Exploring Lift

Derek Chen-Becker, Marius Danciu and Tyler Weir

December 11, 2008

List of Figures

Dedication

Derek would like to thank his wife, Debbie, for her patience and support while writing this book.

Tyler would like to thank his wife Laura, for encouraging him.

Acknowledgements

This book would not have been possible without the Lift Developers and especially David Pollak, without him, we wouldn't have this opportunity.

Part I

The Basics

Chapter 1

Welcome to Lift!

Welcome to *Exploring Lift*. We've created this book to educate you about Lift, which we think is a great framework for building compelling web applications. Lift is designed to make powerful techniques easily accessible, while keeping the overall framework simple and flexible. It may sound like a cliché, but in our experience Lift makes it fun to develop because it lets you focus on the interesting parts of coding. Our goal for this book is that by the end, you'll be able to create and extend any web application you can think of.

1.1 Why Lift?

For those of you have experience with other web frameworks such as Struts, Tapestry, Rails, et cetera, you must be asking yourself "Why another framework? Does Lift really solve problems any differently or more effectively than the ones I've used before?" Based on our experience (and of others in the growing Lift community), the answer is an emphatic "Yes!" Lift has cherry-picked the best ideas from a number of other frameworks, while creating some novel ideas of its own. It's this combination of solid foundation and new techniques that makes Lift so powerful. At the same time, Lift has been able to avoid the mistakes made in the past by other frameworks. In the spirit of "configuration by convention", Lift has sensible defaults for everything, while making it easy to customize precisely what you need to; no more and no less. Gone are the days of XML file after XML file providing *basic configuration* for your application. Instead, a basic Lift app only requires that you add the LiftFilter to your web.xml and add one or more lines telling Lift what package your classes sit in . The methods you code aren't required to implement a specific interface (called a trait), although there are support traits that make things that much simpler. In short, you don't need to write anything that isn't explicitly necessary for the task at hand; Lift is intended to work out of the box, and to make you as efficient and productive as possible.

One of the key strengths of Lift is the clean separation of presentation content and logic, based on the bedrock concept of the Model-View-Controller pattern[1]. One of the original Java web application technologies that's still in use today is JSP, or Java Server Pages[2]. JSP allows you to mix HTML and Java code directly within the page. While this may have seemed like a good idea at the

[1] http://heim.ifi.uio.no/~trygver/1979/mvc-2/1979-12-MVC.pdf
[2] http://java.sun.com/products/jsp/

start, it has proven to be painful in practice. Putting code in your presentation layer makes it more difficult to debug and understand what is going on within a page, and makes it more difficult for the people writing the HTML portion because the contents aren't valid HTML. While many modern programming and HTML editors have been modified to accomodate this mess, proper syntax highlighting and validation don't make up for the fact that you still have to switch back and forth between one or more files to follow the page flow. Lift takes the approach that there should be no code in the presentation layer, but that the presentation layer has to be flexible enough to accomodate any conceivable uses. To that end, Lift uses a powerful templating system, a la Wicket[3], to bind user-generated data into the presentation layer. Lift's templating is built on the XML processing capabilities of the Scala Language[4], and allows things such as nested templates, simple injection of user-generated content, and advanced data binding capabilities. For those coming from JSP, Lift's advanced template and XML processing allows you to essentially write custom tag libraries at a fraction of the cost in time and effort.

Lift has another advantage that no other web framework currently shares: the Scala programming language. Scala is a relatively new language developed by Martin Odersky[5] and his group of fearless rogues at EPFL Switzerland. It compiles to Java bytecode and runs on the JVM, which means that you can leverage the vast ecosystem of Java libraries just as you would with any other java web framework. At the same time, Scala introduces some very powerful features designed to make you, the developer, more productive. Among these features are an extremely rich type system along with powerful type inference, native XML processing, full support for closures and functions as objects, and an extensive high-level library. The power of the type system along with its type inferencing has led people to call it "the statically typed dynamic language"[6]. That means you can write code as quickly as you could with dynamically typed languages (Python, Ruby, etc.), but you have the compile-time type safety of a statically typed language like Java. Scala is also a hybrid functional and Object-oriented language, which means you can get the power of the higher-level functional (or FP) languages (such as Haskell, Scheme, etc) while retaining the modularity and reusability of OO components. In particular, the FP concept of immutability is well represented in Scala, and is one of the simplest means to high throughput scalability. The hybrid model also means that if you haven't touched FP before, you can gradually ease into it. In our experience, Scala allows you to do more in Lift with less lines of code; remember, Lift is all about making you more productive!

Lift strives to encompass advanced features in a very concise and straightforward manner. Lift's powerful support for AJAX and COMET allow you to use Web 2.0 features with very little effort. Lift leverages Scala's Actor library to provide a message-driven framework for COMET updates. In most cases, adding COMET support to a page just involves extending a trait[7] to define the rendering method of your page and adding an extra function call to your links to dispatch the update message; Lift handles all of the backend and page-side coding to effect the COMET polling. AJAX support includes special handlers for doing AJAX form submission via JSON, and almost any link function can easily be turned into an AJAX version with a few keystrokes. In order

[3] http://wicket.apache.org/

[4] Not only does Scala have some extensive library support for XML, but XML syntax is actually part of the language. We'll cover this in more detail as we go through the book

[5] Martin wrote the original `javac` compiler for Sun, and more recently led the Pizza and GJ projects that eventually became Java Generics. His home page is at http://lamp.epfl.ch/~odersky/

[6] http://scala-blogs.org/2007/12/scala-statically-typed-dynamic-language.html

[7] A trait is a Scala construct that's essentially like a Java interface. The main difference is that traits are allowed to provide method bodies and may have variables.

to preform all of this client-side goodness, Lift has a class hierarchy for encapsulating Javascript calls via direct Javascript, jQuery and now YUI. The nice part is that you, too, can utilize these support classes so that you can generate the code and don't have to put Javascript (i.e. logic) into your templates.

1.2 For more information

Lift has a very active community of users and developers. Since its inception in early 2007 the community has grown to hundreds of members from all over the world. The project's leader, David Pollak[8]is constantly attending to the mailing list, answering questions and taking feature requests. There is a core group of developers who work on the project, but submissions are taken from anyone who makes a good case and can turn in good code. While we strive to cover everything you'll need to know in this book, here are several additional resources available for information on Lift:

1. The first place to look is the Wiki at http://liftweb.net/index.php/Main_Page. The Wiki is maintained not only by David, but by many active members of the Lift community (including the authors). Portions of this book as inspired by and borrow from content on the Wiki. In particular, it has links to all of the generated documentation not only for the stable branch, but for the unstable head if you're feeling adventurous. There's also an extensive section of HowTos and articles on advanced topics that covers a wealth of information.

2. The mailing list at http://groups.google.com/group/liftweb is very active and if there are things that this book doesn't cover you can feel free to ask questions there; there are plenty of very knowledgeable people on the list that should be able to answer your questions. Please send specific questions about the book to ..., but anything else Lift-specific is fair game for the mailing list.

3. Lift has an IRC channel at irc://irc.freenode.net/lift that usually has several people on at any given time. It's a great place to chat about issues and ideas concerning Lift.

1.3 Your first Lift application

We've talked a lot about Lift and its capabilities, so now let's get hands-on and try out an application. Before we start, though, we need to take care of some prerequisites:

Java 1.5 JDK Lift runs on Scala, which runs on top of the JVM; the first thing you'll need to install is a modern version of the Java SE JVM, available at http://java.sun.com/. Recently Scala's compiler was changed to target Java version 1.5; 1.4 is still available as a target, but we're going to assume you're using 1.5. Examples in this book have only been tested with Sun's version of the JDK, although most likely other versions (Blackdown, OpenJDK) should work with little or no modification.

Maven 2 Maven is a project management tool which we'll cover in more detail in chapter 2 on page 11. Maven has extensive capabilities for building, dependency management, testing and reporting. If you haven't used Maven before you can think of it as an incredibly

[8] http://blog.lostlake.org/

powerful version of `make` for now. You can download the latest version of Maven from http://maven.apache.org/. Brief installation instructions (enough to get us started) are on the download page, at http://maven.apache.org/download.html.

Programming editor This isn't a strict requirement for this example, but when we start getting into coding it's very helpful to have something a little more capable than notepad. If you'd like a full-blown IDE, with support for things like debugging, continuous compile checking, etc, there are plugins available on the Scala website at http://www.scala-lang.org/node/91. The plugins support:

Eclipse	http://www.eclipse.org/ The Scala plugin developer recommends using the *Eclipse Classic* version of the IDE
NetBeans	http://www.netbeans.org Requires using the NetBeans 6.5 Beta (at the time of writing)
IntelliJ	IDEA http://www.jetbrains.com/idea/index.html Requires version 8 beta

If you'd like something more lightweight, the Scala language distribution comes with plugins for editors like VIM, Emacs, jedit, etc. You can either download the full Scala distribution from http://www.scala-lang.org/ and use the files under `misc/scala-tool-support`, or you can directly access the latest versions via the SVN (Subversion) interface at https://lampsvn.epfl.ch/trac/scala/browser/scala-tool-support/trunk/src. Getting these plugins to work in your IDE or editor of choice is beyond the scope of this book.

Now that we have the prerequisites out of the way, it's time to get started. We're going to leverage Maven's archetypes to do 99% of the work for us in this example. First, change to whatever directory you'd like to work in:

```
cd work
```

Next, we use Maven's `archetype:generate` command to create the skeleton of our project:

```
mvn archetype:generate -U \
  -DarchetypeGroupId=net.liftweb \
  -DarchetypeArtifactId=lift-archetype-blank \
  -DarchetypeVersion=0.9 \
  -DremoteRepositories=http://scala-tools.org/repo-releases \
  -DgroupId=demo.helloworld \
  -DartifactId=helloworld \
  -Dversion=1.0-SNAPSHOT
```

Maven should dump several page's worth of output. It may stop and ask you to confirm the properties configuration, in which case you can just hit `<enter>`. At the end you should get a message that says `BUILD SUCCESSFUL`. You've now successfully created your first project! Don't believe us? Let's run it to confirm:

```
cd helloworld
mvn jetty:run
```

Maven should produce more output, ending with

```
[INFO] Starting scanner at interval of 5 seconds.
```

This means that you now have a web server (Jetty[9]) running on port 8080 of your machine. Just go to http://localhost:8080/ and you'll see your first Lift page, the standard "Hello world!". With just a few simple commands we've built a functional (albeit limited) web app. Let's go into a little more detail and see exactly how these pieces fit together. First, let's examine the index page. Whenever Lift serves up a request where the URL ends in a forward slash, Lift automatically looks for a file called index.html[10] in that directory. For instance, if you tried to go to `http://localhost:8080/test/`, Lift would look for index.html under the `test/` directory in your project. The HTML sources will be located under `src/main/webapp/` in your project directory Here's the `index.html` file from our Hello World project:

```
<lift:surround with="default" at="content">
  <h2>Welcome to your project!</h2>
  <p><lift:helloWorld.howdy /></p>
</lift:surround>
```

This may look a little strange at first. For those with some XML experience you may recognize the use of prefixed elements here. For those who don't know what that is, a prefixed element is an XML element of the form

```
<prefix:element>
```

In our case we have two elements in use: `<lift:surround>` and `<lift:helloWorld.howdy/>`. Lift assigns special meaning to elements that use the "lift" prefix; they form the basis of lift's extensive templating support, which we will cover in more detail in . When lift processes an XML template, it does so from the outermost element inward. In our case, the outermost element is `<lift:surround with=''default'' at=''content''>`. The `<lift:surround>` element basically tells Lift to find the template named by the *with* attribute (*default*, in our case) and to put the contents of our element inside of that template. The *at* attribute tells Lift where in the template to place our content. In Lift, this "filling in the blanks" is called *binding*, and it's a fundamental concept of Lift's template system. Just about everything at the HTML/XML level can be thought of as a series of nested binds. Before we move on to the `<lift:helloWorld.howdy/>` element, let's recurse and look at the default template. You can find it in the `templates-hidden` directory of the web app. Much like the `WEB-INF` and `META-INF` directories in a Java web application, the contents of `templates-hidden` cannot be accessed directly by clients; they can, however be accessed when they're referenced by a `<lift:surround>` element. Here is the `default.html` file:

```
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:lift="http://liftweb.net/">
  <head>
    <meta http-equiv="content-type" content="text/html; charset=UTF-8" />
    <meta name="description" content="" />
    <meta name="keywords" content="" />

    <title>demo.helloworld:helloworld:1.0-SNAPSHOT</title>
```

[9] http://www.mortbay.org/jetty/

[10] Technically, it also searches for some variations on index.html, including any localized versions of the page, but we'll cover that later in section

```
    <script id="jquery" src="/classpath/jquery.js" type="text/javascript"></script>
  </head>
  <body>
    <lift:bind name="content" />
    <lift:Menu.builder />
    <lift:msgs/>
  </body>
</html>
```

As you can see in the listing, this is a proper XHTML file, with `<html>`, `<head>`, and `<body>` tags. This is required since Lift doesn't add these itself; Lift simply processes the XML from each template it encounters. The `<head>` element and its contents are boilerplate; the interesting things happen inside the `<body>` element. There are three elements here:

1. The `<lift:bind name=''content'' />` element determines where the contents of our index.html file are bound (inserted). The *name* attribute should match the corresponding *at* attribute from our `<lift:surround>` element.

2. The `<lift:Menu.builder />` element is a special element that builds a menu based on the SiteMap (to be covered in). The SiteMap is a high-level site directory component that not only provides a centralized place to define a site menu, but allows you to control when certain links are displayed (based on, say, whether a user is logged in or what roles they have) and provides a page-level access control mechanism.

3. The `<lift:msgs/>` element allows Lift (or your code) to display messages on a page as it's rendered. These could be status messages, error messages, etc. Lift has facilities to set one or more messages from inside your logic code.

Now let's look back at the `<lift:helloWorld.howdy/>` element from the `index.html` file. This element (and the `<lift:Menu.builder />` element, actually) is called a *snippet*, and it's of the form

```
<lift:class.method>
```

Where `class` is the name of a Scala class defined in our project in the `demo.helloworld.snippets` package and `method` is a method defined on that class. Lift does a little translation on the class name to change camel-case back into title-case and then locates the class. In our demo the class is located under `src/main/scala/demo/helloworld/snippet/HelloWorld.scala`, and is shown here:

```
package demo.helloworld.snippet

class HelloWorld {
  def howdy = <span>Welcome to helloworld at {new java.util.Date}</span>
}
```

As you can see, the howdy method is pretty straightforward. Lift binds the result of executing the method into the location of the snippet element (in this case a span). It's interesting to note that a method may itself return other `<lift:...>` elements in its content and they will be processed as well. This recursive nature of template composition is part of the fundamental power of Lift; it means that reusing snippets and template pieces across your application is essentially free. You should never have to write the same functionality more than once.

Now that we've covered all of the actual content elements, the final piece of the puzzle is the `Boot` class. The `Boot` class is responsible for the configuration and setup of the Lift framework. As we've stated earlier in the chapter, most of Lift has sensible defaults, so the `Boot` class generally contains only the extras that you need. The `Boot` class is always located in the `bootstrap.liftweb` package and is shown here (we've skipped imports, etc):

```
class Boot {
  def boot {
    // where to search snippet
    LiftRules.addToPackages("demo.helloworld")

    // Build SiteMap
    val entries = Menu(Loc("Home", "/", "Home")) :: Nil
    LiftRules.setSiteMap(SiteMap(entries:_*))
  }
}
```

There are two basic configuration elements, placed in the `boot` method. The first is the `LiftRules.addToPackages` method. It tells lift to base its searches in the `demo.helloworld` package. That means that snippets would be located in the `demo.helloworld.snippets` package, views (which we'll cover in) would be located in the `demo.helloworld.views` package, etc. If you have more than one hierarchy (multiple packages) you can just call `addToPackages` multiple times. The second item in the Boot class is the SiteMenu setup. Obviously this is a pretty simple menu in this demo, but we'll cover more interesting examples in .

