Optional.

- **-generateChecksum** – Generates a checksum for the barcode. Optional.

- **-showCode39StartStop** – Displays start and stop characters ("*") in the text for Code 39 barcodes. Optional.

- **-showEANGuardBars** – Show the guard bars for "EAN" barcodes. Optional.

- **-barHeight** – Sets the height of the bars in points. Requires a decimal value.

- **-barWidth** – Sets the width of the bars in points. Requires a decimal value.

- **-baseLine** – Sets the text baseline in points. Requires a decimal value.

- **-showChecksum** – Keyword parameter sets the generated checksum to be shown in the text.

- **-font** (*pdf_font*) – Sets the text font. Requires a pdf_font object.

- **-barMultiplier** – Sets the bar multiplier for wide bars. Requires a decimal value.

- **-textSize** – Sets the size of the text. Requires a decimal value.

26.11.1 Create a Barcode

Use the **pdf_barcode** type. The example below creates a basic Code 39 barcode with the data "1234567890", and uses the optional Code 39 start and stop characters ("*"). The barcode is then added to a pdf_doc object using **pdf_doc->add**:

```
local(barcode) = pdf_barcode(
    -type='CODE39',
    -code='1234567890',
    -showCode39StartStop
)
#my_pdf->add(#barcode, -left=150.0, -top=100.0)
```

26.11.2 Create a Barcode with a Specified Bar Width

Use the **pdf_barcode** type with the **-barWidth** parameter. The following example sets a pdf_barcode object with a bar width of 0.2 points:

```
local(barcode) = pdf_barcode(
    -type='CODE39',
    -code='1234567890',
    -barWidth=0.2
)
#my_pdf->add(#barcode, -left=150.0, -top=100.0)
```

26.11.3 Create a Barcode with a Specified Bar Multiplier

Use the **pdf_barcode** type with the **-barMultiplier** parameter. The following example sets a pdf_barcode object with a bar multiplier constant of "4.0". The barcode is then added to a pdf_doc object using **pdf_doc->add**:

```
local(barcode) = pdf_barcode(
    -type='CODE39',
    -code='1234567890',
    -barMultiplier=4.0
)
#my_pdf->add(#barcode, -left=150.0, -top=100.0)
```

26.11.4 Create a Barcode with a Specified Text Size

Use the **pdf_barcode** type with the **-textSize** parameter. The following example sets a pdf_barcode object with a text size of 6.0 points. The barcode is then added to a pdf_doc object using **pdf_doc->add**:

```
local(barcode) = pdf_barcode(
    -type='CODE39',
    -code='1234567890',
    -textSize=6.0
)
#my_pdf->add(#barcode, -left=150.0, -top=100.0)
```

26.11.5 Create a Barcode with a Specified Font

Use the **pdf_barcode** type with the **-font** parameter. The following example sets a pdf_barcode object font specified in a pdf_font object named "my_font". The barcode is then added to a pdf_doc object using **pdf_doc->add**:

```
local(barcode) = pdf_barcode(
    -type='CODE39',
    -code='1234567890',
    -font=#my_font
)
#my_pdf->add(#barcode, -left=150.0, -top=100.0)
```

26.12 PDF File Examples

This section provides complete examples of creating PDF files using the methods described in this chapter. Examples include a two-page PDF file with multiple text styles, a PDF file with a form, a PDF file with a table, a PDF file with drawn graphics, and a PDF file with a barcode.

Note: All examples in this section use the OS X and Linux line break character **"\n"** in the text sections. If creating PDF files on the Windows version of Lasso, change all instances of **"\n"** to **"\r\n"**.

26.12.1 PDF Text Example

The following example creates a PDF file that contains two pages of text with multiple text styles:

```
local(text_example) = pdf_doc(-file='Text_Example.pdf', -size='A4')
#text_example->addPage
#text_example->setPageNumber(1)

local(font1) = pdf_font(-face='Helvetica', -size='24', -color='#990000')
local(font2) = pdf_font(-face='Helvetica', -size='14', -color='#000000')
local(font3) = pdf_font(-face='Helvetica', -size='14', -color='#0000CC')

local(title) = pdf_text('Lasso Server', -type='Chunk', -font=#font1)
#text_example->add(#title, -number=1)

local(text1) = pdf_text("\n\nThe Lasso product line consists of authoring and
    serving tools that allow web designers and web developers to quickly build
    and serve powerful data-driven web sites with maximum productivity and
    ease. The product line includes Lasso Server for serving and administering
    data-driven web sites, and LassoLab for building and testing data-driven
    web sites within a graphical editor.\n\nLasso Server works with the
    following data sources:",
    -type='Paragraph',
    -leading=15,
    -font=#font2
)
#text_example->add(#text1)

local(list) = pdf_list(
    -format='Bullet',
    -bullet='-',
    -font=#font2,
    -indent=30
)
#list->add('FileMaker Server')
#list->add('MySQL')
#list->add('Microsoft SQL Server')
#list->add('Frontbase')
#list->add('Sybase')
#list->add('PostgreSQL')
#list->add('DB2')
#list->add('Plus many other ODBC-compliant databases')
#text_example->add(#list)

local(text2) = pdf_text("\nLasso's innovative architecture provides an
    industry-first multi-platform, database-independent and open standards
    approach to delivering database-driven web sites firmly positioning Lasso
    technology within the rapidly evolving server-side web tools market. Lasso
    technology is used on hundreds of thousands of web sites worldwide.\n\n",
    -type='Paragraph',
    -font=#font2
)
#text_example->add(#text2)

local(text3) = pdf_text(
    "Click here to go to the LassoSoft website",
    -type='Phrase',
    -font=#font3,
    -underline='true',
    -anchor='http://www.lassosoft.com'
)
#text_example->add(#text3)
```

```
#text_example->drawText(
    #text_example->getPageNumber->asString,
    -font=#font2,
    -top=30,
    -left=560
)
#text_example->addPage

#text_example->setPageNumber(2)

local(text4) = pdf_text("Lasso Server is server-side software that adds a
    suite of dynamic functionality and administration to your web server. This
    functionality empowers you to build and serve just about any dynamic web
    application and do so with maximum productivity and ease.\n\n",
    -type='Paragraph',
    -leading=15,
    -font=#font2
)
#text_example->add(#text4)

local(text5) = pdf_text("Lasso works by using a simple scripting language,
    which can be embedded in web pages and scripts residing on your web
    server. By default, Lasso Server is designed to run on the most prevalent
    modern web server platforms with the most popular web serving
    applications. Additionally, Lasso's extensibility allows web server
    connectors to be authored for any web server for which default
    connectivity is not provided.\n\n",
    -type='Paragraph',
    -leading=15,
    -font=#font2
)
#text_example->add(#text5)

#text_example->drawText(
    #text_example->getPageNumber->asString,
    -font=#font2,
    -top=30,
    -left=560
)
#text_example->close
```

26.12.2 PDF Form Example

The following example creates a PDF file that contains both text and a form:

```
local(form_example) = pdf_doc(-file='Form_Example.pdf', -size='a4')
local(myFont)       = pdf_font(-face='Helvetica', -size='12')

#form_example->addText(
    'This PDF file contains a form. See below.\n',
    -font=#myFont
)
#form_example->drawText('Select List', -font=#myFont, -left=90, -top=116)
#form_example->addSelectList(
    'mySelectList',
```

```
    (: 'one', 'two', 'three', 'four'),
    -default='one',
    -left=216, -top=104, -width=144, -height=72,
    -font=#myFont
)
#form_example->drawText(
    'Drop-Down Menu',
    -font=#myFont,
    -left=90,
    -top=200
)
#form_example->addComboBox(
    'myComboBox',
    (: 'one', 'two', 'three', 'four'),
    -default='one',
    -left=216, -top=188, -width=144, -height=18,
    -font=#myFont
)
#form_example->drawText('Text Area', -font=#myFont, -left=90, -top=238)
#form_example->addTextArea(
    'myTextArea',
    'Some text',
    -left=216, -top=230, -width=144, -height=72,
    -font=#myFont
)
#form_example->drawText('Password Field', -font=#myFont, -left=90, -top=334)
#form_example->addPasswordField(
    'myPassword',
    '***',
    -left=216, -top=322, -width=144, -height=18,
    -font=#myFont
)
#form_example->drawText('Text Field', -font=#myFont, -left=90, -top=368)
#form_example->addTextField(
    'myTextField',
    'Some More Text',
    -left=216, -top=360, -width=144, -height=18,
    -font=#myFont
)
#form_example->addHiddenField('myHiddenField', 'Shh')
#form_example->addSubmitButton(
    'myButton',
    'Submit Form',
    'Submit',
    'http://www.example.com/response.lasso',
    -left=216, -top=400, -width=100, -height=26,
    -font=#myFont
)
#form_example->addResetButton(
    'Reset',
    'Reset Form',
    'Reset',
    -left=365, -top=400, -width=100, -height=26,
    -font=#myFont
)
#form_example->close
```

26.12.3 PDF Table Example

The following example creates a PDF file that contains both text and a table:

```
local(table_example) = pdf_doc(-file='Table_Example.pdf', -size='A4')

local(font1) = pdf_font(-face='Helvetica', -size='24')
local(text)  = pdf_text(
    "This PDF file contains a table. See below.\n\n",
    -leading=15,
    -font=#font1
)
#table_example->add(#text)

local(font2)    = pdf_font(-face='Helvetica', -size='12')
local(cell1)    = pdf_text('Cell One',   -font=#font2)
local(cell2)    = pdf_text('Cell Two',   -font=#font2)
local(cell3)    = pdf_text('Cell Three', -font=#font2)
local(cell4)    = pdf_text('Cell Four',  -font=#font2)
local(my_table) = pdf_table(2, 2,
    -spacing=4, -padding=4, -width=75, -borderWidth=7
)
#my_table->add(#cell1, 0, 0, -borderWidth=4)
#my_table->add(#cell2, 0, 1, -borderWidth=4)
#my_table->add(#cell3, 1, 0, -borderWidth=4)
#my_table->add(#cell4, 1, 1, -borderWidth=4)

#table_example->add(#my_table)
#table_example->close
```

26.12.4 PDF Graphics Example

The following example shows how to create a PDF file that contains drawn graphic objects:

```
local(graphic_example) = pdf_doc(-file='Graphic_Example.pdf', -height=650, -width=550)
local(text) = pdf_text("This PDF file contains lines and circles. See below.\n")
#graphic_example->add(#text)
#graphic_example->line(200, 400, 400, 400)
#graphic_example->line(200, 500, 400, 500)
#graphic_example->line(266, 333, 266, 566)
#graphic_example->line(333, 333, 333, 566)
#graphic_example->line(200, 333, 400, 566)
#graphic_example->circle(233, 366, 20)
#graphic_example->circle(300, 452, 20)
#graphic_example->circle(366, 533, 20)
#graphic_example->line(220, 432, 240, 472)
#graphic_example->line(220, 472, 240, 432)
#graphic_example->line(360, 432, 380, 472)
#graphic_example->line(360, 472, 380, 432)
#graphic_example->line(220, 517, 240, 558)
#graphic_example->line(220, 558, 240, 517)
#graphic_example->close
```

26.12.5 PDF Barcode Example

The following example shows how to create a PDF file that contains text accompanied by a barcode:

```
local(barcode_example) = pdf_doc(
    -file='Barcode_Example.pdf',
    -height=172,
    -width=300
)
local(font1)     = pdf_font(-face='Courier', -size=12)
local(myBarcode) = pdf_barcode(
    -type='CODE39',
    -code='1234567890',
    -generateCheckSum,
    -showCode39StartStop,
    -textSize=6.0
)
#barcode_example->drawText('The Shipping Company\n',
    -font=#font1,
    -left=72,
    -top=90
)
#barcode_example->add(#myBarcode, -left=72, -top=40)
#barcode_example->close
```

26.13 Serving PDF Files

This section describes how PDF files can be served using Lasso Server. This can be done by supplying a download link to the created PDF file, or by using the **pdf_serve** method described below.

26.13.1 Linking to PDF Files

Named PDF files may be linked to in a Lasso page using basic HTML. Once a user clicks on a link to a file with a ".pdf" extension, the client browser should prompt to download the file or launch the file in PDF reader (if configured to do so).

Link to a PDF file

The example below shows how a PDF can be created and written to file, and then linked to from the Lasso page:

```
<?lasso
    local(my_file) = pdf_doc(-file='MyFile.pdf', -size='A4')
    local(my_text) = pdf_text('Hello World')
    #my_file->add(#my_text)
    #my_file->close
?>
<html>
    <body>
        <p>Click on the following link to download MyFile.pdf.</p>
        <p><a href="MyFile.pdf">Click Here</a></p>
    </body>
</html>
```

26.13.2 Serving PDF Files to Client Browsers

PDF files may also be served directly to a client browser using the **pdf_serve** method. This method automatically informs the client web browser that the data being loaded is a PDF file, and outputs the file with the correct file name. If the client web browser is configured to handle PDF files via a reader, then the served PDF file will automatically be opened in the client's configured PDF reader. Otherwise, the client web browser should prompt the user to save the file.

pdf_serve(*doc::pdf_doc*, *-file*, *-type=?*)

> Serves a PDF file to a client browser with a MIME type of ***application/pdf***. Requires the first parameter to specify the pdf_doc object to serve, and the second parameter, **-file**, specifies the name of the file to be output to the browser. An optional **-type** parameter may be used to specify additional MIME types.

Serve a PDF File to a Client Browser

Use the **pdf_serve** method to serve the created PDF file. The file parameter specifies the file name that should be output.

```
local(my_file) = pdf_doc(-file='MyFile.pdf', -size='A4', -noCompress)
#my_file->add(pdf_text('Hello World'))
#my_file->close
pdf_serve(#my_file, -file='MyFile.PDF')
```

Serve a PDF File Without Writing to File

PDF files may be served to the client browser without ever writing them to file on the local server. This is done by creating a pdf_doc object without the **-file** parameter. This allows a PDF file to be created in the system memory, but does not the save the file to a hard drive location. The resulting file can be saved by the end user to a location on the end user's hard drive.

```
local(my_file) = pdf_doc(-size='A4', -noCompress)
#my_file->add(pdf_text('Hello World'))
#my_file->close
pdf_serve(#my_file, -file='MyFile.PDF')
```

XML Documents

Lasso provides a full suite of objects both for constructing new XML documents and parsing existing XML documents. Lasso's implementation follows the DOM Level 2 Core specification[46] as closely as possible. This introduces a series of objects each representing the various components that can be found within an XML document. The Lasso object names match up with the objects specified in the DOM standard with the addition of an **xml_** prefix. Also provided is a simplified method for parsing existing XML data. This method is called **xml** and does not conform to the DOM specification.

Lasso also provides both XPath and XSLT functionality. This functionality is integrated into the XML object model, though it is not considered part of the DOM specification itself.

In cases where elements are accessed by numeric position, Lasso's implementation conforms to the DOM specification's zero-based indexes, as opposed to Lasso's standard one-based positions. This will be noted in all relevant cases within this chapter.

27.1 Creating XML Documents

XML documents are created either from existing XML character data or as empty documents. An empty XML document will initially contain only the root document node which can then have children or attributes added to it. A document created from existing XML character data will be parsed and validated and the resulting document object tree will be created. When attempting to create an XML document from existing data, if the data is not valid, then a failure will be generated during parsing. The current **error_msg** will indicate the encountered error.

New XML documents can be created in one of two ways: the DOM Level 2-conformant **xml_DOMImplementation** type, or the **xml** method. Both have the same abilities, but the **xml** method provides a simplified interface and is compatible with earlier Lasso versions. It's important to note that **xml** is not itself an object, it is merely a method that provides a moderately easier to use interface to XML document creation. Internally, the **xml** method uses the **xml_DOMImplementation** type and therefore provides equivalent functionality to the **xml_DOMImplementation** type.

27.1.1 Using xml

The **xml** method is presented in five variations; two for parsing existing XML documents and three for creating new blank documents.

xml(*text::string*)

xml(*text::bytes*)
>   These first two methods parse existing XML data in either string or raw bytes form. If the document parsing is successful, these methods return the top-level **xml_document** node object.

xml(*nsUri::string*, *rootNodeName::string*, *dtd::xml_documentType=?*)

xml()
>   These subsequent three methods create a new document consisting of only the root **xml_document** node and no children. These methods return the top-level **xml_document** node object. The first methods create the document given

---

[46] http://www.w3.org/TR/DOM-Level-2-Core/

a namespace and a root element name, along with an optional document type node (an **xml_documentType**, created through the **xml_DOMImplementation->createDocumentType** method). The last method takes zero parameters and returns a document with no namespace and the root element name set to "none".

In all cases, the resulting value from the **xml** method will be the root element of the document. This will be an object of type **xml_element**. It's important to note that this is not the **xml_document** object, which differs from the root element node. This behavior is a departure from that of the **xml_DOMImplementation** type which does return the **xml_document** object itself. The owning **xml_document** object can be obtained from any node within that document by calling the **xml_node->ownerDocument** method.

xml Examples

Example of creating an XML document from existing data:

```
local(myDocumentText) = '<a><b>b content</b><c/></a>'
local(myDocumentObj)  = xml(#myDocumentText)
```

Example of creating a blank XML document:

```
local(myDocumentObj) = xml('my_namespace', 'a')
```

27.1.2 Using xml_DOMImplementation

The **xml_DOMImplementation** type provides comparable functionality to the **xml** method, but follows the DOM Level 2 specification. An object of the type **xml_DOMImplementation** is stateless and can be created with zero parameters. Once an **xml_DOMImplementation** object is obtained it can be used to create or parse XML documents as well as create XML document types.

This functionality is presented in the following four methods.

type **xml_DOMImplementation**

xml_DOMImplementation->createDocument(*nsUri::string*, *rootNodeName::string*, *dtd::xml_documentType=?***)**

xml_DOMImplementation->createDocumentType(*qname::string*, *publicid::string*, *systemid::string***)**

xml_DOMImplementation->parseDocument(*text::bytes***)**

In contrast to the **xml** method, when creating or parsing an XML document the **xml_DOMImplementation** object returns the document node. This will be an object of type **xml_document**. It's important to note that this is not the root element node. The root element node can be obtained through the **xml_document->documentElement** method.

xml_DOMImplementation Examples

Example of creating an XML document from existing data:

```
local(myDocumentText) = '<a><b>b content</b><c/></a>'
local(myDocumentObj)  =
    xml_DOMImplementation->parseDocument(
        bytes(#myDocumentText)
    )
```

Example of creating a blank XML document:

```
local(domImpl) = xml_DOMImplementation
local(docType) = #domImpl->createDocumentType(
    'svg:svg',
    '-//W3C//DTD SVG 1.1//EN',
    'http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd'
)
local(myDocumentObj) = #domImpl->createDocument(
    'http://www.w3.org/2000/svg',
    'svg:svg',
    #docType
)
```

The resulting document would have the following format:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE svg:svg PUBLIC "-//W3C//DTD SVG 1.1//EN" "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg xmlns:svg="http://www.w3.org/2000/svg"/>
```

27.1.3 Creating XML Node Objects

While the **xml_DOMImplementation** object is responsible for creating the initial **xml_document** object, the **xml_document** object is the means through which new XML node object types are created, including element, attribute, and text nodes. All XML objects always belong to a particular instance of the **xml_document** type. No XML node objects can be created without an existing document. Nodes can be copied into another existing **xml_document**, but nodes are never shared between documents.

The following methods are use for creating new nodes:

type **xml_document**

xml_document->**createElement**(*tagName::string*) → xml_element

xml_document->**createElementNS**(*nsUri::string*, *qualifiedName::string*) → xml_element
> The first version creates a new element node without a namespace. The second version permits a namespace to be specified.

xml_document->**createAttribute**(*name::string*) → xml_attr

xml_document->**createAttributeNS**(*nsUri::string*, *qualifiedName::string*) → xml_attr
> The first version creates a new attribute without a namespace. The second version permits a namespace to be specified.

xml_document->**createDocumentFragment()** → xml_documentFragment

xml_document->**createTextNode**(*data::string*) → xml_text

xml_document->**createComment**(*data::string*) → xml_comment

xml_document->**createCDATASection**(*data::string*) → xml_cdataSection

xml_document->**createProcessingInstruction**(*target::string*, *data::string*) → xml_processingInstruction

xml_document->**createEntityReference**(*name::string*) → xml_entityReference

xml_document->**importNode**(*importedNode::xml_node*, *deep::boolean*) → xml_node
> Imports a node from another document into the document of the target object and returns the new node. The new node is not yet placed within the current document and so it has no parent. If "false" is given for the second parameter, then the node's children and attributes are not copied. If "true" is given, then all attributes and child nodes are copied into the current document.

The following table lists all the possible objects that may be encountered within or inserted into an XML document.

Table 27.1: XML Object Names

Lasso XML Object Name	XML DOM Level 2 Name	Description
xml_DOMImplementation	DOMImplementation	Creates **xml_document** and **xml_documentType** objects. Can parse existing XML documents or create new empty documents.
xml_node	Node	Base functionality supported by all objects.
xml_document	Document	Represents the entire document and provides access to the document's data.
xml_element	Element	Represents an XML element node.
xml_attr	Attr	Represents an attribute of an XML element node.
xml_characterData	CharacterData	Represents character data within the document. This is the base object type for **xml_text** and **xml_cdataSection** objects.
xml_text	Text	Represents the character data of an **xml_element** or **xml_attr** node.
xml_cdataSection	CDATASection	Represents a CDATA node.
xml_entityReference	EntityReference	Represents an entity reference.
xml_entity	Entity	Represents a parsed or unparsed entity within the document.
xml_processingInstruction	ProcessingInstruction	Represents a processing instruction located within the document.
xml_comment	Comment	Represents the content of an XML comment node.
xml_documentType	DocumentType	Represents the doctype attribute of an XML document.
xml_documentFragment	DocumentFragment	Represents a minimal document object.
xml_notation	Notation	Represents a notation declared in the DTD.
xml_nodeList	NodeList	Represents a list of node objects. Provides random access to the list. This list uses zero-based indexes, in contrast to Lasso's standard one-based positions.
xml_namedNodeMap	NamedNodeMap	Represents a collection of nodes that can be accessed by name.

27.1.4 Inspecting XML Objects

Lasso's XML interface permits all the various pieces of an XML document to be inspected. This includes accessing attributes, node content, node children etc. The methods listed in this section are not meant to be exhaustive, but instead to show the methods most commonly used when working with an XML document.

type **xml_node**

xml_node->**nodeType()** → string
> Returns the name of the type of node. For example, an **xml_element** node would return "ELEMENT_NODE". This is in contrast to the DOM Level 2 specification which returns an integer value.

xml_node->**nodeName()** → string
> Returns the name of the node. This value will depend on the type of the node in question. For **xml_element** nodes, this will be the same value as the tag name. For **xml_attr** nodes, this will be the same as the attribute name.

xml_node->**prefix()**
> Returns the namespace prefix of the node or "null" if it is unspecified.

xml_node->**localName()**
> Returns the local part of the qualified name of the node.

xml_node->**namespaceURI()**
> Returns the namespace URI of the node or "null" if it is unspecified.

xml_node->**nodeValue()**
> Returns the value of the node as a string. This result will vary depending on the node type. For example an attribute node will return the attribute value. A text node will return the text content for the node. Many node types, such as element nodes, will return "null". This value is read/write for nodes that have values, and in such cases can be set with the **xml_node->nodeValue=** method.

xml_node->**parentNode()**
> Returns the parent of the node or "null" if there is no parent. Some, such as attribute nodes and the document node, do not have parents.

xml_node->**ownerDocument()**
> Returns the **xml_document** that is the owner of the target node. In the case of the document node, this will be "null".

type **xml_element**

xml_element->**tagName()** → string
> Returns the name of the element.

xml_element->**getAttribute(***name::string***)** → string
> Returns the value of the specified attribute. Returns an empty string if the attribute does not exist or has no value.

xml_element->**getAttributeNS(***nsUri::string**, *localName::string***)**
> Returns the value of the attribute matching the given namespace and local name. Returns an empty string if the attribute does not exist or has no value.

xml_element->**getAttributeNode(***name::string***)**
> Returns the specified attribute node. Returns "null" if the attribute does not exist.

xml_element->**getAttributeNodeNS(***nsUri::string**, *localName::string***)**
> Returns the attribute node matching the given namespace and local name. Returns "null" if the attribute does not exist.

xml_element->**hasAttribute(***name::string***)** → boolean
> Returns "true" if the specified attribute exists.

xml_element->**hasAttributeNS(***nsUri::string**, *localName::string***)** → boolean
> Returns "true" if the attribute matching the given namespace and local name exists.

type **xml_attr**

xml_attr->**name()** → string
> Returns the name of the attribute.

xml_attr->**ownerElement()**
> Returns the element node that owns the attribute or "null" if the attribute is not in use.

xml_attr->**value()** → string
> Returns the value of the attribute. This value is read/write.

type **xml_nodeList**

xml_nodeList->**length()** → integer
> Returns the number of nodes in the list.

xml_nodeList->**item(***index::integer***)**
> Returns the node indicated by the index. Indexes start at zero and go up to length-1. Returns "null" if the index is invalid.

type **xml_nodeMap**

`xml_nodeMap->`**`length()`** → integer
> Returns the number of nodes in the map.

`xml_nodeMap->`**`getNamedItem`**(*name::string*)
> Returns the node matching the indicated name.

`xml_nodeMap->`**`getNamedItemNS`**(*nsUri::string*, *localName::string*)
> Returns the node matching the indicated namespace URI and local name.

`xml_nodeMap->`**`item`**(*index::integer*)
> Returns the node indicated by the index. Indexes start at zero and go up to length-1. Returns "null" if the index is invalid.

27.1.5 Modifying XML Objects

Various parts of an XML document can be modified. This includes setting node values, adding or removing child nodes, adding or removing attributes, or removing items from node maps.

`xml_node->`**`nodeValue=`**(*value::string*)
> Sets the value of the node to the indicated string. Only the following node types are able to have their values set: **`xml_attr`**, `xml_cdataSection`, `xml_comment`, `xml_processingInstruction`, `xml_text`.

`xml_node->`**`insertBefore`**(*new::xml_node*, *ref::xml_node*) → xml_node
> Inserts the new node into the document immediately before the ref node. Returns the newly inserted node.

`xml_node->`**`replaceChild`**(*new::xml_node*, *ref::xml_node*) → xml_node
> Replaces the ref node in the document with the new node. Returns the new node.

`xml_node->`**`appendChild`**(*new::xml_node*) → xml_node
> Inserts the new node into the document at the end of the target node's child list. Returns the new node.

`xml_node->`**`removeChild`**(*c::xml_node*) → xml_node
> Removes the indicated child node from the document. Returns the removed node.

`xml_node->`**`normalize()`**
> Modifies the document such that no two text nodes are adjacent. All adjacent text nodes are merged into one text node.

`xml_element->`**`setAttribute`**(*name::string*, *value::string*)
> Adds an attribute with the given name and value. If the attribute already exists then the value is set accordingly.

`xml_element->`**`setAttributeNS`**(*uri::string*, *qname::string*, *value::string*)
> Adds an attribute with the given namespace, name, and value. If the attribute already exists its value is set accordingly.

`xml_element->`**`setAttributeNode`**(*node::xml_attr*)
> Adds the new attribute node. If an attribute with the same name already exists it is replaced. To add a namespace-aware attribute, use **`xml_element->setAttributeNodeNS`** instead.

`xml_element->`**`setAttributeNodeNS`**(*node::xml_attr*)
> Adds the new attribute node. If an attribute with the same namespace/name combination already exists it is replaced.

`xml_element->`**`removeAttribute`**(*name::string*)
> Removes the attribute with the indicated name.

`xml_element->`**`removeAttributeNS`**(*uri::string*, *qname::string*)
> Removes the attribute with the given namespace/name combination.

`xml_element->`**`removeAttributeNode`**(*node::xml_attr*) → xml_attr
> Removes the indicated attribute node. Returns the removed node.

Note: Some node maps are read-only and cannot be modified.

---

xml_nodeMap->**setNamedItem**(*node::xml_node*) → xml_node
> Adds the node to the node map based on the "nodeName" value of the node. Replaces any duplicate node within the map. Returns the added node.

xml_nodeMap->**setNamedItemNS**(*node::xml_node*) → xml_node
> Adds the node to the node map based on the namespace/name combination. Replaces any duplicate node within the map. Returns the added node.

xml_nodeMap->**removeNamedItem**(*name::string*)
> Removes the node with the given name from the map. Returns the removed node.

xml_nodeMap->**removeNamedItemNS**(*uri::string*, *qname::string*)
> Removes the node with the given namespace/name combination from the map. Returns the removed node.

27.2 XPath

Lasso's XML API supports the XPath 1.0 specification for any **xml_node** type through the **xml_node->extract** and **xml_node->extractOne** methods. Consult the XPath specification[47] for the specifics of XPath syntax.

27.2.1 Using XPath

XPath is used to address a specific set of nodes within an XML document. For example, child nodes matching a node name pattern can be located, or nodes with specific attributes can be easily found within the document.

xml_node->**extract**(*xpath::string*)
> Executes the XPath in the node and returns all matches as a staticarray.

xml_node->**extract**(*xpath::string*, *namespaces::staticarray*)
> Executes the XPath in the node and returns all matches as a staticarray. This method should be used for XML documents that use namespaces. The second parameter is a staticarray containing the relevant namespace prefixes and URI pairs that are used within the XPath expression. Note that the namespace prefixes used in the XPath expression do not have to match those used within the document itself.

xml_node->**extractOne**(*xpath::string*)
> Executes the XPath in the node and returns the first matching node or "null" if there are no matches.

xml_node->**extractOne**(*xpath::string*, *namespaces::staticarray*)
> Executes the XPath in the node and returns the first matching node or "null" if there are no matches. This method should be used for XML documents that use namespaces. The second parameter is a staticarray containing the relevant namespace prefixes and URI pairs that are used within the XPath expression. Note that the namespace prefixes used in the XPath expression do not have to match those used within the document itself.

XPath Examples

Extract all child elements of the a node:

```
local(doc) = xml(
    '<a>
        <b at="val"/>
        <c at="val2">C Content</c>
    </a>')
#doc->extract('//a/*')

// => staticarray(<b at="val"/>, <c at="val2">C Content</c>)
```

---

[47] http://www.w3.org/TR/xpath/

Using namespaces, extract all child elements of the a node:

```
local(doc) = xml(
    '<a xmlns="my_uri">
        <b at="val"/>
        <c at="val2">C Content</c>
    </a>')
#doc->extract('//n:a/*', (: 'n'='my_uri'))

// => staticarray(<b at="val"/>, <c at="val2">C Content</c>)
```

Extract the first child element of the a node:

```
local(doc) = xml(
    '<a>
        <b at="val"/>
        <c at="val2">C Content</c>
    </a>')
#doc->extractOne('//a/*')

// => <b at="val"/>
```

Extract the **"at"** attribute from the second child element of the a node:

```
local(doc) = xml(
    '<a xmlns="my_uri">
        <b at="val"/>
        <c at="val2">C Content</c>
    </a>')
#doc->extractOne('//n:a/*[2]/@at', (: 'n'='my_uri'))

// => at="val2"
```

27.3 XSLT

Lasso's XML API supports XSL Transformations (XSLT) 1.0. For the specifics of XSLT, consult the XSLT specification[48].

XSLT support is provided on any **xml_node** type through the **transform** method. This method accepts an XSLT template as a string as well as a list of all variables to be made available during the transformation. The transformation is performed and a new XML document is returned.

xml_node->transform(_sheet::string, variables::staticarray_**)** → xml_document
    Performs an XSLT transformation on the document and returns the resulting newly produced document.

---

[48] http://www.w3.org/TR/xslt/

Logging

Lasso Server has a built-in error logging system that allows warning messages to be logged at several different levels. Each log level can be routed to one or more destinations, allowing for a great deal of flexibility in handling.

The built-in log levels include:

Critical
> Critical errors that affect the operation of Lasso Server. Critical errors are logged to all destinations by default. Typically, the server or site administrator will need to fix whatever is causing the critical error.

Warning
> Warnings are informative messages about possible problems with the functioning of Lasso Server. Warnings do not always require action by the server or site administrator. Warnings are logged only to the console by default.

Detail
> Detailed messages about the normal functioning of Lasso Server. Includes status messages from the email queue and event scheduler, etc. Detail messages are logged only to the console by default.

Deprecated
> Flags any use of deprecated functionality in Lasso code. Deprecated methods are supported in this version of Lasso, but may not be supported in a future version. Any deprecated functionality should be updated to new, preferred syntax for best compatibility with future versions of Lasso. Deprecated messages are logged only to the console by default.

The destinations that the log levels can be routed to include:

Console
> The Lasso Server instance's console, which is viewable from the Instance Manager. It is stored in a file named `lasso.out.txt` in the instance's **LASSO9_HOME** directory and has a max file size of 10 MB by default.

File
> The `lasso_logbook.txt` file, located in the instance's **LASSO9_HOME** directory. This file is also capped at 10 MB by default.

Database
> The "logbook" table in the "lasso_logbook" SQLite database, viewable via the "Log Book" section of Lasso Server Admin (*http://example.com/lasso9/admin/logbook*).

The routing of Lasso's internal log levels can be modified using the "Log Book" section of the Lasso Server Admin interface (*http://example.com/lasso9/admin/logbook*). For details on how to change the log level routing programmatically, see the section *Log Routing* later in this chapter.

28.1 Logging Methods

The **log_critical**, **log_warning**, **log_detail**, and **log_deprecated** methods are used to log custom data to the Lasso internal error logs with a defined Lasso error level of "Critical", "Warning", "Detail", or "Deprecated", respectively.

log_critical(…)
> Logs to Lasso's internal error logs with an error level assignment of "Critical". Requires the text to be logged as a parameter. Logging options for this error level are set in Lasso Server Admin.

log_warning(…)

> Logs to Lasso's internal error logs with an error level assignment of "Warning". Requires the text to be logged as a parameter. Logging options for this error level are set in Lasso Server Admin.

log_detail(…)

> Logs to Lasso's internal error logs with an error level assignment of "Detail". Requires the text to be logged as a parameter. Logging options for this error level are set in Lasso Server Admin.

log_deprecated(…)

> Logs to Lasso's internal error logs with an error level assignment of "Deprecated". Requires the text to be logged as a parameter. Logging options for this error level are set in Lasso Server Admin.

log_always(…)

> Logs to Lasso's console. This error level cannot be routed, and is always sent to Lasso's console.

28.1.1 Create a Log Message

The following example will create a log statement at the level of "Warning" if Lasso throws a "Divide By Zero" error. The displayed result is the log message that gets sent to the console; note that it contains a timestamp in brackets:

```
handle(error_code == error_code_divideByZero) => {
    log_warning('A mathematical error occurred while processing this page')
}

// => [2013-03-23 16:59:41] A mathematical error occurred while processing this page
```

28.2 Logging to Files

In addition to using the built-in log level routing system, it is sometimes desirable to create a separate log file specific to a custom solution. The **log** method can be used to write text messages out to a log file.

log(*path*)

> When executed, the results of the auto-collection from the **log** method's capture block is appended to a specified text file. The **log** method can write to any text file that is accessible from Lasso. The capture block must be an auto-collect block as the collected data from the capture block will be included in the appended data. If you don't use an auto-collect block then no data will be appended to the log file.
>
> The following example outputs a single line containing the date and time with a return at the end to the file specified. The methods are shown first with a Windows path, then with an OS X or Linux path.

```
log('C://Logs/LassoLog.txt') => {^
    date->format('%Q %T')
    '\r\n'
^}

log('//Logs/LassoLog.txt') => {^
    date->format('%Q %T')
    '\n'
^}
```

> The path to the directory where the log will be stored should be specified according to the same rules as those for the **file** methods. See the section *Paths* in the *File System* chapter for full details about relative, absolute, and fully qualified paths on OS X, Linux, and Windows.

28.2.1 Log Site Visits to a File

The following code will log the current date and time, the visitor's IP address, the name of the server, the page they were loading, and the GET and POST parameters that were specified:

```
log('//tmp/foo.bar') => {^
    date->format('%Q %T') +
    ' ' + web_request->remoteAddr +
    ' ' + (web_request->isHttps ? 'https://' | 'http://') +
    web_request->httpHost +
    web_request->requestUri +
    ' ' + web_request->postParams + '\n'
^}
```

28.2.2 Automatically Roll Log Files by Date

Include a date component in the name of the log file. Since the date component will change every day, a new log file will be created daily the first time an item is logged. The following example logs to a file named with the current date, e.g. "2013-05-31.txt":

```
local(cur_date) = date->format('%Q')
log('//Logs/' + #cur_date + '.txt') => {^
    date->format('%Q %T')
^}
```

28.3 Log Routing

Log preferences can be viewed or changed in the "Log Book" section of Lasso Server Admin. Use of the **log_setDestination** method is only necessary to change the log settings programmatically.

log_setDestination(_level::integer, dest1::integer=?, dest2::integer=?, dest3::integer=?_**)**
> The first parameter specifies a log message level. Subsequent parameters specify the destination to which that level of messages should be logged. Both the log level and any destinations are specified with integer values. It is preferred that you use the convenience methods described below as parameters rather than using literal integer values.

log_level_critical()
> Returns the integer value for specifying the "Critical" message level in the **log_setDestination** method. Using this method will help future-proof your code.

log_level_warning()
> Returns the integer value for specifying the "Warning" message level in the **log_setDestination** method. Using this method will help future-proof your code.

log_level_detail()
> Returns the integer value for specifying the "Detail" message level in the **log_setDestination** method. Using this method will help future-proof your code.

log_level_deprecated()
> Returns the integer value for specifying the "Deprecated" message level in the **log_setDestination** method. Using this method will help future-proof your code.

log_destination_console()
> Returns the integer value for specifying the "Console" destination in the **log_setDestination** method. Using this method will help future-proof your code.

log_destination_file()

> Returns the integer value for specifying the "File" destination in the **log_setDestination** method. Using this method will help future-proof your code.

log_destination_database()

> Returns the integer value for specifying the "Database" destination in the **log_setDestination** method. Using this method will help future-proof your code.

28.3.1 Change Logging Preferences

Use the **log_setDestination** method to change the destination of a given log message level. In the following example, detail messages are sent to the console and the errors table of the instance's database:

```
log_setDestination(
    log_level_detail,
    log_destination_database,
    log_destination_console
)
```

28.3.2 Reset Logging Preferences

The following four commands reset the log preferences to their default values. Critical errors are sent to all three destinations; warnings, detail, and deprecation messages are sent only to the console.

```
log_setDestination(
    log_level_critical,
    log_destination_console,
    log_destination_database,
    log_destination_file
)
log_setDestination(log_level_warning,    log_destination_console)
log_setDestination(log_level_detail,     log_destination_console)
log_setDestination(log_level_deprecated, log_destination_console)
```

Shell Commands with sys_process

Lasso provides the ability to execute local processes or shell commands through the **sys_process** type. This type allows local processes to be launched with an array of parameters and shell variables. Some processes will execute and return a result immediately. Other processes can be left open for interactive read/write operations. The **sys_process** type enables Lasso users to do tasks such as execute AppleScripts, print PDF files, and filter data through external applications.

The **sys_process** type works across all three platforms that Lasso supports. The UNIX underpinnings of OS X and Linux mean that those two operating systems can run many of the same commands and shell scripts. Windows presents a very different environment including DOS commands and batch files.

For more information on writing shell scripts with Lasso, see the *Command-Line Tools* chapter.

29.1 Using sys_process

type **sys_process**

sys_process()

sys_process(*cmd::string, args=?, env=?, user::string=?***)**
> The **sys_process** type allows a developer to create a new process on the machine and both read from and write data to it. The process is usually closed after the **sys_process** object is destroyed, but can optionally be left running. The **sys_process** type shares many of the same member methods and conventions as the **file** type.
>
> There are two constructor methods for creating **sys_process** objects: the first allows for an empty object with no information being passed to it. The second takes the same parameters as the **sys_process->open** method and calls that method, thereby immediately running the command passed to it.

sys_process->open(*command::string, arguments::staticarray=?, environment::staticarray=?, user::string=?***)**
> Opens a new process. The command string should consist of the full path to the executable unless it is just a built-in command that does not have a path; in that case just pass the name of the command. An optional staticarray of arguments can be passed as the second parameter. Any arguments are converted to strings and passed to the new process. An optional staticarray of environment strings may be specified as the third parameter, and these too will be passed to the new process. By default, the new process is run as the current user. The fourth parameter allows for optionally specifying a user to run the new process under. This option only works if the current user is the superuser.

sys_process->wait() → integer
> Calling this member method causes execution of your code to pause until the new process you have opened with **sys_process** finishes its execution. It returns the exit code of the command you ran. If you have not yet opened up a new process, it will return "-1".

sys_process->read(*count::integer=?, -timeout=?***)** → bytes
> Reads the specified number of bytes from the process's standard out (STDOUT). Returns a bytes object. The number of bytes of data actually returned from this method may be less than the specified number depending on the number of

bytes that are actually available to read. Calling this method without a byte count will read 1024 bytes. A timeout value may also be specified which is the number of milliseconds to wait for the number of bytes being requested. The default value for this is "0" which means that it will just read what is currently available.

sys_process->**readError**(*count::integer=?, -timeout=?*) → bytes
> Reads the specified number of bytes from the process's standard error (STDERR) output. Returns a bytes object. Calling this method without a byte count will read 1024 bytes. A timeout value may also be specified which is the number of milliseconds to wait for the number of bytes being requested. The default value for this is "0" which means that it will just read what is currently available.

sys_process->**readString**(*count::integer=?, -timeout=?*) → string
> This method is identical to **sys_process->read** but returns a string object instead of a bytes object.

sys_process->**write**(*data::string*)

sys_process->**write**(*data::bytes*)
> Writes the specified data to the new process's standard in (STDIN). If the data is a string, the current encoding is used to convert the data before being sent. If the data is a bytes object, the data is sent unaltered.

sys_process->**setEncoding**(*encoding::string*)
> Sets the encoding for the instance. The encoding controls how string data is written via **sys_process->write** and how string data is returned via **sys_process->readString**. By default, UTF-8 is used.

sys_process->**isOpen()** → boolean
> Returns "true" as long as the process is running. After the process is terminated, it will return "false".

sys_process->**detach()**
> Detaches the **sys_process** object from the process. This will prevent the process from terminating when the **sys_process** object is destroyed.

sys_process->**close()**
> Closes the connection to the process. This will cause the process to terminate unless it has previously been detached from the **sys_process** object by calling **sys_process->detach**.

sys_process->**closeWrite()**
> Closes the "write" portion of the connection to the process. This results in the process's standard in (STDIN) being closed.

sys_process->**exitCode()**
> This method is synonymous with **sys_process->wait** except that it does not return a value if no process has been opened.

sys_process->**testExitCode()**
> Returns the exit code of the process if it has terminated, otherwise it returns "void".

Important: If you wish to run a command that you expect to run briefly and you want to inspect its output after it has run, then don't forget to call either **sys_process->wait** or **sys_process->exitCode** before calling any of the **sys_process->read ...** methods. If you don't wait, your code will more than likely call the read method before the new process fully starts up, and you may miss anything written to STDOUT or STDERR. If the process may take a long time, or output a lot of data, you may want to use either **sys_process->isOpen** or **sys_process->testExitCode** as test conditions in a while loop that does the reading. (See examples below.)

29.2 OS X and Linux Examples

This section includes several examples of using **sys_process** on OS X. Except for the AppleScript example, all of these examples should also work on Linux installations.

29.2.1 Echo

This example uses the **/bin/echo** command to simply echo the input back to STDOUT, which is then read by Lasso:

```
local(proc) = sys_process('/bin/echo', (: 'Hello World!'))
local(_)    = #proc->wait
#proc->read->encodeHtml
#proc->close


// => Hello World!
```

29.2.2 List

This example uses the **/bin/ls** command to list the contents of a directory:

```
local(proc) = sys_process('/bin/ls', (: '/' + sys_homePath))
fail_if(#proc->exitCode != 0, 'Unknown error')
#proc->readString->encodeHtml(true, false)
#proc->close


// =>
// LassoApps
// LassoModules
// LassoStartup
// SQLiteDBs
// lasso.err.txt
// lasso.fastcgi.sock
// lasso.out.txt
```

29.2.3 Create File

This example uses the **/usr/bin/tee** command to create a file "test.txt" in the site folder. The code does not generate any output, it just creates the file.

```
local(proc) = sys_process
handle => {
    #proc->close
}
#proc->open('/usr/bin/tee', (: './test.txt'))
#proc->write('This is a test\n')
#proc->write('This is a test\n')
#proc->close
```

29.2.4 Print

This example uses the **/usr/bin/lpr** command to print some text on the default printer. The result in this case is a page that contains the phrase "This is a test" at the top. This style of printing can be used to output text data using the default font for the printer. The **lpr** command can also be used with some common file formats such as PDF files.

```
local(proc) = sys_process('/usr/bin/lpr')
#proc->write('This is a test')
#proc->write(bytes->import8Bits(4)&)
#proc->closeWrite
#proc->close
```

29.2.5 AppleScript

This example uses the **/usr/bin/osascript** command to run a simple AppleScript. AppleScript is a full scripting language that provides access to the system and running applications in OS X. The script shown returns the current date and time:

```
local(proc) = sys_process('/usr/bin/osascript', (: '-'))
#proc->write('return current date')
local(_) = #proc->closeWrite&wait
#proc->readString->encodeHtml
#proc->close

// => Tuesday, March 21, 2006 11:52:34 AM
```

29.2.6 Web Request

This example uses the **/usr/bin/curl** command to fetch a web page and return the results. The **curl** type or **include_url** method can be used for the same purpose. You'll notice that we don't just wait and then do a read; this is to show how to deal with not knowing how large of a response there will be from STDOUT. Only the first part of the output is shown.

```
local(proc) = sys_process('/usr/bin/curl', (: 'http://www.apple.com/'))
local(data)
while(#proc->isOpen or #data := #proc->readString) => {^
    #data->asString->encodeHtml
^}
#proc->close

// =>
// <!DOCTYPE html>
// <html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en-US" lang="en-US">
// <head>
//    <meta charset="utf-8" />
//    <meta name="Author" content="Apple Inc." />
// ... rest of response ...
```

29.3 Windows Examples

This section includes several examples of using **sys_process** on Windows. Each of the examples uses the command-line processor **CMD** with the option "/C" to interpret an individual command.

29.3.1 Echo

This example uses the **CMD** processor with an **ECHO** command to simply echo the input back to Lasso:

```
local(proc) = sys_process('cmd', (: '/C ECHO Hello World!'))
local(_)    = #proc->wait
#proc->readString->encodeHtml
#proc->close

// => Hello World!
```

29.3.2 List

This example uses the **CMD** processor with a **DIR** command to list the contents of a directory. The "/B" option instructs Windows to only list the contents of the directory without extraneous header and footer information.

```
local(proc) = sys_process('cmd', (: '/C DIR /B .'))
local(_)    = #proc->wait
#proc->readString->encodeHtml
#proc->close

// =>
// JavaLibraries
// JDBCDrivers
// LassoApps
// LassoModules
// LassoStartup
// SQLiteDBs
// JDBCLog.txt
// lasso.err.txt
// lasso.out.txt
```

29.3.3 Help

This example uses the **CMD** processor with a **HELP** command to show the help information for a command. The start of the help file for **CMD** itself is shown. Running **HELP** without a parameter will return a list of all the built-in commands supported by the command processor.

```
local(proc) = sys_process('cmd', (: '/C HELP cmd'))
local(_)    = #proc->wait
#proc->readString->encodeHtml
#proc->close

// =>
// Starts a new instance of the Windows XP command interpreter
// CMD [/A | /U] [/Q] [/D] [/E:ON | /E:OFF] [/F:ON | /F:OFF] [/V:ON | /V:OFF] [[/S] [/C | /K] string]
// /C Carries out the command specified by string and then terminates
// /K Carries out the command specified by string but remains
// /Q Turns echo off
// /A Causes the output of internal commands to a pipe or file to be ANSI
// /U Causes the output of internal commands to a pipe or file to be Unicode
```

29.3.4 Multiple Commands

This example uses the **CMD** processor interactively to run several commands. The processor is started with a parameter of "/Q" which suppresses the echoing of commands back to the output. The result is exactly the same as what would be provided if these commands were entered directly into the command line shell. In order to process the results, it would be necessary to strip off the header and the directory prefix from each line.

```
local(proc) = sys_process('cmd', (: '/Q'))
#proc->write('ECHO Line One\r\n')
#proc->write('ECHO Line Two\r\n')
local(_) = #proc->wait
#proc->read->encodeHtml
#proc->close
```

```
// =>
// Microsoft Windows XP [Version 5.1.2600]
// (C) Copyright 1985-2001 Microsoft Corp.
// C:\Program Files\LassoSoft\Lasso Instance Manager\home>Line One
// C:\Program Files\LassoSoft\Lasso Instance Manager\home>Line Two
```

29.3.5 Batch File

This example uses the **CMD** processor to process a batch file. The contents of batch file "batch.bat" is shown below. The file is assumed to be located in the folder for the current site in the Lasso Server application folder.

```
@ECHO OFF
CLS
ECHO This file demonstrates how to use a batch file.
```

The batch file is executed by calling its name as a command. The results of the batch file are then output. Using a batch file makes executing a sequence of commands easy since all the code can be perfected using local testing before it is run through Lasso.

```
local(proc) = sys_process('cmd', (: '/C batch.bat'))
local(_)    = #proc->wait
#proc->readString->encodeHtml
#proc->close

// => This file demonstrates how to use a batch file.
```

Part V

Application Development

Web Requests and Responses

Lasso Server receives requests from whichever HTTP server it is connected to. Each request consists of the headers and body data as sent by the requesting user agent, as well as data from the HTTP server such as the local web server root directory and other metadata. The request data is parsed and made available for the code that is run to handle the request. Handling a request entails creating the resulting headers and body data for the reply. This data is sent to the web server, which is then sent to the connected user agent. The request is complete after the response data is sent.

The code that is chosen to handle a request is based on the path in the `PATH_INFO` or, if that is not present, the `SCRIPT_NAME` variable, as sent from the web server. That value is appended to the value of the `DOCUMENT_ROOT` or the **LAS-SOSERVER_DOCUMENT_ROOT** variable and the resulting file path is used as the response. That response may be a script file located on the local file system or it may address a component of a LassoApp. Either way, the file is compiled if necessary and executed.

If the indicated file is not present or an unhandled failure occurs while processing the request, then Lasso will look for a file named `error.lasso` at the original file's directory path. If an `error.lasso` file is not found, then Lasso will look up one directory level for the error file, and so on, until the web file root directory is reached or the error file is found. If no error file is found to handle the situation, then a standard error message and stack trace is printed.

Finally, Lasso provides a means for running code before or after a request. This enables interception of the standard request processing flow at either point. This can be useful when using virtual URLs and serving dynamic, database-driven content or when rewriting outgoing response data.

30.1 Web Requests

Lasso Server makes web request data available through a **web_request** object. An instance of this object is created for each request before processing begins. The request handling code can obtain its request object instance by calling the **web_request** method.

The **web_request** object has the following purposes:

- Making available all variables sent by the web server
- Including all client header information
- Making available all data sent by the web client
- Including tokenized GET arguments
- Including processed POST body data

A **web_request** object will process the incoming data to make access to the various components of a web request more convenient. For example, all HTTP cookies are found and separated made available through the **web_request->cookies** or **web_request->cookie(name)** methods. Standard HTTP headers are made available through accessors such as `web_request->requestURI` or `web_request->httpHost`.

The incoming GET arguments are tokenized and can be retrieved by name or iterated over in their entirety. The request's POST body is processed depending on the incoming *Content-Type*. Both *multipart/form-data* and

application/x-www-form-urlencoded content types are automatically handled. This includes the processing of file uploads, the results of which are made available through the **web_request->fileUploads** method, described below.

30.1.1 Reading Request Headers

The incoming HTTP request headers are pre-processed by the web server and then further processed by Lasso. All header names are normalized to uppercase by the web server and prepended with **HTTP_** and all dashes (**-**) replaced with underscores (**_**). Once received by Lasso, any leading **HTTP_** prepended by the web server to each variable is stripped. All underscores (**_**) are then converted to dashes (**-**).

The **web_request** object makes header data available through the following methods. All header names and values are treated as strings.

type **web_request**

web_request->**headers()** → trait_forEach

web_request->**header(***name::string***)**

web_request->**rawHeader(***name::string***)**
> The **headers** method returns all of the headers as an object that can be iterated or used in a query expression. Each header element is presented as a pair object containing the header name and value as the pair's first and second elements, respectively. The **header** method returns the first header pair, which matches the name parameter. It returns "void" if the header is not found. The **rawHeader** method works the same, but fetches the raw unnormalized header name/value as sent by the web server.

The next set of methods is presented in a table matching the method name to its corresponding raw web request variable name. For headers that return a string value, an empty string is returned if the header has no value or is not present. A zero or "false" is returned for other non-existent value types.

Table 30.1: Web Request Variable Methods

Web Request Method	Web Request Variable	Return Type
web_request->contentLength	CONTENT_LENGTH	integer
web_request->contentType	CONTENT_TYPE	string
web_request->gatewayInterface	GATEWAY_INTERFACE	string
web_request->httpAccept	HTTP_ACCEPT	string
web_request->httpAcceptEncoding	HTTP_ACCEPT_ENCODING	string
web_request->httpAcceptLanguage	HTTP_ACCEPT_LANGUAGE	string
web_request->httpCacheControl	HTTP_CACHE_CONTROL	string
web_request->httpConnection	HTTP_CONNECTION	string
web_request->httpCookie	HTTP_COOKIE	string
web_request->httpHost	HTTP_HOST	string
web_request->httpReferer	HTTP_REFERER	string
web_request->httpReferrer	HTTP_REFERER	string
web_request->httpUserAgent	HTTP_USER_AGENT	string
web_request->isHttps	HTTPS	boolean
web_request->path	PATH	string
web_request->pathInfo	SCRIPT_NAME	string
web_request->pathTranslated	PATH_TRANSLATED	string
web_request->remoteAddr	REMOTE_ADDR	string
web_request->remotePort	REMOTE_PORT	integer

Table 30.1 – continued from previous page

Web Request Method	Web Request Variable	Return Type
web_request->requestMethod	REQUEST_METHOD	string
web_request->requestURI	REQUEST_URI	string
web_request->scriptFilename	SCRIPT_FILENAME	string
web_request->scriptName	SCRIPT_NAME	string
web_request->scriptURI	SCRIPT_URI	string
web_request->scriptURL	SCRIPT_URL	string
web_request->serverAddr	SERVER_ADDR	string
web_request->serverAdmin	SERVER_ADMIN	string
web_request->serverName	SERVER_NAME	string
web_request->serverPort	SERVER_PORT	integer
web_request->serverProtocol	SERVER_PROTOCOL	string
web_request->serverSignature	SERVER_SIGNATURE	string
web_request->serverSoftware	SERVER_SOFTWARE	string

30.1.2 Reading GET and POST Arguments

Lasso automatically tokenizes GET arguments and processes the POST body into a series of name/value pairs according to the sent content type. These two sets of pairs can be retrieved separately or treated as a single series of elements. File uploads are not included in the POST arguments, but are made available through the **web_request->fileUploads** method.

The value for any GET or POST argument is always a bytes object. The name is always a string.

web_request->**param**(*name::string*, *joiner=?*)

web_request->**params()**

web_request->**queryParam**(*name::string*)

web_request->**queryParams()**

web_request->**postParam**(*name::string*)

web_request->**postParams()**
>   This set of methods refers to the GET arguments as the "query" params and any POST arguments as the "post" params. Both sets together are just the "params". For the methods that accept a name parameter, they return the first matching argument's string value. If no argument matches, then a "void" value is returned.

>   The **params** method presents both argument sources as a single queriable **tie** object with the POST arguments occurring first. The **param(name::string,joiner)** method presents an interface for accessing arguments that occur more than once. The **joiner** parameter is used to determine the result of the method. If "void" is passed, then the resulting argument values are returned in a staticarray. If a string value is passed, then the argument values are joined with that string in between each value. The result of passing any other object type will depend on the behavior of its **+** operator.

>   The methods that accept zero parameters return all of the GET, POST, or both argument pairs as an object which may be iterated over or used in a query expression.

web_request->**queryString()**

web_request->**postString()**
>   These methods return the respective arguments in a format similar to how they were received. In the case of **queryString** the GET arguments are returned verbatim. The POST string is created by concatenating each POST argument together with "&" in between each name/value, each of which are separated by "=". This will vary from the exact given POST only in the case of *multipart/form-data* input.

30.1.3 Reading Cookies

Cookie values are sent as HTTP header fields. As such, they can be read and parsed from the various header-related **web_request** methods. The **web_request** object provides methods to directly access the pre-parsed cookie data.

web_request->**cookie**(*named::string*)

web_request->**cookies()** → trait_forEach
> The first method searches for the named cookie and returns its value if found. If the cookie is not found then "void" is returned. The second method returns all the cookies as an object, which can be iterated over or used in a query expression. The cookie elements are presented as pair objects containing the cookie names and values as the pairs' first and second members.

30.1.4 Uploading Files

Lasso can process and manage files uploaded to a web server by visitors to your website. To allow visitors to upload files to your web server, use an HTML **<form>** tag along with an **<input>** tag for each file being uploaded. The form tag must have an "enctype" attribute of *multipart/form-data*, and the "input" tags for file uploads need to have a "type" attribute of "file". The following HTML code could be used to upload a single file to the server:

```
<form action="upload_file.lasso" method="post" enctype="multipart/form-data">
    <fieldset>
        <legend>Upload a Photo</legend>
        <input type="file" name="photo">
        <input type="submit" name="submit" value="Upload">
    </fieldset>
</form>
```

The "file" input tells the browser to show controls for selecting a file to be uploaded to the web server. Once a user selects the file and then clicks "Upload", the form will upload the data to the server and the files can be processed by "upload_file.lasso", the Lasso file specified as the action of the form submission.

Uploaded files processed by Lasso are initially stored in a temporary location. If you do nothing with them, they will be deleted. If you wish to keep them, you should move them to a different directory. To inspect and process these uploaded files use the **web_request->fileUploads** method.

web_request->**fileUploads()**
> This method returns an array, each element of which holds information about an uploaded file. The size of this array will be equal to the number of files uploaded. Each element of the array is a staticarray of pairs that houses the following information about the files:

> fieldname
>> The name of the "file" input type. (In our example, "photo")

> contenttype
>> The MIME content type of the file.

> filename
>> The original name of the uploaded file.

> tmpfilename
>> The path to which the file was temporarily uploaded.

> filesize
>> The size of the file in bytes.

The following example code will loop through all uploaded files and display this information:

```
<dl>
[with file_info in web_request->fileUploads do {^]
    <dt>[#file_info->find('filename')->first->second]</dt>
    <dd>
        <ul>
            <li>[#file_info->find('tmpfilename')->first->second]</li>
            <li>[#file_info->find('contenttype')->first->second]</li>
            <li>[#file_info->find('filesize')->first->second]</li>
            <li>[#file_info->find('fieldname')->first->second]</li>
        </ul>
    </dd>
[^}]
</dl>
```

The preceding example produces HTML like this:

```
<dl>
    <dt>MyAvatar.jpg</dt>
    <dd>
        <ul>
            <li>//tmp/lassoqM9SFY37921967.uld</li>
            <li>image/jpeg</li>
            <li>851191</li>
            <li>photo</li>
        </ul>
    </dd>
</dl>
```

The following example will move uploaded files out of their temporary location and into the "/assets/img/avatars/" directory in the web root, overwriting any existing files with the same name:

```
local(path) = '/assets/img/avatars/'
with upload in web_request->fileUploads
do file(#upload->find('tmpfilename')->first->second)
    ->moveTo(#path + #upload->find('filename')->first->second, true)
```

Monitoring Uploads

If you expect the uploads to take a lot of time, either due to uploading many files or a few large ones, you may want to provide feedback to your visitors that the browser and server are working on the uploads. Lasso comes with a method that will allow you to do just that.

To track files, you first need an input named "_lasso_upload_tracker_id" with a unique value in your form. You can use `lasso_uniqueid` to generate a UUID which is essentially guaranteed to be unique each time you call it. With that in place, while the thread that processes the form submission is working on uploading the files, you can check the status of that process in another thread. This is done by passing the unique ID to the `upload_tracker->check` method of the `upload_tracker` thread object. That method returns a staticarray whose first element is the amount of data uploaded, the second is the total size of all the files being uploaded, and the third is the name of the current file being uploaded.

The following basic example has a form set up properly in "index.lasso". When the submit button is pressed it opens another window to display "progress.lasso" before submitting the form. This page calls `upload_tracker->check` with the unique ID that gets passed to it. It also uses `<meta http-equiv="refresh" content="1">` to refresh itself every second. The result is that we get a progress bar that is updated every second.

index.lasso

```
<!DOCTYPE html>
<html>
<head>
    <title>Upload A Photo</title>
    <script type="text/javascript">
        function trackProgress(id) {
            window.open(
                "/progress.lasso?id=" + id,
                null,
                "height=100,width=400,location=no,menubar=no,resizable=yes,scrollbars=yes,title=yes"
            );
        }
    </script>
</head>
<body>
    [local(id) = lasso_uniqueid]
    <form action="upload_file.lasso" method="post" enctype="multipart/form-data">
        <input type="hidden"
            name="_lasso_upload_tracker_id" value="[#id]">
        <fieldset>
            <legend>Upload a Photo</legend>
            <input type="file" name="photo">
            <input type="submit"
                value="Upload"
                onclick="trackProgress('[#id->encodeUrl]')">
        </fieldset>
    </form>
</body>
</html>
```

progress.lasso

```
[local(info) = upload_tracker->check(web_request->param('id'))]
<!DOCTYPE html>
<html>
<head>
[if(#info->first > 0 and #info->first != #info->second)]
    <meta http-equiv="refresh" content="1">
[/if]
</head>
<body>
[if(#info->first > 0 and #info->second > 0)]
[#info->last]
<div style="background-color: white;border: 1px solid black;width:380px;height: 20px;">
    <div style="background-color: black;height: 20px;width: [
        380 * (decimal(#info->first) / decimal(#info->second))
    ]px;"></div>
</div>
[/if]
</body>
</html>
```

30.2 Web Responses

Sending a response to a web request is a simple as having "The Words" in the targeted "*.lasso" text file. Files requested through a web request are assumed to begin as plain text. Lasso code can be inserted into the file between any of the following delimiters: `[ ... ]`, `<?lasso ... ?>`, or `<?= ... ?>`.

Because supporting the `[ ... ]` delimiters can be problematic for embedding with other technologies (i.e., JavaScript and CSS), they can be disabled for the remainder of the file by having the literal `[no_square_brackets]` as the first line.

Any code between the delimiters will have the results of the expressions within its body converted to string objects and included in the response output string. Code between auto-collecting captures is included as well. For example, values produced by code between `inline(...) ... /inline` or `inline(...) => {^ ... ^}` would be included in the output. Such code is free to call any methods or types to formulate the response data.

The request is completed when the initial code has run to the end, when the **abort** method is called, or when an unhandled failure occurs. Outgoing data is buffered for as long as possible, but can be forced out at any point using the `web_response->sendChunk` method. Calling **abort** (either `web_response->abort` or the unbound method; both have the same behavior) will complete the request by halting all processing and sending the existing response data as-is.

The `web_response` object automatically routes requests for LassoApps. Request paths that begin with "/lasso9/" are reserved for LassoApp usage and will be routed there. Lasso Server ignores physical file paths beginning with "/lasso9/" during the processing of a web request.

30.2.1 Including Files

It is often useful to split up large template files into smaller reusable components. For example, a header or footer could be split out and reused on all pages. The `web_response` object provides a variety of methods for including other source code files. Files included in this way behave just as a file accessed directly would. That is, they begin executing as plain text and any Lasso code must be included between delimiters.

The path to an include file can be full or relative. Complete paths from the file system root are accepted as well. (See the *File System* chapter for more details on how file paths are treated in Lasso.) Components of LassoApps can be included as well by beginning the path with "/lasso9/", then the app name and then the path to the component.

Any of the following methods can be used to include file content.

type **web_response**

`web_response->`**include**(*path::string*)

`web_response->`**includeOnce**(*path::string*)

`web_response->`**includeLibrary**(*path::string*)

`web_response->`**includeLibraryOnce**(*path::string*)
>   These methods locate and run the file indicated by the path. The **includeLibrary** and **includeLibraryOnce** member methods run the file but do not insert the result into the response. The **includeOnce** and **includeLibraryOnce** member methods will only include the file if it has not already been included during the course of that request.
>
>   These methods will fail if the indicated file does not exist.

`web_response->`**includeBytes**(*path::string*) → bytes
>   Locates the file and includes the raw file data as bytes. The method will fail if the file does not exist.

`web_response->`**includes**() → trait_forEach
>   Lasso keeps track of web files that are being executed. As execution of a file begins, the file's name is pushed onto an internally kept stack. As a file's execution ends, that name is popped from the stack. This method provides access to that stack. This method returns the list of currently executing file names as an object that can be iterated or used in a query expression.

`web_response->`**`getInclude`**`(`*path::string*`)`

>   Locates the file and will return an object that can be invoked to execute the file. The method will fail if the file does not
>   exist.

For compatibility and simplicity, Lasso supports the following unbound methods which function in the same manner as the
`web_response` bound methods:

`include``(`*path::string*`)`

`library``(`*path::string*`)`

>   These methods include the file indicated by the path in the same manner as the **`web_response->include`** and
>   **`web_response->includeLibrary`** methods.

30.2.2 Writing Response Headers

The **`web_response`** object provides methods for setting the outgoing response's HTTP headers. When a request is begun, a
few default HTTP headers are established. The request handler code can add, modify, or remove these headers as needed.
Headers can be set or removed freely during a request; however, once any data has been sent then headers can no longer be
effectively manipulated.

Note that the HTTP status code and message are not HTTP headers and so are not manipulated through these methods.

`web_response->`**`header`**`(`*name::path*`)`

`web_response->`**`headers`**`()` → trait_ForEach

>   These methods return existing outgoing headers. The first method finds the first occurrence of the indicated header
>   and returns its value. The second method returns all the current headers as an object that can be iterated over or used
>   in a query expression. Each element is a pair object containing the header name/value in the pair's first/second.

`web_response->`**`setHeaders`**`(`*headers::trait_forEach*`)`

`web_response->`**`replaceHeader`**`(`*header::pair*`)`

`web_response->`**`addHeader`**`(`*header::pair*`)`

>   These methods permit headers to be set or replaced. The first method sets all the headers for the response. These
>   headers should be given as a series of pairs containing the header names/values. The second method accepts a header
>   name/value pair and replaces matching header with the new value. If the existing header isn't found, the new header
>   is simply added. The third method accepts a new header name/value pair and adds it to the list of outgoing headers.
>   This method does not check for duplicate headers.

30.2.3 Setting Cookies

Outgoing cookies are added to the response HTTP headers by the **`web_response`** object. It provides a method for setting a
cookie and a method for enumerating all cookies being set.

Setting a cookie requires specifying a name and a value and optionally a domain, expiration, path, and SSL secure flag. These
values are supplied as parameters when setting a cookie. Cookie headers are not created until the request processing is com-
pleted and the response is to be sent to the client.

`web_response->`**`setCookie`**`(`*nv::pair*`, `*-domain=?*`, `*-expires=?*`, `*-path=?*`, `*-secure=false*`)`

>   Sets the indicated cookie. Any duplicate cookie would be replaced. The first parameter must be the cookie *`name=value`*
>   pair. If used, the **`-domain`** and **`-path`** keyword parameters must have string values.
>
>   The **`-expires`** parameter can be either a date object, a duration object, an integer, a string, or any object that will
>   produce a suitable value when converted into a string. A date indicates the absolute date at which the cookie will
>   expire. A duration indicates the time that the cookie should expire based on the time at which the cookie is being set.
>   An integer indicates the number of minutes until the cookie expires. Any other object type is appended directly to the
>   outgoing cookie header string.

```
web_response->cookies() → trait_forEach
```
> Returns a list of all the cookies set for this response. The individual cookies are represented by map objects containing keys for 'name', 'value', 'domain', 'expiration', 'path' and 'secure'. Manipulating a cookie value in the list will alter its resulting cookie header.

30.2.4 Setting the Response Body

Lasso allows you to programatically inspect and set the contents of the response body. This can be useful for code that needs to clear any data that has been already added to the response body and insert something completely different (e.g. display an error message).

```
web_response->rawContent()
```

```
web_response->rawContent=(text)
```
> The first method returns the current contents of the response body. Note that any plain text or auto-collected data in the currently executing code file will not be part of the body until the code file finishes processing. The second method allows for setting the contents of the response body to the value specified by the **text** parameter.

```
web_response->sendChunk()
```
> This method is used in complex HTTP sessions, and allows for sending the HTTP response body in multiple chunks. Each time it is called, it sends the current contents of the response in **rawContent** and then clears it for building the next chunk. If the headers for the response have not yet been sent, it will first send them before sending the first chunk.

30.2.5 Sending Response Data

By default, the result of a request will have a *text/html* content type with a UTF-8 character set and the body data will be generated from a Lasso string object that always consists of Unicode character data. In order to output binary data, the bytes need to be set directly and the response's *Content-Type* header adjusted accordingly. The method **web_response->rawContent** can be used to get or set the outgoing content data.

It is advised to call **web_response->abort** soon after setting binary response data or at least to ensure that no stray character data is inadvertently added into the outgoing data buffer as it will corrupt the output.

When manually setting the raw content, the *Content-Type* header should usually be adjusted to accommodate the change. Use the **web_response->replaceHeader** method to replace the existing header with the new value.

The **web_response** object provides the **sendFile** method which packages together many of the steps required to send binary data to the client to be viewed either inline or downloaded as an attachment.

```
web_response->sendFile(data::trait_each_sub, name=null, -type=null, -disposition='attachment', -charset='',
                       -skipProbe=false, -noAbort=false, -chunkSize=fcgi_bodyChunkSize, -monitor=null)
```
> Sets the raw content and headers for the response. It then optionally aborts, ending the request and delivering the data to the client. This method replaces all existing headers with new *MIME-Version*, *Content-Type*, *Content-Disposition* and *Content-Length* headers.
>
> The first parameter ("data") can be any object that supports **trait_each_sub**. This includes objects such as string, bytes, and file. The second parameter ("name") is optional, but if given it will trigger the addition of a "filename=" element to the *Content-Disposition* header. This controls the file name that the user agent will use to save a downloaded file.
>
> The subsequent keyword parameters control the following:
>
> > Parameters
> >
> > - **-type** (*string*) – Indicates the value for the *Content-Type* header. If this is not specified and **-skipProbe** is not set to "false", then the incoming data will be lightly probed to determine what type of data it is. The following data types are automatically recognized: GIF, PDF, PNG, JPEG. Unrecognized data types are set to have the *application/octet-stream* content type.

- **-disposition** (*string*) – Indicates the value for the *Content-Disposition* header. This value defaults to "attachment". The other possible value is "inline".

- **-charset** (*string*) – If given, this string will be appended to the *Content-Type* header as a ";charset=" component.

- **-skipProbe** (*boolean*) – Defaults to "false". If set to "true", no content type probe will occur.

- **-noAbort** (*boolean*) – Defaults to "false". This means that **sendFile** will abort by default after the data is delivered to the client. Set this parameter to "true" in order to prevent the abort.

- **-chunkSize** (*integer*) – Sets the size of the buffer with which the data is read and sent to the client. This mainly has a benefit when sending physical file data as it controls the memory usage. This value defaults to "65535", the result of the **fcgi_bodyChunkSize** method.

- **-monitor** – An object can be given to monitor the send process. Whatever object is given here will have its invoke method called for each chunk sent. The invoke will be passed the bytes object for the current chunk as well as an integer indicating the overall size of the bytes being sent.

If the **sendFile** method succeeds and does not abort, no value is returned.

web_response->abort()
> Stops Lasso from sending any further data. Same as calling **abort**.

30.2.6 HTTP Response Status

The HTTP response status line consists of a numeric code and a short textual message. When a request is first started it is given a "200 OK" status line. If a file is requested that does not exist, Lasso will respond with a "404 Not Found" status. An unhandled failure will generate a "500 Unhandled Failure" status.

The status can be set or reset multiple times. Its value is not used until the request data is sent to the client. However, once any data has been sent then the status can no longer effectively be set.

The following methods get or set the HTTP response status:

web_response->setStatus(*code::integer*, *msg::string***)**

web_response->getStatus() → pair
> The first method sets the HTTP status code and message. The second returns the status as a pair containing the code/message as the pair's first/second.

30.3 At Begin and End

Lasso permits arbitrary code to be run immediately before and immediately after a request with full access to the **web_request** and **web_response** objects. Code run before a request can manipulate the request data that will be used by the request handler code. Code run after a request can manipulate the outgoing headers and content body, doing tasks such as rewriting HTML links or compressing data for efficiency.

Code to be run after a request completes is added during the request itself through the **web_response->addAtEnd** method. Since code to be run before a request must be added outside of any request, the **define_atBegin** method is used. These methods are described below.

define_atBegin(*code***)**
> Installs code to be invoked at the beginning of each request. The code will have access to the **web_request** and **web_response** objects that will be available during the request's duration. At-begin code can set response headers and data and complete the request if it chooses, thus fully intercepting the normal request URI file request and processing routines. This is the recommended route for applications wanting to provide virtual URLs. Once an at-begin is in place it

cannot be removed. Multiple at-begins are supported and are run in the order in which they are installed. (The easiest way to install an at-begin is to place it in the instance's "LassoStartup" directory.)

The object installed as the at-begin code is copied to each request's thread each time. This means that a capture's local variables or any object's data members are deeply copied each time. The most efficient steps would be to define a method as the at-begin handler and then pass a reference to that method as the at-begin code. For example, passing \foo to **define_atBegin** would pass the foo method to **define_atBegin**. It would be invoked for each request and use the **web_request** and **web_response** within it.

define_atEnd(*code***)**

web_response->addAtEnd(*code***)**

These methods set the parameter to be run at the request's end. (The **define_atEnd** method just calls **web_response->addAtEnd**.) At-end code is normally run before data is sent to the client, but this may not be the case if data has been manually pushed using the **web_response->sendChunk** method. At-begins are executed before the session link rewriter is run. Multiple at-ends are supported and each are run in the order in which they were installed.

At-ends are added on a per-request basis, as opposed to at-begins which are added globally. At-end code is not copied in any way. A capture passed to this method will be detached.

Authentication

Lasso Server provides a built-in users and groups system. Initially, this system is only used to secure access to the Lasso Server Admin application. It can be used to provide authentication for your own web apps; however, Lasso is also flexible enough to support custom security and authentication mechanisms.

Lasso's security system data is stored in a SQLite database located in the instance's "SQLiteDBs" directory. Passwords are not stored in plain text, though other information such as user names and group names are unencrypted.

Within the system, users are grouped into particular realms. A *realm* completely separates its users such that the same username+password combination could exist in two different realms and they would be considered two unique users. A user only ever belongs to one realm which it is assigned to when the user is created. When a Lasso Server instance is first initialized, a "Lasso Security" realm is created. This is the default realm used in all the security-related methods and types. Alternate realms can be specified when needed.

Users can be grouped together. Each *group* can contain zero or more users. Users can belong to multiple groups at the same time. Users from different realms can belong to the same group. The special group "ANYUSER" always consists of all users. The special group "ADMINISTRATORS" is used to control who can access the Lasso Server Admin application as well as other system-related applications.

The built-in security system is accessed through two different interfaces: the set of **auth_...** methods and the **security_registry** object.

31.1 Authenticating Users

Web apps use the **auth_...** methods to execute simple security checks. The checks acquire the username, password, and realm information from the current web request and, therefore, require that a request be active. In all cases, if the check fails or if no username and password are provided, then the auth methods will generate an "HTTP 401 Unauthorized" response with a **WWW-Authenticate: Digest** header. The request is then aborted, by default. If the security checks succeed, then the methods return nothing. If electing not to abort when the check fails, a caller can check **web_response->getStatus** to determine the result.

auth_admin(*-realm::string='Lasso Security'*, *-noAbort=false*, *-errorResponse=''*, *-noResponse=false*)
> Checks that the current authenticated HTTP client user is in the "ADMINISTRATORS" group. An alternate realm can be given and the default abort behavior can be altered. By default, a simple "Not Authorized" content body is generated; this can be specified with the **-errorResponse** parameter or the body can be left empty by passing **-noResponse**.

auth_user(*name::string*, *-realm::string='Lasso Security'*, *-noAbort=false*, *-errorResponse=''*, *-noResponse=false*)
> Checks that the current authenticated HTTP client user matches the given name.

auth_group(*group::string*, *-realm::string='Lasso Security'*, *-noAbort=false*, *-errorResponse=''*, *-noResponse=false*)
> Checks that the current authenticated HTTP client user is in the specified group.

31.2 Managing Users

The **security_registry** object provides a more complete interface to Lasso's security system. It does not rely on an ongoing web request and can be used freely once the system is initialized. The **security_registry** methods permit a realm to be specified, but the object otherwise defaults to using the "Lasso Security" realm.

Before the security system can be used, it must be initialized by calling the **security_initialize** method. Lasso Server calls this method as it starts up and so this step can be safely skipped by web applications. Command-line or other tools that want to use the security system should call this method as early as possible when starting up.

A **security_registry** object can be created with zero parameters. When created, it will open a connection to the security database. The object must be closed once it is no longer required.

security_initialize()
> Initializes Lasso's ability to connect to the security SQLite database. Lasso Server calls this automatically, but you will need to call it if you wish to use the **security_registry** type.

type **security_registry**

security_registry()
> Creates a new **security_registry** object. Once created, it can be used to:
>
> - Add/remove groups
> - Alter group metadata (name, enabled)
> - Add/remove users
> - Alter user metadata (password, comment, enabled)
> - Assign/unassign users to groups
> - Validate username/password/realm combinations

security_registry->close()
> Closes the **security_registry** object's connection to the security information database.

security_registry->addGroup(_name::string, enabled::boolean=true, comment::string=""_**)**
> Attempts to add the specified group. A group is enabled by default, but it can be explicitly disabled. A comment can be provided when the group is created and will be stored in the database for reference.

security_registry->getGroupID(_name::string_**)**
> Returns the integer ID for the indicated group. This ID can be passed to subsequent methods to identify the group.

security_registry->listGroups(_-name::string_**)**

security_registry->listGroupsByUser(_userid::integer_**)**

security_registry->listGroupsByUser(_username::string_**)**
> These methods list groups in a variety of ways. The first method will list all groups. A **-name** parameter can be specified to perform wildcard searches. The wildcard character is "%". The second and third methods return a list of groups that the indicated user belongs to.
>
> Each group is represented by a map object containing the keys 'id', 'name', 'enabled', and 'comment'.

security_registry->removeGroup(_groupid::integer_**)**

security_registry->removeGroup(_name::string_**)**
> These methods will remove the indicated group. All users are disassociated from the group.

security_registry->updateGroup(_groupid::integer, -name=null, -enabled=null, -comment=null_**)**
> Modifies the information for the group. Passing any of the **-name**, **-enabled** or **-comment** parameters will set the appropriate data.

`security_registry->`**addUser(***username::string, password::string, enabled::boolean=true, comment::string='',*
        *-realm='Lasso Security'***)**
> Adds a new user to the system. A username and password must be supplied. An optional **enabled** and **comment** parameter can be provided. The **-realm** parameter controls which realm the user is placed in. The default realm is "Lasso Security". The user's information record is then returned as a map object containing the keys 'id', 'name', 'enabled', 'comment', 'email', 'real_name' and 'realm'.

Note: The 'email' and 'real_name' fields are not used at this time.

`security_registry->`**addUserToGroup(***userid::integer, groupid::integer***)**
> Adds a user to a group. Both user and group must be indicated by their integer IDs.

`security_registry->`**checkUser(***username::string, password::string, -realm::string='Lasso Security'***)**
> Authenticates the given username and password and will return user's record if it succeeds. The return value will be a map object containing the keys 'id', 'name', 'enabled', 'comment', 'email', 'real_name' and 'realm'. If the check fails, this method will return "void". The check will fail if the user account is not enabled.

`security_registry->`**countUsersByGroup(***groupid::integer***)**
> Returns the number of users in the indicated group.

`security_registry->`**getUser(***userid::integer***)**

`security_registry->`**getUser(***name::string, -realm::string='Lasso Security'***)**

`security_registry->`**getUserID(***name::string, -realm::string='Lasso Security'***)**
> The first two methods return the user record for the indicated user. The third method returns the ID of the indicated user.

`security_registry->`**listUsers(***-name::string='', -realm=null***)**

`security_registry->`**listUsersByGroup(***name::string***)**
> These methods list users and return their user records. The first method permits a **-name** pattern to be specified as well as a realm. Not specifying **-realm** will result in all realms being searched. The second method lists all of the users in the indicated group.

`security_registry->`**removeUser(***userid::integer***)**

`security_registry->`**removeUserFromGroup(***userid::integer, groupid::integer***)**

`security_registry->`**removeUserFromAllGroups(***userid::integer***)**
> These methods can be used to remove a user from the system, remove a user from a group, or remove a user from all groups, respectively.

`security_registry->`**userPassword=(***password::string, userid::integer***)**

`security_registry->`**userEnabled=(***enabled::boolean, userid::integer***)**

`security_registry->`**userComment=(***comment::string, userid::integer***)**
> Given a user ID, these setter methods will assign that user's password, enabled state, or associated comment, respectively. Call these by specifying the user ID as a parameter and the value as an assignment.

```
security_registry->userComment(1) = "I am the first user!"
```

Sessions

Sessions allow variables to be created that persist between requests within a website. Rather than passing data using HTML forms or URLs, visitor-specific data can be stored in Lasso variables that are automatically saved and retrieved by Lasso for each page a visitor loads.

Sessions can be used for a variety of purposes, including:

- **Saving state** – Sessions can store the current state of a website for a given visitor. They can retain what the last search they performed was, how the data on a results page was sorted, or in what format the data should be presented.

- **Storing references to database data** – Key column values can be stored in a session for quick access. These might include records in a user database or a shopping cart database.

- **Storing authentication information** – After a visitor has authenticated using a username and password, that authentication information can be stored in a session and then checked to verify that the same visitor is accessing data from page request to page request.

- **Storing data without using a database** – Any type that imports the **trait_serializable** trait can be stored in a session variable. A website with multiple forms can have the data from each form stored in a session and only placed in the database once the final form is submitted. Or, a shopping cart can be stored in a session and only placed in an orders table upon checkout.

32.1 How Sessions Work

A session has three characteristics: a name, a list of variables that should be stored, and an ID string that identifies a particular site visitor.

Name
> The session name is defined when the session is created by the **session_start** method. The same session name must be used for each request that wants to load the session. The name typically represents the type of data being stored in the session, e.g. "Shopping_Cart" or "Site_Preferences".

Variables
> Each session maintains a list of variables that are being stored. Variables can be added to the session using **session_addVar**. The values for all variables in the session are remembered at the end of each request that loads the session. The value for each saved variable is restored when that session is next loaded.

ID
> Lasso automatically creates an ID string for each site visitor when a session is created. The ID string is either stored in a cookie or passed from page to page using the "-lassosession" GET or POST parameter. When a session is loaded, the ID of the current visitor is combined with the name of the session to locate and load the particular set of variables for that session and the current visitor.

Important: Only **thread variables** can be added to a session.

Sessions are created and loaded using the **session_start** method. This method should be used early for each request that needs access to the session variables. The **session_start** method either creates a new session or loads an existing session depending on whether there are existing variables currently stored for the site visitor.

Sessions can be set to expire after a specified amount of idle time. The default is 15 minutes. If the visitor has not loaded a page that starts the session within the idle time limit, then the session will be deleted automatically. Note that the idle timeout resets each time a request loads the session.

Once a variable has been added to a session using the **session_addVar** method, its stored value will be set each time the **session_start** method is called. The variable does not need to be added to the session on each request, though it is safe to do so. A variable can be removed from a session using the **session_removeVar** method. This method does not alter a variable's current value, but does prevent the value of the variable from being saved for the session. This means the variable will not be available on future session loads.

32.2 Session Methods

Below is a description of each of the session methods:

session_start(…)
>   Starts a new session or loads an existing session.

>   Parameters

>   - **name** (*string*) – The name of the session. This is the only required parameter. All other parameters are optional and have default values that cover the majority of usages.

>   - **-expires** (*integer=15*) – The idle expiration time for the session in minutes.

>   - **-id** (*string=null*) – Optionally sets the ID for the current visitor. This permits the ID to be supplied explicitly by the developer. If no ID is specified, then Lasso will automatically create an ID.

>   - **-useCookie** (*boolean=true*) – If "true", then sessions will be tracked by cookie. **-useCookie** defaults to "true" unless **-useLink**, **-useAuto**, or **-useNone** is specified.

>   - **-useLink** (*boolean=false*) – If "true", then sessions will be tracked by modifying all the absolute and relative links in the outgoing response data.

>   - **-useNone** (*boolean=false*) – If specified, no links on the current page will be modified and a cookie will not be set. **-useNone** allows custom session tracking to be used, bypassing the automated methods provided by Lasso.

>   - **-useAuto** (*boolean=true*) – This option automatically uses **-useCookie** if cookies are available on the visitor's browser or **-useLink** if they are not. Since Lasso has no way of knowing if cookies are enabled when a session is first started, **-useLink** is implicitly "true" on that first request and links will be adjusted to carry the session. If the session cookie is present on subsequent requests, **-useLink** will be implicitly "false" and links will not be adjusted.

>   - **-cookieExpires** (*integer=null*) – Optionally sets the expiration in minutes for the session cookie. This permits the cookie expiration to be set, regardless of the overall expiration for the session itself.

>   - **-domain** (*string=null*) – Optionally sets the domain for the session cookie.

>   - **-path** (*string='/'*) – Optionally sets the path for the session cookie.

>   - **-secure** (*boolean=false*) – If "true", the session cookie will only be sent back to the web server on requests for HTTPS secure web pages. The **session_end** should also be specified with **-secure** if this option is desired.

>   - **-rotate** (*boolean=false*) – If "true", the session will have a new ID generated for it on each request.

session_id(*sessionName::string*)

> Returns the current session ID. Accepts a single parameter: the name of the session for which the session ID should be returned.

session_addVar(*sessionName::string*, *varName::string*)

> Adds a variable to a specified session. Accepts two parameters: the name of the session and the name of the variable.

session_removeVar(*sessionName::string*, *varName::string*)

> Removes a variable from a specified session. Accepts two parameters: the name of the session and the name of the variable.

session_end(*sessionName::string*, *-secure=false::boolean*)

> Deletes the stored information about a named session for the current visitor. Accepts a required parameter: the name of the session to be deleted, and an optional keyword parameter: `-secure`. The `-secure` parameter should be "true" if the `-secure` parameter was "true" when **session_start** was called.

session_abort(*sessionName::string*)

> Prevents the session from being stored at the end of the current request. This allows graceful recovery from an error that would otherwise corrupt data stored in the session. Accepts a single parameter: the name of the session to be aborted.

session_result(*sessionName::string*)

> When called immediately after the **session_start** method, it returns "new", "load", or "expire" depending on whether a new session was created, an existing session loaded, or an expired session forced a new session to be created, respectively. If **session_start** is called with the optional `-rotate` keyword parameter, the word "rotate" may also be returned from this method.

session_deleteExpired()

> This method is used internally by the session manager and does not normally need to be called directly. It triggers a cleanup routine that deletes expired sessions from the current session storage location.

Tip: The `-useCookie` is the default for **session_start** unless `-useLink` or `-useNone` are specified. Use `-useLink` to track a session using only links. Use both `-useLink` and `-useCookie` to track a session using both links and a cookie.

32.3 Starting a Session

The **session_start** method is used to start a new session or to load an existing session. When the **session_start** method is called with a given **name** parameter it first checks to see whether an ID is defined for the current visitor. The ID is searched for in the following three locations:

- **Parameter** – If the **session_start** method has an `-id` keyword parameter then it is used as the ID for the current visitor.

- **Cookie** – If a session tracker cookie is found for the name of the session then the ID stored in the cookie is used.

- **-lassosession** – If a "-lassosession" parameter for the name of the session was specified as a GET or POST parameter then that value is used as the session ID.

The name of the session and the ID are used to check whether a session has already been created for the current visitor. If it has, then the variables in the session are loaded, replacing the values for any variables of the same name that are already active on the current page.

If no ID can be found, the specified ID is invalid, or if the session identified by the name and ID has expired, then a new session is created.

After the **session_start** method has been called, the **session_id** method can be used to retrieve the ID of the current session. It is guaranteed that either a valid session will be loaded or a new session will be created when **session_start** is called.

Note: The **session_start** method must be used once for each request that will access session variables.

32.4 Session Tracking

The session ID for the current visitor can be tracked using two different methods, or a custom tracking system can be devised. The tracking system to be used depends on which parameters are specified when the **session_start** method is called.

32.4.1 Using Cookies

The default session tracking method is to use a browser cookie. If no other method is specified when creating a session, then the **-useCookie** method is used by default. The cookie will be inspected automatically when the visitor makes another request that includes a call to the **session_start** method. No additional programming is required.

The session tracking cookie is of the following form: the name of the cookie starts with "_LassoSessionTracker_" and is followed by the name given to the session in **session_start**. The value for the cookie is the session ID as returned by **session_id**.

32.4.2 Using Links

If the **-useLink** parameter is specified in the **session_start** method, Lasso will automatically modify links contained on the current page. No additional programming beyond specifying the **-useLink** parameter is required.

By default, links contained in the "href" parameter of anchor tags will be modified. Links are only modified if they reference a file on the same machine as the current website. Any links that start with any of the following strings are not modified: "file://", "ftp://", "http://", "https://", "javascript:", "mailto:", "telnet://", "#".

Links are modified by adding a **-lassosession:SessionName** parameter to the end of the link. The value of the parameter is the session ID, as returned by the **session_id** method. For example, an **<a>** tag referencing the current file with a session named "Cart" would have **?-lassosession:Cart=** and the session ID appended after the URL path.

32.4.3 Using Cookies with a Link Fallback

If the **-useAuto** parameter is specified in the **session_start** method, Lasso will check for a cookie with an appropriate name for the current session. If the cookie is found then **-useCookie** will be used to propagate the session. If the cookie cannot be found, then **-useLink** will be used to propagate the session. This allows a site to preferentially use cookies to propagate the session but fall back on links if cookies are disabled in the visitor's browser.

32.4.4 Using Custom Tracking

If the **-useNone** parameter is specified in the **session_start** method, Lasso will not attempt to propagate the session. The techniques described later in this chapter for manually propagating the session must then be used.

32.5 Using Sessions

Use the **session_...** methods to work with sessions in Lasso.

32.5.1 Start a Session

The following example starts a session named "Site_Preferences" with an idle expiration of 24 hours (1440 minutes). The session will be tracked using both cookies and links.

```
session_start('Site_Preferences', -expires=1440, -useLink, -useCookie)
```

32.5.2 Add Variables to a Session

Use the **session_addVar** method to add a variable to a session. Once a variable has been added to a session its value will be restored when **session_start** is next called. In the following example, a variable named "real_name" is added to a session named "Site_Preferences":

```
session_addVar('Site_Preferences', 'real_name')
```

32.5.3 Remove Variables from a Session

Use the **session_removeVar** method to remove a variable from a session. The variable will no longer be stored with the session, and its value will not be restored in subsequent requests. The value of the variable in the current request will not be affected. In the following example, a variable named "real_name" is removed from a session named "Site_Preferences":

```
session_removeVar('Site_Preferences', 'real_name')
```

32.5.4 Delete a Session

A session can be deleted using the **session_end** method with the name of the session. The session will be ended immediately. None of the variables in the session will be affected in the current request, but their values will not be restored in subsequent requests. Before a session can be ended, it has to be loaded, so you must call **session_start** before you can call **session_end** Sessions can also end automatically if the timeout specified by the **-expires** keyword parameter is reached. In the following example the session "Site_Preferences" is ended:

```
session_start('Site_Preferences')
session_end('Site_Preferences')
```

32.5.5 Pass a Session in an HTML Form

Sessions can be added to URLs automatically using the **-useLink** keyword parameter in the **session_start** method. In order to pass a session using a form, a hidden input must be added explicitly. The hidden input should have the name "-lassosession:SessionName" and a value of **session_id**. In the following example, the ID for a session "Site_Preferences" is returned using **session_id** and passed explicitly in an HTML form:

```
<form action="save.lasso" method="post">
<input type="hidden" name="-lassosession:Site_Preferences" value="[session_id('Site_Preferences')]" />
</form>
```

32.5.6 Conditionally Track a Session Using Links

The following example shows how to start a session using links if cookies are disabled. The **-useAuto** parameter will first try setting a cookie and decorate the links on the current page. If the session cookie is found on subsequent page loads, it will be

used and the links on the page will not be decorated. If the cookie cannot be found, then links will be used to propagate the session.

```
session_start('Site_Preferences', -useAuto)
```

32.5.7 Session Example

This example demonstrates how to use sessions to store user-specific values that are persistent from request to request. It displays a form which the user can manipulate. The user's selections are saved from one request to the next.

Sessions will be used to track the visitor's name, email address, favorite color, and favorite forms of faster-than-light travel in session variables.

```
<?lasso
    local(
        wr = web_request,
        sessionName = 'sessions_example'
    )
    // Start the session
    session_start(#sessionName)
    if(session_result(#sessionName) != 'load') => {
        // The session did not already exist,
        // so set the variables we want to be saved
        session_addVar(#sessionName, 'realName')
        session_addVar(#sessionName, 'emailAddress')
        session_addVar(#sessionName, 'favoriteColor')
        session_addVar(#sessionName, 'hyperDrive')
        session_addVar(#sessionName, 'warpDrive')
        session_addVar(#sessionName, 'wormHole')
        session_addVar(#sessionName, 'improbabilityDrive')
        session_addVar(#sessionName, 'spaceFold')
        session_addVar(#sessionName, 'jumpGate')

        // Initialize our vars to empty values
        var(realName, emailAddress, favoriteColor, hyperDrive, warpDrive,
            wormHole, improbabilityDrive,  spaceFold, jumpGate)

    else(#wr->param('submit'))
        // The session existed
        var(realName)          = #wr->param('realName')
        var(emailAddress)      = #wr->param('emailAddress')
        var(favoriteColor)     = #wr->param('favoriteColor')
        var(hyperDrive)        = #wr->param('hyperdrive')
        var(warpDrive)         = #wr->param('warpdrive')
        var(wormHole)          = #wr->param('wormhole')
        var(improbabilityDrive) = #wr->param('improbabilitydrive')
        var(spaceFold)         = #wr->param('spacefold')
        var(jumpGate)          = #wr->param('jumpgate')
    }
?>
<html>
<body>
    <form action="[include_currentPath]" method="POST">
        Your Name:
        <input type="text" name="realName" value="[$realName]" />
        <br />
        Your Email Address:
```

```
<input type="text" name="emailAddress" value="[$emailAddress]" />
<br />
Your Favorite Color:
<select name="favoriteColor">
    <option value="blue"[
        $favoriteColor == 'blue' ? ' selected="yes"'
    ]> Blue </option>
    <option value="red"[
        $favoriteColor == 'red' ? ' selected="yes"'
    ]> Red </option>
    <option value="green"[
        $favoriteColor == 'green' ? ' selected="yes"'
    ]> Green </option>
</select>
<br />
Your Favorite Forms of Superluminal Travel:<br />
<input type="checkbox" name="hyperdrive" value="hyperdrive"
    [$hyperDrive ? ' checked="yes"'] /> Hyper Drive<br />
<input type="checkbox" name="warpdrive" value="warpdrive"
    [$warpDrive ? ' checked="yes"'] /> Warp Drive<br />
<input type="checkbox" name="wormhole" value="wormhole"
    [$wormHole ? ' checked="yes"'] /> Worm Hole<br />
<input type="checkbox" name="improbabilitydrive" value="improbabilitydrive"
    [$improbabilityDrive ? ' checked="yes"'] /> Improbability Drive<br />
<input type="checkbox" name="spacefold" value="spacefold"
    [$spaceFold ? ' checked="yes"'] /> Space Fold<br />
<input type="checkbox" name="jumpgate" value="jumpgate"
    [$jumpGate ? ' checked="yes"'] /> Jump Gate<br />
<br />
<input type="submit" name="submit" value="Submit" />
<a href="[include_currentPath]">Reload This Page</a>
    </form>
</body>
</html>
```

LassoApps

Lasso Server provides a means for bundling source files, HTML, images, and other media types into a single deployable unit called a *LassoApp*. LassoApps are served over the web using Lasso Server's FastCGI interface. Lasso Server is required to run LassoApps. A single server can run multiple LassoApps at the same time.

The LassoApp system provides a framework of features to make app development easier and to support a clean and maintainable design. This system also permits data in one app to be accessed and shared by another, allowing multiple apps to work in concert.

33.1 LassoApp Concepts

LassoApps consist of regular files, logically structured into a tree of nodes and resources. This node tree is constructed to match the file and directory structure inside the LassoApp bundle. Each node is associated with one or more resources. Resources are generally either Lasso pages, CSS, JavaScript, HTML/XML, XHR, image, or other raw or binary file types.

This node/resource/content representation system permits the logic for producing a particular application object, such as a "user" or a set of database result rows, to be isolated from logic for its display. It also allows application objects to be represented in a variety of manners, and for those representations to be modified, without having to extend the application objects themselves.

Additionally, the system is unobtrusive, permitting the developer to use their own methodologies and frameworks while still taking advantage of the LassoApp system in pieces or as a whole.

33.1.1 Nodes

Nodes represent the object structure behind a live LassoApp. This object structure is hierarchical, like a directory structure. The node tree begins with the "root" node. That root node has a series of subnodes and those subnodes have zero or more subnodes beneath them. In the case of the root node, each of its subnodes represent the currently installed and running LassoApps.

Each node has a name and this name is used when locating a particular node within the tree. Nodes are addressed using standard forward-slash path syntax. The root node is named "lasso9", so it is accessed using the path `/lasso9`. The names of subnodes are appended to the path following a "/".

```
/lasso9/LassoAppName/resourceName
/lasso9/AddressBook/groups/userX
```

The default web server configuration for Lasso Server will direct all paths beginning with `/lasso9` to Lasso Server. This is the default method for accessing LassoApps over the web, though the configuration can be modified for other situations or server requirements. See the section **Server Configuration** for more information.

33.1.2 Resources

Nodes not only serve as containers for subnodes, they also represent zero or more *resources*. These resources represent data files, such as images, CSS, or Lasso source files. Resources are used to produce an object that the LassoApp system must then transform into a format suitable for sending back to the client. Each resource is associated with a content type. This association is done either explicitly using the resource file's name, or by relying on the default content type, which is *text/html*.

LassoApp Node Hierarchy
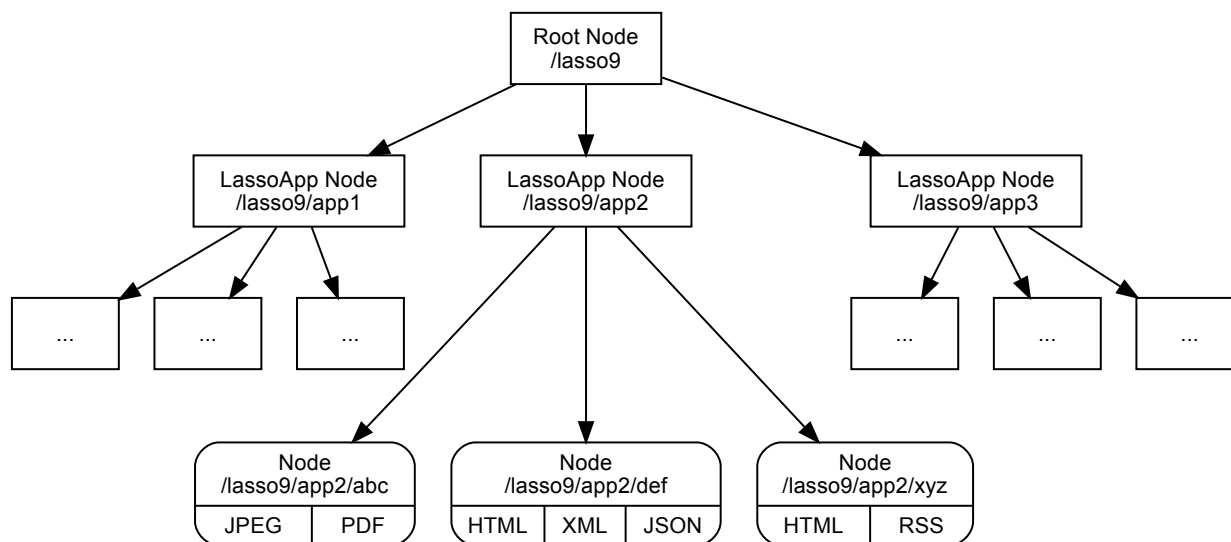
33.1.3 Content Representations

Each resource is associated with a content type which is used when handling, or representing, the object produced by a resource. This handling occurs automatically when a node is requested via a web request and is formatted for output via HTTP. This handling is performed by a variety of *content representation* objects, each tailored for specific file extension, like ".jpg" or ".js", and content types such as *image/jpeg* or *application/javascript*. New content representation objects can be added and existing representations can be tailored for specific application objects.

If there exists a content representation object for a given node resource and content type, then that resource can be invoked and the resulting object given to the content representation object for transformation or special handling.

To illustrate, consider a resource such as a PNG image that comes from a static, unchanging PNG file within a LassoApp. After the LassoApp is bundled for deployment, that image file may not actually exist on disk; instead it is contained within the LassoApp in a specialized format. Given the resource's PNG content type, the system chooses the appropriate content representation object. In turn, that object sets an *Expires* header for that web request, improving application performance by preventing future redundant image requests. The content representation object does not have to modify the object data, and in this case with PNGs, sets an HTTP header but returns the unaltered binary image data.

Another example would be a node resource that produces a "user" object containing a first name, last name, etc. A content representation can be added to handle that particular object type and formats it for display as HTML. Another content representation can be added to format it for sending back as JSON data, while another can be added to convert it to the vCard format.

33.2 Constructing a LassoApp

All LassoApps reside as either a file or a directory located within the "LassoApps" directory, which is located within the current Lasso home. (See the section *Instance Home Directory Contents* in the *Lasso Instance Manager* chapter for more details.)

LassoApps begin as a directory named according to the application. This directory contains all of the files for the application. Before deployment, this directory can be precompiled into the LassoApp format. However, Lasso Server will happily serve a plain LassoApp directory as long as it is placed in the proper location. This means that an application can be deployed as a regular directory of files and also that a developer needn't take any special steps transitioning between developing and testing an application.

Important: While the above is generally true, it is currently required to restart Lasso Server when *adding* or *removing* files from an in-development LassoApp. We aim to remove this restriction in a future release. (File content can be modified without any such restrictions.)

33.2.1 Directory Organization

By using the concepts of nodes, resources, and content representation, a LassoApp can be organized logically and provide clean, hierarchical, natural language URLs.

For example a simple "Contacts" LassoApp could have a structure similar to the following:

```
LassoApps/
    mycontacts/
        contacts/
            index.lasso
        css/
            appstyle.css
        index.lasso
        js/
            scripts.js
        other/
            footer.lasso
            header.lasso
```

This structure would provide the "root" of the LassoApp as *http://example.com/lasso9/mycontacts* which will serve the "index.lasso" file.

33.3 Serving Content

Serving simple content such as images or raw text and HTML is as simple as putting the file into the LassoApp root directory. As long as the file has the appropriate file extension (e.g. ".jpg", ".txt", ".html") then it will be served as expected. Files with a extension other than ".lasso", ".lasso9" or ".inc" will be served as plain data, meaning they will not be parsed, compiled and executed by Lasso Server.

33.3.1 Serving Processed Content

Processed content is any data produced programmatically by executing Lasso source code files. Such data can be generated wholly by Lasso code, or partially by embedding Lasso code in HTML or other types of templates. This type of content must reside in a file with an extension of ".lasso", ".lasso9" or ".inc".

The outgoing content type of processed content is very important. The content type determines any modifications or special handling that the data will receive before it is ultimately converted into a stream of bytes and sent to the client. By default, the content type for a "*.lasso" file is *text/html*. Lasso Server will automatically set the outgoing content type accordingly. A file will be given the default content type when accessed via a URL with either no extension, a ".html" extension or a ".lasso" extension. For example, requests for the following URLs:

```
http://localhost/lasso9/AddressBook/users
http://localhost/lasso9/AddressBook/users.html
http://localhost/lasso9/AddressBook/users.lasso
```

will, assuming the standard Lasso Server web server configuration, be mapped to these files in the LassoApp and served with the content type *text/html*:

```
/AddressBook/users.html
/AddressBook/users.html
/AddressBook/users.lasso
```

33.3.2 Explicit Content Types

The outgoing content type for a source file can be indicated in the file's name by placing the content type's file extension between square brackets. These files will be executed and the resulting value will be returned to the client using the indicated content type. The following shows some valid file names:

```
/AddressBook/users[html].lasso
/AddressBook/users[xml].lasso
/AddressBook/users[rss].lasso
/AddressBook/users[xhr].lasso
```

The files shown above will expose the following URLs:

```
http://localhost/lasso9/AddressBook/users.html
http://localhost/lasso9/AddressBook/users.xml
http://localhost/lasso9/AddressBook/users.rss
http://localhost/lasso9/AddressBook/users.xhr
```

Tip: A filename with an explicit content type overrides a file with a plain extension. For example, a request for *http://localhost/lasso9/AddressBook/users* or *http://localhost/lasso9/AddressBook/users.html* will be mapped to **/Address-Book/users[html].lasso** if it exists, and then fall back on **/AddressBook/users.html**.

33.3.3 Primary and Secondary Processing

Explicit content types can be used jointly with a similarly named regular "*.lasso" file. In this situation, first the *primary* file is executed and then its value is made available to the *secondary* file as it is executed. The primary file is always executed. Only then would the secondary file, which corresponds to the requested content type, be executed.

```
/AddressBook/users.lasso - primary content
/AddressBook/users[html].lasso - secondary
/AddressBook/users[xml].lasso - secondary
/AddressBook/users[rss].lasso - secondary
/AddressBook/users[xhr].lasso - secondary
```

Given the files shown above, if the URL *http://example.com/lasso9/AddressBook/users.html* was accessed, first the file "users.lasso" would be executed, and then the file "users[html].lasso" would be executed. The value produced by the first

would be made available to the second. This technique is used to separate the object produced by the primary file from its display, which is handled by the secondary file.

In this scenario, the file "users.lasso" could return an array of all the users in the address book. That list of users might need to be presented to the client in a variety of formats, like HTML, XML, or RSS. The primary file "users.lasso" is concerned only with producing the array of users. The secondary files each handle presenting that array in the desired format.

Since primary files usually return structured data, it is generally required to return the value using a **return** statement. However, primary files that simply need to return string data can do so without a return statement by surrounding the code with Lasso delimiters, which will cause the auto-collected value generated by executing that file to be returned. In such cases, the delimiter must be first visible character in the file; otherwise Lasso Server will treat the entire file as executable code.

The following examples show a series of files that produce and format a list of users for both HTML and XML display. The list is generated first by the "users.lasso" file, then that list is processed by the "users[html].lasso" and "users[xml].lasso" files.

users.lasso

```
// Note: Usually the type definition would be in an _init file
define user => type {
   data
      public firstname::string,
      public middleName::string,
      public lastname::string

   public oncreate(firstname::string, lastname::string) => {
      .firstname = #firstname
      .lastname = #lastname
   }
   public oncreate(firstname::string, middle::string, lastname::string) => {
      .firstname = #firstname
      .middlename = #middle
      .lastname = #lastname
   }
}

// Return an array of users
return array(user('Stephen', 'J', 'Gould'),
      user('Francis', 'Crick'),
      user('Massimo', 'Pigliucci'))
```

users[html].lasso

```
<html>
<title>Users List</title>
<body>
<table>
   <tr><th>First Name</th><th>Middle Name</th><th>Last Name</th></tr>
<?lasso
   // The primary value is given to us as the first parameter
   local(users) = #1

   // Start outputting HTML for each user
   with user in #users
   do {^
      '<tr><td>' + #user->firstName + '</td>
```

```
        <td>' + #user->middleName + '</td>
        <td>' + #user->lastName + '</td>
    </tr>'
    ^}
?>
</table>
</body>
</html>
```

users[xml].lasso

```
<userslist>
<?lasso
    // The primary value is given to us as the first parameter
    local(users) = #1

    // Start outputting XML for each user
    with user in #users
    do {^
        '<user><firstname>' + #user->firstName + '</firstname>
            <middlename>' + #user->middleName + '</middlename>
            <lastname>' + #user->lastName + '</lastname>
        </user>'
    ^}
?>
</userslist>
```

Pass Multiple Values from Primary to Secondary

To pass multiple values from primary to secondary processors, use a staticarray as a return from the primary:

```
// Return from primary processor
return (:
    'hello world',
    array(
        user('Stephen', 'J', 'Gould'),
        user('Francis', 'Crick'),
        user('Massimo', 'Pigliucci')
    )
)
```

The following sets local variables to the returned values from the primary processor, in the order they are specified. The number of local variables being set must match the number of elements in the returned staticarray. (See the section *Decompositional Assignment* in the *Variables* chapter.)

```
local(txt, users) = #1
```

33.4 Special Files

Using the naming conventions below, a LassoApp can be made to install or load files as needed.

33.4.1 Customizing Installation

One or more specially named files can be placed in the root level of a LassoApp directory to be executed the first time a LassoApp is loaded into Lasso Server. These files are named beginning with "_install." followed by any additional naming characters and ending with a ".lasso" extension. The simplest install file could be named "_install.lasso". For example, an install file for performing a specific task, such as creating database required by the app, could be named "_install.create_dbs.lasso".

Lasso Server will record the first time a particular install file is run. That file will not be executed again, even when the instance restarts. Only install files at the root of the LassoApp are executed.

33.4.2 Customizing Initialization

LassoApps can contain a special set of files that are executed every time the LassoApp is loaded. This loading occurs whenever Lasso Server starts up. These files are named beginning with "_init." followed by any additional naming characters and ending with ".lasso". The file "_init.lasso" is the simplest valid init file name. Only initialization files at the root of the LassoApp are executed.

Initialization files are used to define types, traits, and methods used within the application. This includes the definition of a thread object that can be used to synchronize aspects of the application, hold globally shared data, or perform periodic tasks.

During the normal operation of an application, definitions should be avoided. Redefining a method can have an impact on performance and memory usage, potentially leading to bottlenecks in your application. However, during application development, redefining a method is a common occurrence while source code is frequently modified. In this case, definitions can be placed in non-init files (i.e., a regular file) and included in the _init files using **lassoApp_include**. This allows the definition be loaded at startup while also letting the developer execute the file "manually" as it is updated during development.

33.4.3 Ignored Files

When serving a LassoApp, Lasso Server will ignore certain file paths based on their names. Although they can be included in a LassoApp, Lasso will not serve or process files or directories whose names begin with a period (**.**), hyphen (**-**), or two underscores (**__**). All other file names are permitted without restriction.

33.5 LassoApp Links

Use the **lassoApp_link** or **lassoApp_include** methods to link between resources in a LassoApp.

33.5.1 Internal Links

When creating a LassoApp, it is important not to hard-code paths to files within the app. Because the files within a LassoApp are not real files, Lasso Server will alter paths used in HTML links to be able to access the file data. The **lassoApp_link** method must be used for all intra-app file links.

lassoApp_link(*path::string*)
> Use this method to make links to files that are internal to a LassoApp. A LassoApp is compiled out of all the files within a folder. Any links between files within that code must be made using **lassoApp_link**, including links in HTML anchor tags (**<a>**), image tags (**<img src="...">**), and form actions.

To illustrate, consider a LassoApp that contains an image file called "icon.png" within an "images" subdirectory. In order to display the image, the **lassoApp_link** method would be used to alter the path, at runtime, to point to the true location of the file data. The following shows how **lassoApp_link** would be used to display the image. This example assumes that the link is being embedded in an HTML **<img>** tag:

```
<img src="<?= lassoApp_link('/images/icon.png') ?>" />
```

The path that gets inserted into the HTML document will vary depending on the system's configuration, but the end result would be the same: the image would be displayed.

In the context of our "AddressBook" LassoApp from earlier in the chapter, using a default server configuration, the link above would be "/lasso9/AddressBook/images/icon.png".

The **lassoApp_link** method must be used whenever a path to a file within the app is needed. Behind the scenes, Lasso Server will alter the path so that it points to the right location. However, **lassoApp_link** only operates on paths to files within the current LassoApp. That is, **lassoApp_link** does not work with paths to files in other LassoApps running on the same system.

33.5.2 LassoApp Includes

It is possible to directly access, or *include*, a LassoApp node given its path. This can be used to pull in file data within the current LassoApp as well as other LassoApps running on the system. This technique can be used to assemble a result page based on multiple files working together.

lassoApp_include(*path::string*)

lassoApp_include(*path::string, as::string, extra=?*)

lassoApp_include_current(*path::string, extra=?*)
> These methods allow you to include content from the LassoApp node specified in the **path** parameter. The default is to determine the content type by the extension of the node, but the second method allows you to specify the extension to use.
>
> The first two methods allow you to specify a node in any LassoApp, where the root of their path is the top level for all LassoApps. The last method **lassoApp_include_current** has its root set to the current LassoApp.

To include a LassoApp file from a Lasso file external to the LassoApp, the **lassoApp_include** method is used. This method accepts one string parameter, which is the path to the file to include. This path does not need to be altered via the **lassoApp_link** method. However, the path should be a full path to the file starting with the name of the LassoApp that contains the file. Additionally, **lassoApp_include** takes content representations into account. Therefore, if the HTML representation of a file is desired, the file path should include the ".html" extension.

For example, a LassoApp result page could consist of pulling in two other LassoApp files. Earlier in this chapter, several files were described representing a users list. These files represented the users list in several formats, particularly XML and HTML. Combined with a groups list, an opening page from the hypothetical AddressBook LassoApp could look as follows:

```
<html>
    <head><title>Title</title></head>
    <body>
        Users list:
        <?= lassoApp_include('/AddressBook/users.html') ?>
        Groups list:
        <?= lassoApp_include('/AddressBook/groups.html') ?>
    </body>
</html>
```

A **lassoApp_include** can be used to pull in any of the content representations for a file, including the primary content. If the raw user list (as shown earlier in this chapter) were desired, the **lassoApp_include** method would be used, but the ".lasso" extension would be given in the file path instead of the ".html" extension. Because of this, the return type of the **lassoApp_include** method may vary. It may be plain string data, bytes data from such as an image, or any other type of object.

The following example includes the users list and assigns it to a variable. It then prints a message pertaining to how many users exist. This illustrates how the result of **lassoApp_include** is not just character data, but is whatever type of data the LassoApp file represents. In this case, it is an array.

```
local(usersList) = lassoApp_include('/AddressBook/users.lasso')
'There are: ' + #usersList->size + ' users'
```

33.6 Packaging and Deploying LassoApps

A LassoApp can be packaged in one of three ways: as a directory of files, as a zipped directory, and as a compiled platform-specific binary. Each method has its own benefits. Developers can choose the packaging mechanism most suitable to their needs.

33.6.1 As a Directory

The first method is as a directory containing the application's files. This is the simplest method, requiring no extra work by the developer. The same directory used during development of the LassoApp can be moved to another machine running Lasso Server and run as-is. Of course, when using this method, the user has access to all the source code for the application. Generally, this packaging method would be used by an in-house application where source code availability is not a concern and the LassoApp is installed manually on a server by copying the LassoApp directory.

33.6.2 As a Zip File

The second method is to zip the LassoApp directory. This produces a single zip file that can be installed on a Lasso Server instance. Lasso Server will handle unzipping the file in-memory and serving its contents. LassoApps zipped in this manner provide easy downloading and distribution while still making the source code for the application accessible. Zipped LassoApps must have a ".zip" file extension.

Developers should verify that a LassoApp directory is zipped properly. Specifically, Lasso requires that all of the files and folders inside the LassoApp directory be zipped and not the LassoApp directory itself. On UNIX-based platforms (OS X or Linux) the **zip** command-line tool can be used to create zipped LassoApps. To accomplish this, a developer would **cd** *into* the LassoApp directory and issue the zip command. Assuming a LassoApp name of "AddressBook", the following command would be used.

```
$> zip -qr ../AddressBook.zip *
```

The above would zip the files and folders within the AddressBook directory and create a file named "AddressBook.zip" at the same level as the "AddressBook" directory. The "r" option indicates to zip that it should recursively zip all subdirectories, while the "q" option simply indicates that zip should do its job quietly (by default, zip outputs verbose information on its activities).

33.6.3 As a Compiled Binary

Using the **lassoc** tool, included with Lasso Server, a developer can compile a LassoApp directory into a single distributable file. LassoApps packaged in this manner will have the file extension ".lassoapp". Packaging in this manner provides the greatest security for one's source code because the source code is not included in the package and is not recoverable by the end user.

Compiled binary LassoApps are platform-specific. Because these LassoApps are compiled to native OS-specific executable code, a binary compiled for OS X, for example, will not run on Linux.

Both lassoc and the freely available **gcc** compiler tools are required to compile a binary LassoApp. Several steps are involved in this task. However, you can use a "makefile" which simplifies this process on Linux and OS X. To use this example makefile[49], copy the file into the same location as the LassoApp directory. Then, on the command line, type:

---

[49] http://source.lassosoft.com/svn/lasso/lasso9_source/trunk/makefile

```
$> make DirectoryName.lassoapp
```

Replace "DirectoryName" with the name of the LassoApp directory in the above command. The resulting file will have a ".lassoapp" extension and can be placed in the "LassoApps" directory. Lasso Server will load the LassoApp once it is restarted.

For information on compiling without using a makefile or on Windows, see the section **Compiling Lasso Code** in the **Command-Line Tools** chapter.

Installing the GCC compiler

On OS X, either:

- Install and open Xcode, then go to *Preferences* → *Downloads* → *Components* → *Command Line Tools*, and click *Install*.
- Or, install the Command Line Tools package directly from https://developer.apple.com/download/more/ (Apple ID required).

On CentOS:

- run **sudo yum install make** on the command line. This will install all required dependencies including **gcc**.

On Ubuntu:

- run **sudo apt-get install make** on the command line. As with CentOS this will install all required dependencies.

33.6.4 Platform-Specific Considerations

It is important to note that the target for each compiled LassoApp is specific to that which it is compiled on. If your development platform is OS X and you wish to deploy your compiled LassoApp on 64-bit CentOS, you must compile the LassoApp on a 64-bit CentOS machine. The same issue exists for 32-bit vs. 64-bit architectures on the same distribution. A LassoApp compiled for 32-bit Ubuntu will not run on 64-bit Ubuntu.

33.7 Server Configuration

Although LassoApps are available through the path **/lasso9/***AppName***/**, it is often desirable to dedicate a site to serving a single LassoApp. This can be accomplished by having the web server set an environment variable for Lasso to indicate which LassoApp the website is serving. The environment variable is named **LASSOSERVER_APP_PREFIX**. Its value should be the path to the root of the LassoApp. For example, if a site were dedicated to serving the Lasso Server Admin app, the value for the **LASSOSERVER_APP_PREFIX** variable would be "/lasso9/admin". Having the variable set in this manner would cause all **lassoApp_link** paths to be prefixed with "/lasso9/admin".

The **LASSOSERVER_APP_PREFIX** variable is used along with other web server configuration directives to provide transparent serving of a LassoApp. The following example for the Apache 2 web server illustrates how the Lasso Server Admin app would be served out of a virtual host named "admin.local".

```
<VirtualHost *:80>
    ServerName admin.local
    ScriptAliasMatch ^(.*)$ /lasso9/admin$1
    RewriteEngine on
    RewriteRule ^ - [E=LASSOSERVER_APP_PREFIX:/lasso9/admin]
</VirtualHost>
```

Consult your web server's documentation for further information.

33.8 LassoApp Tips

33.8.1 Loading Required Types/Traits/Methods at Initialization

It is a good habit to load all types and methods required by the LassoApp at the time Lasso Server loads it. This can be achieved by using "_init.lasso":

_init.lasso

```
// Load traits
lassoApp_include('core/traits/mytrait.lasso')
lassoApp_include('core/traits/anothertrait.lasso')

// Load types
local(coretypes) = array('my_usertype', 'my_addresstype', 'my_companytype')
with i in #coretypes
do lassoApp_include('core/types-methods/' + #i + '.lasso')
```

This will load the specified traits and types at the time the LassoApp is loaded. All documents in the LassoApp can then assume these types exist. Note that these types can later be redefined individually by accessing the URL directly; in this case, at *http://example.com/lasso9/myLassoApp/core/types-methods/my_usertype.lasso*.

33.8.2 Creating Required SQLite Database on Installation

It is often desirable to keep configuration data for your LassoApp in a database rather than a local config file. One method of storing this is to leverage Lasso Server's embedded SQLite data source.

The following code demonstrates automatically creating a SQLite database whenever the LassoApp is installed on a new instance:

_install.lasso

```
define myLassoApp_sqlite_dbname  => 'myLassoApp_db'
define myLassoApp_sqlite_db      => sys_databasesPath + myLassoApp_sqlite_dbname
define myLassoApp_config_table   => 'config'

local(sql) = sqlite_db(myLassoApp_sqlite_db)

#sql->doWithClose => {
    #sql->executeNow(
        "CREATE TABLE IF NOT EXISTS " + myLassoApp_config_table +
        " (host PRIMARY KEY, dbname, username, pwd, status INTEGER, registerkey);"
    )
}
```

The code within "_install.lasso" will only ever be executed when this LassoApp is first placed in the "LassoApps" directory of an instance and the instance is restarted.

33.8.3 Serving JSON and XHR Files

Content Representation can be leveraged to provide a range of data formats. One of these is **XHR** (XMLHttpRequest), also known as **AJAX** (Asynchronous JavaScript and XML), which in most cases will use a GET request to send data to the server, e.g. *http://example.com/lasso9/myLassoapp/userdata.xhr?id=123*.

While discussions directly regarding AJAX, jQuery, XHR, REST, XML, and JSON are outside the scope of this chapter, XHR response data can be in various forms, including JSON, which we will use for this example.

Consider the following JavaScript (using jQuery):

```
var dataObj       = new Object;
dataObj.id        = $('#userid').val();
$.ajax({
        url:         '/lasso9/myLassoapp/userdata.xhr',
        data:        dataObj,
        async:       true,
        type:        'post',
        cache:       false,
        dataType:    'json',
        success:     function(xhr) {
            alert('User name: ' + xhr.firstname + ' ' + xhr.lastname);
        }
});
```

The XHR request is for "userdata.xhr", which Lasso Server will interpret as a request for "userdata[xhr].lasso" and serve as an XHR file with the correct MIME type.

userdata[xhr].lasso

```
local(id)     = integer(web_request->param('id')->asString)
local(mydata) = map
inline(
    -database='example',
    -sql="SELECT firstname, lastname FROM mytable WHERE id = " + #id + " LIMIT 1;"
) => {
    records => {
        #mydata->insert('firstname' = field('firstname')->asString)
        #mydata->insert('lastname'  = field('lastname')->asString)
    }
}
local(xout) = json_serialize(#mydata)
#xout
```

Command-Line Tools

The Lasso platform comes with various command-line tools to assist you. Lasso uses some of these tools to create and start the instances of Lasso that run on the web server. This chapter will contain an overview of those tools and describe how you can run them yourself.

34.1 lassoserver

The **lassoserver** executable is installed at `/usr/sbin/lassoserver` on OS X and Linux operating systems and at `C:\Program Files\LassoSoft\Lasso Instance Manager\home\LassoExecutables\lassoserver` on Windows. This program creates a FastCGI server that interfaces with web servers to process Lasso files in response to web requests. Each instance of Lasso has its own lassoserver process running a FastCGI server. Additionally, the lassoserver executable can start up an HTTP server instead of a FastCGI server. As an HTTP server, it can serve both static files and Lasso files. This is useful for local development, though you should run a production web server (such as Apache) on any production servers.

The following is the list of options for running lassoserver:

-p `<tcp_listen_port>`
> Set the port that either the FastCGI or HTTP server binds on. This option is ignored if you choose to create a FastCGI socket.
>
> Default is 8999.

-addr `<tcp_bind_address>`
> Set the IP address to bind to when running as either a FastCGI or HTTP server. This option is ignored if you choose to create a FastCGI socket.
>
> Default is 0.0.0.0, which will bind to all IPs associated with your machine.

-fproxy `<fcgi_proxy_socket>`
> Specify the path to create a socket for FastCGI proxy requests to be sent to. This path will be relative to **LASSO9_HOME** unless you start the path with two slashes.
>
> Default is to not create this socket.

-flisten `<fcgi_listen_socket>`
> Specify the path to create a socket for FastCGI requests to be sent to. This path is always relative to **LASSO9_HOME**.
>
> Default is to not create this socket.

-user `<user>`
> Specifies the OS user to run lassoserver as. In order for this to be effective, you must be running lassoserver with root privileges.
>
> Default is to run as the user invoking lassoserver.

-group `<group>`
> Specify the OS group to run lassoserver as. In order for this to be effective, you must be running lassoserver with root privileges.

Default is to run as the primary group of the user invoking lassoserver.

-httproot `<path>`

This option tells lassoserver to start an HTTP server instead of a FastCGI server and to use the path specified as the web root. This option will be ignored if either **-fproxy** or **-flisten** is specified.

Default is to not start up as an HTTP server.

-scriptextensions `<ext1[;ext2] ... >`

Identify which file extensions should be considered Lasso files. This option is used in conjunction with **-httproot** to tell the HTTP server which files should be processed as Lasso code. Note that multiple extensions are delimited by semicolons.

Default is to not treat any files as Lasso code.

-addapp `<path>`

This option specifies a path to a LassoApp that is to be installed when lassoserver starts up. This allows you to include LassoApps that are outside the LassoApp directory in your instance home directory. This option can be specified multiple times with different paths and all specified LassoApps will be installed.

Default is to not install any additional LassoApps.

34.1.1 Starting lassoserver

To start lassoserver as a FastCGI server listening on port 9000:

```
$> lassoserver -p 9000
```

To start lassoserver as a FastCGI server listening on a socket at "$LASSO9_HOME/lasso.sock":

```
$> lassoserver -flisten lasso.sock
```

To start lassoserver as a FastCGI proxy server listening on a socket at "/tmp/lasso.sock":

```
$> lassoserver -fproxy //tmp/lasso.sock
```

To start lassoserver as an HTTP server that processes "*.lasso" and "*.inc" files as Lasso code:

```
$> lassoserver -httproot /path/to/webroot -scriptextensions "lasso;inc"
```

34.2 lassoim(d)

The **lassoim(d)** executable is installed at `/usr/sbin/lassoim` on OS X, `/usr/sbin/lassoimd` on Linux operating systems, and `C:\Program Files\LassoSoft\Lasso Instance Manager\home\LassoExecutables\lassoim` on Windows. This program creates the FastCGI server that runs Lasso's Instance Manager web application. It also makes sure that all enabled instances are running.

To manually start lassoim(d), just call it from the command line. (It ignores any arguments passed to it.)

```
$> lassoim
```

When running this executable, it is important to set the **LASSO9_HOME** environment variable to a path of a directory containing all the built-in Lasso libraries. By default, this should be `/var/lasso/home` on OS X and Linux operating systems.

34.3 lasso9

The **lasso9** executable is installed at `/usr/bin/lasso9` on OS X and Linux operating systems and at `C:\Program Files\LassoSoft\Lasso Instance Manager\home\LassoExecutables\lasso9` on Windows. This program can execute Lasso code from a file, piped from STDIN, passed in as a string, or inside an interactive interpreter. This executable doesn't load and start up everything that **lassoserver** does. See the section *Loading Libraries in Shell Scripts* for what isn't loaded and how to load the extra components if you need them.

To execute a file of Lasso code, pass the path to the file as the argument to lasso9. For example:

```
$> lasso9 /path/to/code.lasso
```

-s <code>
> Use **-s** to execute the string passed to lasso9 as Lasso code:
>
> ```
> $> lasso9 -s "lasso_version"
> ```

--
> Use **--** to execute Lasso code from STDIN:
>
> ```
> $> echo 'lasso_version' | lasso9 --
> ```

-i
> Use **-i** to execute Lasso code interactively. When you do this a new prompt will appear (**>:**), and what you type there will be processed as Lasso code when you hit **return**. You can also paste small amounts of multi-line code into the prompt; just be sure to hit **return** right after pasting so that the last line of code will be included. When finished, type **Control-C** to exit.

```
$> lasso9 -i
>: lasso_version
Mac OS X 9.2
>: loop(3) => { stdoutnl(loop_count) }
1
2
3
```

Note: Each chunk of code is processed as if it were a separate file, so local variables processed in one chunk are unavailable to future chunks. You'll either need to copy and paste multi-line code, or use thread variables.

For more details, see the section *Calling Lasso from the CLI* in the *Calling Lasso* chapter.

34.4 lassoc

The **lassoc** executable is installed at `/usr/bin/lassoc` on OS X and Linux operating systems and at `C:\Program Files\LassoSoft\Lasso Instance Manager\home\LassoExecutables\lassoc` on Windows. This program is used to compile LassoApps, Lasso libraries, and Lasso executables. See the section *Compiling Lasso Code* below for more information.

34.5 Special Environment Variables

There are several environment variables that have various effects on running **lasso9**, **lassoserver**, or custom Lasso executables. The following lists the variables and a description of their function:

LASSO9_HOME

This variable is set to the path of a directory containing either the instance-specific libraries and startup items, or to a path containing all of the Lasso built-in libraries. If set to an instance-specific home directory, be sure to also set the **LASSO9_MASTER_HOME** variable.

Default is **/var/lasso/home** for OS X and Linux.

LASSO9_MASTER_HOME

This variable must be set to a directory containing all the built-in Lasso libraries if the **LASSO9_HOME** variable is set to an instance-specific home directory.

Default is not set.

LASSO9_PRINT_FAILURES

This variable can be set to an integer that specifies how verbose a Lasso executable should be in its error reporting. Setting it to "1" outputs the most information, with larger integer values making it less verbose.

Default is not set, which is the least verbose.

LASSO9_RETAIN_COMMENTS

If this variable is set to "1", Lasso will retain any doc comments in the code it loads, allowing you to programmatically view and process these comments.

Default is not set.

LASSO9_PRINT_LIB_LOADS

If this variable is set to "1", Lasso will print diagnostic information to STDOUT regarding the on-demand libraries that it loads. This can be useful when debugging your own on-demand Lasso libraries.

Default is not set.

LASSOSERVER_APP_PREFIX

If this variable is set by the web server, lassoserver will assume the host is dedicated to serving a single LassoApp, and will prepend this path to all **lassoApp_link** paths. For details and an example, see the section *Server Configuration* in the *LassoApps* chapter.

Default is not set.

LASSOSERVER_DOCUMENT_ROOT

If this variable is set by the web server, lassoserver will use this path instead of the standard **DOCUMENT_ROOT** to serve files from. This can be useful when using Apache's **VirtualDocumentRoot** or **UserDir** features. In the example below, Apache will serve any of the folder names in "/srv/lasso/sites/" as virtual hosts, and Lasso will use the value of **LASSOSERVER_DOCUMENT_ROOT** as each host's document root.

```
<VirtualHost *:80>
    ServerName admin.local
    VirtualDocumentRoot "/srv/lasso/sites/%1"
    RewriteEngine on
    RewriteRule ^ - [E=LASSOSERVER_DOCUMENT_ROOT:/srv/lasso/sites/%{HTTP_HOST}]
</VirtualHost>
```

Default is not set.

LASSOSERVER_FASTCGIPORT

Set the port that the FastCGI server binds on. Same as specifying the **-p** option.

LASSOSERVER_USER

Specifies the OS user to run lassoserver as. Same as specifying the **-user** option.

LASSOSERVER_GROUP

Specifies the OS group to run lassoserver as. Same as specifying the **-group** option.

34.6 Lasso Shell Scripts on OS X and Linux

While most developers use Lasso to create dynamic websites, you can also write Lasso code that can be run from the command line to assist you in administrative or repetitive tasks. These files that run from the command line are often called *shell scripts* since you run them from your terminal's shell.

34.6.1 Running Scripts

There are two ways to run a file containing Lasso code from the command line:

- Pass the path of the file to the **lasso9** executable:

      $> lasso9 /path/to/code.lasso

- Make sure the file has execute permissions turned on and that it starts with the proper hashbang, then call the file directly:

      $> /path/to/code.lasso

This second option requires having the file's executable permissions set. You can do this in OS X or Linux with the **chmod** command:

```
$> chmod +x /path/to/code.lasso
```

Calling the file directly also requires that the file contain the proper hashbang, which tells your shell which interpreter to use when executing the file. It must be the first line of the file and it starts with the pound sign and an exclamation mark followed by the path to the interpreter. For Lasso code, it should look like this:

```
#!/usr/bin/env lasso9
```

If you have a custom installation of Lasso, adjust the path to the lasso9 executable accordingly.

34.6.2 Reading Command-Line Arguments

When running Lasso shell scripts, Lasso provides two special thread variables for inspecting the command that was run and the arguments that were passed to it: "argc" and "argv". The "argc" variable returns the number of arguments, including the command. The "argv" variable returns a staticarray in which the first element is the command and the remaining elements are the arguments passed to the command.

The following example outputs the values of **$argc** and **$argv** when the script is run using the lasso9 tool. The contents of the file "/path/to/code.lasso" are:

```
stdoutnl($argc)
stdoutnl($argv)
```

Here's what happens when you run the code:

```
$> lasso9 /path/to/code.lasso -moose hair
3
staticarray(/path/to/code.lasso, -moose, hair)
```

The following example shows the values of **$argc** and **$argv** when the script is run directly. The contents of the file "/path/to/code.lasso" are:

```
#!/usr/bin/env lasso9
stdoutnl($argc)
stdoutnl($argv)
```

Here's what happens when you run the script directly:

```
$> /path/to/code.lasso -moose hair
3
staticarray(/path/to/code.lasso, -moose, hair)
```

As you can see, calling the script with lasso9 produces the same result as calling the script directly, so you don't ever need to worry about the first element in **$argv** being "lasso9".

Using these two thread variables, you can create scripts whose behavior changes when different arguments are passed to them. In fact, the lasso9 executable itself is a Lasso shell script (source[50]), written in Lasso and compiled into a binary.

34.7 Loading Libraries in Shell Scripts

Lasso shell scripts are not run in the **lassoserver** context. This means that various libraries and tools that lassoserver loads are not loaded or available by default when your script runs. Although all the core libraries are available, the LCAPI modules, LJAPI modules, logging system, email queue, security registry, web request and response environment, LassoApps, and files in "LassoStartup" are not loaded. This is actually beneficial since your script would otherwise take as long as lassoserver to start up before getting to running your code. If you find you need something that isn't loaded, you can load it yourself. The sections below will show you how.

34.7.1 Load All Database and LCAPI Modules

If you want to have access to all database connectors and to all the LCAPI modules such as the ImageMagick methods or the **os_process** type, you can load them all with the **database_initialize** method:

```
#!/usr/bin/env lasso9
database_initialize
```

34.7.2 Load Specific LCAPI Modules

If you want, you can just load individual LCAPI modules. The following example loads just the MySQL database connector:

```
#!/usr/bin/env lasso9
// If LASSO9_MASTER_HOME is specified, find module there
// Otherwise, find it in the LASSO9_HOME path
lcapi_loadModule((sys_masterHomePath || sys_homePath) + '/LassoModules/MySQLConnector.' + sys_dll_ext)
```

34.7.3 Set Up the LJAPI Environment

To create the JVM and set up the LJAPI environment, you must first load the LJAPI9 LCAPI module and then call the **ljapi_initialize** method:

---

[50] http://source.lassosoft.com/svn/lasso/lasso9_source/trunk/lasso9.lasso

```
#!/usr/bin/env lasso9
match(lasso_version(-lassoplatform)) => {
    case('Linux')
        lcapi_loadModule((sys_masterHomePath || sys_homePath) + '/LassoModules/LJAPI.so')
    case('Mac OS X')
        lcapi_loadModule((sys_masterHomePath || sys_homePath) + '/LassoModules/LJAPI9.dylib')
    // Fail if unknown OS
    case
        fail('Unknown platform')
    }
ljapi_initialize
```

34.7.4 Load a LassoApp

LassoApps have the ability to run or load code when they are initialized. Often this code adds methods, types, or traits that you may want available in your Lasso shell scripts. The code below contains three examples of loading up LassoApps: one for compiled LassoApps, one for zipped LassoApps, and one for a LassoApp directory.

```
#!/usr/bin/env lasso9
// Load a compiled LassoApp from LASSO9_MASTER_HOME if specified
// Otherwise, load it from LASSO9_HOME
lassoapp_installer->install(
    lassoapp_compiledsrc_appsource(
        (sys_masterHomePath || sys_homePath) +
        '/LassoApps/example.lassoapp'
    )
)

// Load a zipped LassoApp from LASSO9_HOME
lassoapp_installer->install(
    lassoapp_zipsrc_appsource(sys_appsPath + 'example.zip')
)

// Load a LassoApp from the specified directory
lassoapp_installer->install(
    lassoapp_dirsrc_appsource('//path/to/example/')
)
```

34.7.5 Include Another File with Lasso Code

If you would like to run Lasso code in another file from your script, you can include that file using the **sourcefile** type. The following example will have "/path/to/code.lasso" running the code from "/path/to/doc.lasso":

```
// Contents of /path/to/code.lasso
local(doc) = sourcefile(file('//path/to/doc.lasso'))
stdoutnl("Calling " + #doc->filename + "...")
#doc->invoke
stdoutnl("This is heavy.")

// Contents of /path/to/doc.lasso
stdoutnl("Great Scott!")
```

Here's what happens when you run "/path/to/code.lasso":

```
$> lasso9 /path/to/code.lasso
Calling //path/to/doc.lasso...
Great Scott!
This is heavy.
```

34.7.6 Include Another File Relative to the Script

Sometimes it's helpful to have the script you are running able to include a file that is relative to the script. If you pass a relative path to the **file** type, it will expect the file you are trying to reference to be included relative from your shell's current working directory. To get around this, you must have the current script figure out the absolute path to its parent directory so you can append the relative path. The following code does just that:

```
#!/usr/bin/env lasso9
// Contents of /path/to/project/sub1/code.lasso

// This should let us run this file anywhere and still properly import relative files
local(path_here) = currentCapture->callsite_file->stripLastComponent
not #path_here->beginsWith('/') ?
    #path_here = io_file_getcwd + '/' + #path_here
not #path_here->endsWith('/') ?
    #path_here->append('/')
local(f) = file(#path_here + '../sub2/code.lasso')

stdoutnl("Loading ../sub2/code.lasso")
sourcefile(#f)->invoke
stdoutnl("Done.")