Now that we've covered a basic example we hope you're beginning to see why Lift is so powerful and cna make you more productive. We've barely scratched the surface on Lift's templating and binding capabilities, but what we've shown here is already a big step. In roughly 10 lines of Scala code and about 30 in XML we have a functional site. If we wanted to add more pages we've already got our default template set up so we don't need to write the same boilerplate HTML over and over. In our example we're directly generating the content for our `helloWorld.howdy` snippet, but in later examples we'll show just how easy it is to actually pull content *from the template itself* into the snippet and modify it as needed.

In the following chapters we'll be covering

- Much more complex templating and snippet binding, including input forms and programmatic template selection
- How to use SiteMap and its ancillary classes to provide a context-aware site menu and access control layer
- How to handle state within your application
- Lift's ORM layer, which provides a powerful yet lightweight interface to databases
- Advanced AJAX and COMET support in Lift for Web 2.0 style applications
- Deployment on a range of platforms

We hope you're as excited about getting started with Lift as we are!

Chapter 2

An introduction to Maven and Project Layout

In this chapter we'll discuss the Maven build tool and how Lift uses it. We'll also take a brief look at the typical layout of a Lift application.

http://suereth.blogspot.com/2008/10/maven-for-beginners.html
http://scala-blogs.org/2008/01/maven-for-scala.html

2.1 Maven

2.1.1 What is Maven?

It's a project management tool. The Maven site

2.1.2 Build Lifecycles

Maven is built on the concept of lifecycles and there are three that are built-in, `default, clean` and `site`

The `default` lifecycle builds and deploys your project. The `clean` lifecycle cleans compiled objects or anything else that needs to be removed or reset. Finally, the `site` lifecycle creates the projects documentation.

Within each of the lifecycle there are a number of phases. Below is a listing of the phases that make up the `default` lifecycle.

- `validate` - validate the project is correct and all necessary information is available
- `compile` - compile the source code of the project
- `test` - test the compiled source code using a suitable unit testing framework. These tests should not require the code be packaged or deployed
- `package` - take the compiled code and package it in its distributable format, such as a JAR.

- `integration-test` - process and deploy the package if necessary into an environment where integration tests can be run
- `verify` - run any checks to verify the package is valid and meets quality criteria
- `install` - install the package into the local repository, for use as a dependency in other projects locally
- `deploy` - done in an integration or release environment, copies the final package to the remote repository for sharing with other developers and projects.

When you run `mvn deploy`, maven with run through each of these phases, executing the tasks associated with them. Since `deploy` is last in the list of tasks, all preceeding tasks will be executed as well. If you ran `mvn package`, only `validate`, `compile`, `test` and `package` will be run. Typically, during the development of your Lift application you'll run `mvn clean jetty:run`.

2.1.3 Plugins

Plugins add functionality to the build system. Lift is written in Scala, so we need to add the Maven Scala Plugin that adds the ability to compile Scala code. Below is a snippet from the main pom.xml file for a Lift application. You can see the Scala plugin adds a `compile` and `testCompile` goal for the build phase.

```
<plugin>
<groupId>org.scala-tools</groupId>
<artifactId>maven-scala-plugin</artifactId>
<executions>
<execution>
<goals>
<goal>compile</goal>
<goal>testCompile</goal>
</goals>
</execution>
</executions>
<configuration>
<scalaVersion>${scala.version}</scalaVersion>
</configuration>
</plugin>
```

2.1.4 Dependencies

Dependency management is one of the more useful features of Maven. Below is a declaration of the Jetty dependancy for the default Lift application. The details of the specification are straight forward, naming the dependency, setting a minimum version and defining the scope, in this case, the `test` phase is defined.

```
<dependency>
<groupId>org.mortbay.jetty</groupId>
<artifactId>jetty</artifactId>
<version>[6.1.6,)</version>
<scope>test</scope>
</dependency>
```

2.1.4.1 Adding a Dependency

Let's say that you'd like to add a new library and you want Maven to make sure you've got the most up-to-date version. We're going to add Configgy[1] as a dependency. Configgy is "a library for handling config files and logging for a scala daemon. The idea is that it should be simple and straightforward, allowing you to plug it in and get started quickly, writing small useful daemons without entering the shadowy world of java frameworks."

First we need to tell Maven where we can get Configgy, so in the <repositories> section add the following:

```
<repository>
  <id>http://www.lag.net/repo/</id>
  <name>http://www.lag.net/repo/</name>
  <url>http://www.lag.net/repo/</url>
</repository>
```

Then in the <dependencies> section add:

```
<dependency>
  <groupid>net.lag</groupid>
  <artifactid>configgy</artifactid>
  <version>[1.2,)</version>
</dependency>
```

That's it, you're done. The next time you run Maven for your project, it will pull down the Configgy jars into your M2_REPO.

2.1.5 Using Maven with Lift

As we touched on earlier, the interaction with Maven is generally limited with Lift. Getting started with either a sample application or starting from scratch you will mostly run the command `mvn clean jetty:run`. This command asks Maven to `clean` all your compiled classes and then execute the build phases that are pre-requisites for the `jetty:run` goal. You can see an example pom.xml file for Jetty here: http://mirrors.ibiblio.org/pub/mirrors/maven2/org/mortbay/jetty/jetty/6.1.12.rc5/jetty-6.1.12.rc5.pom

[1] Configgy's home is http://www.lag.net/configgy/

2.1.6 Further Resources

- http://maven.apache.org - Home page.
- http://maven.apache.org/what-is-maven.html - Their definition of Maven.
- http://maven.apache.org/guides/introduction/introduction-to-the-pom.html - Introduction to the pom file.
- http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html#Lifecycle_Reference - Introduction to the lifecycles.
- http://suereth.blogspot.com/2008/10/maven-for-beginners.html - General introduction to Maven.
- http://scala-blogs.org/2008/01/maven-for-scala.html - Introduction to Maven for Scala projects.
- http://mirrors.ibiblio.org/pub/mirrors/maven2/org/mortbay/jetty/jetty/6.1.12.rc5/jetty-6.1.12.rc5.pom - pom.xml file for Jetty 6.1.12.RC5
- http://mvnrepository.com/ - Search for plugins and dependancies for your application. This site is invaluable.

2.2 Project Layout

The directory structure of a Lift application is straight-forward, but we'll highlight a few important locations here.

<application_root>/src/main/scala This directory will contain your Scala source, such as snippets, model objects, and any libraries you write. The subfolder structure follows the traditional Java packaging style.

<application_root>/src/main/webapp All of the static aspects of your application, such as images, XHTML, JavaScript and CSS will be in this directory.

<application_root>/src/main/webapp/templates-hidden You'll find Lift templates are contained here. [See LiftViewFirst]

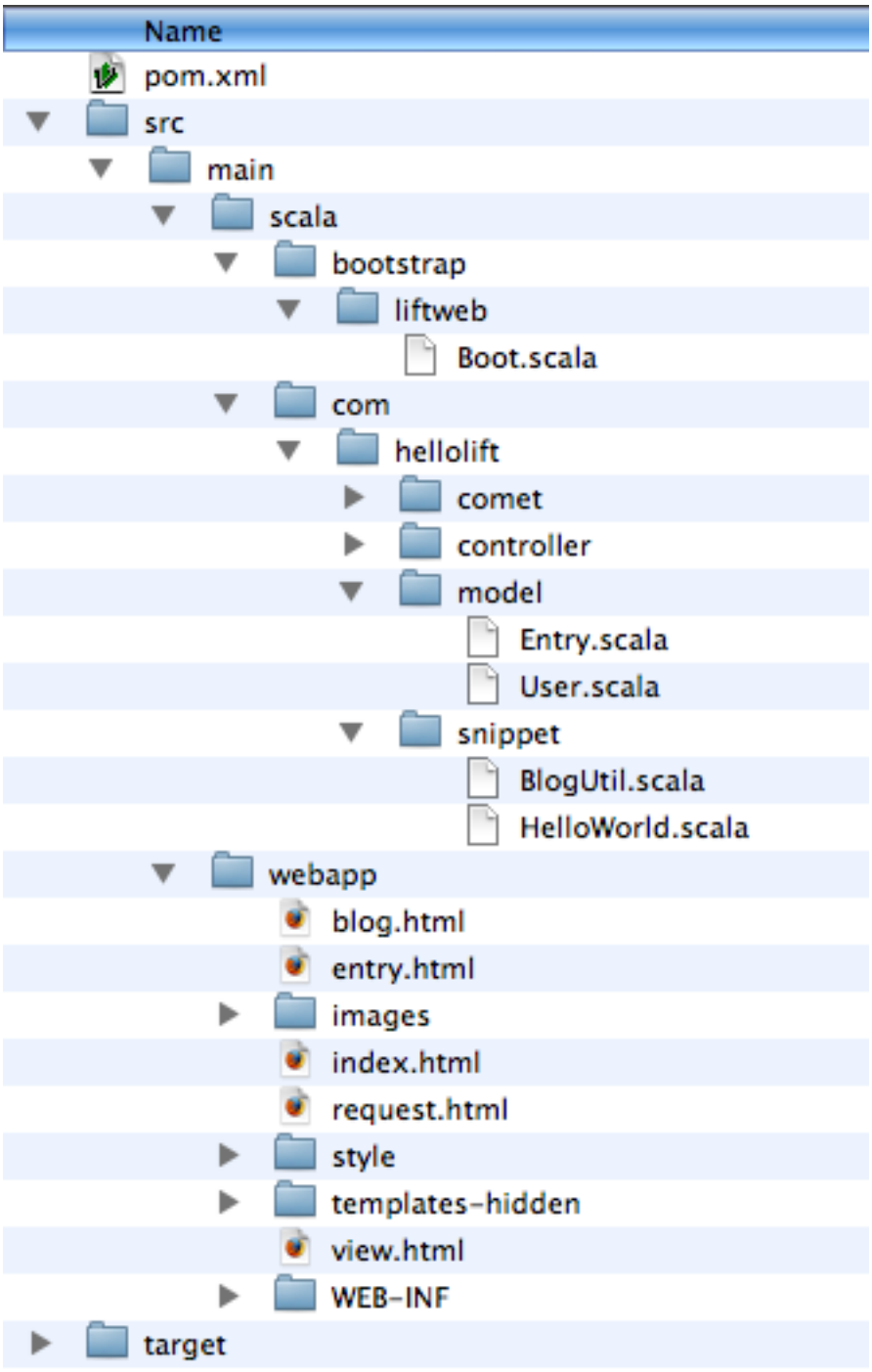


Figure 2.1: Project Directory Layout

Chapter 3

Demo Application: PocketChange

Planning
- describe the app
- outline the feature/chapter tie-ins
- Add a few screenshots

3.1 Our Demo Application

We decided to build an application throughout the book as evolutionary example. The application we've picked is an Expense Tracker and we're calling it *PocketChange.*

SCREENSHOT

PocketChange will track your expenses, keep a running total of what you've spent, allow you to organize by tags, and visualize the data. During the later chapters of the book we'll add a few fun features such as multiple people per account and <super awesome feature goes here when we think of it.>.

Chapter 4

Lift Architecture

4.1 Introduction

This chapter will walk you through fundamental aspects of understanding how Lift works, lifecycle of various entities, understanding how you can add you own functions to be part of processing and rendering pipeline etc. We will be describing parts of the Lift API such as LiftRules object.

4.2 Entry into Lift

The first step in Lift's request processing is intercepting the HTTP request. Originally, Lift used a Servlet instance to process incoming requests. This was changed to use a Filter instance[1] because this allows the container to handle any requests that Lift does not (in particular, static content). The filter acts as a thin wrapper on top of the existing LiftServlet (which still does all of the work), so don't be confused when you look at the ScalaDoc and see both classes. The main thing to remember is that your web.xml should specify the filter and not the servlet:

Listing 4.1: LiftFilter setup in web.xml

```
1  <?xml version="1.0" encoding="ISO-8859-1"?>
2  <!DOCTYPE web-app
3  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
4  "http://java.sun.com/j2ee/dtds/web-app_2_3.dtd">
5
6  <web-app>
7    <filter>
8      <filter-name>LiftFilter</filter-name>
9      <display-name>Lift Filter</display-name>
10     <description>The Filter that intercepts lift calls</description>
11     <filter-class>net.liftweb.http.LiftFilter</filter-class>
12   </filter>
13   <filter-mapping>
14     <filter-name>LiftFilter</filter-name>
15     <url-pattern>/*</url-pattern>
16   </filter-mapping>
17 </web-app>
```

[1] http://groups.google.com/group/liftweb/browse_thread/thread/b484ea2a13b6f84b/90ba1ef1115055a6

A full web.xml example is shown in section H.10 on page 130. In particular, the filter-mapping specifies that the Filter is responsible for everything. When the filter receives the request, it checks a set of rules to see if it can handle it. If the request is one that Lift handles, it passes it on to an internal LiftServlet instance for processing; otherwise, it chains the request and allows the container to handle it.

4.3 Bootstrap

When Lift starts up there are a number of things that you'll want to set up before any requests are processed. These things include setting up a SiteMenu, URL rewriting, custom dispatch, classpath search and pretty much all what LiftRules object has to offer. The Lift servlet looks for the bootstrap.liftweb.Boot class and executes the boot method in the class. You can also specify your own Boot instance by using the following context param in web.xml"

Listing 4.2: web.xml Boot param

```
<context-param>
     <param-name>bootloader</param-name>
  <param-value>foo.bar.baz.MyBoot</param-value>
</context-param>
```

Your MyBoot class needs to be a Bootable

Listing 4.3: Bootable class

```
abstract class Bootable {
       def boot() : Unit;
}
```

The boot method will only be run once, so you can place any initialization calls for other libraries here as well.

It is important to note that a lot of aspects related with Lift's behavior can be customized from boot by using LiftRules object.

4.4 Class Resolution

As part of our discussion of the Boot class, it's important to cover a small detail of how Lift determines where to find classes for Views and Snippet rendering. The LiftRules.addToPackages method tells lift what Scala packages to look in for a given class. Lift has implicit extensions to the paths you enter; in particular, if you tell Lift to use the "com.pocketchangeapp" package, Lift will look for View classes under "com.pocketchangeapp.view" and will look for Snippet classes under "com.pocketchangeapp.snippet". Typically the `addToPackages` method is excuted in your Boot class. A minimal Boot class would look like

Listing 4.4: Minimal Boot class

```
class Boot {
  def boot = {
```

```
    LiftRules.addToPackages("com.pocketchangeapp")
  }
}
```

4.5 Request/Response Lifecycle

Now that the request has entered the LiftServlet instance, it's time to process it. There are a quite a few fundamental steps that must be mentioned in order have a clear picture how and where certain things are happening. For more details regarding LiftRules object please see 4.6

1. Execute early functions. See LiftRules.appendEarly example. Essentially this is a mechanism that allows user function to be called very early before lift processes the HTTP request.
2. URL Rewriting (See LiftRules.prependRewrite/appendRewrite) - this is useful when you want to transform a URI path into something else such as query paramters etc. The result of the transformation will be passed to futher processing. Please see 4.9
3. Call LiftRules.onBeginServicing hooks. This is a mechanism that allows you to add your own hook functions that will be called when Lift is starting to procss the reqest.
4. Check for user-defined stateless dispatch (See LiftRules.prependStatelessDispatchPF/appendStatelessDispatchPF). These are partial functions that if they are defined for a given HTTP request they will return a LiftResponse which internally is turned in the bytes stream that is sent to client. These are veryusefull to build REST API's. The term stateless means that when the DispatchPF function is called the S object is not available and LiftSession not created/obtained yet.
5. Create Lift session (If there is no applicable stateless dispatch function)
6. Call LiftSession.onSetupSession when the HTTP session is activated by container. This is a mechanism for adding hook functions when LiftSession is created.
7. Init S object (S stands for Stateful)
8. Call LoanWrapper-s (See S.addAround). Essentially a LoanWrapper is a way of wrapping the lift's processing logic by your own code. This means that when your LoanWrapper implementation is called, lift is also passing you a function impersonating the entire processing logic. Therefore you have the opportunity to as pre and post conditions to lift's processing code.
9. Handle stateful request
   (a) Check the stateful dispatch functions (See LiftRules.prependDispath/appendDispatch). Similar with what was described on step #4 except that these functions are executed in the context of a LiftSession and veryimportatnt in the context of S object (See 4.15). Of course the first function defined for this request is called and the LiftResponse is returned. For an overview of Diaptch functions please see 4.10
   (b) If it's a **Comet** request process it. For an overview of what Comet is, please see 4.19
   (c) If it's an **Ajax** request process it:

i. Call LiftSession.onBeginServicing. Lift comes again with hooking functions. Note that we have LiftRules.onBeginServicing and LiftSession.onBeginServicing. The differences are whne these hooks ar called. In this case when lift is about to process the stateful request.

ii. Bascally execute the user's function mapped with that specific request token (impersonated by a request parameter) and return the response which can be a JavaScript, an XML construct or virtualy any LiftReponse. For an overview about LiftResponse please see 4.18

iii. Call LiftSession.onEndServicing. It'sprobably very intuitive what this does. It calls the hooks when finishing process the Ajax request.