// Contents of /path/to/project/sub2/code.lasso
stdoutnl("I am a relative include.")
```

Here's what happens when you run "/path/to/project/sub1/code.lasso":

```
$> /path/to/project/sub1/code.lasso
Loading ../sub2/code.lasso
I am a relative include.
Done
```

34.7.7 Change the Working Directory

Occasionally you may find it helpful to change the directory context your script is running in. You can use the **dir->setcwd** method to do so:

```
#!/usr/bin/env lasso9
// Contents of /path/to/code.lasso

stdoutnl("We are here: " + io_file_getcwd)
dir('/etc/')->setcwd
stdoutnl("Now we are here: " + io_file_getcwd)
```

Here's what happens when you run this file:

```
$> cd /path/to/
$> lasso9 ./code.lasso
```

```
We are here: /path/to
Now we are here: /etc
```

34.7.8 Read and Set Environment Variables

Lasso can read and set shell environment variables using **sys_getEnv** and **sys_setEnv** respectively. The following example adds a directory to the "PATH" environment variable for the script:

```
#!/usr/bin/env lasso9
// Contents of /path/to/code.lasso

// Ignore the return value of sys_setEnv
local(_) = sys_setEnv(`PATH`, `/var/lasso/home/bin:` + sys_getEnv(`PATH`))
stdoutnl(sys_getEnv(`PATH`))
```

Here's what happens when you run this script:

```
$> /path/to/code.lasso
/var/lasso/home/bin:/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin
```

34.8 Compiling Lasso Code

All Lasso code is compiled before it is executed. Whether the code is a Lasso page being served by Lasso Server or a script being run by the **lasso9** command-line tool, behind the scenes Lasso compiles the code and then executes the compiled code. (Lasso does cache the compiled code for re-use, but that is beyond the scope of this section.)

There are certain cases where it is advantageous to compile the Lasso code ahead of time. The Lasso platform comes with the **lassoc** command-line tool which aids in compiling LassoApps, Lasso libraries, and Lasso executables. Compilation can result in faster startup times, lower memory usage, and obfuscation of the source code.

Libraries help keep memory usage down because only objects that are actually used are loaded. They also improve startup time. Lasso can start up by only loading the very basic built-in functions and objects and then let the rest of the system load in over time.

A special type of library called a *bitcode* file can also be produced, which has a ".bc" file extension. Bitcode is an LLVM-specific format that Lasso knows how to load. Bitcode files can be shared across platforms on the same processor. For example, the same bitcode file could be used on OS X x86 and CentOS x86. Bitcode files don't load as fast, have about 80% larger file size and consume more memory than library files compiled into a shared library, but they don't require GCC and are cross-platform.

34.8.1 Prerequisites

The following must be installed to compile Lasso code:

- Lasso Server installed on a supported OS

- Your operating systems's developer command-line tools. (Consult the documentation for your OS on how to install a compiler, linker, etc.)

- For OS X, you will also need the 10.5 SDK libraries in order to create binaries that are compatible with all supported versions of OS X. See this link for unsupported help with installing older SDKs[51].

The examples below are shown running from a command-line prompt. For Windows, make sure you are running these commands from the Visual Studio command prompt.

---

[51] http://hints.macworld.com/article.php?story=20110318050811544

34.8.2 Compiling Executables

You can compile shell scripts into executable files. This decreases the overhead of running the script through the **lasso9** tool, and allows you to distribute your own command-line tools without distributing the source code. The examples below take a shell script named "myscript.lasso" and compile it into the executable "myscript".

OS X

```
$> lassoc -O -app -n -obj -o myscript.a.o myscript.lasso
$> gcc -o myscript myscript.a.o -isysroot /Developer/SDKs/MacOSX10.5.sdk \
-Wl,-syslibroot,/Developer/SDKs/MacOSX10.5.sdk -mmacosx-version-min=10.5 \
-macosx_version_min=10.5 -F/Library/Frameworks -framework Lasso9
```

Linux

```
$> lassoc -O -app -n -obj -o myscript.a.o myscript.lasso
$> gcc -o myscript myscript.a.o -llasso9_runtime
```

Windows

```
$> lassoc -O -app -n -obj -o myscript.obj myscript.lasso
$> link myscript.obj \
> /LIBPATH:"C:\Program Files\LassoSoft\Lasso Instance Manager\home\LassoExecutables" \
> lasso9_runtime.lib -defaultlib:libcmt
```

34.8.3 Compiling Libraries

You can create your own library of methods and types and then compile it into one library file for distribution. Libraries compiled this way go into the "LassoLibraries" directory of an instance's **LASSO9_HOME** or **LASSO9_MASTER_HOME** directory. The advantages of doing this instead of placing the source code in the "LassoStartup" directory are that Lasso starts faster and consumes less memory. This is because Lasso only loads the methods and types in libraries when they are first used instead of at startup. This makes starting an instance of Lasso Server faster as the code will be loaded when first needed, and it helps keep memory down as only those methods and types that are actually used by the instance get loaded.

The examples below take a file named "mylibs.inc" and compile it into a dynamically loaded Lasso library.

OS X

```
$> lassoc -O -dll -n -obj -o mylibs.d.o mylibs.inc
$> gcc -dynamiclib -o mylibs.dylib mylibs.d.o -isysroot /Developer/SDKs/MacOSX10.5.sdk \
-Wl,-syslibroot,/Developer/SDKs/MacOSX10.5.sdk -mmacosx-version-min=10.5 \
-macosx_version_min=10.5 -F/Library/Frameworks -framework Lasso9
```

Linux

```
$> lassoc -O -dll -n -obj -o mylibs.d.o mylibs.inc
$> gcc -shared -o mylibs.so mylibs.d.o -llasso9_runtime
```

```
$> lassoc -O -dll -n -obj -o mylibs.obj mylibs.inc
$> link /DLL mylibs.obj /OUT:mylibs.dll \
/LIBPATH:"C:\Program Files\LassoSoft\Lasso Instance Manager\home\LassoExecutables" \
lasso9_runtime.lib -defaultlib:libcmt
```

34.8.4 Compiling LassoApps

LassoApps allow you to create an easily deployable and distributable web application. They are installed into the "LassoApps" directory of an instance's **LASSO9_HOME** or **LASSO9_MASTER_HOME** directory. (See the *LassoApps* chapter for more information.) Compiling them allows Lasso to start up faster and allows for distributing closed-sourced solutions.

The examples below take a folder named "myapp" and compile it into a LassoApp named "myapp.lassoapp".

OS X

```
$> lassoc -O -dll -n -obj -lassoapp -o myapp.ap.o myapp/
$> gcc -dynamiclib -o myapp.lassoapp myapp.ap.o -isysroot /Developer/SDKs/MacOSX10.5.sdk \
-Wl,-syslibroot,/Developer/SDKs/MacOSX10.5.sdk -mmacosx-version-min=10.5 \
-macosx_version_min=10.5 -F/Library/Frameworks -framework Lasso9
```

Linux

```
$> lassoc -O -dll -n -obj -lassoapp -o myapp.ap.o myapp/
$> gcc -shared -o myapp.lassoapp myapp.ap.o -llasso9_runtime
```

Windows

```
$> lassoc -O -dll -n -obj -lassoapp -o myapp.lassoapp.obj myapp
$> link /DLL myapp.lassoapp.obj /OUT:myapp.lassoapp \
/LIBPATH:"C:\Program Files\LassoSoft\Lasso Instance Manager\home\LassoExecutables" \
lasso9_runtime.lib -defaultlib:libcmt
```

34.8.5 Using Build Utilities

Instead of manually executing those commands each time you want to compile your code, it is recommended you use a build utility like **make** for OS X and Linux or **nmake** for Windows. Both of these utilities are very powerful and you should explore their documentation. The Lasso source tree has an example of both a make file[52] and an nmake file[53] which you can download and modify to fit your solutions.

---

[52] http://source.lassosoft.com/svn/lasso/lasso9_source/trunk/makefile
[53] http://source.lassosoft.com/svn/lasso/lasso9_source/trunk/makefile.nmake

Part VI

External Communication

Network Requests with Curl

Lasso provides a complete interface to the open source curl library[54]. Curl transfers data with URL syntax, supporting a wide variety of protocols such as "DICT", "FILE", "FTP", "FTPS", "Gopher", "HTTP", "HTTPS", "IMAP", "IMAPS", "LDAP", "LDAPS", "POP3", POP3S, "RTMP", "RTSP", "SCP", "SFTP", "SMTP", "SMTPS", "Telnet" and "TFTP".

Curl has built-in support for SSL certificates, HTTP POST, HTTP PUT, FTP uploading, proxies, cookies, user+password authentication, proxy tunnelling, and more.

35.1 Lasso Curl API

type **curl**

curl()

curl(url::string, -username::string=?, -password::string=?**)**
> There are two **curl** creator methods. The first creates an empty curl object. The second takes a string representing the URL to be eventually called, and it optionally takes a username and password to be used for authentication.

curl->url()
> Returns the current URL set for the curl object.

curl->url=(s::string**)**
> Sets the URL for the current curl object.

curl->postFields=(s::string**)**

curl->postFields=(b::bytes**)**
> Sets the full data to post in an HTTP POST operation. You must make sure that the data is formatted the way you want the server to receive it. The curl object will not convert or encode it. Most web servers will assume this data will be URL encoded.

> Use the method taking a bytes object in order to have control over the encoding of the data to be sent to the destination server. An example of this would be sending a binary image file.

curl->contentType=(s::string**)**
> Override the default HTTP *Content-Type* header by setting this value.

curl->close()
> Close the current curl object.

curl->asString()
> Return the result of performing the current curl object's action as a string. If no URL is set, it will just return an empty string.

curl->asBytes()
> Returns the result of performing the current curl object's action as bytes.

---

[54] https://curl.haxx.se/

`curl->`**`done()`**
> Returns "true" or "false", indicating the completion state of the current curl operation.

`curl->`**`get(`**_key_**`)`**
> Request internal information from the curl session. The key should be one of the `CURLINFO_...` methods.

`curl->`**`set(`**_key_, _value_**`)`**
> Used to set specific **`curl`** option behavior. The key should be one of the `CURLOPT_...` methods. These options and appropriate values can be reviewed in the curl documentation[55].

`curl->`**`header()`**
> Returns the header data as a bytes object for the current curl request.

`curl->`**`result()`**
> Returns the result of performing the current curl object's action as bytes. (For HTTP requests, it just returns the body portion, not the headers.)

`curl->`**`statusCode()`**
> Return the last received HTTP, FTP, or SMTP response code. The value will be zero if no server response code has been received.

`curl->`**`raw()`**
> Returns the result of performing the current curl object request as a staticarray containing the ready state (boolean), the header response (bytes), and the body response (bytes).

`curl->`**`reset()`**
> Resets the current curl object to an empty curl object.

`curl->`**`version(`**_info=?_**`)`**
> Returns a string of the version of curl currently deployed on the host system. If the optional **`info`** parameter is supplied as "true", then more detailed information will be returned as a staticarray.

`curl->`**`readSomeBytes()`**
> This is a low-level function and is not recommended to be for casual use. If a request is still in progress, it returns the current response as a bytes object and clears the internal mechanism that is buffering those bytes.

`curl->`**`download(`**_f::string=?_, _-asBytes::boolean=?_**`)`**
> Triggers the download of the file specified by the URL. The default is to download the file to the path specified in the first optional parameter. If the **`-asBytes`** option is passed or set to "true", then it will just return a bytes object representing the file's data. Refer to the detailed documentation later in this chapter for example usage.

`curl->`**`upload(`**_f::string_**`)`**

`curl->`**`upload(`**_f::file_**`)`**

`curl->`**`upload(`**_f::bytes_**`)`**
> Triggers the uploading of a specified file to the file location specified by the URL. The file to be uploaded can be specified as either a string of the file path and name, a file object, or a bytes object of the data. Refer to the detailed documentation later in this chapter for example usage.

`curl->`**`ftpDeleteFile()`**
> Deletes the file specified by the URL from the FTP server.

`curl->`**`ftpGetListing(`**_-listOnly::boolean=?_, _-options::array=?_**`)`**
> Retrieves the directory listing from the FTP server and directory path specified by the URL. If the **`-listOnly`** option is specified, the result will just be returned as a staticarray while the default is to return an array of maps with each map having the following data about the files: "filetype", "filesize", "filemoddate", and "filename".
>
> There is an optional **`-options`** parameter that can take an array of pairs specifying additional **`curl`** options. The first item in the pair should be one of the `CURLOPT_...` methods and the second should be the corresponding value you

---

[55] https://curl.haxx.se/libcurl/c/curl_easy_setopt.html

wish to set.

35.2 Curl Options

A myriad of **curl** options can be set for the current curl object to customize its behavior. This can be done by using the **curl->set** method, passing it the **CURLOPT_...** macro methods representing the option you wish to set and the value you wish to set it to as the second param. What follows is a list of options that can be set on Lasso's curl object. It has been adapted from the curl documentation[56], with the options grouped in a similar manner as is found there. This should allow you to easily find the option if you need more detail.

35.2.1 Behavior Options

CURLOPT_VERBOSE()
>   Used with **curl->set**. If set to "1", it directs curl to output a lot of verbose information about its operations. This is very useful for debugging. The verbose information will be sent to STDERR which gets logged to **lasso.err.txt** in your instance's home directory for Lasso Server. You will almost never want to set this in production, but you will want to use it to help you debug and report problems.

CURLOPT_HEADER()
>   Used with **curl->set**. If set to "1", it directs curl to include the header in the body output. This is only relevant for protocols that actually have headers preceding the data (like HTTP).

CURLOPT_NOPROGRESS()
>   Used with **curl->set**. If set to "1", it directs curl to shut off the progress meter completely. It will also prevent **CUR-LOPT_PROGRESSFUNCTION** from being called. Future versions of libcurl are likely not to have any built-in progress meter at all.

35.2.2 Callback Options

CURLOPT_WRITEDATA()
>   Used with **curl->set**. This option expects a **filedesc** object which curl will use when calling its file writing function.

CURLOPT_READDATA()
>   Used with **curl->set**. This option expects either a **filedesc** or bytes object to be used when curl calls its file reading function.

35.2.3 Error Options

CURLOPT_FAILONERROR()
>   Used with **curl->set**. If set to a value of "1", curl should fail silently if the HTTP status code is equal to or larger than 400. The default action would be to return the page normally, ignoring that code. This method is not fail-safe, and there are scenarios where unsuccessful response codes will slip through.

35.2.4 Network Options

CURLOPT_URL()
>   Used with **curl->set**. You can use this instead of **curl->url=** to change the URL for the curl object. All URLs should be in the general form of "scheme://host:port/path" as detailed in RFC 3986[57].

---

[56] https://curl.haxx.se/libcurl/c/curl_easy_setopt.html
[57] https://tools.ietf.org/html/rfc3986.html

CURLOPT_PROXY()
> Used with **curl->set**. Sets the HTTP proxy to use for the current curl object's request. This value should be passed as a string.

CURLOPT_PROXYPORT()
> Used with **curl->set**. Sets the proxy port to connect to unless it is specified in the proxy string set with **CURLOPT_PROXY**. This value should be an integer.

CURLOPT_PROXYTYPE()
> Used with **curl->set**. Sets type of the proxy. The value passed should be one of the following methods:

> **CURLPROXY_HTTP()**

> **CURLPROXY_SOCKS4()**

> **CURLPROXY_SOCKS5()**

CURLOPT_HTTPPROXYTUNNEL()
> Used with **curl->set**. If set to a value of "1", curl will tunnel all operations through a given HTTP proxy. This is different from simply using a proxy.

CURLOPT_INTERFACE()
> Used with **curl->set**. Sets the interface name to use as the outgoing network interface. The name can be an interface name, an IP address, or a host name. This value should be passed as a string.

CURLOPT_BUFFERSIZE()
> Used with **curl->set**. Pass an integer that will be used to indicate your preferred size (in bytes) for the receive buffer used by curl. This is just a request to the library; the actual buffer size used may be different than your request.

CURLOPT_PORT()
> Used with **curl->set**. Specifies which remote port number to connect to instead of the one specified in the URL, or specifies the default port for the protocol used. This value should be an integer.

CURLOPT_TCP_NODELAY()
> Used with **curl->set**. Specifies whether the **TCP_NODELAY** option is to be set or cleared (1 = set, 0 = clear). The option is cleared by default. Setting this option after the connection has been established will have no effect.

35.2.5 Authentication Options

CURLOPT_NETRC()
> Used with **curl->set**. This option controls the preference of curl between using user names and passwords from your **~/.netrc** file, relative to user names and passwords in the URL. The value passed should be one of the following methods:

> **CURL_NETRC_OPTIONAL()**
> > The use of your **~/.netrc** file is optional, and information in the URL is to be preferred.

> **CURL_NETRC_IGNORED()**
> > Curl will ignore the **~/.netrc** file and use only the information in the URL.

> **CURL_NETRC_REQUIRED()**
> > The use of your **~/.netrc** file is required, and curl should ignore the information in the URL.

CURLOPT_NETRC_FILE()
> Used with **curl->set**. Set to a string containing the full path name to the file you want libcurl to use as the **.netrc** file. If this option is omitted and **CURLOPT_NETRC** is set to use a **.netrc** file then curl will attempt to find a **.netrc** file in the current user's home directory.

CURLOPT_USERPWD()
> Used with **curl->set**. The option expects a string that will be used to authenticate with the remote server. The string should be formatted to include both username and password in the following manner: **'myname:mypassword'**.

CURLOPT_PROXYUSERPWD()
> Used with **curl->set**. This option expects a string that specifies the authentication for the HTTP proxy in the format of `'username:password'`. Use **CURLOPT_PROXYAUTH** to specify the authentication method.

CURLOPT_HTTPAUTH()
> Used with **curl->set**. Use this option to specify which HTTP authentication method(s) you want curl to use. If you specify more than one method, curl will first query the server to see which methods it supports and pick the best one you allow it to use.
>
> The value passed can be either of the following methods:
>
> > **CURLAUTH_ANY()**
> > > Allows any authentication method.
> >
> > **CURLAUTH_ANYSAFE()**
> > > Allows any authentication method except **CURLAUTH_BASIC**.
>
> Or, one or more of the following methods added together can be specified:
>
> > **CURLAUTH_BASIC()**
> >
> > **CURLAUTH_DIGEST()**
> >
> > **CURLAUTH_GSSNEGOTIATE()**
> >
> > **CURLAUTH_NTLM()**

CURLOPT_PROXYAUTH()
> Used with **curl->set**. Use this option to specify which HTTP authentication method(s) you want curl to use. See **CURLOPT_HTTPAUTH** for a list of values for this option.

35.2.6 HTTP Options

CURLOPT_ENCODING()
> Used with **curl->set**. This option takes a string value specifying the *Accept-Encoding* header which also enables decoding of a response when a *Content-Encoding* header is received. The string value passed should be one of the following: "identity", which does nothing; "deflate", which requests the server to compress its response using the zlib algorithm; or "gzip", which requests the gzip algorithm.

CURLOPT_AUTOREFERER()
> Used with **curl->set**. If set to "1", then curl will set the *Referer* header when it follows a *Location* redirect.

CURLOPT_FOLLOWLOCATION()
> Used with **curl->set**. If set to "1", then curl will follow any *Location* header the server sends as part of its HTTP response. This means that curl will send the same request to the new location and follow any new *Location* headers all the way until no more such headers are returned. **CURLOPT_MAXREDIRS** can be used to limit the number of redirects curl will follow.

CURLOPT_UNRESTRICTED_AUTH()
> Used with **curl->set**. If set to "1", then curl will continue to send authentication (username+password) when following locations, even if the hostname changes. (This option is meaningful only when setting **CURLOPT_FOLLOWLOCATION**.)

CURLOPT_MAXREDIRS()
> Used with **curl->set**. Expects an integer value specifying the number of times curl will repeat the recursive following of the *Location* header. A value of "0" will mean that no redirects will be followed while a value of "-1" (the default) means that an infinite number of redirects will be followed.

CURLOPT_PUT()
> *Deprecated since version 7.12.1:* This option is deprecated in curl in favor of using **CURLOPT_UPLOAD**.

Used with **curl->set**. If set to "1", then curl will use the HTTP PUT method to transfer data. The data should be set with **CURLOPT_READDATA** and **CURLOPT_INFILESIZE**.

CURLOPT_POST()

Used with **curl->set**. If set to "1", then curl will use the HTTP POST method for its request. This will also have the request use a *Content-Type: application/x-www-form-urlencoded* header (by far the most commonly used *Content-Type* for the POST method). You can override this header by setting your own with **CURLOPT_HTTPHEADER**.

Use **CURLOPT_POSTFIELDS** to specify what data to post in the request and **CURLOPT_POSTFIELDSIZE** or **CURLOPT_POSTFIELDSIZE_LARGE** to set the data size.

CURLOPT_POSTFIELDS()

Used with **curl->set**. You can use this instead of **curl->postFields=** to specify the data to post in an HTTP POST operation. The value can be either bytes or a string. You must make sure that the data is formatted the way you want the server to receive it; curl will not convert or encode it for you. Most web servers will assume this data will be URL encoded.

Using **CURLOPT_POSTFIELDS** implies **CURLOPT_POST**; that option will be automatically set along with all of its other side effects.

If you want to do a zero-byte POST, set **CURLOPT_POSTFIELDSIZE** explicitly to zero. Simply setting **CURLOPT_POSTFIELDS** to "null" or an empty string effectively disables the sending of the specified string, and curl will instead assume that you'll send the POST data using the **read** callback.

CURLOPT_POSTFIELDSIZE()

Used with **curl->set**. By default, curl will use **strlen()** (the C function for getting a string's length) to measure the size of the post data field being sent. This option allows you to pass an integer value specifying the size of the post field data. Generally speaking, posting binary data will require you to set this option.

CURLOPT_POSTFIELDSIZE_LARGE()

Used with **curl->set**. This is the large file version of **CURLOPT_POSTFIELDSIZE**.

CURLOPT_REFERER()

Used with **curl->set**. This option takes a string value specifying the value for the *Referer* header in the HTTP request sent to the remote server.

CURLOPT_USERAGENT()

Used with **curl->set**. This option takes a string value specifying the value for the *User-Agent* header in the HTTP request sent to the remote server.

CURLOPT_HTTPHEADER()

Used with **curl->set**. This option allows for adding new headers, replacing automatically generated internal headers, and removing automatically generated internal headers. The value passed should be an array of pairs with the first element in the pair being the string value of the header and the second value being the data to set it to. Header values specified here will override any automatically generated headers of the same name. Setting the value to an empty string will remove the header from the request.

CURLOPT_HTTP200ALIASES()

Used with **curl->set**. Some server responses use a custom response status line. For example, IceCast servers respond with "ICY 200 OK". This option allows you to specify that response is the same as "HTTP/1.0 200 OK". The value passed should be an array of strings, each string specifying another alias for the success status.

CURLOPT_COOKIE()

Used with **curl->set**. This option expects a string value that sets the cookie value for the HTTP header. The format of the string should be *NAME=CONTENTS*, where "NAME" is the cookie name and "CONTENTS" is what the cookie should contain. To send multiple cookies, separate each cookie in the string with a semicolon and a space like this: **'name1=content1; name2=content2;'**. Using this option multiple times will only make the latest string override the previous ones.

CURLOPT_COOKIEFILE()

> Used with **curl->set**. This option takes a string value for the path to and name of a file holding cookie data to read and send with the request. The cookie data may be in Netscape/Mozilla cookie data format or just regular HTTP-style headers dumped to a file.

CURLOPT_COOKIEJAR()

> Used with **curl->set**. This option takes a string value specifying the path and file name for curl to store cookies in. If the file can't be created, no error will be reported. (Using **CURLOPT_VERBOSE** will have a warning printed, but this is the only way to get this feedback.)

CURLOPT_COOKIESESSION()

> Used with **curl->set**. If set to "1", curl will not use any session cookies that had been previously set by requests in the session. (Session cookies are cookies without expiry date and they are meant to be alive and existing for this "session" only.)

CURLOPT_HTTPGET()

> Used with **curl->set**. If set to "1", it will force the curl request to use the HTTP GET method. Useful if an HTTP POST, PUT, or HEAD request had been set.

CURLOPT_HTTP_VERSION()

> Used with **curl->set**. This option forces curl to use a specific HTTP version. (This is not recommended unless you have a good reason.) The value passed should be one of the following methods:

> **CURL_HTTP_VERSION_NONE()**
>> Let curl use whichever version it wants.

> **CURL_HTTP_VERSION_1_0()**
>> Force HTTP 1.0 requests.

> **CURL_HTTP_VERSION_1_1()**
>> Force HTTP 1.1 requests.

35.2.7 FTP Options

CURLOPT_FTPPORT()

> Used with **curl->set**. This option expects a string value specifying the address to use for the FTP PORT instruction. The string may be an IP address, a host name, a network interface name (under Unix) or just a dash character (**-**) to let curl use your system's default IP address. The address can then be followed by a colon and a port number or port range separated by a dash.

CURLOPT_QUOTE()

> Used with **curl->set**. The value for this option should be an array of strings specifying FTP commands to run on the server prior to the FTP request. These will be done before any other commands are issued (even before the CWD command for FTP).

CURLOPT_POSTQUOTE()

> Used with **curl->set**. The value for this option should be an array of strings specifying FTP commands to run on the server after the FTP transfer request has been completed. The commands will only be run if no error occurred in the request.

CURLOPT_PREQUOTE()

> Used with **curl->set**. The value for this option should be an array of strings specifying FTP commands to run on the server after the transfer type is set.

CURLOPT_FTPLISTONLY()

> Used with **curl->set**. If set to "1", curl will just list the file names in a folder instead of doing a full listing of names, sizes, dates, and so on.

CURLOPT_FTPAPPEND()
>   Used with **curl->set**. If set to "1", curl will append to the remote file the data it's uploading instead of overwriting it.

CURLOPT_FTP_USE_EPRT()
>   Used with **curl->set**. If set to "1", curl will use EPRT and LPRT command for active FTP downloads.

CURLOPT_FTP_USE_EPSV()
>   Used with **curl->set**. If set to "1", curl will use the EPSV command for passive FTP downloads. (This is actually the default; turn it off by setting it to "0".)

CURLOPT_FTP_CREATE_MISSING_DIRS()
>   Used with **curl->set**. If set to "1", curl will try to create directories that don't exist for it to CWD into.

CURLOPT_FTP_RESPONSE_TIMEOUT()
>   Used with **curl->set**. This option takes an integer value specifying the number of seconds to wait for the server to respond to a command before considering the session hung.

CURLOPT_FTPSSLAUTH()
>   Used with **curl->set**. When doing FTP over SSL, this option specifies which authentication method to use. The value passed should be one of the following methods:
>
>   **CURLFTPAUTH_DEFAULT()**
>   >   Let curl decide.
>
>   **CURLFTPAUTH_SSL()**
>   >   Try "AUTH SSL" first, but if it fails try "AUTH TLS".
>
>   **CURLFTPAUTH_TLS()**
>   >   Try "AUTH TLS" first, but if it fails try "AUTH SSL".

CURLOPT_FTP_ACCOUNT()
>   Used with **curl->set**. This option takes a string that specifies the data sent in an ACCT command when an FTP server asks for "account data" after a user name and password have been provided.

35.2.8 Protocol Options

CURLOPT_TRANSFERTEXT()
>   Used with **curl->set**. If set to "1", curl will use ASCII mode for FTP transfers instead of binary.

CURLOPT_CRLF()
>   Used with **curl->set**. If set to "1", curl will convert Unix newlines to CRLF.

CURLOPT_RANGE()
>   Used with **curl->set**. This option takes a string for its value specifying the range you want in the form of *X-Y* where either "X" or "Y" may be omitted. Ranges work for HTTP, FTP, and FILE transfers only. HTTP transfers also support intervals separated by commas, such as "X-Y,N-M".

CURLOPT_RESUME_FROM()
>   Used with **curl->set**. This option takes an integer value specifying the offset in number of bytes to start the transfer from.

CURLOPT_RESUME_FROM_LARGE()
>   Used with **curl->set**. This is the large file version of **CURLOPT_RESUME_FROM** and also takes an integer for its value.

CURLOPT_CUSTOMREQUEST()
>   Used with **curl->set**. This option takes a string value specifying a custom HTTP, FTP, or POP3 request. This is particularly useful, for example, for performing an HTTP DELETE request.

CURLOPT_FILETIME()
>   Used with **curl->set**. If set to "1", curl will try to get the modification date for the document in the transfer.

CURLOPT_NOBODY()
> Used with **curl->set**. If set to "1", curl will only output the header portion of the received response. (Only relevant for protocols such as HTTP that have separate header and body parts.)

CURLOPT_INFILESIZE()
> Used with **curl->set**. This option takes an integer specifying the expected size of the input file for an upload. It does not limit how much data curl actually sends.

CURLOPT_INFILESIZE_LARGE()
> Used with **curl->set**. This is the large file version of **CURLOPT_INFILESIZE**.

CURLOPT_UPLOAD()
> Used with **curl->set**. Set this option to "1" to tell curl to prepare for an upload.

CURLOPT_MAXFILESIZE()
> Used with **curl->set**. This option takes an integer value specifying the maximum size of the file to download in bytes. If the requested file is larger then this size, nothing will be transferred and an error of **CURLE_FILESIZE_EXCEEDED** will be produced.

CURLOPT_MAXFILESIZE_LARGE()
> Used with **curl->set**. This is the large file version of **CURLOPT_MAXFILESIZE**.

35.2.9 Connection Options

CURLOPT_TIMEOUT()
> Used with **curl->set**. This option takes an integer value specifying the maximum time in seconds to wait for the curl transfer.

CURLOPT_LOW_SPEED_LIMIT()
> Used with **curl->set**. This option takes an integer value specifying the number of bytes per second the transfer should be below for the duration of **CURLOPT_LOW_SPEED_TIME** for curl to consider too slow and abort.

CURLOPT_LOW_SPEED_TIME()
> Used with **curl->set**. This option takes an integer value specifying the number of seconds a curl transfer must be below the rate set by **CURLOPT_LOW_SPEED_LIMIT** for curl to abort due to bad connection.

CURLOPT_MAXCONNECTS()
> Used with **curl->set**. This option takes an integer value specifying the maximum number of persistent cached connections this curl operation can have simultaneously opened. The default is 5.

CURLOPT_FRESH_CONNECT()
> Used with **curl->set**. Set this to "1" to force the next operation to use a new connection. (This option should be used with caution and only if you understand what it does.)

CURLOPT_FORBID_REUSE()
> Used with **curl->set**. If set to "1", curl will close the connection for the next operation after it finishes. (This option should be used with caution and only if you understand what it does.)

CURLOPT_CONNECTTIMEOUT()
> Used with **curl->set**. This option takes an integer value specifying the number of seconds to wait before timing out during the connection phase. (Once connected, this option is of no value.) The default is 300 seconds.

CURLOPT_IPRESOLVE()
> Used with **curl->set**. This option specifies which type of IP address to use if a host name resolves to more than one kind of IP address. The value passed should be one of the following methods:
>
> > **CURL_IPRESOLVE_WHATEVER()**
> > > This is the default, and it will resolve to all that your system allows.

> **CURL_IPRESOLVE_V4()**
>> Specifies using IPv4 addresses.
>
> **CURL_IPRESOLVE_V6()**
>> Specifies using IPv6 addresses.

CURLOPT_FTP_SSL()

CURLOPT_USE_SSL()
> Used with **curl->set**. This option specifies your SSL connection preferences to curl. The value passed should be one of the following methods:
>
> **CURLFTPSSL_NONE()**
>> Don't attempt to use SSL.
>
> **CURLFTPSSL_TRY()**
>> Try using SSL, but proceed as normal otherwise.
>
> **CURLFTPSSL_CONTROL()**
>> Require SSL for the control part of the connection or fail with **CURLE_USE_SSL_FAILED**.
>
> **CURLFTPSSL_ALL()**
>> Require SSL for all communication or fail with **CURLE_USE_SSL_FAILED**.

35.2.10 SSL and Security Options

CURLOPT_SSLCERT()
> Used with **curl->set**. This option expects a string value specifying the path to and file name of your certificate, or, with NSS, the nickname of the certificate you want to use. (If you want to use a file from the current directory, precede it with a "./" prefix in order to avoid confusion with a nickname.)

CURLOPT_SSLCERTTYPE()
> Used with **curl->set**. This option expects a string value of either **'PEM'** (the default) or **'DER'**. It is used to tell curl the format of your certificate.

CURLOPT_SSLKEY()
> Used with **curl->set**. This option expects a string value specifying the path to and file name of your private key.

CURLOPT_SSLKEYTYPE()
> Used with **curl->set**. This option expects a string value of either **'PEM'** (the default), **'DER'**, or **'ENG'**. It is used to tell curl the format of your private key.

CURLOPT_SSLKEYPASSWD()
> Used with **curl->set**. If your private key needs a password to be used, then pass a string value of the password with this option.

CURLOPT_SSLENGINE()
> Used with **curl->set**. This option expects a string value specifying which crypto engine to use. If the crypto device cannot be loaded, a **CURLE_SSL_ENGINE_NOTFOUND** error is returned.

CURLOPT_SSLENGINE_DEFAULT()
> Used with **curl->set**. If set to any value (recommended you set it to "1"), this option will set the crypto engine to curl's default asymmetric crypto engine. If the crypto engine cannot be set, a **CURLE_SSL_ENGINE_SETFAILED** error is returned.

CURLOPT_SSLVERSION()
> Used with **curl->set**. This option is used to control which version(s) of SSL/TLS can be used. The value passed should be one of the following methods to force using the version specified by the method name:
>
> **CURL_SSLVERSION_TLSv1()**

CURL_SSLVERSION_SSLv2()

CURL_SSLVERSION_SSLv3()

CURL_SSLVERSION_DEFAULT()
>   Can be passed instead to tell curl to figure out the protocol used by the remote server, though it won't use
>   **CURL_SSLVERSION_SSLv2**.

CURLOPT_SSL_VERIFYPEER()
>   Used with **curl->set**. This option expects an integer value of either "1" or "0", and it defaults to "1". It is used to specify
>   whether or not curl verifies the authenticity of the peer's certificate with a value of "1" meaning it does the verification
>   and "0" meaning it does not.

CURLOPT_CAINFO()
>   Used with **curl->set**. This option expects a string value specifying the path to and name of a file containing one or
>   more certificates needed to do peer verification. By default, this option is set to the path curl believes your system keeps
>   its CA cert bundle.

CURLOPT_CAPATH()
>   Used with **curl->set**. This option expects a string value specifying the path to a directory containing multiple CA
>   certificates to be used for peer verification.

CURLOPT_SSL_VERIFYHOST()
>   Used with **curl->set**. This option expects an integer value of either "0", "1", or "2". When the value is "0", the connection
>   to the remote server will succeed regardless of the SSL credentials. When the value is "1", curl will return a failure if the
>   authenticity of the server's SSL credentials cannot be verified, and when the value is "2", the connection will fail without
>   verification. The default for this option is "2".

CURLOPT_RANDOM_FILE()
>   Used with **curl->set**. This option expects a string value specifying the path to and file name of a file whose contents
>   will be used in seeding the random engine for SSL.

CURLOPT_EGDSOCKET()
>   Used with **curl->set**. This option expects a string value specifying the path to and file name of the Entropy Gathering
>   Daemon socket that will be used when seeding the random engine for SSL.

CURLOPT_SSL_CIPHER_LIST()
>   Used with **curl->set**. This option expects a string value specifying the list of ciphers that can be used in the SSL
>   connection. See the curl documentation for CURLOPT_SSL_CIPHER_LIST[58] for a discussion of the proper syntax needed.

CURLOPT_KRB4LEVEL()
>   Used with **curl->set**. This option expects a string value of either `'clear'`, `'safe'`, `'confidential'`, or `'private'`.
>   It is used to set the Kerberos security level for FTP and enable Kerberos awareness. Set the option to "null" to disable
>   Kerberos.

35.3 Using the Curl Type

The **curl** type is meant to be a low-level implementation, which means that it is usually not necessary to use it directly. For
the most part, the **include_url** method is all that is needed for HTTP requests and the **ftp_...** methods handle your FTP
needs. In fact, the examples below could have easily been done using one of those methods, but are provided to give you an
understanding of how to use the **curl** type in case you find yourself needing more control.

35.3.1 Make an HTTP HEAD Request

The following example uses the **curl** type to make a HEAD request to an HTTP server:

[58] https://curl.haxx.se/libcurl/c/curl_easy_setopt.html#CURLOPTSSLCIPHERLIST

```
local(req) = curl('http://www.example.com')
handle => { #req->close }

// Not verifying the return status of setting the option
local(_) = #req->set(CURLOPT_NOBODY, 1)

#req->raw
#req->close

// =>
// staticarray(true, HTTP/1.1 200 OK
// Accept-Ranges: bytes
// Cache-Control: max-age=604800
// Content-Type: text/html
// Date: Wed, 28 Aug 2013 13:42:53 GMT
// Etag: "3012602696"
// Expires: Wed, 04 Sep 2013 13:42:53 GMT
// Last-Modified: Fri, 09 Aug 2013 23:54:35 GMT
// Server: ECS (atl/5834)
// X-Cache: HIT
// x-ec-custom-error: 1
// Content-Length: 1270
//
// , )
```

35.3.2 List an FTP Directory

The following example lists the file and folder names at the specified FTP location:

```
local(req) = curl(
    "ftp://ftp.example.com/",
    -username=`MyUsername`,
    -password=`Shh...Secret`
)
handle => { #req->close }

#req->set(CURLOPT_FTPLISTONLY, 1)

#req->result

// =>
// .
// ..
// file1
// file2
// folder1
```

35.4 include_url

The **include_url** method is a wrapper around the **curl** type for requesting data via HTTP. We strongly recommend using this method for your HTTP request needs if possible.

include_url(*url::string*, …)
>   Requires a string representing a URL in the form of *http://www.example.com* (*https://* can also be used). By default, this

method returns the HTML body result of performing an HTTP GET request at the specified URL.

This method has several optional parameters that modify its behavior:

Parameters

- **-getParams** – Pass this parameter a staticarray or array of key/value pairs. This data is then converted into a query string and appended to the URL when making the HTTP request.

- **-postParams** – This option can take either a string, bytes, or **trait_forEach** object. For string and bytes objects, the data is set as the POST field (**CURLOPT_POSTFIELDS**) for the request without modification. If passed a **trait_forEach** object, each value should be a key/value pair object that will then first be converted into the query string format before being set as the POST field.

- **-sendMimeHeaders** – This option can take either a string, bytes, or **trait_forEach** object. For string and bytes objects, the data is set as additional HTTP headers for the request without modification. If passed a **trait_forEach** object, each value should be a key/value pair object whose first value is the header name and the second value is the value. These will then first be converted into the form "Header: Value" and joined with **"\r\n"** before being set as additional HTTP headers.

- **-username** – This option allows you to specify the username for connections that require authentication.

- **-password** – This option allows you to specify the password for connections that require authentication.

- **-noData** – Passing this option does not change any aspect of the curl HTTP request, but tells **include_url** not to return any data.

- **-verifyPeer** – Use this option to specify whether or not Lasso should verify the SSL certificate of the HTTP peer being connected to. The default is "true".

- **-sslCert** – This parameter is used to set the **CURLOPT_SSLCERT** option.

- **-sslCertType** – This parameter is used to set the **CURLOPT_SSLCERTTYPE** option.

- **-sslKey** – This parameter is used to set the **CURLOPT_SSLKEY** option.

- **-sslKeyType** – This parameter is used to set the **CURLOPT_SSLKEYTYPE** option.

- **-sslKeyPasswd** – This parameter is used to set the **CURLOPT_SSLKEYPASSWD** option.

- **-timeout** – This parameter is used to set the **CURLOPT_TIMEOUT** option.

- **-connectTimeout** – This parameter is used to set the **CURLOPT_CONNECTTIMEOUT** option.

- **-retrieveMimeHeaders** – This parameter expect a string specifying the name of a thread variable to store the HTTP response header data in.

- **-options** – Pass this parameter a staticarray or array of pairs, the first value of the pair should be one of the **CURLOPT_...** methods and the second value should be the appropriate setting for that **curl** option.

- **-string** – The default is for **include_url** to return a bytes object, but if this parameter is set, then it will return a string object. You can pass a string to this parameter to specify the character set to use. Setting the parameter to "true" causes **include_url** to first check the curl headers for the character set to use, otherwise Lasso will try to determine the character set itself from the body of the response. If that fails, the default is to use UTF-8 encoding.

- **-basicAuthOnly** – Setting this option to "true" causes **include_url** to only use HTTP Basic authentication.

35.4.1 Make an HTTP GET Request

The following example issues a basic HTTP GET request for the specified URL:

```
include_url('http://www.example.com/')

// =>
// <!doctype html>
// <html>
// <head>
//     <title>Example Domain</title>
// (... rest of response ...)
```

35.4.2 Send Data with an HTTP PUT Request

The following example issues an HTTP PUT request, passing data in the body of the request. The example result is a JSON-formatted string, but would be the body of the HTTP response given by the server.

```
include_url(
    'http://www.example.com/',
    -postParams=(: 'id'=5, 'animal'='rhino'),
    -options=(: CURLOPT_CUSTOMREQUEST='PUT')
)

// => {"status": "Success"}
```

35.4.3 Specify HTTP Headers

The following example adds a *User-Agent* header to the HTTP request:

```
include_url(
    'http://www.example.com/',
    -sendMimeHeaders=(: 'User-Agent'='LassoBrowse/1.0')
)

// =>
// <!doctype html>
// <html>
// <head>
//     <title>Example Domain</title>
// ... rest of response ...
```

35.4.4 Read Response Headers

The following example gets the response headers for the request stored in a variable named "my_headers" and then displays them:

```
local(my_body) = include_url(
    'http://www.example.com/',
    -retrieveMimeHeaders='my_headers'
)
$my_headers

// =>
// HTTP/1.1 200 OK
// Accept-Ranges: bytes
// Cache-Control: max-age=604800
```

```
// Content-Type: text/html
// Date: Wed, 28 Aug 2013 20:00:21 GMT
// Etag: "3012602696"
// Expires: Wed, 04 Sep 2013 20:00:21 GMT
// Last-Modified: Fri, 09 Aug 2013 23:54:35 GMT
// Server: ECS (atl/FCAA)
// X-Cache: HIT
// x-ec-custom-error: 1
// Content-Length: 1270
```

35.5 FTP Methods

The `ftp_...` methods are simple wrappers around the `curl` type for requesting and sending data via FTP. We strongly recommend using these methods for your FTP needs if possible.

ftp_getData(*url::string, -username::string=?, -password::string=?, -options::array=?*)
> Returns a bytes object representing the remote file's contents at the specified FTP URL. It can also optionally take a username and password to be used for authentication to the FTP server. Also, the `-options` parameter can be passed an array of pairs, the first value of the pair should be one of the `CURLOPT_...` methods and the second value should be the appropriate setting for that `curl` option.

ftp_getFile(*url::string, -file::string, -username::string=?, -password::string=?, -options::array=?*)
> Downloads the remote file specified by the FTP URL in the first parameter to the location specified by the `-file` parameter. It can also optionally take a username and password to be used for authentication to the FTP server. Also, the `-options` parameter can be passed an array of pairs, the first value of the pair should be one of the `CURLOPT_...` methods and the second value should be the appropriate setting for that `curl` option.

ftp_getListing(*url::string, -username=?, -password=?, -listOnly::boolean=?, -options::array=?*)
> Acquires a directory listing of the remote directory specified by the FTP URL. If you only want the names of the files and folders in the specified remote directory, pass the `-listOnly` parameter. You can also optionally specify a username and password to be used for authentication to the FTP server. The method can also take the `-options` parameter which expects an array of pairs; the first value of the pair should be one of the `CURLOPT_...` methods and the second value should be the appropriate setting for that `curl` option.

ftp_putData(*url::string, -data::bytes, -username=?, -password=?, -options::array=?*)
> Takes an FTP URL and a bytes object representing file data. If a file doesn't exist at the location specified by the URL, one will be created with the data specified by the `-data` parameter. If a file does exist at the path specified by the URL then its contents will be overwritten with the new data. (See the example below for how to change the method's behavior to append the data instead.)

> This method can optionally take a username and password to be used for authentication to the FTP server. Also, the `-options` parameter can be passed an array of pairs, the first value of the pair should be one of the `CURLOPT_...` methods and the second value should be the appropriate setting for that `curl` option.

ftp_putFile(*url::string, -file, -username=?, -password=?, -options::array=?*)
> Uploads the local file specified by the `-file` parameter to the remote location specified by the FTP URL passed as the first parameter. If a file doesn't exist at the location specified by the URL, one will be created, otherwise the contents of the existing remote file will be overwritten with the new data from the local file.

> This method can optionally take a username and password to be used for authentication to the FTP server. Also, the `-options` parameter can be passed an array of pairs, the first value of the pair should be one of the `CURLOPT_...` methods and the second value should be the appropriate setting for that `curl` option.

ftp_deleteFile(*url::string, -username=?, -password=?, -options::array=?*)
> Deletes the remote file specified by the FTP URL in the first parameter. It can optionally take a username and password to be used for authentication to the FTP server. Also, the `-options` parameter can be passed an array of pairs, the first

value of the pair should be one of the **CURLOPT_...** methods and the second value should be the appropriate setting for that **curl** option.

35.5.1 Retrieve Contents of a Remote File

The following example downloads the data in a file named "test.txt" from the remote server, and then displays it:

```
ftp_getData(
    'ftp://example.com/test.txt',
    -username=`MyUsername`,
    -password=`Shh...Secret`
)

// => "Hello, world."
```

35.5.2 Download a Remote File

The following example downloads the remote file "test.txt" to "/tmp/file.txt" from the root of the file system:

```
ftp_getFile(
    'ftp://example.com/test.txt',
    -file='//tmp/file.txt',
    -username=`MyUsername`,
    -password=`Shh...Secret`
)
```

35.5.3 List Contents of a Remote Directory

The following example gets a list of all the files and folders at the FTP root of the "example.com" server and displays its size and then its name (with a trailing slash if it is a directory):

```
local(listing) = ftp_getListing(
    'ftp://example.com/test.txt',
    -username=`MyUsername`,
    -password=`Shh...Secret`
)
with item in #listing
    let item_type = #item->find('filetype')
    let item_size = #item->find('filesize')
    let item_name = #item->find('filename') + (#item_type == 'directory' ? '/' | '')
do {^
    #item_size + 'B  ' + #item_name + '\n'
^}

// =>
// 170B  ./
// 170B  ../
// 387B  directory/
// 15B  test.txt
```

35.5.4 Update an Existing Remote File

The following example will take the data "\nAs You Wish" and append it to the remote "test.txt" file. (The **CURLOPT_FTPAPPEND** option changes the behavior to append the data.)

```
ftp_putData(
    'ftp://example.com/test.txt',
    -data=bytes('\nAs You Wish'),
    -username=`MyUsername`,
    -password=`Shh...Secret`,
    -options=array(CURLOPT_FTPAPPEND=1)
)
```

35.5.5 Upload a Local File to the Remote Server

The following example takes the local file "test.txt" at the current web root and uploads it as "file.txt" to the specified path in the URL. (The **CURLOPT_FTP_CREATE_MISSING_DIRS** option specifies that any missing intermediary directories on the remote server will be created.)

```
ftp_putFile(
    'ftp://example.com/new_dir/test.txt',
    -file='/test.txt',
    -username=`MyUsername`,
    -password=`Shh...Secret`,
    -options=array(CURLOPT_FTP_CREATE_MISSING_DIRS=1)
)
```

35.5.6 Delete a Remote File

The following example will delete the "test.txt" file at the FTP root of the remote server:

```
ftp_deleteFile(
    'ftp://example.com/test.txt',
    -username=`MyUsername`,
    -password=`Shh...Secret`
)
```

Sending Email

Lasso includes a built-in system for queuing and sending email to SMTP servers. Email messages can be sent to site visitors to notify them when they create a new account or to remind them of their login information, or to administrators when various errors or other conditions occur. Email messages can even be sent in bulk to many email addresses to notify site visitors of updates to the website or other news.

36.1 SMTP Email Basics

Email messages are queued using the **email_send** method. All outgoing messages are stored in tables of the Site database. The queue can be examined and messages removed in the *Email Queue section of Lasso Server Admin*.

Lasso's email system checks the queue periodically and sends any waiting messages. If the email system encounters an error when sending an email then it stores the error in the database and requeues the message. If too many errors are encountered then the message send will be cancelled.

By default, Lasso sends queued messages directly to the SMTP server that corresponds to each recipient address. This means that a single message may end up being sent to multiple SMTP servers in order to deliver it to each recipient. It is also possible to specify SMTP hosts directly within the **email_send** method.

Note: If a local SMTP server is being used, then Lasso must either have valid SMTP AUTH credentials or otherwise be allowed to send messages through the SMTP server unrestricted. Consult the SMTP server documentation for details about how to set up SMTP AUTH security or how to allow specific IP addresses to relay messages.

By default Lasso will send up to 100 messages to each SMTP server with every connection. Lasso will open up to five outgoing SMTP connections at a time. Lasso selects messages to send in order of priority, but once it connects to an SMTP server it delivers as many messages as possible. This means that a batch send to an SMTP server will contain high-priority messages as well as medium- and low-priority messages.

Note: The maximum size of an email message including all attachments must be less than 8 MB when using the **email_send** method. If necessary, larger messages can be sent using the **-immediate** parameter or the **email_immediate** method described in the section *Composing and Queueing Email* below.

36.1.1 Email Composition

The structure of a composed email message will depend on what type of message is being sent. Lasso supports the following structure variations depending on which parameters are specified in the **email_send** or **email_compose** methods.

Plain Text
    Simple messages specified with a **-body** parameter are sent as a single *text/plain* part with no boundaries.

HTML
Simple HTML messages with an **-html** parameter are sent as a single *text/html* part with no boundaries.

HTML with Plain Text
Messages that have both an **-html** parameter and a **-body** parameter are sent as *multipart/alternative* messages with both *text/plain* and *text/html* parts.

HTML with Embedded Images
Messages that use **-htmlImages** replace the *text/html* part with a *multipart/related* part with enclosed *text/html* and inline attachment parts.

Attachments
Messages with attachments are sent as *multipart/mixed* and include the *text/plain*, *text/html*, *multipart/alternative*, or *multipart/related* part that is appropriate based on the type of message and the attachment parts.

See each of the following sections for details about how other **email_send** and **email_compose** parameters affect the composition of each part.

36.2 Composing and Sending Email

The **email_send** method is used to send email from Lasso. This method supports the most common types of email including plain text, HTML, HTML with a plain text alternative, embedded HTML images, and attachments.

email_send(*-subject*, *-from*, ...)
Adds a message to the email queue. The method requires a **-subject** parameter, a **-from** parameter, and one of either **-to**, **-cc**, or **-bcc** parameters. Also required is one of either **-body** or **-html** parameters. Below is a description of each of the parameters.

Parameters

- **-subject** – The subject of the message. Required.

- **-from** – The sender of the message. Required.

- **-to** – The recipient of the message. Multiple recipients can be specified by separating their email addresses with commas.

- **-cc** – Carbon copy recipients of the message.

- **-bcc** – Blind carbon copy recipients of the message.

- **-body** – The body of the message. Either a **-body** or **-html** part (or both) is required. See the section *Send HTML Messages* for details about how to create HTML and mixed messages.

- **-html** – The HTML part of the message. Either a **-body** or **-html** part (or both) is required.

- **-htmlImages** – Specifies a list of files that will be used as images for the HTML part of an outgoing message. Accepts either an array of file paths or an array of pairs containing a file name as the first part and the data for the file as the second part.

- **-attachments** – Specifies a list of files that will be attached to the outgoing message. Accepts either an array of file paths or an array of pairs containing a file name as the first part and the data for the file as the second part.

- **-tokens** – Specifies a map of token names and values that will be merged into the email message. The same tokens will be used on every message.

- **-merge** – Specifies a map of email addresses. Each email address should have as its value a map of token names and values. The values in this merge map will override those in the tokens map if both are specified.

- **-priority** – Specifies the priority of the message. Valid values include "High" or "Low". Default is "Medium".

- **-replyTo** – The email address that should be used for replies to this message.

- **-sender** – The email address that should be reported as the sender of this message.

- **-transferEncoding** – The value for the *Transfer-Encoding* header of the message.

- **-contentType** – The value for the *Content-Type* header of the message.

- **-characterSet** – The character set in which the message should be encoded.

- **-extraMIMEHeaders** – A pair array defining extra MIME headers that should be added to the email message.

- **-immediate** – If specified then the email is sent immediately without using the outgoing message queue. This option can be used for messages that have very large attachments.

- **-host** – SMTP host through which to send messages.

- **-port** – SMTP port. Defaults to "25".

- **-username** – Specifies the username for SMTP AUTH if required by the SMTP server. If specified a **-password** is also required.

- **-password** – Specifies the password for SMTP AUTH if required by the SMTP server. If specified a **-username** is also required.

- **-timeout** – Specifies the timeout for the SMTP server in seconds.

- **-ssl** – If specified then SSL is used when connecting to the SMTP server.

- **-simpleform** – If specified then the message is sent without a body.

- **-date** – A date object specifying a time in the future to send the message.

36.2.1 Send a Plain Text Message

An email can be sent with a hard-coded body by specifying the message directly within the **email_send** method. The following example shows an email sent to "*example@example.com*" with a hard-coded message body:

```
email_send(
    -to      = 'example@example.com',
    -from    = 'example@example.com',
    -subject = 'An Email',
    -body    = 'This is the body of the email.'
)
```

The body of an email message can be assembled in a variable in the current Lasso page and then sent using the **email_send** method. The following example shows a variable "email_body" which has several items added to it before the message is finally sent:

```
local(email_body) = 'This is the body of the email'
#email_body += '\nSent on: ' + server_date + ' at ' + server_time
#email_body += '\nCurrent visitor: ' + client_username + ' at ' + client_ip

email_send(
    -to      = 'example@example.com',
    -from    = 'example@example.com',
    -subject = 'An Email',
```

```
    -body    = #email_body
)
```

A Lasso page on the web server can be used as the message body for an email message using the **include** method. A Lasso page created to be a message body should contain no extra whitespace. The following example shows a Lasso page "format.lasso", which is in the same folder as the current Lasso page being used as the message body for an email. Any Lasso code within "format.lasso" will be executed before the email is sent.

```
email_send(
    -to      = 'example@example.com',
    -from    = 'example@example.com',
    -subject = 'An Email',
    -body    = include('format.lasso')
)
```

36.2.2 Send an Email to Multiple Recipients

Email can be sent to multiple recipients by including their addresses as a comma-delimited list in the **-to** parameter, the **-cc** parameter, or the **-bcc** parameter.

The following example shows an **email_send** method with two recipients in the **-to** parameter. The recipients' email addresses are specified with a comma between them: **'example@example.com,someone@example.com'**. No extraneous information such as the recipients' real names needs to be included.

```
email_send(
    -to      = 'example@example.com, someone@example.com',
    -from    = 'example@example.com',
    -subject = 'An Email',
    -body    = include('format.lasso')
)
```

The following example shows an **email_send** method with one recipient in the **-to** parameter and two recipients in the **-cc** parameter. The carbon copy parameter is generally used to include recipients who are not the primary recipient of the email, but need to be informed of the correspondence. The addresses for the carbon-copied recipients are stored in variables and concatenated together with a comma between them.

```
local(president) = 'president@example.com'
local(someone)   = 'someone@example.com'

email_send(
    -to      = 'example@example.com',
    -cc      = #president + ',' + #someone,
    -from    = 'example@example.com',
    -subject = 'An Email',
    -body    = include('format.lasso')
)
```

The following example shows an **email_send** method with one recipient in the **-to** parameter and two recipients in the **-bcc** parameter. The Blind Carbon Copy parameter can be used to send email to many recipients without disclosing the full list of recipients to everyone who receives the email. Each recipient will receive an email that contains only the address in the **-to** parameter; in this case, "*announce@example.com*".

```
email_send(
    -to      = 'announce@example.com',
    -bcc     = 'example@example.com, someone@example.com',
    -from    = 'example@example.com',
```

```
    -subject = 'An Email',
    -body    = include('format.lasso')
)
```

36.2.3 Send HTML Messages

HTML messages can be sent from Lasso by specifying the HTML body for the message using the **-html** parameter. Images can be embedded in the email message using the **-htmlImages** parameter. If a message includes both an **-html** parameter and a **-body** parameter then it will be sent as a *multipart/alternative* message so mail clients that do not recognize HTML messages will see only the plain text part.

An HTML page can be sent as the body of the message by using an **include** method call as the value to the **-html** parameter. Image references or URLs in the HTML page should be specified including the *http://* prefix and server name. (Alternatively, images can be embedded within the email using the **-htmlImages** parameter as shown in a later example.)

For example, the following HTML would reference an example web page and an image showing a coupon graphic. Both addresses are fully specified since they will need to be loaded from within the email client without any other information about the web server.

```
<h2>Money Saving Coupon</h2>
<p>Print out the money saving coupon below or click on it to order directly from our website.<br />
    <a href="http://www.example.com/couponoffer.html">
        <img src="http://www.example.com/couponoffer.gif" border="0" width="288" height="288" />
    </a>
</p>
```

If that HTML were in a file named "email_body.html", then a Lasso page in the same folder could contain the following code to email it out:

```
email_send(
    -to      = 'example@example.com',
    -from    = 'example@example.com',
    -subject = 'An HTML Email',
    -html    = include('email_body.html')
)
```

An HTML/plain text alternative email can be sent by specifying both a **-body** parameter and an **-html** parameter. The message of both parts should be equivalent. (If equivalent text and HTML parts can't be generated then it is preferable to send just an HTML part. Email clients that don't render HTML will display the raw HTML to the user, but this is preferable to seeing a message that simply says that the message was sent as HTML.) Recipients with text-based email clients will see the text part while recipients with HTML-based email clients will see the HTML part.

```
email_send(
    -to      = 'example@example.com',
    -from    = 'example@example.com',
    -subject = 'A Multi-Part Email',
    -body    = include('format.lasso'),
    -html    = include('email_body.html')
)
```

HTML messages can include embedded images using the **-htmlImages** parameter. This parameter can be specified with either a single file name or an array of file names. Within the email message the images can be referenced in two ways.

1. If the **email_send** method contains the parameter **-htmlImages=array('/apache_pb.gif')** then Lasso will automatically update any HTML **<img>** tags that have that same image referenced in their **"src"** parameter. Note that the path must be exactly the same for Lasso to be able to make this replacement.

```
email_send(
    -to        = 'example@example.com',
    -from      = 'example@example.com',
    -subject   = 'An HTML Email With Embedded Image',
    -html      = '<h2>Embedded Image</h2><br /><img src="/apache_pb.gif" />',
    -htmlImages = array('/apache_pb.gif')
)
```

2. Alternatively, the "Content-ID" of the embedded image could be referenced in the `<img>` tag following a "cid:" prefix. Lasso automatically uses the image file name as the "Content-ID" without any path information so the same image referenced above can also be referenced like this: `<img src="cid:apache_pb.gif" />`.

```
email_send(
    -to        = 'example@example.com',
    -from      = 'example@example.com',
    -subject   = 'An HTML Email With Embedded Image',
    -html      = '<h2>Embedded Image</h2><br /><img src="cid:apache_pb.gif" />',
    -htmlImages = array('/apache_pb.gif')
)
```

Images that are generated programatically can be embedded in an HTML message by specifying a pair consisting of the name and data of the image. In the example below the image data comes from the **include_raw** method, but it could also be generated using the **image** methods or retrieved from a database field. Note that the name of the image does not have to match, but the name that is specified in the first part of the pair should be used within the HTML body.

```
email_send(
    -to        = 'example@example.com',
    -from      = 'example@example.com',
    -subject   = 'An HTML Email With Embedded Image',
    -html      = '<h2>Embedded Image</h2><br /><img src="myimage.jpg" />',
    -htmlImages = array('myimage.jpg'=include_raw('/apache_pb.jpg'))
)
```

36.2.4 Send Attachments with an Email

Files can be included as attachments to email messages using the `-attachments` parameter. This parameter takes an array of file paths as a value. When the email is sent, each file is read from disk and encoded using Base64 encoding. The recipient's email client will automatically decode the attached files and make them available.

Note: The maximum size of an email message including all attachments must be less than 8 MB when using the **email_send** method. If necessary, larger messages can be sent using the `-immediate` parameter or the **email_immediate** method described in the section *Composing and Queueing Email* below.

The following example shows a pair of files being sent with an email message. The attachments are named "MyAttachment.txt" and "MyAttachment2.txt". They are located in the same folder as the Lasso page that is sending the email. These text files will not be processed by Lasso before they are sent.

```
email_send(
    -to          = 'example@example.com',
    -from        = 'example@example.com',
    -subject     = 'An Email with Two Attachments',
    -body        = 'This is the body of the Email.',
    -attachments = array('MyAttachment.txt', 'MyAttachment2.txt')
)
```

Files can be generated programmatically and attached to an email message by specifying a pair with the name and contents of the file. For example, the following **email_send** example uses the **pdf_doc** type to create a PDF file. The generated PDF file is sent as an attachment without it ever being written to disk.

```
local(my_file) = pdf_doc(-size='A4', -margin=(: 144.0, 144.0, 72.0, 72.0))
#my_file->add(
    pdf_text("I'm a PDF document", -font=pdf_font(-face='Helvetica', -size=36))
)

email_send(
    -to          = 'example@example.com',
    -from        = 'example@example.com',
    -subject     = 'An Email with a PDF',
    -body        = 'This is the body of the Email.',
    -attachments = array('MyPDF.pdf'=string(#my_file))
)
```

36.2.5 Change the Priority of a Message

Most messages should be sent at the default priority. Sending bulk messages like a newsletter at "Low" priority will ensure that normal email from the site is sent as soon as possible rather than waiting for the entire newsletter to be sent first. The "High" priority should be reserved for time-dependent messages such as confirmation emails that a site visitor will be looking for immediately within their email client.

To specify the priority, use the **-priority** parameter:

```
email_send(
    -to       = 'example@example.com',
    -from     = 'example@example.com',
    -subject  = 'Password Reset Instructions',
    -body     = include('password_reset.lasso'),
    -priority = 'High'
)
```

36.2.6 Send a Message with a "Reply-To" and "Sender" Header

The **-replyTo** parameter specifies an address different from the **-from** address that should be used for replies. Most email clients will use this address when composing a response to a message. The **-sender** parameter allows an alternate sender from the **-from** address to be specified. This can be useful if a message is forwarded by Lasso, but the original sender should still be recorded.

```
email_send(
    -to      = 'example@example.com',
    -from    = 'example@example.com',
    -replyTo = 'repsonses@example.com',
    -sender  = 'otheruser@example.com',
    -subject = 'An Email',
    -body    = include('format.lasso')
)
```

36.2.7 Send a Message with Extra Headers

The **-extraMIMEHeaders** parameter can be used to send any additional required header parameters. The value should be an array of name/value pairs. Each of the pairs will be inserted into the email as an additional header.

```
email_send(
    -to                = 'example@example.com',
    -from              = 'example@example.com',
    -subject           = 'An Email',
    -body              = include('format.lasso'),
    -extraMIMEHeaders = array('Header'='Value', 'Header'='Value')
)
```

36.2.8 Use an Alternate SMTP Server

Specify the **-host** parameter in the **email_send** method directly. If required the port of the SMTP server can be changed with the **-port** parameter. An SMTP AUTH username and password can be provided with the **-username** and **-password** parameters. And the **-timeout** parameter sets the timeout for the SMTP server in seconds.

```
email_send(
    -host     = 'mail.example.com',
    -username = 'SMTP_USER',
    -password = 'USER_PASS',
    -timeout  = 120,
    -to       = 'example@example.com',
    -from     = 'example@example.com',
    -subject  = 'An Email',
    -body     = include('format.lasso')
)
```

36.3 Email Merge

Lasso can merge values into email messages just before sending them. This allows a single email message to be composed and then customized for several recipients. The **-tokens** and **-merge** parameters make this possible.

In order to use the **-tokens** and **-merge** parameters the email message must contain one or more email tokens. The preferred method for specifying tokens is to use the **email_token** method. In plain text messages or messages that can't be processed through Lasso the **#TOKEN#** marker can be used instead. For example, the method call **email_token('FirstName')** corresponds to the marker **#FirstName#**.

email_token(*name::string*)
> Email tokens are created using this method. It requires a single value containing the name of the email token.

For example, an email message can be marked up with email tokens for the first name and last name of the recipient. The start of the message, stored in a file called "body.lasso" could be:

```
<p>Dear [email_token('FirstName')] [email_token('LastName')],</p>
```

The email message is going to be sent to two recipients: "John Doe" at "*john@example.com*" and "Jane Doe" at "*jane@example.com*". Each element of the merge map includes an email address as the key and a map of token values as its value, constructed as follows:

```
local(myMergeTokens) = map(
    'john@example.com' = map('FirstName'="John", 'LastName'="Doe"),
    'jane@example.com' = map('FirstName'="Jane", 'LastName'="Doe"),
)
```

A default token map can also be constructed. The values from this map would be used if any tokens are missing from the specified email address maps shown above.

```
local(myDefaultTokens) = map('FirstName'="Lasso User", 'LastName'="")
```

The **email_send** method call would be written as follows. The email message is being sent to two recipients. The method references "body.lasso" as the **-body** of the email message that has the included **email_token** methods, **-merge** specifies #myMergeTokens, and **-tokens** specifies #myDefaultTokens.

```
email_send(
    -to      = 'john@example.com, jane@example.com',
    -from    = 'example@example.com',
    -subject = 'Mail Merge',
    -body    = include('body.lasso'),
    -merge   = #myMergeTokens,
    -tokens  = #myDefaultTokens
```

The message to John Doe would contain this text:

```
Dear John Doe,
```

email_merge(_data, tokens, charset=?, transferEncoding=?_**)**
> Allows the email merge operation to be performed on any text. Requires two parameters: the text that is to be modified and a map of tokens to be replaced in the text. Optional `charset` and `transferEncoding` parameters can specify what type of encoding should be applied to the merged tokens.

36.4 Email Sending Status

Email messages that are sent using the **email_send** method are stored in an outgoing email queue temporarily and then sent by a background process. Any errors encountered when sending a message can be viewed in the _Email Queue section of Lasso Server Admin_.

However, it is often desirable to get information about a message that was sent programatically without examining the queue table. The following documented methods allow the status of a recently sent message to be examined.

email_result()
> Can be called immediately after calling **email_send** to get a unique ID string for the queued message.

email_status(_id_**)**
> Accepts an ID from the **email_result** method and returns the status of the queued message: "sent", "queued", or "error".

Important: The email sender may take a few seconds or longer to send a message. Checking the status immediately after calling **email_send** will always return "queued", so be sure to add delay before checking the status.

The following example shows an **email_send** method call that sends a message. The **email_result** method is called immediately after to store the unique ID of the sent message. After a delay of 30 seconds the **email_status** method is called to see if the message was successfully sent.

```
email_send(
    -to      = 'example@example.com',
    -from    = 'example@example.com',
    -subject = 'An Email',
    -body    = 'This is the body of the email.'
)
local(my_email) = email_result
```

```
sleep(30000)
email_status(#my_email)
```

In a practical solution the unique ID returned by **email_result** would be stored in a session variable or in a database table and then would be checked sometime later using **email_status** to see if the email message was sent or if the address it was sent to was invalid.

36.5 Composing and Queueing Email

The **email_send** method handles all of the most common types of email that can be sent through Lasso including plain text messages, HTML messages, HTML messages with a plain text alternative messages, and messages with attachments.

For more complex messages structures the **email_compose** type can be used directly to create the MIME text of the message. The message can then be sent with the **email_queue** method. Both of these methods are used internally by **email_send**.

The **email_compose** type accepts the same parameters as **email_send** except those which specify the SMTP server and priority of the outgoing message. After creating an object with **email_compose**, member methods can be used to add additional text parts, HTML parts, attachments, or generic MIME parts. This allows very complex email structures to be created with much more control than **email_send** provides.

The **email_compose** type can also be used to create email parts. When the creator method is called without a **-to**, **-from**, or **-subject** parameter, then a MIME part is created rather than a complete email message. This part can then be fed into the **email_compose->addPart** method or into the **-attachments** or **-htmlImages** parameters to place the part within a complex email message.

The **email_queue** method is designed to be fed an **email_compose** object. It requires three parameters: the **-data**, **-from**, and **-recipients** parameters as attributes of an **email_compose** object. Additionally, SMTP server parameters and the sending priority can be specified just like in **email_send**. Queued emails must be less than 8 MB in size including all encoded attachments.

The **email_immediate** method takes the same parameters as the **email_queue** method, but sends the message immediately rather than adding it to the email queue. This method can be used to send messages larger than 8 MB if required. Use of the **email_immediate** method is not recommended since it bypasses the priority, error-handling, and connection-handling features of the email sending system.

type **email_compose**

email_compose(-to=?, -from=?, -cc=?, -bcc=?, -subject=?, -sender=?, -replyTo=?, -body=?, -html=?, -date=?, -contentType=?, -characterSet=?, -transferEncoding=?, -contentDisposition=?, -headerType=?, -extraMIMEHeaders=?, -attachments=?, -attachment=?, -htmlImages=?, -parts=?**)**
> Creates an **email_compose** object, accepting similar parameters as **email_send**. If the **-to**, **-from**, and **-subject** parameters are not specified then a MIME part is created, otherwise a full MIME email is created.

email_compose->addAttachment(-data=?, -name=?, -path=?, -type=?**)**
> Adds an attachment to an **email_compose** object. The data of the attachment can be specified directly in the **-data** parameter or the path to a file can be specified in the **-path** parameter. The name of the attachment can be specified in the **-name** parameter. The MIME type can be specified with the **-type** parameter.

email_compose->addHTMLPart(-data=?, -path=?, -images=?**)**
> Adds an HTML part to an **email_compose** object. The text of the HTML part can be specified directly in the **-data** parameter or the path to a file can be specified in the **-path** parameter. Additionally, the **-images** parameter can take the same values as the **-htmlImages** parameter of the **email_send** method.

email_compose->addTextPart(-data=?, -path=?**)**
> Adds a text part to an **email_compose** object. The text of the part can be specified directly in the **-data** parameter or the path to a file can be specified in the **-path** parameter.

email_compose->**addPart**(*-data=?*)
> Adds a generic part to an **email_compose** object. Requires a parameter **-data** that specifies the data for the part. The part must be properly formatted as a MIME part. No formatting or encoding will be performed by Lasso.

email_compose->**data**(*-prefix::boolean=?, -force::boolean=?*)
> Returns the MIME text of the composed email.

email_compose->**from()**
> Returns the from address of the composed email.

email_compose->**recipients()**
> Returns a list of recipients of the composed email.

email_batch()
> Takes a block of code, and within this code it temporarily suspends some back-end operations of the email queue so that a batch of email messages can be queued quickly. Any messages that are already queued will continue to send while the code in the specified block is running.

email_queue(*-data=?, -recipients=?, -from=?, -host=?, -username=?, -password=?, -port=?, -timeout=?, -priority=?, -tokens=?, -merge=?, -date=?, -ssl=?*)
> Queues a message for sending. Requires a **-data** parameter with the MIME text of the email to send, **-from** specifying the from address for the email, and **-recipients** an array of recipients for the email. Can also accept **-priority** and SMTP server **-host**, **-port**, **-timeout,``-username**, and **-password** parameters. A different **-tokens** parameter can be specified for each queued message to perform an email merge.

email_immediate(*-data, -recipients=?, -from=?, -host=?, -username=?, -password=?, -port=?, -timeout=?, -ssl=?*)
> The same as **email_queue**, but sends the message immediately without storing it in the database.

36.5.1 Compose an Email Message

The **email_compose** method can be used to compose an email message. In this example a simple email message is created in a variable "message":

```
local(message) = email_compose(
    -to      = 'example@example.com',
    -from    = 'example@example.com',
    -subject = 'Example Message',
    -body    = 'Example Message'
)
```

The text of the composed email message can be viewed by outputting the variable "message" to the page. Note that **string->encodeHtml** should always be used since certain headers of the email message use angle brackets to surround values. Also, HTML **<pre>** tags make it much easier to see the formatting of the email message.

```
<pre>[#message->asString->encodeHtml]</pre>
```

Additional text or HTML parts or attachments can be added using the appropriate member methods on the object in the "message" variable. For example, an attachment can be added using the **email_compose->addAttachment** method as follows:

```
#message->addAttachment(-path='ExampleFile.txt')
```

36.5.2 Queue an Email Message

An email message created using the **email_compose** type can be queued for sending using the **email_queue** method. The following example shows how to send the email message created above. The three required parameters **-data**, **-from**, and **-recipients** are all fetched from the **email_compose** object.

```
email_queue(
    -data       = #message->data,
    -from       = #message->from,
    -recipients = #message->recipients
)
```

36.5.3 Send a Batch of Messages

The **email_batch** method can be used when a number of messages need to be queued all at once. The method temporarily suspends some back-end operations of the email queue so that the messages can be queued faster. When the capture block is processed the queue is allowed to resume sending the queue messages.

The example below shows how an inline could be used to find a collection of email addresses. The **email_batch** method ensures that the messages are queued as quickly as possible.