(d) If it's a regular HTTP request process it such as:

i. Call LiftSession.onBeginServicing hooks. Similar with Ajax request processing hooks are called for nomal HTTP requests.

ii. Check the user's dispatch functions that are set per-session. If there is a function applicable execute it and return its response.If there is no per-session dispatch function process the request by executing the Scala function that user set up for specific events (such as when clicking a link, or pressing the submit button, or a function that will be executed when a form field is set etc.). Please see SHtml obejct 4.16

iii. Check the SiteMap and Loc functions. For an overview of SiteMap please see 4.17

iv. Lookup for the template based on the Request path. Lift will locate the templates using various aproaches:

A. Check for ViewDispatchPF functions. If there is a function defined for this path invoke it and return an Either

B. If there is no ViewDispatchPF look for the template in the path specified in the request. This is a very interesting mechanism that allows you toplugin your own templates or views. Nomally you markup templates are found in templates-hidden folder of your web application. But again Lift provides excellents means for extending so you can provide your markup templates virtually from anywhere.

v. Process the templates by executing snippets combining templates etc.

A. Merge <head> elements

B. Update the internal functions map. Basically associate user's scala functions with tokens that are impersonated in subsequent requests by HTTP parameters

C. Clean up notices (see S.error, S.warning, S.notice) since they were already rendered

D. Call LiftRules.convertResponse. Basically this glues together different pieces if information such as the actual markup, the response headers, cookies etc.

E. Check to see if lift needs to send HTTP redirect. For an overview please see 4.20

(e) Call LiftSession.onEndServicing. Onviously calling the end-serviceing hooks.

(f) Call LiftRules.performTransform. See LiftRules.responseTransformers. Essentially this is a list of functions that allows the user to make certain changes to the LiftResponse before being send to client.

10. Call LiftRules.onEndServicing hooks. These are the outer end-servicing hooks called after the S object context was destroyed.

11. Call before-send functions. I guess you may have seen this comming ... Again we have hooks that are called right before sending the response down to the pipe line.

12. Convert the LiftResponse to raw bytes stream and send it to client as HTTP response.

13. Call after-send functions. And of course the hooks that are called after the response was sent.

At a first glance certain things of this flow may seem unclear but after you're familiar with Lift framework and start using it is really important to know when certain things are happening so you can use the Lift goodiesin the right place.

4.6 LiftRules object

LiftRules object is the main way for configuring Lift's behavior which is typically done at startup in Boot. For more details of the LiftRules API please see I.1

Listing 4.5: LiftRules example

```
class Boot {
  def boot {
    // where to search snippet
    LiftRules.addToPackages("demo.helloworld")

    LiftRules.browserResponseToException = {
          // If there is an unnexpecte exception in your application during processing a request
      // this anonymous function is called and you can redirect to the error page
      case (mode, state, ex) => {ex.printStackTrace();RedirectResponse("/error")}
        }
  }
}
```

4.7 Templates

Templates form the backbone of Lift's flexibility and power. A template is essentially an XML file that contains Lift-specific tags as well as whatever content you want returned to the user. There are a number of built-in XML tags that Lift uses for specific reasons, these are of the form `<lift:name />`. Lift also allows you to create your own tags, which are called *snippets*. These user-defined tags are linked directly to Scala methods and these methods can process the contents of the snippet tag, or can generate their own content from scratch. Below is a simple template:

Listing 4.6: Sample template

```
<lift:surround  with="default" at="content">
  <head><title>Hello!</title></head>
  < lift:snippet  type="Hello.world" />
</lift:surround>
```

Notice the tags that are of the form <lift:name> which in this case are <lift:surround> and <lift:snippet>. These are two examples of Lift-specific tags. We'll discuss all of the tags that users will use in section Tags 4.11, but let's discuss the two contained here. We use the built-in <lift:surround> tag to make Lift embed our current template inside the "default" template. We also use <lift:snippet> to execute a snippet that we defined. In this case we execute the method world in the class Hello to generate some content.

Following rewriting and custom dispatch, Lift checks to see if it can find a file in the WAR tree that matches the request. Lift tries several suffixes (html, xhtml, htm, and no suffix) and also tries to match based on the client's Accept-Language header. The pattern Lift uses is

```
<path to template>[_<language, optional>][.<suffix>]
```

Because Lift will implicitly search for suffixes, it's best to leave the suffix off of your links within the web app. If you have a link with an href of /test/template.xhtml, it will only match that file, but if you use /test/template for the href and you have the following templates in your web app:

- /test/template.xhtml
- /test/template_es-ES.xhtml
- /test/template_ja.xhtml

then Lift will use the appropriate template based on the user's requested language if a corresponding template is available. In addition to normal templates, your application can make use of hidden templates. These are templates that are located under the /templates-hidden directory of your web app. Like the WEB-XML directory, the contents cannot be directly requested by clients. They can, however, be used by other templates through mechanisms like the lift:surround and lift:embed tags. If Lift cannot locate an appropriate template based on the request path then it will return a 404 to the user.

Once Lift has located the correct template, the next step is to process the contents. It is important to understand that Lift processes XML tags from the outside in. That means that in our example listing 4.6, the surround tag gets processed first. In this case the surround loads the default template and embeds our content at the appropriate location. The next tag to be processed is the <lift:Hello.world/> snippet. This tag is essentially an alias for the lift:snippet tag (specifically, <lift:snippet type=''Hello:world''>) , and will locate the Hello class and execute the world method on it. If you omit the "method" part of the type and only specify the class (<lift:Hello> or <lift:snippet type=''Hello''>) then Lift will attempt to call the render method of the class.

So templates are a nice way of setting up your layout and then writing a few methods to fill in the XML fragments that make up your web applications. They provide a simple way to generate a uniform look for your site. If you'd like more control or don't need a template for a certain section, you'll want to use a View, which is coming up in the next section.

Sometimes it might be tempted to load manually a template (from a snippet for example) using LiftSession.processSurroundAndInclude. Even if Lift will merge your templates and invoke snippets it is HIGHLY recommended to NOT use this as the processing functions will not be invoked as the functions mapping will not happen as this is called outside of normal rendering pipeline. So in other words let's say that in a cetain point in your application you want to return another page then the one that was supposed to be served

normally. One classic example would be that when un unnexpected exception is thrown you want toreturn an error page. Let's also say that this error page has a form allowing the user to submit *comments about the failure. There is a way of "catching" soch failure by using LiftRules.browserResponseToException which willbe called by Lift and you can return your own LiftResponse. But most certainly you do not want to hardcode the response so you want to use an existent error.html page. So you can loda this from disk using LiftRules.loadResourceAsXml and the call the LiftSession.processSurroundAndInclude so that Lift will process the <lift:xxx> tags, call your snippets etc. Once you have this done you can return your own LiftResponse by using XhtmlResponse. When you test it you'll be surprised to see that your form is not processed by lift; your functions are not called. That's because you did these things manully but your functions set through SHtml object (most likely) are not seen by Lift. In othertems you did these things outside of Lift's rendering pipeline. To solve this it is recommended that youdo not manually process templates in your own function unless you really know what you're doing. In this particular case instead of doing all this, you can just use a RedirectResponse object. Please see 4.20*

4.8 Views

We just discussed Templates and we saw how through a combination of an XML file, Lift tags, and Scala code we can respond to requests made by a user. You can also generate those responses entirely in code by using Views.

Views are implicitly defined custom dispatch methods. We'll cover Dispatch in more depth in the next section. **??**. A view is a normal Scala method that returns a NodeSeq; the main difference is that with custom dispatch we explicitly define the path that will lead to the method via LiftRules, whereas in a view, the class itself defines the path. In either case, View lookup and dispatch is done after template resolution (to be covered later), so templates take priority. There are two options for implementing a view class: one is to extend the LiftView trait, the other is to implement the InsecureLiftView trait. As you may be able to tell from the names, we would prefer that you use the LiftView trait. The InsecureLiftView determines method dispatch by turning a request path into a class and method name. For instance, if we have a path `/MyStuff/enumerate`, then Lift will look for a class called `MyStuff` in the view subpackage (class resolution is covered in section 4.4) and if it finds it and it has a method called `enumerate`, then Lift will execute the method and return its results to the user. The main issue there is that Lift uses reflection to get the method, so it can access any method in the class, even ones marked private. A better way to do it is to use the LiftView trait, which defines a dispatch partial function. This dispatch function maps a string (the "method name") to a function that will return a NodeSeq. Listing 4.7 shows a custom LiftView class where the path `/ExpenseView/enumerate` will map to the ExpenseView.doEnumerate method. If someone attempts to go to `/ExpenseView/privateMethod` they'll get a 404 since it's not defined in the dispatch function.

Listing 4.7: Dispatch in LiftView

```
class ExpenseView extends LiftView {
  override def dispatch = {
    case "enumerate" => doEnumerate _
  }

  def doEnumerate () : NodeSeq = {
    ...
```

```
    <lift:surround with="default" at="content">
     { expenseItems.toTable }
    </lift:surround>
  }
}
```

Another difference between custom dispatch and Views is that the NodeSeq returned from the view method is processed for template tags including `surrounds` and `includes`, just like snippets. That means that you can use the full power of the templating system from within your View, as shown in listing 4.7's `doEnumerate` method.

Since you can choose to not include any of the pre-defined template XHTML, you generate Atom or RSS feeds using a View. Resources of that nature, such as XML responses, or JSON, or an image response, would most likely be handled by a View.

4.9 URL Rewriting

Now that we've gone over Templates and Views and how the dispatching of a request to a Class.method works we can discuss how to intercept requests and handle them any way we want. URL rewriting is the mechanism that allows you to modify the incoming request so that it dispatches to a different URL. It can be used, among other things, to allow you to:

- Use user-friendly URLs like www.example.com/user/joe instead of www.example.com/admin/users/88
- Use short URLs instead of long, hard to remember ones, similar to tinyurl.com
- Use portions of the URL to determine how a particular snippet or view responds. For example, you could make it so that a user's profile is displayed via a URL like `http://someplace.com/user/derek` instead of having the username sent as part of a query string.

The mechanism is fairly simple to set up. We need to write a function of type `PartialFunction[RewriteRequest,Re` to determine if and how we want to rewrite particular requests. The simplest way to do this is with a match statement which will allow us to selectively match on some or all of the request information. It is important to understand that the Lift session is not created at the point when the rewrite functions run; that means that you generally can't set or access properties in the S object. RewriteRequest is a case object that contains three items: the parsed path, the request type and the original HttpServletRequest object.

The parsed path of the request in a ParsePath case class instance. The ParsePath class contains

1. The parsed path as a List[String]
2. The suffix of the request (i.e. "html", "xml", etc)
3. Whether the path is absolute
4. Whether the path ends in a slash ("/")

The latter three properties are useful only in specific circumstances, but the parsed path is what lets us work magic. The path of the request is defined as the parts of the URI between the context path and the query string. The following table shows examples of parsed paths for a Lift application under the "myapp" context path:

Requested URL	Parsed Path
http://foo.com/myapp/home?test_this=true	List[String](“home”)
http://foo.com/myapp/user/derek	List[String](“user”, “derek”)
http://foo.com/myapp/view/item/14592	List[String](“view”,”item”,”14592”)

The RequestType basically maps to the five HTTP methods: GET, POST, HEAD, PUT and DELETE. These are represented by the corresponding GetRequest, PostRequest, etc case classes, with an UnknownRequest case class to cover anything strange.

The flexibility of Scala’s matching system is what really makes this powerful. With matching on Lists in particular, we can match parts of the path and capture others. For instance, for our second example we’d like to rewrite the “/user/<username>” path so that it’s handled by the “/viewUser” template:

Listing 4.8: Simple rewrite example

```
val rewriter = {
  case RewriteRequest(ParsePath(user :: username :: Nil,_,_,_),_,_) =>
       RewriteResponse(viewUser :: Nil, Map(username -> username))
}
```

The RewriteResponse simply contains the new path to send and can also take a Map that contains parameters that will be accessible via S.param in the snippet or view. As we stated before, the LiftSession (and therefore most of S) isn’t available at this time, so the Map is the only way to pass information on to the rewritten location. Technically, a rewrite results in a “302 Moved Temporarily” result code that forwards the client to a new request. Because of that, even if you could set data in LiftSession or S it would disappear when the redirect occurs.

We can combine the ParsePath matching with the RequestType and HttpServletRequest to be very specific with our matches. For example, if we wanted to support the DELETE HTTP verb for a RESTful interface through an existing template, we could redirect it like so:

Listing 4.9: Complex rewrite example

```
val rewriter = {
  case RewriteRequest(ParsePath(username :: Nil, _, _, _),
                      DeleteRequest,
                      httpreq)
                      if isMgmtSubnet(httpreq.getRemoteHost()) =>
       RewriteResponse(deleteUser :: Nil, Map(username -> username))
}
```

We’ll go into more detail about how you can use this in the following sections. In particular, SiteMap provides a mechanism for doing rewrites combined with menu entries.

4.10 Dispatch functions

TBC

4.11 Tags

4.11.1 surround

Example: `<lift:surround with="template_name">children</lift:surround>`

Surrounds the child nodes with a named template (located in the <app_root>/webapp/templates-hidden). It is used to apply a unified template to all the pages in a site.

Demo example:<lift:surround with="default"> Dude... this is my page... but it'll appear in your browser in a template. </lift:surround>

In the target template there must be a <lift:bind /> tag to indicate where the contents should be bound. Note that you can use multiple surround templates by adding them to the /templates-hidden directory. For example, you might want to have a separate template for your administrative pages. In that case, you might add that template as admin.html in the /templates-hidden directory and then call it from your other pages using:<lift:surround with="admin">page code here</lift:surround>

Notes: You cannot have a hidden template with the same name as a sub-directory of your webapp directory. For example, if you had an admin.html template in /templates-hidden, you could not also have an admin directory.

4.11.2 snippet

Example: `<lift:snippet form="METHOD" type="ClassName:method" multipart="true" />`

The `form` and `multipart` attributes are optional. If `form` is included the attribute options are `GET` and `POST`. The `type` attribute resolves to the method call `ClassName.method`. The `multipart` attribute is a boolean.