```
email_batch => {
    inline(-search, ...) => {
        records => {
            email_send(-from='sender@example.com', -to=field('email_address'), ...)
        }
    }
}
```

Tip: The **email_merge** method discussed earlier in this chapter can also be used to send an email message to a collection of recipients quickly.

36.6 Sending SMTP Commands

All communication with remote SMTP servers is handled by a type called **email_smtp**. These connections are normally handled automatically by the **email_send**, **email_queue**, **email_immediate**, and background email sending processes.

The **email_smtp** type can be used directly for low-level access to remote SMTP servers, but this is not generally necessary.

type **email_smtp**

email_smtp(*-host::string=?, -port::integer=?, -timeout::integer=?, -username=?, -password=?, -ssl::boolean=?, -clientIp=?*)
    Creates a new SMTP connection object. Can optionally pass in the SMTP server parameters.

email_smtp->open(*-host=?, -port=?, -timeout=?, -username=?, -password=?, -ssl=?, -clientIp=?*)
    Requires a **-host** that specifies the SMTP host to connect to. Also accepts optional **-port**, **-username**, **-password**, and **-timeout** parameters.

email_smtp->command(*-send=?, -expect=?, -multi=?, -read=?, -timeout=?*)
    Sends a raw command to the SMTP server. The **-send** parameter specifies the command to send. The **-expect** parameter specifies the numeric result code that is expected as a result. This method normally returns "true" or "false" depending on whether the expected result code was found. The **-read** parameter can be specified to have it return the result from the SMTP server.

email_smtp->send(*-from::string, -recipients::array, -message::string*)
    Sends a single message to the SMTP server. Requires a **-message** parameter with the MIME data for the message, **-recipients** with an array of recipient email address, and **-from** with the email address of the sender.

email_smtp->close()
    Closes the connection to the remote server.

email_mxlookup(*domain, -refresh=?, -hostname=?***)**

> Takes a domain as a parameter and returns a map that describes the MX server for the domain. The map includes the 'domain', 'host', 'username', 'password', 'timeout', and 'SSL' preference for the MX server.

36.6.1 Look Up an SMTP Server

Use the **email_mxlookup** method. This returns a map that describes the preferred MX server for the domain. An example lookup for Gmail is shown below. The first time an MX record is looked up it will be cached and the same information will be returned on subsequent lookups.

```
email_mxlookup('gmail.com')
// => map(domain = gmail.com, host = gmail-smtp-in.l.google.com, priority = 5)
```

36.6.2 Communicate with an SMTP Server

The **email_smtp** type can be used to send one or more messages directly to an SMTP server. In the following example a message is created using the **email_compose** type. That message is then sent to an example SMTP server "smtp.example.com" using an SMTP AUTH username and password. Once the message is sent the connection is closed.

This example does not perform any error checking and only sends one message. The actual source code for the built-in email sender background process presents a good example of how this code looks in a full working solution:

```
local(message) = email_compose(
    -to      = 'example@example.com',
    -from    = 'example@example.com',
    -subject = 'Example Message',
    -body    = 'Example Message'
)
local(smtp) = email_smtp

#smtp->open(
    -host     = 'smtp.example.com',
    -port     = 25,
    -username = 'SMTPUSER',
    -password = 'mysecretpassword',
    -timeout  = 60
)
#smtp->send(
    -from       = #message->from,
    -recipients = #message->recipients,
    -message    = #message->data + '\r\n'
)
#smtp->close
```

Retrieving Email

Lasso allows messages to be downloaded from an account on a POP email server. This enables developers to create solutions such as:

- A list archive for a mailing list
- A webmail interface allowing users to check POP accounts
- An auto-responder that can reply to incoming messages with information

Lasso's flexible POP implementation allows messages to be easily retrieved from a POP server with a minimal amount of coding. Additionally, Lasso allows the messages available on the POP server to be inspected without downloading or deleting them. Mail can be downloaded but left on the server so it can be checked by other clients (and deleted at a later point if necessary).

All messages are downloaded as raw MIME text. The **email_parse** type can be used to extract the different parts of the downloaded messages, inspect their headers, or extract attachments from them.

Note: Lasso does not support downloading email via the IMAP protocol.

37.1 Sending POP Commands

The **email_pop** type is used to establish a connection to a POP email server, inspect the available messages, download one or more messages, and mark messages for deletion.

37.1.1 POP Methods

The following describes the **email_pop** type and some of its member methods:

type **email_pop**

email_pop(*-server=?, -port=?, -username=?, -password=?, -APOP=?, -timeout=?, -log=?, -debug=?, -get=?, -host=?, -ssl=?,*
    *...***)**
> Creates a new POP connection object. Requires a **-host** parameter. Takes optional **-port** and **-timeout** parameters. The **-APOP** parameter selects authentication method. If **-username** and **-password** are specified then a connection is opened to the server with authentication. The **-get** parameter specifies which command to perform when calling **email_pop->get**.

email_pop->size()
> Returns the number of messages available for download.

email_pop->get(*command::string=?***)**
> Performs the command specified when the object was created. "UniqueID" by default, or can be set to "Retrieve", "Headers", or "Delete".

email_pop->retrieve(*position::integer=?***)**

email_pop->**retrieve**(*position::integer*, *maxLines::integer*)
> Retrieves the current message from the server. Optionally accepts a position to retrieve a specific message. Optional second parameter specifies the maximum number of lines to fetch for each email.

email_pop->**headers**(*position::integer=?*)
> Retrieves the headers of the current message from the server. Optionally accepts a position to get the headers of a specific message.

email_pop->**uniqueID**(*position::integer=?*)
> Retrieves the unique ID of the current message from the server. Optionally accepts a position to get the unique ID of a specific message.

email_pop->**delete**(*position::integer=?*)
> Marks the current message for deletion. Optionally accepts a position to mark a specific message.

email_pop->**close**()
> Closes the POP connection, performing any specified deletes.

email_pop->**cancel**()
> Closes the POP connection, but does not perform any deletes.

email_pop->**noOp**()
> Sends a ping to the server. Allows the connection to be kept open without timing out.

email_pop->**authorize**(*-username::string*, *-password::string*, *-APOP::boolean=true*)
> Requires a **-username** and **-password** parameter. Optional **-APOP** parameter specifies whether APOP authentication should be used or not. Opens a connection to the server if one is not already established.

37.1.2 Message Retrieval

The **email_pop** type is intended to be used with the **iterate** method to quickly loop through all available messages on the server. The **email_pop->size** method returns the number of available messages. The **email_pop->get** method fetches the "UniqueID" of the current message by default or can be set to "Retrieve" the current message, the "Headers" of the current message, or even to "Delete" the current message.

The **-host**, **-username**, and **-password** should be passed to the **email_pop** object when it is created. The **-get** parameter specifies what command the **email_pop->get** method will perform. In this case it is set to "UniqueID" (the default).

```
local(myPOP) = email_pop(
    -host     = 'mail.example.com',
    -username = 'POPUSER',
    -password = 'MySecretPassword',
    -get      = 'UniqueID'
)
```

The **iterate** method can then be used on the "myPOP" variable. For example, this code will download and delete every message from the target server. The variable "myID" is set to the unique ID of each message in turn. The **email_pop->retrieve** method fetches the current message and the **email_pop->delete** method marks it for deletion.

```
iterate(#myPOP, local(myID)) => {^
    #myID
    '<br />'
    #myPOP->retrieve
    #myPOP->delete
    '<hr />'
^}

// =>
// 000000025280dd26
```

```
// <br />
// Return-Path: <joe@example.com>
// X-Original-To: jane@example.com
// Delivered-To: jane@example.com
// Received: from mail.example.com (mail.example.com [127.0.0.1])
//     by mail.example.com (Postfix) with ESMTP id 1B11410A37
//     for <jane@example.com>; Mon, 11 Nov 2013 08:33:59 -0500 (EST)
// Received: (qmail 4313 invoked from network); 11 Nov 2013 08:36:28 -0500
// Message-ID: <5280DCC0.6070809@example.com>
// Date: Mon, 11 Nov 2013 08:33:52 -0500
// From: joe@example.com
// MIME-Version: 1.0
// To: jane@example.com
// Subject: Test
// Content-Type: text/plain; charset=ISO-8859-1; format=flowed
// Content-Transfer-Encoding: 7bit
//
// Testing
// <hr />
```

Both **email_pop->retrieve** and **email_pop->delete** could be specified with the current **loop_count** as a parameter, but it is unnecessary since they pick up the loop count from the surrounding **iterate** method. This example only downloads and displays the text of each message. Most solutions will use the **email_parse** type defined below to parse and process the downloaded :messages.

None of the deletes will actually be performed until the connection to the remote server is closed. The **email_pop->close** method performs all deletes and closes the connection. The **email_pop->cancel** method closes the connection, but cancels all of the marked deletes.

```
#myPOP->close
```

37.1.3 Using Email_Pop

This section includes examples of the most common tasks that are performed using the **email_pop** type. See the section *Parsing Email* for examples of downloading messages and parsing them for storage in a database.

Download and Delete All Emails from a POP Server

Open a connection to the POP server using **email_pop** with the appropriate host, username, and password. The following example shows how to use **email_pop->retrieve** and **email_pop->delete** to download and delete each message from the server:

```
local(myPOP) = email_pop(
    -host     = 'mail.example.com',
    -username = 'POPUSER',
    -password = 'MySecretPassword'
)

iterate(#myPOP, local(myID)) => {
    local(myMsg) = #myPOP->retrieve

    // ... process message ...

    #myPOP->delete
```

```
    }
    #myPOP->close
```

Each downloaded message can be processed using the techniques described in the section *Parsing Email* or can be stored in a database.

Leave Mail on Server and Only Download New Messages

In order to download only new messages it is necessary to store a list of all the unique IDs of messages that have already been downloaded from the server. This is usually done by storing the unique ID of each message in a database. As messages are inspected the unique ID is compared to see if the message is new or not. No deletion of messages is performed in this example.

For the purposes of this example, it is assumed that unique IDs are being stored in a variable array called "myUniqueIDs". For each waiting message this variable is checked to see if it contains the unique ID of the current message. If it does not then the message is downloaded and the unique ID is inserted into "myUniqueIDs".

```
    local(myPOP) = email_pop(
        -host     = 'mail.example.com',
        -username = 'POPUSER',
        -password = 'MySecretPassword'
    )
    iterate(#myPOP, local(myID)) => {
        #myUniqueIDs->contains(#myID) ? loop_continue

        #myUniqueIDs->insert(#myID)

        // ... process message ...

    }
    #myPOP->close
```

Inspect Message Headers

The **email_pop->headers** method can be used to fetch the headers of each waiting email message. This allows the headers to be inspected prior to deciding which emails to actually download. In the following example the headers are fetched with **email_pop->headers** and two variables, "needDownload" and "needDelete", are set to determine whether either action should take place.

```
    local(myPOP) = email_pop(
        -host     = 'mail.example.com',
        -username = 'POPUSER',
        -password = 'MySecretPassword',
        -get      = 'UniqueID'
    )
    iterate(#myPOP, local(myID)) => {
        local(needDownload) = false
        local(needDelete)   = false
        local(myHeaders)    = #myPOP->headers

        // ... process headers and set #needDownload or #needDelete to true ...

        #needDownload ? #myPOP->retrieve
        #needDelete ? #myPOP->delete
```

```
    }
#myPOP->close
```

The downloaded headers can be processed using the techniques described in the section *Parsing Email*.

37.2 Parsing Email

Each of the messages that are downloaded from a POP server is returned in raw MIME text form. This section describes the basic structure of email messages, the **email_parse** type that can be used to parse them into headers and parts, and finally gives some examples of parsing messages.

37.2.1 Email Structure

The basic structure of a simple email message is shown below. The message starts with a series of headers. The headers of the message are followed by a blank line, then the body of the message.

Each server that handles the message adds its own *Received* headers, so there may be many of them. The *Mime-Version*, *Content-Type*, and *Content-Transfer-Encoding* headers specify what type of email message it is and how it is encoded. The *Message-ID* is a unique ID given to the message by the email server. The *To*, *From*, *Subject*, and *Date* headers are all specified by the sending user in their email client (or in Lasso using **email_send**).

```
Received: From [127.0.0.1] BY example.com ([127.0.0.1]) WITH ESMTP;
    Thu, 08 Jul 2004 08:07:42 -0700
Mime-Version: 1.0
Content-Type: text/plain; charset=US-ASCII;
Message-Id: <8F6A8289-D0F0-11D8-B21D-0003936AD948@example.com>
Content-Transfer-Encoding: 7bit
From: Example Sender <example@example.com>
Subject: Test Message
Date: Thu, 8 Jul 2004 08:07:42 -0700
To: Example Recipient <example@example.com>

This is the email message!
```

The order of headers is unimportant and each header is usually specified only once (except for the *Received* headers which are in reverse chronological order). A header can be continued on the following line by starting the second line with a space or tab. Beyond those standard headers shown here, email messages can also contain many other headers identifying the sending software, logging spam and virus filtering actions, or even adding meta information like a picture of the sender.

A more complex email message is shown below. This message has a *Content-Type* of *multipart/alternative*. The body of the message is divided into two parts, one text part and one HTML part. The parts are divided using the boundary specified in the *Content-Type* header (*---=_NEXT_fda4fcaab6*).

Each of the parts is formatted similarly to an email message. They have several headers followed by a blank line and the body of the part. Each part has a *Content-Type* and a *Content-Transfer-Encoding* which specify the type part (either *text/plain* or *text/html*) and encoding.

```
Received: From [127.0.0.1] BY example.com ([127.0.0.1]) WITH ESMTP;
Thu, 08 Jul 2004 08:07:42 -0700
Mime-Version: 1.0
Message-Id: <14501276655.1089394748105@example.com>
From: Example Sender <example@example.com>
Subject: Test Message
Date: Thu, 8 Jul 2004 08:07:42 -0700
To: Example Recipient <example@example.com>
```

```
Content-Type: multipart/alternative; boundary="---=_NEXT_fda4fcaab6";


-----=_NEXT_fda4fcaab6
Content-Type: text/plain; charset=ISO-8859-1
Content-Transfer-Encoding: 8bit


This is the text part of the email message!


-----=_NEXT_fda4fcaab6
Content-Type: text/html; charset=ISO-8859-1
Content-Transfer-Encoding: 8bit


<html>
<body>
<h3>This is the HTML part of the email message!</h3>
</body>
</html>
-----=_NEXT_fda4fcaab6--
```

Attachments to an email message are included as additional parts. Typically, the file that is attached is encoded using Base64 encoding so it appears as a block of random letters and numbers. It is possible for one part of an email to itself have a *Content-Type* of *multipart/alternative* and its own boundary. In this way, very complex recursive email structures can be created.

Lasso allows access to the headers and each part (including recursive parts) of downloaded email messages through the **email_parse** type.

37.2.2 Parsing Methods

The **email_parse** type requires the raw MIME text of an email message as a parameter when it is created. It returns an object whose member methods can be used to inspect the headers and parts of the email message. Outputting an **email_parse** object to the page will result in a message formatted with the most common headers and the default body part. An **email_parse** object can be used with the **iterate** method to inspect each part of the message in turn.

type **email_parse**

email_parse(_mime::string_**)**
> Parses the raw MIME text of an email. Requires a single string parameter. Outputs the raw data of the email if displayed on the page or converted to a string.

email_parse->headers()
> Returns an array of pairs containing all the headers of the message.

email_parse->header(_name::string, …_**)**
> Returns a single specified header. Requires one parameter, the name of the header to be returned. See also the shortcuts for specific headers listed below. If **-extract** is specified then any comments in the header will be stripped. If **-comment** is specified then only the comments will be returned. If **-safeEmail** is specified then the email address will be obscured for display on the web. If **-noDecode** is specified then the raw header is returned without Quoted-Printable or BinHex decoding. This method returns an array if multiple headers with the same name are found. Optionally, **-join** can be used to specify a character to be used to combine the values in the array into a string.

email_parse->mode()
> Returns the mode from the *Content-Type* for the message. Usually either text or multipart.

email_parse->body(_-type=void, -preamble=void, -array=void, …_**)**
> Returns the body of the message. Optional parameter specifies the preferred type of body to return (e.g. *text/plain* or *text/html*). If the body is encoded using Quoted-Printable or Base64 encoding then it is automatically decoded before being returned by this method.

email_parse->**size()** → integer
> Returns the number of parts in the message.

email_parse->**get(**_position::integer_**)**
> Returns the specified part of the message. Requires a position parameter. The part is returned as an **email_parse** object that can be further inspected.

email_parse->**data()**
> Returns the raw data of the message.

email_parse->**rawHeaders()**
> Returns the raw data of the headers.

email_parse->**recipients()**
> Returns an array containing all of the email addresses in the _To_, _Cc_, and _Bcc_ headers.

email_parse->**to(**…**)**

email_parse->**from(**…**)**

email_parse->**cc(**…**)**

email_parse->**bcc(**…**)**

email_parse->**subject()**

email_parse->**date()**

email_parse->**content_type()**

email_parse->**boundary()**

email_parse->**charset()**

email_parse->**content_disposition()**

email_parse->**content_transfer_encoding()**
> These methods are shortcuts that return the value for the corresponding header from the email message. The table below maps the method to the header. (The Bcc header will always be empty for received emails.)

Table 37.1: Email Header Methods

Email Header Method	Email Header
email_parse->to	_To_
email_parse->from	_From_
email_parse->cc	_Cc_
email_parse->bcc	_Bcc_
email_parse->subject	_Subject_
email_parse->date	_Date_
email_parse->content_type	_Content-Type (MIME Type)_
email_parse->boundary	_Content-Type (boundary)_
email_parse->charset	_Content-Type (charset)_
email_parse->content_disposition	_Content-Disposition_
email_parse->content_transfer_encoding	_Content-Transfer-Encoding_

> The methods **to**, **from**, **cc**, and **bcc** also accept **-extract**, **-comment**, and **-safeEmail** parameters like the **email_parse->header** method. These methods join multiple parameters by default, but **-join=null** can be specified to return an array instead.

37.2.3 Using Email_Parse

This section includes examples of the most common tasks that are performed using the **email_parse** type. See the preceding section on the *email_pop type* for examples of downloading messages from a POP email server.

Display a Downloaded Message

Simply use the **email_parse** type on the downloaded message and display it on the page. The **email_parse** object will output a formatted version of the email message including a plain text body if one exists.

The following example shows how to download and display all the waiting messages on an example POP mail server. The unique ID of each downloaded message is shown as well as the output of **email_parse** in a set of **<pre>** tags.

```
<?lasso
    local(myPOP) = email_pop(
        -host     = 'mail.example.com',
        -username = 'POPUSER',
        -password = 'MySecretPassword'
    )
    iterate(#myPOP, local(myID))
        local(myMsg) = #myPOP->retrieve
?>
<h3>Message: [#myID]</h3>
<pre>[email_parse(#myMsg)]</pre>
<hr />
<?lasso
    /iterate
    #myPOP->close
?>


// =>
// <h3>Message: 000000045280dd26</h3>
// <pre>Date: Mon 11 Nov 2008 9:0:0 -0500
// From: joe@example.com
// To: jane@example.com
// Subject: Test
// Content-Type: text/plain; charset=ISO-8859-1; format=flowed
// Content-Transfer-Encoding: 7bit
//
// Just Testing
// </pre>
// <hr />
```

Inspect Headers of a Downloaded Message

There are three ways to inspect the headers of a downloaded message.

1. The basic headers of a message can be inspected using the shortcut methods such as **email_parse->from**, **email_parse->to**, **email_parse->subject**, etc. The following example shows how to display the basic headers for a message, where the variable "myMsg" is assumed to be the output from an **email_pop->retrieve** method:

```
local(myParse) = email_parse(#myMsg)
'<br />To:      ' + #myParse->to->encodeHtml + '\n'
'<br />From:    ' + #myParse->from->encodeHtml + '\n'
'<br />Subject: ' + #myParse->subject->encodeHtml + '\n'
'<br />Date:    ' + #myParse->date->asString->encodeHtml + '\n'
```

```
// =>
// <br />To: Example Recipient
// <br />From: Example Sender
// <br />Subject: Test Message
// <br />Date: Thu 8 Jul 2004 08:07:42 -0700
```

These headers can be used in conditionals or other code as well. For example, this conditional would perform different tasks based on whether the message is to one address or another:

```
local(myParse) = email_parse(#myMsg)
if(#myParse->to >> 'mailinglist@example.com') => {
    // ... store the message in the mailing list database ...
else(#myParse->to >> 'help@example.com')
    // ... forward the message to technical support ...
else
    // ... unknown recipient ...
}
```

2. The value for any header, including application-specific headers, headers added by mail processing gateways, etc. can be inspected using the **email_parse->header** method. For example, the following code can check whether the message has SpamAssassin headers:

```
local(myParse)       = email_parse(#myMsg)
local(spam_version) = string(#myParse->header('X-Spam-Checker-Version'))
local(spam_level)   = string(#myParse->header('X-Spam-Level'))
local(spam_status)  = string(#myParse->header('X-Spam-Status'))
'<br />Spam Version: ' + #spam_version->encodeHtml + '\n'
'<br />Spam Level:   ' + #spam_level->encodeHtml + '\n'
'<br />Spam Status:  ' + #spam_status->encodeHtml + '\n'

// =>
// <br />Spam Version: SpamAssassin 2.61
// <br />Spam Level:
// <br />Spam Status: No, hits=-4.6 required=5.0 tests=AWL,BAYES_00 autolearn=ham
```

The spam status can then be checked with a conditional in order to ignore any messages that have been marked as spam (note that the details will depend on what server-side spam checker is being used and which version).

```
if(#spam_status >> 'Yes') => {
    // ... message is spam ...
else
    // ... message is not spam ...
}
```

3. The value for all the headers in the message can be displayed using the **email_parse->headers** method, as the following example shows:

```
local(myParse) = email_parse(#myMsg)
iterate(#myParse->headers, local(header))
    '<br />' + #header->first->encodeHtml + ': ' + #header->second->encodeHtml
/iterate

// =>
// <br />Received: From [127.0.0.1] BY example.com ([127.0.0.1]) WITH ESMTP;
//    Thu, 08 Jul 2004 08:07:42 -0700
// <br />Mime-Version: 1.0
// <br />Content-Type: text/plain; charset=US-ASCII;
```

```
// <br />Message-Id: <8F6A8289-D0F0-11D8-B21D-0003936AD948@example.com>
// <br />Content-Transfer-Encoding: 7bit
// <br />From: Example Sender <example@example.com>
// <br />Subject: Test Message
// <br />Date: Thu, 8 Jul 2004 08:07:42 -0700
// <br />To: Example Recipient <example@example.com>
```

Locate Parts of a Downloaded Message

The **email_parse->body** method can be used to find the plain text and HTML parts of a message. The following example shows both the plain text and HTML parts of a downloaded message:

```
local(myParse) = email_parse(#myMsg)
'<pre>' + #myParse->body(-type='text/plain')->encodeHtml + '</pre>'
'<hr />' + #myParse->body(-type='text/html')->encodeHtml + '<hr />'
```

The **email_parse->size** and **email_parse->get** methods can be used with the **iterate** method to inspect every part of an email message in turn. This will show information about plain text and HTML parts as well as information about attachments. The headers and body of each part is shown:

```
local(myParse) = email_parse(#myMsg)
iterate(#myParse, local(myPart))
    iterate(#myPart->header, local(header))
        '<br />' + #header->first->encodeHtml + ': ' + #header->second->encodeHtml + '\n'
    /iterate
    '<br />' + #myPart->body->encodeHtml + '\n'
    '<hr />\n'
/iterate

// =>
// <br />Content-Type: text/plain; charset=ISO-8859-1
// <br />Content-Transfer-Encoding: 8bit
// <br />This is the text part of the email message!
// <hr />
// <br />Content-Type: text/html; charset=ISO-8859-1
// <br />Content-Transfer-Encoding: 8bit
// <br />&lt;html&gt;
// &lt;body&gt;
// &lt;h3&gt;This is the HTML part of the email message!&lt;/h3&gt;
// &lt;/body&gt;
// &lt;/html&gt;
// <hr />
```

Extract Attachments of a Downloaded Message

Attachments of a multipart message appear as parts with a *Content-Disposition* of "attachment". The name of the attachment can be found by looking at the "name" field of the *Content-Type* header. The data for the attachment is returned as the body of the part.

The attachments can be extracted and written out as files that re-create the attached file, or they can be stored in a database, processed by the **image** methods, or served immediately using **web_response->sendFile**.

The following example finds all of the attachments for a message using the **iterate** method to cycle through each part in the message and inspect the *Content-Disposition* header using **email_parse->content_disposition**. The name

(`email_parse->content_type('name')`) and data (`email_parse->body`) of each part that includes an attachment is used to write out a file using **file->openWrite** and **file->writeBytes** which re-creates the attachment.

```
local(myParse) = email_parse(#myMsg)
if(#myParse->mode >> 'multipart') => {
    iterate(#myParse, local(myPart)) => {
        if(#myPart->content_disposition >> 'attachment') => {
            local(myFile)     = file('/Attachments/' + #myPart->content_type('name'))
            local(myFileData) = #myPart->body
            #myFile->doWithClose => {
                #myFile->openWrite&writeBytes(#myFileData)
            }
        }
    }
}
```

Note: In order for this code to work, the "Attachments" folder must already exist and have permissions allowing Lasso Server to write to it.

Store a Downloaded Message in a Database

Messages can be stored in a database in several different ways depending on how the messages are going to be used later.

- The simple headers and body of a message can be stored by calling **email_parse->asString** directly in an inline:

```
local(myPOP) = email_pop(
    -host     = 'mail.example.com',
    -username = 'POPUSER',
    -password = 'MySecretPassword'
)
handle => {
    #myPOP->close
}
iterate(#myPOP, local(myID)) => {
    local(myMsg)   = #myPOP->retrieve
    local(myParse) = email_parse(#myMsg)

    inline(
        -add,
        -database='example',
        -table='archive',
        'email_format'=#myParse->asString
    ) => {}
}
```

- Often it is desirable to store the common headers of the message in individual fields as well as the different body parts. This example shows how to do this:

```
local(myPOP) = email_pop(
    -host     = 'mail.example.com',
    -username = 'POPUSER',
    -password = 'MySecretPassword'
)
handle => {
    #myPOP->close
}
```

```
iterate(#myPOP, local(myID)) => {
    local(myMsg)   = #myPOP->retrieve
    local(myParse) = email_parse(#myMsg)

    inline(
        -add,
        -database        = 'example',
        -table           = 'archive',
        'email_format'  = #myParse->asString,
        'email_to'      = #myParse->to,
        'email_from'    = #myParse->from,
        'email_subject' = #myParse->subject,
        'email_date'    = #myParse->date,
        'email_cc'      = #myParse->cc,
        'email_text'    = #myParse->body(-type='text/plain'),
        'email_html'    = #myParse->body(-type='text/html')
    ) => {}
}
```

- The raw text of messages can be stored using **email_parse->data**. It is generally recommended that the raw text of a message be stored in addition to a more friendly format. This allows additional information to be extracted from the message later if required.

```
local(myPOP) = email_pop(
    -host     = 'mail.example.com',
    -username = 'POPUSER',
    -password = 'MySecretPassword'
)
handle => {
    #myPOP->close
}
iterate(#myPOP, local(myID)) => {
    local(myMsg)   = #myPOP->retrieve
    local(myParse) = email_parse(#myMsg)
    inline(
        -add,
        -database     = 'example',
        -table        = 'archive',
        'email_text' = #myParse->asString,
        'email_raw'  = #myParse->data
    ) => {}
}
#myPOP->close
```

Ultimately, the choice of which parts of the email message need to be stored in the database will be solution dependent.

37.3 Email Helper Methods

The email methods use a number of helper methods for their implementation. The following describes a number of these methods and how they can be used independently.

email_extract()
> Strips all comments out of a MIME header. If specified with a **-comment** parameter returns the comments instead. Used as a utility method by **email_parse->header**.

> **email_extract** allows the different parts of email headers to be extracted. Email headers containing email addresses

are often formatted in one of the three formats below:

```
john@example.com
"John Doe" <john@example.com>
john@example.com (John Doe)
```

In all three of these cases the **email_extract** method will return "*john@example.com*". The angle brackets in the second example identify the email address as the important part of the header. The parentheses in the third example identify that portion of the header as a comment.

If **email_extract** is called with the optional **-comment** parameter then it will return "*john@example.com*" for the first example and "John Doe" for the two following examples.

email_findEmails()

Returns an array of all email addresses found in the input. Used as a utility method by **email_parse->recipients**.

email_safeEmail()

This method is used as a utility method by **email_parse->header**. It obscures an email address by returning the comment portion or only the username before the "@" character, and can be used to safely display email headers on the web without attracting email address harvesters. This method returns the following output for the example headers above:

```
// =>
// john
// John Doe
// John Doe
```

email_translateBreaksToCRLF()

Translates all return characters and line feeds in the input into **"\r\n"** pairs.

DNS

DNS (Domain Name System) is an essential part of the Internet's infrastructure for mapping people-friendly domain names like "www.lassosoft.com" to machine-friendly IP addresses like "127.0.0.1". Every URL entered into a web browser or email address entered into an email client requires consulting the DNS system to determine which server to submit the request or route the message to.

DNS servers can handle many different types of requests. Some of the most common are listed here:

*
Returns all available information about the domain name. The results of this type of request are returned in human-readable form.

A
This is the most common type of request and simply returns the IP address that corresponds with the domain name.

CNAME
This is a request for the common name associated with a domain name.

MX
This is a request for the mail server that is associated with a domain name. A prioritized list of mail servers are returned.

NS
This is a request for the name servers responsible for providing definitive information about the domain name.

PTR
This type of request allows a reverse lookup to be performed, returning the domain name associated with an IP address.

TXT
Domain name servers can store additional information about a domain name. Specially formatted domain names are sometimes used as keys that will return useful information when queried with this option.

Any query can return either a single value or an array of values. For example, a single domain name may be served by a collection of web servers. When the A record for that domain name is looked up, a list of servers will be returned. The DNS server may round-robin the list of servers so a different server is on top for each request. This effectively spreads traffic among all the servers in the pool more or less evenly.

38.1 Domain Names

Domain names are written as a series of words separated by periods. Reading from left to right the domain name gets progressively more general. In a typical three word domain name like "www.lassosoft.com" the first word represents a particular machine or a particular service, the second word represents the domain in which the machine or service resides, and the third word represents the top-level domain that has authorized the use of the domain name.

Top-level domains are controlled by an organization that has been designated by the **IANA** (Internet Assigned Name Authority). Two common, general-purpose top-level domains are ".com" and ".net", ".edu" is a top-level domain reserved for educational institutions, ".gov" is a top-level domain reserved for U.S. government institutions, ".org" is a top-level domain intended for non-profit organizations.

Each country has its own top-level domain defined by its standard two letter abbreviation, e.g. ".us" is the top-level domain for the United States, ".uk" is the top-level domain for the United Kingdom, and ".cn" is the top-level domain for China. The domain ".tv", frequently used to refer to television, is actually the country domain for Tuvalu. Each country decides how it wants to assign domain names within their own top-level domain. Some have created virtual top-level domains like ".com.uk", ".org.uk", ".edu.uk", etc.

38.2 IP Addresses

IPv4 addresses consist of four numbers from 0 to 255 separated by periods. Each number represents a single 8-bit integer and the entire IP address represents a 32-bit integer, so there are effectively about 4 billion IPv4 addresses. A typical IPv4 address appears as follows:

```
17.149.160.49
```

In order to expand the range of IP addresses that are available, a new Internet Protocol has been designed and is in the process of being adopted. This is version 6 of the Internet Protocol and is abbreviated IPv6. The most recent versions of Windows, OS X, and Linux all support IPv6 addresses. IPv6 addresses are essentially 128-bit integers. A typical IPv6 address may appear as follows, though abbreviated forms also exist:

```
2001:0db8:0000:0000:0000:ff00:0042:8329
```

Note: The DNS lookup methods in Lasso do not support IPv6 addresses at this time.

38.3 Querying for DNS Records

DNS queries are performed with the **dns_lookup** method.

dns_lookup(name::string, …**)**

> This method is used to query a DNS server for information about a specified domain name. It requires one parameter, the domain name being queried. The optional parameters are described in below. This method will return either a string, array, or **dns_response** object.

> Parameters

>> • **name** (*string*) – The domain name being queried.

>> • **-type** – The type of data to look up. Defaults to "*" if the name parameter is a domain name or "PTR" if it is an IP address. Possible values include "*", "A", "NS", "MD", "MF", "CNAME", "SOA", "MB", "MG", "MR", "NULL", "WKS", "PTR", "HINFO", "MINFO", "MX", "TXT", "AXFR", "MAILB", "MAILA".

>> • **-class** – The class in which to perform the lookup. Defaults to "IN" which represents the Internet DNS system. Searching other classes is very rare. Possible values include "*", "IN", "CS", "CH".

>> • **-noRecurse** (*boolean*) – By default the local DNS server will automatically query other DNS servers to find the answer to a request. If this parameter is included then the query will only return information that is known directly by the local DNS server.

>> • **-inverse** (*boolean*) – Sets the inverse bit in the DNS query.

>> • **-status** (*boolean*) – Sets the status bit in the DNS query.

>> • **-showQuery** (*boolean*) – If specified the query is not actually performed, but a **dns_response** object representing the query is returned.

- **-formatQuery** (*boolean*) – If specified the query is not actually performed, but a string describing the constructed query is returned.

- **-bitQuery** (*boolean*) – If specified the query is not actually performed, but a string is returned that shows the low-level bit representation of the constructed query.

- **-showResponse** (*boolean*) – If specified the response is returned as **dns_response** object that can be inspected using the member methods described in the documentation below.

- **-format** (*boolean*) – If specified a string is returned that describes the response from the DNS server.

- **-bitFormat** (*boolean*) – If specified a string is returned that shows the low-level bit representation of the response from the DNS server.

- **-hostname** – Allows you to specify the name of a specific DNS server to query. Defaults to the DNS server set up in the OS.

- **-port** (*integer*) – The port of the DNS server to connect to when doing a DNS lookup.

- **-timeout** (*integer*) – How long to wait for a response when doing a DNS lookup.

38.3.1 IP Lookup

The following example looks up the associated IP address(es) for a specified domain name. Using a **-type** of "A" will always return an array, even if there is only one IP address. An empty array will be returned if no information about the specified domain name can be found.

```
dns_lookup('www.apple.com', -type='A')
// => array(17.149.160.49, 17.178.96.59, 17.172.224.47)
```

38.3.2 Reverse Lookup

Reverse lookups are performed when an IP address is passed to the **dns_lookup** method, or when the "PTR" type is specified, and return an array of domain names. An empty array will be returned if no domain name could be found for the specified IP address.

```
dns_lookup('23.208.45.15')
// => array(a23-208-45-15.deploy.static.akamaitechnologies.com)
```

38.3.3 MX Records Lookup

"MX" lookups return an array of pairs. The first element of each pair is a priority and the second element of each pair is an IP address. The mail servers should be used in order of priority to provide fallback if the preferred mail servers cannot be reached.

```
dns_lookup('lassosoft.com', -type='MX')
// => array((10 = smtp1.lassosoft.com), (15 = smtp2.lassosoft.com))
```

38.3.4 Return Different Formats

The following output shows the human-readable response to a DNS request:

```
dns_lookup('www.apple.com', -format)

// =>
// Length: 73
// ID: 32569
// Type: Answer
// Flags: RD, RA
// Counts: QD 1, AN 1
// QD 1: www.apple.com.. * IN
// AN 1: www.apple.com.. CNAME IN 1331 www.isg-apple.com.akadns.net..
```

The following output shows the low-level bit formatting of a DNS response. The actual response is fairly long and not shown here:

```
dns_lookup('www.lassosoft.com', -bitFormat)

// =>
// ASCII
// 3  T  X
// ... rest of response ...
```

38.4 DNS Response Helper Type

The **dns_response** type is a helper type which is used to format both DNS requests and responses. Normally a value of this type will only be returned from the **dns_lookup** method when **-showResponse** is specified. However, this type can also be used to parse raw DNS requests or responses if necessary.

type **dns_response**

dns_response(*message::bytes*)
>   Create a new **dns_response** object. An object of this type can be returned from the **dns_lookup** method when **-showResponse** is specified.

dns_response->**format()**
>   Returns a formatted display of the entire response from the DNS server.

dns_response->**bitFormat()**
>   Returns a formatted display of the raw bits returned by the DNS server.

dns_response->**answer()**
>   Returns an array of answers for most DNS responses. Address lookups or reverse lookups will return an array of IP addresses or host names. MX record lookups will return an array of pairs, each with a priority and an IP address. Other lookups may return an array of strings or other data.

dns_response->**data()**
>   Returns the response as a raw byte stream.

LDAP

LDAP (Lightweight Directory Access Protocol) is an industry-standard protocol for publishing directory information within an organization. LDAP servers are used for many different tasks, such as publishing the contact information for employees or other publicly accessible information. LDAP servers are also used to publish authentication information so all servers within an organization can use the same usernames and passwords.

An LDAP server provides access to a "directory information tree" (DIT). Each element in the tree is called an "entry" and has several attributes. Any element in the tree can be found using its "distinguished name" (DN). The distinguished name is like the path to a file in an operating system. For example, the DN of the record for John Doe in the directory might be "cn=John Doe, ou=People, o=LassoSoft".

The DN is made up of three parts separated by commas. Each part of the DN is called a "relative distinguished name" (RDN) and must be unique for all entries at that level. The RDN functions much like a primary key and includes one or more name/value pairs that uniquely identify the element from all of its siblings.

The attributes of each entry make up the data of the entry. Every entry will have an "objectClass" that tells what kind of entry it is. The remainder of the attributes will be determined by the type of directory that is being searched, but may include first name, last name, email address, phone number, etc. The attributes are often named with one or two-character abbreviations like "cn" for combined name, "ln" for last name, "fn" for first name, or "ou" for operational unit. Attributes may also have longer names like "email", "telephonenumber", and so on.

39.1 LDAP Searches

A search is defined starting at a DN within the directory tree. This DN will usually be provided by the LDAP server administrator. The scope allows the search to be limited to the object itself (i.e., is the object contained within the tree?), children of the object, or the entire tree below the object. Some possible DNs are shown below:

```
dc=lassosoft, dc=com
ou=People, o=LassoSoft
```

The search query is defined by the filter, which is a series of query terms (attributes and values) joined by logical operators. The most basic filter specifies that all objects in the tree should be returned:

```
(objectClass=*)
```

This is actually a special case of the "exists" filter. This filter returns any entries that have a defined objectClass. Similarly, all entries that have a full name attribute ("cn") could be found with this filter:

```
(cn=*)
```

A filter can specify an attribute name, operator, and value. Any of the attributes of the entries in the directory tree can be used in the filter. The operators include "equals" (=), "sounds like" (~=), "greater than" (>=), and "less than" (<=). The equals operator supports the asterisk (*) as a wildcard character, allowing for "contains", "begins with", and "ends with" searches. Operators for "greater than" (>) and "less than" (<) may only be supported on numeric fields. For example, the following simple filters would find all entries whose full name starts with "John", ends with "Doe", or are exactly "John Doe":

```
(cn=John*)
(cn=*Doe)
(cn=John Doe)
```

Two or more filters can be combined using the logical operators "and" (**&**) or "or" (**|**), or a filter can be negated using "not" (**!**). The following three filters would find all entries who have a first name of "John" and a last name of "Doe", a first name of "John" or a last name of "Doe", and a first name that is not "John" and a last name that is not "Doe":

```
(& (cn=John*) (cn=*Doe))
(| (cn=John*) (cn=*Doe))
(& (! (cn=John*)) (! (cn=*Doe)))
```

Note that there are no quotes around the values in the filters. The parentheses are used to delimit the values. In order to find a value that contains parentheses, an asterisk ("*****"), a backslash ("****"), or a null character, the following escape sequences can be used: **\2a** for "**(**", **\28** for "**)**", **\29** for "*****", **\5c** for "****", and **\00** for null.

39.2 LDAP Results

The results of an LDAP search will be an array of pairs. The first element of each pair will be the DN of the entry. The second element of each pair will be an array of pairs including the attribute names and values for the entry. For example, a search that found entries for "John Doe" and "Jane Doe" could contain the following elements:

```
(:
    pair('cn=John Doe, ou=People, o=LassoSoft' = (:
        pair('cn'='John Doe'),
        pair('mail'='john@example.com')
    )),
    pair('cn=Jane Doe, ou=People, o=LassoSoft' = (:
        pair('cn'='Jane Doe'),
        pair('mail'='jane@example.com')
    ))
)
```

LDAP allows the results to be customized in two ways. A list of desired attributes can be passed with the search. The results will only include those attributes. An asterisk wildcard (*****) specifies that all attributes should be returned (the default). A plus sign wildcard (**+**) specifies that only operational attributes should be returned (these are attributes that are generally used internally by the LDAP directory). Finally, a flag allows only attribute names to be returned without any values. By default both attribute names and values are returned.

39.3 LDAP Methods

The **ldap** type can be used to create a connection to an LDAP server and then to send queries to the server.

type **ldap**

ldap(…)
> Creates a new ldap object. Accepts an optional host name and port to immediately open a connection to a server.

ldap->open(…)
> Opens a connection to an LDAP server. Requires a host name and optionally a port.

ldap->authenticate(…)
> Logs into the LDAP server. Requires a username and password.

`ldap->`**`search`**`(…)`

> Performs a search on the remote LDAP server. Requires a parameter specifying the base of the query. Additional parameters specify the scope, filter, attributes, and attributes-only option for the query. See the following list for details about these parameters. Returns no value.

> Parameters

> > - **base** – The DN of the entry at which to start the search. Required.
> > - **scope** – The scope of the search. Optional. This parameter should be one of the following methods:
> > > - `ldap_scope_base` – Search the object itself.
> > > - `ldap_scope_onelevel` – Search the object's immediate children.
> > > - `ldap_scope_subtree` – Search the object and all its descendants.
> > - **filter** – The filter to apply to the search. Optional.
> > - **attributes** – An array of strings specifying the attribute types to return in the search results. Optional.
> > > - `*` (asterisk) may be specified in the array to indicate that all attributes are to be returned.
> > > - `+` (plus sign) may be specified in the array to indicate that all operational attributes should be returned.
> > > - `1.1` may be specified in the array to indicate that no attributes should be returned.
> > - **attribute-only** – A boolean indicating that only attributes and no values should be returned. Defaults to "false". Optional.

`ldap->`**`results`**`()`

> Returns results from the last search operation as an array containing a series of nested array and pair values. Each element in the top level array is a pair representing an entry found in the search. The first element of the pair is the DN of the found entry. The second element of the pair is an array of pairs containing the entry's attribute names and values.

`ldap->`**`referrals`**`()`

> Returns an array of referral strings if any are generated by the server.

`ldap->`**`code`**`()`

> Returns the code generated by the previous operation. A code of "0" means success. The most common codes are listed in the table below.

`ldap->`**`close`**`(…)`

> Closes the connection to the LDAP server.

For example, the following code performs an LDAP query against a server "ldap.example.com". The base of the query is `'dc=example,dc=com'`. The scope is `ldap_scope_subtree` indicating that the object and all of its descendants should be searched. The filter is `'(objectClass=*)'` indicating that all object classes are to be returned. The filter attribute is "*" indicating that all attributes are to be returned. And, the "attribute-only" parameter is automatically set to "false" indicating that both attributes and values should be returned. After each line is executed the return code is verified to be "0" indicating success. If the result code is greater than "0" then an error is reported.

```
local(my_ldap) = ldap
#my_ldap->open('ldap.example.com')
fail_if(#my_ldap->code != 0, #my_ldap->code, 'LDAP Error ' + #my_ldap->code)
#my_ldap->authenticate('myusername', 'mysecretpassword')
fail_if(#my_ldap->code != 0, #my_ldap->code, 'LDAP Error ' + #my_ldap->code)
#my_ldap->search('dc=example,dc=com', ldap_scope_subtree, '(objectClass=*)')
fail_if(#my_ldap->code != 0, #my_ldap->code, 'LDAP Error ' + #my_ldap->code)
local(my_result) = #my_ldap->results
#my_ldap->close
```

The result of this operation will be a staticarray of pairs. The first element of each pair is the DN of the entry. The second element of each pair is a staticarray of pairs containing the names and attributes of the element.

Table 39.1: Common LDAP Status Codes

Code	Description
0	Success (No Error)
1	Operations Error
2	Protocol Error
3	Time Limit Exceeded
4	Size Limit Exceeded
5	Compare False
6	Compare True
7	Auth Method Not Supported
8	Strong Auth Required
10	Referral
11	Admin Limit Exceeded
12	Unavailable Critical Extension
13	Confidentiality Required
14	SASL Bind In Progress
16	No Such Attribute
17	Undefined Attribute Type
18	Inappropriate Matching
19	Constraint Violation
20	Attribute Or Value Exists
21	Invalid Attribute Syntax
32	No Such Object
33	Alias Problem
34	Invalid DN Syntax
36	Alias Dereferencing Problem
48	Inappropriate Authentication
49	Invalid Credentials
50	Insufficient Access Rights
51	Busy
52	Unavailable
53	Unwilling To Perform
54	Loop Detect
64	Naming Violation
65	Object Class Violation
66	Not Allowed On Non-Leaf
67	Not Allowed On RDN
68	Entry Already Exists
69	Object Class Mods Prohibited
71	Affects Multiple DSAs
80	Other

Networking Protocols and Named Pipes

Lasso provides objects for TCP, TCP/SSL and UDP networking. It also provides objects for local communications over named pipes. These networking objects are designed to fit tightly into the language runtime's threading model. Each method call that might block accepts a timeout parameter. All such timeouts are in seconds.

40.1 TCP

TCP (Transmission Control Protocol) networking is provided through the **net_tcp** type. Objects of this type represent either the client or the server end of a connection.

40.1.1 Creating net_tcp Objects

type **net_tcp**

net_tcp()

net_tcp(_fd::filedesc_**)**
> A **net_tcp** object is created with no parameters. Once an object is obtained it can be used to open or accept TCP connections. Alternatively, can be passed a **filedesc** object that it will use to read and write data.

40.1.2 Opening TCP Connections

net_tcp->connect(_to::string, port::integer, timeout::integer=4_**)**
> Opens a TCP connection to the indicated server. TCP connections are made based on an address string and a port number. A server must be listening at the address and port before connections can be made to it. The address can be either a host name or an IP address. The addresses "0.0.0.0" or "127.0.0.1" can be used for local connections.
>
> If the connection succeeds this method will return "true", otherwise it returns "false". The method does not cause a failure if the connection cannot be made. By default, this method will timeout after 4 seconds and return "false" if a connection cannot be made. It will return faster than that in cases where the indicated server is not on the network or has no server listening on the indicated port. This timeout is more likely to be hit when connecting to a server that is available but under heavy load and not processing new connections in a timely manner. This timeout value can be tailored for the expected network conditions. A value of "-1" indicates no timeout.

40.1.3 Accepting TCP Connections

A TCP server listens on a specific port for client connections. Once a client connects, a new **net_tcp** object is returned for that connection. There are several steps for establishing a server. The series of methods is generally: **bind**, **listen** and then either

accept or **forEachAccept**.

net_tcp->**bind**(*port::integer*, *address::string='0.0.0.0'*)

net_tcp->**listen**(*backlog::integer=128*)

> When acting as a server, the **net_tcp** object must first be bound to a local port and optional address. The address can be ignored in most cases, but is useful on machines that have multiple network interfaces. The bind can be called before a client connection is made as well, however the operating system will automatically bind a client connection to a random port if it is not already bound, so binding a client connection is usually skipped.

> When creating a server, **listen** is called after **bind**. This method allows the new object to begin accepting client connections.

net_tcp->**accept**(*timeoutSeconds::integer=-1*)

net_tcp->**forEachAccept()**

> After a **net_tcp** object has been bound and is listening, client connections can then be accepted. The **accept** method is called to accept one connection. The process of accepting a connection does not actually establish a connection; instead, a new object is returned for that connection. Usually, the new connection should be passed to the new thread. This permits the server's thread to continue accepting new connections in a loop while the newly accepted connection is free to handle itself independently.

> By default, **accept** will wait forever for a client to connect. The timeout parameter can be used to have the call return null if no client has connected in that period.

> The **forEachAccept** method is used to accept connections in a loop. This method is called and given a capture. Each accepted connection will be passed to that capture to be handled.

40.1.4 Closing TCP Connections

net_tcp->**close()**

> TCP connections should be closed as soon as they are no longer needed. Once a **net_tcp** object has been closed it should not be used again.

net_tcp->**shutdownRd()**

net_tcp->**shutdownWr()**

net_tcp->**shutdownRdWr()**

> These methods give greater control over closing the connection at the TCP level. Respectively, these methods close down communications channels for the read, write, or read and write directions. A **close** should still be called after a shutdown.

40.1.5 Reading TCP Data

net_tcp->**readSomeBytes**(*count::integer*, *timeoutSeconds::integer*)

> Attempts to read up to the indicated number of bytes. If any bytes are immediately available then those will be returned and may be fewer than the requested amount. The timeout parameter controls how long the method will wait for data if there is none to be read. The method will return "null" if the timeout is reached.

40.1.6 Writing TCP Data

net_tcp->**writeBytes**(*data::bytes*, *offset::integer=0*, *length::integer=-1*)

> Attempts to send the indicated bytes. An optional zero-based **offset** parameter indicates how far in the bytes to skip before sending. An optional **length** parameter indicates how many bytes to sent. The default value of "-1" indicates that all the bytes should be sent.

This method returns the number of bytes that were sent. However, this number will always match the number of bytes requested to be sent. This method automatically handles TCP flow control, but does not accept a timeout value.

40.1.7 Simple Multi-Threaded Server

The example below creates a simple server that returns an HTTP response that simply echos back the request data it received.

```
local(server) = net_tcp
handle => { #server->close }

#server->bind(8080) & listen & forEachAccept => {
    // New client connection
    local(con) = #1

    // Move connection into new thread
    split_thread => {
        handle => { #con->close }
        local(request) = ''

        // Read in the entire request in chunks
        {
            #request->append(#con->readSomeBytes(8096))
            not #request->contains('\r\n\r\n') ? currentCapture->restart
        }()

        // Write out the HTTP response with the request in the body
        local(response) = 'HTTP/1.1 200 OK\r\n\
                Content-Type: text/html; charset=UTF-8\r\n\r\n\
                ' + #request
        #con->writeBytes(bytes(#response))
    }
}
```

While that server was running, if you were to open up a terminal shell on the same machine and execute `curl local-host:8080`, the following would be the result:

```
$> curl localhost:8080
GET / HTTP/1.1
User-Agent: curl/7.30.0
Host: localhost:8080
Accept: */*
```

40.2 TCP/SSL

SSL (Secure Sockets Layer) support is provided through the **net_tcp_ssl** type. This type inherits from **net_tcp**, so all of its methods are available plus a few SSL-specific additions. SSL is turned on and off for connections that are already established. When being used as a server, creating new **net_tcp_ssl** objects will return **net_tcp** objects with SSL turned on.

40.2.1 Creating net_tcp_ssl Objects

type **net_tcp_ssl**
> *Changed in version 9.2.6:* Renamed from **net_tcpssl**.

net_tcp_ssl()

net_tcp_ssl(_fd::filedesc_**)**
>   The first method creates and returns a new **net_tcp_ssl** object and accepts no parameters. The second creator method can be passed a **filedesc** object that will use to read and write data.

40.2.2 Loading SSL Certificates

net_tcp_ssl->loadCerts(_cert::string, privateKey::string_**)**
>   Accepts the file paths to a certificate file and a private key file. This method is required when creating a TCP SSL server. The paths should be full OS-specific paths to the files. This method calls through to the OpenSSL functions **SSL_CTX_use_certificate_chain_file** and **SSL_CTX_use_PrivateKey_file**. This method will fail if an error is returned from the OpenSSL functions, in which case the OpenSSL-specific error code and message will be set.

40.2.3 Beginning and Ending SSL Sessions

net_tcp_ssl->beginTLS(_timeoutSecs::integer=5_**)**
>   Begins SSL communications for the connection. Because starting SSL requires a series of communications between the two hosts, this method accepts a timeout parameter which will terminate the action if it takes too long to complete.
>
>   This method returns no value, but will fail if the underlying OpenSSL library produces an error.

net_tcp_ssl->endTLS()
>   Ends the SSL session and returns the connection to its non-SSL state. The connection is not terminated in any way.

40.2.4 Accepting SSL Connections

Accepting SSL connections is accomplished in the same manner as accepting non-SSL connections. However, serving SSL requires setting the certificate and private key files through the **net_tcp_ssl->loadCerts** method.

The **net_tcp_ssl** object supports both **accept** and **forEachAccept** just as **net_tcp** does. Accepting a connection using either of those methods will return a **net_tcp_ssl** object that has started the SSL session. Because some protocols require connections to be established first and then switched to SSL, **net_tcp_ssl** also provides an **acceptNoSSL** method.

net_tcp_ssl->acceptNoSSL(_timeoutSeconds::integer=-1_**)** → net_tcp_ssl
>   Accepts a new connection and returns a **net_tcp_ssl** object for it. This connection has not yet started an SSL session and operates just as a **net_tcp** connection would. SSL can be started via the **net_tcp_ssl->beginTLS** method.

40.3 UDP

UDP (User Datagram Protocol) is a connectionless protocol. It is used to transmit individual packets of data to a server.

40.3.1 Creating net_udp Objects

type **net_udp**

net_udp()

net_udp(_fd::filedesc_**)**
>   The first method accepts no parameters and returns a new **net_udp** object. Alternatively, a **filedesc** object that will be used to read and write data can be passed as a parameter.

40.3.2 Reading UDP Data

Reading UDP data requires first binding a **net_udp** object to a specific port and optional address. Once bound, data can be read through the **net_udp->readPacket** method which returns data as an object of type **net_udp_packet**. This contains the bytes sent as well as the address of the sender and the port from which it was sent.

net_udp->**readPacket(***maxBytes::integer*, *timeoutSeconds::integer=-1***)**
> Waits to receive a new UDP packet. The "maxBytes" parameter indicates the maximum size of data to receive. The number of bytes returned may be fewer than indicated, though individual packets will not be segmented. This value affects the size of the memory buffer allocated internally to hold incoming data.

> The timeout parameter indicates how long the method should wait before returning a "null" value. The default value of "-1" indicates that the method should wait forever.

> When successful, this method returns a **net_udp_packet** object.

type **net_udp_packet**

net_udp_packet(*bytes*, *name*, *port*)

net_udp_packet->**bytes()** → bytes
> Returns the bytes received.

net_udp_packet->**fromName()** → string
> Returns the server name that the data was sent from.

net_udp_packet->**fromPort()** → integer
> Returns the port that the data was sent from.

40.3.3 Writing UDP Data

With a **net_udp** object, data is sent one packet at a time to a particular address and port combination. The receivers must be waiting to accept packets from other hosts.

net_udp->**writeBytes(***b::bytes*, *toAddress::string*, *toPort::integer***)** → integer
> Sends the specified bytes to the indicated host. It returns the number of bytes that were sent.

40.3.4 Closing net_udp Objects

net_udp->**close()**
> Although **net_udp** objects do not maintain a connection, they must still be closed when they are no longer needed to free up resources.

40.4 Named Pipes

A *named pipe* is a means of communication between processes on a single local machine. One process begins listening on a pipe with a particular name. Other processes connect to that pipe and data is exchanged. The **net_named_pipe** type inherits from **net_tcp** and so all of the same methods for reading and writing bytes data are available. Named pipe usage differs in that the bind and connect methods takes a pipe name parameter (with no port number). The **net_named_pipe->accept** method will return a **net_named_pipe** object for the new connection.

The **net_named_pipe** objects are implemented as UNIX domain sockets on UNIX-based systems and as named pipes on Windows.

40.4.1 Creating net_named_pipe Objects

type **net_named_pipe**

net_named_pipe()

net_named_pipe(fd::filedesc**)**
> The first method accepts no parameters and returns a new **net_named_pipe** object. Alternatively, a **filedesc** object that will be used to read and write data can be passed as a parameter.

40.4.2 Opening Named Pipe Connections

net_named_pipe->**connect(**to::string, timeoutSeconds::integer=4**)**
> Attempts to connect to the indicated named pipe. This method returns "true" if the connection was made, and "false" otherwise.

40.4.3 Accepting Named Pipe Connections

net_named_pipe->**bind(**to::string**)**

net_named_pipe->**listen(**backlog::integer=128**)**

net_named_pipe->**accept(**timeoutSeconds::integer=-1**)**
> The **bind** method attempts to create a pipe with the given name. It accepts one parameter which is the name of the pipe to create. There can be only one listener on any given pipe name. The method will fail if there is a problem creating the pipe.
>
> The **listen** and **accept** methods operate as described for their **net_tcp** counterparts, except that **accept** will return new **net_named_pipe** objects for each new connection.

Part VII

Database Operations

Database Interaction Fundamentals

A database is the cornerstone of any significant web application. One of the primary applications of Lasso is to perform database actions and format the results of those actions. This chapter introduces the fundamentals of specifying database actions in Lasso.

41.1 Using Inlines

The **inline** method is used to specify a database action and to present the results of that action within a Lasso page. The database action is specified using keyword parameters passed to the inline. Additional name/value parameters specify the user-defined parameters of the database action. Each inline normally represents a single database action, but when using SQL statements a single inline can be use to perform batch operations as well. Additional actions can be performed in subsequent or nested **inline** methods.

inline(…)
> Performs the database action specified by the parameters. The results of the database action are available inside the required capture block or, if an **-inlineName** is specified, later on the page within **resultSet**, **records**, or **rows** methods.

> Parameters

>> - **-database** – Specifies the name of the database that will be used to perform the database action. If no **-host** is specified then the database is used to look up the data source specified in Lasso Admin for that database. Optional.

>> - **-host** – Specifies the connection parameters for a host within the inline. This provides an alternative to setting up data source hosts within Lasso Admin. Optional. See the table *Host Array Parameters* for the options available.

>> - **-inlineName** – Specifies a name for the inline. The same name can be used with **resultSet**, **records**, or **rows** methods to return the records from the inline later on in the page. Optional.

>> - **-statementOnly** – Specifies that the inline should generate the internal statement required to perform the action, but not actually perform the action. The statement can be fetched with **action_statement**. Optional.

>> - **-table** – Specifies the table that should be used to perform the database action. Most database actions require that a table be specified. The **-table** is used to determine what encoding will be used when interpreting database results, so a **-table** may be necessary even for an inline with an **-sql** action. Optional.

The results of the database action can be displayed within the contents of the inline's capture block using the **records** or **rows** methods along with **field** or **column** methods. Alternately, the inline can be named using **-inlineName** and the results can be displayed later using **resultSet**, **records**, or **rows** methods.

The entire database action can be specified directly in the opening **inline** method, or visitor-defined aspects of the action can be retrieved from query or post parameters. Nested **inline** methods can be used to create complex database actions.

The **-statementOnly** option instructs the data source to generate the implementation-specific statement required to perform the desired database action, but not to actually perform it. The generated statement can be returned with **action_statement**. This is useful for seeing the statement Lasso will generate for an action.

41.1.1 Database Actions

A *database action* is performed to retrieve data from a database or to manipulate data stored in a database. Database actions can be used in Lasso to query records in a database that match specific criteria, to return a particular record from a database, to add a record to a database, to delete a record from a database, to fetch information about a database, or to navigate through the found set from a database search. Additionally, database actions can be used to execute SQL statements in databases that understand SQL.

The database actions in Lasso are defined according to which action parameter is used to trigger the action. The following table lists the parameters that perform database actions that are available in Lasso.

Table 41.1: Database Action Parameters

Parameter	Description
-search	Finds records in a database that match specific criteria, returns detail for a particular record in a database, or navigates through a found set of records.
-findAll	Returns all records in a specific database table.
-random	Returns a single, random record from a database table.
-add	Adds a record to a database table.
-update	Updates a specific record in a database table.
-delete	Removes a specified record from a database table.
-show	Returns information about the tables and fields within a database.
-sql=?	Executes a SQL statement in a compatible data source. Only works with SQLite, MySQL, and other SQL databases.
-prepare=?	Creates a prepared SQL statement in a compatible data source. Nested inlines will execute the prepared statement with different values.
-nothing	The default action which performs no database actions, but simply passes the parameters of the action.

Note: The table *Database Action Parameters* lists all of the database actions that Lasso supports. Individual data source connectors may only support a subset of these parameters. For example, the Lasso Connector for FileMaker Server does not support the **-sql** action. See the documentation for third-party data source connectors for information about which actions they support.

Each database action parameter requires additional parameters in order to execute the action properly. These parameters are specified using additional keyword parameters. For example, a **-database** parameter specifies the database in which the action should take place and a **-table** parameter specifies the specific table from that database in which the action should take place. Keyword parameters specify the query for a **-search** action, the initial values for the new record created by an **-add** action, or the updated values for an **-update** action.

Full documentation of which inline parameters are required for each action are detailed in the section specific to that action in this chapter or in subsequent chapters.

Specifying a -FindAll Action Within an Inline

The following example shows an **inline** method that has a **-findAll** database action specified. The inline includes a **-findAll** parameter to specify the action, **-database** and **-table** parameters to specify the database and table from which records should be returned, and a **-keyField** parameter to specify the key field for the table. The entire database action is hard-coded within the **inline** method.

The method **found_count** returns how many records are in the database. The **records** method executes the code in the capture block for each record in the found set. The **field** methods are repeated for each found record, creating a listing of the names of all the people stored in the "people" table.

```
inline(
    -findAll,
    -database='contacts',
    -table='people',
    -keyField='id'
) => {^
    'There are ' + found_count + ' record(s) in the People table.\n'
    records => {^
        '<br />' + field('first_name') + ' ' + field('last_name') + '\n'
    ^}
^}

// =>
// There are 2 record(s) in the People table.
// <br />John Doe
// <br />Jane Doe
```

Specifying a -Search Action Within an Inline

The following example shows an **inline** method that has a **-search** database action . The inline includes a **-search** parameter to specify the action, **-database** and **-table** parameters to specify the database and table records from which records should be returned, and a **-keyField** parameter to specify the key field for the table. The subsequent keyword parameters, **'first_name'='John'** and **'last_name'='Doe'**, specify the query that will be performed in the database. Only records for John Doe will be returned. The entire database action is hard-coded within the inline.

The method **found_count** returns how many records for "John Doe" are in the database. The **records** method executes the code in the capture block for each record in the found set. The **field** methods are repeated for each found record, creating a listing of all the records for "John Doe" stored in the "people" table:

```
inline(
    -search,
    -database='contacts',
    -table='people',
    -keyField='id',
    'first_name'='John',
    'last_name'='Doe'
) => {^
    'There were ' + found_count + ' record(s) found in the People table.\n'
    records => {^
        '<br />' + field('first_name') + ' ' + field('last_name') + '\n'
    ^}
^}

// =>
// There were 1 record(s) found in the People table.
// <br />John Doe
```

Displaying the Generated Action Statement

Use the **action_statement** method within the **inline** method. This will return the action statement that was generated by the data source connector to fulfill the specified database action. For SQL data sources like MySQL and SQLite a SQL statement will be returned. Other data sources may return a different style of action statement.

```
inline(-search, -database='example', -table='example', /* etc. */) => {^
    action_statement
    // ...
^}
```

To see the action statement that would be generated by the data source without actually performing the database action the **-statementOnly** parameter can be specified in the **inline** method. The **action_statement** method will return the same value it would for a normal inline database action, but the database action will not actually be performed.

```
inline(-search, -database='example', -table='example', -statementOnly, /* etc. */) => {^
    action_statement
    // ...
^}
```

41.1.2 Inlines and HTML Forms

The previous two examples show how to specify a hard-coded database action completely within an **inline** method. This is an excellent way to embed a database action that will be the same every time a page is loaded, but does not provide any room for visitor interaction.

A more powerful technique is to use values from an HTML form or URL to allow a site visitor to modify the database action that is performed within the inline. The following two examples demonstrate two different techniques for doing this using the singular **web_request->param** method and the **tie**-based **web_request->params** method.

Using HTML Form Values Within an Inline

An inline-based database action can make use of visitor-specified parameters by reading values from an HTML form that the visitor customizes and submits to trigger the page containing the **inline** method.

The following HTML form provides two inputs into which the visitor can type information. An input is provided for "first_name" and one for "last_name". These correspond to the names of fields in the "people" table. The action of the form is set to "response.lasso" which will contain the inline that performs the actual database action:

```
<form action="response.lasso" method="POST">
    <br />First Name: <input type="text" name="first_name" value="" />
    <br />Last Name: <input type="text" name="last_name" value="" />
    <br /><input type="submit" name="submit" value="Search" />
</form>
```

The inline in "response.lasso" contains the **pair** parameter **'first_name'=web_request->param('first_name')**. The **web_request->param** method instructs Lasso to fetch the input named "first_name" from the form post parameters submitted to the current page being served, namely the form shown above. The inline contains a similar pair parameter for "last_name".

```
inline(
    -search,
    -database='contacts',
    -table='people',
    -keyField='id',
```

```
    'first_name'=web_request->param('first_name'),
    'last_name'=web_request->param('last_name')
) => {^
    'There were ' + found_count + ' record(s) found in the People table.\n'
    records => {^
        '<br />' + field('first_name') + ' ' + field('last_name') + '\n'
    ^}
^}
```

If the visitor entered "Jane" for the first name and "Doe" for the last name then the following results would be returned:

```
// =>
// There were 1 record(s) found in the People table.
// <br />Jane Doe
```

As many parameters as needed can be named in the HTML form and then retrieved in the response page via the inline.

Tip: The **web_request->param** member method is a replacement for the **action_param** or **form_param** methods used in prior versions of Lasso for fetching GET or POST data.

Using an Array of Form Values Within an Inline

Rather than specifying each **web_request->param** individually, an entire set of HTML form parameters can be entered into an **inline** method using the **web_request->params** method. Inserting the **web_request->params** method into an inline functions as if all the parameters and name/value pairs in the HTML form were placed into the inline at the location of the **web_request->params** parameter.

The **inline** method in our updated "response.lasso" contains the parameter **web_request->params**. This instructs Lasso to take all the parameters from the HTML form or URL which results in the current page being loaded and insert them in the inline as if they had been typed at the location of **web_request->params**. This will cause the name/value pairs for "first_name" and "last_name" entered in the form above to be inserted into the inline.

```
inline(
    web_request->params,
    -search,
    -database='contacts',
    -table='people',
    -keyField='id'
) => {^
    'There were ' + found_count + ' record(s) found in the People table.\n'
    records => {^
        '<br />' + field('first_name') + ' ' + field('last_name') + '\n'
    ^}
^}
```

If the visitor entered "Jane" for the first name and "Doe" for the last name then the following results would be returned:

```
// =>
// There were 1 record(s) found in the People table.
// <br />Jane Doe
```

As many parameters as needed can be named in the HTML form. They will all be passed into the inline at the location of the **web_request->params** method.

Tip: The **web_request->params** member method is a replacement for the **action_params** method used in prior versions of Lasso for fetching GET or POST data.

Setting HTML Form Values

If the Lasso page containing an HTML form is the action to an HTML form or the URL has query parameters, then the values of the HTML form inputs can be set to values passed from the previous Lasso page using **web_request->param**.

For example, if a form is on "default.lasso" and the action of the form is also "default.lasso" then the same page will be reloaded with the visitor-specified form values each time the form is submitted. The following HTML form uses **web_request->param** calls to automatically restore the values the user specified in the form previously each time the page is reloaded:

```
<form action="default.lasso" method="POST">
   First Name: <br />
   <input type="text" name="first_name" value="[web_request->param('first_name')]" />
   Last Name: <br />
   <input type="text" name="last_name" value="[web_request->param('last_name')]" />
   <br />
   <input type="submit" name="submit" value="Submit" />
</form>
```

41.1.3 Nesting Inline Database Actions

Database actions can be combined to perform compound database actions. All the records in a database that meet certain criteria could be updated or deleted. Or, all the records from one database could be added to a different database. Or, the results of searches from several databases could be merged and used to search another database.

Database actions are combined by nesting **inline** methods. For example, if inlines are placed inside a **records** method within another inline then the inner **inline** methods will execute once for each record found in the outer **inline** method.

All database result methods function for only the innermost **inline** method. Variables can pass through into nested inlines.

Tip: SQL nested inlines can also be used to perform reversible SQL transactions in transaction-compliant data sources. See the section *SQL Transactions* in the *SQL Data Sources* chapter for more information.

Updating Specific Records with Nested Inlines

This example will use nested **inline** methods to change the last name of all people whose last name is currently "Doe" in a database to "Person". The outer inline performs a hard-coded search for all records with "last_name" equal to "Doe". The inner inline updates each record so "last_name" is now equal to "Person". The output confirms that the conversion went as expected by outputting the new values.