Demo example:

Listing 4.10: Snippet that generates a NodeSeq

```
1 class Entry {
2 def add(xhtml: Group): NodeSeq =
3   selectedUser.is .openOr(new User).toForm(Empty, saveUser _) ++
4   <tr>
5     <td><a href="/simple/index.html">Cancel</a></td>
6     <td><input type="submit" value="Create"/></td>
7   </tr>
8 }
```

Y ou could use this snippet in the following manner:

```
...
<lift:Entry.add form="POST"/>
or
<lift:snippet type="Entry.add" form="POST" />
```

Listing 4.11: Snippet that generates Nodes to bind... TODO: Explain the difference better...

```
1 class Entry {
2   def showCount(in: NodeSeq): NodeSeq = {
3     val attr : String = S. attr ("name").openOr("N/A")
```

```
4       val value = CountHolder.is(attr)
5       bind("count", in, "value" -> value,
6         "incr" -> link("/count", () => CountHolder.is(attr) = value + 1, Text("++")),
7         "decr" -> link("/count", () => CountHolder.is(attr) = 0 max (value - 1), Text("--")))
8     }
9   }
```

You would then use this snippet like this:

```
...
<lift:Entry.showCount>
  Click this to increse the counter: <count:incr /><br />
  Click this to decrease the counter: <count:decr />
</lift:Entry.showCount>
...
```

4.11.3 embed

Example: `<lift:embed what="template" />`

Uses: Allows you to embed a template within another template (or to access a template from a JsCmd such as SetHtml, ModalDialog, etc.)

Note that incoming requests that contain *-hidden in the request will not be serviced, but you can access templates in directories named *-hidden. So, you can put AJAX templates in /ajax-templates-hidden in webapps.

Also, lift's i18n support extends to templates as well, so you can specify "/ajax-templates-hidden/welcome" and lift will serve the appropriate localized template. For example, if the current locale is set to French Canadian lift will look for /ajax-templates-hidden/welcome_fr_CA.html, /ajax-templates-hidden/welcome_fr.html, and /ajax-templates-hidden/welcome.html

Demo example:

Listing 4.12: Embedding Templates within Templates

```
<lift:embed what="/templates-hidden/supersecret-template"/>
```

<lift:embed what="/ajax-templates-hidden/welcome" />

<lift:embed what="/templates-hidden/ajax"/>

Caveats: JavaScript contained in templates rendered via JsCmd (sent in response to AJAX requests) will not be executed. This includes Comet Widgets.

4.11.4 comet

Example: `<lift:comet type="ClassName" name="optional"/>`

Uses: Defines a block of the XML document that is to be con

Example:

Listing 4.13: Comet Html template

```
1 <div class="widget">
2   < lift :comet type="Clock">Current Time:
3     <clk:time>Missing Clock</clk:time>
```

```
4     </lift:comet>
5   </div>
```

Will turn into this after processing:

Listing 4.14: Comet as rendered in the browser

```
1   <div style="text-align: center" class="widget">
2   <span id="LCHVVTE3H5CHZ31L1C2ZEQ_outer">
3   <span id="LCHVVTE3H5CHZ31L1C2ZEQ" lift:when="12">
4     Current Time:
5     <span id="LCHVVTE3H5CHZ31L1C2ZEQ_timespan">Fri Nov 14 17:29:53 EST 2008</span>
6   <script>
7   // <![CDATA[     /* JSON Func clk $$ F1226701773224999000_BBP */
8   function F1226701773224999000_BBP(obj) {
9     lift_ajaxHandler('F1226701773224999000_BBP='+ encodeURIComponent(JSON.stringify(obj)),
10    null, null);
11  }
12  // ]]>
13  </script>
14  </span>
15  <script>
16  // <![CDATA[
17  var destroy_LCHVVTE3H5CHZ31L1C2ZEQ = function() {}
18  // ]]>
19  </script>
20  </span>
21  </div>
```

Caveats: if you have a <lift:comet /> tag and you're using the tag from within sending AJAX stuff back, things might not work well.

More in section **??**

4.12 Snippets

A snippet method takes a single Scala.xml.NodeSeq argument and is expected to return a NodeSeq. The argument passed to the method is the XML contents of the snippet tag; because Lift processes from the outside in, the contents are not processed by default before being passed to the snippet method. For our current example, we could simply have a method that looks like this:

Listing 4.15: Hello World snippet

```
class Hello {
  def world (content : NodeSeq) : NodeSeq =
    Text("Hello, world!")
}
```

We simply return an XML Text node with our greeting. Note that the XML that a snippet returns is further processed from the outside in, so if your snippet instead looked like

Listing 4.16: Returning tags from a snippet

```
class Hello {
  def world (content : NodeSeq) : NodeSeq =
```

```
    <p>{"Hello, "}<lift:User.name /></p>
}
```

then the lift:User.name snippet will be processed as well after our snippet method returns. It is this hierarchical processing of template tags that makes Lift so flexible.

4.12.1 Stateless Snippets

All of the above examples showed Stateless snippets, those that execute and render a block of XML and send it back to the broswer. The life This is the default type of snippet.

4.12.2 Stateful Snippets

Lift also has a the concept of a stateful snippet. All of our previous examples, while simple, were stateless. That is to say, from request to request, there is no saved state on the server-side. Stateless snippets are useful in the sense that many requirements for a web applications need no state to be preserved as the user moves from page to page. There are a similar number of cases where we would like to preserve some state as the user navigates. For example, if you think about a graph that shows your spending for the last 3 months. You've changed the time span from the default 1 month to 3 months. Now you'd like to limit the exspense categories to show only "Food." More than likely, you'll want the application to respect the change in time-span as well. This is a case where keeping a bit of state around will allow you to do this. The same StatefulSnippet instance is used across a given page rendering. If you use a StatefulSnippet to render a form, a hidden field is added to that form that causes the same instance to be used on the page that is the target of the form submission.

Below is an example of a stateful snippet that handles the above example.

4.12.3 Default Snippet Method

If you define a snippet as `<lift:CSS />` rather than `<lift:CSS.show />` Lift will look for the method named `render` in the class `CSS`.

4.13 Eager Eval

If you want the contents of a snippet tag to be processed *before* the snippet, then you would specify the `eager_eval` attribute on the tag:

```
<lift:Hello.world eager_eval=''true''>...</lift:Hello.world>
```

This is especially useful if you're using a `lift:embed` tag; without the eager_eval attribute your snippet just sees the <lift:embed> tag, but with eager_eval set to true you can put *bindable* common snippet content into a single embedded template instead of copying it between templates. In addition to eager_eval, there are attributes such as `form` and `multipart` which we will cover in extensive detail in section 4.11.2.

4.14 Head Merge

Another feature of Lift's template processing is the ability to merge the HTML `head` tag from within a template. In our example listing 4.6, notice that we've specified a `head` tag inside the template. Without the head merge, this head tag would show up in the default template where our template gets bound. Lift is smart about this, though, and instead takes the contents of the head tag and merges it into the outer template's head tag. This means that you can use a surround tag to keep a uniform default template, but still change things like the title of the tag, add in scripts or special CSS, etc. For example, if you have a table in a page that you'd like to style with jQuery's TableSorter, you would add a Lift tag:

```
<lift:tohead><script src="/scripts/tablesorter.js" type="text/javascript" /></lift:tohead>
```

And just for this snippet, you'll import TableSorter.

4.15 S object

4.16 SHtml

The SHtml object is where most of the form element methods are defined. If you browse through the source you'll see what convenience functions are defined. There are a lot so we won't list them here, but throughout the book you'll see them appear in the example application we're building.

Some of the functions are text, button, select, checkbox, there are a lot more helper functions, so take a look at the ScalaDocs and at net/liftweb/http/SHtml.scala.

Here's a short snippet for generating a form:

Listing 4.17: SHtml Example - Snippet

```
class Ajax {
  def addForm = {
    <div>
     {SHtml.text("Date", println _)}
     {SHtml.textarea("Description", println _)}
     {SHtml.text("Tags", println _)}
     {SHtml.text("Value", println _)}
     {SHtml.submit("Submit", () => S.notice("Submitted"))}
    </div>
  }
}
```

And the template would be:

Listing 4.18: SHtml Example - Template

```
...
<body>
  <lift:Ajax.addForm form="POST" />
</body>
...
```

Which generates the markup:

Listing 4.19: SHtml Example - Generated Markup

```
<form method="post" action="/ajax">
  <div>
    <input name="F1228357378876744000_D3J" type="text" value="Date" /><br />
    <textarea name="F1228357378876890000_GXL">Description</textarea><br />
    <input name="F1228357378876940000_BLK" type="text" value="Tags" /><br />
    <input name="F1228357378877015000_RED" type="text" value="Value" /><br />
    <input name="F1228357378877116000_SZF" type="submit" value="Submit" /><br />
  </div>
</form>
```

4.17 SiteMap

We're only going to touch on SiteMap here as we have a whole chapter dedicated to it due to it's size, but an introduction, we think, is required.

SiteMap has a number of roles and they all build off the main goal, which is acting as a URL whitelist, arbitrating a users access to resources. And because it acts like the gatekeeper of resources, it can automatically generate your menu items. SiteMap can use all of the URL rewriting methods we discussed above as well.

Take a look at the SiteMap chapter to get a better feel of what it can do.

4.18 Lift reponses

4.19 Comet

4.20 HTTP redirects

4.21 Conclusion

We just gave you a detailed view of Lift's architecture. This is important material, so you'll want to understand it before moving on.

Chapter 5

SiteMap

SiteMap is a very powerful part of Lift that does essentially what it says: provides a map (menu) for your site. Of course, if all it did was generate a set of links on your pages we wouldn't have a whole chapter dedicated to it. SiteMap not only handles the basic menu generation functionality, but it also provides:

- Access control mechanisms that dela not only with whether a menu item is visible, but whether the page it points to is accessible
- Grouping of menu items so that you can easily display portions of menus where you want them
- Nested Menus so you can have hierarchies
- Request rewriting (similar to section 4.9)
- State-dependent computations for things like page title, page-specific snippets, etc.

The beauty of SiteMap is that it's very easy to start out with the basic functionality and expand it as needed as you grow.

5.1 Basic SiteMap Definition

Let's start with our basic menu for PocketChange. To keep things simple, we'll just define four menu items at the beginning:

1. The Homepage. Depending on whether the user is logged in or not, this page should display either the user's entries or a welcome page
2. Login and registration links if the user isn't logged in, a logout link if they are
3. View/edit profile if the user is logged in
4. A help page

We'll assume that we have the corresponding pages, "homepage", "login", "logout", and "profile" written and functional. We'll also assume that the help page(s) reside under the "help" subdirectory to keep things neat, and that the entry to help is `/help/index`.

5.1.1 Creating Menu Entries

Menu entries are created using the Menu[1] class, and its corresponding Menu object. A Menu, in turn, holds a Loc[2] trait instance, which is where most of the interesting things happen. A menu can also hold one or more child menus, which we'll cover in section 5.1.2. Note that the Loc object has several implicit methods that make defining Locs easier, so you generally want to import them into scope. The simplest way is to import `net.liftweb.sitemap.Loc._`, but you can import specific methods by name if you prefer. A Loc can essentially be thought of as a link in the menu, and contains four basic items:

ort Loc implicits

1. The name of the Loc. This must be unique across your sitemap because it can be used to look up specific Menu items if you customize your menu display (section 5.2.3)

2. The link that the Loc refers to. Usually this will refer to a specific page, but Lift allows a single Loc to match based on prefix as well (section 5.2.2)

3. The text of the menu item. This what will be displayed to the user. You can use a static string or you can generate it with a function (section 5.2.2)

4. An optional set of parameters that control behavior and appearance of the menu item. These parameters are covered in sections 5.2,5.3, 5.4, and 5.5

For our example we'll tackle the help page link first, since it's the simplest (a static link, essentially). The definition is shown in listing 5.1. We're assuming that you've imported the Loc implicit methods to keep things simple; we'll cover instantiating the classes directly in later sections of this chapter.

Listing 5.1: Help Menu Definition

```
val helpMenu = Menu(Loc("helpHome",
                        ("help" :: Nil) -> true,
                        "Help"))
```

For our example we've named this menu item "helpHome". The second parameter is a Pair[List[String],Boolean] which does two things: the list defines the path of the request to match. The path in this case is considered to be the portion of the request path following our web app context. For example, if we deploy the WAR at `/pchange/`, then `/pchange/help/` would match this Loc. The boolean parameter (false if you just pass in a List[String] to Loc) controls whether the path acts as a prefix match. In our case, but passing in true, we're saying that anything under the help directory will also match. Remember that SiteMap also acts as access control, so by doing this we're allowing full access to all of the help files. The final parameter, "Help", is the text for the menu link.

[1] `net.liftweb.sitemap.Menu`
[2] `net.liftweb.sitemap.Loc`

5.1.2 Nested Menus

5.1.3 Setting the Global SiteMap

5.2 Customizing Display

5.2.1 Hidden

5.2.2 Link and LinkText

5.2.3 Using <lift:Menu>

5.3 Access Control

5.3.1 If

5.3.2 Unless

5.4 Validation

5.4.1 Test

5.5 Page-Specific Rendering

5.5.1 The Template Parameter

5.5.2 The Snippet and LocSnippets Parameters

5.5.3 Title

5.6 Writing Your Own Loc

- Can handle all of what LocParams do
  - Rewriting with type-safe params (rewrite)
  - Template resolution (calcTemplate)
  - Special snippet handling (snippets)
  - Link Text (text)
  - URL (link)
  - Access control (Add your own If/Unless on params)
- Can specify your own Params still (params)

Chapter 6

The Mapper and Record Frameworks

Usually we find that most webapps end up wanting to store user data somewhere. Once you start working with user data, though, you start dealing with issues like coding up input forms, validation, persistence, etc to handle the data. That's where the Mapper and Record frameworks comes in. These frameworks provides a scaffolding for all of your data manipulation needs. Mapper is the original Lift persistence framework, and is closely tied to JDBC for its storage. Record is a new refactorization of Mapper that is backing-store agnostic at its core, so it doesn't matter whether you want to save your data to JDBC, JPA, or even something like XML. With Record, selecting the proper driver is as simple as hooking the proper traits into your class. Because these frameworks are based on the same concepts, we will be covering them in parallel in this chapter.

> Unless we note otherwise, functionality for each framework is the same. We will point out minor differences like method names inline, while major differences will be covered in their own sections.

6.1 Introduction to Mapper and MetaMapper

Let's start by discussing the relationship between the Mapper and MetaMapper traits (and the corresponding Record and MetaRecord). Mapper provides the *per-instance* functionality for your class, while MetaMapper handles the *global* operations for your class and provides a common location to define per-class static specializations of things like field order, form generation and HTML representation. In fact, many of the Mapper methods actually delegate to methods on MetaMapper. In addition to Mapper and MetaMapper there is a third trait, MappedField, that provides the per-field functionality for your class. In Record the trait is simply called "Field". The MappedField trait lets you define the individual validators as well as things transform filters and filed name. Under Record, Field adds things like tab order and default error messages for form input handling.

6.1.1 Setting Up the Database Connection

The first step you need to get out of the way is defining the database connection. We do this by defining an object called `DBVendor` (you can call it whatever you want). This object extends the net.liftweb.mapper.ConnectionManager trait and must implement two methods: newConnection and releaseConnection. You can make this as sophisticated as you want, with pooling, caching, etc, but for now, listing 6.1shows a basic implementation to set up a PostgreSQL driver.

Listing 6.1: Setting up the Database

```
1  object DBVendor extends ConnectionManager {
2    // Force load the driver
3    Class.forName("org.postgresql.Driver")
4    // define methods
5    def newConnection(name : ConnectionIdentifier) = {
6      try {
7        Full(DriverManager.getConnection(
8             "jdbc:postgresql://localhost/mydatabase",
9             "root", "secret"))
10     } catch {
11       case e : Exception => e.printStackTrace; Empty
12     }
13   }
14   def releaseConnection (conn : Connection) { conn.close }
15 }
16
17 DB.defineConnectionManager(DefaultConnectionIdentifier, DBVendor)
```

A few items to note:

1. The name parameter for newConnection can be used in case you need to have connections to multiple distinct databases. One specialized case of this is when you're doing DB sharding (horizontal scaling). Multiple database usage is covered in more depth in section 6.4.1

2. The newConnection method needs to return a Can[java.sql.Connection]. Returning an Empty Can indicates failure

3. The releaseConnection method exists so that you have complete control over the lifecycle of the connection. For instance, if you were doing connection pooling yourself you would return the connection to the available pool rather than closing it

4. The DB.defineConnectionManager call is what binds our manager into Mapper. Without it your manager will never get called

6.1.2 Constructing a Mapper-enabled class

Now that we've covered some basic background, we can start constructing some Mapper classes to get more familiar with the framework. We'll start with a simple example of a class for a ledger entry with the following fields:

- Date

- Description, with a max length of 100 chars
- Amount, a decimal value with a precsion of two places.

The first thing we do is declare our Entry class, as shown in listing 6.2.

Listing 6.2: Entry class in Mapper

```
class MapEntry extends KeyedMapper[Long,MapEntry] {
  def getSingleton = MapEntryMeta
  def primaryKeyField = id
  object id extends MappedLongIndex(this)
  object date extends MappedDateTime(this)
  object description extends MappedString(this,100)
  // 2 digits to the right
  object amount extends MappedDecimal(this,2)
}
```

The Record version is shown in listing 6.3; the only differences are that the `getSingleton` method has been renamed to `meta`, and the Field traits use different names under the Record framework (DateTimeField vs MappedDateTime).

Listing 6.3: Entry class in Record

```
class Entry extends KeyedRecord[Entry,Long] {
  def meta = EntryMeta
  def primaryKey = id
  object id extends LongField(this) with KeyField[Long,Entry]
  object date extends DateTimeField(this)
  object description extends StringField(this, 100)
  // 2 digits to the right
  object amount extends DecimalField(this, 2)
}
```

As you can see, we've set Entry to extend the KeyedMapper trait (we need to tell Mapper what the subtype is) and we've added the fields required by our class. We need to provide a primary key on the entity, so we use the KeyedMapper trait instead of Mapper (or, correspondingly, KeyedRecord instead of Record). When you use the KeyedMapper trait you need to provide an implementation for the primaryKeyField def, which must match the type of the KeyedMapper trait and be a subtype of IndexedField. Currently, Mapper supports both indexed Longs and Strings, so if you want to use some other type for your primary key you'll need to roll your own (section 6.2.8). Technically Int indexes are supported as well, but there is no corresponding trait for an Int foreign key. That means that if you use an Int for the primary key you may not be able to do a relationship with other objects (section 6.1.3) unless you write your own. Record is a little more flexible in primary key selection since it uses, in effect, a marker trait (KeyField) to indicate that a particular field is a key field. One thing to note is that in Mapper, the table name foryour entity defaults to the name of the class (Entry in our case). If you want to change this then you just need to override the dbTableName def.

Looking at these examples, you've probably noticed that the fields are defined as objects rather than instance members (vars). The basic reason for this is that the MetaMapper needs access to fields for its validation and form functionality; it wouldn't be possible to cleanly define these

properties in the MetaMapper if it had to access member vars on each instance since a MetaMapper instance is itself an object. Also note that DecimalField is a custom field type, which we will cover in section 6.2.9.

In order to tie all of this together, we need to define a matching KeyedMetaMapper object as the singleton for our entity, as shown in listing 6.4. The Meta object (whether MetMapper or MetaRecord) is where you define most behavior that is common across all of your instances.

Listing 6.4: EntryMeta object

```
object EntryMeta extends Entry with KeyedMetaMapper[Long,Entry] {
  override def fieldOrder = List(date, description, amount)
}
```

In our case we're simply defining the order of fields as they'll be displayed in XHTML and forms by overriding the `fieldOrder` method. The default behavior is an empty list, which means no fields are involved in display or form generation; generally you will want to override `fieldOrder` since this is not very useful. If you don't want a particular field to show up in forms or XHTML output, simply omit it from the fieldOrder list.

Since fields aren't actually instance members, operations on them are slightly different that a regular var. The biggest difference is in how we set fields: we use the apply method. In addition, field access can be chained so that you can set multiple field values in one statement, as shown in listing 6.5:

Listing 6.5: Setting field values

```
myEntry.date(new Date).description("A sample entry")
myEntry.amount("127.20")
```

The underlying value of a given field can be retrieved with the `is` method (`value` method in Record) as shown in listing 6.6.

Listing 6.6: Accessing field values in Record

```
// mapper
val tenthOfAmount = myEntry.amount.is / 10
val formatted = String.format("%s : %s",
                              myEntry.description.is,
                              myEntry.amount.is.toString)
// record
if (myEntry.description.value == "Doughnuts") {
  println("Diet ruined!")
}
```

6.1.3 Object Relationships

Often it's appropriate to have relationships between different entities. The archetypical example of this is the parent-child relationship. In SQL, a relationship can be defined with a foreign key that associates one table to another based on the primary key of the associated table. In Mapper, there is a corresponding MappedForeignKey trait, with concrete implementations for Long and String foreign keys. To give an example, our ledger entries should be owned by a particular user. Listing

6.7 shows a rough User entity definition along with the addition of the MappedForeignKey field on our existing Ledger class to specify a one-to-many from User to Entry.

Listing 6.7: Foreign Key Definition

```
class User extends KeyedMapper[Long,User] {
  def getSingleton = UserMeta // not shown here
  def primaryKeyField = id
  object id extends MappedLongIndex(this)
}

class Entry extends KeyedMapper[Long,Entry] {
  ...
  object owner extends MappedLongForeignKey(this,UserMeta)
}
```

Once we have this defined, accessing the object via the relationship is achieved by using the obj method on the foreign key field, as shown in listing 6.8.

Listing 6.8: Accessing Foreign Objects

```
class Entry extends KeyedMapper[Long,Entry] {
 ...
 def parentName = Text("My parent is " + owner.obj.name.is)
}
```

With the foreign key functionality you can easily do one-to-many and many-to-one relationships (depending on where you put the foreign key). If you want to do many-to-many mappings you'll need to provide your own "join" class with foreign keys to both of your mapped entities. An example would be if we wanted to have categories for our ledger entries and wanted to be able to have a given entry have multiple categories (i.e. you purchase a book for your mother's birthday, so it has the categories Gift, Mom and Books). First we define the category entity, as shown in listing6.9 .

Listing 6.9: Category Entity

```
class Category extends KeyedMapper[Long,Category] {
  def getSingleton = CategoryMeta
  def primaryKeyField = id
  object id extends MappedLongIndex(this)
  object name extends MappedString(this,100)
}
object CategoryMeta extends Category with KeyedMetaMapper[Long,Category] {
  override def fieldOrder = List(name)
}
```

Next, we define our join entity, as shown in listing . It's a KeyedMapper just like the rest of the entities, but it only contains foreign key fields to the other entities.

Listing 6.10: Join Entity

```
class CategoryEntry extends KeyedMapper[Long,CategoryEntry] {
  ... singleton, primary key defs here ...
```

```
  object entry extends MappedLongForeignKey(this,EntryMeta)
  object category extends MappedLongForeignKey(this,CategoryMeta)
}

object CategoryEntryMeta extends CategoryEntry with KeyedMetaMapper[Long,CategoryEntryMeta] {
  def join (category : Category, entry : Entry) = {
    val joiner = this.create
    joiner.entry(entry)
    joiner.category(category)
    joiner
  }
}
```

To use the join entity, you'll need to create a new instance and set the appropriate foreign keys to point to the associated instances. As you can see, we've defined a convenience method on out CategoryJoinMeta object to do just that. To make the many-to-many accessible as a field on our entities, we can use the HasManyThrough trait, as shown in listing 6.11.

Listing 6.11: HasManyThrough for Many-to-Many Relationships

```
object categories extends
  HasManyThrough(this,
                 CategoryMeta,
                 CategoryEntryMeta,
                 CategoryEntryMeta.entry,
                 CategoryEntryMeta.category)
```

A similar field could be set up on the Category entity to point to entries. It's important to note a few items:

- The only way to add new entries is to directly construct the CategoryEntry instances and save them. You can't make any modifications via the HasManyThrough trait
- Although the field is defined as a query, the field is actually lazy and only runs once. That means if you query it and then add some new CategoryEntry instances, they won't show up in the field contents

If you want a way to retrieve the joined results such that it pulls fresh from the database each time, you can instead define a helper method as shown in listing 6.39 on page 58.

6.1.4 Persistence Operations on an Entity

Now that we've defined our entity we probably want to use it in the real world to load and store data. There are several operations on KeyedMetaMapper that we can use :

create Creates a new instance of the entity

save Saves an instance to the database.

delete Deletes the given entity instance

find* A whole host of means for locating entities, by primary key, typesafe queries, raw SQL queries, etc. These will be covered in detail in their own section, 6.1.5

In general these operations will be supported in both Record and Mapper. However, because Record isn't coupled tightly to a JDBC backend some of the find methods may not be supported and there may be additional methods not available in Mapper for persistence. For this reason, this section will deal specifically with Mapper's persistence operations; Record's backends and their supported operations are covered in section 6.3.

Creating an Instance

Once we have the KeyedMetaMapper object defined we can use it to create objects using the `create` method. You can't just use the "new" operator because the framework has to set up internal data for the instance such as field owner, etc. This is important to remember, since nothing will prevent you from creating an instance manually; you'll just get all kinds of errors when you go to use the instance.

Saving an Instance

Saving an instance is as easy as calling the `save` method on the instance you want to save. Optionally, you can call the `save` method on the Meta object, passing in the instance you want to save. The save method uses the id field to determine whether an insert or update is required to persist the current state to the database, and returns a boolean to indicate whether the save was successful or not.

Deleting an Instance

There are several ways to delete instances. The simplest way is to call the `delete_!` method on the instance you'd like to remove. An alternative is to call the `delete_!` method on the Meta object, passing in the instance to delete. In either case, the `delete_!` method returns a boolean indicating whether the delete was successful or not.

Another approach to deleting entities is to use the `bulkDelete_!!` method on MetaMapper.This method allows you to specify query parameters to control which entities are deleted. We will cover query parameters in section 6.1.5 (an example is in listing 6.17 on page 48).

6.1.5 Querying for Entities

There are a variety of methods on MetaMapper for querying for instances of a given entity. The simplest method is `findAll` called with no parameters. The "bare" `findAll` returns a List of all of the instances of a given entity loaded from the database. Note that each findAll... method has a corresponding method that takes a database connection for sharding or multiple database usage (see sharding in section 6.4.1). Of course, for all but the smallest datasets, pulling the entire model to get one entity from the database is inefficient and slow. Instead, the MetMapper provides methods for using queries to narrow the set of entities returned.

The ability to use fine-grained queries to select data is a fundamental feature of relational databases, and Mapper provides first-class support for constructing queries in a manner that is not only easy to use, but type-safe. This means that you can catch query errors at compile time

instead of runtime. The basis for this functionality is the QueryParam trait, which has several concrete implementations that are used to construct the actual query. The QueryParam impls can be broken up into two main groups:

1. Comparison - These are typically items that would go in the where clause of an SQL query. They are used to refine the set of instances that will be returned

2. Control - These are items that control things like sort order and pagination of the results

Although Mapper provides a large amount of the functionality in SQL, some features are not covered directly or at all. In some cases we can define helper methods to make querying easier, particularly for joins (section 6.1.6).

The simplest QueryParam to refine your query is the By object, and its related objects. By is used for a direct value comparison of a given field, essentially an "=" in SQL. For instance, to get an instance of our ledger entry by its primary key, we use findAll as shown in listing 6.12.

Listing 6.12: Retrieving by Primary Key

```
val myEntry = EntryMeta.findAll(By(Entry.id, myId)).firstOption
```

As you can see, we use the firstOption to convert from a List to an Option. If the Option is None it means that the entity could not be loaded. Loading a single entity is somewhat of a special case, but in this example we're selecting on the primary key, which has a unique constraint and thus should return either one entity or none (perfect for an Option). Besides By, the other basic clauses are:

- NotBy - Selects entities whose queried field is not equal to the given value

- By_> - Selects entities whose queried field is larger than the given value

- By_<- Selects entities whose queried field is less than the given value

- ByList - Selects entities whose queried field is equal to one of the values in the given List. This corresponds to the "field IN (x,y,z)" syntax in SQL.

- NullRef - Selects entities whose queried field is NULL

- NotNullRef - Select entities whose queried field is not NULL

- Like - Select entities whose queried field is like the given string. As in SQL, the percent sign is used as a wildcard

In addition to the basic clauses there are some slightly more complex ways to control the query. The first of these is ByRef, which selects entities whose queried field is equal to the value of another query field *on the same entity*. . The related NotByRef tests for inequality between two query fields.

Getting slightly more complex, we come to the In QueryParameter, which is used just like an "IN" clause with a subselect in an SQL statement. As a totally contrived example , let's say we wanted to get all of the entries that belong to categories that start with the letter "c". Listing 6.13 shows the full breakdown.

Listing 6.13: Using In

```
EntryMeta.findAll(
  ByRef(EntryMeta.id, CategoryJoinMeta.entry),
  In(CategoryJoinMeta.category,
     CategoryMeta.id,
     Like(CategoryMeta.name, "c%")))
```

We use the ByRef params to do the join between the many-to-many entity on the query. Related to In is InRaw, which allows you to specify your own SQL subquery for the "IN" portion of the where clause. An example in listing 6.14 shows how we could use this to find Categories for ledger entries made in the last 30 days.

Listing 6.14: Using InRaw

```
def recentCategories = {
  val joins = CategoryJoinMeta.findAll(
    InRaw(CategoryJoinMeta.entry,
          "select id from Entry where Entry.date > (CURRENT_DATE - interval '30 days')",
          IHaveValidatedThisSQL("dchenbecker", "2008-12-03"))
  joins.map(_.category.obj).sort(_.id < _.id).removeDuplicates
}
```

Here things are starting to get a little hairy . The InRaw only allows us to specify the subquery for the IN clause, so we have to do some postprocessing to get unique results. If you want to do this in the query itself you'll have to use the findAllByInsecureSql or findAllByPreparedStatement methods, which are covered later in this section on page 61. The final parameter for InRaw acts as a code audit mechanism that says that someone has checked the SQL to make sure it's safe to use. Since the query fragment is added to the master query as-is, no escaping or other filtering is performed on the string. That means that if you take user input you need to be very careful about it or you run the risk of an SQL injection attack on your site.

The next QueryParam we'll cover is BySql, which lets you use a complete SQL fragment that gets put into the where clause. An example of this would be if we want to find all ledger entries within the last 30 days, as shown in listing 6.15. The IHaveValidatedThieSQL case class is required as a code audit mechanism to make sure someone has verified that the SQL used is safe.

Listing 6.15: Using BySql

```
val recentEntries = EntryMeta.findAll(
  BySql("Entry.date > (CURRENT_DATE - interval '30 days')",
        IHaveValidatedThisSQL("dchenbecker","2008-12-03"))
```

The tradeoff with using BySql is that you need to be careful with what you allow into the query string. BySql supports parameterized queries as shown in listing 6.16, so use those if you need to have dynamic queries. Whatever you do, don't use string concatenation unless you really know what you're doing.