```
inline(
   -search,
   -database='contacts',
   -table='people',
   -keyField='id',
   'last_name'='Doe',
   -maxRecords='all'
) => {^
   records => {^
```

```
    inline(
        -update,
        -database='contacts',
        -table='people',
        -keyField='id',
        -keyValue=keyField_value,
        'last_name'='Person'
    ) => {^
        '<br />Name is now ' + field('first_name') + ' ' + field('last_name') + '\n'
    ^}
  ^}
^}

// =>
// <br />Name is now John Person
// <br />Name is now Jane Person
```

41.1.4 Array-based Inline Parameters

Most parameters used within an **inline** method specify an action. Additionally, keyword parameters and name/value pair parameters can be stored in an array and then passed into an inline as a group. Any single value in an inline that is an array object will be interpreted as a series of parameters inserted at the location of the array. This technique is useful for programmatically assembling database actions.

Many parameters can only take a single value within an **inline** method. For example, only a single action can be specified and only a single database can be specified. The last parameter defines the value that will be used for the action. For example, the last **-database** parameter defines the value that will be used for the database of the action. If an array parameter comes first in an inline then all subsequent parameters will override any conflicting values within the array parameter.

Using an Array to Pass Values Into an Inline

The following Lasso code performs a **-findAll** database action with the parameters first specified in an array and stored in the variable "params", then passed into an **inline** method all at once. The value for **-maxRecords** in the inline overrides the value specified within the array parameter since it is specified later. Only the number of records found in the database are returned:

```
local(params) = (:
    -findAll,
    -database='contacts',
    -table='people',
    -maxRecords=50
)
inline(#params, -maxRecords=100) => {^
    'There are ' + found_count + ' record(s) in the People table.'
^}

// => There are 2 record(s) in the People table.
```

41.2 Inline Introspection Methods

Lasso has a set of methods that return information about the current inline's action. The parameters of the action itself can be returned or information about the action's results can be returned.

The following methods can be used within an **inline** method's capture block to return information about the action specified by the inline.

action_param(*name::string, join::string='rn'*)

action_param(*name::string, -count*)

action_param(*name::string, position::integer*)
> Requires a parameter specifying the name of a keyword or pair parameter passed to the **inline** method. If no other parameter is specified, then it returns all values it finds for the specified name joined together with a line break. An optional second parameter can specify what string to use as a separator when it finds more than one parameter with the specified name.
>
> To find the number of parameters passed to an **inline** method that share a specified name, you can specify **-count** as the second parameter. This will return the number of parameters sharing the same name. To get the value of a specific one of these parameters, instead pass an integer specifying which parameter you want. For example, if an inline is passed four parameters that share the same name, you can retrieve the one that comes third by passing a "3" as the second value to **action_param**.

action_params()
> Returns an array containing all of the keyword parameters and pair parameters that define the current action.

action_statement()
> Returns the statement that was generated for the data source to implement the requested action. For SQL databases this will return a SQL statement. Other data sources may return different values.

database_name()
> Returns the name of the current database.

keyField_name()

keyColumn_name()
> Returns the name of the current key field.

keyField_value()

keyColumn_value()
> Returns the name of the current key value if defined. Can also be used for actions that add a new record to get the newly generated ID.

lasso_currentAction()
> Returns the name of the current action.

maxRecords_value()
> Returns the number of records from the found set that are currently being displayed.

skipRecords_value()
> Returns the current offset into a found set.

table_name()

layout_name()
> Returns the name of the current table.

search_arguments()
> Executes a capture block once for each pair parameter in the current action.

search_fieldItem()
> Used in the capture block of a **search_arguments** method. This method returns the "name" portion of the current pair parameter.

search_valueItem()

Used in the capture block of a **search_arguments** method. This method returns the "value" portion of the current pair parameter.

search_operatorItem()

Used in the capture block of a **search_arguments** method. This method returns the operator associated with the current pair parameter.

sort_arguments()

Executes a capture block once for each sort parameter in the current action.

sort_fieldItem()

Used in the capture block of a **sort_arguments** method. This method returns the field that will be sorted.

sort_orderItem()

Used in the capture block of a **sort_arguments** method. This method returns the direction in which the field will be sorted.

41.2.1 Display Parameters of the Current Database Action

The value of the **action_params** method in the following example is formatted to clearly show the elements of the returned array:

```
inline(
    -search,
    -database='contacts',
    -table='people',
    -keyField='id'
) => {^
    action_params
^}

// =>
// staticarray(
//     (-search = true),
//     (-database = contacts),
//     (-table = people),
//     (-keyField = id)
// )
```

41.2.2 Display Parameter Pairs of the Current Database Action

Loop through the **action_params** method and display only name/value pairs for which the name does not start with a hyphen, i.e., any pair parameters and not keyword parameters. The following example shows a search of the "people" table of the "contacts" database for a person named "John Doe":

```
inline(
    -search,
    -database='contacts',
    -table='people',
    -keyField='id',
    'first_name'='John',
    'last_name'='Doe'
) => {^
    with param in action_params
    where not #param->first->beginsWith('-')
```

```
    sum '<br />' + #param->asString->encodeHtml + '\n'
^}

// =>
// <br />(first_name = John)
// <br />(last_name = Doe)
```

41.2.3 Display Action Parameters to a Site Visitor

The **search_arguments** method can be used in conjunction with the **search_fieldItem**, **search_valueItem** and **search_operatorItem** methods to return a list of all pair parameters and associated operators specified in a database action.

```
inline(
    -search,
    -database='contacts',
    -table='people',
    -keyField='id',
    'first_name'='John',
    'last_name'='Doe'
) => {^
    search_arguments => {^
        '<br />' + search_fieldItem + ' ' + search_operatorItem + ' ' + search_valueItem + '\n'
    ^}
^}

// =>
// <br />first_name BW John
// <br />last_name BW Doe
```

The **sort_arguments** method can be used in conjunction with the **sort_fieldItem** and **sort_orderItem** methods to return a list of all sort parameters specified in a database action.

```
inline(
    -search,
    -database='contacts',
    -table='people',
    -keyField='id',
    -sortField='first_name', -sortOrder='descending',
    -sortField='last_name'
) => {^
    sort_arguments => {^
        '<br />' + sort_fieldItem + ' ' + sort_orderItem + '\n'
    ^}
^}

// =>
// <br />first_name descending
// <br />last_name ascending
```

41.3 Inline Action Result Methods

The following documentation details the methods that allow information about the results of the current action to be returned. These methods provide information about the current found set rather than providing data about the database or

providing information about what database action was performed. Examples of using most of these methods are provided in the *Searching and Displaying Data* and *SQL Data Sources* chapters.

field(*name::string, …*)

column(*name::string, …*)
> Returns the value for a specified field from the result set. Can optionally take one of the following encoding keyword parameters: **-encodeNone**, **-encodeHtml**, **-encodeBreak**, **-encodeSmart**, **-encodeUrl**, **-encodeStrictUrl**, **-encodeXml**.

found_count() → integer
> Returns the number of records found by the database action.

records(*inlineName::string*)

records(*-inlineName::string=?*)

rows(*inlineName::string*)

rows(*-inlineName::string=?*)
> Loops once for each record in the found set. Any **field** methods within the **records** or **rows** methods will return the value for the specified field in each row in turn. Can be used outside of an inline capture block by specifying the name of a previously declared inline method with an **-inlineName** keyword parameter or just by passing in an inline name.

records_array()

rows_array()
> Returns the complete found set in a staticarray of staticarrays. The outer staticarray contains one staticarray for every row in the found set. The inner staticarrays contain one item for each field in the result set.

records_map(*…*)
> Returns the complete found set in a map of maps. See the table below for details about the parameters and output of **records_map**.
>
> Parameters
>> * **-keyField** – The name of the field to use as the key for the outer map. Defaults to the current **keyField_name**, "ID", or the first field of the database results.
>>
>> * **-returnField** – Specifies a field name that should be included in the inner map. Should be called multiple times to include multiple fields. If no **-returnField** is specified then all fields will be returned.
>>
>> * **-excludeField** – The name of a field to exclude from the inner map. If no **-excludeField** is specified then all fields will be returned.
>>
>> * **-fields** – An array of field names to use for the inner map. By default the value for **field_names** will be used.
>>
>> * **-type** – By default the method returns a map of maps. By specifying **-type='array'** the method will instead return an array of maps. This can be useful when the order of records is important.

resultSet_count(*-inlineName=?*)
> Returns the number of result sets that were generated by the inline. This will generally only be applicable to inlines that include a **-sql** parameter with multiple statements. An optional **-inlineName** parameter will return the number of result sets that a named inline has, outside of the inline's capture block.

resultSet(*-inlineName=?*)

resultSet(*num::integer, -inlineName=?*)

resultSet(*num::integer, inlineName::string*)
> Returns a single result set from an inline. The method can take an integer for its parameter to specify which result set to return. This defaults to the first set if it is not specified. An optional **-inlineName** keyword parameter or just passing in an inline name will return the indicated result set from a named inline.

---

shown_count()
> Returns the number of records shown in the current found set. Less than or equal to **maxRecords_value**.

shown_first()
> Returns the number of the first record shown from the found set. Usually **skipRecords_value** plus one.

shown_last()
> Returns the number of the last record shown from the found set.

The action result methods can be used to display information about the current found set. For example, the following code generates a status message that can be displayed under a database listing:

```
'Found ' + found_count + ' records.\n'
'<br />Displaying ' + shown_count + ' records from ' + shown_first + ' to ' + shown_last + '.'

// =>
// Found 100 records.
// Displaying 10 records from 61 to 70.
```

These methods can also be used to create links that allow a visitor to navigate through a found set.

41.3.1 Using a Records Array

The **records_array** method can be used to get access to all of the data from an inline operation. The method returns a staticarray with one element for each record/row in the found set. Each element is itself a staticarray that contains one element for each field/column in the found set.

The method can either be used to quickly output all of the data from the inline operation or can be used with the **iterate** methods or query expressions to access the data programmatically. (Of course, at that point, you probably just want to use the **records** or **rows** methods with the **field** or **column** methods.)

```
inline(-search, -database='contacts', -table='people') => {^
    records_array
^}

// => staticarray(staticarray(1, John, Doe), staticarray(1, Jane, Doe), ...)
```

The output can be made easier to read on a web page using the **iterate** method and the **array->join** method:

```
inline(-search, -database='contacts', -table='people') => {^
    iterate(records_array, local(record)) => {^
        '<br />' + ('"' + #record->join('", "') + '"')->encodeHtml + '\n'
    ^}
^}

// =>
// <br />&quot;1&quot;, &quot;John&quot;, &quot;Doe&quot;
// <br />&quot;2&quot;, &quot;Jane&quot;, &quot;Doe&quot;
// ...

// Web output
// =>
// "1", "John", "Doe"
// "2", "Jane", "Doe"
// ...
```

The output can be listed with the appropriate field names by using the **field_names** method. This method returns an array that contains each field name from the current found set. The **field_names** method will always contain the same number

of elements as the elements of the **records_array** method.

```
inline(-search, -database='contacts', -table='people') => {^
    '<table>\n'
    '<tr><td>' + field_names->join('</td><td>')->encodeHtml(false, true) + '</td></tr>\n'
    iterate(records_array, local(record)) => {^
        '<tr>\n'
        '    <td>' + #record->join('</td><td>')->encodeHtml(false, true) + '</td>\n'
        '</tr>\n'
    ^}
    '</table>\n'
^}

// =>
// <table>
// <tr><td>id</td><td>first_name</td><td>last_name</td></tr>
// <tr>
//     <td>1</td><td>John</td><td>Doe</td>
// </tr>
// <tr>
//     <td>2</td><td>Jane</td><td>Doe</td>
// </tr>
// ...
// </table>
```

Together the **field_names** and **records_array** methods provide a programmatic process of accessing all the data returned by an inline action. There may be some cases when these methods yield better performance than using **records**, **field**, and **field_name** methods.

41.3.2 Using a Records Map

The **records_map** method functions similarly to the **records_array** method, but returns all of the data from an inline operation as a map of maps. The keys for the outer map are the key field values for each record from the table. The keys for the inner map are the field names for each record in the found set.

```
inline(-search, -database='contacts', -table='people', -keyField='id') => {^
    records_map
^}

// => map(1 = map(first = John, last = Doe), 2 = map(first = Jane, last = Doe), ...)
```

41.4 Database Schema Inspection Methods

The schema of a database can be inspected using the **database_...** methods or the inline **-show** action parameter which allows information about a database to be returned using the **field_name** method. Value lists within FileMaker Server databases can also be accessed using the **-show** parameter. This is documented in the *FileMaker Data Sources* chapter.

The **-show** action parameter functions like the **-search** parameter except that no name/value pair parameters, sort parameters, result parameters, or operator parameters are required. The only other parameters required for a **-show** action are the **-database** and **-table** parameters. It is also recommended that you specify the **-keyField** parameter.

The methods detailed below allow the schema of a database to be inspected. The **field_name** method must be used in concert with a **-show** action or any database action that returns results including **-search**, **-add**, **-update**, **-random**, or **-findAll**. The **database_names** and **database_tableNames** methods can be used on their own.

database_names()
> Executes the capture block for every database specified in Lasso Admin. Requires using **database_nameItem** to show results.

database_nameItem()
> Used inside the capture block of a **database_names** method to return the name of the current database.

database_realName(*alias::string*)
> Returns the real name of a database given the alias that Lasso uses for the name.

database_tableNames(*dbname::string*)
> Executes the capture block for every table in the specified database. Requires using **database_tableNameItem** to show results.

database_tableNameItem()
> Used inside the capture block of a **database_tableNames** method to return the name of the current table.

field_name(*-count*)

field_name(*position::integer, -type=?*)

column_name(*-count*)

column_name(*position::integer, -type=?*)
> If passed the parameter **-count** then it returns the number of fields in the current table. If passed an integer, it returns the name of a field at that position in the current database and table. If passed an integer and then the **-type** parameter, it returns the type of field rather than the name. Types include "Text", "Number", "Date/Time", "Boolean", and "Unknown".

field_names()

column_names()
> Returns an array containing all the field names in the current result set. This is the same data as returned by **field_name**, but in a format more suitable for iterating or other data processing.

41.4.1 List All Databases Entered in Lasso Admin

The following example shows how to list the names of all databases set in Lasso Admin using the **database_names** and **database_nameItem** methods:

```
database_names => {^
    '<br />' + loop_count + ': ' + database_nameItem + '\n'
^}

// =>
// <br />1: Contacts
// <br />2: Examples
// <br />3: Site
```

41.4.2 List All Tables Within a Database

The following example shows how to list the names of all the tables within a database using the **database_tableNames** and **database_tableNameItem** methods. The tables within the "Site" database are listed:

```
database_tableNames('contacts') => {^
    '<br />' + loop_count + ': ' + database_tableNameItem + '\n'
^}

// =>
```

```
// <br />1: companies
// <br />2: people
```

41.4.3 List All Fields Within a Table

The following example demonstrates how to return information about the fields in a table using the **inline** method to perform a **-show** action. A **loop** method loops through the number of fields in the table and the name and type of each field is returned. The fields within the "contacts" table are shown:

```
inline(
    -show,
    -database='contacts',
    -table='people',
    -keyField='id'
) => {^
    loop(field_name(-count)) => {^
        '<br />' + loop_count + ': ' + field_name(loop_count) +
            ' (' + field_name(loop_count, -type) + ')\n'
    ^}
^}

// =>
// <br />1: creation_date (Date)
// <br />2: id (Number)
// <br />3: first_name (Text)
// <br />4: last_name (Text)
```

41.5 Inline Connection Options

Lasso provides two different ways to specify the data source that should execute an inline database action. The connection characteristics for the data source host can be specified entirely within the inline or the connection characteristics can be specified within Lasso Admin and then looked up based on which **-database** is specified within the inline.

Each of these options is described in more detail below including when one may be preferable to the other and the drawbacks of each. The database method is used throughout most of the examples in this documentation.

41.5.1 Database Name Method

If an inline contains only a **-database** parameter, then it is used to look up what host and data source should be used to service the inline. If there is a **-table** parameter, Lasso uses this to look up what encoding should be used for the results of the database action. If an inline does not have a specified **-database** then it inherits the **-database** (and **-table** and **-keyField**) from the surrounding inline.

Advantages
    When using the database method, all of the connection characteristics for the data source host are defined in Lasso Admin. This makes it easy to change the characteristics of a host, and even move databases from one host to another, without modifying any Lasso code.

Disadvantages
    Setting up a new data source when using the database method requires visiting Lasso Admin. This helps promote good security practices, but can be an impediment when working on simple web sites or when quickly mocking up solutions. Additionally, having part of the set up for a website in Lasso Admin means that Lasso must be configured

properly in order to deploy a solution. It is sometimes desirable to have all of the configuration of a solution contained within the code files of the solution itself.

41.5.2 Host Array Method

With the host array method, all of the characteristics of the data source host that will be used to process the inline database action are specified directly within the inline.

Advantages

Data source hosts can be quickly specified directly within an inline. No need to visit Lasso Admin to set up a new data source host. Additionally, there is reduced overhead since the connection information doesn't need to be retrieved from the SQLite database.

Disadvantages

The username and password for the host must be embedded within the Lasso code. (Although this can be in code that is not in the web root, thereby mitigating this disadvantage.) Also, switching data source hosts can be more difficult if inline hosts have been hard-coded.

Inline hosts are specified using a **-host** parameter within the inline. The value for the parameter is an array that specifies the connection characteristics for the database host. The following example shows an inline host for the MySQL data source that connects to "localhost" using a username of "lasso":

```
inline(
    -host=(:
        -datasource='mysqlds',
        -name='localhost',
        -username='lasso',
        -password='secret'
    ),
    -sql="SHOW DATABASES;"
) => {^
    records_array
^}

// => staticarray(staticarray(contacts), staticarray(examples), staticarray(site))
```

The following table lists all of the parameters that can be specified within the **-host** array. Some data sources may require that just the **-datasource** be specified, but most data sources will require **-datasource**, **-name**, **-username**, and **-password**.

The **-host** parameter can also take a value of "inherit" which specifies that the **-host** from the surrounding inline should be used. This is necessary when specifying a **-database** within nested inlines to prevent Lasso from looking up the database as it would using the database method.

Table 41.2: Host Array Parameters

Parameter	Description
-dataSource=?	Required data source name. The name for each data source can be found in the "Datasources" section of Lasso Server Admin.
-name=?	The IP address, DNS host name, or connection string for the data source. Required for most data sources.
-port=?	The port for the data source. Optional.
-username=?	The username for the data source connection. Required for most data sources.
-password=?	The password for the username. Required for most data sources.
-schema=?	The schema for the data source connection. Required for some data sources.
-tableEncoding=?	The table encoding for the data source connection. Defaults to "UTF-8". Optional.
-extra=?	Configuration information that may be used by some data sources. Optional.

Note: Consult the documentation for each data source for details about which parameters are required, their format, and whether the **-extra** parameter is used.

Once a **-host** array has been specified the rest of the parameters of the inline will work much the same as they do in inlines that use a configured data source host. The primary differences are explained here:

- Nested inlines will inherit the **-host** from the surrounding inline if they are specified with **-host='inherit'** or if they do not contain a **-database** parameter.

- Nested inlines that have a **-database** parameter and no **-host** parameter will use the **-database** parameter to look up the data source host.

- Nested inlines can specify a different **-host** parameter than the surrounding inline. Lasso can handle arbitrarily nested inlines each of which use a different host.

- The parameters **-database**, **-table**, **-keyField** (or **-key**), and **-schema** may be required depending on the database action. Inline actions such as **-search**, **-findAll**, **-add**, **-update**, **-delete**, etc. require that the database, table, and key field be specified just as they would need to be in any inline.

- Some SQL statements may also require that a **-database** be specified. For example, in MySQL, a host-level SQL statement like **SHOW DATABASES** doesn't require that a **-database** be specified. A table-level SQL statement like **SELECT * FROM 'people'** won't work unless the **-database** is specified in the inline. (A fully qualified SQL statement like **SELECT * FROM 'contacts'.'people'** will also work without a **-database**.)

Searching and Displaying Data

Lasso provides several parameters for the **inline** method for retrieving records within Lasso-compatible databases. These parameters are used in conjunction with name/value pair parameters in order to perform the desired database action in a specific database and table or within a specific record.

The **inline** action parameters documented in this chapter are listed below. The sections that follow describe the additional keyword and pair parameters required for each database action.

-search
Searches for records within a database.

-findAll
Finds all records within a database.

-random
Returns a random record from a database. (Only works with FileMaker Server databases.)

42.1 How Searches are Performed

The following describes each step that takes place every time a search is performed using Lasso:

1. Lasso checks the database, table, and field name specified in the search to verify that they are all valid.

2. The search query is formatted and sent to the database application. FileMaker Server search queries are formatted as URLs and submitted to the Web Publishing Engine. MySQL search queries are formatted as SQL statements and submitted directly to MySQL.

3. The database application performs the desired search and assembles a found set. The database application is responsible for interpreting search criteria, wild cards in search strings, field operators, and logical operators.

4. The database application sorts the found set based on sort criteria included in the search query. The database application is responsible for determining the order of records returned to Lasso.

5. A subset of the found set is sent to Lasso as the result set. Only the number of records specified by **-maxRecords** starting at the offset specified by **-skipRecords** is returned to Lasso. If any **-returnField** parameters are included in a search then only those fields they specify are returned to Lasso.

6. The result set can be displayed and manipulated using methods that return information about the result set and methods that return fields or other values.

42.2 Character Encoding

Lasso stores and retrieves data from data sources based on the preferences established in the "Datasources" section of Lasso Server Admin. The following rules apply for each standard data source:

Inline Host
> The character encoding can be specified explicitly using a `-tableEncoding` parameter within the `-host` array.

Inline Table
> The character encoding of the table specified using the `-table` parameter is used if `-tableEncoding` is not specified within the `-host` array.

MySQL
> By default all communication is encoded as UTF-8.

FileMaker Server
> By default all communication is in the MacRoman character set when Lasso Server is hosted on OS X, or in the Latin-1 (ISO-8859-1) character set when Lasso Server is hosted on Windows.

ODBC
> Encoding of communication with ODBC data sources is dependent on the encoding of the table being accessed.

42.3 Error Reporting

After a database action has been performed, Lasso reports any errors that occurred via the **error_currentError** method. The value of this method should be checked to verify that the database action was successfully performed.

42.3.1 Display Current Error Code and Message

The following code can be used to display the current error message. This code should be placed in a Lasso page that is a response to a database action or within the capture block of an **inline** method.

```
error_code + ': ' + error_msg
```

If the database action was performed successfully then the following result will be returned:

```
// => 0: No Error
```

42.3.2 Check for a Specific Error Code and Message

The following example shows how to report a specific error if one occurs using a conditional **if** statement to check if the current error message is equal to **error_databaseTimeout**:

```
if(error_currentError == error_databaseTimeout)
    'Connection to database lost!'
/if
```

Full documentation about error methods and error codes can be found in the *Error Handling* chapter.

42.4 Searching Records

Searches can be performed within any Lasso-compatible database using the `-search` parameter in an **inline** method. The `-search` parameter requires that a number of additional parameters be defined in order to perform the search. The additional required parameters are detailed in the table *-Search Action Requirements* along with a description of other recommended or optional parameters specific to the `-search` action.

Additional optional parameters are described in the tables *Search Operator Parameters* and *Result Parameters* in the sections that follow.

Table 42.1: -Search Action Requirements

Parameter	Description
`-search`	The action that is to be performed. Required.
`-database=?`	The database that should be searched. Required.
`-table=?`	The table from the specified database that should be searched. Required.
`-keyField=?`	The name of the field that holds the primary key for the specified table. Recommended.
`-keyValue=?`	The particular value for the primary key of the record that should be returned. Using `-keyValue` overrides all the other search parameters and returns the single record specified. Optional.
`-key=?`	An array that specifies the search field operators and pair parameters to find the matching records. Using `-key` overrides any other specified name/value pairs.
`-host=?`	Optional inline host array. See the section *Inline Connection Options* in the *Database Interaction Fundamentals* chapter for more information.
name/value pairs	A variable number of name/value pair parameters specify the query that will be performed. Any pair parameters included in the search action will be used to define the query that is performed in the specified table. All pair parameters must reference a field within the database. Any fields that are not referenced will be ignored for the purposes of the search.

42.4.1 Search a Database Using an Inline

The following example shows how to search a database by specifying the required parameters within an **inline** method. The `-database` is set to "contacts", `-table` is set to "people", and `-keyField` is set to "id". The search returns records that contain "John" with the field "first_name".

The results of the search are displayed to the visitor inside the inline. The **records** method will repeat for each record in the found set. The **field** methods will display the value for the specified field from the current record being shown.

```
inline(
    -search,
    -database='contacts',
    -table='people',
    -keyField='id',
    'first_name'='John'
) => {^
    records => {^
        '<br />' + field('first_name') + ' ' + field('last_name') + '\n'
    ^}
^}
```

If the search was successful then the following results will be returned:

```
// =>
// <br />John Person
// <br />John Doe
```

Additional pair parameters and keyword parameters can be used to generate more complex searches. These techniques are documented in the section *Search Operators* later in this chapter.

42.4.2 Search a Database Using Visitor-Supplied Values

The following example shows how to search a database by specifying the required parameters within an **inline** method, but allowing a site visitor to specify the search criteria in an HTML form. The visitor is presented with an HTML form in the Lasso page "default.lasso". The HTML form contains two text inputs for "first_name" and "last_name" and a submit button. The action of the form is the response page "response.lasso" which contains the inline that will perform the search. The contents of the "default.lasso" file include the following:

```
<form action="response.lasso" method="POST">
    <br />First Name: <input type="text" name="first_name" value="" />
    <br />Last Name: <input type="text" name="last_name" value="" />
    <br /><input type="submit" name="submit" value="Search" />
</form>
```

The search is performed and the results of the search are displayed to the visitor inside the **inline** method in "response.lasso". The values entered by the visitor in the HTML form in "default.lasso" are inserted into the inline using the **web_request->param** method. The **records** method will execute the capture block for each record in the found set. The **field** methods will display the value for the specified field from the current record being shown. The contents of the "response.lasso" file include the following:

```
inline(
    -search,
    -database='contacts',
    -table='people',
    -keyField='id',
    'first_name'=web_request->param('first_name'),
    'last_name'=web_request->param('last_name')
) => {^
    records => {^
        '<br />' + field('first_name') + ' ' + field('last_name') + '\n'
    ^}
^}
```

If the visitor entered "John" for "first_name" and "Person" for "last_name" then the following result would be returned:

```
// =>
// <br />John Person
```

42.5 Search Operators

Lasso inlines include a set of parameters that allow operators to be used to create complex database queries. These parameters are summarized in the table *Search Operator Parameters*.

Table 42.2: Search Operator Parameters

Parameter	Description
`-operatorLogical=?` or `-opLogical=?`	Specifies the logical operator for the search. Abbreviated as `-opLogical`. Defaults to AND.
`-operator=?` or `-op=?`	When specified before a pair parameter, sets the search operator for that parameter. Abbreviated as `-op`. Defaults to "bw". See below for a full list of field operators, which can also be written as `-bw`, `-ew`, `-cn`, etc.
`-operatorBegin=?` or `-opBegin=?`	Specifies the logical operator for all search parameters until `-operatorEnd` is reached. Abbreviated as `-opBegin`.
`-operatorEnd=?` or `-opEnd=?`	Specifies the end of a logical operator grouping started with `-operatorBegin`. Abbreviated as `-opEnd`.

The operator parameters are divided into two categories:

Field Operators
: These are specified using the `-operator` parameter before a name/value pair parameter. The field operator changes the way that the named field is searched for the value. If no field operator is specified then the default begins with operator ("bw") is used. See the table *Search Field Operators* for a list of the possible values. Field operators can also be abbreviated as `-bw`, `-ew`, `-cn`, etc.

Logical Operators
: These are specified using the `-operatorLogical`, `-operatorBegin`, and `-operatorEnd` parameters. These parameters specify how the results of different pair parameters are combined to form the full results of the search. You cannot mix `-operatorLogical` with `-operatorBegin` and `-operatorEnd`.

42.5.1 Field Operators

The possible values for the `-operator` parameter are listed in the table *Search Field Operators*. The default operator is begins with ("bw"). Case is not considered when specifying operators. Several of the field operators are only supported in MySQL or other SQL databases. These include the "ft" full-text operator and the "rx" and "nrx" regular expression operators, which are described further in the table *MySQL Additional Search Field Operators*.

Table 42.3: Search Field Operators

Operator	Description
`-op='bw'` or `-bw`	Begins With. Default if no operator is set.
`-op='nbw'` or `-nbw`	Not Begins With.
`-op='cn'` or `-cn`	Contains.
`-op='ncn'` or `-ncn`	Not Contains.
`-op='eq'` or `-eq`	Equals.
`-op='neq'` or `-neq`	Not Equals.
`-op='ew'` or `-ew`	Ends With.
`-op='new'` or `-new`	Not Ends With.
`-op='gt'` or `-gt`	Greater Than.
`-op='gte'` or `-gte`	Greater Than or Equals.
`-op='lt'` or `-lt`	Less Than.
`-op='lte'` or `-lte`	Less Than or Equals.
`-op='ft'` or `-ft`	Full-Text Search. MySQL databases only.
`-op='rx'` or `-rx`	Regular Expression Search. MySQL databases only.
`-op='nrx'` or `-nrx`	Not Regular Expression Search. MySQL databases only.

Field operators are interpreted differently depending on which data source is being accessed. For example, FileMaker Server interprets "bw" to mean that any word within a field can begin with the value specified for that field. MySQL interprets "bw" to mean that the first word within the field must begin with the value specified. See the chapters on each data source or the documentation that came with a third-party data source connector for more information.

Specify a Field Operator in an Inline

Specify the field operator before the name/value pair parameter that it will affect. The following **inline** method searches for records where the "first_name" begins with "J" and the "last_name" ends with "son":

```
inline(
    -search,
    -database='contacts',
    -table='people',
    -keyField='id',
    -operator='bw', 'first_name'='J',
    -operator='ew', 'last_name'='son'
) => {^
    records => {^
        '<br />' + field('first_name') + ' ' + field('last_name')
    ^}
^}
```

The same could be accomplished by using a **-key** parameter:

```
inline(
    -search,
    -database='contacts',
    -table='people',
    -keyField='id',
    -key=(: -bw, 'first_name'='J', -ew, 'last_name'='son')
) => {^
    records => {^
```

```
        '<br />' + field('first_name') + ' ' + field('last_name') + '\n'
    ^}
^}
```

The results of the search would include the following records:

```
// =>
// <br />John Person
// <br />Jane Person
```

42.5.2 Logical Operators

The logical operator parameter **-operatorLogical** can be used with a value of either "And" or "Or". The parameters **-operatorBegin** and **-operatorEnd** can be used with values of "And", "Or", or "Not". An **-operatorLogical** applies to all search parameters specified with an action while **-operatorBegin** applies to all search parameters until the matching **-operatorEnd** parameter is reached. (Thus the two cannot be mixed into the same inline.) The case of the value is unimportant when specifying a logical operator.

- **AND** – Specifies that records that are returned should fulfill all of the search parameters listed.

- **OR** – Specifies that records that are returned should fulfill one or more of the search parameters listed.

- **NOT** – Specifies that records that match the search criteria contained between the **-operatorBegin** and **-operatorEnd** parameters should be omitted from the found set. The NOT operator cannot be used with the **-operatorLogical** keyword parameter.

Tip: In lieu of a NOT option for **-operatorLogical**, many field operators can be negated individually by substituting the opposite field operator. The following pairs of field operators are the opposites of each other: "eq" and "neq", "lt" and "gte", and "gt" and "lte".

Note: The **-operatorBegin** and **-operatorEnd** parameters do not work with Lasso Connector for FileMaker Server.

Perform a Search Using an AND Operator

Use the **-operatorLogical** command tag with an "And" value. The following **inline** method returns records for which the "first_name" field begins with "John" and the "last_name" field begins with "Doe". The position of the **-operatorLogical** parameter within the inline is unimportant since it applies to the entire action.

```
inline(
    -search,
    -database='contacts',
    -table='people',
    -keyField='id',
    -operatorLogical='And',
    'first_name'='John',
    'last_name'='Doe'
) => {^
    records => {^
        '<br />' + field('first_name') + ' ' + field('last_name')
    ^}
^}
```

```
// => <br />John Doe
```

Perform a Search Using an OR Operator

Use the `-operatorLogical` parameter with an "Or" value. The following **inline** method returns records for which the "first_name" field begins with either "John" or "Jane". The position of the `-operatorLogical` parameter within the inline is unimportant since it applies to the entire action.

```
inline(
    -search,
    -database='contacts',
    -table='people',
    -keyField='id',
    -operatorLogical='Or',
    'first_name'='John',
    'first_name'='Jane'
) => {^
    records => {^
        '<br />' + field('first_name') + ' ' + field('last_name') + '\n'
    ^}
^}

// =>
// <br />John Doe
// <br />Jane Doe
// <br />John Person
```

Perform a Search Using a NOT Operator

Use the `-operatorBegin` and `-operatorEnd` parameters with a "Not" value. The following **inline** method returns records for which the "first_name" field begins with "John" and the "last_name" field is not "Doe". The operator parameters must surround the parameters of the search that is to be negated.

```
inline(
    -search,
    -database='contacts',
    -table='people',
    -keyField='id',
    'first_name'='John',
    -operatorBegin='Not',
        'last_name'='Doe',
    -operatorEnd='Not'
) => {^
    records => {^
        '<br />' + field('first_name') + ' ' + field('last_name')
    ^}
^}

// => <br />John Person
```

Perform a Search with a Complex Query

Use the `-operatorBegin` and `-operatorEnd` parameters to build up a complex query. As an example, a query can be constructed to find records in a database whose "first_name" and "last_name" both begin with the same letter "J" or "M". The desired query could be written in pseudocode as follows:

```
( (first_name begins with J) AND (last_name begins with J) )
OR
( (first_name begins with M) AND (last_name begins with M) )
```

To translate this into an inline statement, each line of the query becomes a pair of `-opBegin='And'` and `-opEnd='And'` parameters with a pair parameter for "first_name" and "last_name" contained inside. The two lines are then combined using a pair of `-opBegin='Or'` and `-opEnd='Or'` parameters. The nesting of the parameters works like the nesting of parentheses in the pseudocode above to clarify how Lasso should combine the results of different name/value pair parameters.

```
inline(
    -search,
    -database='contacts',
    -table='people',
    -keyField='id',
    -opBegin='Or',
        -opBegin='And',
            'first_name'='J',
            'last_name'='J',
        -opEnd='And',
        -opBegin='And',
            'first_name'='M',
            'last_name'='M',
        -opEnd='And',
    -opEnd='Or'
) => {^
    records => {^
        '<br />' + field('first_name') + ' ' + field('last_name') + '\n'
    ^}
^}
```

The returned result might look something like this:

```
// =>
// <br />Johnny Johnson
// <br />Jimmy James
// <br />Mark McPerson
```

42.6 Returning Records

Lasso inlines include a set of parameters that allow the results of a search to be customized. These parameters do not change the found set of records that are returned from the search, but they do change the data that is returned for formatting and display to the visitor. The result parameters are summarized in the table *Result Parameters*.

See also:

- SQL data source–specific methods and parameters in the *SQL Data Sources* chapter

- FileMaker Server–specific methods and parameters in the *FileMaker Data Sources* chapter

Table 42.4: Result Parameters

Parameter	Description
`-sortField=?` or `-sortColumn=?`	Specifies that the results should be sorted based on the data in the named field. Multiple `-sortField` parameters can be used for complex sorts. Optional, defaults to returning data in the order it appears in the database.
`-sortOrder=?`	When specified after a `-sortField` parameter, specifies the order of the sort, either "ascending", "descending" or custom. Optional, defaults to "ascending" for each `-sortField`.
`-maxRecords=?`	Specifies how many records should be shown from the found set. Optional, defaults to "50".
`-skipRecords=?`	Specifies an offset into the found set at which records should start being shown. Optional, defaults to "1".
`-returnField=?` or `-returnColumn=?`	Specifies a field that should be returned in the results of the search. Multiple `-returnField` parameters can be used to return multiple fields. Optional, defaults to returning all fields in the searched table.

The result parameters are divided into three categories:

1. **Sorting** is specified using the `-sortField` and `-sortOrder` parameters. These parameters change the order of the records that the search returns. The database application performs the sort before Lasso receives the record set.

2. The portion of the **Found Set** being shown is specified using the `-maxRecords` and `-skipRecords` parameters. `-maxRecords` sets the number of records that will be iterated over in the **records** method, while `-skipRecords` sets the offset into the found set that is shown. These two parameters define the window of records that are shown and can be used to navigate through a found set.

3. The **Fields** that are available are specified using the `-returnField` parameter. Normally, all fields in the searched table are returned. If any `-returnField` parameters are specified then only those fields will be available for display using the **field** method. Specifying `-returnField` parameters can improve the performance of Lasso by not sending unnecessary data between the database and the web server.

Note: In order to use the **keyField_value** method within an inline, the `-keyField` must be specified as one of the `-returnField` values.

42.6.1 Return Sorted Results

Specify `-sortField` and `-sortOrder` parameters within an inline search. The following inline includes sort parameters. The records are first sorted by "last_name" in ascending order, then sorted by "first_name" in ascending order:

```
inline(
    -search,
    -database='contacts',
    -table='people',
    -keyField='id',
    'first_name'='J',
    -sortField='last_name',  -sortOrder='ascending',
    -sortField='first_name', -sortOrder='ascending'
) => {^
    records => {^
        '<br />' + field('first_name') + ' ' + field('last_name') + '\n'
    ^}
^}
```

The following results could be returned when this inline is run. The returned records are sorted in order of "last_name". If the "last_name" of two records are equal then those records are sorted in order of "first_name".

```
// =>
// <br />Jane Doe
// <br />John Doe
// <br />Jane Person
// <br />John Person
```

42.6.2 Return a Portion of a Found Set

A portion of a found set can be returned by manipulating the values for `-maxRecords` and `-skipRecords`. In the following example, a search is performed for records where the "first_name" begins with "J". This search returns four records, but only the second two records are shown. `-maxRecords` is set to "2" to show only two records and `-skipRecords` is set to "2" to skip the first two records.

```
inline(
    -search,
    -database='contacts',
    -table='people',
    -keyField='id',
    'first_name'='J',
    -maxRecords=2,
    -skipRecords=2
) => {^
    records => {^
        '<br />' + field('first_name') + ' ' + field('last_name') + '\n'
    ^}
^}
```

The following results could be returned when this inline is run. Neither of the "Doe" records from the previous example are shown since they are skipped over.

```
// =>
// <br />Jane Person
// <br />John Person
```

42.6.3 Limit Fields Returned in Search Results

Use the `-returnField` parameter. If a single `-returnField` parameter is used then only the fields that are specified will be returned. If no `-returnField` parameters are specified then all fields within the current table will be returned. In the following example, only the "first_name" field is shown since it is the only field specified within a `-returnField` parameter:

```
inline(
    -search,
    -database='contacts',
    -table='people',
    -keyField='id',
    'first_name'='J',
    -returnField='first_name'
) => {^
    records => {^
        '<br />' + field('first_name') + '\n'
    ^}
^}
```

The "last_name" field cannot be shown for any of these records since it was not specified in a "-returnField" parameter. The above code would result in something like the following:

```
// =>
// <br />John
// <br />Jane
// <br />Jane
// <br />John
```

If the data source is MySQL, the **-distinct** parameter can be added to just return two records instead of four; one with the first name of "John" and the other with "Jane" See the *SQL Data Sources* chapter for details on the **-distinct** parameter.

42.7 Finding All Records

All records can be returned from a database using the **-findAll** parameter. The **-findAll** parameter functions exactly like the **-search** parameter except that no name/value pair parameters or operator parameters are required. Parameters that sort and limit the found set work the same as they do for **-search** actions.

Table 42.5: -FindAll Action Requirements

Parameter	Description
-findAll	The action that is to be performed. Required.
-database=?	The database that should be searched. Required.
-table=?	The table from the specified database that should be searched. Required.
-keyField=?	The name of the field that holds the primary key for the specified table. Recommended.
-host=?	Optional inline host array. See the section *Inline Connection Options* in the *Database Interaction Fundamentals* chapter for more information.

42.7.1 Return All Records from a Database

The following **inline** method finds all records within a table named "people" in the "contacts" database and displays them. The results are shown below:

```
inline(
    -findAll,
    -database='contacts',
    -table='people',
    -keyField='id'
) => {^
    records => {^
        '<br />' + field('first_name') + ' ' + field('last_name') + '\n'
    ^}
^}

// =>
// <br />John Doe
// <br />Jane Doe
// <br />John Person
// <br />Jane Person
```

42.8 Finding Random Records

A random record can be returned from a FileMaker database using the `-random` parameter. The `-random` parameter functions exactly like the `-search` parameter except that no name/value pair parameters or operator parameters are required.

Table 42.6: -Random Action Requirements

Parameter	Description
`-random`	The action that is to be performed. Required.
`-database=?`	The database that should be searched. Required.
`-table=?`	The table from the specified database that should be searched. Required.
`-keyField=?`	The name of the field that holds the primary key for the specified table. Recommended.
`-host=?`	Optional inline host array. See the section *Inline Connection Options* in the *Database Interaction Fundamentals* chapter for more information.

42.8.1 Return a Random Record from a Database

The following inline finds a single random record from a FileMaker Server database "contacts" and displays it. The `-maxRecords` is set to "1" to ensure that only a single record is shown. One potential result is shown below. Each time this inline is run a different record will be returned.

```
inline(
    -random,
    -database='contacts',
    -table='people',
    -keyField='id',
    -maxRecords=1
) => {^
    records => {^
        '<br />' + field('first_name') + ' ' + field('last_name')
    ^}
^}

// => <br />Jane Person
```

42.9 Displaying Data

The examples in this chapter have all relied on the **records** method and **field** method to display the results of the search that have been performed. This section describes the use of these methods in more detail. (See the section *Inline Action Result Methods* in the *Database Interaction Fundamentals* chapter for method documentation and more information.)

The **field** method always returns the value for a field from the current record when it is used within a capture block of a **records** method. If the **field** method is used outside of **records** block but inside an **inline** capture block, then it returns the value for the field from the first record in the found set. If the found set has only one record then the **records** method is optional.

Note: For clarity, the example code in these chapters display data exactly as returned from the database, but production code should use **encodeHtml**, **encodeXml**, or an encoding parameter with **field** calls to ensure characters are proplerly formatted

for the chosen output format.

42.9.1 Display Results of a Search

Use the **records** method and **field** method to display the results of a search. The following **inline** method performs a
-**findAll** action in a database "contacts". The results are returned each formatted on a line by itself. The **loop_count** method
is used to indicate the order within the found set.

```
inline(
    -findAll,
    -database='contacts',
    -table='people',
    -keyField='id'
) => {^
    records => {^
        '<br />' + loop_count + ': ' + field('first_name') + ' ' + field('last_name') + '\n'
    ^}
^}

// =>
// <br />1: John Doe
// <br />2: Jane Doe
// <br />3: John Person
// <br />4: Jane Person
```

42.9.2 Display Result for a Single Record

Use **field** methods within the capture block of an **inline** method. The **records** methods are unnecessary if only a single
record is returned. The following inline performs a -**search** for a single record whose primary key 'id' equals "1". The **key-
Field_value** is shown along with the **field** values for the record.

```
inline(
    -search,
    -database='contacts',
    -table='people',
    -keyField='id',
    -keyValue=1
) => {^
    '<br />' + keyField_value + ': ' + field('first_name') + ' ' + field('last_name') + '\n'
^}

// =>
// <br />1: Jane Doe
```

42.9.3 Display Results from a Named Inline

Use the -**inlineName** parameter in both the **inline** method and in the **records** method. The **records** method can be
located anywhere in the code after the inline that define the database action. The following example shows a -**findAll**
action at the top of a page of code with the results formatted later:

```
inline(
    -inlineName='FindAll Results',
```

```
    -findAll,
    -database='contacts',
    -table='people',
    -keyField='id'
) => {}

// ...

records(-inlineName='FindAll Results') => {^
    '<br />' + loop_count + ': ' + field('first_name') + ' ' + field('last_name') + '\n'
^}

// =>
// <br />1: John Doe
// <br />2: Jane Doe
// <br />3: John Person
// <br />4: Jane Person
```

Adding and Updating Records

Lasso provides action parameters for using the **inline** method for adding, updating, and deleting records within Lasso-compatible databases. These action parameters are used in conjunction with additional keyword and pair parameters in order to perform the desired database action in a specific database and table or within a specific record.

The **inline** action parameters documented in this chapter are listed below. The sections that follow describe the additional keyword and pair parameters required for each database action.

-add
Adds a record to a database.

-update
Updates a record or records within a database.

-delete
Removes a record or records from a database.

The same instructions for character encoding and error reporting from the *Searching and Displaying Data* chapter apply when writing to databases.

43.1 Adding Records

Records can be added to any Lasso-compatible database using the **-add** parameter. The **-add** parameter requires that a number of additional parameters be defined in order to perform the **-add** action. The required parameters are detailed in the following table.

Table 43.1: -Add Action Requirements

Parameter	Description
-add	The action that is to be performed. Required.
-database=?	The database where the record should be added. Required.
-table=?	The table from the specified database to which the record should be added. Required.
-keyField=?	The name of the field that holds the primary key for the specified table. Recommended.
-host=?	Optional inline host array. See the section *Inline Connection Options* in the *Database Interaction Fundamentals* chapter for more information.
name/value pairs	A variable number of name/value pair parameters specifying the field name and initial field values for the added record. Optional.

Any name/value pair parameters included in the **-add** action will be used to set the starting values for the record that is added to the database. All pair parameters must reference a writable field within the database. Any fields that are not referenced will be set to their default values according to the database's configuration.

Lasso returns a reference to the record that was added to the database. The reference is different depending on what type of database to which the record was added.

SQL Data Sources

> The **-keyField** parameter should be set to the primary key field or auto-increment field of the table. Lasso will return the added record as the result of the action by checking the specified key field for the last inserted record. The **keyField_value** method can be used to inspect the value of the auto-increment field for the inserted record.
>
> If no **-keyField** is specified, the specified **-keyField** is not an auto-increment field, or **-maxRecords** is set to "0" then no record will be returned as a result of the **-add** action. This can be useful in situations where a large record is being added to the database and there is no need to inspect the values that were added.

FileMaker Server

> The **keyField_value** method is set to the value of the internal Record ID for the new record. The Record ID functions as an auto-increment field that is automatically maintained by FileMaker Server for all records.
>
> FileMaker Server automatically performs a search for the record that was added to the database. The found set resulting from an **-add** action is equivalent to a search for the single record using the **keyField_value** method.
>
> The value for **-keyField** is ignored when adding records to a FileMaker Server database. The value for **keyField_value** is always the internal Record ID value.

Note: Consult the documentation for third-party data sources to see what behavior they implement when adding records to the database.

43.1.1 Add a Record Using an Inline

The following example shows how to perform an **-add** action by specifying the required parameters within an **inline** method. The **-database** is set to "contacts", **-table** is set to "people", and **-keyField** is set to "id". Feedback that the **-add** action was successful is provided to the visitor inside the inline using the **error_currentError** method. The added record will only include default values as defined within the database itself.

```
inline(
   -add,
   -database='contacts',
   -table='people',
   -keyField='id'
) => {^
   '<p>' + error_code + ': ' + error_msg + '</p>'
^}
```

If the **-add** action is successful then the following will be returned:

```
// => <p>0: No Error</p>
```

43.1.2 Add a Record with Data Using an Inline

The following example shows how to perform an **-add** action by specifying the required parameters within an **inline** method. Additionally, the inline includes a series of name/value pair parameters that define the values for various fields within the record that is to be added. The "first_name" field is set to "John" and the "last_name" field is set to "Doe". The added record will include these values as well as any default values defined in the database itself.

```
inline(
   -add,
   -database='contacts',
   -table='people',
   -keyField='id',
```

```
   'first_name'='John',
   'last_name'='Doe'
) => {^
   '<p>' + error_code + ': ' + error_msg + '</p>\n'
   'Record ' + field('id') + ' was added for ' + field('first_name') + ' ' + field('last_name') + '.'
^}
```

The results of the **-add** action contain the values for the record that was just added to the database:

```
// =>
// <p>0: No Error</p>
// Record 2 was added for John Doe.
```

43.1.3 Add a Record Using an HTML Form

The following example shows how to perform an **-add** action using an HTML form to send values into an **inline** method through **web_request->param**. The text inputs provide a way for the site visitor to define the initial values for various fields in the record that will be added to the database. The site visitor can set values for the fields "first_name" and "last_name".

```
<form action="response.lasso" method="POST">
   <br />First Name: <input type="text" name="first_name" value="" />
   <br />Last Name:  <input type="text" name="last_name" value="" />
   <br /><input type="submit" name="submit" value="Add" />
</form>
```

The response page for the form, "response.lasso", contains the following code that performs the action using an **inline** method and provides feedback that the record was successfully added to the database. The field values for the record that was just added to the database are automatically available within the inline.

```
inline(
   -add,
   -database='contacts',
   -table='people',
   -keyField='id',
   'first_name'=web_request->param('first_name'),
   'last_name'=web_request->param('last_name')
) => {^
   '<p>' + error_code + ': ' + error_msg + '</p>\n'
   'Record ' + field('id') + ' was added for ' + field('first_name') + ' ' + field('last_name') + '.'
^}
```

If the form is submitted with "Mary" in the "first_name" input and "Person" in the "last_name" input then the following will be returned:

```
// =>
// <p>0: No Error</p>
// Record 3 was added for Mary Person
```

43.1.4 Add a Record Using a URL

The following example shows how to perform an **-add** action using a URL to send values into an **inline** method through **web_request->param**. The name/value pair parameters in the URL define the initial values for various fields in the database: "first_name" is set to "John" and "last_name" is set to "Person".

```
<a href="response.lasso?first_name=John&last_name=Person">
   Add John Person
</a>
```

Using the same response page from the previous example, if the link for "Add John Person" is activated then the following will be returned:

```
// =>
// <p>0: No Error</p>
// Record 4 was added for John Person.
```

43.2 Updating Records

Records can be updated within any Lasso-compatible database using the `-update` parameter. The `-update` parameter requires that a number of additional parameters to be defined in order to perform the `-update` action. The required parameters are detailed in the following table.

Table 43.2: -Update Action Requirements

Parameter	Description
`-update`	The action that is to be performed. Required.
`-database=?`	The database where the record should be updated. Required.
`-table=?`	The table from the specified database which contains the record that should be updated. Required.
`-keyField=?`	The name of the field that holds the primary key for the specified table. Either a `-keyField` and `-keyValue` or a `-key` is required.
`-keyValue=?`	The value of the primary key of the record being updated.
`-key=?`	An array that specifies the search field operators and pair parameters to find the records to be updated. Either a `-keyField` and `-keyValue` or a `-key` is required. Using `-key` overrides any other specified name/value pairs.
`-host=?`	Optional inline host array. See the section *Inline Connection Options* in the *Database Interaction Fundamentals* chapter for more information.
name/value pairs	A variable number of name/value pair parameters specifying the field name and values that need to be updated. Optional.

Lasso has two methods for finding which records are to be updated.

`-keyField` and `-keyValue`

Lasso can identify the record to be updated using the values for the `-keyField` and `-keyValue` parameters. The `-keyField` must be set to the name of a field in the table. Typically, this is the primary key field for the table. The `-keyValue` must be set to a valid value for the `-keyField` in the table. If no record can be found with the specified `-keyValue` then nothing will be updated and an error will be returned.

The following inline would update the record with an "id" of "1" so it has a last name of "Doe":

```
inline(
   -update,
   -database='contacts',
   -table='people',
   -keyField='id',
   -keyValue=1,
   'last_name'='Doe'
) => {}
```

Note that if the specified key value returns multiple records then all of those records will be updated within the target table. If the `-keyField` is set to the primary key field of the table (or any field in the table that has a unique value for every record in the table) then the inline will only update one record.

-key

Lasso can identify the records that are to be updated using a search that is specified in an array. The search can use any of the fields in the current database table and any of the operators and logical operators which are described in the *Searching and Displaying Data* chapter.

The following inline would update all records in the "people" table that have a first name of "John" to have a last name of "Doe":

```
inline(
    -update,
    -database='contacts',
    -table='people',
    -key=(: -eq, 'first_name'='John'),
    'last_name'='Doe'
) => {}
```

Caution: Care should be taken when creating the search in a `-key` array. An update can very quickly modify all of the records in a database and there is no undo. Update inlines should be tested carefully before they are deployed on live data.

Any pair parameters included in the update action will be used to set the field values for the record being updated. All pair parameters must reference a writable field within the database. Any fields that are not referenced will maintain the values they had before the update.

Lasso returns a reference to the record that was updated within the database. The reference is different depending on what type of database is being used.

SQL Data Sources

The `keyField_value` method is set to the value of the key field that was used to identify the record to be updated. The `-keyField` should always be set to the primary key or auto-increment field of the table. The results when using other fields are undefined.

If the `-keyField` is not set to the primary key field or auto-increment field of the table or if `-maxRecords` is set to "0" then no record will be returned as a result of the `-update` action. This is useful if a large record is being updated and the results of the update do not need to be inspected.

FileMaker Server

The `keyField_value` method is set to the value of the internal Record ID for the updated record. The Record ID functions as an auto-increment field that is automatically maintained by FileMaker Server for all records.

Lasso automatically performs a search for the record that was updated within the database. The found set resulting from an `-update` action is equivalent to a search for the single record using the `keyField_value`.

Note: Consult the documentation for third-party data sources to see what behavior they implement when updating records within a database.

43.2.1 Update a Record with Data Using an Inline

The following example shows how to perform an `-update` action by specifying the required parameters within an `inline` method. The record with the value "2" in field "id" is updated. The inline includes a series of pair parameters that defines the

new values for various fields within the record that is to be updated. The "first_name" field is set to "Bob" and the "last_name" field is set to "Surname". The updated record will include these new values, but any fields that were not included in the action will be left with the values they had before the update.

```
inline(
    -update,
    -database='contacts',
    -table='people',
    -keyField='id',
    -keyValue=2,
    'first_name'='Bob',
    'last_name'='Surname'
) => {^
    '<p>' + error_code + ': ' + error_msg + '</p>\n'
    'Record ' + field('id') + ' was updated to ' +
        field('first_name') + ' ' + field('last_name') + '.'
^}
```

The updated field values from the **-update** action are automatically available within the inline:

```
// =>
// <p>0: No Error</p>
// Record 2 was updated to Bob Surname.
```

43.2.2 Update a Record Using an HTML Form

The following example shows how to perform an **-update** action using an HTML form to send values into an **inline** method. The text inputs provide a way for the site visitor to define the new values for various fields in the record that will be updated in the database. The site visitor can see and update the current values for the fields "first_name" and "last_name".

```
[inline(
    -search,
    -database='contacts',
    -table='people',
    -keyField='id',
    -keyValue=3
)]
<form action="response.lasso" method="POST">
    <input type="hidden" name="keyValue" value="[keyField_value]" />
    <br />First Name: <input type="text" name="first_name" value="[field('first_name')]" />
    <br />Last Name: <input type="text" name="last_name" value="[field('last_name')]" />
    <br /><input type="submit" name="submit" value="Update" />
</form>
[/inline]
```

The response page for the form, "response.lasso", contains the following code that performs the action using an **inline** method and provides feedback that the record was successfully updated in the database. The field values from the updated record are automatically available within the inline.

```
inline(
    -update,
    -database='contacts',
    -table='people',
    -keyField='id',
    -keyValue=web_request->param('keyValue'),
    'first_name'=web_request->param('first_name'),
    'last_name'=web_request->param('last_name')
```

```
) => {^
   '<p>' + error_code + ': ' + error_msg + '</p>\n'
   'Record ' + field('id') + ' was updated to ' +
         field('first_name') + ' ' + field('last_name') + '.'
^}
```

The form initially shows "Mary" for the "first_name" input and "Person" for the "last_name" input. If the form is submitted with the "last_name" changed to "Peoples" then the following will be returned. (The "first_name" field is unchanged since it was left set to "Mary".)

```
// =>
// <p>0: No Error</p>
// Record 3 was updated to Mary Peoples.
```

43.2.3 Update a Record Using a URL

The following example shows how to perform an **-update** action using a URL to send values into an **inline** method through **web_request->param**. The name/value pair parameters in the URL define the new values for various fields in the database: "first_name" is set to "John" and "last_name" is set to "Person".

```
<a href="response.lasso?keyValue=4&first_name=John&last_name=Person">
   Update John Person
</a>
```

Using the same response page from the previous example, if the link for "Update John Person" is activated then the following will be returned:

```
// =>
// <p>0: No Error</p>
// Record 4 was updated to John Person.
```

43.2.4 Update Several Records at Once

The following example shows how to perform an **-update** action on several records at once within a single database table. The goal is to update every record in the database with the last name of "Person" to the new last name of "Peoples".

There are two methods to accomplish this. The first method is to use the **-key** parameter to find the records that need to be updated within a single **-update** inline. The second method is to use an outer inline to find the records to be updated and then an inner inline that is repeated once for each record.

The **-key** method has the advantage of speed and is the best choice for simple updates. The nested inline method can be useful if additional processing is required on each record before it is updated within the data source.

Using -Key to Update Records

The inline uses a **-key** array that performs a search for all records in the database with a "last_name" equal to "Person". The update is performed automatically on this found set.

```
inline(
   -update,
   -database='contacts',
   -table='people',
   -key=(: -eq, 'last_name'='Person'),
```

```
    -maxRecords='all',
    'last_name'='Peoples'
) => {}
```

Using Nested Inlines to Update Records

The outer **inline** method performs a search for all records in the database with "last_name" equal to "Person". This forms the found set of records that need to be updated. The **records** method executes once for each record in the found set. The **-maxRecords='all'** parameter ensures that all records that match the criteria are returned.

The inner **inline** method performs an update on each record in the found set. Methods are used to retrieve the values for the required **-database**, **-table**, **-keyField**, and **-keyValue** parameters. This ensures that these values match those from the outer inline exactly. The pair parameter **'last_name'='Peoples'** updates the field to the new value.

```
inline(
    -search,
    -database='contacts',
    -table='people',
    -keyField='id',
    -maxRecords='all',
    'last_name'='Person'
) => {^
    records => {^
        inline(
            -update,
            -database=database_name,
            -table=table_name,
            -keyField=keyField_name,
            -keyValue=keyField_value,
            'last_name'='Peoples'
        ) => {^
            '<p>' + error_code + ': ' + error_msg + '</p>\n'
            'Record ' + field('id') + ' was updated to ' +
                field('first_name') + ' ' + field('last_name') + '.'
        ^}
    ^}
^}
```

This particular search only finds one record to update. If the update action is successful then the following will be returned for each updated record:

```
// =>
// <p>0: No Error</p>
// Record 4 was updated to John Peoples.
```

43.3 Deleting Records

Records can be deleted from any Lasso-compatible database using the **-delete** parameter. The **-delete** parameter requires that a number of additional parameters be defined in order to perform the **-delete** action. The required parameters are detailed in the following table.

Table 43.3: -Delete Action Requirements

Parameter	Description
-delete	The action that is to be performed. Required.
-database=?	The database where the record should be deleted. Required.
-table=?	The table from the specified database from which the record should be deleted. Required.
-keyField=?	The name of the field that holds the primary key for the specified table. Either a **-keyField** and **-keyValue** or a **-key** is required.
-keyValue=?	The value of the primary key of the record being deleted.
-key=?	An array that specifies the search field operators and pair parameters to find the records to be deleted. Either a **-keyField** and **-keyValue** or a **-key** is required.
-host=?	Optional inline host array. See the section *Inline Connection Options* in the *Database Interaction Fundamentals* chapter for more information.

Lasso has two methods to find which records are to be deleted.

-keyField and **-keyValue**

Lasso can identify the record to be deleted using the values for the **-keyField** and **-keyValue** parameters. The **-keyField** must be set to the name of a field in the table. Typically, this is the primary key field for the table. The **-keyValue** must be set to a valid value for the **-keyField** in the table. If no record can be found with the specified **-keyValue** then nothing will be deleted and no error will be returned.

The following inline would delete the record with an "id" of "1":

```
inline(
    -delete,
    -database='contacts',
    -table='people',
    -keyField='id',
    -keyValue=1
) => {}
```

Note that if the specified key value returns multiple records then all of those records will be deleted from the target table. If the **-keyField** is set to the primary key field of the table (or any field in the table that has a unique value for every record in the table) then the inline will only delete one record.

-key

Lasso can identify the records that are to be deleted using a search that is specified in an array. The search can use any of the fields in the current database table and any of the operators and logical operators which are described in the *Searching and Displaying Data* chapter.

The following inline would delete all records in the people database that have a first name of "John":

```
inline(
    -delete,
    -database='contacts',
    -table='people',
    -key=(: -eq, 'first_name'='John')
) => {}
```

Caution: Care should be taken when creating the search in a **-key** array. A delete can very quickly remove all of the records in a database and there is no undo. Delete inlines should be tested carefully before they are deployed on live data.

Lasso returns an empty found set in response to a **-delete** action. Since the record has been deleted from the database the **field** method can no longer be used to retrieve any values from it. The **error_currentError** method should be checked to verify that it has a value of "No Error" in order to confirm that the record has been successfully deleted.

There is no confirmation or undo of a delete action. When a record is removed from a database it is removed permanently. It is important to set up security appropriately so accidental or unauthorized deletes don't occur.

43.3.1 Delete a Record with Data Using an Inline

The following example shows how to perform a delete action by specifying the required parameters within an **inline** method. The record with the value "2" in field "id" is deleted:

```
inline(
    -delete,
    -database='contacts',
    -table='people',
    -keyField='id',
    -keyValue=2
) => {^
    '<p>' + error_code + ': ' + error_msg + '</p>'
^}
```

If the delete action is successful then the following will be returned:

```
// => <p>0: No Error</p>
```

43.3.2 Delete Several Records at Once

The following example shows how to perform a **-delete** action on several records at once within a single database table. The goal is to delete every record in the database with the last name of "Peoples".

> *Warning:* These techniques can be used to remove all records from a database table. They should be used with extreme caution and tested thoroughly before being added to a production website.

There are two methods to accomplish this. The first method is to use the **-key** parameter to find the records that need to be deleted within a single **-delete** inline. The second method is to use an outer inline to find the records to be deleted and then an inner inline that is repeated once for each record.

The **-key** method has the advantage of speed and is the best choice for simple deletes. The nested inline method can be useful if additional processing is required to decide if each record should be deleted.

Using -Key to Delete Records

This inline uses a **-key** array that performs a search for all records in the database with a "last_name" equal to "Peoples". The records in this found set are automatically deleted.

```
inline(
    -delete,
    -database='contacts',
    -table='people',
    -key=(: -eq, 'last_name'='Peoples')
) => {}
```

Using Nested Inlines to Delete Records

The outer **inline** method performs a search for all records in the database with "last_name" equal to "Peoples". This forms the found set of records that need to be deleted. The **records** method executes once for each record in the found set. The `-maxRecords='all'` parameter ensures that all records that match the criteria are returned.

The inner **inline** method deletes each record in the found set. Methods are used to retrieve the values for the required `-database`, `-table`, `-keyField`, and `-keyValue` parameters. This ensures that these values match those from the outer inline exactly.

```
inline(
    -search,
    -database='contacts',
    -table='people',
    -keyField='id',
    -maxRecords='all',
    'last_name'='Peoples'
) => {^
    records => {^
        inline(
            -delete,
            -database=database_name,
            -table=table_name,
            -keyField=keyField_name,
            -keyValue=keyField_value
        ) => {^
            '<p>' + error_code + ': ' + error_msg + '</p>'
        ^}
    ^}
^}
```

This particular search only finds one record to delete. If the delete action is successful then the following will be returned for each deleted record:

```
// => <p>0: No Error</p>
```

SQL Data Sources

This chapter documents methods and behaviors that are specific to the SQL data sources in Lasso. These include the data source connectors for MySQL, SQLite, Oracle, PostgreSQL, and SQL Server. Most of the features of Lasso work equally across all data sources. The differences specific to each SQL data source are noted in the features list and in the descriptions of individual features.

MySQL
    Supports MySQL 3.x, 4.x, or 5.x data sources.

Oracle
    Supports Oracle data sources. The Oracle "Instant Client" libraries must be installed in order to activate this data source.

PostgreSQL
    Supports PostgreSQL data sources. The PostgreSQL client libraries must be installed in order to activate this data source.

SQL Server
    Supports Microsoft SQL Server. The SQL Server client libraries must be installed in order to activate this data source.

SQLite
    Supports SQLite 3 data sources. SQLite is the internal data source that is used for the storage of Lasso's preferences and security settings.

44.1 Supported Features for SQL Data Sources

The following lists detail the features of each data source in this chapter. Since some features are only available in certain data sources it is important to check these tables when reading the documentation in order to verify that each data source supports your solution's required features.

44.1.1 MySQL Data Source

Friendly Name
    Lasso Connector for MySQL

Internal Name
    mysqlds

Module Name
    MySQLConnector.dll, MySQLConnector.dylib, or MySQLConnector.so

Inline Host Attributes
    Requires `-name` specifying connection URL (e.g. "mysql.example.com"), `-username`, and `-password`. Optional `-port` defaults to "3306".

Actions
    `-add`, `-delete`, `-exec`, `-findAll`, `-prepare`, `-search`, `-show`, `-sql`, `-update`

Operators

    `-bw`, `-cn`, `-eq`, `-ew`, `-gt`, `-gte`, `-lt`, `-lte`, `-nbw`, `-ncn`, `-new`, `-ft`, `-rx`, `-nrx`, `-opBegin`/`-opEnd` with "And", "Or", "Not".

KeyField

    `-keyField`/`-keyValue` and `-key=array`

44.1.2 Oracle Data Source

Friendly Name

    Lasso Connector for Oracle

Internal Name

    oracle

Module Name

    SQLConnector.dll, SQLConnector.dylib, or SQLConnector.so

Inline Host Attributes

    Requires `-name` specifying connection URL (e.g. "oracle.example.com:1521/mydatabase"), `-username`, and `-password`.

Actions

    `-add`, `-delete`, `-findAll`, `-search`, `-show`, `-sql`, `-update`

Operators

    `-bw`, `-cn`, `-eq`, `-ew`, `-gt`, `-gte`, `-lt`, `-lte`, `-nbw`, `-ncn`, `-new`, `-opBegin`/`-opEnd` with "And", "Or", "Not".

KeyField

    `-keyField`/`-keyValue`

Note: Field names are case-sensitive. All field names and key field names within the inline must be specified with the proper case.

44.1.3 PostgreSQL Data Source

Friendly Name

    Lasso Connector for PostgreSQL

Internal Name

    postgresql

Module Name

    SQLConnector.dll, SQLConnector.dylib, or SQLConnector.so

Inline Host Attributes

    Requires `-name` specifying connection URL (e.g. "postgresql.example.com"), `-username`, and `-password`.

Actions

    `-add`, `-delete`, `-findAll`, `-search`, `-show`, `-sql`, `-update`

Operators

    `-bw`, `-cn`, `-eq`, `-ew`, `-gt`, `-gte`, `-lt`, `-lte`, `-nbw`, `-ncn`, `-new`, `-opBegin`/`-opEnd` with "And", "Or", "Not".

KeyField

    `-keyField`/`-keyValue`

Note: Field names are case-sensitive. All field names and key field names within the inline must be specified with the proper case.

44.1.4 SQL Server Data Source

Friendly Name
Lasso Connector for SQL Server

Internal Name
sqlserver

Module Name
SQLConnector.dll, SQLConnector.dylib, or SQLConnector.so

Inline Host Attributes
Requires `-name` specifying connection URL (e.g. "sqlserver.example.com\mydatabase"), `-username`, and `-password`.

Actions
`-add`, `-delete`, `-findAll`, `-search`, `-show`, `-sql`, `-update`

Operators
`-bw`, `-cn`, `-eq`, `-ew`, `-gt`, `-gte`, `-lt`, `-lte`, `-nbw`, `-ncn`, `-new`, `-opBegin`/`-opEnd` with "And", "Or", "Not".

KeyField
`-keyField`/`-keyValue`

44.1.5 SQLite Data Source

Friendly Name
Lasso Internal

Internal Name
sqliteconnector

Module Name
SQLiteConnector.dll, SQLiteConnector.dylib, or SQLiteConnector.so

Actions
`-add`, `-delete`, `-findAll`, `-search`, `-show`, `-sql`, `-update`

Operators
`-bw`, `-cn`, `-eq`, `-ew`, `-gt`, `-gte`, `-lt`, `-lte`, `-nbw`, `-ncn`, `-new`, `-opBegin`/`-opEnd` with "And", "Or", "Not".

KeyField
`-keyField`/`-keyValue`

44.2 SQL Data Source Tips

- Always specify a primary key field using the `-keyField` parameter for `-search`, `-add`, and `-findAll` actions. This will ensure that the **keyField_value** method will always return a value.

- Use `-keyField` and `-keyValue` parameters to reference a particular record for updates or deletes.

- Data sources can be case-sensitive. For best results, reference database and table names in the same letter case as they appear on disk in your Lasso code. Field names may also be case-sensitive (such as in Oracle and PostgreSQL).

- Some data sources will truncate any data beyond the length they are set up to store. Verify that all fields have sufficient capacity for the values that need to be stored in them.

- Use `-returnField` parameters to reduce the number of fields that are returned from a `-search` action. Returning only the fields that need to be used for further processing or shown to the site visitor reduces the amount of data that needs to travel between Lasso and the data source.

- When an `-add` or `-update` action is performed on a database, the data from the added or updated record is available inside the capture block of the `inline` method. If the `-returnField` parameter is used, then only those fields specified should be returned from an `-add` or `-update` action. Setting `-maxRecords=0` can be used as an indication that no record should be returned.

44.3 Security Tips

- SQL statements that are generated using visitor-defined data should be screened carefully for unwanted commands such as "DROP" or "GRANT".

- Always sanitize any inputs from site visitors that are incorporated into SQL statements. Any SQL strings that have visitor-defined data should be sanitized using the **string->encodeSql** method for MySQL data sources and the **string->encodeSql92** method for SQL92-compliant data sources such as SQLite, PostgreSQL, or ODBC data sources. Encoding the values in this manner ensures that quotes and other reserved characters are properly escaped within the SQL statement, thereby helping to prevent SQL injection attacks.

  For example, the following SQL "SELECT" statement contains a SQL string in the LIKE clause and uses **string->encodeSql** to encode the value of the "company" **web_request->param**. This encoding causes all single quotes within the passed **company** parameter to be encoded with a backslash.

  ```
  local(sql_statement) = "SELECT * FROM contacts.people WHERE company LIKE '" +
          string(web_request->param('company'))->encodeSql + "%';"
  ```

  If **web_request->param('company')** returns "McDonald's" then the SQL statement generated by this code would appear as follows:

  ```
  SELECT * FROM Contacts.People WHERE Company LIKE "McDonald's%";
  ```

44.4 SQL Data Source Methods

Lasso includes methods to identify which type of SQL data source is being used. These methods are summarized below.

lasso_datasourceIsMySQL(*name***)**
    Returns "true" if the specified database is hosted by MySQL. Requires one string parameter, which is the name of a database.

lasso_datasourceIsSybase(*name***)**
    Returns "true" if the specified database is hosted by Sybase. Requires one string parameter, which is the name of a database.

lasso_datasourceIsOracle(*name***)**
    Returns "true" if the specified database is hosted by Oracle. Requires one string parameter, which is the name of a database.

lasso_datasourceIsPostgreSQL(*name***)**
    Returns "true" if the specified database is hosted by PostgreSQL. Requires one string parameter, which is the name of a database.

lasso_datasourceIsSQLServer(*name*)

>   Returns "true" if the specified database is hosted by Microsoft SQL Server. Requires one string parameter, which is the name of a database.

lasso_datasourceIsSQLite(*name*)

>   Returns "true" if the specified database is hosted by SQLite. Requires one string parameter, which is the name of a database.

44.4.1 Check Whether a Database is Hosted by MySQL

The following example shows how to use **lasso_datasourceIsMySQL** to check whether the database "example" is hosted by MySQL or not:

```
if(lasso_datasourceIsMySQL('example'))
    stdoutnl("Example is hosted by MySQL!")
else
    stdoutnl("Example is not hosted by MySQL.")
/if

// => Example is hosted by MySQL!
```

44.4.2 List All Databases Hosted by MySQL

Use the **database_names** method to list all databases available to Lasso. The **lasso_datasourceIsMySQL** method can be used to check each database and only list those which are MySQL hosts. The result shows two databases, "site" and "example", which are available through MySQL:

```
database_names
    if(lasso_datasourceIsMySQL(database_nameItem))
        '<br />' + database_nameItem + '\n'
    /if
/database_names

// =>
// <br />example
// <br />site
```

44.5 Searching Records with SQL Data Sources

In Lasso, there are unique search operations that can be performed using SQL data sources. These search operations take advantage of special functions such as full-text indexing, regular expressions, record limits, and distinct values to allow optimal performance and power when searching. All these search operations can be used on MySQL data sources in addition to all search operations described in the *Searching and Displaying Data* chapter.

44.5.1 Search Field Operators for MySQL

Additional field operators are available for the **-operator** (or **-op**) parameter when searching MySQL data sources. These operators are summarized in the table below. Basic use of the **-operator** parameter is described in the *Searching and Displaying Data* chapter. See the MySQL documentation[59] for more information on full-text searches and the regular expressions supported in MySQL.

---

[59] http://dev.mysql.com/doc/

Table 44.1: MySQL Additional Search Field Operators

Operator	Description
`-op='ft'` or `-ft`	Full-Text Search. If used, a MySQL full-text search is performed on the field specified. Will only work on fields that are full-text indexed in MySQL. Records are automatically returned in order of high relevance (contains many instances of that value) to low relevance (contains few instances of the value). Only one `-ft` operator may be used per action, and no `-sortField` parameter should be specified.
`-op='nrx'` or `-rx`	Regular Expression Search. If used, then regular expressions may be used as part of the search field value. Returns records matching the regular expression value for that field.
`-op='nrx'` or `-nrx`	Not Regular Expression Search. If used, then regular expressions may be used as part of the search field value. Returns records that do not match the regular expression value for that field.

Perform a Full-Text Search on a Field

If a MySQL field is indexed as full-text, then using `-op='ft'` before the field in a search inline performs a MySQL full-text search on that field. The example below performs a full-text search on the "jobs" field in the "people" table, and returns the "first_name" field for each record that contain the word "Manager". Records that contain the most instances of the word "Manager" are returned first.

```
inline(
    -search,
    -database='contacts',
    -table='people',
    -op='ft', 'jobs'='Manager'
) => {^
    records => {^
        '<br />' + field('first_name') + '\n'
    ^}
^}

// =>
// <br />Mike
// <br />Jane
```

Use Regular Expressions as Part of a Search

Regular expressions can be used as part of a search value for a field by using `-op='rx'` before the field in a search inline. The following example searches for all records where the "last_name" field contains eight characters using a regular expression:

```
inline(
    -search,
    -database='contacts',
    -table='people',
    -op='rx', 'last_name'='.{8}',
    -maxRecords='all'
) => {^
    records => {^
        '<br />' + field('last_name') + ', ' + field('first_name')
    ^}
^}

// =>
```

```
// <br />Lastname, Mike
// <br />Lastname, Mary Beth
```

The following example searches for all records where the "last_name" field doesn't contain eight characters. This is easily accomplished using the same inline search above using `-op='nrx'` instead.

```
inline(
    -search,
    -database='contacts',
    -table='people',
    -op='nrx', 'last_name'='.{8}',
    -maxRecords='all'
) => {^
    records => {^
        '<br />' + field('last_name') + ', ' + field('first_name') + '\n'
    ^}
^}

// =>
// <br />Doe, John
// <br />Doe, Jane
// <br />Surname, Bob
// <br />Surname, Jane
// <br />Surname, Margaret
// <br />Unknown, Thomas
```

44.5.2 Result Keyword Parameters

Additional result keyword parameters are available when searching the data sources in this chapter using the **inline** method. These parameters are summarized in the following table.

Table 44.2: SQL Additional Result Parameters

Parameter	Description
-distinct	Causes a `-search` action to only output records that contain unique field values, comparing only across returned fields, or a `findAll` action to return records that are distinct across all fields. Does not require a value. May be used with the `-returnField` parameter to limit which fields are checked for distinct values. MySQL only.
-groupBy=?	Specifies a field name that should by used as the "GROUP BY" statement for a search action. Allows data to be summarized based on the values of the specified field.
-sortRandom	Requests that returned records be sorted randomly. Is used in place of the `-sortField` and `-sortOrder` parameters. Does not require a value. MySQL only.
-useLimit	Prematurely ends a `-search` or `-findAll` action once the specified number of records for the `-maxRecords` parameter have been found and returns the found records. Requires the `-maxRecords` parameter. This issues a "LIMIT" or "TOP" statement.

Return Only Unique Records in a Search

Use the `-distinct` parameter in a search inline. In the following example, a `-findAll` action is used on the "people" table of the "contacts" database. Only distinct values from the "last_name" field are returned.

```
inline(
    -findAll,
    -database='contacts',
    -table='people',
    -returnField='last_name',
    -distinct
) => {^
    records => {^
        '<br />' + field('last_name') + '\n'
    ^}
^}

// =>
// <br />Doe
// <br />Surname
// <br />Lastname
// <br />Unknown
```

The -distinct parameter is especially useful for generating lists of values that can be used in a drop-down list. The following example is a drop-down list of all the last names in the "people" table:

```
inline(
    -findAll,
    -database='contacts',
    -table='people',
    -returnField='last_name',
    -distinct
) => {^
    '<select name="last_name">\n'
    records => {^
        '    <option value="' + field('last_name') + '">' + field('last_name') + '</option>\n'
    ^}
    '</select>\n'
^}

// =>
// <select name="last_name">
//     <option value="Doe">Doe</option>
//     <option value="Surname">Surname</option>
//     <option value="Lastname">Lastname</option>
//     <option value="Unknown">Unknown</option>
// </select>
```

Use the -groupBy parameter to specify a field whose values should be distinct without limiting which fields are returned. The following query will return the same result as above, but have all fields available for display:

```
inline(
    -search,
    -database='contacts',
    -table='people',
    -groupBy='last_name'
) => {^
    '<select name="last_name">\n'
    records => {^
        '    <option value="' + field('last_name') + '">' + field('last_name') + '</option>\n'
    ^}
    '</select>\n'
^}
```

Sort Results Randomly

Use the `-sortRandom` parameter in a search inline. In the following example, all records from the "people" table of the "contacts" database are returned in random order:

```
inline(
   -findAll,
   -database='contacts',
   -table='people',
   -keyField='id',
   -sortRandom
) => {^
   records => {^
      field('id')
   ^}
^}

// => 5 2 8 1 3 6 4 7
```

Note: Due to the nature of the `-sortRandom` parameter, the results of this example will vary upon each execution of the inline.

Return Records Once a Limit is Reached

Use the `-useLimit` parameter in the search inline. Normally, Lasso will find all records that match the inline search criteria and then pare down the results based on `-maxRecords` and `-skipRecords` values. The `-useLimit` parameter instructs the data source to terminate the specified search process once the number of records specified for `-maxRecords` is found. The following example searches the "people" table with a limit of five records:

```
inline(
   -findAll,
   -database='contacts',
   -table='people',
   -maxRecords='5',
   -useLimit
) => {^
   found_count
^}

// => 5
```

Note: If the `-useLimit` parameter is used, the value of the **found_count** method will always be the same as the `-maxRecords` value if the limit is reached. Otherwise, the **found_count** method will return the total number of records in the specified table that match the search criteria if `-useLimit` is not used.

44.5.3 Searching for Null Values

When searching tables in a SQL data source, "NULL" values may be explicitly searched for within fields using the **null** object. A "NULL" value in a SQL data source designates that there is no value stored in that particular field. This is similar to searching a field for an empty string (e.g. `'fieldname'=''`), however "NULL" values and empty strings are not the same in SQL data sources. For more information about how "NULL" values are handled, see the documentation for each data source.

```
inline(
    -search,
    -database='contacts',
    -table='people',
    -op='eq', 'title'=null,
    -maxRecords='all'
) => {^
    records => {^
        '<br />Record ' + field('id') + ' does not have a title.\n'
    ^}
^}

// =>
// <br />Record 7 does not have a title.
// <br />Record 8 does not have a title.
```

44.6 Adding and Updating Records

In Lasso, there are special add and update operations that can be performed using SQL data sources in addition to all the add and update operations described in the *Adding and Updating Records* chapter.

44.6.1 Multiple Field Values

When adding or updating data to a field in MySQL, the same field name can be used several times in an **-add** or **-update** inline. The result is that all data added or updated in each instance of the field name will be concatenated in a comma-delimited form. This is particularly useful for adding data to "SET" field types.

Add or Update Multiple Field Values

The following example adds a record with two comma-delimited values in the "Jobs" field:

```
inline(
    -add,
    -database='contacts',
    -table='people',
    -keyField='id',
    'jobs'='Customer Service',
    'jobs'='Sales'
) => {^
    field('jobs')
^}

// => Customer Service,Sales
```

The following example updates the "jobs" field of a record with three comma-delimited values:

```
inline(
    -update,
    -database='contacts',
    -table='people',
    -keyField='id',
    -keyValue=5,
    'jobs'='Customer Service',
```

```
    'jobs'='Sales',
    'jobs'='Support'
) => {^
    field('jobs')
^}

// => Customer Service,Sales,Support
```

Note: The individual values being added or updated should not contain commas.

44.6.2 Null Values

"NULL" values can be explicitly added to fields using the **null** object. A "NULL" value in a SQL data source designates that there is no value stored in that particular field. This is similar to setting a field to an empty string (e.g. **'fieldname'=''**), however the two are different in SQL data sources. For more information about how "NULL" values are handled, see the documentation for each data source.

Add or Update a Null Field Value

Use the **null** object as the field value. The following example adds a record with a "NULL" value in the "last_name" field:

```
inline(
    -add,
    -database='contacts',
    -table='people',
    -keyField='id',
    'last_name'=null
) => {}
```

The following example updates a record with a "NULL" value in the "last_name" field:

```
inline(
    -update,
    -database='contacts',
    -table='people',
    -keyField='id',
    -keyValue=5,
    'last_name'=null
) => {}
```

44.7 Value Lists for ENUM or SET Fields

A *value list* in Lasso is a set of possible values that can be used for a field. Value lists in MySQL are lists of predefined and stored values for a "SET" or "ENUM" field type. A value list from a "SET" or "ENUM" field can be displayed using the methods defined below. None of these methods will work in **-sql** inlines or if **-noValueLists** is specified.

value_list(*name::string*)
> Executes a capture block once for each value allowed for an "ENUM" or "SET" field. Requires a single parameter: the name of an "ENUM" or "SET" field from the current table. This method will not work in **-sql** inlines or if the **-noValueLists** parameter is specified.

---

value_listItem()

    While in a **value_list** capture block, it returns the value for the current item.

selected()

    Displays the word "selected" if the current value list item is contained in the data of the "ENUM" or "SET" field.

checked()

    Displays the word "checked" if the current value list item is contained in the data of the "ENUM" or "SET" field.

Tip: See the section *Database Schema Inspection Methods* for information about the `-show` parameter which is used throughout these examples.

44.7.1 Display Allowed Values for an ENUM or SET Field

Perform a `-show` action to return the schema of a MySQL database and use the **value_list** method to display the allowed values for an "ENUM" or "SET" field. The following example shows how to display all values from the "ENUM" field "title" in the "people" table. "SET" fields function in the same manner as "ENUM" fields, and all examples in this section may be used with either "ENUM" or "SET" field types.

```
inline(
    -show,
    -database='contacts',
    -table='people'
) => {^
    value_list('title') => {^
        '<br />' + value_listItem + '\n'
    ^}
^}

// =>
// <br />Mr.
// <br />Mrs.
// <br />Ms.
// <br />Dr.
```

The following example shows how to display all values from a value list using a named inline. The same name "values" is referenced by `-inlineName` in both the **inline** method and **resultSet** method.

```
inline(
    -inlineName='Values',
    -show,
    -database='contacts',
    -table='people'
) => {}

// ...

resultSet(1, -inlineName='Values') => {^
    value_list('title') => {^
        '<br />' + value_listItem + '\n'
    ^}
^}

// =>
// <br />Mr.
// <br />Mrs.
```

```
// <br />Ms.
// <br />Dr.
```

44.7.2 Display a Drop-Down Menu with All Values from a Value List

The following example shows how to format an HTML **<select>** drop-down menu to show all the values from a value list. A select list can be created with the same code by including size and multiple parameters within the **<select>** tag. This code is usually used within an HTML form that calls a response page that performs an **-add** or **-update** action so the visitor can select a value from the value list for the record they create or modify.

The example shows a single **<select>** within an **inline** method with a **-show** action. If many value lists from the same database are being formatted, they can all be contained within a single inline.

```
'<form action="response.lasso" method="POST">\n'
inline(
    -show,
    -database='contacts',
    -table='people'
) => {^
    '<select name="title">\n'
    value_list('title') => {^
        '    <option value="' + value_listItem + '">' + value_listItem + '</option>\n'
    ^}
    '</select>\n'
^}
'<p><input type="submit" name="submit" value="Add Record"></p>\n</form>\n'

// =>
// <form action="response.lasso" method="POST">
// <select name="title">
//     <option value="Mr.">Mr.</option>
//     <option value="Mrs.">Mrs.</option>
//     <option value="Ms.">Ms.</option>
//     <option value="Dr.">Dr.</option>
// </select>
// <p><input type="submit" name="submit" value="Add Record"></p>
// </form>
```

44.7.3 Display Radio Buttons with All Values from a Value List

The following example shows how to format a set of HTML **<input>** tags to show all the values from a value list as radio buttons. The visitor will be able to select one value from the value list. Checkboxes can be created with the same code by changing the type from radio to checkbox.

```
'<form action="response.lasso" method="POST">\n'
inline(
    -show,
    -database='contacts',
    -table='people'
) => {^
    value_list('title') => {^
        '    <input type="radio" name="title" value="' + value_listItem + '" /> ' + value_listItem + '\n'
    ^}
^}
'<p><input type="submit" name="submit" value="Add Record"></p>\n</form>\n'
```

```
// =>
// <form action="response.lasso" method="POST">
//    <input type="radio" name="title" value="Mr." /> Mr.
//    <input type="radio" name="title" value="Mrs." /> Mrs.
//    <input type="radio" name="title" value="Ms." /> Ms.
//    <input type="radio" name="title" value="Dr." /> Dr.
// <p><input type="submit" name="submit" value="Add Record"></p>
// </form>
```

44.7.4 Display Only Selected Values from a Value List

The following example shows how to display the selected values from a value list for the current record. The record for "John Doe" is found within the database and the selected value for the "title" field, "Mr.", is displayed.

The **selected** method is used to ensure that only selected value list items are shown. The following example uses a conditional to check whether **selected** is empty and only shows the **value_listItem** if it is not:

```
inline(
    -search,
    -database='contacts',
    -table='people',
    -keyField='id',
    -keyValue=126
) => {^
    value_list('title') => {^
        if(selected != '') => {^
            '<br />' + value_listItem
        ^}
    ^}
^}

// => <br />Mr.
```

The **field** method can also be used simply to display the current value for a field without reference to the value list.

```
'<br />' + field('title')

// => <br />Mr.
```

44.7.5 Display a Drop-Down Menu with Selected Values from a Value List

The following example shows how to format an HTML **<select>** list to show all the values from a value list with the selected values highlighted. The **selected** method returns "selected" if the current value list item is selected in the database or nothing otherwise.

```
'<form action="response.lasso" method="POST">\n'
inline(
    -search,
    -database='contacts',
    -table='people',
    -keyField='id',
    -keyValue=126
) => {^
    '<select name="title" multiple size="4">\n'
```

```
    value_list('title') => {^
        '    <option value="' + value_listItem + '" ' + selected + '>' + value_listItem + '</option>\n'
    ^}
    '</select>\n'
^}
'<input type="submit" name="submit" value="Update Record">\n</form>\n'

// =>
// <form action="response.lasso" method="POST">
// <select name="title" multiple size="4">
//     <option value="Mr." selected>Mr.</option>
//     <option value="Mrs." >Mrs.</option>
//     <option value="Ms." >Ms.</option>
//     <option value="Dr." >Dr.</option>
// </select>
// <input type="submit" name="submit" value="Update Record">
// </form>
```

44.7.6 Display Checkboxes with Selected Values from a Value List

The following example shows how to format a set of HTML `<input>` tags to show all the values from a value list as checkboxes with the selected checkboxes checked. The **checked** method returns "checked" if the current value list item is selected in the database or nothing otherwise. Radio buttons can be created with the same code by changing the type from "checkbox" to "radio".

```
'<form action="response.lasso" method="POST">\n'
inline(
    -search,
    -database='contacts',
    -table='people',
    -keyField='id',
    -keyValue=126
) => {^
    value_list('title') => {^
        '    <input type="checkbox" name="title" value="' + value_listItem + '" ' + checked + '>'
            + value_listItem + '\n'
    ^}
^}
'<input type="submit" name="submit" value="Update Record">\n</form>\n'

// =>
// <form action="response.lasso" method="POST">
//     <input type="checkbox" name="title" value="Mr." checked> Mr.
//     <input type="checkbox" name="title" value="Mrs." > Mrs.
//     <input type="checkbox" name="title" value="Ms." > Ms.
//     <input type="checkbox" name="title" value="Dr." > Dr.
// <input type="submit" name="submit" value="Update Record">
// </form>
```

Note: Storing multiple values is only supported using "SET" field types.

44.8 SQL Statements

Lasso provides the ability to issue SQL statements directly to SQL-compliant data sources, including the MySQL data source. SQL statements are specified within the **inline** method using the **-sql** parameter. Many third-party databases that support SQL statements also support the use of the **-sql** parameter. SQL inlines can be used as the primary method of database interaction in Lasso, or they can be used alongside standard inline actions (e.g. **-search**, **-add**, **-update**, **-delete**) where a specific SQL function is desired that cannot be replicated using standard database commands.

Note: SQL statements are not supported for FileMaker data sources.

For most data sources multiple SQL statements can be specified within the **-sql** parameter separated by a semicolon. Lasso will issue all of the statements to the data source at once and will collect all of the results into result sets. The **resultSet_count** method returns the number of result sets that Lasso found. The **resultSet** method can then be used with an integer parameter to return the results from one of the result sets.

Caution: Visitor-supplied values must be sanitized when they are concatenated into SQL statements. Sanitizing these values ensures that no invalid characters are passed to the data source and helps to prevent SQL injection attacks. The **string->encodeSql** method should be used to encode values for MySQL strings. The **string->encodeSql92** method should be used to encode values for strings for other SQL-compliant data sources including ODBC data sources and SQLite. The **-search**, **-add**, **-update**, etc. database actions automatically sanitize values passed as pairs into an inline.

Table 44.3: SQL Statement Parameters

Parameter	Description
-sql=?	Issues one or more SQL command to a compatible data source. Multiple commands are delimited by a semicolon. When multiple commands are used, all will be executed, however only the first command issued will return results to the **inline** method unless the **resultSet** method is used.
-database=?	The database in the data source in which to execute the SQL statement.
-table=?	A table in the database (used for encoding information).
-maxRecords=?	The maximum number of records to return. Optional, defaults to "50".
-skipRecords=?	The offset into the found set at which to start returning records. Optional, defaults to "1".

The **-database** parameter can be any database within the data source in which the SQL statement should be executed. The **-database** parameter will be used to determine the data source, and table references within the statement can include both a database name and a table name (e.g. "contacts.people") in order to fetch results from multiple tables. For example, to create a new database in MySQL, a **CREATE DATABASE** statement can be executed with **-database** set to a name of a database in the host you want the new database to reside in.

When referencing the name of a database and table in a SQL statement (e.g. "contacts.people"), only the true names of a database can be used as MySQL does not recognize Lasso database aliases in a SQL command.

Results from a SQL statement are returned in a record set within the **inline** method. The results can be read and displayed using the **records** or **rows** methods and the **field** or **column** method. However, many SQL statements return a synthetic record set that does not correspond to the names of the fields of the table being operated upon. This is demonstrated in the examples that follow.

Note: Documentation of SQL itself is beyond the scope of this guide. Please consult the documentation included with your data source for information on which SQL statements it supports.

44.8.1 Issuing SQL Statements

SQL statement are specified within an **inline** method with a **-sql** keyword parameter.

The following example calculates the results of a mathematical expression "1 + 2" and returns the value as a field named "result". Note that even though this SQL statement does not reference a database, a **-database** parameter is still required so Lasso knows to which data source to send the statement:

```
inline(
    -database='example',
    -sql="SELECT 1+2 AS result;"
) => {^
    'The result is: ' + field('result')
^}

// => The result is 3
```

The following example calculates the results of several mathematical expressions and returns them as field values "one", "two", and "three":

```
inline(
    -database='example',
    -sql="SELECT 1+2 AS one, sin(.5) AS two, 5%2 AS three;"
) => {^
    'The results are: ' + field('one') + ', ' + field('two') + ', and ' + field('three')
^}

// => The results are 3, 0.579426, and 1
```

The following example calculates the results of several mathematical expressions using Lasso and returns them as field values "one", "two", and "three". It demonstrates how the results of Lasso expressions and methods can be used in a SQL statement:

```
inline(
    -database='example',
    -sql="SELECT " + (1+2) + " AS one, " + math_sin(0.5) + " AS two, " + (5%2) + " AS three;"
) => {^
    'The results are: ' + field('one') + ', ' + field('two') + ', and ' + field('three')
^}

// => The results are 3, 0.579426, and 1
```

The following example returns records from the "phone_book" table where "first_name" is equal to "John". This is equivalent to a **-search** action:

```
inline(
    -database='contacts',
    -sql="SELECT * FROM phone_book WHERE first_name = 'John';"
) => {^
    records => {^
        '<br />' + field('first_name') + ' ' + field('last_name') + '\n'
    ^}
^}

// =>
// <br />John Doe
// <br />John Person
```

Issue a SQL Statement with Multiple Commands

Specify several SQL statements within an **inline** method in a **-sql** keyword parameter, with each SQL command separated by a semicolon. The following example adds three unique records to the "people" table of the "contacts" database:

```
inline(
    -database='contacts',
    -sql="INSERT INTO people (first_name, last_name) VALUES ('John',  'Jakob');
          INSERT INTO people (first_name, last_name) VALUES ('Tom',   'Smith');
          INSERT INTO people (first_name, last_name) VALUES ('Sally', 'Brown');"
) => {}
```

Determine the Actual Database Name for a SQL Statement

Use the **database_realName** method. When using the **-sql** parameter to issue SQL statements to a host, only true database names may be used (bypassing the alias). The **database_realName** method can be used to automatically determine the true name of a database, allowing them to be used in a valid SQL statement.

```
local(real_db) = database_realName('Contacts_alias')
inline(
    -database='contacts_alias',
    -sql="SELECT * FROM `" + #real_db + "`.people;"
) => {}
```

44.8.2 Sanitizing Visitor-Supplied Values in a SQL Statement

All visitor supplied values must be sanitized before they are concatenated into a SQL statement in order to ensure the validity of the SQL statement and to prevent SQL injection. Values from the **web_request->param**, **web_request->cookie**, and **field** methods should be encoded as well as values from any calculations that rely on these methods. The **string->encodeSql** method should be used to encode values within SQL statements for MySQL data sources. The **string->encodeSql92** method should be used to encode values for other SQL-compliant data sources including ODBC data sources and SQLite.

string->encodeSql()
    Encodes illegal characters in MySQL string literals by escaping them with a backslash. Helps to prevent SQL injection attacks and ensures that SQL statements only contain valid characters. This method must be used to encode visitor supplied values within SQL statements for MySQL strings.

string->encodeSql92()
    Encodes illegal characters in SQL string literals by escaping a single quote with two single quotes. Helps to prevent SQL injection attacks and ensures that SQL statements only contain valid characters. This method can be used to encode values for SQLite and most other SQL-compliant data sources.

The following example encodes the query or post parameter for "first_name" for a MySQL data source:

```
inline(
    -database='contacts',
    -sql="SELECT * FROM phone_book WHERE first_name = '" +
        string(web_request->param('first_name'))->encodeSql + "';"
) => {}
```

The following example encodes the query or post parameter "first_name" for a SQLite (or other SQL-compliant) data source:

```
inline(
    -database='contacts',
    -sql="SELECT * FROM phone_book WHERE first_name = '" +
```

```
        string(web_request->param('first_name'))->encodeSql92 + "';"
) => {}
```

Important: The **string->encodeSql** and **string->encodeSql92** methods can only be used to sanitize data being used as SQL string data in the SQL expression. If you need to sanitize data being used as integer or decimal data, use those creator methods to verify the object is of those types. To sanitize a date object, use the **date->format** method and make sure the format string doesn't contain invalid characters. If you need to use variables to specify database, table, or column names inside a SQL statement, then you will need to take additional precautions that vary by data source. All of this is to say that you should always sanitize your inputs, and simply using the **encodeSql** methods is not enough.

44.8.3 Automatically Formatting SQL Statement Results

Use the **field_name** method and **loop** method to create an HTML table that automatically formats the results of a **-sql** command. The **-maxRecords** parameter should be set to "All" so all records are returned rather than the default (50).

The following example shows a **REPAIR TABLE contacts.people** SQL statement being issued to a MySQL database, and the result is automatically formatted. The statement returns a synthetic record set that shows the results of the repair.

Notice that the database "contacts" is specified explicitly within the SQL statement. Even though the database is identified in the **-database** parameter within the inline it may still be explicitly specified in each table reference within the SQL statement.

```
inline(
    -database='contacts',
    -sql="REPAIR TABLE contacts.people;",
    -maxRecords='all'
) => {^
    '<table border="1">\n'
    '<tr>\n'
    loop(field_name(-count)) => {^
        '    <td><b>' + field_name(loop_count) + '</b></td>\n'
    ^}
    '</tr>\n'
    records => {^
        '<tr>\n'
        loop(field_name(-count)) => {^
            '    <td>' + field(field_name(loop_count)) + '</td>\n'
        ^}
        '</tr>\n'
    ^}
    '</table>\n'
^}
```

The results are returned in a table with bold column headings. The following results show that the table did not require any repairs. If repairs are performed then many more records will be returned.

```
// =>
// <table border="1">
// <tr>
//     <td><b>Table</b></td>
//     <td><b>Op</b></td>
//     <td><b>Msg_type</b></td>
//     <td><b>Msg_text</b></td>
// </tr>
// <tr>
//     <td>people</td>
```

```
//     <td>Check</td>
//     <td>Status</td>
//     <td>OK</td>
// </tr>
// </table>
```

44.8.4 Using Result Sets

An inline that uses a `-sql` action can return multiple result sets. Each SQL statement within the `-sql` action is separated by a semicolon and generates its own result set. This allows multiple SQL statements to be issued to a data source in a single connection and for the results of each statement to be reviewed individually.

In the following example the **resultSet_count** method is used to report the number of result sets that the inline returned. Since the `-sql` parameter contains two SQL statements, two result sets are returned. The two result sets are then looped through by passing the **resultSet_count** method to the **loop** method and passing the **loop_count** as the parameter for the **resultSet** method. Finally, the **records** method is used as normal to display the records from each result set.

```
inline(
    -database='contacts',
    -sql="SELECT CONCAT(first_name, ' ', last_name) AS name FROM people; SELECT name FROM companies;"
) => {^
    resultSet_count + ' Result Sets\n'
    '<hr />\n'
    loop(resultSet_count) => {^
        resultSet(loop_count) => {^
            records => {^
                '<br />' + field('name') + '\n'
            ^}
            '<hr />\n'
        ^}
    ^}
^}

// =>
// 2 Result Sets
// <hr />
// <br />John Doe
// <br />Jane Doe
// <hr />
// <br />LassoSoft
// <hr />
```

The same example can be rewritten using a named inline. An `-inlineName` parameter with the name "MyResults" is added to the **inline** method, the **resultSet_count** method, and the **resultSet** method. This way the result sets can be output from anywhere after the inline. The results of the following example will be the same as those shown above:

```
inline(
    -inlineName='MyResults',
    -database='contacts',
    -sql="SELECT CONCAT(first_name, ' ', last_name) AS name FROM people; SELECT name FROM companies;"
) => {}

// ...

resultSet_count(-inlineName='MyResults') + ' Result Sets\n<hr />'
loop(resultSet_count(-inlineName='MyResults')) => {^
```

```
    resultSet(loop_count, -inlineName='MyResults') => {^
        records => {^
            '<br />' + field('name')
        ^}
        '<hr />'
    ^}
^}
```

44.9 SQL Transactions

Lasso supports the ability to perform *SQL transactions*, which are reversible groups of statements, provided that the data source used (e.g. MySQL 4 and later with certain storage engines) supports this functionality. See your data source documentation to see if transactions are supported.

SQL transactions can be achieved within nested **inline** methods. A single connection to MySQL or ODBC data sources will be held open around the outer inline. Any nested inlines that use the same data source will make use of the same connection.

Note: When using named inlines, the connection is not available in subsequent `records(-inlineName='Name')` methods.

44.9.1 Open a Transaction and Commit or Rollback in MySQL

Use nested `-sql` inlines, where the outer inline performs a transaction, and the inner inline commits or rolls back the transaction depending on the results of a conditional statement.

```
inline(
    -database='contacts',
    -sql="START TRANSACTION;
          INSERT INTO contacts.people (title, company) VALUES ('Mr.', 'LassoSoft');"
) => {
    if(error_currentError != error_msg_noerror) => {
        inline(-database='contacts', -sql="ROLLBACK;") => {}
    else
        inline(-database='contacts', -sql="COMMIT;") => {}
    }
}
```

44.9.2 Fetch the Last Inserted ID in MySQL

Use nested `-sql` inlines, where the outer inline performs an insert query, and the inner inline retrieves the ID of the last inserted record using the MySQL `last_insert_id()` function. Because the two inlines share the same connection, the inner inline will always return the value added by the outer inline.

```
inline(
    -database='contacts',
    -sql="INSERT INTO people (title, company) VALUES ('Mr.', 'LassoSoft');"
) => {^
    inline(-sql="SELECT last_insert_id();") => {^
        field('last_insert_id()')
    ^}
^}
```

```
// => 23
```

44.10 Prepared Statements

Lasso supports the ability to use prepared statements to speed up database operations provided that the data source used (e.g. MySQL 4 and later) supports this functionality. See your data source documentation to see if prepared statements are supported.

A *prepared statement* is a cached database query that can speed up database operations by cutting down on the amount of overhead that the data source needs to perform for each statement. For example, processing the following "INSERT" statement requires the data source to load the people table, determine its primary key, load information about its indexes, and determine default values for fields not listed. After the new record is inserted the indexes must be updated. If another "INSERT" is performed then all of these steps are repeated from the beginning:

```
INSERT INTO people (`first_name`, `last_name`) VALUES ("John", "Doe");
```

When this statement is changed into a prepared statement then the data source knows to expect multiple executions of the statement. The data source can cache information about the table in memory and reuse that information for each execution. The data source might also be able to defer some operations such as finalizing index updates until after several statements have been executed.

The specific details of how prepared statements are treated are dependent on the data source. The savings in overhead and increase in speed may vary depending on what type of SQL statement is being issued, the size of the table and indexes that are being used, and other factors.

The statement above can be rewritten as a prepared statement by replacing the values with question marks. The name of the table and field list are defined just as they were in the original SQL statement. This statement is a template into which particular values will be placed before the data source executes it:

```
INSERT INTO people (`first_name`, `last_name`) VALUES (?, ?)
```

The particular values are specified as an array. Each element of the array corresponds with one question mark from the prepared statement. To insert "John Doe" into the "people" table the following array would be used:

```
array('John', 'Doe')
```

One new database action is used to prepare statement and execute them: `-prepare` is similar to `-sql`, but informs Lasso that you want to create a prepared statement. Nested inlines are then issued with an array and the `-sql` parameter. The array should contain values that should be plugged into the prepared statement.

The prepared statement and values shown above would be issued by the following inlines. The outer inline prepares the statement and the inner inline executes it with specific values. Note that the inner inline does not contain any `-database` or `-table` parameters. These are inherited from the outer inline so they don't need to be specified again.

```
inline(
    -database='contacts',
    -prepare="INSERT INTO people (`first_name`, `last_name`) VALUES (?, ?);"
) => {
    inline(array('John', 'Doe'), -sql) => {}
}
```

If the executed statement returns any values then those results can be inspected within the inner inline. The inline with the `-prepare` action will not return any results itself, but each inner inline with a `-sql` parameter may return a result as if the full equivalent SQL statement were issued in that inline.

ODBC Data Sources

This chapter documents methods and behaviors that are specific to the ODBC data source in Lasso. Native support for ODBC data sources is included in Lasso. This feature allows Lasso to communicate with dozens of ODBC-compliant data sources including Sybase, DB2, Frontbase, OpenBase, Interbase, and Microsoft SQL Server. For more information on ODBC connectivity and availability for a particular data source, see the documentation for the data source or contact the data source provider.

Lasso accesses ODBC drivers that are set up as System DSNs. An ODBC Data Source Administrator utility or control panel should be used to configure the driver as a System DSN, after which the data source name can be entered into Lasso. See the *Datasource Setup* chapter for additional details.

45.1 Supported Features for ODBC Data Sources

The following chart details the features of this data source connector.

Friendly Name
: Lasso Connector for ODBC

Internal Name
: odbc

Module Name
: SQLConnector.dll, SQLConnector.dylib, or SQLConnector.so

Inline Host Attributes
: The `-name` should specify the data source name (System DSN). A `-username` and `-password` may also be required.

Actions
: `-add`, `-delete`, `-findAll`, `-search`, `-show`, `-sql`, `-update`

Operators
: `-bw`, `-cn`, `-eq`, `-ew`, `-gt`, `-gte`, `-lt`, `-lte`, `-nbw`, `-ncn`, `-new`; `-opBegin`/`-opEnd` with "And", "Or", "Not".

KeyField
: `-keyField`/`-keyValue`

45.2 ODBC Data Source Tips

The following is a list of guidelines to follow when writing Lasso code that interfaces with ODBC data sources.

- Always specify a primary key field using the `-keyField` parameter in `-search`, `-add`, and `-findAll` actions. This will ensure that the **keyField_value** method will always return a value.

- Use `-keyField` and `-keyValue` to reference a particular record for updates or deletes.

- Some data sources will truncate any data beyond the length they are set up to store. Verify that all fields have sufficient capacity for the values that need to be stored in them.

- Use **-returnField** parameters to reduce the number of fields that are returned from a **-search** action. Returning only the fields that need to be used for further processing or shown to the site visitor reduces the amount of data that needs to travel between Lasso and the data source.

- When an **-add** or **-update** action is performed on a database, the data from the added or updated record is available inside the capture block of the **inline** method. If the **-returnField** parameter is used, then only those fields specified should be returned from an **-add** or **-update** action. Setting **-maxRecords=0** can be used as an indication that no record should be returned.

- SQL statements that are generated using visitor-defined data should be screened carefully for unwanted commands such as "DROP" or "GRANT".

- Always sanitize any inputs from site visitors that are incorporated into SQL statements. For example, any SQL strings that have visitor-defined data should be sanitized using the **string->encodeSql** method for MySQL-like data sources or the **string->encodeSql92** method for SQL92-compliant data sources or ODBC data sources. Encoding the values in this manner ensures that quotes and other reserved characters are properly escaped within the SQL statement, thereby helping to prevent SQL injection attacks.

  For example, the following SQL "SELECT" statement contains a SQL string in the LIKE clause and uses **string->encodeSql92** to encode the value of the **'company' web_request->param**. This encoding causes all single quotes within the passed **company** parameter to be encoded with an additional single quote.

  ```
  local(sql_statement) = "SELECT * FROM contacts.people WHERE company LIKE '" +
        string(web_request->param('company'))->encodeSql92 + "%';"
  ```

  If **web_request->param('company')** returns "McDonald's" then the SQL statement generated by this code would appear as follows:

  ```
  SELECT * FROM Contacts.People WHERE Company LIKE "McDonald's%";
  ```

- Lasso Server uses connection pooling when connecting to data sources via ODBC, and the ODBC connections will remain open during the time that Lasso Server is running.

45.3 Using ODBC Data Sources

Data source operations outlined in the *Database Interaction Fundamentals*, *Searching and Displaying Data*, and *Adding and Updating Records* chapters are supported with ODBC data sources. Because ODBC is a standardized API for connecting to tabular data sources, there are no unique methods in Lasso that are specific to ODBC data sources or invoke special functions specific to any ODBC data source.

FileMaker Data Sources

Lasso Server allows access to FileMaker Server 7–12 Advanced and FileMaker Server 9–12 through the Lasso Connector for FileMaker. Lasso provides several methods and options that are unique to FileMaker Server connections including `-layoutResponse` and `-noValueLists`.

While Lasso is a predominantly data source–independent platform, it does include many FileMaker-specific options as documented in this chapter. However, all of the common procedures outlined in the *Database Interaction Fundamentals*, *Searching and Displaying Data*, and *Adding and Updating Records* chapters can be used with FileMaker data sources.

Important: The methods and options defined in this chapter can only be used with FileMaker data sources. Any solution that relies on the methods in this chapter cannot be easily retargeted to work with a different data source.

46.1 Lasso and FileMaker

Since Lasso works with many different data sources this documentation uses data source–agnostic terms to refer to databases, tables, and fields. The following terms that are used in the FileMaker documentation are equivalent to their Lasso counterparts:

Database
> Database is used to refer to a single FileMaker database file. FileMaker databases differ from other databases in Lasso in that they contain layouts rather than individual data tables. Even in FileMaker Server 7–12, Lasso sees individual layouts rather than data tables. From a data storage point of view, a FileMaker database is equivalent to a single MySQL table.

Layout
> Within Lasso a FileMaker layout is treated as equivalent to a table. The two terms can be used interchangeably. This equivalence simplifies Lasso security and makes transitioning between data sources easier. All FileMaker layouts can be thought of as views of a single data table. Lasso can only access fields that are contained in the layout named within the current database action.

Record
> FileMaker records are referenced using a single `-keyValue` rather than a `-keyField` and `-keyValue` pair. The `-keyField` in FileMaker is always the Record ID that is set internally.

Field
> The value for any field in the current layout in FileMaker can be returned including the values for related fields, repeating fields, and fields in portals.

46.1.1 Performance Tips

This section contains a number of tips that will help get the best performance from a FileMaker database. Since queries must be performed sequentially within FileMaker Server, even small optimizations can yield significant increases in the speed of web serving under heavy load.

- **Dedicated FileMaker Server Machine** – For best performance, place the FileMaker Server on a different machine from Lasso Server and the web server.

- **FileMaker Server** – If a FileMaker database must be accessed by a mix of FileMaker clients and web visitors through Lasso, it should be hosted on FileMaker Server. Lasso can access the database directly through FileMaker Server 7–12 Advanced and FileMaker Server 9–12.

- **Index Fields** – Any fields that will be searched through Lasso should have indexing turned on. Avoid searching on non-stored calculation fields, related fields, and summary fields.

- **Custom Layouts** – Layouts should be created with the minimal number of fields required for Lasso. All the data for the fields in the layout will be sent to Lasso with the query results. Limiting the number of fields can dramatically cut down the amount of data that needs to be sent from FileMaker Server to Lasso.

- **Value Lists** – For FileMaker Server data sources use the `-noValueLists` parameter to suppress the automatic sending of value lists from FileMaker when those value lists are not going to be used on the response page.

- **Layout Response** – For FileMaker Server data sources use the `-layoutResponse` parameter to specify what layout should be used to return results from FileMaker. A different layout from what was specified in the request can be used for the result. This is a replacement for the `-returnField` parameter, which is not supported for FileMaker data sources.

- **Sorting** – Sorting can have a serious impact on performance if large numbers of records must be sorted. Avoid sorting large record sets and avoid sorting on calculation fields, related fields, unindexed fields, or summary fields.

- **Contains Searching** – FileMaker is optimized for the default "Begins With" searches (and for numerical searches). Use of the contains operator (`-cn`) can dramatically slow down performance since FileMaker will not be able to use its indices to optimize searches.

- **Max Records** – Using `-maxRecords` to limit the number of records returned in the result set from FileMaker Server can speed up performance. Use `-maxRecords` and `-skipRecords` methods to navigate a visitor through the found set.

- **Calculation Fields** – Calculation fields should be avoided if possible. Searching or sorting on unindexed, uncached calculation fields can have a negative effect on FileMaker Server performance.

- **FileMaker Scripts** – The use of FileMaker scripts should be avoided if possible. While FileMaker executes a script, no other database actions can be performed. FileMaker scripts can usually be rewritten as Lasso code to achieve the same effect, often with greater performance.

In addition to these tips, MySQL or PostgreSQL can be used to shift some of the burden off of FileMaker Server. MySQL and PostgreSQL can usually perform database searches much faster than FileMaker. Lasso also includes sessions and collection types that can be used to perform some of the tasks of a database, but with higher performance for small amounts of data.

46.1.2 Compatibility Tips

Following these tips will help to ensure that it is easy to transfer data from a FileMaker database to another data source, such as a PostgreSQL database, at a future date.

- **Database Names** – Database, layout, and field names should contain only a mix of letters, numbers, and the underscore character.

- **Calculation Fields** – Avoid the use of calculation fields. Instead, perform calculations within Lasso and store the results back into regular fields if they will be needed later.

- **Summary Fields** – Avoid the use of summary fields. Instead, summarize data using `inline` searches within Lasso.

- **Scripts** – Avoid the use of FileMaker scripts. Most actions performed with scripts can be performed using the database actions available within Lasso.

- **Record ID** – Create a calculation field with the calculation `Status(CurrentRecordID)` and name it "id". Always use the `-keyField='id'` within `inline` database actions. This ensures that when moving to a database that relies on storing the key field value explicitly, a unique key field value is available.

46.2 FileMaker Queries

The queries generated by inlines for FileMaker data sources differ from the queries generated for other data sources in several significant ways. This section includes a description of how search operators, logical operators, and other keyword parameters are used to construct queries for each of the FileMaker data sources.

46.2.1 Search Operators

By default FileMaker performs a "begins with" search for each field in a query. In FileMaker Server each field can only be specified one time within each search query. See the information below on FileMaker search symbols for strategies to perform complex queries in FileMaker Server.

Lasso also provides the following operators that allow different queries to be performed. Each operator should be specified immediately before the field and its search value are specified. Note that this list of operators is somewhat different from those supported by other data source connectors including older FileMaker data source connectors.

Table 46.1: FileMaker Search Field Operators

Operator	Description
`-op='bw'` or `-bw`	Begins With. Matches records where any word in the field begins with the specified substring. This is the default if no other operator is specified.
`-op='cn'` or `-cn`	Contains. Matches records where any word in the field contains the substring.
`-op='eq'` or `-eq`	Equals. Matches records where any word in the field exactly matches the string.
`-op='ew'` or `-ew`	Ends With. Matches records where any word in the field ends with the specified substring.
`-op='gt'` or `-gt`	Greater Than. Matches records where the field value is greater than the parameter.
`-op='gte'` or `-gte`	Greater Than or Equals.
`-op='lt'` or `-lt`	Less Than. Matches records where the field value is less than the parameter.
`-op='lte'` or `-lte`	Less Than or Equals.
`-op='rx'` or `-rx`	Use a FileMaker search expression. See the table below for a list of symbols.

Note that there is no `-neq` operator or other negated operators. It is necessary to use a `-not` query to omit records from the found set instead, as explained further below. For example, to find records where the field "first_name" is not "Joe" the following search terms must be used: `-not,-op='eq','first_name'='Joe'`

The `-rx` operator can be used to pass a raw FileMaker search expression as a query. This allows the use of any of the FileMaker search symbols. See the FileMaker documentation[60] for a full explanation of how these symbols work.

---

[60] http://www.filemaker.com/support/product/documentation.html

Table 46.2: FileMaker Search Symbols

Symbol	Description
@	Matches one character.
*	Matches zero or more characters. A single * matches non-empty fields.
..	Matches a range of values such as "1..10" or "A..Z". Can be written as two or three periods.
#	Matches one number.
" "	Quotes surround a substring that should be matched literally.
=	Matches a whole word. "=John" will match "John", but not "Johnny". A single = matches empty fields.
==	Matches a whole field value rather than per-word. Should be specified at the start of the search term.
<	Matches values less than a specified value.
<=	Matches values less than or equal to a specified value.
>	Matches values greater than a specified value.
>=	Matches values greater than or equal to a specified value.
//	Matches today's date.
?	Matches a record with invalid date data in the field.
!	Matches records that have a duplicate value. Both records will be returned.

The range symbol (`..`) is most useful for performing searches within a date range. For example a date in 2006 can be found by searching for `-rx,'date_field'='1/1/2006..12/31/2006'`.

46.2.2 Logical Operators

FileMaker data sources default to performing an AND search. The records that are returned from the data source must match all of the specified criteria. It is also possible to specify `-opLogical` to switch to an OR search where the records that are returned from the data source may match any of the specified criteria.

For example, the following criteria returns records where the "first_name" is "John" and the "last_name" is "Doe": `-eq,'first_name'='John',-eq,'last_name'='Doe'`

The following criteria instead returns records where the "first_name" is "John" or the "last_name" is "Doe". This would return records for "John Doe" as well as "Jane Doe" and "John Walker": `-opLogical='Or',-eq,'first_name'='John',-eq,'last_name'='Doe'`

46.2.3 Complex Queries with FileMaker Server 9 and Later

Starting with FileMaker Server 9, a search request is made up of one or more queries. By default a single query is generated and all of the search terms within it are combined using an AND operator. Additional queries can be added to either extend the found set using an OR operator or to omit records from the found set using a NOT operator. These queries correspond precisely to find requests within the FileMaker Server user interface.

Each field can only be listed once per query. The standard Lasso operators can be used for most common search parameters like equals, begins with, ends with, contains, less than, greater than, etc. FileMaker's standard find symbols can be used for more complex criteria. It may also be necessary to use multiple queries for more complex search criteria.

Search requests in FileMaker Server 9 and later do not support the "Not Equals" operator or any of the NOT-variant operators. Instead, these should be created by combining an omit query with the appropriate affirmative operator. The `-opLogical`, `-opBegin`, and `-opEnd` operators are not supported. The `-or` and `-not` operators must be used instead.

Table 46.3: FileMaker Search Operator Parameters

Parameter	Description
-or	Starts a new query. Records that match the query will be added to the result set.
-not	Starts an omit query. Records that match the query will be omitted from the result set.

A search with a single query uses an AND operator to combine each of the search terms. Records where the field "first_name" begins with the letter "J" and the field "last_name" begins with the letter "D" can be found using the following search terms in Lasso. Each record in the result set will match every search term in the query:
`-bw,'first_name'='J',-bw,'last_name='D'`

We start an additional query using an **-or** parameter. FileMaker runs the first and second queries independently and then combines the search results. The result of the following search terms will be to find every record where the field "first_name" begins with the letter "J" and the field "last_name" begins with either the letter "D" or the letter "S". Each record in the result set will match either the first query or the second query.

```
-bw, 'first_name'='J',
-bw, 'last_name'='D',
-or,
-bw, 'first_name'='J',
-bw, 'last_name'='S'
```

Note that each field name can only appear once per query, but the same field name can be used in multiple queries. The "first_name" search term is repeated in both queries so that all returned records will have a "first_name" starting with "J". If the "first_name" search term was left out of the second query then the result set would contain every record where the field "first_name" begins with the "J" and the field "last_name" begins with the letter "D" and every record where the field "last_name" begins with the letter "S".

The result set can be narrowed by adding an omit query using a **-not** parameter. FileMaker will run the first query and any **-or** queries first, generating a complete result set. Then, the **-not** queries will be run and any records that match those queries will be omitted from the found set. The result of the following search terms will be to find every record where the field "first_name" begins with the letter "J" and the field "last_name" begins with the letter "D" except for the record for "John Doe". Each record in the result set will match the first query and will not match the second query.

```
-bw, 'first_name'='J',
-bw, 'last_name'='D',
-not,
-bw, 'first_name'='John',
-bw, 'last_name'='Doe'
```

It is possible to construct most searches positively using only a single query or a few **-or** queries, but sometimes it is more logical to construct a large result set and then use one or more **-not** queries to omit records from it.

46.2.4 Additional Commands for FileMaker Server 9 and Later

FileMaker Server 9 supports a number of additional unique commands that are summarized in the following table. Most of these commands are passed through to FileMaker Server without modification by Lasso. The *FileMaker Server 9 Custom Web Publishing with XML and XSLT documentation* should be consulted for full details about these commands.

Table 46.4: FileMaker Additional Parameters

Parameter	Description
`-layoutResponse=?`	Returns the result set using the layout specified in this parameter rather than the layout used to specify the database action.
`-noValueLists`	Suppresses the fetching of value list data for FileMaker Server data sources.
`-relatedSets.filter=?`	If set to "layout", FileMaker Server will return only the number of related records shown in portals on the current layout. Defaults to returning all records up to the number set by `-relatedSets.max`.
`-relatedSets.max=?`	Sets the number of related records returned. Can be set to a number or "All".
`-script=?` and `-script.param=?`	Runs a script after the find has been processed and sorted. The optional parameter can be accessed from within the script.
`-script.preFind=?` and `-script.preFind.param=?`	Runs a script before the find is processed.
`-script.preSort=?` and `-script.preSort.param=?`	Runs a script after the find has been processed, but before the results are sorted.

46.3 Primary Key Field and Record ID

FileMaker databases include a built-in primary key value called the Record ID. This value is guaranteed to be unique for any record in a FileMaker database. It is predominantly sequential, but should not be relied upon to be sequential. The values of the Record IDs within a database may change after an import or after a database is compressed using *Save a Copy As…*. Record IDs can be used within a solution to refer to a record on multiple pages, but should not be stored as permanent references to FileMaker records.

46.3.1 Return the Current Record ID

The Record ID for the current record can be returned using **keyField_value**. The following example shows an **inline** method that performs a **-findAll** action and returns the Record ID for each returned record using the **keyField_value** method:

```
inline(
    -findAll,
    -database='contacts',
    -table='people'
) => {^
    records => {^
        '<br />' + keyField_value + ': ' + field('first_name') + ' ' + field('last_name') + '\n'
    ^}
^}

// =>
// <br />126: John Doe
// <br />127: Jane Doe
// <br />4096: Jane Person
```

46.3.2 Reference a Record by Record ID

For **-update** and **-delete** action parameters the Record ID for the record being operated upon can be referenced using **-keyValue**. The **-keyField** does not need to be defined or should be set to an empty string if it currently is (**-keyField=''**). The following example shows a record in "contacts" being updated with **-keyValue=126**. The name of the person referenced by the record is changed to "John Surname".

```
inline(
    -update,
    -database='contacts',
    -table='people',
    -keyValue=126,
    'first_name'='John',
    'last_name'='Surname'
) => {^
    keyField_value + ': ' + field('first_name') + ' ' + field('last_name')
^}

// => 126: John Surname
```

The following example shows a record in "contacts" being deleted with **-keyValue=127**. The **-keyField** keyword parameter is included, but its value is set to the empty string.

```
inline(
    -delete,
    -database='contacts',
    -table='people',
    -keyField='',
    -keyValue=127
) => {}
```

Tip: The calculation value **Status(CurrentRecordID)** can be used to access the Record ID for the current record.

46.4 Sorting Records

In addition to the "ascending" and "descending" values for the **-sortOrder** keyword parameter, FileMaker data sources can also accept a custom value. In FileMaker Server, the value for **-sortOrder** should name a value list. The order of that value list will be used as the custom sorting order for records in the result set. Note also that FileMaker Server only supports specifying up to nine sort fields in a single database search.

46.4.1 Return Results in Custom Sort Order

Specify **-sortField** and **-sortOrder** keyword parameters within the search inline. The records are first sorted by "title" in custom order, then by "last_name" and "first_name" in ascending order. The "title" field will be sorted in the order of the elements within the value list associated with the field in the database. In this case, it will be sorted as "Mr., Mrs., Ms.".

```
inline(
    -findAll,
    -database='contacts',
    -table='people',
    -keyField='id',
    -sortField='title',      -sortOrder='title',
```

```
      -sortField='last_name',  -sortOrder='ascending',
      -sortField='first_name', -sortOrder='ascending'
  ) => {^
    records => {^
      '<br />' + field('title') + ' ' + field('first_name') + ' ' + field('last_name') + '\n'
    ^}
  ^}
```

The following results could be returned when this page is loaded. Each of the records with a title of "Mr." appear before each of the records with a title of "Mrs.". Within each title, the names are sorted in ascending alphabetical order.

```
  // =>
  // <br />Mr. John Doe
  // <br />Mr. John Person
  // <br />Mrs. Jane Doe
  // <br />Mrs. Jane Person
```

46.5 Displaying Data

FileMaker includes a number of methods that allow the different types of FileMaker fields to be displayed. These methods are summarized below, and examples are included in the sections that follow.

field(…)
    Can be used to reference FileMaker fields including related fields and repeating fields. Fields from the current table are named simply (e.g. **field('first_name')**). Fields from a related record are named with the related database name, two colons, and the name of the field (e.g. **field('Calls::Approved')**). Repeating fields include the repetition number in parentheses (e.g. **field('Responses(3)')**).

repeating(*name::string*)
    Executes a capture block once for each defined repetition of a repeating field. Requires a single parameter, the name of the repeating field from the current layout.

repeating_valueItem()
    Returns the value for each repetition of a repeating field.

portal(*name::string*)
    Executes a capture block once for each record in a portal. Requires a single parameter, the name of the portal relationship from the current layout. Fields from the portal can be found using the same method as for related records (e.g. **field('Calls::Approved')** within a portal showing records from the "Calls" database).

Note: All fields that Lasso references must be contained in the current layout in FileMaker. For portals and repeating fields only the number of repetitions shown in the current layout will be available to Lasso.

46.5.1 Related Fields

Related fields are named using the relationship name followed by two colons and the field name. For example, a related field "call_duration" from a "calls" database might be referenced as **calls::call_duration**. Any related fields included in the layout specified for the current Lasso action can be referenced using this syntax. Data can be retrieved from related fields or it can be set in related fields when records are added or updated.

Return Data from a Related Field

Specify the name of the related field within a **field** method. The related field must be contained in the current layout either individually or within a portal. In a one-to-one relationship, the value from the single related record will be returned. In a one-to-many relationship, the value from the first related record as defined by the relationship options will be returned. See the section *Portals* below for more control over one-to-many relationships.

The following example shows a **-findAll** action being performed in a database "contacts". The related field "last_call_time" from the "calls" database is returned for each record through a relationship named "calls".

```
inline(
    -findAll,
    -database='contacts',
    -table='people'
) => {^
    records => {^
        '<br />' + keyField_value + ': ' + field('first_name') + ' ' + field('last_name') +
        '(Last call at: ' + field('calls::last_call_time') + ').\n'
    ^}
^}

// =>
// <br />126: John Doe (Last call at 12:00 pm).
// <br />127: Jane Doe (Last call at 9:25 am).
// <br />496: Jane Person (Last call at 4:46 pm).
```

Set Value for a Related Field

Specify the name of the related field, along with the related field's Record ID, within the action that adds or updates a record. The related field must be contained in the current layout either individually or within a portal.

In one-to-one or one-to-many relationships, the fully qualified field name must be used along with the Record ID of the related field in the format **table::field.id**, where "id" is the related field's Record ID. See the section *Portals* below for more information.

The following example shows an **-update** action being performed in a database "contacts". The related field "last_call_time", with a Record ID of "9", from the "calls" database is updated for "Jane Person". The new value is returned.

```
inline(
    -update,
    -database='contacts',
    -table='people',
    -keyField='',
    -keyValue='7',
    'Calls::last_call_time.9'='12:14:56'
) => {^
    field('calls::last_call_time')
^}

// => 12:14:56
```

Important: Every database that is referenced by a related field or a portal must have the same permissions defined. If a related database does not have the proper permissions then not only will FileMaker Server leave the related fields blank, but will deny the entire database request.

46.5.2 Portals

A *portal* allows one-to-many relationships to be displayed within FileMaker databases. Portals allow data from many related records to be retrieved and displayed in a single Lasso page. A portal must be present in the current FileMaker layout in order for its values to be retrieved using Lasso.

Only the number of repetitions formatted to display within FileMaker will be displayed using Lasso. A portal must contain a scroll bar in order for all records from the portal to be displayed using Lasso.

Fields in portals are named using the same convention as related fields. The relationship name is followed by two colons and the field name. For example, a related field "call_duration" from a "calls" database might be referenced as `calls::call_duration`.

Tip: Everything that is possible to do with portals can also be performed using nested **inline** capture blocks to perform actions in the related database. Portals are unique to FileMaker databases.

Return Values from a Portal

Use the **portal** method with the name of the portal referenced. The **field** method within the **portal** capture block should reference the fields from the current portal row using the relationship field syntax.

The following example shows a portal "calls" that is contained in the "people" layout of the "contacts" database. The "time", "duration", and "number" of each call is displayed.

```
inline(
    -findAll,
    -database='contact',
    -table='people'
) => {^
    records => {^
        '<p>Calls for ' + field('first_name') + ' ' + field('last_name') + ':\n'
        portal('calls') => {^
            '<br />' + field('calls::number') + ' at ' + field('calls::time') +
            'for ' + field('calls::duration') + ' minutes.\n'
        ^}
        '</p>\n'
    ^}
^}

// =>
// <p>Calls for John Doe:
// <br />555-1212 at 12:00 pm for 15 minutes.
// </p>
// <p>Calls for Jane Doe:
// <br />555-1212 at 09:25 am for 60 minutes.
// </p>
// <p>Calls for Jane Person:
// <br />555-1212 at 2:23 pm for 55 minutes.
// <br />555-1212 at 4:46 pm for 5 minutes.
// </p>
```

Add a Record to a Portal

A record can be added to a portal by adding the record directly to the related database. In the following example the "calls" database is related to the "contacts" database by means of a field "contact_id" that stores the ID for the contact to which the

calls were made. New records added to "calls" with the appropriate "contact_id" will be shown through the portal to the next site visitor.

In the following example a new call is added to the "calls" database for John Doe. John Doe has an ID of "123" in the "people" table of the "contacts" database. This is the value used for the "contact_id" field in "calls".

```
inline(
    -add,
    -database='calls',
    -table='people',
    'contact_id'=123,
    'number'='555-1212',
    'time'='12:00 am',
    'duration'=55
) => {}
```

46.5.3 Value Lists

Value lists in FileMaker allow a set of possible values to be defined for a field. The items in the value list associated with a field on the current layout for a Lasso action can be retrieved using the **value_list** methods as shown in the examples below. See the FileMaker documentation[61] for more information on how to create and use value lists.

In order to display values from a value list, the layout referenced in the current database action must contain a field formatted to show the desired value list as a drop-down menu, select list, checkboxes, or radio buttons. Lasso cannot reference a value list directly, but can reference a value list through a formatted field in the current layout.

value_list(_colName::string_**)**
> Executes a capture block for each value in the named value list. Requires a single parameter: the name of a field from the current layout that has a value list assigned to it.

value_listItem()
> While in a **value_list** capture block, it returns the value for the current item.

selected()
> Displays the word "selected" if the current value list item is selected in the field associated with the value list.

checked()
> Displays the word "checked" if the current value list item is selected in the field associated with the value list.

Display All Values from a Value List

The following example shows how to display all values from a value list using a **-show** action within an **inline** capture block. The field "title" in the "people" table contains four values: "Mr.", "Mrs.", "Ms.", and "Dr.". The **-show** action allows the values for value lists to be retrieved without performing a database action.

```
inline(
    -show,
    -database='contacts',
    -table='people'
) => {^
    value_list('title') => {^
        value_listItem + ', '
    ^}
^}

// => Mr., Mrs., Ms., Dr.,
```

---

[61] http://www.filemaker.com/support/product/documentation.html

Display a Drop-Down Menu with All Values from a Value List

The following example shows how to format an HTML `<select>` drop-down menu to show all the values from a value list. A select list can be created with the same code by including a `"size"` and/or `"multiple"` option within the `<select>` tag. This code is usually used within an HTML form that submits to a page that performs an `-add` action so the visitor can select a value from the value list for the record they create.

The example shows a single `<select>` tag within an **inline** capture block with a `-show` command. If many value lists from the same database are being formatted, they can all be contained within a single inline.

```
'<form action="response.lasso" method="post">\n'
inline(
    -show,
    -database='contacts',
    -table='people'
) => {^
    '<select name="title">\n'
        value_list('title') => {^
            '    <option value="' + value_listItem + '">' + value_listItem + '</option>\n'
        ^}
    '</select>\n'
^}
'<p><input type="submit" name="submit" value="Add Record">\n</form>\n'

// =>
// <form action="response.lasso" method="post">
// <select name="title">
//     <option value="Mr." selected>Mr.</option>
//     <option value="Mrs." >Mrs.</option>
//     <option value="Ms." >Ms.</option>
//     <option value="Dr." >Dr.</option>
// </select>
// <p><input type="submit" name="submit" value="Add Record"></p>
// </form>
```

Display Radio Buttons with All Values from a Value List

The following example shows how to format a set of HTML `<input>` tags to show all the values from a value list as radio buttons. The visitor will be able to select one value from the value list. Checkboxes can be created with the same code by changing the type from "radio" to "checkbox".

```
'<form action="response.lasso" method="post">\n'
inline(
    -show,
    -database='contacts',
    -table='people'
) => {^
    value_list('title') => {^
        '    <input type="radio" name="title" value="' + value_listItem + '" /> ' + value_listItem + '\n'
    ^}
^}
'<p><input type="submit" name="submit" value="Add Record">\n</form>\n'

// =>
```

```
// <form action="response.lasso" method="post">
//     <input type="radio" name="title" value="Mr." /> Mr.
//     <input type="radio" name="title" value="Mrs." /> Mrs.
//     <input type="radio" name="title" value="Ms." /> Ms.
//     <input type="radio" name="title" value="Dr." /> Dr.
// <p><input type="submit" name="submit" value="Add Record"></p>
// </form>
```

Part VIII

Extending Lasso

Lasso C API

47.1 LCAPI Overview

The *Lasso C/C++ Application Programming Interface* (LCAPI) lets you write C or C++ code to add new Lasso methods, types, or data source connectors to Lasso. Writing in LCAPI can offer speed and system performance advantages over LJAPI and custom Lasso libraries. However, modules must be compiled separately for Windows, OS X, and Linux.

This chapter provides a walkthrough for building and debugging an example LCAPI method. You can download the source code[62] for this and other examples online.

47.1.1 Requirements

In order to compile LCAPI methods, types, or data source connectors you need the following:

OS X

- Lasso Server installed on a supported OS X version

- Xcode

- The 10.5 SDK, which does not come with the newest development tools. See this link for unsupported help with installing older SDKs[63].

Linux

- Lasso Server installed on a supported Linux distribution

- The gcc C/C++ development libraries and executables

Windows

- Lasso Server installed on a supported Windows version

- Microsoft Visual C++ .NET

47.1.2 Quick Start

This section provides a walkthrough for building sample LCAPI method modules.

Build a sample LCAPI module in Windows

1. Download and expand the example code.

2. In the "MathFuncsTags" folder, double-click the `MathFuncsCAPI.sln` project file (you need Microsoft Visual C++ .NET in order to open it).

3. Choose *Build* → *Build Solution* to compile and make the `MathFuncsCAPI.DLL` module.

---

[62] http://lassoguide.com/_downloads/lcapi_examples.zip
[63] http://devernay.free.fr/hacks/xcodelegacy/

4. After building, a "Debug" folder will have been created inside your "MathFuncsCAPI" project folder. Open it and drag `MathFuncsCAPI.DLL` into "LassoModules" in a Lasso instance home directory.

5. Restart the Lasso instance.

6. New methods `example_math_abs`, `example_math_sin`, and `example_math_sqrt` are now part of your Lasso installation.

7. Drag the sample Lasso page called `MathFuncsCAPI.lasso` into the web server root.

8. View the `MathFuncsCAPI.lasso` page in a web browser to see the new Lasso methods in action.

Build a sample LCAPI module in OS X or Linux

1. Download and expand the example code.

2. Open a terminal window and change the working directory to the "MathFuncsTags" folder in the example code.

3. Build the sample project using the provided makefile by running **make**.

4. After building, a file named `MathFuncsCAPI.dylib` on OS X and `MathFuncsCAPI.so` on Linux will be in the current folder. Move that file into "LassoModules" in a Lasso instance home directory.

5. Restart the Lasso instance.

6. New methods `example_math_abs`, `example_math_sin`, and `example_math_sqrt` are now part of your Lasso installation.

7. Drag the sample Lasso page called `MathFuncsCAPI.lasso` into the web server root.

8. View the `MathFuncsCAPI.lasso` page in a web browser to see the new Lasso methods in action.

47.1.3 Debugging

You can set breakpoints in your LCAPI-compiled libraries and perform source-level debugging for your own code. In order to set this up, follow the example below. For this section, we will use the "MathFuncsCAPI" example.

Debug in Windows

1. Select *Debug → Processes…*.

2. In the "Processes" window, select each instance of "lassoserver.exe" and choose to *Attach*.

3. Close the "Processes" window and set a breakpoint in the `tagMathAbsFunc` function.

4. Use a web browser to access the sample `MathFuncsCAPI.lasso` file on the web server. Visual Studio will stop at the location that the breakpoint was placed.

Debug in OS X or Linux

1. The provided makefile compiles with the DEBUG options by default, so there is no need to recompile.

2. Find the process ID number of **lassoserver** so you can attach to it later with GNU Debugger:

```
$> ps -ax | grep lassoserver
2081 ?? 2:32.39 /usr/sbin/lassoserver -flisten lasso.fastcgi.sock
```

3. Start the GNU Debugger as the root user:

```
$> sudo gdb
```

Tip: For newer versions of OS X, use **lldb** instead of **gdb**.

4. From within GNU Debugger's command line, attach to the **lassoserver** process ID by entering the following (replacing <PROCESS ID> with the actual process ID):

```
attach <PROCESS ID>
```

5. Instruct GNU Debugger to break whenever the function **tagMathAbsFunc** is called by entering the following:

```
break tagMathAbsFunc
```

6. Use a web browser to access the sample `MathFuncsCAPI.lasso` file on the web server. GNU Debugger will break at the first line in **tagMathAbsFunc** when the **example_math_abs** method is called.

Tip: Type "help" in GNU Debugger for more information about using the GNU Debugger, or search for gdb tutorials on the web for more in-depth information.

47.1.4 Frequently Asked Questions

How do I install my custom module?

Once you've compiled your module, move it to the "LassoModules" directory for the instance you want it to run in or the "LassoModules" directory in the master Lasso home directory. You'll need to restart any running instances for them to pick up the new method/type/data source connector.

How do I return text from my custom module?

Use either **lasso_returnTagValueString** to return UTF-8 data, or **lasso_returnTagValueStringW** to return UTF-16 data. Character data in other encoding methods can be returned by first allocating a **string** type using **lasso_typeAllocStringConv** and then returning it using **lasso_returnTagValue**.

How do I return binary data from my custom method?

Use **lasso_returnTagValueBytes** to return binary data.

How do I prevent Lasso from automatically encoding text returned from my custom method?

Make sure that your method is registered with the **flag_noDefaultEncoding** flag. This flag is specified when you call **lasso_registerTagModule** at startup.

How do I debug my custom method?

You can set breakpoints in your code and attach your debugger to **lassoserver**. See the section *Debugging* above.

How do I get parameters that were passed into my method?

Most of the parameters passed into your custom method can be retrieved using the **lasso_getTagParam** and **lasso_findTagParam** parameter info functions. The **lasso_getTagParam** function retrieves parameters by index and **lasso_findTagParam** retrieves them by name. All parameters retrieved using these functions will be returned as strings. To access the parameters as Lasso type instances, use **lasso_getTagParam2** and **lasso_findTagParam2**.

How do I get the value of unnamed parameters passed into my method?

While there is no direct way to get unnamed parameters (how do you know what name to ask for?), you can enumerate through all the parameters by index, and then pick out the ones that do not have names. If, after retrieving a parameter, you discover that its data member is an empty string, then that means it is an unnamed parameter, and you can get its value from the name member. An example of this is in the *LCAPI method tutorial*.

What's an auto_lasso_value_t and how do I use it?

It's a data structure that contains both a name and a value (a name/value pair). Many LCAPI APIs fill in this structure for you, and you can access the name and data members directly as null-terminated C strings.

What is a lasso_type_t and how do I use it?

A **lasso_type_t** represents an instance of a Lasso type. Any Lasso type can be represented by a **lasso_type_t**, including strings, integers, or custom types. LCAPI provides many functions for allocating or manipulating **lasso_type_t** objects.

All **lasso_type_t** objects encountered inside an LCAPI method will be automatically garbage-collected after the function returns. Therefore, a **lasso_type_t** object should not be saved unless it is freed from the garbage collector using **lasso_typeDetach**.

How do I access variables from the Lasso page I'm in?

You may need to get or even create Lasso variables (the same variables that a Lasso programmer makes when using the **var(dozen) = 12** variable syntax in a Lasso page) from within your LCAPI module. You can retrieve a thread variable, as long as it has already been assigned before your custom method is executed, by calling **lasso_getVariable** with the variable's name. Using this method, one could directly set the "__html_reply__" variable.

How do I return fatal and non-fatal error codes?

It is very important that your method return an error code of **osErrNoErr(0)** if nothing fatal happened. An example of a fatal error would be a missing required parameter. If you encounter a fatal error, return a non-zero result code from your function; at that point Lasso will stop processing the page and display an error page.

How do I write code that will compile easily across multiple operating systems?

While we cannot provide a complete cross-platform programming tutorial here, we can at least provide some guidance. The simplest way to make sure code compiles across platforms is to make sure you use standard library functions (from **stdio.h** and **stdlib.h**) as much as possible: functions like **strcpy()**, **malloc()**, and **strcmp()** are always available on all platforms. Also note that *nix platforms are case-sensitive, so when you **#include** files, just make sure you keep the case the same as the file on disk. Finally, stay away from platform-specific functions, such as Windows APIs that are most often not available on *nix platforms. Take a look at the *nix makefiles that are provided with the sample projects: notice the same source code is used for Windows, and all source files are saved with DOS-style CR/LF line breaks so as not to confuse the Windows compilers. As a last resort, you can use **#ifdef** to show/hide platform-specific portions of source code.

47.2 Creating Lasso Methods

When Lasso first starts up, it looks for module files (Windows DLLs, OS X DYLIBs, or Linux SOs) in its "LassoModules" directory. As it encounters each module, it executes that module's **registerLassoModule** function once and only once. LCAPI developers must write code to register each new custom method (or type or data source connector) in this **registerLassoModule** function. The following example function is required in every LCAPI module. It gets called once when Lasso starts up:

```
void registerLassoModule() {
    lasso_registerTagModule( "CAPITester", "testtag", myTagFunc,
        REG_FLAGS_TAG_DEFAULT, "simple test LCAPI tag" );
}
```

The preceding example registers a C function named **myTagFunc** to execute whenever the Lasso **CAPITester_testtag** method call is encountered in Lasso code. The first parameter **CAPITester** is the namespace in which **testtag** will be placed.

Once the method function is registered, Lasso will call it at appropriate times while parsing and executing Lasso code. The custom method functions will not be called if none of the custom methods are encountered while executing a script. When Lasso encounters one of your custom methods, it will be called with two parameters: an opaque data structure called a **token**, and an integer **action** which is currently unused. LCAPI provides many function calls that can be used to get information about the environment, variables, parameters, etc., when provided with a token.

The passed-in token can also be used to acquire any parameters and to return a value from your custom method function.

47.2.1 Basic Custom Method Function

Use the following C++ code:

```
osError myTagFunc(lasso_request_t token, tag_action_t action)
{
    const char * retString = "Hello, World!";
    return lasso_returnTagValueString(token, retString, strlen(retString));
}
```

Below is the Lasso code needed to get the custom method to execute:

```
<p>Here is the custom tag:</p>
[CAPITester_testtag]
<!-- This should display "Hello, World" when this page executes -->
```

This will produce:

```
Here is the custom tag:
Hello, World
```

47.2.2 Custom Method Tutorial

This section provides a walkthrough of building an example method to show how LCAPI features are used. This code can be found in the "SampleMethod" folder in the LCAPI examples[64], which can be downloaded online.

The method will simply display its parameters, and it will look like the example below when called in Lasso code:

```
sample_method('some text here', -option1='named param', -option2=12.5)
```

Notice the method takes one unnamed parameter, one string keyword parameter **-option1**, and one numeric keyword parameter **-option2**. For keyword parameters, Lasso does not care about the order in which you pass them, so plan to make this method as flexible as possible by not assuming anything about their order. The following variations should work exactly the same:

```
sample_method('some text here', -option1='named param', -option2=12.5)
sample_method('some text here', -option2=12.5, -option1='named param')
```

LCAPI Method Module Code

Below is the C++ code for the custom method:

```
void registerLassoModule()
{
    lasso_registerTagModule( "sample", "method", myTagFunc,
        REG_FLAGS_TAG_DEFAULT, "sample test" );
}

osError myTagFunc( lasso_request_t token, tag_action_t action )
{
    std::basic_string<char> retValue;
    lasso_type_t opt2 = NULL;
    auto_lasso_value_t v;
    INITVAL(&v);

    if( lasso_findTagParam(token, "-option1", &v) == osErrNoErr ) {
        retValue.append("The value of -option1 is ");
```

---

[64] http://lassoguide.com/_downloads/lcapi_examples.zip

```
            retValue.append(v.data);
        }

        if( lasso_findTagParam2(token, "-option2", &opt2) == osErrNoErr ) {
            double tempValue;
            char tempValueFmtd[128];

            lasso_typeGetDecimal(token, opt2, &tempValue);
            sprintf(tempValueFmtd, "%.15lg", tempValue);

            retValue.append(" The value of -option2 is ");
            retValue.append(tempValueFmtd);
        }

        int count = 0;
        lasso_getTagParamCount(token, &count);

        for( int i = 0; i < count; ++i )
        {
            lasso_getTagParam(token, i, &v);
            if( v.name == v.data ) {
                retValue.append(" The value of unnamed param is ");
                retValue.append(v.data);
            }
        }

        return lasso_returnTagValueString(token, retValue.c_str(), (int)retValue.length());
    }
```

Method Module Code Walkthrough

This section provides a step-by-step walkthrough of the code for the custom method module.

1. First, the new method is registered in the required **registerLassoModule** export function:

    ```
    void registerLassoModule()
    {
        lasso_registerTagModule( "sample", "method", myTagFunc,
            REG_FLAGS_TAG_DEFAULT, "sample test" );
    }
    ```

2. Implement **myTagFunc**, which gets called when **sample_method** is encountered. All method functions have this prototype. When the method function is called, it's passed an opaque **token** data structure.

    ```
    osError myTagFunc( lasso_request_t token, tag_action_t action )
    {
    ```

    The remainder of the code in the walkthrough includes the implementation for the **myTagFunc** function.

3. Allocate a string to be this method's return value:

    ```
    std::basic_string<char> retValue;
    ```

4. The **lasso_type_t** variable named "opt2" and the **auto_lasso_value_t** variable named "v" will be temporary variables for holding parameter values. Start off by initializing them:

```
lasso_type_t opt2 = NULL;
auto_lasso_value_t v;
INITVAL(&v);
```

5. Call **lasso_findTagParam** in order to get the value of the **-option1** parameter. If it is found (no error while finding the named parameter), append some information about it to our return value string:

```
if( lasso_findTagParam(token, "-option1", &v) == osErrNoErr ) {
    retValue.append("The value of -option1 is ");
    retValue.append(v.data);
}
```

6. Look for the other named parameter **-option2** and store its value into variable "opt2". Because **-option2** should be a decimal value, use **lasso_findTagParam2**, which will preserve the original data type of the value as opposed to converting it into a string like **lasso_findTagParam** will.