Listing 6.16: Parameterized BySql

```
val amountRange = EntryMeta.findAll(
  BySql("Entry.amount between ? and ?", lowVal, highVal))
```

As we mentioned in section 6.1.4 on page 45, we can use the query parameters to effect bulk deletes in addition to querying for instances. Simply use the QueryParam classes to constrain what you want to delete. Obviously the control params that we'll cover next make no sense but the compiler won't complain . Listing 6.17 shows an example of deleting all entries older than a certain date.

Listing 6.17: Bulk Deletion

```
def deleteBefore (date : Date) =
  EntryMeta.bulkDelete_!!(By_<(EntryMeta.date, date))
```

Now that we've covered the selection and comparison QueryParams, we can start to look at the control params. The first one that we'll look at is OrderBy. This operates exactly like the order by clause in SQL, and allows you to order on a given field in either ascending or descending order. Listing 6.18 shows an example of ordering our ledger entries by amount. The Ascending and Descending case objects are in the net.liftweb.mapper package. The OrderBySql case class operates similarly except you provide your own SQL fragment for the ordering, as shown in the example. Again, you need to validate this SQL.

Listing 6.18: OrderBy Clause

```
val cheapestFirst =
  EntryMeta.findAll(OrderBy(EntryMeta.amount,Ascending))
// or
val cheapestFirst =
  EntryMeta.findAll(OrdeBySql("Entry.amount asc"),
    IHaveValidatedThisSQL("dchenbecker", "2008-12-03"))
```

Pagination of results is another feature that people often want to use, and Mapper provides a simple means for controlling it with two more QueryParam classes: StartAt and MaxRows, as shown in listing 6.19. In this example we take the offset from a parameter passed to our snippet, with a default of zero.

Listing 6.19: Pagination of Results

```
val offset = S.param("offset").map(_.toLong) openOr 0
EntryMeta.findAll(StartAt(offset), MaxRows(20))
```

An important feature of the methods that take QueryParams is that they can take multiple params, as shown in this example. A more complex example is shown in listing 6.20. In this example we're doing a query using a Like clause, ordering on the date of the entries, and paginating the results, all in one statement!

Listing 6.20: Multiple QueryParams

```
EntryMeta.findAll(Like(EntryMeta.description, "Gift for%"),
                  OrderBy(EntryMeta.date,Descending),
                  StartAt(offset),
                  MaxRows(pageSize))
```

Another useful QueryParam is the Distinct case class, which acts exactly the same way as the DISTINCT keyword in SQL. The final "control" QueryParam that we'll cover is PreCache. It's

used when you have a mapped foreign key field on an entity. Normally, when Mapper loads your main entity it leaves the foreign key field in a lazy state, so that the query to get the foreign object isn't executed until you access the field. This can obviously be inefficient when you have a whole lot of entities loaded that you need to access, so the PreCache parameter forces Mapper to preload the foreign objects as part of the query. Listing 6.21 shows how we could use this to fetch a ledger entry as well as the owner of the entry (we would need to set up an owner field for this example to work).

Listing 6.21: Using PreCache

```
def loadEntry (id : Long) =
  EntryMeta.findAll(By(EntryMeta.id, id),
                    PreCache(EntryMeta.owner))
```

6.1.6 Making Joins a Little Friendlier

In case you would prefer to keep your queries type-safe but you want a little more convenience in your joins between entities, you can define helper methods on your entities. One example would be finding all of the Categories for a given Entry, as shown in listing 49. Using this method in our example has an advantage over using HasManyThrough in that it will pull from the database each time instead of just once at query.

Listing 6.22: Join Convenience Method

```
def categoriesPull = CategoryEntryMeta.
  findAll(By(CategoryEntryMeta.entry, this.id)).map(_.category.obj)
```

6.2 Utility Functionality

In addition to the first-class persistence support in Mapper and Record, the frameworks provide additional functionality to make writing data-driven applications much simpler. This includes things like automatic XHtml representation of objects and support for generating everything from simple forms for an entity up to a full-fledged CRUD[1] implementation for your entities.

6.2.1 Display generation

If you want to display a mapper instance as XHTML, simply call the `asHtml` method (`toXHtml` in Record) on your instance. The default implementation turns each field's value into a Text node via the `toString` method and concatenates the results, separated by newlines. If you want to change this behavior, override the `asHtml` on your field definitions. For example, if we wanted to control formatting on our date we could modify the field as shown in listing

Listing 6.23: Custom field display

```
import java.util.DateFormat
```

[1] An abbreviation (Create, Read, Update and Delete) representing the standard operations that are performed on database records. Taken from http://provost.uiowa.edu/maui/Glossary.html

```
...
object date extends MappedDateTime(this) {
  final val dateFormat =
    DateFormat.getDateInstance(DateFormat.SHORT)
  override def asHtml = Text(dateFormat.format(value.getTime))
}
```

Note that the DateTimeField contains a `java.util.Calendar` instance, which is why we need to use the `getTime` method on the value. A similar method, `asJSON`, will return the JSON representation of the instance.

6.2.2 Form generation

One of the biggest pieces of functionality in the Mapper framework is the ability to generate entry forms for a given record. The `toForm` method on Mapper is overloaded so that you can control how your form is created. All three `toForm` methods on Mapper take a `Can[String]` as their first parameter to control the submit button; if the Can is Empty, no submit button is generated, otherwise, the String contents of the Can are used as the button label. If you opt to skip the submit button you'll need to provide it yourself via binding or some other mechanism, or you can rely on implicit form submission (when the user hits enter in a text field, for instance). The first `toForm` method simply takes a function to process the submitted form and returns the XHTML as shown in listing 6.24:

Listing 6.24: Default toForm method

```
myEntry.toForm(Full("Save"), { _.save })
```

As you can see, this makes it very easy to generate a form for editing an entity. The second `toForm` method allows you to provide a URL which the Mapper will redirect to if validation succeeds on form submission (this is not provided in Record). This can be used for something like a login form, as shown in listing 6.25:

Listing 6.25: Custom submit button

```
myEntry.toForm (Full("Login"), "/member/profile")
```

The third form of the toForm method is similar to the first form, with the addition of "redo" snippet parameter. This allows you to keep the current state of the snippet when validation fails so that the user doesn't have to re-enter all of the data in the form.

The Record framework allows for a little more flexibility in controlling form output. The MetaRecord object allows you to change the default template that the form uses by setting the formTemplate var. The template may contain any XHTML you want, but specifically, the `toForm` method will do special handling for the following tags:

<lift:field_label name="..." /> The label for the field with the given name will be rendered here.

<lift:field name="..." /> The field itself (specified by the given name) will be rendered here. Typically this will be an input field, although it can be anything type-appropriate. For example, a BooleanField would render a checkbox.

<lift:field_msg name="..." /> Any messages, such as from validation, for the field with the given name will be rendered here.

As an example, if we wanted to use tables to lay out the form for our ledger entry, the row for the description field could look like listing 6.26:

Listing 6.26: Custom form template

```
<tr>
  <th>< lift:field_label  name="description" /></th>
  <td>< lift:field  name="description" /> <lift:field_msg name="description" /></td>
</tr>
```

Technically, the field_msg binding looks up lift messages (section) based on the field's uniqueId, so you can set your own messages outside of validation using the S.{error,notice,warning} methods as shown in listing 6.27:

Listing 6.27: Setting messages via S

```
S.warning(myEntry.amount.uniqueFieldId,
         "You have entered a negative amount!")
S.warning("amount_id", "This is brittle")
```

For most purposes, though, using the validation mechanism discussed in the next section would be the appropriate way to handle error checking and reporting.

6.2.3 Validation

Validation is the process of checking a field during form processing to make sure that the submitted value meets requirements. This can be something as simple as ensuring that a value was submitted, or as complex as comparing multiple field values together. Validation is achieved via a `List` of functions on a field that take the field value as input and return a `List[FieldError]]` (`Can[Node]` in Record). To indicate that validation succeeded, simply return an empty List, otherwise the list of `FieldErrors` you return are used as the failure messages to be presented to the user. A FieldError is simply a case class that associates an error message with a particular field. As an example, let's say we don't want someone to be able to add a ledger entry in the future. First, we need to define a function for our date field that takes a Date as an input (For Record, java.util.Calendar and not Date is the actual value type of DateTimeField) and returns the proper List. We show a simple function in listing 6.28. In the method we simply check to see if the millisecond count is greater than "now" and return an error message if so.

Listing 6.28: Date validation

```
def noFutureDates (time : java.util.Calendar) = {
  if (time.getTimeInMillis > System.currentTimeMillis) {
    List(FieldError(this, "You cannot make future ledger entries"))
  } else {
    List[FieldError]()
  }
}
```

The next step is to tie the validation into the field itself. We do this by slightly modifying our field definition for `date` to set our list of validators as shown in listing 6.29:

Listing 6.29: Setting validators

```
object date extends DateTimeField[Entry](this) {
  override def validations = noFutureDates _ :: Nil
}
```

Note that we need to add the underscore for each validation function to be partially applied on the submitted value. When our form is submitted, all of the validators for each field are run, and if all of them return Empty then validation succeeds. If any validators return a Full Can, then the contents of the Can are displayed as error messages to the user.

6.2.4 CRUD Support

Adding CRUD support to your Mapper classes is very simple. We just mix in the CRUDify trait to our class and it provides a full set of add, edit, list, delete and view pages automatically. Listing 6.30 shows our Entry class with CRUDify mixed in.

Listing 6.30: Mixing in CRUDify

```
class Entry extends KeyedMapper[Long,Entry] with CRUDify[Long,Entry] {
  ... normal def here ...
  // disable delete functionality
  override def deleteMenuLoc = Empty
}
```

The CRUDify behavior is very flexible, with plenty of defs you can override to control the templates for pages or whether pages are shown at all (as we do in our example). CRUDify automatically creates a set of menus for SiteMap (section) that we can use by appending them onto the rest of our menus as shown in listing 6.31.

Listing 6.31: Using CRUDify Menus

```
val menus = ... Menu(Loc(...)) :: Entry.menus :: Nil
LiftRules.setSiteMap(SiteMap(menus : _*))
```

6.2.5 Mutable vs Immutable on Record

Immutability of instances is an important property that programmers can use to simplify thread-safety; if an object is immutable then you don't need to worry about locking between multiple threads since no thread can modify the object. Record (but not Mapper) supports this via the `mutable_?` method on MetaRecord. By default, Records are mutable, so if you need to use multiple threads (with actors, for instance) you need to be careful about changing the fields of a given Record instance. If you want a Record type to be immutable then simply override the `mutable_?` method to return false. Note that making a record immutable doesn't mean that you can't change values on it, rather that changing the value of a field actually generates a new instance that copies

over the original instances values and uses the new value for the given field. An example of using immutable fields is shown in listing 6.32:

Listing 6.32: Immutable fields

```
val firstDesc = myEntry.description("one")
val secondDesc = firstDesc("two")
println(firstDesc.value) // prints "one"
println(secondDesc.value) // prints "two"
```

Record's chaining of field operation makes this essentially transparent in general use.

6.2.6 Lifecycle Callbacks

Mapper and Record provide for a set of callbacks that allow you to perform actions at various points during the lifecycle of a given instance. If you want to define your own handling for one of the lifecycle events, simply add the LifecycleCallbacks trait to your object. Note that there is a separate LifecycleCallback trait in each of the record and mapper pacakges, so make sure that you import the correct one. For example, if we wanted to notify a comet actor whenever a new ledger entry is saved, we could change our Entry as shown in listing 6.33:

Listing 6.33: Lifecycle callbacks

```
object Entry extends Mapper[Entry] with LifecycleCallbacks {
  ...
  override def afterSave { myActor ! this }
}
```

The lifecycle hooks come at the main operations in an instance lifecycle:

- Create - When a new instance is created
- Delete - When an instance is deleted
- Save - When a fresh instance is first saved (corresponding to a table insert)
- Update - When an instance that already exists in the database is updated (corresponding to a table update)
- Validation - When form validation occurs.

For each of these points you can execute your code before or after the operation is run.

6.2.7 Base Field Types

The Record and Mapper frameworks define several basic field types. The following table shows the corresponding types between Mapper and Record, as well as a brief description of each type.

Mapper	Record	Notes

Mapper	Record	Notes
MappedBinary	BinaryField	Represents a byte array. You must provide your own overrides for toForm and asXHtml/asHtml for input and display
MappedBirthYear	N/A	Holds an Int that represents a birth year. The constructor takes a minAge parameter that is used for validation
MappedBoolean	BooleanField	Represents a Boolean value. The default form representation is a checkbox
MappedCountry	CountryField	Represents a choice from an enumeration of country phone codes as provided by the net.liftweb.mapper.Countries.I18NCountry class. The default form representation is a select
MappedDateTime	DateTimeField	Represents a timestamp (java.util.Calender for Record, java.util.Date for Mapper). The default form representation is a text input
MappedDouble	DoubleField	Represents a Double value
MappedEmail	EmailField	Represents an email address with a maximum length
MappedEnum	EnumField	Represents a choice from a given scala.Enumeration. The default form representation is a select
MappedEnumList	N/A	Represents a choice of multiple Enumerations. The default form representation is a set of checkboxes, one for each enum value
MappedFakeClob	N/A	Fakes a CLOB value (really stores String bytes to a BINARY column)
MappedGender	N/A	Represents a Gender enumeration. Display values are localized via the I18NGenders.
MappedInt	IntField	Represents an Int value
MappedIntIndex	N/A	Represents an indexed Int field (typically a primary key). In Record this is achieved with the KeyField trait
MappedLocale	LocaleField	Represents a locale as selected from the java.util.Locale.getAvailableLocales method. The default form representation is a select
MappedLong	LongField	Represents a Long value
MappedLongForeignKey	N/A	Represents a mapping to another entity via the other entities Long primary key. This functionality in Record is not yet supported
MappedLongIndex	N/A	Represents an indexed Long field (typically a primary key). In Record this is achieved with the KeyField trait
MappedPassword	PasswordField	Represents a password string. The default form representation is a password input (obscured text)
MappedPoliteString	N/A	Just like MappedString, but the default value is an empty string and the input is automatically truncated to fit the database column size

Mapper	Record	Notes
MappedPostalCode	PostalCodeField	Represents a validated postal code string. The field takes a reference to a MappedCountry (CountryField in Record) at definition and validates the input string against the selected country's postal code format
MappedString	StringField	Represents a string value with a maximum length and optional default value
MappedStringForeignKey	N/A	Represents a mapping to another entity via the other entities String primary key. This functionality in Record is not yet supported
MappedStringIndex	N/A	Represents an indexed String field (typically a primary key). In Record this is achieved with the KeyField trait
MappedText	N/A	Represents a String field that stores to a CLOB column in the database. This can be used for large volumes of text.
MappedTextArea	TextAreaField	Represents a String field that will use an HTML textarea element for its form display. When you define the field you can override the textareaCols and textareaRows defs to control the dimensions of the textarea.
MappedTimeZone	TimeZoneField	Represents a time zone selected from java.util.TimeZone.getAvailableIDs. The default form representation is a select
MappedUniqueId	N/A	Represents a unique string of a specified length that is randomly generated. The implementation doesn't allow the user to write new values to the field. This can be thought of as a GUID field.

6.2.8 Defining Custom Field Types in Mapper

The basic MappedField types cover a wide range of needs, but sometimes you may find yourself wanting to cover a specific type. In our example, we would like a decimal value for our ledger amount. Using a double would be inappropriate due to imprecision and rounding errors[2], so instead we base it on BigDecimal. Our first task is to specify the class signature and constructors, as shown in listing 6.34.

Listing 6.34: MappedDecimal constructors

```
class MappedDecimal[OwnerType <: MappedField[T]]
    (val owner : OwnerType, scale : Int)
    extends MappedField[BigDecimal,T] {
  def this(rec : OwnerType, newVal : BigDecimal) = {
    this(rec, newVal.scale)
    set(newVal)
  }
  var rounding = RoundingMode.HALF_EVEN
```

[2] http://stephan.reposita.org/archives/2008/01/11/once-and-for-all-do-not-use-double-for-money/

The first part of the class definition is the type signature; basically the type `[T <: MappedField[T]]` indicates that whatever type "owns" this field must be a Mapper subclass (<: specifies an upper type bound). With our constructor we specify the owner record as well as the "scale" of the decimal value. The scale in BigDecimal essentially represents the number of digits to the right of the decimal point. In addition, we specify a second constructor that allows us to initialize the field to a value. This second constructor uses the scale of the provided value as the default. At the end of the listing we provide a var to control the rounding mode of the field.

Now that we have the constructors in place, there are several abstract methods on MappedField that we need to define. The first of these is a method to provide a default value. The default value is used for uninitialized fields or if validation failes. We also need to specify the class for our value type by implementing the `dbFieldClass` method. Listing 6.35 shows both of these methods. In our case, we default to a zero value, with the scale set as specified in the contructor. We also provide the vars and methods that handle the before and after values of the field. These values are used to handle persistence state; if you change the value of the field, the original value is held until the instance is saved to the database.

Listing 6.35: Setting a default value

```
def defaultValue = (new BigDecimal("0")).setScale(scale)
def dbFieldClass = classOf[BigDecimal]

private var data : BigDecimal = defaultValue
private var orgData : BigDecimal = defaultValue
private def st (in : BigDecimal) {
  data = in
  orgData = in
}
protected def i_is_! = data
protected def i_was_! = orgData
override def doneWithSave() {
  orgData = data
}
```

The next set of methods we need to provide deal with when and how we can access the data. Listing 6.36 shows the overrides that set the read and write permissions to true (default to false for both) as well as the i_obscure_! and real_i_set_! methods. The i_obscure_! method returns the a value that is used then the user doesn't have read permissions. The real_i_set_! method is what actually stores the internal value and sets the dirty flag when the field is updated.

Listing 6.36: Access Control

```
override def readPermission_? = true
override def writePermission_? = true
protected def i_obscure_!(in : BigDecimal) = defaultValue
protected def real_i_set_!(value : BigDecimal): BigDecimal = {
  if (value != data) {
    data = value
    dirty_?(true)
  }
  data
```

```
}
```

The next two methods that we need to provide deal with actually setting the value of the field. The first is `setFromAny`, which takes an `Any` parameter and must convert it into a `BigDecimal`. The second, `setFromString` is a subset of `setFromAny` in that it takes a `String` parameter and must return a `BigDecimal`. Our implementation of these two methods is shown in listing 6.37. We've also added a setAll method so that we have a common place to properly set scale and rounding modes on the value of the field.

Listing 6.37: setFrom... Methods

```
def setFromAny (in : Any) : BigDecimal =
  in match {
    case n :: _ => setFromString(n.toString)
    case Some(n) => setFromString(n.toString)
    case Full(n) => setFromString(n.toString)
    case None | Empty | Failure(_, _, _) | null => setFromString("0")
    case n => setFromString(n.toString)
  }

def setFromString (in : String) : BigDecimal = {
  this.setAll(new BigDecimal(in))
}

protected def setAll (in : BigDecimal) = set(in.setScale(scale, rounding))
```

Our implementations are relatively straightforward. The only special handling we need for setFromAny is to properly deal with Lists, Cans, Options and the null value. The BigDecimal constructor takes Strings, so the setFromString method is easy. The only addition we make over the BigDecimal constructor is to properly set the scale and rounding on the returned value.

Our final step is to define the database-specific methods for our field, as shown in listing 6.38. The first method we implement is targetSQLType. This method tells Mapper what the corresponding SQL type is for our database column. The jdbcFriendly method returns a value that can be used in a JDBC statement; since BigDecimal is a type directly supported by JDBC we simply delegate to the internal value method, `i_is_!`. The buildSet... methods return functions that can be used to set the value of our field based on different input types. These are essentially conversion functions. Finally, the fieldCreatorString specifices what we would need in a CREATE TABLE statement to define this column.

Listing 6.38: Database-Specific Methods

```
def targetSQLType = Types.DECIMAL
def jdbcFriendly(field : String) = i_is_!
def buildSetBooleanValue(accessor : Method, columnName : String) : (T, Boolean, Boolean) =>
     Unit = null
def buildSetDateValue(accessor : Method, columnName : String) : (T, Date) => Unit =
  (inst, v) => doField(inst, accessor, {case f: MappedDecimal[T] => f.st(if (v == null)
       defaultValue else (new BigDecimal(v.getTime).setScale(scale)))}) def
       buildSetStringValue(accessor: Method, columnName: String): (T, String) => Unit =
```

```
6   (inst, v) => doField(inst, accessor, {case f: MappedDecimal[T] => f.st(new BigDecimal(v).
        setScale(scale))})
7 def buildSetLongValue(accessor: Method, columnName : String) : (T, Long, Boolean) =>
8   Unit = (inst, v, isNull) => doField(inst, accessor, {case f: MappedDecimal[T] => f.st(if
        (isNull) defaultValue else (new BigDecimal(v).setScale(scale)))}) def
        buildSetActualValue(accessor: Method, data: AnyRef, columnName: String) : (T, AnyRef
        ) =>
9   Unit = (inst, v) => doField(inst, accessor, {case f: MappedDecimal[T] => f.st(new
        BigDecimal(v.toString).setScale(scale))})
10 def fieldCreatorString(dbType: DriverType, colName: String): String = colName + " DECIMAL
     (20," + scale + ")"
```

To make the field a little more useful, we can also specify some convenience methods on it to make arithmetic a little more natural, as shown in listing

Listing 6.39: Convenience methods

```
def += (other : BigDecimal) = setAll(data.add(other))
def -= (other : BigDecimal) = setAll(data.subtract(other))
def *= (other : BigDecimal) = setAll(data.multiply(other))
def /= (other : BigDecimal) = setAll(data.divide(other))

// Use in practice:
myEntry.amount *= BigDecimal.valueOf(2)
```

You could expand on the convenience methods as you wish to cover Ints, Longs, Doubles, etc.

6.2.9 Defining Custom Field Types in Record

Similar to the example in section 6.2.8, when using Record we would like a DecimalField to represent decimal currency amounts. Our implementation starts off looking very similar to the Mapper version, as shown in listing 6.40. The main difference so far is that the owner type has to be a subtype of Record instead of Mapper.

Listing 6.40: DecimalField Constructors

```
class DecimalField[OwnerType <: Record[OwnerType]]
    (rec : OwnerType, scale : Int)
    extends Field[BigDecimal,OwnerType] {
  def this(rec : OwnerType, newVal : BigDecimal) = {
    this(rec, newVal.scale)
    set(newVal)
  }
  var rounding = RoundingMode.HALF_EVEN
```

The next step is to start defining the methods that are left abstract on Field. Our first set deals with some basic housekeeping and presentation layer details, shown in listing 6.41. The defaultValue method simply defines the value that we use for new instances. The owner method lets Record know which Record instance owns this field. asXHtml and toForm define the methods for display and form generation, respectively. We use the setFromString method (defined later in this

section) to handle form setting, which keeps the logic clean.

Listing 6.41: DecimalField Methods

```
def defaultValue = (new BigDecimal("0")).setScale(scale)
def owner = rec
def asXHtml = Text(value.toString)
def toForm = text(value.toString, this.setFromString)
```

Our final set of methods deals with setting the value of the field, shown in listing 6.42. The setFromAny method, and its counterpart setFromString, have replaced the buildSet... method functionality that we would define in Mapper. setFromAny essentially defers to setFromString for all operations, and the only special handling we need is for Lists, Options and Cans. setFromString is required to catch any exceptions thrown during conversion; if we have a failure we need to return an Empty as well as calling couldNotSetValue to set a flag for validation.

Listing 6.42: DecimalField Set Methods

```
def setFromAny (in : Any) : Can[BigDecimal] =
  in match {
    case n :: _ => setFromString(n.toString)
    case Some(n) => setFromString(n.toString)
    case Full(n) => setFromString(n.toString)
    case None | Empty | Failure(_, _, _) | null => setFromString("0")
    case n => setFromString(n.toString)
  }
def setFromString (in : String) : Can[BigDecimal] = {
  try {
    Full(this.setAll(new BigDecimal(in)))
  } catch {
    case e : Exception => couldNotSetValue; Empty
  }
}
/** Set the value along with proper scale and rounding */
protected def setAll (in : BigDecimal) = set(in.setScale(scale,rounding))
```

Since BigDecimal takes a string as a constructor (in fact, this is the recommended way to initialize it to avoid precision issues with numerics), the setFromString method is relatively easy to implement. To make things even simpler, we add the setAll method that takes a BigDecimal as input and then sets the proper scale and rounding on it.

6.3 Record Backends

6.3.1 JDBC Records

TBD

6.3.2 JPA Records

TBD

6.4 Advanced Features

In this section we'll cover some of the advanced features of Mapper

6.4.1 Using Multiple Databases

It's common for an application to need to access data in more than one database. Lift supports this feature through the use of overrides on your MetaMapper classes. First, we need to define the identifiers for the various databases using the ConnectionIdentifier trait and overriding the jndiName def. Lift comes with one pre-made: DefaultConnectionIdentifier. It's jndiName is set to "lift", so it's recommended that you use something else. Let's say we have two databases: sales and employees. Listing 6.43 shows how we would define the ConnectionIdentifier objects for these.

Listing 6.43: Defining Connection Identifiers

```
object SalesDB extends ConnectionIdentifier {
  def jndiName = "sales"
}

object EmployeeDB extends ConnectionIdentifier {
  def jndiName = "employees"
}
```

Simple enough. Now, we need to create connection managers for each one, or we can combine the functionality into a single manager. To keep things clean we'll use a single manager, as shown in listing 6.44. Scala's match operator allows us to easily return the correct connection.

Listing 6.44: Multi-database Connection Manager

```
object DBVendor extends ConnectionManager {
  Class.forName("org.postgresql.Driver")

  def newConnection(name : ConnectionIdentifier) = {
    try {
      name match {
        case SalesDB =>
          Full(DriverManager.getConnection(
            "jdbc:postgresql://localhost/sales",
            "root", "secret"))
        case EmployeeDB =>
          Full(DriverManager.getConnection(
            "jdbc:postgresql://server/employees",
            "root", "hidden"))
    } catch {
      case e : Exception => e.printStackTrace; Empty
    }
  }
  def releaseConnection (conn : Connection) { conn.close }
}
```

A special case of using multiple databases is *sharding*[3]. Sharding is a means to scale your database capacity by associating entities with one database instance out of a federation of servers based on some property of the entity. For instance, we could distribute user entites across 3 database servers by using the first character of the last name: A-H goes to server 1, I-P goes to server 2, and Q-Z goes to server 3. As simple as this sounds, there are some important factors to remember:

- Sharding increases the complexity of your code a little
- To get the most benefit out of sharding, you need to carefully choose and tune your "selector". If you're not careful you can get an uneven distribution where some servers handle significantly more load than others, defeating the purpose of sharding. If you decide to retune your selctor later on you need to redistribute the data to the proper servers in concert with changing your code or you won't be able to find things
- When you use sharding, you can't just use normal joins anymore, since the data isn't all within one instance. This means more work on your part to properly retrieve and associate data

Mapper provides a handy feature for sharding that allows you to choose which database connection you want to use for a specific entity. There are two methods we can use to control the behavior: `dbSelectDBConnectionForFind`, and `dbCalculateConnectionIdentifier`. dbSelect... is used in a find by primary key, and takes a partial function (typically a match clause) to determine which connection to use. dbCalculate... is used when a new instance is created to decide where to store the new instance. As an example, say we've defined two database connections, SalesA and SalesB. We want to place new instances in SalesA if the amount is > $100 and SalesB otherwise. Listing 6.45 shows our method in action.

Listing 6.45: Sharding in Action

```
class Entry extends Mapper[Entry] {
  ... fields, etc ...

  override def dbCalculateConnectionIdentifier = {
    case n if n.amount.is > 100 => SalesA
    case _ => SalesB
  }
}
```

6.4.2 SQL-based Queries

If, despite all that Mapper covers, you find yourself still wanting more control over the query, there are two more options available to you: findAllByPreparedStatement and findAllByInsecureSql. The findAllByPreparedStatement method allows you to, in essence, construct your query completely by hand. The added benefit of using a PreparedStatement[4] means that you can easily include user-defined data in your queries. The findAllByPreparedStatement method takes a single

[3]For more information on sharding, see this article: http://highscalability.com/unorthodox-approach-database-design-coming-shard

[4]http://java.sun.com/javase/6/docs/api/java/sql/PreparedStatement.html

function parameter; this function needs to take a SuperConnection[5] and return a PreparedStatement instance. Listing 6.46 shows our previous example of looking up all Categories for a recent ledger entries using findAllByPreparedStatement instead. The query that you provide must at least return the fields that are mapped by your entity, but you can return other columns as well (they'll just be ignored) in case you just want to do a "select *".

Listing 6.46: Using findAllByPreparedStatement

```
def recentCategories = CategoryMeta.findAllByPreparedStatement({ superconn =>
  superconn.connection.prepareStatement(
    "select distinct category.id, category.name " +
    "from Category category " +
    "join CategoryJoin cj on category.id = cj.category " +
    "join Entry entry on entry.id = cj.entry " +
    "where entry.date > (CURRENT_DATE - interval '30 days')")
})
```

The findAllByInsecureSql method goes even further, executing the string you submit directly as a statement without any checks. The same general rules apply as for findAllByPreparedStatement, although you need to add the IHaveValidatedThisSQL parameter as a code audit check. In either case, the ability to use full SQL queries can allow you to do some very powerful things, but it comes at the cost of losing type safety and possibly making your app non-portable.

[5] Essentially a thin wrapper on java.sql.Connection, http://scala-tools.org/mvnsites/liftweb/lift-webkit/scaladocs/net/liftweb/mapper/SuperConnection.html

Part II

Advanced Topics

Chapter 7

Request/Response Lifecycle

Planning:
Detailed request/response handling info (should cover how Actors are used, too)

1. Request hits LiftFilter
2. LiftFilter executes each item in LiftRules.early. This would allow you to do advance processing of the HttpRequest
3. Rewrites occur per the LiftRules.rewriteTable (set up via LiftRules.addRewriteBefore/After)
4. LiftFilter determines whether the request should be handled by Lift or if it should be chained. If the processing fails it's automatically chained
5. Processing takes place via the LiftFilter's internal LiftServlet instance
6. If Lift is running in Jetty, any continuations are invoked (explain a little about Jetty continuations here...). If continuation exists and returns a response, return
7. LiftServlet checks LiftRules.statelessDispatchTable and returns if matched (what is this for?)
8. LiftServlet checks LiftRules.dispatchTable (set up via LiftRules.addDispatchBefore/After) and if dispatches match it dispatches there and returns the result * detail dispatch handling
9. LiftServlet then checks to see if the request starts with LiftRules.cometPath (default "/comet_request") and if so, handles the request as a COMET request * detail COMET handling path
10. LiftServlet then checks to see if the request starts with LiftRules.ajaxPath (default "/ajax_request") and if so, handles the request as an AJAX request * detail AJAX handling
11. If nothing else has occurred, do normal template processing at this point
    (a) Lookup template based on path
    (b) Process template recursively (surround, include, etc)

* Additional topics
    * S.addAround and LoanWrapper
    * How S.attr works in conjunction with XML attributes on snippets, etc and with Rewriting
    This is going to be super-helpful for the later chapters, like WebServices and Rewriting.