```
if( lasso_findTagParam2(token, "-option2", &opt2) == osErrNoErr ) {
```

7. Declare a temporary floating-point (double) value to hold the number passed in and then declare a temporary string to hold the converted number for display. Get the value of "opt2" as a decimal then print it to the "tempValueFmtd" variable.

```
double tempValue;
char tempValueFmtd[128];

lasso_typeGetDecimal(token, opt2, &tempValue);
sprintf(tempValueFmtd, "%.15lg", tempValue);
```

8. Append the parameter's information to the return string:

```
retValue.append(" The value of -option2 is ");
retValue.append(tempValueFmtd);
```

9. Now, we're going to look for the unnamed parameter. Because there's no way to ask for unnamed parameters, we're going to enumerate through all the parameters looking for one without a name. The integer "count" will hold the number of parameters found. Use **lasso_getTagParamCount** to find out how many parameters were passed into our method. The variable "count" now contains the number "3", if we were indeed passed three parameters.

```
int count = 0;
lasso_getTagParamCount(token, &count);

for( int i = 0; i < count; ++i )
{
```

10. Use **lasso_getTagParam** to retrieve a parameter by its index. If you design methods that require parameters to be in a particular order, then use this function to retrieve parameters by index, starting at index 0. If the parameter is unnamed, that means it's the one needed. Note that if the user passes in more than one unnamed parameter, this loop will find all of them, and will ignore any named parameters. (A parameter is unnamed if both the name and data of the struct point to the same value.)

```
lasso_getTagParam(token, i, &v);
if( v.name == v.data ) {
```

11. Again, append a descriptive line of text about the unnamed parameter and its value.

```
if( v.name == v.data ) {
    retValue.append(" The value of unnamed param is ");
```

```
        retValue.append(v.data);
    }
```

12. Returning an error code is very important. If you return a non-zero error code, then the interpreter will throw an exception indicating that this method failed fatally, causing Lasso's standard page error routines to display an error message. In our example, **lasso_returnTagValueString** will return an error if it has a problem setting the return value.

```
    return lasso_returnTagValueString(token, retValue.c_str(), (int)retValue.length());
```

47.3 Creating Lasso Types

Creating a new Lasso type in LCAPI is similar to creating a custom method. When Lasso Server starts up, it scans the "Lasso-Modules" directory for module files (Windows DLLs, OS X DYLIBs, or Linux SOs). As it encounters each module, it executes the **registerLassoModule** function for that module. The developer registers the LCAPI types or methods implemented by the module inside this function. Registering type initializers differs from registering normal methods in that the third parameter in **lasso_registerTagModule** is the value "REG_FLAGS_TYPE_DEFAULT":

```
void registerLassoModule()
{
    lasso_registerTagModule( "test", "type", myTypeInitFunc,
        REG_FLAGS_TYPE_DEFAULT, "simple test LCAPI type" );
}
```

The prototype of an LCAPI type initializer is the same as a regular LCAPI custom method function. Lasso will call the type initializer each time a new instance of the type is created:

```
osError myTypeInitFunc( lasso_request_t token, tag_action_t action );
```

When the type initializer function is called, a new instance of the type is created using **lasso_typeAllocCustom**. This new instance will be created without data members or member methods:

```
osError myTypeInitFunc( lasso_request_t token, tag_action_t action );
{
    lasso_type_t theNewInstance = NULL;
    lasso_typeAllocCustom( token, &theNewInstance, "test_type" );
```

Once the type is created, new data members and member methods can be added to it using **lasso_typeAddMember**. Data members can be of any type and should be allocated using any of the LCAPI type allocation calls. Member methods are allocated using **lasso_typeAllocTag**. LCAPI member method functions are implemented just like any other LCAPI method. In the example below, **myTagMemberFunction** is a function with the standard LCAPI prototype:

```
const char * kStringData = "This is a string member.";
lasso_type_t stringMember = NULL;
lasso_typeAllocString( token, &stringMember, kStringData, strlen(kStringData) );
lasso_typeAddDataMember( token, theNewInstance, "member1", stringMember );

lasso_type_t tagMember = NULL;
lasso_typeAllocTag( token, &tagMember, myTagMemberFunction );
lasso_typeAddMember( token, theNewInstance, "member2", tagMember );
```

The final step in creating a new LCAPI type instance is to return the new type to Lasso as the initializer's return value. After the type is returned, Lasso will complete the creation of the type by instantiating the new type's parent types:

```
    lasso_returnTagValue( token, theNewInstance );
    return osErrNoErr;
}
```

47.3.1 Basic Custom Type

This tutorial walks through the main points of creating a custom type using LCAPI. The resulting type is an **example_file** type, and the ability to open, close, read, and write to the file are implemented via the following member methods: **example_file->open**, **example_file->close**, **example_file->read**, **example_file->write**.

The example project is the "CAPIFile" project in the LCAPI examples[65] found online. Due to the length of the code in that file, the entire code is not reproduced here. Instead, this section provides a conceptual overview of the **example_file** type and describes the basic LCAPI functions used to implement it.

1. The first step in creating a custom type is to register the type's initializer. Type initializers are registered in the same way that regular method functions are registered. The only difference being that "REG_FLAGS_TYPE_DEFAULT" should be passed for the fourth (flags) parameter.

   This concept is illustrated in lines 247–282 of the **CAPIFile.cpp** file:

   ```
   void registerLassoModule()
   {
      ...
      lasso_registerTagModule("", kFileTypeName, file_init,
         REG_FLAGS_TYPE_DEFAULT, "Initializer for the file type.");
   }
   ```

2. The registered type initializer will be called when the module is loaded. In the above case, the LCAPI function **file_init** was registered as being the initializer. The prototype for **file_init** should look like any other LCAPI function, as shown on line 285 of the **CAPIFile.cpp** file:

   ```
   osError file_init(lasso_request_t token, tag_action_t action)
   ```

3. The **file_init** function will now be called whenever the module is loaded. Within the type initializer, the type's member methods are added. Each member method is implemented by its own LCAPI function. However, before members can be added, the new blank type must be created using **lasso_typeAllocCustom**.

   You can only use **lasso_typeAllocCustom** within a properly registered type initializer. The value it produces should always be the return value of the method as set by the **lasso_returnTagValue** function. See lines 289–290 of the **CAPIFile.cpp** file:

   ```
   lasso_type_t file;
   lasso_typeAllocCustom(token, &file, kFileTypeName);
   ```

4. Once the blank type has been created, members can be added to it. LCAPI types often need to store pointers to allocated structures or memory. LCAPI provides a means to accomplish this by using the **lasso_setPtrMember** and **lasso_getPtrMember** functions. These functions allow the developer to store a pointer with a specific name. The pointer is stored as a regular integer data member. The names of all pointer members should begin with an underscore. Naming a pointer as such will indicate to Lasso that it should not be copied when a copy is made of the type instance. In the initializer function, add the integer data member as seen on lines 293–295:

   ```
   lasso_type_t i;
   lasso_typeAllocInteger(token, &i, 0);
   lasso_typeAddDataMember(token, file, kPrivateMember, i);
   ```

---

[65] http://lassoguide.com/_downloads/lcapi_examples.zip

This LCAPI **example_file** type stores its private data in a structure named **file_desc_t**. The actual call to **lasso_setPtrMember** is in the method's **onCreate** method as shown on lines 344–345 of the **CAPIFile.cpp** file:

```
file_desc_t * desc = new file_desc_t;
lasso_setPtrMember(token, self, kPrivateMember, desc, &cleanUp);
```

5. Member methods for **open**, **close**, **read**, and **write** could be written like this:

```
lasso_type_t mem;
lasso_typeAllocTag(token, &mem, file_open);
lasso_typeAddMember(token, file, "open", mem);

lasso_typeAllocTag(token, &mem, file_close);
lasso_typeAddMember(token, file, "close", mem);

lasso_typeAllocTag(token, &mem, file_read);
lasso_typeAddMember(token, file, "read", mem);

lasso_typeAllocTag(token, &mem, file_write);
lasso_typeAddMember(token, file, "write", mem);
```

But to avoid the repetitive nature of this, the **CAPIFile.cpp** file defines a macro named **ADD_TAG** to do the work as seen on lines 300–309:

```
#define ADD_TAG(NAME, FUNC) {
    lasso_type_t mem;\
    lasso_typeAllocTag(token, &mem, FUNC);\
    lasso_typeAddMember(token, file, NAME, mem);\
}

// Add the type's member tags
ADD_TAG(kMemOpen, file_open);
ADD_TAG(kMemClose, file_close);
ADD_TAG(kMemRead, file_read);
ADD_TAG(kMemWrite, file_write);
```

6. At this point, the return value should be set. Keep in mind that the new **example_file** type is completely blank except for the members that were added above. No inherited members are available at this point. Inherited members are only added after the LCAPI type initializer returns. Line 324 of the **CAPIFile.cpp** file sets the return value:

```
lasso_returnTagValue(token, file);
```

7. There were no errors in the type initialization process, so return a "no error" code to Lasso, completing the type's initialization. See line 325 of the **CAPIFile.cpp** file:

```
return osErrNoErr;
```

Note: For brevity, this example will not cover accepting parameters in the type's **onCreate** method. The full "CAPIFile" project illustrates accepting parameters in the **onCreate** member method to open the file under various read and write permissions.

---

8. The new file type has now been initialized and made available to the caller in the script. The first member method of the file type is **example_file->open**, which is implemented as the LCAPI function **file_open** beginning on line 385 of the **CAPIFile.cpp** file:

```
osError file_open(lasso_request_t token, tag_action_t action)
{
```

9. The first step in implementing a member method is to acquire the "self" instance. The "self" is the instance upon which the member call was made. This is illustrated on lines 387–390 of the **CAPIFile.cpp** file:

```
lasso_type_t self = NULL;
lasso_getTagSelf(token, &self);
if(!self)
    return osErrInvalidParameter;
```

10. Once the "self" is successfully acquired and is not "null", the rest of the member method can proceed. This member method accepts one parameter, which is the path to the file that will be opened. Since the path is a string value, it can be acquired using **lasso_getTagParam**. If the path parameter was not passed to the open member method, an error should be returned and indicated to the user. All of this can be seen on lines 400–418 of the **CAPIFile.cpp** file:

```
// See what parameters we are being initialized with
int count;
lasso_getTagParamCount(token, &count);

if( count < 2 )
{
    lasso_setResultMessage(token, "file->open requires at least a file path and open mode.");
    return osErrInvalidParameter;
}

if( count > 0 ) // We are given *at the least* a path
{
    // First param is going to be a string, so use the LCAPI call to get it
    auto_lasso_value_t pathParam;
    pathParam.name = "";
    lasso_getTagParam(token, 0, &pathParam);

    desc->fPath = pathParam.name;
}
```

11. Once the path is properly converted, the actual file can be opened using the file system calls supplied by the operating system. This concept is illustrated on line 225 of the **CAPIFile.cpp** file:

```
FILE * f = fopen(xformPath, openMode);
```

12. The **FILE** pointer can now be retrieved using the **lasso_typeGetCustomPtr** LCAPI function. No error has occurred while opening the file, so complete the function call and return "no error". See line 449 of the **CAPIFile.cpp** file:

```
return osErrNoErr;
```

13. The remaining method functions are implemented in a similar manner. Study the CAPIFile example for a more in-depth and complete example of how to properly construct custom Lasso types in LCAPI.

47.4 Creating Lasso Data Sources

When Lasso Server starts up, it looks for module files (Windows DLLs, OS X DYLIBs, or Linux SOs) in the "LassoModules" directory. As Lasso encounters each module, it executes the module's **registerLassoModule** function once and only once. It is your job as an LCAPI developer to write code to register each of your new data source function entry points in this **registerLassoModule** function. Custom methods, types, and data sources may be registered at the same time, and the code for

them can reside in the same module. The only difference between registering a data source and a custom method is whether you call **lasso_registerTagModule** or **lasso_registerDSModule**.

Data sources are a bit more complex than custom methods because Lasso calls them with many different actions during the course of various database operations. Whereas a custom method only needs to know how to format itself, a data source needs to enumerate its tables, search through records, add new records, delete records, etc. Even so, this added complexity is easily handled with a single **switch()** statement, as you will see in the following tutorial.

47.4.1 Data Source Connectors and Lasso Server Admin

Once a custom data source connector module is registered by Lasso Server, it will appear in the "Datasources" section of Lasso Server Admin. If a connector appears here, then it has been installed correctly.

The administrator adds the data source connection information to the "Hosts" form, which sets the parameters used by Lasso to connect to the data source via the connector. The information is stored in the site's **database_registry** SQLite database, where the connector can retrieve and use the data via function calls.

The "Hosts" information includes the following:

Name
:   The connection URL string used to connect to a data source. This is typically the IP address or hostname of the machine hosting the data source.

Port
:   The TCP/IP port number for the data source.

Enabled
:   Allows administrators to enable or disable the connection to the data source.

Username
:   The username Lasso uses to authenticate to the data source.

Password
:   The password for the username Lasso uses to authenticate to the data source.

These values are passed to the data source via the **lasso_getDataHost** function, which is described later in this chapter:

```
LCAPICALL osError lasso_getDataHost( lasso_request_t token,
    auto_lasso_value_t * host, auto_lasso_value_t * usernamepassword );
```

47.4.2 Basic Data Source Connector

This section provides a walkthrough of an example data source to show how some of the LCAPI features are used. This code can be found in the "SampleConnector" example project which can be downloaded with the other LCAPI examples[66] online.

This example data source simply displays some simple text as each action is called from a Lasso **inline**. It is not an effective or useful data source; it's meant to just provide an overview of what functions must be implemented. The sample data source will simulate a data source that has two databases, an "Accounting" database and a "Customers" database. Each of those databases, in turn, will report that it has a few tables within it. For a more complete example of a data source that is useful, look at the SQLiteDS source code[67] in the Lasso source code repository.

LCAPI Data Source Connector Code

Below is the code for the Sample Data Source Connector:

---

[66] http://lassoguide.com/_downloads/lcapi_examples.zip
[67] http://source.lassosoft.com/svn/lasso/lasso9_source/trunk/SQLiteDS/

```
void registerLassoModule()
{
    lasso_registerDSModule( "SampleDSConnector", sampleds_func, 0 );
    lasso_log(LOG_LEVEL_ALWAYS, "Loading Sample Data Source Connector");
}


osError sampleds_func
    ( lasso_request_t token, datasource_action_t action, const auto_lasso_value_t * param )
{
    osError err = osErrNoErr;
    auto_lasso_value_t v1, v2, notused;
    bool boolnotused = false;
    const char * ret;
    switch( action )
    {
        case datasourceInit:
            break;
        case datasourceTerm:
            break;
        case datasourceCloseConnection: // Connections only get closed through here
            // Here's where to gracefully close the connection
            break;
        case datasourceTickle:
            //
            break;
        case datasourceNames:
            // Database Names
            lasso_addDataSourceResult(token, "Accounting");
            lasso_addDataSourceResult(token, "Customers");
            break;
        case datasourceTableNames:
            if( strcmp(param->data, "Accounting") == 0 ) {
                lasso_addDataSourceResultUTF8(token, "Payroll");
                lasso_addDataSourceResultUTF8(token, "Payables");
                lasso_addDataSourceResultUTF8(token, "Receivables");
            }
            if( strcmp(param->data, "Customers") == 0 ) {
                lasso_addDataSourceResultUTF8(token, "ContactInfo");
                lasso_addDataSourceResultUTF8(token, "ItemsPurchased");
            }
            break;
        case datasourceSearch:
        case datasourceFindAll:
            lasso_getDataSourceName(token, &v1, &boolnotused, &notused);
            lasso_getTableName(token, &v2);

            if( strcmp(v1.data, "Accounting") == 0 ) {
                int count, i;
                lasso_getInputColumnCount(token, &count);
                for( i=0; i < count; i++) {
                    auto_lasso_value_t columnItem;
                    lasso_getInputColumn(token, i, &columnItem);
                }
                if( strcmp(v2.data, "Payroll") == 0 ) {
                    const char ** values = new const char*[3];
                    unsigned long * sizes = new unsigned long[3];
                    values[0] = "Samuel Goldwyn";
                    values[1] = "1955-03-27";
```

```
            values[2] = "15000.00";
            sizes[0] = 14;
            sizes[1] = 10;
            sizes[2] =  8;

            lasso_addColumnInfo(token, "Employee",  true, lpTypeString,   kProtectionNone);
            lasso_addColumnInfo(token, "StartDate", true, lpTypeDateTime, kProtectionNone);
            lasso_addColumnInfo(token, "Wages",     true, lpTypeDecimal,  kProtectionNone);

            lasso_addResultRow(token, values, sizes, 3);
            lasso_setNumRowsFound(token, 1);

            delete [] sizes;
            delete [] values;
         }
      }
      if( strcmp(v1.data, "Customers") == 0 ) {
      }
      break;

   case datasourceAdd:
      ret = "datasourceAdd was called to append a record<br />";
      lasso_returnTagValueString(token, ret, (int)strlen(ret));

   case datasourceUpdate:
      ret = "datasourceUpdate was called to replace a record<br />";
      lasso_returnTagValueString(token, ret, (int)strlen(ret));

   case datasourceDelete:
      ret = "datasourceDelete was called to remove a record<br />";
      lasso_returnTagValueString(token, ret, (int)strlen(ret));

   case datasourceInfo:
      ret = "datasourceInfo was called<br />";
      lasso_returnTagValueString(token, ret, (int)strlen(ret));

   case datasourcePrepareSQL:
      ret = "datasourcePrepareSQL was called<br />";
      lasso_returnTagValueString(token, ret, (int)strlen(ret));

   case datasourceUnprepareSQL:
      ret = "datasourceUnprepareSQL was called<br />";
      lasso_returnTagValueString(token, ret, (int)strlen(ret));

   case datasourceExecSQL:
      ret = "datasourceExecSQL was called<br />";
      lasso_returnTagValueString(token, ret, (int)strlen(ret));

   default:
      break;
   }

   return err;
}
```

Data Source Connector Walkthrough

This section provides a step-by-step walkthrough of the code for the custom data source connector.

1. Register the new data source in the **registerLassoModule** function:

```
void registerLassoModule()
{
    lasso_registerDSModule( "SampleDSConnector", sampleds_func, 0 );
    lasso_log(LOG_LEVEL_ALWAYS, "Loading Sample Data Source Connector");
}
```

2. Implement the **sampleds_func** function which gets called when any database operations for this data source are encountered:

```
osError sampleds_func
    ( lasso_request_t token, datasource_action_t action, const auto_lasso_value_t * param )
```

All data source functions have this prototype. When your data source function is called, it's passed an opaque **token** data structure, an integer **action** telling it what it should do, and an optional parameter that sometimes contains extra information (like a database name) needed by the action being requested at the time.

3. Set a default error return value to indicate no error. Returning a non-zero value will cause Lasso to report a fatal error and stop processing code. We are also declaring a few temporary variables to be used later to retrieve values such as database names and table names:

```
osError err = osErrNoErr;
auto_lasso_value_t v1, v2, notused;
bool boolnotused = false;
const char * ret;
```

4. This function is called with various actions passed to it as Lasso translates the **inline** requests to one of many actions. The **switch** statement is used with various enumerated values to determine the requested action:

```
switch( action )
{
```

5. The **datasourceInit** action is called once when Lasso Server starts up. This gives us a chance to initialize any communications with our database back-end, and do any initial setup if needed.

   The **datasourceTerm** action is called once when Lasso Server shuts down. This allows for any graceful cleanup that may be necessary for your data source.

   The **datasourceCloseConnection** action is called to close the connection to a data source.

   Because this data source is so simple, it needs no special initialization, shutdown code, or close connection code:

```
case datasourceInit:
    break;
case datasourceTerm:
    break;
case datasourceCloseConnection: // Connections only get closed through here
    // Here's where to gracefully close the connection
    break;
```

6. The **datasourceNames** action is called whenever Lasso needs to get a list of databases that your data source provides access to. The developer must write code that discovers the list of all databases your data source host "knows about" and call **lasso_addDataSourceResult** once for each found database, passing the name of the database. If the data source has five databases, then you would call **lasso_addDataSourceResult** five times. In our example, we have two databases:

```
case datasourceNames:
    // Database Names
    lasso_addDataSourceResult(token, "Accounting");
    lasso_addDataSourceResult(token, "Customers");
    break;
```

7. Lasso will also need to know about all the tables each of the databases in your data source knows about, and for this it calls the function with the **datasourceTableNames** action, passing the database name in the **param->data** value. In our example, we are adding three tables to the "Accounting" database and two to "Customers":

```
case datasourceTableNames:
    if( strcmp(param->data, "Accounting") == 0 ) {
        lasso_addDataSourceResultUTF8(token, "Payroll");
        lasso_addDataSourceResultUTF8(token, "Payables");
        lasso_addDataSourceResultUTF8(token, "Receivables");
    }
    if( strcmp(param->data, "Customers") == 0 ) {
        lasso_addDataSourceResultUTF8(token, "ContactInfo");
        lasso_addDataSourceResultUTF8(token, "ItemsPurchased");
    }
    break;
```

8. The **datasourceSearch** and **datasourceFindAll** actions are used to search a data source. All pertinent information (database and table names, search arguments, sort arguments, etc.) can be retrieved, and a search can be performed by calling various LCAPI functions such as **lasso_getDataSourceName** and **lasso_getTableName** to get the name of the database and table, respectively:

```
case datasourceSearch:
case datasourceFindAll:
    lasso_getDataSourceName(token, &v1, &boolnotused, &notused);
    lasso_getTableName(token, &v2);
```

9. In our example, only the "Payroll" table in the "Accounting" database has any data in it, so we have a conditional to check to see if the "Accounting" database was specified. We then use **lasso_getInputColumnCount** to get the number of search fields passed to the **inline**. We have a **for** loop to retrieve the name/value text for each search parameter. For example, calling the following Lasso code:

```
inline(-database='Accounting', -table='Payroll', 'Employee'='Fred', 'Wages'='15000')
```

will fill the "columnItem" variable with the values "Employee, Fred" the first time through the loop, and "Wages, 15000" the second time through the loop:

```
if( strcmp(v1.data, "Accounting") == 0 ) {
    int count, i;
    lasso_getInputColumnCount(token, &count);
    for( i=0; i < count; i++) {
        auto_lasso_value_t columnItem;
        lasso_getInputColumn(token, i, &columnItem);
    }
```

10. Next, set a conditional statement to ask if the "Payroll" table is being searched. If so, we'll set up some fake hard-coded data in the next few lines of code. Declare an array of strings that represent the three fields we will return for this search. Declare an array of field sizes to match the lengths of the strings created on the previous line.

The **lasso_addColumnInfo** function tells Lasso the column name and data type for a column. Call it once for each column and then call **lasso_addResultRow** with the values and their sizes to add a row to the result. Finally, the number of found rows must be specified using **lasso_setNumRowsFound**:

```
if( strcmp(v2.data, "Payroll") == 0 ) {
    const char ** values = new const char*[3];
    unsigned long * sizes = new unsigned long[3];
    values[0] = "Samuel Goldwyn";
    values[1] = "1955-03-27";
    values[2] = "15000.00";
    sizes[0] = 14;
    sizes[1] = 10;
    sizes[2] =  8;

    lasso_addColumnInfo(token, "Employee",  true, lpTypeString,   kProtectionNone);
    lasso_addColumnInfo(token, "StartDate", true, lpTypeDateTime, kProtectionNone);
    lasso_addColumnInfo(token, "Wages",     true, lpTypeDecimal,  kProtectionNone);

    lasso_addResultRow(token, values, sizes, 3);
    lasso_setNumRowsFound(token, 1);

    delete [] sizes;
    delete [] values;
}
```

11. The rest of the actions simply return the fact that they had been called. In a real data source connector, you would add code for those actions to add, update, delete, and query data from the data source.

47.5 C/C++ Reference for LCAPI

47.5.1 LassoCAPI.h

LassoCAPI Main Header File.

This file contains all of the available LCAPI defines and functions.

Tag Registration Flags

The following flags may be OR'd together in various combinations and passed to lasso_registerTagModule or lasso_registerTagModuleW as the *flags* parameter to control how the tag behaves when it is called.

const int **flag_typeInitializer**
> The tag is to be treated as an initializer for a custom type.

const int **flag_typeSubstitutionTag**
> The tag is to be treated as a "regular" tag.

> This is the default behavior so this flag is not required.

const int **flag_typeAsync**
> The tag is to be run in its own thread.

> It will return no value to the caller.

const int **flag_typeContainerTag**
> The tag is a container tag.

> The tag must be called using the proper syntax or an error is generated. The result of executing any body statements can be retrieved using the lasso_childrenRun function.

const int **flag_typeInterstitial**
> Used internally.

const int **flag_typeSkipSecurityCheck**
> Bypass any security checks for the tag.

> Useful to avoid any performance hits associated with security checks. Should only be used when a tag performs only trival or completely secure operations.

const int **flag_INTERNALONLY**
> Used internally.

const int **flag_typeLoopingTag**
> The tag is a looping container tag.

> For any tag registered with this flag, Lasso will automatically keep track of the loop_count and increment it each time the tag calls lasso_childrenRun.

const int **flag_typeEval**
> Used internally. Not useful for LCAPI tag types.

const int **flag_noGlobalImport**
> The tag should not be automatically imported into the global namespace.

> All LCAPI tags, starting with LP8, which are registered with a namespace should specify this flag.

const int **flag_INTERNALONLY2**
> Used internally.

const int **flag_deprecated**
> Use of the tag is deprecated.

> Depending on the administrator's configuration, calling the tag will automatically output a warning to the logging system.

const int **flag_noDefaultEncoding**
> Never apply default HTML encoding to the tag's return value.

const int **flag_INTERNALONLY3**

const int **flag_prototype**
> Applies only to types registered with lasso_registerTypeModule.

> An instance of the type will be created immediately (onCreate will not be called). Any further calls to this type will result in a fast copy of the prototype being created.

const int **flag_atomic**

const int **flag_private**

const int **flag_nonBlockingCAPI**

const int **flag_User1**
> User defined flag.

> May be used with lasso_tagSetFlag, lasso_tagHasFlag, lasso_tagClearFlag OR lasso_typeSetFlag, lasso_typeHasFlag, lasso_typeClearFlag

const int **flag_User2**
> User defined flag.

> May be used with lasso_tagSetFlag, lasso_tagHasFlag, lasso_tagClearFlag OR lasso_typeSetFlag, lasso_typeHasFlag, lasso_typeClearFlag

const int **flag_User3**
> User defined flag.
>
> May be used with lasso_tagSetFlag, lasso_tagHasFlag, lasso_tagClearFlag OR lasso_typeSetFlag, lasso_typeHasFlag, lasso_typeClearFlag

const int **flag_User4**
> User defined flag.
>
> May be used with lasso_tagSetFlag, lasso_tagHasFlag, lasso_tagClearFlag OR lasso_typeSetFlag, lasso_typeHasFlag, lasso_typeClearFlag

const int **REG_FLAGS_TAG_DEFAULT**
> Recommended default registration flags.
>
> For a normal tag.

const int **REG_FLAGS_CONTAINER_DEFAULT**
> Recommended default registration flags.
>
> For a container tag.

const int **REG_FLAGS_LOOPING_DEFAULT**
> Recommended default registration flags.
>
> For a looping container tag.

const int **REG_FLAGS_TYPE_DEFAULT**
> Recommended default registration flags.
>
> For a type initializer tag.

Type Registration

These functions are called to register a new type.

LCAPICALL osError **lasso_registerTypeModule(**const char * *namespaceName*, const char * *tagName*, **lasso_tag_func** *func*, int *flags*, const char * *description*, const char * *superType***)**
> Registers a new type.
>
> **Return**
>> If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.
>
> **Parameters**
>> - **namespaceName** - The namespace in which the type will be registered.
>> - **tagName** - The name which the type can be called by.
>> - **func** - The user supplied callback for the type initializer.
>> - **flags** - Any special flags for the type. If flag_typeInitializer is ommited, it will be automatically added. Any conflicting flags such as flag_typeContainerTag or flag_typeLoopingTag will be ignored.
>> - **description** - A description for the type.
>> - **superType** - Optionally, the name of the new type's super type.

LCAPICALL osError **lasso_registerTypeModuleW(**const UChar * *namespaceName*, const UChar * *tagName*, **lasso_tag_func** *func*, int *flags*, const UChar * *description*, const UChar * *superType***)**

Datasource Module Registration

These functions are called to register a new datasource module.

LCAPICALL osError **`lasso_registerDSModule`**(const char * *moduleName*, **lasso_ds_func** *func*, int *flags*)
>   Registers a new datasource module.

>   **Return**
>>   If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

>   **Parameters**
>>   • **`moduleName`** - The name of the datasource module.
>>   • **`func`** - The User supplied callback for the datasource module.
>>   • **`flags`** - Any special flags for the datasource module.

LCAPICALL osError **`lasso_registerDSModuleW`**(const UChar * *moduleName*, **lasso_ds_func** *func*, int *flags*)

LCAPICALL osError **`lasso_registerDSModule2`**(const   char   * *moduleName*,   **lasso_ds_func** *func*,   int *flags*,   void * *userData*, void(**userDataDtor*)(void *))

LCAPICALL osError **`lasso_registerDSModule2W`**(const   UChar   * *moduleName*,   **lasso_ds_func** *func*,   int *flags*,   void * *userData*, void(**userDataDtor*)(void *))

Allocating Built-in Type Instances

The following functions allocate instances of specific built-in types. The lasso_request_t token may be null. If it is null, the allocated types are created as "detached" and must be manually freed using lasso_typeFree.

LCAPICALL osError **`lasso_typeAllocNull`**(lasso_request_t *token*, **lasso_type_t** * *outNull*)
>   Allocates a new instance of type null.

>   **Return**
>>   If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

>   **Parameters**
>>   • **`token`** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.
>>   • **`outNull`** - The resulting new type instance.

LCAPICALL osError **`lasso_typeAllocVoid`**(lasso_request_t *token*, **lasso_type_t** * *outVoid*)
>   Allocates a new instance of type void.

>   **Return**
>>   If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

>   **Parameters**
>>   • **`token`** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.
>>   • **`outVoid`** - The resulting new type instance.

LCAPICALL osError **`lasso_typeAllocString`**(lasso_request_t *token*,   **lasso_type_t** * *outString*,   const   char * *value*, int *length*)
>   Allocates a new instance of type string.

Return

If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **outString** - The resulting new type instance.

- **value** - The UTF-8 data from which to copy for the new string instance.

- **length** - The length of the UTF-8 data in characters.

LCAPICALL osError **lasso_typeAllocStringConv**(lasso_request_t *token*, lasso_type_t * *outString*, const char * *value*, int *length*, const char * *conv*)
Allocates a new instance of type string using the specified conversion method.

Return

If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **outString** - The resulting new type instance.

- **value** - The source data from which to copy for the new string instance.

- **length** - The length of the source data in bytes.

- **conv** - The encoding of the source data.

LCAPICALL osError **lasso_typeAllocStringW**(lasso_request_t *token*, lasso_type_t * *outString*, const UChar * *value*, int *length*)
Allocates a new instance of type string.

Return

If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **outString** - The resulting new type instance.

- **value** - The UTF-16 data from which to copy for the new string instance.

- **length** - The length of the UTF-16 data in characters.

LCAPICALL osError **lasso_typeAppendStringW**(lasso_request_t *token*, lasso_type_t *type*, const UChar * *val*, int *len*)

LCAPICALL osError **lasso_typeAllocInteger**(lasso_request_t *token*, lasso_type_t * *outInteger*, int64_t *value*)
Allocates a new instance of type integer.

Return

If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **outInteger** - The resulting new type instance.

- **value** - The integer value which the new type instance will hold.

LCAPICALL osError **lasso_typeAllocDecimal**(lasso_request_t *token*, lasso_type_t * *outDecimal*, double *value*)
Allocates a new instance of type decimal.

> **Return**
>> If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

> **Parameters**
>> - **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.
>>
>> - **outDecimal** - The resulting new type instance.
>>
>> - **value** - The double value which the new type instance will hold.

LCAPICALL osError **lasso_typeAllocDecimal2**(lasso_request_t *token*, lasso_type_t * *outDecimal*, double *value*, int *precision*)
Allocates a new instance of type decimal.

> **Return**
>> If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

> **Parameters**
>> - **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.
>>
>> - **outDecimal** - The resulting new type instance.
>>
>> - **value** - The double value which the new type instance will hold.
>>
>> - **precision** - The decimal precision that the new type instance will output to.

LCAPICALL osError **lasso_typeAllocPair**(lasso_request_t *token*, lasso_type_t * *outPair*, lasso_type_t *first*, lasso_type_t *second*)
Allocates a new instance of type pair.

> **Return**
>> If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

> **Parameters**
>> - **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.
>>
>> - **outPair** - The resulting new type instance.
>>
>> - **first** - The instance to use for the first member of the pair. A reference to the instance will be made.
>>
>> - **second** - The instance to use for the second member of the pair. A reference to the instance will be made.

LCAPICALL osError **lasso_typeAllocReference**(lasso_request_t *token*, lasso_type_t * *outRef*, lasso_type_t *referenced*)
Allocates a new hard reference to a type instance. The new instance will point to the original. This is a no-op under Lasso 9.

> **Return**
>> If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **outRef** - The resulting new type instance.

- **referenced** - The instance to be referenced.

LCAPICALL osError **lasso_typeAllocTag**(lasso_request_t *token*,   lasso_type_t   * *outTag*,   lasso_tag_func *nativeTagFunction*)

Allocates a new instance of type tag.

Return

If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **outTag** - The resulting new type instance.

- **nativeTagFunction** - The LCAPI lasso_tag_func which will be called when the tag is used.

LCAPICALL osError **lasso_typeAllocTagFromSource**(lasso_request_t *token*, lasso_type_t * *outTag*, const char * *source*, int *length*)

Allocates a new instance of type tag from the given source text.

If the source text is UTF-8, it must contain a BOM or it will be treated as the default platform encoding.

Return

If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **outTag** - The resulting new type instance.

- **source** - The source text which will be compiled to become the body of the tag.

- **length** - The length of the source text in characters.

LCAPICALL osError **lasso_typeAllocArray**(lasso_request_t *token*, lasso_type_t * *outArray*, int *count*, lasso_type_t * *elements*)

Allocates a new instance of type array.

Return

If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **outArray** - The resulting new type instance.

- **count** - The number of new array elements to insert. The number of items in the *elements* parameter.

- **elements** - A pointer to an array of lasso_type_t which will become the elements of the array. Each item will be referenced.

LCAPICALL osError **lasso_typeAllocBoolean**(lasso_request_t *token*, lasso_type_t * *outBool*, bool *inValue*)
>   Allocates a new instance of type boolean.

>   **Return**
>   >   If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

>   **Parameters**
>   >   • **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.
>   >
>   >   • **outBool** - The resulting new type instance.
>   >
>   >   • **inValue** - The boolean value which the new type instance will hold.

Getting or Setting Values of Built-in Type Instances

The following functions get and set the data on a previously created built-in type instance. When getting a value, the source type instance will not be altered. When setting a value, the provided type instance is converted, if required. The lasso_request_t token may be null.

LCAPICALL osError **lasso_typeGetString**(lasso_request_t *token*, lasso_type_t *type*, auto_lasso_value_t * *val*)
>   Retrieves character data from a type instance.

>   If the type is a string instance, the data will be converted to UTF-8. If the type is a bytes instance, the data will be provided unaltered. For any other type, the result will be the same as converting the type into a string and returning the data as UTF-8.

>   **Return**
>   >   If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

>   **Parameters**
>   >   • **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.
>   >
>   >   • **type** - The type instance from which to retrieve the character data.
>   >
>   >   • **val** - A pointer to a *lasso_value_t* struct in which to store the result. The data will be stored in the *name* portion of the struct.

LCAPICALL osError **lasso_typeGetStringConv**(lasso_request_t *token*,   lasso_type_t *type*,   auto_lasso_value_t
>   * *val*, const char * *conv*)
>   Retrieves character data from a type instance using the specified conversion method. If the special conversion method of "BINARY" is used, and the source type is a string, the resulting data will be UTF-16 data. If the special conversion method of "BINARY" is used, and the source type is bytes, the resulting data will be provided unaltered. For any other type, if the special conversion method of "BINARY" is used, the result will be the same as converting the instance to a string and retrieving the UTF-16 data.

>   The remaining possible values for the conversion method are any of the character encoding methods supported by ICU or any of the converters stored in the '**external_converters**' global variable.

>   In all cases, the *nameSize* portion of the resulting value struct will be the number of bytes in the result data.

>   **Return**
>   >   If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

>   **Parameters**
>   >   • **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **type** - The type instance from which to retrieve the character data.

- **val** - A pointer to a *lasso_value_t* struct in which to store the result. The data will be stored in the *name* portion of the struct.

- **conv** - The encoding method to use when transforming the Unicode string data.

LCAPICALL osError **lasso_typeGetStringW**(lasso_request_t *token*,   lasso_type_t *type*,   auto_lasso_value_w_t
* val)
Retrieves Unicode character data from a type instance.

For any type other than string, the result will be the same as converting the type into a string and returning the data as UTF-16.

Return

If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **type** - The type instance from which to retrieve the character data.

- **val** - A pointer to a *lasso_value_w_t* struct in which to store the result. The data will be stored in the *name* portion of the struct.

LCAPICALL osError **lasso_typeGetInteger**(lasso_request_t *token*, lasso_type_t *type*, int64_t * *out*)
Retrieves the integer value from a type instance.

Return

If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **type** - The type instance from which to retieve the integer value.

- **out** - A pointer to the resulting 64-bit integer.

LCAPICALL osError **lasso_typeGetDecimal**(lasso_request_t *token*, lasso_type_t *type*, double * *out*)
Retrieves the decimal value from a type instance.

Return

If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **type** - The type instance from which to retieve the decimal value.

- **out** - A pointer to the resulting double.

LCAPICALL osError **lasso_typeGetBoolean**(lasso_request_t *token*, lasso_type_t *type*, bool * *out*)
Retrieves the boolean value from a type instance.

Return

If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **type** - The type instance from which to retieve the boolean value.

- **out** - A pointer to the resulting boolean.

LCAPICALL osError **lasso_typeSetString**(lasso_request_t *token*, lasso_type_t *type*, const char * *val*, int *len*)
Converts the type instance into a string and sets the value.

Return

If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **type** - The type instance to set.

- **val** - The source data in UTF-8 encoding.

- **len** - The length of the source UTF-8 data.

LCAPICALL osError **lasso_typeSetStringW**(lasso_request_t *token*, lasso_type_t *type*, const UChar * *val*, int *len*)
Converts the type instance into a string and sets the value.

Return

If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **type** - The type instance to set.

- **val** - The source data.

- **len** - The length of the source data.

LCAPICALL osError **lasso_arrayGetSize**(lasso_request_t *token*, lasso_type_t *array*, int * *len*)
Retrieves the size of the provided array instance.

Return

If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **array** - The array instance.

- **len** - A pointer in which to store the resulting size.

LCAPICALL osError **lasso_arrayGetElement**(lasso_request_t *token*, lasso_type_t *array*, int *index*, lasso_type_t
* *out*)
Retrieves the specified element of the provided array instance.

Return
> If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters
- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.
- **array** - The array instance.
- **index** - The zero based index at which to retreive.
- **out** - The pointer in which to store the result.

LCAPICALL osError **lasso_arraySetElement**(lasso_request_t *token*,   lasso_type_t *array*,   int *index*, lasso_type_t *elem*)
> Sets the specified element of the provided array instance.

Return
> If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters
- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.
- **array** - The array instance.
- **index** - The zero based index at which to set. If the index is greater than the array's current size or is less than zero, the new item is added to the end of the array.
- **elem** - The type instance which will be placed at the specified index. The instance will be referenced.

LCAPICALL osError **lasso_pairGetFirst**(lasso_request_t *token*, lasso_type_t *pr*, lasso_type_t * *out*)
> Retrieves the first member from the provided pair.

Return
> If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters
- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.
- **pr** - The pair instance.
- **out** - The pointer in which to store the result.

LCAPICALL osError **lasso_pairGetSecond**(lasso_request_t *token*, lasso_type_t *pr*, lasso_type_t * *out*)
> Retrieves the second member from the provided pair.

Return
> If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters
- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.
- **pr** - The pair instance.
- **out** - The pointer in which to store the result.

LCAPICALL osError **lasso_pairSetFirst**(lasso_request_t *token*, lasso_type_t *pr*, lasso_type_t *frst*)
Sets the first member of the provided pair.

Return
If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **pr** - The pair instance.

- **frst** - The instance to use as the first member. The instance will be referenced.

LCAPICALL osError **lasso_pairSetSecond**(lasso_request_t *token*, lasso_type_t *pr*, lasso_type_t *scnd*)
Sets the second member of the provided pair.

Return
If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **pr** - The pair instance.

- **scnd** - The instance to use as the second member. The instance will be referenced.

Datasource Module Functions

LCAPICALL osError **lasso_addDataSourceResult**(lasso_request_t *token*, const char * *data*)
Adds a datasource result value.

Datasource actions which require returning multiple values can use this to add those values. For example, this call can be used to add the name of a datasource that the module services or the names of the tables in a particular datasource.

Return
If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **data** - The data to add. Data must be in ISO8859-1 encoding.

LCAPICALL osError **lasso_addDataSourceResultUTF8**(lasso_request_t *token*, const char * *data*)
Adds a datasource result value.

Datasource actions which require returning multiple values can use this to add those values. For example, this call can be used to add the name of a datasource that the module services or the names of the tables in a particular datasource.

Return
If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **data** - The data to add. Data must be in UTF-8 encoding.

LCAPICALL osError **lasso_getDataSourceName**(lasso_request_t *token*, auto_lasso_value_t ** t*, bool ** useHostDefault*, auto_lasso_value_t ** usernamepassword*)

Gets the currently specified database name and associated data.

Return

If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **t** - The resulting database name.

- **useHostDefault** - The setting which specifies whether the hosts username/password combo should be used in the absence of a database specific combo.

- **usernamepassword** - The username/password combo for the database.

LCAPICALL osError **lasso_getTableName**(lasso_request_t *token*, auto_lasso_value_t ** t*)

Gets the currently specified table name.

Return

If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **t** - The resulting table name which will be placed in the *name* portion of the struct.

LCAPICALL osError **lasso_getTableEncoding**(lasso_request_t *token*, auto_lasso_value_t ** t*)

Gets the currently specified table encoding. This is the encoding as set in SiteAdmin for the current table.

Return

If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **t** - The resulting table encoding which will be placed in the *name* portion of the struct.

LCAPICALL osError **lasso_getSchemaName**(lasso_request_t *token*, auto_lasso_value_t ** schema*)

Gets the currently specified schema name.

Return

If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **schema** - The resulting schema name which will be placed in the *name* portion of the struct.

LCAPICALL osError **lasso_getDataHost**(lasso_request_t *token*, auto_lasso_value_t ** host*, auto_lasso_value_t
* usernamepassword)

Returns the host that maintains the current database.

The host name will be placed in the *name* portion of the struct while the port will be placed in the *data* portion.

> **Return**
>> If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

> **Parameters**
>> • **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.
>>
>> • **host** - The resulting host data.
>>
>> • **usernamepassword** - The resulting username/password combo.

LCAPICALL osError **lasso_getDataHostID**(lasso_request_t *token*, int ** outId*)

Returns the id of the host that maintains the current database.

> **Return**
>> If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

> **Parameters**
>> • **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.
>>
>> • **outId** - The resulting host id.

LCAPICALL osError **lasso_getDataHost2**(lasso_request_t *token*, auto_lasso_value_t ** host*, auto_lasso_value_t
* defaultSchema, auto_lasso_value_t ** usernamepassword*)

Returns the host that maintains the current database.

The host name will be placed in the *name* portion of the struct while the port will be placed in the *data* portion.

> **Return**
>> If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

> **Parameters**
>> • **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.
>>
>> • **host** - The resulting host data.
>>
>> • **defaultSchema** - The resulting default schema name which will be placed in the *name* portion of the struct.
>>
>> • **usernamepassword** - The resulting username/password combo.

LCAPICALL osError **lasso_getDataHostExtra**(lasso_request_t *token*, auto_lasso_value_t ** data*)

Returns the "extra" information associated with the current host.

The data, if any, will be placed in the *name* portion of the struct.

> **Return**
>> If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

> **Parameters**
>> • **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **data** - The resulting extra host data.

LCAPICALL osError **lasso_getDataHostIsDynamic**(lasso_request_t *token*, bool * *wasDyn*)
Indicates if the host is dynamic.

> ### Return
>
> If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.
>
> ### Parameters
>
> - **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.
> - **wasDyn** - Will be set to true if the host was dynamic and false if it was not.

LCAPICALL osError **lasso_getSkipRows**(lasso_request_t *token*, int * *recs*)
The number of rows that should be skipped in the found set.

-skiprecords

> ### Return
>
> If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.
>
> ### Parameters
>
> - **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.
> - **recs** - The resulting -skiprecords value.

LCAPICALL osError **lasso_getMaxRows**(lasso_request_t *token*, int * *recs*)
The maximum number of rows in the found set to return.

-maxrecords

> ### Return
>
> If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.
>
> ### Parameters
>
> - **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.
> - **recs** - The resulting -maxrecords value.

LCAPICALL osError **lasso_setRowID**(lasso_request_t *token*, int *id*)
Sets the currently specified record id (FileMaker specific).

> ### Return
>
> If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.
>
> ### Parameters
>
> - **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.
> - **id** - The numeric row id.

LCAPICALL osError **lasso_setRowID2**(lasso_request_t *token*, unsigned long long *id*)

LCAPICALL osError **lasso_getRowID2**(lasso_request_t *token*, unsigned long long * *id*)

LCAPICALL osError **lasso_getRowID**(lasso_request_t *token*, int * *id*)

Gets the currently specified record id (FileMaker specific).

Return

If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **id** - The numeric row id.

LCAPICALL osError **lasso_getPrimaryKeyColumn**(lasso_request_t *token*, auto_lasso_value_t * *v*)

Gets the name and the value of the currently specified primary key column.

Return

If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **v** - The resulting key name and value.

LCAPICALL osError **lasso_getPrimaryKeyColumn2**(lasso_request_t *token*, int *index*, auto_lasso_value_t * *v*, LP_TypeDesc * *desc*)

Gets the name and the value of the primary key column specified by index.

Return

If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **index** - The zero based index.

- **v** - The resulting key name and value.

- **desc** - A pointer to a LP_TypeDesc in which to place the original type of the input data.

LCAPICALL osError **lasso_getPrimaryKeyColumn3**(lasso_request_t *token*, int *index*, auto_lasso_value_t * *keyName*, lasso_type_t * *type*)

Gets the name and the value of the primary key column specified by index, preserving the original type.

Return

If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **index** - The zero based index.

- **keyName** - The name of the key field will be stored in the *name* member.

- **type** - The value portion of the key field in its original type.

LCAPICALL osError **lasso_getPrimaryKeyColumnCount**(lasso_request_t *token*, int * *count*)
    Gets the number of key columns.

    Return

        If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

    Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.
- **count** - A pointer to an int in which the number of key columns will be placed.

LCAPICALL osError **lasso_getInputColumnCount**(lasso_request_t *token*, int * *count*)
    Gets the number of input columns for this database action.

    Return

        If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

    Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.
- **count** - The resulting number of input columns.

LCAPICALL osError **lasso_getSortColumnCount**(lasso_request_t *token*, int * *count*)
    Gets the number of sort columns for this database action.

    Return

        If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

    Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.
- **count** - The resulting number of sort columns.

LCAPICALL osError **lasso_getInputColumn**(lasso_request_t *token*, int *num*, **auto_lasso_value_t** * *v*)
    Gets an individual input column by index.

    Return

        If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

    Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.
- **num** - the zero based index of the column to retrieve.
- **v** - The resulting value.

LCAPICALL osError **lasso_getInputColumn2**(lasso_request_t *token*,   int *num*,   **auto_lasso_value_t**   * *v*,   **LP_TypeDesc** * *desc*)
    Gets an individual input column by index.

    Return

        If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

    Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **num** - the zero based index of the column to retrieve.

- **v** - The resulting value.

- **desc** - A pointer to a LP_TypeDesc in which to place the original type of the input data.

LCAPICALL osError **lasso_getInputColumn3**(lasso_request_t *token*, int *num*, **auto_lasso_value_t** * *colName*, **lasso_type_t** * *type*)
Gets an individual input column by index, preserving the original type.

Return

If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **num** - the zero based index of the column to retrieve.

- **colName** - The name of the column will be stored in the *name* member.

- **type** - The value portion of the input column in its original type.

LCAPICALL osError **lasso_getSortColumn**(lasso_request_t *token*, int *num*, auto_lasso_value_t * *v*)
Gets an individual sort column by index.

Return

If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **num** - the zero based index of the sort column to retrieve.

- **v** - The resulting value.

LCAPICALL osError **lasso_findInputColumn**(lasso_request_t *token*, const char * *name*, **auto_lasso_value_t** * *value*)
Gets an individual input column by name.

Return

If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **name** - the name of the input column to retrieve.

- **value** - The resulting value.

LCAPICALL osError **lasso_findInputColumnW**(lasso_request_t *token*, const UChar * *name*, **auto_lasso_value_t** * *value*)
Gets an individual input column by name.

Return

If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **name** - the name of the input column to retrieve.

- **value** - The resulting value.

LCAPICALL osError **lasso_getLogicalOp**(lasso_request_t *token*, LP_TypeDesc * *op*)

Gets the logical operator for this database action.

Return

If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **op** - The resulting operator code. Operator codes are declared at the top of this file.

LCAPICALL osError **lasso_getReturnColumnCount**(lasso_request_t *token*, int * *count*)

Gets the number of return columns for this action.

Return

If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **count** - The resulting number of return columns.

LCAPICALL osError **lasso_getReturnColumn**(lasso_request_t *token*, int *num*, auto_lasso_value_t * *v*)

Gets an individual return column by index.

Return

If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **num** - The zero based index of the return column to retrieve.

- **v** - The resulting value.

LCAPICALL osError **lasso_addColumnInfo**(lasso_request_t *token*, const char * *name*, int *nullOK*, LP_TypeDesc *type*,
LP_TypeDesc *protection*)

Adds information about a particular column.

Column information should be added in the order in which the columns occur inthe database. Column information should be added no matter what the action is. For the show action, only column information is added.

Return

If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **name** - The column name.

- **nullOK** - IS a null value for this column ok?

- **type** - The data type for the column as represented by a type code. osConfig.h.

- **protection** - The code for the column's protection. osConfig.h.

LCAPICALL osError **lasso_addColumnInfo2**(lasso_request_t *token*,   const   char   * *name*,   int *nullOK*,
LP_TypeDesc *type*,   LP_TypeDesc *protection*,   const   char   ** *valueList*,
int *countValueList*)
Adds information, including valuelists, about a particular column.

Column information should be added in the order in which the columns occur inthe database. Column information should be added no matter what the action is. For the show action, only column information is added.

Return
If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **name** - The column name.

- **nullOK** - IS a null value for this column ok?

- **type** - The data type for the column as represented by a type code. osConfig.h.

- **protection** - The code for the column's protection. osConfig.h.

- **valueList** - An array of strings for the column's valuelist.

- **countValueList** - The number of values in the value list.

LCAPICALL osError **lasso_addResultRow**(lasso_request_t *token*, const char ** *columns*, unsigned long * *dataSizes*,
int *numColumns*)
Add the column data for the next result row.

Return
If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **columns** - An array of column values.

- **dataSizes** - An array of column value lengths. Every column in *columns* should have an associated length.

- **numColumns** - The number of items in the *columns* and *dataSizes* arrays.

LCAPICALL osError **lasso_addResultRow2**(lasso_request_t *token*, lasso_type_t * *cols*, int *num*)
Add the column data for the next result row. Column data is represented as Lasso types.

Return
If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **cols** - An array of column values.

- **num** - The number of items in the *cols* array.

LCAPICALL osError **lasso_addResultSet**(lasso_request_t *token*)

Adds a new result set to the datasource results.

At the onset of each LCAPI datasource call, there is an initial blank result set created. This is the first result set. After a datasource has added all the data for the first result set, if there is a second result set, the datasource should call lasso_addResultSet to start a new set and then proceed to populate the column information and data for that set as normal. This should be repeated for each result set. lasso_addResultSet should be called for each result set after the first. Calling lasso_addResultSet before populating the first result set will result in the first set being empty.

Return

If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

LCAPICALL osError **lasso_setNumRowsFound**(lasso_request_t *token*, int *num*)

Sets the number of rows found in the query.

This will not always be the same value as the number of rows added with the lasso_addResultRow call as the skip recs and max recs parameters effect that number.

Return

If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **num** - the number of found rows.

LCAPICALL osError **lasso_getDataSourceModuleName**(lasso_request_t *token*, auto_lasso_value_t * *val*)

Returns the name the current datasource module was registered with.

Return

If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **val** - The resulting datasource module name which will be palced in the *name* portion of the struct.

LCAPICALL osError **lasso_setDSPreparedPtr**(lasso_request_t *token*, void * *ptr*)

Allows a datasource to set a prepared statement pointer.

Return

If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **ptr** - The pointer value to set for later retrieval.

LCAPICALL osError **lasso_getDSPreparedPtr**(lasso_request_t *token*, void ** *ptr*)
Allows a datasource to get a previously set prepared statement pointer.

Return
If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **ptr** - The pointer, if it has been presiously set, will be placed in this parameter.

LCAPICALL osError **lasso_getDSConnection**(lasso_request_t *token*, lasso_dsconnection_t * *conn*)
Called to access the current datasource connection.

Datasource connections are set using the lasso_setDSConnection function.

Return
If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **conn** - A pointer in which to store the result.

LCAPICALL osError **lasso_setDSConnection**(lasso_request_t *token*, lasso_dsconnection_t *conn*)
Called to set the current connection for the datasource.

May recurse to deliver the datasourceCloseConnection message if there is already a valid lasso_dbconnection_t.

Return
If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **conn** - The connection value to set.

LCAPICALL osError **lasso_getDSUserData**(lasso_request_t *token*, void ** *outPtr*)
Provides access to the "user data" that was set when the datasorce was registered.

Return
If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **outPtr** - The pointer in which the user data is returned.

LCAPICALL osError **lasso_setActionStatement**(lasso_request_t *token*, const char * *stat*)
Called to set the statement for the current action.

Datasources must call this to support the action_statement tag.

Return
If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters
- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.
- **stat** - The statement value.

LCAPICALL osError **lasso_setActionStatementW**(lasso_request_t *token*, const UChar * *stat*)
Called to set the statement for the current action.

Datasources must call this to support the action_statement tag.

Return
If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters
- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.
- **stat** - The statement value.

LCAPICALL osError **lasso_getIsStatementOnly**(lasso_request_t *token*, bool * *out*)
Used to check for the -statementonly inline param.

Return
If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters
- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.
- **out** - A pointer in which to store the result.

Logging Functions

enum **log_level_t**
The various log levels.

Each log level can be configured to go to zero or more destinations.

Values:

LOG_LEVEL_ALWAYS
Always printed to window (only). Used internally in a few places.

typedef **log_level_t**
The various log levels.

Each log level can be configured to go to zero or more destinations.

LCAPICALL osError **lasso_log**(log_level_t *msgLevel*, const char * *fmt*, ...)
Log a message to stdout.

Defines

MACHINE_PC

MACHINE_UNIX

UCP(X)

kLPOpBegin

kLPOperatorBegin

kLPOpEnd

kLPOperatorEnd

kLPOpBeginW

kLPOperatorBeginW

kLPOpEndW

kLPOperatorEndW

FD_SETSIZE

LPW(X)

LCAPICALL

LCAPI_DEPRECATED

LCAPI_VERSION
> Defines for testing the LCAPI versionCurrent Lasso Version.

LCAPI_VERSION_1
> Lasso Version 5. First LCAPI release

LCAPI_VERSION_2
> Lasso Version 6. Second LCAPI release

LCAPI_VERSION_3
> Lasso Version 7. Third LCAPI release

LCAPI_VERSION_4
> Lasso Version 8. Fourth LCAPI release

LCAPI_VERSION_9
> Lasso Version 9

INITVAL(X)
> Initialize a *lasso_value_t* or *lasso_value_w_t* to be blank.
>
> It is recommended that each *lasso_value_t* be initialized using this macro before use.
>
> Example:

```
lasso_value_t myVal;
INITVAL(&myVal);
```

SET_MATCHED_NAME(param)

Typedefs

typedef **encodingMethod**

typedef **LP_TypeDesc**

typedef **auto_lasso_value_t**

> Special typedef so programmers know when Lasso will automatically dispose of the value.

typedef **auto_lasso_value_w_t**

> Special typedef so programmers know when Lasso will automatically dispose of the value.

typedef **tag_action_t**

> Types of actions tag modules could be called for.

> Ignore this for now, it may be put into use in a future version but is not utliized at present.

typedef **lasso_request_t**

> Special value passed to modules that identify the request.

> The same value should be used when calling into any LassoCAPI function.

typedef **lasso_type_t**

> Represents a type within Lasso.

> This opaque value represents an instance of a type within Lasso

typedef **lasso_dsconnection_t**

> Represents a datasource module connection.

> This opaque value is only interpreted by the datasource module itself. It can be stored via lasso_setDSConnection and retrieved via lasso_getDSConnection. Lasso will automatically send the datasource the datasourceCloseConnection message when it is time to close the connection.

typedef **(* lasso_ds_func)**(lasso_request_t *token*, datasource_action_t *action*, const auto_lasso_value_t **param*)
Service function for Lasso Datasource modules.

> A LCAPI datasource module should implement one of these and pass it to the lasso_registerDSModule or lasso_registerDSModuleW function when the datasource module is registered. The function will be called by Lasso when a datasource operation is to be performed.

> **Return**

>> If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

> **Parameters**

>> • **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

>> • **action** - The current action for the datasource module to perform.

>> • **param** - Any additional data associated with the action.

typedef **(* lasso_tag_func)**(lasso_request_t *token*, tag_action_t *action*)
Service function for a Lasso tag.

> An LCAPI tag should implement one of these and pass it to the lasso_registerTagModule or lasso_registerTagModuleW function when the LCAPI module is registered. The function will be called by Lasso every time the tag is called in a script.

> **Return**

>> If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

> **Parameters**

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **action** - Unused. Do not rely on this parameter to hold any particular value.

typedef **(* register_module_func)(**void**)**
LCAPI module registration callback.

All Lasso modules must export a function named "registerLassoModule". When Lasso attempts to load an LCAPI module, it will look for this exported funtion and, if found, call it. Within that function, the user may register any number of datasource modules or any number of tags using the lasso_registerDSModule, lasso_registerDSModuleW, lasso_registerTagModule or lasso_registerTagModuleW functions.

typedef **char osPathname[1024]**

Enums

enum **SortOrder**
Values:

orderAscending

orderDescending

orderCustom = 4

enum **[anonymous]**
Values:

kLassoGreaterThan = '> '

kLassoGreaterThanEquals = '>= '

kLassoEquals = '= '

kLassoLessThan = '< '

kLassoLessThanEquals = '<= '

kLassoBeginsWith = 'bgwt'

kLassoEndsWith = 'ends'

kLassoContains = 'cont'

kLassoNotContains = '!cts'

kLassoNotBeginsWith = '!bgs'

kLassoNotEndsWith = '!end'

kLassoAND = 'AND '

kLassoOR = 'OR '

kLassoNOT = 'NOT '

kLassoNo = 'no '

kLassoAny = 'any '

kLassoInList = ' nlt'

kLassoNotInList = '!nlt'

kLassoInFullText = ' ftx'

kLassoInRegExp = ' rxp'

kLassoNotInRegExp = '!rxp'

enum **lpEncodingType**
Values:

encodeURL

encodeRaw

encodeSmart

encodeBreak

encodeDefault

encodeStrictURL

encodeXML

encodeNone

encodeHTML

enum **osError**
Values:

osErrNoErr = 0

osErrAssert = -10000

osErrStreamReadError

osErrStreamWriteError

osErrMemory

osErrInvalidMemoryObject

osErrOutOfMemory

osErrOutOfStackSpace

osErrCouldNotLockMemory

osErrCouldNotUnlockMemory

osErrCouldNotDisposeMemory

osErrFile

osErrFileInvalid

osErrFileInvalidAccessMode

osErrCouldNotCreateOrOpenFile

osErrCouldNotCloseFile

osErrCouldNotDeleteFile

osErrFileNotFound

osErrFileAlreadyExists

osErrFileCorrupt

osErrVolumeDoesNotExist

osErrDiskFull

osErrDirectoryFull

osErrIOError

osErrInvalidPathname

osErrInvalidFilename

osErrFileLocked

osErrFileUnlocked

osErrFileIsOpen

osErrFileIsClosed

osErrBOF

osErrEOF

osErrCouldNotWriteToFile

osErrCouldNotReadFromFile

osErrResNotFound

osErrResource

osErrNetwork

osErrInvalidUsername

osErrInvalidPassword

osErrInvalidDatabase

osErrNoPermission

osErrFieldRestriction

osErrWebAddError

osErrWebUpdateError

osErrWebDeleteError

osErrInvalidParameter

osErrOverflow

osErrNilPointer

osErrUnknownError

osErrLoopAborted

osErrSyntaxError

osErrDivideByZero

osErrIllegalInstruction

osErrTagNotFound

osErrVarNotFound

osErrAborted

osErrFailure

osErrPreconditionFailed

osErrPostconditionFailed

osErrCriteriaNotMet

osErrIllegalUseOfFrozenInstance

osErrCompilationError

osErrNotImplemented

osErrSyntaxWarning

osErrWebRequiredFieldMissing = -800

osErrWebRepeatingRelatedField = -801

osErrWebNoSuchObject = -1728

osErrWebTimeout = -1712

osErrWebActionNotSupported = -802

osErrConnectionInvalid = -609

osErrWebModuleNotFound = -2000

osErrHTTPFileNotFound = 404

osErrDatasourceError = -3000

enum **datasource_action_t**

Types of actions datasources could be called for.

One of these will be passed as the *action* parameter for the lasso_ds_func.

Values:

datasourceInit
> Sent when a new "instance" of the datasource module is created.

datasourceTerm
> Sent when the "instance" of the datasource module is destroyed.

datasourceExists

datasourceNames
> Sent when Lasso attempts to gather the names of all the databases that the datasource module supports. Call lasso_addDataSourceResult once for each supported database.

datasourceTableNames
> Sent when Lasso attempts to gather the names of the tables available for the given database. The name of the database itself will be passed in the *param* parameter. Call lasso_addDataSourceResult once for each available table.

datasourceSearch
> Sent when the datasource module is to perform a -search action.

datasourceAdd
> Sent when the datasource module is to perform a -add action.

datasourceUpdate
> Sent when the datasource module is to perform a -update action.

datasourceDelete
> Sent when the datasource module is to perform a -delete action.

datasourceInfo
> Sent when the datasource module is to perform a -show action.

datasourceExecSQL
> Sent when the datasource module is to perform a -sql action.

datasourceRandom
> Sent when the datasource module is to perform a -random action.

datasourceSchemaNames
> Sent when Lasso attempts to gather the names of the schemas available for the given database. The name of the database itself will be passed in the *param* parameter. Call lasso_addDataSourceResult once for each available schema.
>
> This is currently only utilized for LJAPI.

datasourceCloseConnection
> Sent when the datasource module should close a connection previously set via the lasso_setDSConnection function.

datasourceTickle
> Sent to the datasource module when -database and -table are specified in an inline, but no action, or a -nothing action is used. The database could, perhaps, set or reset its connection to the database via lasso_setDSConnection. Or, it could do nothing.

datasourceDuplicate

datasourceScripts

datasourceImage

datasourceFindAll
> Sent when the datasource module is to perform a -findall action.

datasourceMatchesName

> Sent to the datasource to find out if it "goes" under the given name. The name which is being tested will be passed in the name member of the *param* parameter. The data member of the *param* parameter will be NULL and the dataSize member will be zero. If the name matches the name the datasource goes under, the datasource should both set the dataSize member to non-zero and return osErrNoErr. Otherwise, it is assumed that the name is not a matching name for the datasource module.

datasourcePrepareSQL

> Sent to the datasource to prepare a sql statement for later execution. This is sent when the datasource action was -prepare. The statement text is sent to the datasource in the *data* member of the *param* parameter that is passed to each datasource call.

datasourceUnprepareSQL

> Sent to the datasource after a datasourcePrepareSQL action. This permits the datasource to perform and necessary cleanup activities after executing a prepared statement.

datasourceMAXIMUM

datasourceNothing = -1

Functions

LCAPICALL osError **lasso_allocValue**(lasso_value_t * *result*, const char * *name*, unsigned int *nameSize*, const char * *data*, unsigned int *dataSize*, **LP_TypeDesc** *type*)

> Allocates a *lasso_value_t* with the indicated data.

Anything allocated with this function will NOT be garbage collected by Lasso and must be freed using lasso_freeValue.

Return

> If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **result** - A pointer to the *lasso_value_t* which will be constructed.
- **name** - The name portion which will be copied and set.
- **nameSize** - The size of the name parameter in bytes.
- **data** - The data portion which will be copied and set.
- **dataSize** - The size of the data parameter in bytes.
- **type** - The type code for the value.

LCAPICALL osError **lasso_allocValueW**(lasso_value_w_t * *result*, const UChar * *name*, unsigned int *nameSize*, const UChar * *data*, unsigned int *dataSize*, **LP_TypeDesc** *type*)

> Allocates a *lasso_value_w_t* with the indicated data.

Anything allocated with this function will NOT be garbage collected by Lasso and must be freed using lasso_freeValueW.

Return

> If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **result** - A pointer to the *lasso_value_w_t* which will be constructed.
- **name** - The name portion which will be copied and set.
- **nameSize** - The number of characters in the name portion.
- **data** - The data portion which will be copied and set.

- **dataSize** - The number of characters in the data portion.

- **type** - The type code for the value.

LCAPICALL osError **lasso_allocValueConv**(lasso_value_t *result*, const UChar *name*, unsigned int *nameSize*, const char * *nameEncoding*, const UChar * *data*, unsigned int *dataSize*, const char * *dataEncoding*, **LP_TypeDesc** *type*)

Allocates a *lasso_value_t* with the indicated data and encoding method.

This function can be used to convert Unicode data into any of the supported encodings.

Anything allocated with this function will NOT be garbage collected by Lasso and must be freed using lasso_freeValue.

Return

If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **result** - A pointer to the *lasso_value_w_t* which will be constructed.

- **name** - The name portion which will be converted and set.

- **nameSize** - The number of characters in the name portion.

- **nameEncoding** - The destination encoding for the name portion.

- **data** - The data portion which will be converted and set.

- **dataSize** - The number of characters in the data portion.

- **dataEncoding** - The destination encoding for the data portion.

- **type** - The type code for the value.

LCAPICALL osError **lasso_freeValue**(lasso_value_t * *result*)

Frees a previously allocated *lasso_value_t*.

Do not pass an auto_lasso_value_t to this function or you will end up with a double free.

Return

If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **result** - The *lasso_value_t* whose members will be freed.

LCAPICALL osError **lasso_freeValueW**(lasso_value_w_t * *result*)

Frees a previously allocated *lasso_value_w_t*.

Do not pass an auto_lasso_value_w_t to this function or you will end up with a double free.

Return

If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **result** - The *lasso_value_w_t* whose members will be freed.

LCAPICALL osError **lasso_registerTagModule**(const char * *namespaceName*, const char * *tagName*, **lasso_tag_func** *func*, int *flags*, const char * *description*)

Registers a new tag.

Return

If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **namespaceName** - The namespace in which the tag will be registered.

- **tagName** - The name which the tag can be called by.

- **func** - The user supplied callback for the tag.

- **flags** - Any special flags for the tag.

- **description** - A description for the tag.

LCAPICALL osError **lasso_registerTagModuleW(**const UChar * *namespaceName*, const UChar * *tagName*, lasso_tag_func *func*, int *flags*, const UChar * *description* **)**

LCAPICALL osError **lasso_typeGetTrait(**lasso_request_t *token*, lasso_type_t *from*, lasso_type_t * *into* **)**

LCAPICALL osError **lasso_setResultMessage(**lasso_request_t *token*, const char * *msg* **)**
Set result message string.

Return
If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **msg** - The result message string.

LCAPICALL osError **lasso_setResultMessageW(**lasso_request_t *token*, const UChar * *msg* **)**
Set result message string.

Return
If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **msg** - The result message string.

LCAPICALL osError **lasso_typeAlloc(**lasso_request_t *token*, const char * *typeName*, int *paramCount*, lasso_type_t * *paramsArray*, lasso_type_t * *outType* **)**
Allocates a new type instance.

The name of the type to allocate is signified by the second parameter. If a type initializer is found for the given name, it will be executed. An array of lasso_type_t parameters can be passed to the type initializer.

Return
If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **typeName** - The name of the type to instantiate.

- **paramCount** - The number of lasso_type_t parameters to pass to the initializer.

- **paramsArray** - An array of parameters to pass to the type initializer.

- **outType** - A pointer to the newly instantiated type.

LCAPICALL osError **lasso_typeAllocW**(lasso_request_t *token*,   const   UChar   * *typeName*,   int *paramCount*,
lasso_type_t * *paramsArray*, lasso_type_t * *outType*)

Allocates a new type instance.

The name of the type to allocate is signified by the second parameter. If a type initializer is found for the given name, it will be executed. An array of lasso_type_t parameters can be passed to the type initializer.

Return

If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **typeName** - The name of the type to instantiate.

- **paramCount** - The number of lasso_type_t parameters to pass to the initializer.

- **paramsArray** - An array of parameters to pass to the type initializer.

- **outType** - A pointer to the newly instantiated type.

LCAPICALL osError **lasso_typeAllocCustom**(lasso_request_t *token*, lasso_type_t * *outCustom*, const char * *name*)

Allocates a custom type within a type initializer.

This function is used within lasso_tag_funcs that were registered as being a type initializer (flag_typeInitializer). It initializes a blank custom type and sets the type's name to the provided value. The new type does not yet have a lineage and has no members added to it. New data or tag members should be added using lasso_typeAddMember. The new type must be the return value of the tag call, set via lasso_returnTagValue. Any inherited members will be added to the type after the LCAPI call returns.

Warning

Do NOT call this unless you are in a type initializer. If you are not in a type initializer, the result will be a type that will never be fully initialized.

Return

If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **outCustom** - The resulting newly allocated custom type.

- **name** - The new type's name.

LCAPICALL osError **lasso_typeAllocCustomW**(lasso_request_t *token*, lasso_type_t * *outCustom*, const UChar
* *name*)

Allocates a custom type within a type initializer.

This function is used within lasso_tag_funcs that were registered as being a type initializer (flag_typeInitializer). It initializes a blank custom type and sets the type's name to the provided value. The new type does not yet have a lineage and has no members added to it. New data or tag members should be added using lasso_typeAddMember. The new type must be the return value of the tag call, set via lasso_returnTagValue. Any inherited members will be added to the type after the LCAPI call returns.

Warning

Do NOT call this unless you are in a type initializer. If you are not in a type initializer, the result will be a type that will never be fully initialized.

Return

If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **outCustom** - The resulting newly allocated custom type.

- **name** - The new type's name.

LCAPICALL osError **lasso_typeAllocFromProto**(lasso_request_t *token*, lasso_type_t *proto*, lasso_type_t * *out*)

Allocate a new type instance based on the given type instance.

The given type's tag members will be referenced in the new type. No data members are added to the new type.

Return

If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **proto** - The original type instance whose tag members will be referenced.

- **out** - The resulting new type instance.

LCAPICALL osError **lasso_typeAddMember**(lasso_request_t *token*, lasso_type_t *to*, const char * *named*, lasso_type_t *member*)

Adds a member to a type instance.

If the new member is a tag, it will be added to the tag members for the type. Otherwise, the new member will be added as a data member.

Return

If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **to** - The type instance to which the new member will be added.

- **named** - The name for the new member.

- **member** - The new member to add.

LCAPICALL osError **lasso_typeAddMemberW**(lasso_request_t *token*, lasso_type_t *to*, const UChar * *named*, lasso_type_t *member*)

Adds a member to a type instance.

If the new member is a tag, it will be added to the tag members for the type. Otherwise, the new member will be added as a data member.

Return

If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **to** - The type instance to which the new member will be added.

- **named** - The name for the new member.

- **member** - The new member to add.

LCAPICALL osError **lasso_typeAddTagMember**(lasso_request_t *token*, lasso_type_t *to*, const char **named*, lasso_type_t *member*)
Adds a tag member to a type instance.

Return
    If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **to** - The type instance to which the new tag member will be added.

- **named** - The name for the new member.

- **member** - The new tag member to add.

LCAPICALL osError **lasso_typeAddTagMember2**(lasso_request_t *token*, lasso_type_t *to*, const char **named*, lasso_tag_func *nativeTagFunction*)
Adds a tag member to a type instance.

Return
    If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **to** - The type instance to which the new tag member will be added.

- **named** - The name for the new member.

- **nativeTagFunction** - The function add.

LCAPICALL osError **lasso_typeAddDataMember**(lasso_request_t *token*, lasso_type_t *to*, const char **named*, lasso_type_t *member*)
Adds a data member to a type instance.

Return
    If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **to** - The type instance to which the new data member will be added.

- **named** - The name for the new member.

- **member** - The new data member to add.

LCAPICALL osError **lasso_typeAddTagMemberW**(lasso_request_t *token*, lasso_type_t *to*, const UChar **named*, lasso_type_t *member*)
Adds a tag member to a type instance.

Return
> If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters
- `token` - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.
- `to` - The type instance to which the new tag member will be added.
- `named` - The name for the new member.
- `member` - The new tag member to add.

LCAPICALL osError **lasso_typeAddTagMember2W**(lasso_request_t *token*, lasso_type_t *to*, const UChar * *named*, lasso_tag_func *nativeTagFunction*)
Adds a tag member to a type instance.

Return
> If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters
- `token` - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.
- `to` - The type instance to which the new tag member will be added.
- `named` - The name for the new member.
- `nativeTagFunction` - The function to add.

LCAPICALL osError **lasso_typeAddDataMemberW**(lasso_request_t *token*, lasso_type_t *to*, const UChar * *named*, lasso_type_t *member*)
Adds a data member to a type instance.

Return
> If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters
- `token` - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.
- `to` - The type instance to which the new data member will be added.
- `named` - The name for the new member.
- `member` - The new data member to add.

LCAPICALL osError **lasso_typeGetDataMember**(lasso_request_t *token*, lasso_type_t *from*, const char * *named*, lasso_type_t * *out*)
Retrieves a data member from a type instance.

Return
> If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters
- `token` - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.
- `from` - The type instance in which to search.
- `named` - The name of the member to look for.

- **out** - A pointer to a type instance in which to store the found member.

LCAPICALL osError **lasso_typeGetDataMemberW**(lasso_request_t *token*, lasso_type_t *from*, const UChar * *named*,
lasso_type_t * *out*)
Retrieves a data member from a type instance.

Return
If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **from** - The type instance in which to search.

- **named** - The name of the member to look for.

- **out** - A pointer to a type instance in which to store the found member.

LCAPICALL osError **lasso_typeSetDataMember**(lasso_request_t *token*, lasso_type_t *from*, const char * *named*,
lasso_type_t *to*)
Sets a data member of a type instance.

Return
If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **from** - The type instance in which to search.

- **named** - The name of the member to look for.

- **to** - A pointer to a type instance. The new value for the data member.

LCAPICALL osError **lasso_typeSetDataMemberW**(lasso_request_t *token*, lasso_type_t *from*, const UChar * *named*,
lasso_type_t *to*)
Sets a data member of a type instance.

Return
If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **from** - The type instance in which to search.

- **named** - The name of the member to look for.

- **to** - A pointer to a type instance. The new value for the data member.

LCAPICALL osError **lasso_setPtrMember**(lasso_request_t *token*, lasso_type_t *self*, const char * *name*, void * *data*,
void(**dtor*)(void *))
Allows storage of an opaque pointer value.

The pointer member is given and name and is stored as an integer in the type instance.

Return
> If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters
- `token` - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.
- `self` - The type instance in which to add the pointer member.
- `name` - The name for the new member.
- `data` - The pointer value which will be added.
- `dtor` - A pointer to a function which will be called when the member is destroyed.

LCAPICALL osError **lasso_setPtrMember2**(lasso_request_t *token*, **lasso_type_t** *from*, const char * *name*, void * *data*, void(**dtor*)(void *obj), void *(*copyFunc)(void *obj)**)**
> Allows storage of an opaque pointer value.

The pointer member is given and name and is stored as an integer in the type instance.

Return
> If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters
- `token` - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.
- `from` - The type instance in which to add the pointer member.
- `name` - The name for the new member.
- `data` - The pointer value which will be added.
- `dtor` - A pointer to a function which will be called when the member is destroyed.
- `copyFunc` - A pointer to a function which will be called when the member is copied.

LCAPICALL osError **lasso_setPtrMemberW**(lasso_request_t *token*, **lasso_type_t** *self*, const UChar * *name*, void * *data*, void(**dtor*)(void *)**)**
> Allows storage of an opaque pointer value.

The pointer member is given and name and is stored as an integer in the type instance.

Return
> If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters
- `token` - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.
- `self` - The type instance in which to add the pointer member.
- `name` - The name for the new member.
- `data` - The pointer value which will be added.
- `dtor` - A pointer to a function which will be called when the member is destroyed.

LCAPICALL osError **lasso_setPtrMember2W**(lasso_request_t *token*, **lasso_type_t** *from*, const UChar * *named*, void * *data*, void(**dtor*)(void *obj), void *(*copyFunc)(void *obj)**)**
> Allows storage of an opaque pointer value.

The pointer member is given and name and is stored as an integer in the type instance.

Return

If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **from** - The type instance in which to add the pointer member.

- **named** - The name for the new member.

- **data** - The pointer value which will be added.

- **dtor** - A pointer to a function which will be called when the member is destroyed.

- **copyFunc** - A pointer to a function which will be called when the member is copied.

LCAPICALL osError **lasso_getPtrMember**(lasso_request_t *token*, lasso_type_t *self*, const char * *name*, void ** *data*)

Retrieves a previously added pointer value.

Return

If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **self** - The type instance from which to retrieve.

- **name** - The name of the previously added pointer value.

- **data** - A pointer to a void * in which to store the found value.s

LCAPICALL osError **lasso_getPtrMemberW**(lasso_request_t *token*, lasso_type_t *self*, const UChar * *name*, void ** *data*)

Retrieves a previously added pointer value.

Return

If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **self** - The type instance from which to retrieve.

- **name** - The name of the previously added pointer value.

- **data** - A pointer to a void * in which to store the found value.s

LCAPICALL osError **lasso_getTagSelf**(lasso_request_t *token*, lasso_type_t * *self*)

Returns the type instance that the current tag call was a member of.

This is used in member tags of custom types to return the target of the current call.

Return

If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **self** - A pointer in which to store the resulting type instance.

LCAPICALL osError **lasso_typeGetName**(lasso_request_t *token,* lasso_type_t *target,* auto_lasso_value_t * *outName*)

Returns the name of the target type.

Return

If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **target** - The type instance to get the name of.

- **outName** - The resulting name which will be stored in the *name* portion of the struct.

LCAPICALL osError **lasso_typeGetNameW**(lasso_request_t *token,* lasso_type_t *target,* auto_lasso_value_w_t * *outName*)

Returns the name of the target type.

Return

If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **target** - The type instance to get the name of.

- **outName** - The resulting name which will be stored in the *name* portion of the struct.

LCAPICALL osError **lasso_typeIsA**(lasso_request_t *token,* lasso_type_t *target,* LP_TypeDesc *type*)

Tests to see if a type is an instance of another type.

Test by type code.

Return

If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **target** - The target instance to test.

- **type** - The type code to test for.

LCAPICALL osError **lasso_typeIsA2**(lasso_request_t *token,* lasso_type_t *target,* const char * *typeName*)

Tests to see if a type is an instance of another type.

Test by type name.

Return

If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **target** - The target instance to test.

- **typeName** - The type name to test for.

LCAPICALL osError **lasso_typeIsA2W**(lasso_request_t *token*, **lasso_type_t** *target*, const UChar * *typeName*)

Tests to see if a type is an instance of another type.

Test by type name.

Return

If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **target** - The target instance to test.

- **typeName** - The type name to test for.

LCAPICALL osError **lasso_typeIsA3**(lasso_request_t *token*, **lasso_type_t** *target*, **lasso_type_t** *type*)

Tests to see if a type is an instance of another type.

Test based on another type instance.

Return

If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **target** - The target instance to test.

- **type** - The type name to test for.

LCAPICALL osError **lasso_returnTagValue**(lasso_request_t *token*, **lasso_type_t** *value*)

Specifies the return value of the current LCAPI tag call.

Any type can be returned. A tag can only have one return value. Setting another return value will overwrite the previous.

Return

If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **value** - The value to return.

LCAPICALL osError **lasso_returnTagValueBoolean**(lasso_request_t *token*, bool *b*)

Return a boolean value from the current LCAPI tag call.

Return

If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **b** - The boolean value to return.

LCAPICALL osError **lasso_returnTagValueInteger**(lasso_request_t *token*, int64_t *i*)
    Return an integer value from the current LCAPI tag call.

    Return

        If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

    Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **i** - The integer value to return.

LCAPICALL osError **lasso_returnTagValueString**(lasso_request_t *token*, const char * *p*, int *l*)
    Return a string value from the current LCAPI tag call.

    Return

        If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

    Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **p** - The character data to be returned. The data should be in UTF-8 encoding.

- **l** - The length of the character data to return.

LCAPICALL osError **lasso_returnTagValueStringW**(lasso_request_t *token*, const UChar * *p*, int *l*)
    Return a string value from the current LCAPI tag call.

    Return

        If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

    Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **p** - The character data to be returned.

- **l** - The length of the character data to return.

LCAPICALL osError **lasso_returnTagValueDecimal**(lasso_request_t *token*, double *d*)
    Return a decimal value from the current LCAPI tag call.

    Return

        If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

    Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **d** - The double value to return.

LCAPICALL osError **lasso_returnTagValueBytes**(lasso_request_t *token*, const char * *data*, int *length*)
> Return binary data from the current LCAPI tag call.

> When using this, the result of the LCAPI call will be a bytes type. This can be called as many times as needed and new data will be appended to any previous data.

> ### Return
>> If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

> ### Parameters
>> - **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.
>> - **data** - The binary data to return.
>> - **length** - The number of bytes to return.

LCAPICALL osError **lasso_getTagParam2**(lasso_request_t *token*, int *paramIndex*, **lasso_type_t** * *result*)
> Retrieves a parameter that was passed to the LCAPI tag call.

> The parameter index is zero based.

> ### Return
>> If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

> ### Parameters
>> - **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.
>> - **paramIndex** - The zero based index of the desired parameter.
>> - **result** - The resulting parameter value.

LCAPICALL osError **lasso_findTagParam2**(lasso_request_t *token*, const char * *paramName*, **lasso_type_t** * *result*)
> Retrieves a parameter that was passed to the LCAPI tag call.

> The parameter must have been specified by keyword.

> ### Return
>> If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

> ### Parameters
>> - **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.
>> - **paramName** - The keyword of the desired parameter.
>> - **result** - The resulting parameter value.

LCAPICALL osError **lasso_findTagParam2W**(lasso_request_t *token*, const UChar * *paramName*, **lasso_type_t** * *result*)
> Retrieves a parameter that was passed to the LCAPI tag call.

> The parameter must have been specified by keyword.

> ### Return
>> If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

> ### Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **paramName** - The keyword of the desired parameter.

- **result** - The resulting parameter value.

LCAPICALL osError **lasso_registerConstant2(**const char * *namespaceName*, const char * *name*, **lasso_type_t** *val* **)**

Register a constant value.

Constants can be called just like tags, but the resulting value will only have a single instance and will have both its type and value frozen. This is usually called at the same time that tags are registered.

Return

If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **namespaceName** - The namespace for the constant.

- **name** - The name for the constant.

- **val** - The value for the constant.

LCAPICALL osError **lasso_registerConstant2W(**const   UChar   * *namespaceName*,   const   UChar   * *name*, **lasso_type_t** *val* **)**

Register a constant value.

Constants can be called just like tags, but the resulting value will only have a single instance and will have both its type and value frozen. This is usually called at the same time that tags are registered.

Return

If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **namespaceName** - The namespace for the constant.

- **name** - The name for the constant.

- **val** - The value for the constant.

LCAPICALL osError **lasso_registerConstant(**const char * *name*, **lasso_type_t** *val* **)**

Register a constant value.

Constants can be called just like tags, but the resulting value will only have a single instance and will have both its type and value frozen. This is usually called at the same time that tags are registered.

Return

If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **name** - The name for the constant.

- **val** - The value for the constant.

LCAPICALL osError **lasso_registerConstantW(**const UChar * *name*, **lasso_type_t** *val* **)**

Register a constant value.

Constants can be called just like tags, but the resulting value will only have a single instance and will have both its type and value frozen. This is usually called at the same time that tags are registered.

Return
> If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters
- `name` - The name for the constant.
- `val` - The value for the constant.

LCAPICALL osError **`lasso_getTagName`**(lasso_request_t *token*, auto_lasso_value_t * *result*)
> Fetches the name of the tag that triggered this call.

> For example, in the case of: [my_tag: ...] the resulting value would be "my_tag".

Return
> If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters
- `token` - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.
- `result` - The resulting tag name which will be placed in the *name* portion of the struct.

LCAPICALL osError **`lasso_getTagNameW`**(lasso_request_t *token*, auto_lasso_value_w_t * *result*)
> Fetches the name of the tag that triggered this call.

> For example, in the case of: [my_tag: ...] the resulting value would be "tag".

Return
> If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters
- `token` - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.
- `result` - The resulting tag name which will be placed in the *name* portion of the struct.

LCAPICALL osError **`lasso_getTagParamCount`**(lasso_request_t *token*, int * *result*)
> Fetches the number of parameters that were passed to the tag.

Return
> If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters
- `token` - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.
- `result` - A pointer in which to store the result.

LCAPICALL osError **`lasso_getTagParam`**(lasso_request_t *token*, int *paramIndex*, auto_lasso_value_t * *result*)
> Fetches the indicated parameter that was sent to the tag.

> Parameter indexes start at zero. If the parameter was specified as a keyword/value pair, the keyword will be placed in the *name* portion of the struct and the value in the *data* portion. If the parameter was provided without a keyword, the *name* portion of the struct will be NULL.

> All parameters values will be converted to string, regardless of the original type.

Return
> If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **paramIndex** - The zero based index of the desired parameter.

- **result** - The resulting parameter data.

LCAPICALL osError **lasso_getTagParamW**(lasso_request_t *token*, int *paramIndex*, **auto_lasso_value_w_t** * *result*)
Fetches the indicated parameter that was sent to the tag.

Parameter indexes start at zero. If the parameter was specified as a keyword/value pair, the keyword will be placed in the *name* portion of the struct and the value in the *data* portion. If the parameter was provided without a keyword, the *name* portion of the struct will be NULL.

All parameters values will be converted to string, regardless of the original type.

Return
If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **paramIndex** - The zero based index of the desired parameter.

- **result** - The resulting parameter data.

LCAPICALL osError **lasso_tagParamIsDefined**(lasso_request_t *token*, const char * *paramName*)
Returns osErrNoErr if the param WAS defined.

Any other result means it wasn't defined. Only parameters specified with a keyword should be searched for.

Return
If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **paramName** - The name of the keyworded parameter to search for.

LCAPICALL osError **lasso_tagParamIsDefinedW**(lasso_request_t *token*, const UChar * *paramName*)
Returns osErrNoErr if the param WAS defined.

Any other result means it wasn't defined. Only parameters specified with a keyword should be searched for.

Return
If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **paramName** - The name of the keyworded parameter to search for.

LCAPICALL osError **lasso_findTagParam**(lasso_request_t *token*, const char * *paramName*, **auto_lasso_value_t** * *result*)
Retrieves a parameter that was passed to the LCAPI tag call.

The parameter must have been specified by keyword. The keyword will be placed in the *name* portion of the struct and the value in the *data* portion. If only a keyword was provided, the *data* portion will be NULL.

Return

If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **paramName** - The keyword of the desired parameter.

- **result** - The resulting parameter value.

LCAPICALL osError **lasso_findTagParamW**(lasso_request_t *token*, const UChar * *paramName*, auto_lasso_value_w_t * *result*)

Retrieves a parameter that was passed to the LCAPI tag call.

The parameter must have been specified by keyword. The keyword will be placed in the *name* portion of the struct and the value in the *data* portion. If only a keyword was provided, the *data* portion will be NULL.

Return

If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **paramName** - The keyword of the desired parameter.

- **result** - The resulting parameter value.

LCAPICALL osError **lasso_setVariable**(lasso_request_t *token*, const char * *named*, const char * *value*, int *index*)

LCAPICALL osError **lasso_setVariableW**(lasso_request_t *token*, const UChar * *named*, const UChar * *value*, int *index*)

LCAPICALL void **lasso_getPlatformSpecificPath**(const char * *inInternalPath*, osPathname *outPlatformPath*)

This function is a no-op in Lasso 9.

Parameters

- **inInternalPath** - The internal pathname.

- **outPlatformPath** - The resulting platform specific pathname.

LCAPICALL void **lasso_getInternalPath**(const char * *inPlatformPath*, osPathname *outInternalPath*)

This function is a no-op in Lasso 9.

Parameters

- **inPlatformPath** - the platform specific path name.

- **outInternalPath** - The resulting internal pathname.

LCAPICALL void **lasso_fullyQualifyPath**(lasso_request_t *token*, const char * *inRelativePath*, osPathname *outFullyQualified*)

This function is a no-op in Lasso 9.

If the path is already from the root it won't be changed.

Return

If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **inRelativePath** - The relative path.

- **outFullyQualified** - the resulting fully qualified path.

LCAPICALL void **lasso_resolvePath**(lasso_request_t *token*, const char * *inPath*, osPathname *outFullPath*)

This function is a no-op in Lasso 9.

Return

If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **inPath** - The path to resolve.

- **outFullPath** - The resulting full path.

LCAPICALL void **lasso_resolveIncludePath**(lasso_request_t *token*, const char * *inPath*, osPathname *outFullPath*)

This function is a no-op in Lasso 9.

Resulting path must name an item within the web-root or a blank string is returned.

Return

If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **inPath** - The path to resolve.

- **outFullPath** - The resulting full path.

LCAPICALL void **lasso_internalToFullPlatformPath**(lasso_request_t *token*, const char * *relativeOrFullInternalPath*, osPathname *fullPlatformPath*)

This function is a no-op in Lasso 9.

Takes a path, such as one which may have been passed to any Lasso tag, and converts it into a full platform specific path name.

Return

If the function succeeds, osErrNoErr is returned, otherwise, an error code indicating the problem is returned.

Parameters

- **token** - The request token which should be passed to subsequent LCAPI functions for the duration of the current service function call.

- **relativeOrFullInternalPath** - The input path to convert.

- **fullPlatformPath** - The resulting converted path.

LCAPICALL bool **lasso_isFullInternalPath**(const char * *path*)
> Returns true if the path begins with a forward slash.

> ### Return
>> True if the given internal path is a full path from the filesystem root.

> ### Parameters
>> • **path** - The path to test.

Variables

const int **kNumErrors**

const int **kMaxPathLen**

struct **lasso_value_t**
> *#include <LassoCAPI.h>* Used for retrieving data from some LCAPI functions.

> The *lasso_value_t* struct is used for shuffling data into and out of LassoCAPI functions.

> Any LCAPI function that constructs one of these will insure that Lasso properly disposes of the data once the user's LassoCAPI function call is complete.

> *lasso_value_t* is suitable for transferring binary data as long as the *nameSize* or *dataSize* members are properly set with the size of the data in bytes.

> Depending on the purpose of the function which is constructing the *lasso_value_t*, any combination of the *name* or *data* members may be filled in or may be NULL. Consult the documentation of the specific function for more details.

> Should be initialized using the *INITVAL(X)* macro before use.

> ### Public Members

>> const char* **name**
>>> The *name* portion.

>>> May be NULL.

>> unsigned int **nameSize**
>>> The size of the *name* member, in bytes.

>> const char* **data**
>>> The *value* portion.

>>> May be NULL.

>> unsigned int **dataSize**
>>> The size of the *data* member, in bytes.

>> LP_TypeDesc **type**
>>> The type code for the *data* member.

struct **lasso_value_w_t**
> *#include <LassoCAPI.h>* Used for retrieving Unicode character data from some LCAPI functions.

> The *lasso_value_w_t* struct is used for shuffling Unicode character data into and out of LassoCAPI functions.

> Any LCAPI function that constructs one of these will insure that Lasso properly disposes of the data once the user's LassoCAPI function call is complete.

Depending on the purpose of the function which is constructing the *lasso_value_t*, any combination of the *name* or *data* members may be filled in or may be NULL. Consult the documentation of the specific function for more details.

Should be initialized using the *INITVAL(X)* macro before use.

Public Members

const UChar* **name**
> The *name* portion.

> May be NULL.

unsigned int **nameSize**
> The size of the *name* member, in bytes.

const UChar* **data**
> The *value* portion.

> May be NULL.

unsigned int **dataSize**
> The size of the *data* member, in bytes.

LP_TypeDesc **type**
> The type code for the *data* member.

struct **lasso_request_t_**
> *#include <LassoCAPI.h>* Special value passed to modules that identify the request.

> The same value should be used when calling into any LassoCAPI function.

struct **lasso_type_t_**
> *#include <LassoCAPI.h>* Represents a type within Lasso.

> This opaque value represents an instance of a type within Lasso

struct **lasso_dsconnection_t_**
> *#include <LassoCAPI.h>* Represents a datasource module connection.

> This opaque value is only interpreted by the datasource module itself. It can be stored via lasso_setDSConnection and retrieved via lasso_getDSConnection. Lasso will automatically send the datasource the datasourceCloseConnection message when it is time to close the connection.

namespace **LPTypes**

Variables

const LP_TypeDesc **lpTypeString**

const LP_TypeDesc **lpTypeNull**

const LP_TypeDesc **lpTypeInteger**

const LP_TypeDesc **lpTypeBoolean**

const LP_TypeDesc **lpTypeBytes**

const LP_TypeDesc **lpTypeDecimal**

const LP_TypeDesc **lpTypeDateTime**

const LP_TypeDesc **lpTypeArray**

const LP_TypeDesc **lpTypeTag**

const LP_TypeDesc **lpTypePair**

const LP_TypeDesc **lpTypeCustom**

const LP_TypeDesc **kDateDataType**

const LP_TypeDesc **kProtectionNone**

const LP_TypeDesc **kProtectionReadOnly**

Lasso Java API

48.1 LJAPI Overview

The *Lasso Java Application Programming Interface* (LJAPI) allows you to run Java code from within Lasso. This allows for custom Java code to be created using Java's libraries that can then be run on all platforms Lasso supports. It also gives you access to use Java's standard classes to create and manipulate Java objects without writing a line of Java code.

The LJAPI functionality is implemented in an LCAPI module that bridges the C/C++ Java Native Interface (JNI) to Lasso. See the Oracle website for more information about interoperating with Java using JNI[68].

48.1.1 Requirements

- Lasso Server installed on a supported OS
- Java installed
- Any other OS-specific packages required for Java support in Lasso installed

48.1.2 Executing a Static Method

Static methods are methods that are associated with a class, but are not run on an instantiated object of that class. This example will walk you through running the Java static class method `Math.scalb`. This method takes in a floating point and an integer and returns the value of multiplying the float by 2 to the power of the integer.

Note: If you are running the example code in a shell script or via the command-line interpreter instead of in a Lasso Server instance, you'll need to load the LJAPI environment. This can be done with the following two lines of code (replace "LJAPI9.dylib" with the name of the library for your OS's installation). See the section **Loading Libraries in Shell Scripts** in the **Command-Line Tools** chapter for more information.

```
lcapi_loadModule((sys_masterHomePath || sys_homePath) + '/LassoModules/LJAPI9.dylib')
ljapi_initialize
```

Static Method Code

```
local(class) = java_jvm_getenv->FindClass('java/lang/Math')
local(mID)   = java_jvm_getenv->GetStaticMethodID(#class, 'scalb', '(FI)F')

java_jvm_getenv->CallStaticFloatMethod(#class, #mID, jfloat(4.0), jint(3))

// => 32.000000
```

---

[68] http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html

Static Method Walkthrough

One thing to notice is that all the communication is done using **java_jvm_getenv**. This method returns the **java_jnienv** object for the Lasso instance, and it is this object that allows Lasso to communicate with the Java Virtual Machine (JVM).

1. The first line of code finds the Java class we want to work with and returns a Lasso **jobject**; storing it into the local variable "class". The string value that gets passed to **FindClass** is the fully qualified class name signature (or array type signature). For more information, see the FindClass[69] documentation.

   ```
   local(class) = java_jvm_getenv->FindClass('java/lang/Math')
   ```

2. The next line of code looks up the method ID for the method we want to execute and returns it as a **jmethodid** type; storing it into the "mID" variable. **GetStaticMethodID** takes in the class (jobject) object we found in the first line, the name of the method as the second parameter, and the signature for that method as the third parameter. For more information, see the GetStaticMethodID[70] documentation.

   ```
   local(mID)   = java_jvm_getenv->GetStaticMethodID(#class, 'scalb', '(FI)F')
   ```

3. The method signature **(FI)F** specifies that it takes a float and an int parameter and returns a float. The easiest way to find the signature for a method is to use the **javap** command on the command line. In the example below, we run **javap -s -p java.lang.Math** to get all the method signatures found in the "java.lang.Math" class, and we use **grep** to filter and find the "scalb" method. You'll notice in the result that there are actually two methods with the same name but with different signatures, and we're using the second one:

   ```
   $> javap -s -p java.lang.Math | grep -A 1 scalb
   public static double scalb(double, int);
     Signature: (DI)D
   --
   public static float scalb(float, int);
     Signature: (FI)F
   ```

4. Finally, we execute the method using **CallStaticFloatMethod** which takes in the class object from the first step and the method ID from the second step and then the required parameters for the method we are calling, if any. Note that we must convert Lasso decimal objects to **jfloat** and Lasso integer objects to **jint**.

   ```
   java_jvm_getenv->CallStaticFloatMethod(#class, #mID, jfloat(4.0), jint(3))
   ```

48.1.3 Instantiating a Java Object and Executing a Member Method

Member methods are methods that are associated with a class and are run on an instantiated object of that class. This example will walk you through creating a ZipFile object and running the **size** method on that object to find out how many items are in the zip file.

To run this example yourself, supply a zip file and replace the path and file name in the example with the path and name of your zip file.

Java Object Member Method Code

```
local(class) = java_jvm_getenv->FindClass('java/util/zip/ZipFile')
local(mID)   = java_jvm_getenv->GetMethodID(#class, '<init>', '(Ljava/lang/String;)V')
local(obj)   = java_jvm_getenv->NewObject(#class, #mID, '/path/to/zipfile.zip')
```

---

[69] http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp16027
[70] http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp20950

```
local(class) = java_jvm_getenv->GetObjectClass(#obj)
local(mID)   = java_jvm_getenv->GetMethodID(#class, 'size', '()I')

java_jvm_getenv->CallIntMethod(#obj, #mID)

// => 92
```

Java Object Member Method Walkthrough

Once again, all the communication is done using the **java_jvm_getenv** method, which wraps the Lasso instance's **java_jnienv** object.

1. The first line of code gets the specified Java class and stores a Lasso **jobject** into the local variable "class". The value that gets passed to **FindClass** is the fully qualified class name signature (or array type signature). For more information, see the FindClass[71] documentation.

   ```
   local(class) = java_jvm_getenv->FindClass('java/util/zip/ZipFile')
   ```

2. Next, the code finds the method ID for the constructor method by passing the class object we found in the first step, "<init>" for the method name, and the method signature as the third argument:

   ```
   local(mID)   = java_jvm_getenv->GetMethodID(#class, '<init>', '(Ljava/lang/String;)V')
   ```

3. The method signature **(Ljava/lang/String;)V** specifies that it takes a string parameter and returns "void". The easiest way to find the signature for a method is to use the **javap** command on the command line. In the example below, we run **javap -s -p java.util.zip.ZipFile** to get all the method signatures found in the "java.util.zip.ZipFile" class, and we use **grep** to filter and find the constructor methods. You'll notice in the result that there are actually three constructor methods, each with different signatures, and we are using the first one:

   ```
   $> javap -s -p java.util.zip.ZipFile | grep -A 1 "public java.util.zip.ZipFile"
   public java.util.zip.ZipFile(java.lang.String) throws java.io.IOException;
     Signature: (Ljava/lang/String;)V
   --
   public java.util.zip.ZipFile(java.io.File, int) throws java.io.IOException;
     Signature: (Ljava/io/File;I)V
   --
   public java.util.zip.ZipFile(java.io.File) throws java.util.zip.ZipException, java.io.
     →IOException;
     Signature: (Ljava/io/File;)V
   ```

4. After finding the constructor method for our class, the code instantiates an object by passing that information into **NewObject**. The line of code below stores a Java object into "obj" by calling **NewObject** with the class information, method ID, and any additional parameters required by the constructor (in this case, the path to the zipped file). For more information on **NewObject**, see the NewObject[72] documentation.

   ```
   local(obj)   = java_jvm_getenv->NewObject(#class, #mID, '/path/to/zipfile.zip')
   ```

5. The next line isn't actually necessary since the "class" variable already has the class information for "java.util.zip.ZipFile", but we have it here to demonstrate how you could deal with wanting to call methods on Java objects that were returned by other methods. So, **GetObjectClass** returns the class information for the specified object. For more information, see the GetObjectClass[73] documentation.

---

[71] http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp16027
[72] http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp4517
[73] http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp16454

```
local(class) = java_jvm_getenv->GetObjectClass(#obj)
```

6. The next line gets the method ID for the **size** member method and stores it in the local variable "mID":

```
local(mID)   = java_jvm_getenv->GetMethodID(#class, 'size', '()I')
```

7. Finally, we execute the **size** member method by calling **CallIntMethod** with the Java object as the first parameter and the method ID for **size** as the second parameter. Notice that the return type (int) is in the name of the method. There are a number of these methods for various return types[74].

```
java_jvm_getenv->CallIntMethod(#obj, #mID)
```

48.2 Lasso Types and Methods for LJAPI

This chapter provides a reference to all of the types and functions in LJAPI.

48.2.1 Methods

ljapi_initialize()
> Creates a Java Virtual Machine for the running Lasso thread. A Lasso Server instance calls this method when it starts up.

java_jvm_getenv(…)
> This is the wrapper method for the **java_jnienv** object associated with the Lasso instance's Java Virtual Machine. This is the method you will use to access the Java Native Interface functions documented as member methods of **java_jnienv**.

48.2.2 Main Lasso Type

type **java_jnienv**

java_jnienv()
> This type creates an object that is used to call Java Native Interface (JNI) functions. These functions are all documented in the JNI documentation[75].

> For your convenience, the sections below are arranged in the same order and grouping as the JNI documentation.

Version

java_jnienv->GetVersion(…)
> Returns the version of the Java Native Interface.

> See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp15951 for more information.

Class Operations

java_jnienv->FindClass(…)
> Returns a reference to a Java class. It takes a string of the fully qualified class name or array type signature.

> See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp16027 for more information.

---

[74] http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp4256
[75] http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html

Exceptions

`java_jnienv->`**`Throw`**`(…)`
>   Throws a Java error (java.lang.Throwable). It takes a **`jobject`** thrown error reference and returns a **`jint`**.
>
>   See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp16086 for more information.

`java_jnienv->`**`ThrowNew`**`(…)`
>   Creates and throws a Java error with the message passed to it. It takes a **`jobject`** class reference to use to create the error, and a string with the error message. It returns a **`jint`**.
>
>   See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp16104 for more information.

`java_jnienv->`**`ExceptionOccurred`**`(…)`
>   Returns whether or not a Java exception was thrown.
>
>   See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp16124 for more information.

`java_jnienv->`**`ExceptionDescribe`**`(…)`
>   Outputs the error and stack trace for the Java exception.
>
>   See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp16146 for more information.

`java_jnienv->`**`ExceptionClear`**`(…)`
>   Clears any exceptions that have been thrown.
>
>   See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp16166 for more information.

`java_jnienv->`**`FatalError`**`(…)`
>   Throws a fatal error to the JVM. It takes a string as the error message.
>
>   See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp16186 for more information.

`java_jnienv->`**`ExceptionCheck`**`(…)`
>   Returns "true" if a Java exception has been thrown, otherwise returns "false".
>
>   See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp16270 for more information.

Global and Local References

`java_jnienv->`**`NewGlobalRef`**`(…)`
>   Creates a global reference from the specified object. It takes a **`jobject`** reference to an object and returns a new **`jobject`** global object reference.
>
>   See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#NewGlobalRef for more information.

`java_jnienv->`**`DeleteGlobalRef`**`(…)`
>   Removes the specified global reference. It takes a **`jobject`** reference to a global object.
>
>   See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#DeleteGlobalRef for more information.

`java_jnienv->`**`DeleteLocalRef`**`(…)`
>   Removes the specified local reference. It takes a **`jobject`** reference to an object.
>
>   See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#DeleteLocalRef for more information.

Object Operations

java_jnienv->AllocObject(…)
>   Allocates a Java object without calling any of the constructor methods. It takes a **jobject** class reference (like the return value of **java_jnienv->FindClass**). It returns a reference to the object.
>
>   See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp16337 for more information.

java_jnienv->NewObject(…)
>   Allocates and constructs a Java object. It takes a **jobject** class reference to the new object's class, a **jmethodid** reference to the constructor method to use, and any other parameters as required by the Java constructor method. It returns a reference to the object.
>
>   See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp4517 for more information.

java_jnienv->GetObjectClass(…)
>   Returns a class reference for the specified object. It takes a **jobject** object reference.
>
>   See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp16454 for more information.

java_jnienv->IsInstanceOf(…)
>   Returns "true" if the specified object is an instance of the specified class, otherwise returns "false". It takes a **jobject** object reference and a **jobject** class reference.
>
>   See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp16472 for more information.

java_jnienv->IsSameObject(…)
>   Returns "true" if both specified objects refer to the same Java object, otherwise returns "false". It takes two **jobject** object references.
>
>   See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp16514 for more information.

Accessing Fields of Objects

java_jnienv->GetFieldID(…)
>   Returns the field ID of a Java object's instance field. It takes a **jobject** class reference, a string with the value of the field's name, and a string of the signature for the field. It returns a **jfieldid** reference.
>
>   See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp16540 for more information.

java_jnienv->GetObjectField(…)
>   Returns the value of the specified Java object instance field. This method should be used for field values that are Java objects. It takes in a **jobject** object reference and a **jfieldid** reference and returns a **jobject** object reference.
>
>   See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp16572 for more information.

java_jnienv->GetBooleanField(…)
>   Returns the value of the specified Java object instance field. This method should be used for field values that are boolean primitives. It takes in a **jobject** object reference and a **jfieldid** reference and returns a boolean.
>
>   See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp16572 for more information.

java_jnienv->GetByteField(…)
>   Returns the value of the specified Java object instance field. This method should be used for field values that are Java byte primitives. It takes in a **jobject** object reference and a **jfieldid** reference and returns a **jbyte**.
>
>   See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp16572 for more information.

java_jnienv->GetCharField(…)
>   Returns the value of the specified Java object instance field. This method should be used for field values that are Java char primitives. It takes in a **jobject** object reference and a **jfieldid** reference and returns a **jchar**.

See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp16572 for more information.

java_jnienv->**GetShortField**(…)
> Returns the value of the specified Java object instance field. This method should be used for field values that are Java short primitives. It takes in a **jobject** object reference and a **jfieldid** reference and returns a **jshort**.
>
> See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp16572 for more information.

java_jnienv->**GetIntField**(…)
> Returns the value of the specified Java object instance field. This method should be used for field values that are Java int primitives. It takes in a **jobject** object reference and a **jfieldid** reference and returns a **jint**.
>
> See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp16572 for more information.

java_jnienv->**GetLongField**(…)
> Returns the value of the specified Java object instance field. This method should be used for field values that are Java long primitives. It takes in a **jobject** object reference and a **jfieldid** reference and returns a Lasso integer.
>
> See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp16572 for more information.

java_jnienv->**GetFloatField**(…)
> Returns the value of the specified Java object instance field. This method should be used for field values that are Java float primitives. It takes in a **jobject** object reference and a **jfieldid** reference and returns a Lasso decimal.
>
> See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp16572 for more information.

java_jnienv->**GetDoubleField**(…)
> Returns the value of the specified Java object instance field. This method should be used for field values that are Java double primitives. It takes in a **jobject** object reference and a **jfieldid** reference and returns a Lasso decimal.
>
> See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp16572 for more information.

java_jnienv->**SetObjectField**(…)
> Sets the value of the specified Java object instance field. This method should be used for fields that contain Java objects. It takes a **jobject** object reference, a **jfieldid** reference, and the new **jobject** value for the field.
>
> See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp16613 for more information.

java_jnienv->**SetBooleanField**(…)
> Sets the value of the specified Java object instance field. This method should be used for fields that contain Java boolean primitives. It takes a **jobject** object reference, a **jfieldid** reference, and the new boolean value for the field.
>
> See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp16613 for more information.

java_jnienv->**SetByteField**(…)
> Sets the value of the specified Java object instance field. This method should be used for fields that contain Java byte primitives. It takes a **jobject** object reference, a **jfieldid** reference, and the new **jbyte** value for the field.
>
> See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp16613 for more information.

java_jnienv->**SetCharField**(…)
> Sets the value of the specified Java object instance field. This method should be used for fields that contain Java char primitives. It takes a **jobject** object reference, a **jfieldid** reference, and the new **jchar** value for the field.
>
> See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp16613 for more information.

java_jnienv->**SetShortField**(…)
> Sets the value of the specified Java object instance field. This method should be used for fields that contain Java short primitives. It takes a **jobject** object reference, a **jfieldid** reference, and the new **jshort** value for the field.
>
> See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp16613 for more information.

`java_jnienv->`**`SetIntField`**`(…)`
> Sets the value of the specified Java object instance field. This method should be used for fields that contain Java int primitives. It takes a **jobject** object reference, a **jfieldid** reference, and the new **jint** value for the field.
>
> See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp16613 for more information.

`java_jnienv->`**`SetLongField`**`(…)`
> Sets the value of the specified Java object instance field. This method should be used for fields that contain Java long primitives. It takes a **jobject** object reference, a **jfieldid** reference, and the new integer value for the field.
>
> See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp16613 for more information.

`java_jnienv->`**`SetFloatField`**`(…)`
> Sets the value of the specified Java object instance field. This method should be used for fields that contain Java float primitives. It takes a **jobject** object reference, a **jfieldid** reference, and the new **jfloat** value for the field.
>
> See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp16613 for more information.

`java_jnienv->`**`SetDoubleField`**`(…)`
> Sets the value of the specified Java object instance field. This method should be used for fields that contain Java double primitives. It takes a **jobject** object reference, a **jfieldid** reference, and the new decimal value for the field.
>
> See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp16613 for more information.

Calling Instance Methods

`java_jnienv->`**`GetMethodID`**`(…)`
> Returns a **jmethodid** Lasso object for the Java object's specified instance member method. For constructor methods, use "<init>" as the method name.
>
> See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp16660 for more information.

`java_jnienv->`**`CallVoidMethod`**`(…)`
> Calls the specified Java instance method with the expected parameters passed as the remaining Lasso parameters to this method. This method should be used when the method doesn't return a value. It takes a **jobject** object reference, a **jmethodid**, and any parameters to be passed to the instance method.
>
> See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp4256 for more information.

`java_jnienv->`**`CallObjectMethod`**`(…)`
> Calls the specified Java instance method with the expected parameters passed as the remaining Lasso parameters to this method. This method should be used when the return value will be a Java object returned as a Lasso **jobject** object reference. It takes a **jobject** object reference, a **jmethodid**, and any parameters to be passed to the instance method.
>
> See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp4256 for more information.

`java_jnienv->`**`CallBooleanMethod`**`(…)`
> Calls the specified Java instance method with the expected parameters passed as the remaining Lasso parameters to this method. This method should be used when the return value will be a boolean value. It takes a **jobject** object reference, a **jmethodid**, and any parameters to be passed to the instance method.
>
> See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp4256 for more information.

`java_jnienv->`**`CallByteMethod`**`(…)`
> Calls the specified Java instance method with the expected parameters passed as the remaining Lasso parameters to this method. This method should be used when the return value will be a Java byte primitive. It takes a **jobject** object reference, a **jmethodid**, and any parameters to be passed to the instance method.
>
> See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp4256 for more information.

java_jnienv->**CallCharMethod**(…)
> Calls the specified Java instance method with the expected parameters passed as the remaining Lasso parameters to this method. This method should be used when the return value will be a Java char primitive. It takes a **jobject** object reference, a **jmethodid**, and any parameters to be passed to the instance method.
>
> See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp4256 for more information.

java_jnienv->**CallShortMethod**(…)
> Calls the specified Java instance method with the expected parameters passed as the remaining Lasso parameters to this method. This method should be used when the return value will be a Java short primitive. It takes a **jobject** object reference, a **jmethodid**, and any parameters to be passed to the instance method.
>
> See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp4256 for more information.

java_jnienv->**CallIntMethod**(…)
> Calls the specified Java instance method with the expected parameters passed as the remaining Lasso parameters to this method. This method should be used when the return value will be a Java int primitive. It takes a **jobject** object reference, a **jmethodid**, and any parameters to be passed to the instance method.
>
> See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp4256 for more information.

java_jnienv->**CallLongMethod**(…)
> Calls the specified Java instance method with the expected parameters passed as the remaining Lasso parameters to this method. This method should be used when the return value will be a Java long primitive. It takes a **jobject** object reference, a **jmethodid**, and any parameters to be passed to the instance method.
>
> See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp4256 for more information.

java_jnienv->**CallFloatMethod**(…)
> Calls the specified Java instance method with the expected parameters passed as the remaining Lasso parameters to this method. This method should be used when the return value will be a Java float primitive. It takes a **jobject** object reference, a **jmethodid**, and any parameters to be passed to the instance method.
>
> See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp4256 for more information.

java_jnienv->**CallDoubleMethod**(…)
> Calls the specified Java instance method with the expected parameters passed as the remaining Lasso parameters to this method. This method should be used when the return value will be a Java double primitive. It takes a **jobject** object reference, a **jmethodid**, and any parameters to be passed to the instance method.
>
> See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp4256 for more information.

java_jnienv->**CallNonvirtualVoidMethod**(…)
> Calls the specified Java instance method with the expected parameters passed as the remaining Lasso parameters to this method. This method should be used when there will be no return value. It takes a **jobject** object reference, a **jobject** class reference, a **jmethodid**, and any parameters to be passed to the instance method.
>
> See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp4581 for more information.

java_jnienv->**CallNonvirtualObjectMethod**(…)
> Calls the specified Java instance method with the expected parameters passed as the remaining Lasso parameters to this method. This method should be used when the return value will be a Java object. It takes a **jobject** object reference, a **jobject** class reference, a **jmethodid**, and any parameters to be passed to the instance method.
>
> See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp4581 for more information.

java_jnienv->**CallNonvirtualBooleanMethod**(…)
> Calls the specified Java instance method with the expected parameters passed as the remaining Lasso parameters to this method. This method should be used when the return value will be a boolean. It takes a **jobject** object reference, a **jobject** class reference, a **jmethodid**, and any parameters to be passed to the instance method.
>
> See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp4581 for more information.

java_jnienv->**CallNonvirtualByteMethod**(…)
>   Calls the specified Java instance method with the expected parameters passed as the remaining Lasso parameters to this method. This method should be used when the return value will be a Java byte primitive. It takes a **jobject** object reference, a **jobject** class reference, a **jmethodid**, and any parameters to be passed to the instance method.
>
>   See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp4581 for more information.

java_jnienv->**CallNonvirtualCharMethod**(…)
>   Calls the specified Java instance method with the expected parameters passed as the remaining Lasso parameters to this method. This method should be used when the return value will be a Java char primitive. It takes a **jobject** object reference, a **jobject** class reference, a **jmethodid**, and any parameters to be passed to the instance method.
>
>   See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp4581 for more information.

java_jnienv->**CallNonvirtualShortMethod**(…)
>   Calls the specified Java instance method with the expected parameters passed as the remaining Lasso parameters to this method. This method should be used when the return value will be a Java short primitive. It takes a **jobject** object reference, a **jobject** class reference, a **jmethodid**, and any parameters to be passed to the instance method.
>
>   See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp4581 for more information.

java_jnienv->**CallNonvirtualIntMethod**(…)
>   Calls the specified Java instance method with the expected parameters passed as the remaining Lasso parameters to this method. This method should be used when the return value will be a Java int primitive. It takes a **jobject** object reference, a **jobject** class reference, a **jmethodid**, and any parameters to be passed to the instance method.
>
>   See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp4581 for more information.

java_jnienv->**CallNonvirtualLongMethod**(…)
>   Calls the specified Java instance method with the expected parameters passed as the remaining Lasso parameters to this method. This method should be used when the return value will be a Java long primitive. It takes a **jobject** object reference, a **jobject** class reference, a **jmethodid**, and any parameters to be passed to the instance method.
>
>   See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp4581 for more information.

java_jnienv->**CallNonvirtualFloatMethod**(…)
>   Calls the specified Java instance method with the expected parameters passed as the remaining Lasso parameters to this method. This method should be used when the return value will be a Java float primitive. It takes a **jobject** object reference, a **jobject** class reference, a **jmethodid**, and any parameters to be passed to the instance method.
>
>   See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp4581 for more information.

java_jnienv->**CallNonvirtualDoubleMethod**(…)
>   Calls the specified Java instance method with the expected parameters passed as the remaining Lasso parameters to this method. This method should be used when the return value will be a Java double primitive. It takes a **jobject** object reference, a **jobject** class reference, a **jmethodid**, and any parameters to be passed to the instance method.
>
>   See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp4581 for more information.

Accessing Static Fields

java_jnienv->**GetStaticFieldID**(…)
>   Returns a **jfieldid** reference to a Java class's static field. It takes a **jobject** class reference, a string with the value of the field's name, and a string of the signature for the field.
>
>   See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp16823 for more information.

java_jnienv->**GetStaticObjectField**(…)
>   Returns the value of the specified Java class static field. This method should be used for field values that are Java objects. It takes in a **jobject** class reference and a **jfieldid** reference and returns a **jobject** object reference.

See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp20752 for more information.

java_jnienv->**GetStaticBooleanField**(…)
> Returns the value of the specified Java class static field. This method should be used for field values that are boolean primitives. It takes in a **jobject** class reference and a **jfieldid** reference and returns a boolean.
>
> See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp20752 for more information.

java_jnienv->**GetStaticByteField**(…)
> Returns the value of the specified Java class static field. This method should be used for field values that are Java byte primitives. It takes in a **jobject** class reference and a **jfieldid** reference and returns a **jbyte**.
>
> See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp20752 for more information.

java_jnienv->**GetStaticCharField**(…)
> Returns the value of the specified Java class static field. This method should be used for field values that are Java char primitives. It takes in a **jobject** class reference and a **jfieldid** reference and returns a **jchar**.
>
> See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp20752 for more information.

java_jnienv->**GetStaticShortField**(…)
> Returns the value of the specified Java class static field. This method should be used for field values that are Java short primitives. It takes in a **jobject** class reference and a **jfieldid** reference and returns a **jshort**.
>
> See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp20752 for more information.

java_jnienv->**GetStaticIntField**(…)
> Returns the value of the specified Java class static field. This method should be used for field values that are Java int primitives. It takes in a **jobject** class reference and a **jfieldid** reference and returns a **jint**.
>
> See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp20752 for more information.

java_jnienv->**GetStaticLongField**(…)
> Returns the value of the specified Java class static field. This method should be used for field values that are Java long primitives. It takes in a **jobject** class reference and a **jfieldid** reference and returns a Lasso integer.
>
> See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp20752 for more information.

java_jnienv->**GetStaticFloatField**(…)
> Returns the value of the specified Java class static field. This method should be used for field values that are Java float primitives. It takes in a **jobject** class reference and a **jfieldid** reference and returns a Lasso decimal.
>
> See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp20752 for more information.

java_jnienv->**GetStaticDoubleField**(…)
> Returns the value of the specified Java class static field. This method should be used for field values that are Java double primitives. It takes in a **jobject** class reference and a **jfieldid** reference and returns a Lasso decimal.
>
> See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp20752 for more information.

java_jnienv->**SetStaticObjectField**(…)
> Sets the value of the specified Java class static field. This method should be used for fields that contain Java objects. It takes a **jobject** class reference, a **jfieldid** reference, and the new **jobject** value for the field.
>
> See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp20829 for more information.

java_jnienv->**SetStaticBooleanField**(…)
> Sets the value of the specified Java class static field. This method should be used for fields that contain Java boolean primitives. It takes a **jobject** class reference, a **jfieldid** reference, and the new boolean value for the field.
>
> See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp20829 for more information.

`java_jnienv->`**`SetStaticByteField`**`(…)`
>   Sets the value of the specified Java class static field. This method should be used for fields that contain Java byte primitives. It takes a **jobject** class reference, a **jfieldid** reference, and the new **jbyte** value for the field.
>
>   See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp20829 for more information.

`java_jnienv->`**`SetStaticCharField`**`(…)`
>   Sets the value of the specified Java class static field. This method should be used for fields that contain Java char primitives. It takes a **jobject** class reference, a **jfieldid** reference, and the new **jchar** value for the field.
>
>   See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp20829 for more information.

`java_jnienv->`**`SetStaticShortField`**`(…)`
>   Sets the value of the specified Java class static field. This method should be used for fields that contain Java short primitives. It takes a **jobject** class reference, a **jfieldid** reference, and the new **jshort** value for the field.
>
>   See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp20829 for more information.

`java_jnienv->`**`SetStaticIntField`**`(…)`
>   Sets the value of the specified Java class static field. This method should be used for fields that contain Java int primitives. It takes a **jobject** class reference, a **jfieldid** reference, and the new **jint** value for the field.
>
>   See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp20829 for more information.

`java_jnienv->`**`SetStaticLongField`**`(…)`
>   Sets the value of the specified Java class static field. This method should be used for fields that contain Java long primitives. It takes a **jobject** class reference, a **jfieldid** reference, and the new integer value for the field.
>
>   See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp20829 for more information.

`java_jnienv->`**`SetStaticFloatField`**`(…)`
>   Sets the value of the specified Java class static field. This method should be used for fields that contain Java float primitives. It takes a **jobject** class reference, a **jfieldid** reference, and the new **jfloat** value for the field.
>
>   See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp20829 for more information.

`java_jnienv->`**`SetStaticDoubleField`**`(…)`
>   Sets the value of the specified Java class static field. This method should be used for fields that contain Java double primitives. It takes a **jobject** class reference, a **jfieldid** reference, and the new decimal value for the field.
>
>   See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp20829 for more information.

Calling Static Methods

`java_jnienv->`**`GetStaticMethodID`**`(…)`
>   Returns a **jmethodid** Lasso object for the specified static method. It takes a **jobject** class reference, a string specifying the name of the method, and a string of the method's signature.
>
>   See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp20950 for more information.

`java_jnienv->`**`CallStaticVoidMethod`**`(…)`
>   This method is used to call a Java class static method that doesn't return a value. It takes a **jobject** class reference, a **jmethodid** for the method, and any parameters to be passed to the static method.
>
>   See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp4796 for more information.

`java_jnienv->`**`CallStaticObjectMethod`**`(…)`
>   This method is used to call a Java class static method that returns a Java object. It takes a **jobject** class reference, a **jmethodid** for the method, and any parameters to be passed to the static method.
>
>   See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp4796 for more information.

java_jnienv->**CallStaticBooleanMethod**(…)
> This method is used to call a Java class static method that returns a Java boolean. It takes a **jobject** class reference, a **jmethodid** for the method, and any parameters to be passed to the static method.
>
> See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp4796 for more information.

java_jnienv->**CallStaticByteMethod**(…)
> This method is used to call a Java class static method that returns a Java byte primitive. It takes a **jobject** class reference, a **jmethodid** for the method, and any parameters to be passed to the static method.
>
> See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp4796 for more information.

java_jnienv->**CallStaticCharMethod**(…)
> This method is used to call a Java class static method that returns a Java char primitive. It takes a **jobject** class reference, a **jmethodid** for the method, and any parameters to be passed to the static method.
>
> See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp4796 for more information.

java_jnienv->**CallStaticShortMethod**(…)
> This method is used to call a Java class static method that returns a Java short primitive. It takes a **jobject** class reference, a **jmethodid** for the method, and any parameters to be passed to the static method.
>
> See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp4796 for more information.

java_jnienv->**CallStaticIntMethod**(…)
> This method is used to call a Java class static method that returns a Java int primitive. It takes a **jobject** class reference, a **jmethodid** for the method, and any parameters to be passed to the static method.
>
> See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp4796 for more information.

java_jnienv->**CallStaticLongMethod**(…)
> This method is used to call a Java class static method that returns a Java long primitive. It takes a **jobject** class reference, a **jmethodid** for the method, and any parameters to be passed to the static method.
>
> See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp4796 for more information.

java_jnienv->**CallStaticFloatMethod**(…)
> This method is used to call a Java class static method that returns a Java float primitive. It takes a **jobject** class reference, a **jmethodid** for the method, and any parameters to be passed to the static method.
>
> See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp4796 for more information.

java_jnienv->**CallStaticDoubleMethod**(…)
> This method is used to call a Java class static method that returns a Java double primitive. It takes a **jobject** class reference, a **jmethodid** for the method, and any parameters to be passed to the static method.
>
> See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp4796 for more information.

String Operations

java_jnienv->**NewString**(…)
> Takes in a Lasso string and returns a Lasso **jobject** that corresponds to a Java object of class `java.lang.String`.
>
> See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp4925 for more information.

java_jnienv->**GetStringLength**(…)
> Returns the number of characters in the specified Java string object.
>
> See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp17132 for more information.

java_jnienv->**GetStringChars**(…)
> Takes in a **jobject** of a Java string and returns a Lasso string object.

See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp17158 for more information.

Array Operations

`java_jnienv->`**`GetArrayLength`**`(…)`
    Returns the number of elements in the specified Java array.

    See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp21732 for more information.

`java_jnienv->`**`NewObjectArray`**`(…)`
    Returns a **jobject** of a Java array containing Java objects of the specified class. It takes the length of the array, a **jobject** class reference for the type of objects in the array, and the initial value to set each item in the array to.

    See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp21619 for more information.

`java_jnienv->`**`GetObjectArrayElement`**`(…)`
    Returns the specified element of a Java object array. It takes the **jobject** containing the array and an integer specifying the index into the array.

    See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp21671 for more information.

`java_jnienv->`**`SetObjectArrayElement`**`(…)`
    Sets the value at the specified index of the specified Java object array. It takes a **jobject** of the array, an integer specifying the index into the array, and the new **jobject** object.

    See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp21699 for more information.

`java_jnienv->`**`NewBooleanArray`**`(…)`
    Returns a **jobject** of a Java array containing Java booleans. It takes the length of the array.

    See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp17318 for more information.

`java_jnienv->`**`NewByteArray`**`(…)`
    Returns a **jobject** of a Java array containing Java byte primitives. It takes the length of the array.

    See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp17318 for more information.

`java_jnienv->`**`NewCharArray`**`(…)`
    Returns a **jobject** of a Java array containing Java char primitives. It takes the length of the array.

    See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp17318 for more information.

`java_jnienv->`**`NewShortArray`**`(…)`
    Returns a **jobject** of a Java array containing Java short primitives. It takes the length of the array.

    See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp17318 for more information.

`java_jnienv->`**`NewIntArray`**`(…)`
    Returns a **jobject** of a Java array containing Java int primitives. It takes the length of the array.

    See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp17318 for more information.

`java_jnienv->`**`NewLongArray`**`(…)`
    Returns a **jobject** of a Java array containing Java long primitives. It takes the length of the array.

    See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp17318 for more information.

`java_jnienv->`**`NewFloatArray`**`(…)`
    Returns a **jobject** of a Java array containing Java float primitives. It takes the length of the array.

    See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp17318 for more information.

java_jnienv->**NewDoubleArray**(…)
> Returns a **jobject** of a Java array containing Java double primitives. It takes the length of the array.
>
> See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp17318 for more information.

java_jnienv->**GetBooleanArrayElements**(…)
> Takes a **jobject** Java boolean array and returns a Lasso staticarray of the elements.
>
> See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp17382 for more information.

java_jnienv->**GetByteArrayElements**(…)
> Takes a **jobject** Java byte array and returns a Lasso staticarray of the elements.
>
> See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp17382 for more information.

java_jnienv->**GetCharArrayElements**(…)
> Takes a **jobject** Java char array and returns a Lasso staticarray of the elements.
>
> See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp17382 for more information.

java_jnienv->**GetShortArrayElements**(…)
> Takes a **jobject** Java short array and returns a Lasso staticarray of the elements.
>
> See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp17382 for more information.

java_jnienv->**GetIntArrayElements**(…)
> Takes a **jobject** Java int array and returns a Lasso staticarray of the elements.
>
> See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp17382 for more information.

java_jnienv->**GetLongArrayElements**(…)
> Takes a **jobject** Java long array and returns a Lasso staticarray of the elements.
>
> See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp17382 for more information.

java_jnienv->**GetFloatArrayElements**(…)
> Takes a **jobject** Java float array and returns a Lasso staticarray of the elements.
>
> See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp17382 for more information.

java_jnienv->**GetDoubleArrayElements**(…)
> Takes a **jobject** Java double array and returns a Lasso staticarray of the elements.
>
> See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp17382 for more information.

java_jnienv->**GetBooleanArrayRegion**(…)
> Returns the specified region of elements from a Java boolean array in a Lasso staticarray. It takes a **jobject** of the array, an integer for the start index of the array region, and an integer specifying the number of elements.
>
> See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp6212 for more information.

java_jnienv->**GetByteArrayRegion**(…)
> Returns the specified region of elements from a Java byte array in a Lasso staticarray. It takes a **jobject** of the array, an integer for the start index of the array region, and an integer specifying the number of elements.
>
> See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp6212 for more information.

java_jnienv->**GetCharArrayRegion**(…)
> Returns the specified region of elements from a Java char array in a Lasso staticarray. It takes a **jobject** of the array, an integer for the start index of the array region, and an integer specifying the number of elements.
>
> See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp6212 for more information.

java_jnienv->**GetShortArrayRegion**(…)
> Returns the specified region of elements from a Java short array in a Lasso staticarray. It takes a **jobject** of the array, an integer for the start index of the array region, and an integer specifying the number of elements.

See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp6212 for more information.

java_jnienv->**GetIntArrayRegion**(...)

Returns the specified region of elements from a Java int array in a Lasso staticarray. It takes a **jobject** of the array, an integer for the start index of the array region, and an integer specifying the number of elements.

See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp6212 for more information.

java_jnienv->**GetLongArrayRegion**(...)

Returns the specified region of elements from a Java long array in a Lasso staticarray. It takes a **jobject** of the array, an integer for the start index of the array region, and an integer specifying the number of elements.

See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp6212 for more information.

java_jnienv->**GetFloatArrayRegion**(...)

Returns the specified region of elements from a Java float array in a Lasso staticarray. It takes a **jobject** of the array, an integer for the start index of the array region, and an integer specifying the number of elements.

See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp6212 for more information.

java_jnienv->**GetDoubleArrayRegion**(...)

Returns the specified region of elements from a Java double array in a Lasso staticarray. It takes a **jobject** of the array, an integer for the start index of the array region, and an integer specifying the number of elements.

See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp6212 for more information.

java_jnienv->**SetBooleanArrayRegion**(...)

Replaces the specified portion of a Java boolean array with the values specified in a Lasso staticarray. It takes a **jobject** of the array, an integer for the start index of the array region, an integer specifying the number of elements to replace, and a staticarray containing the values to use.

See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp22933 for more information.

java_jnienv->**SetByteArrayRegion**(...)

Replaces the specified portion of a Java byte array with the values specified in a Lasso staticarray. It takes a **jobject** of the array, an integer for the start index of the array region, an integer specifying the number of elements to replace, and a staticarray containing the values to use.

See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp22933 for more information.

java_jnienv->**SetCharArrayRegion**(...)

Replaces the specified portion of a Java char array with the values specified in a Lasso staticarray. It takes a **jobject** of the array, an integer for the start index of the array region, an integer specifying the number of elements to replace, and a staticarray containing the values to use.

See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp22933 for more information.

java_jnienv->**SetShortArrayRegion**(...)

Replaces the specified portion of a Java short array with the values specified in a Lasso staticarray. It takes a **jobject** of the array, an integer for the start index of the array region, an integer specifying the number of elements to replace, and a staticarray containing the values to use.

See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp22933 for more information.

java_jnienv->**SetIntArrayRegion**(...)

Replaces the specified portion of a Java int array with the values specified in a Lasso staticarray. It takes a **jobject** of the array, an integer for the start index of the array region, an integer specifying the number of elements to replace, and a staticarray containing the values to use.

See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp22933 for more information.

java_jnienv->**SetLongArrayRegion**(...)

Replaces the specified portion of a Java long array with the values specified in a Lasso staticarray. It takes a **jobject**

of the array, an integer for the start index of the array region, an integer specifying the number of elements to replace, and a staticarray containing the values to use.

See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp22933 for more information.

java_jnienv->**SetFloatArrayRegion**(…)
:   Replaces the specified portion of a Java float array with the values specified in a Lasso staticarray. It takes a **jobject** of the array, an integer for the start index of the array region, an integer specifying the number of elements to replace, and a staticarray containing the values to use.

    See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp22933 for more information.

java_jnienv->**SetDoubleArrayRegion**(…)
:   Replaces the specified portion of a Java double array with the values specified in a Lasso staticarray. It takes a **jobject** of the array, an integer for the start index of the array region, an integer specifying the number of elements to replace, and a staticarray containing the values to use.

    See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp22933 for more information.

Monitor Operations

java_jnienv->**MonitorEnter**(…)
:   Enters into the monitor associated with the specified Java object. Requires a non-null **jobject** object.

    See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp23124 for more information.

java_jnienv->**MonitorExit**(…)
:   Decrements the monitor counter for the current thread and the specified Java object. Requires a non-null **jobject** object.

    See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp5252 for more information.

Reflection Support

java_jnienv->**FromReflectedMethod**(…)
:   Converts a specified Java reflection object into a Lasso **jmethodid**.

    See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#from_reflected_method for more information.

java_jnienv->**FromReflectedField**(…)
:   Converts a specified Java reflection field object into a Lasso **jfieldid**.

    See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#from_reflected_field for more information.

java_jnienv->**ToReflectedMethod**(…)
:   Converts a specified Lasso **jmethodid** to a Java reflection object returned as a **jobject**.

    See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#to_reflected_method for more information.

java_jnienv->**ToReflectedField**(…)
:   Converts a specified Lasso **jfieldid** to a Java reflection field object returned as a **jobject**.

    See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#to_reflected_field for more information.

48.2.3 Return Types

type **jobject**

jobject()
>   Stores a reference to either a Java class, instantiated object, or thrown error.
>
>   See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/types.html#wp15954 for more information.

type **jmethodid**

jmethodid()
>   Stores the JNI ID for a specific method (both member methods and class methods).
>
>   See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/types.html#wp1064 for more information.

type **jfieldid**

jfieldid()
>   Stores the JNI ID for data field members of a class (both an object's and the class's).
>
>   See http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/types.html#wp1064 for more information.

48.2.4 Helper Types for Java Data Primitives

type **jfloat**

jfloat(*value::decimal*)

jfloat(*value::integer*)

jfloat(*value::jfloat*)
>   Creates an object that can be passed to a Java method as a Java float primitive.

type **jint**

jint(*value::integer*)
>   Creates an object that can be passed to a Java method as a Java integer primitive.

type **jshort**

jshort(*value::integer*)
>   Creates an object that can be passed to a Java method as a Java short primitive.

type **jchar**

jchar(*value::string*)
>   Creates an object that can be passed to a Java method as a Java char primitive.

type **jchararray**

jchararray(*value::string*)
>   Creates an object that can be passed to a Java method as a Java array of char primitives.

type **jbyte**

jbyte(*value::bytes*)
>   Creates an object that can be passed to a Java method as a Java byte primitive.

type **jbytearray**

jbytearray(*value::bytes*)
>   Creates an object that can be passed to a Java method as a Java array of byte primitives.

Index

D

F

G