Chapter 8

URL Rewriting

What is URL rewriting

- Using LiftRules.addRewriteBefore.After
- What constitutes a rewrite function?
- Using the params Map to pass parts of the path back into your code via S.param
- Example: user-friendly URLs
- Example: REST API pattern

Chapter 9

JSON Handling

10. JSON handling
* What is JSON?
* Why is JSON awesome? [example]
* Technically, just a data format for Javascript (http://www.json.org/)
* Used as an RPC transfer format
* Well-suited to AJAX because it's essentially javascript
* How does Lift support JSON?
* JsonHandler allows simple wrapping and processing of JSON (AJAX) forms
* JsonCmd allows matching of submitted JSON
* Go through example JSON/AJAX submission form. Maybe it would be best to make it an extension of one of the other examples?

JavaScript Object Notation, or JSON, was designed to be a simple data format. Simple for both humans and computers.

A quick overview of JSON from http://www.json.org

Here's the RFC: http://tools.ietf.org/html/rfc4627

JSON is built on two structures:

* A collection of name/value pairs. In various languages, this is realized as an object, record, struct, dictionary, hash table, keyed list, or associative array.

* An ordered list of values. In most languages, this is realized as an array, vector, list, or sequence.

These are universal data structures. Virtually all modern programming languages support them in one form or another. It makes sense that a data format that is interchangable with programming languages also be based on these structures.

In JSON, they take on these forms:

An object is an unordered set of name/value pairs. An object begins with { (left brace) and ends with } (right brace). Each name is followed by : (colon) and the name/value pairs are separated by , (comma).

Below is a JSON representation of an address book entry, note it's simplicity.

```
{
"firstName": "John",
```

```
"lastName": "Smith",
"address": {
  "streetAddress": "21 2nd Street",
  "city": "New York",
  "state": "NY",
  "postalCode": 10021
  },
"phoneNumbers": [ "212 555-1234", "646 555-4567" ]
}
```

Lift has extensive support for JSON that is made available by the JSON parser in Scala (scala.util.parsing.json)

JSONResponse
toJSON
JSONForms

Chapter 10

JsCommands

11. JsCommands

- Integrated javascript handling without hard-coding it in templates
- Further the goal of pushing logic out of templates
- jQuery intro: Steal from this: http://docs.jquery.com/How_jQuery_Works
- Utility library that makes IE6 less ugly ;)
- Powerful support for dynamic attributing of elements
- Based on CSS selectors
- Talk about Marius JS abstraction for YUI and jQuery

Chapter 11

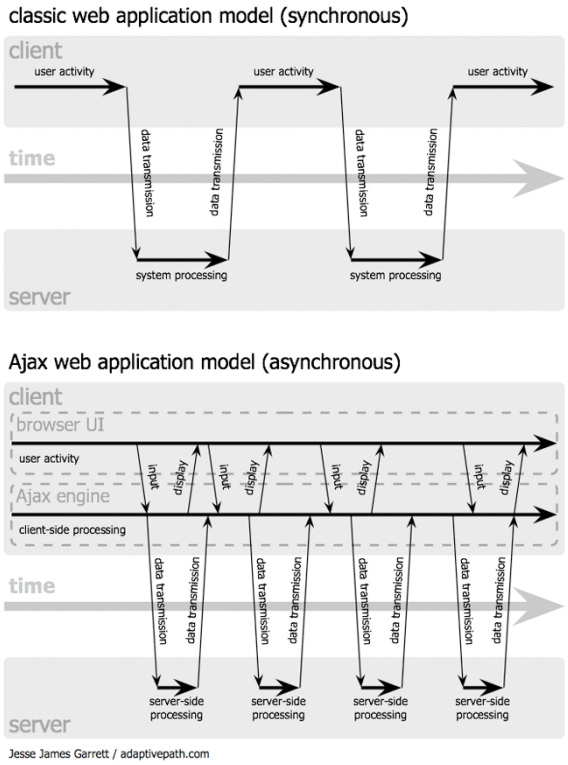
AJAX and COMET

- This is probably the place for the Spreadsheet example
- Talk about design "patterns" ex. the proper TagCloud example
- http://en.wikipedia.org/wiki/Comet_(programming)
- http://en.wikipedia.org/wiki/AJAX_(programming)
- Jorge gave us permission to use his scala-blogs.com example

11.1 Traditional Browser/Server Interaction

AJAX and Comet are variations on the traditional model of web application interaction. The traditional model starts with the user making a request for a page, the server doing some amount of processing and then sending the appropriate response. This model is common today but models are evolving to include AJAX and Comet. AJAX and Comet differ in implementation of the traditional model. AJAX adds the concept of asynchronisity. This translates into a more responsive User Interface for a web application. If we take the example of adding a comment to a blog post, the traditional model has the user fill in a form, hit the submit button to send the request to the server, the server processes and add the comment and sends the updated blog post with the newly added comment. The AJAX model of this session would change such that the display of the new comment is not tied to the response from the server. After the user hits submit, the request is sent to the server and while it's being processed, the User Interface has responded by added the comment without the need of a full page request.

Comet takes the AJAX model and adds to it with the idea of multiple users. The AJAX model increases the richness of the User Experience for a single client at a time. If we think about the previous example but add another person looking at the same page, they will not see the new comment until they refresh the post. Comet changes this by enabling the server to push to multiple clients, so the previous example changes such that when Client A hits submit, that new comment is pushed to Client B's browser, without the need to refresh the page.



(a) AJAX Application Model

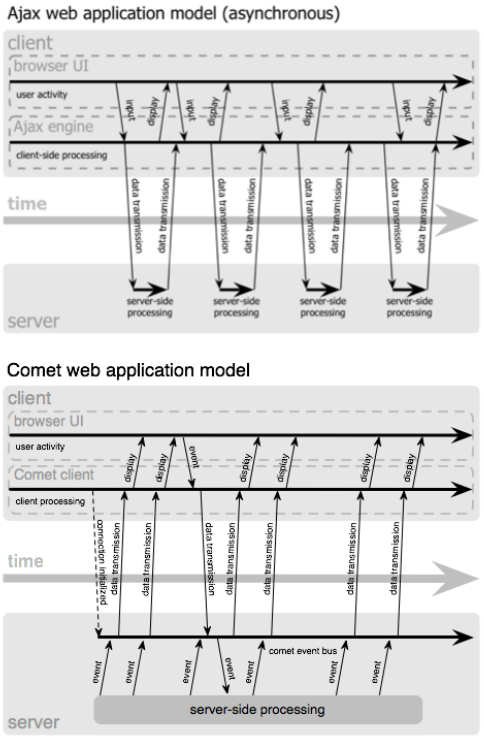
Figure 11.1: AJAX Architecture

11.2 AJAX in Lift

/ajax_request

11.3 COMET in Lift

- /comet_request
- What happens to the snippet code



(a) COMET Application Model

Figure 11.2: COMET Architecture

Chapter 12

JPA Integration

The Java Persistence API[1], or JPA for short, is the evolution of a number of frameworks in Java to provide a simple database access layer for plain java objects (and, transitively, Scala objects). JPA was developed as part of the EJB3 specification, with the goal of simplifying the persistence model. Prior versions had used the Container Managed Persistence (CMP) framework, which required many boilerplate artifacts in the form of interfaces and XML descriptors. As part of the overarching theme of EJB3 to simplify and use configuration by convention, JPA uses annotations heavily, while allowing for targetted overrides of behavior via XML descriptors. JPA also does away with many of the interfaces used in CMP and provides a single EntityManager object for all persistence operations. An additional benefit is that JPA was designed so that it could be used both inside and outside of the Enterprise container, and several projects (Hibernate, TopLink, JPOX, etc) provide standalone implementations of EntityManager.

As we've seen in chapter 6, Lift already comes with a very capable database abstraction layer, so why would we want to use something else? There are a number of reasons:

1. JPA is easily accessible from both Java and Scala. If you are using Lift to complement part of a project that also contains Java components, JPA allows you to use a common database layer between both and avoid duplication of effort. It also means that if you have an existing project based on JPA, you can easily integrate it into Lift

2. JPA gives you more flexibility with complex and/or large schemas. While Lift's Recordprovides most of the functionality you would need, JPA provides additional lifecycle methods and mapping controls when you have complex needs

3. JPA can provide additional performance improvements via second-level object caching. It's possible to roll your own in Lift, but JPA allows you to cache frequently-accessed objects in memory so that you avoid hitting the database entirely

4. JPA and Lift's Record are not mutually exclusive. You can use an adapter to allow Record to utilize JPA as the underlying provider, thus gaining the best of both worlds.

[1] http://java.sun.com/javaee/overview/faq/persistence.jsp

12.1 Introduction to JPA

In order to provide a concrete example to build on while learning how to integrate JPA, we'll be building a small Lift app to manage a library of books. The completed example is available under the Lift Git repository in the sites directory, and is called "JPADemo". Basic coverage of the JPA operations is in section 12.4 on page 84; if you want more detail on JPA, particularly with advanced topics like locking and hinting, there are several very good tutorials to be found online[2]. Our first step is to set up a master project for Maven. This project will have two modules under it, one for the JPA library and one for the Lift application. The only thing we actually need for the master project is the pom.xml. The complete pom.xml is shown in section H.1 on page 120. Note that the dependencies specifically exclude the Hibernate JTA dependence so that we can use the geronimo-specs version of JTA. If you don't do that then you'll have to manually download and install the Sun versions from their website.

The master pom.xml sets up some basic dependencies and plugin configurations which the modules will inherit. Our next step will be to create the JPA module using an archetype for Maven to create the scaffolding of our project. From your master project directory, use the `mvn archetype:generate` command, select the "Scala JPA" archetype and when prompted use com.foo.jpaweb for the group ID, JPADemo-spa for the artifact ID, and com.foo.jpaweb.model for the package. You should now have a JPADemo-spa module that we can explore.

12.1.1 Entity Classes in Scala

The main components of a JPA library are the entity classes that comprise your data model. For our example application we need two primary entities: Author and Book. Let's take a look at the Author class first, shown in listing H.2 on page 122. The listing shows our import of the entire javax.persistence package as well as several annotations on a basic class. For those of you coming from the Java world in JPA, the annotations should look very familiar. The major difference between Java and Scala annotations is that each parameter in a Scala annotation is considered a val, which explains the presence of the val keyword in lines 11, 14 and 17-18. In line 18 you may also note that we must specify the target entity class; although Scala uses generics, the generic types aren't visible from Java, so the Java JPA libraries can't deduce the correct type. You may also notice that on line 19 we need to use the Java collections classes for Set, List, etc. With a little bit of implicit conversion magic (to be shown later), this has very little impact on our code. On final item item to note is that the Scala compiler currently does not support nested annotations [3], so where we would normally use them (join tables, named queries, etc), we will have to use the orm.xml descriptor, which we cover next.

12.1.2 Using the orm.xml descriptor

As we stated in the last section, there are some instances where the Scala compiler doesn't fully cover the JPA annotations (nested annotations in particular). Some would also argue that queries and other ancillary data (table names, column names, etc) should be separate from code. Because of that, JPA allows you to specify an external mapping descriptor to define and/or override the

[2] http://java.sun.com/developer/technicalArticles/J2EE/jpa/, http://www.jpox.org/docs/1_2/tutorials/jpa_tutorial.html

[3] https://lampsvn.epfl.ch/trac/scala/ticket/294

Listing 12.1: Author override

```
<entity  class="Author">
  <named−query name="findAllAuthors">
    <query><![CDATA[from Author a order by a.name]]></query>
  </named−query>
  <attribute−override name="name">
    <column name="author_name" length="30" />
  </attribute−override>
</entity>
```

mappings for your entity classes. The basic orm.xml file starts with the DTD type declaration, as shown in listing H.3 on page 123. Following the preamble, we can define a package that will apply to all subsequent entries so that we don't need to use the fully-qualified name for each class. In our example, we would like to define some named queries for each class. Putting them in the orm.xml allows us to modify them without requiring a recompile. The complete XML Schema Definition can be found at `http://java.sun.com/xml/ns/persistence/orm_1_0.xsd`.

In this case we have used the orm.xml file to augment our entity classes. If, however, we would like to override the configuration, we may use that as well on a case-by-case basis. Suppose we wished to change the column name for the Author's name property. We can add (per the XSD) a section to the Author entity element as shown in listing 12.1. The `attribute-override` element lets us change anything that we would normally specify on the `@Column` annotation. This gives us an extremely powerful method for controlling our schema mapping outside of the source code. We can also add named queries in the orm.xml so that we have a central location for defining or altering the queries.

12.1.3 Object attachment and detachment

JPA operates with entities in one of two modes: attached and detached. An attached object is one that is under the direct control of a live JPA session. That means that the JPA provider monitors the state of the object and writes it to the database at the appropriate time. Objects can be attached either explicitly via the `persist` and `merge` methods (section 12.4.1), or implicitly via query results, the `getReference` method or the `find` method. As soon as the session ends, any formerly attached objects are now considered detached. You can still operate on them as normal objects but any changes are not directly applied to the database. If you have a detached object, you can re-attach it to your current session with the `merge` method; any changes since the object was detached, as well as any subsequent changes to the attached object, will be applied to the database at the appropriate time. The concept of object attachment is particularly useful in Lift because it allows us to generate or query for an object in one request cycle and then make modifications and merge in a different cycle.

12.1.4 Support for user types

JPA can handle any Java primitive type, their corresponding Object versions (java.lang.Long, java.lang.Integer, etc), and any entity classes comprised of these types [4]. Occasionally, though, you

[4] It can technically handle more; see the JPA spec, section 2.1.1 for details

Listing 12.2: Genre and GenreType

```
1 object Genre extends Enumeration with Enumv {
2   val Mystery = Value("Mystery", "Mystery")
3   val Science = Value("Science",  "Science")
4   val Theater = Value("Theater",  "Drama literature")
5   // more values here ...
6 }
7
8 class GenreType extends EnumvType(Genre) {}
```

may have a requirement for a type that doesn't fit directly with those specifications. One example in particular would be Scala's enumerations. Unfortunately, the JPA spec currently doesn't have a means to handle this directly, although the various JPA providers such as Toplink and Hibernate provide mechanisms for resolving custom user types. JPA does provide direct support for *Java* enumerations, but that doesn't help us here since Scala enumerations aren't an extension of Java enumerations. In this example, we'll be using Hibernate's UserType to support an enumeration for the Genre of a Book.

We begin by implementing a few helper classes besides the Genre enumeration itself. First, we define an Enumv trait, shown in listing H.4 on page 124. Its main purpose is to provide a `valueOf` method that we can use to resolve the enumerations database value to the actual enumeration. We also add some extra methods so that we can encapsulate a description along with the database value. Scala enumerations can use either Ints or Strings for the identity of the enumeration value (unique to each val), and in this case we've chosen Strings. By adding a map for the description (since Scala enumeration values must extend the Enumeration#Value class and therefore can't carry the additional string) we allow for the additional info. We could extend this concept to make the Map carry additional data, but for our purposes this is sufficient.

In order to actually convert the Enumeration class into the proper database type (String, Int, etc), we need to implement the Hibernate UserType interface, shown in listing H.5 on page 125. We can see on line 18 that we will be using a varchar column for the enumeration value. Since this is based on the Scala Enumeration's Value method, we could technically use either Integer or character types here. We override the `sqlTypes` and `returnedClass` methods to match our preferred type, and set the `equals` and `hashCode` methods accordingly. Note that in Scala, the "==" operator on objects delegates to the `equals` method, so we're not testing reference equality here. The actual resolution of database column value to Enumeration is done in the `nullSafeGet` method; if we decided, for instance, that the null value should be returned as unknown, we could do this here with some minor modifications to the Enumv class (defining the unknown value, for one).The rest of the methods are set appropriately for an immutable object (Enumeration). The great thing about the EnumvType class, is that it can easily be used for a variety of types due to the "et" constructor argument; as long as we mix in the Enumv trait to our Enumeration objects, we get persistence essentially for free. If we determined instead that we want to use Integer enumeration IDs, we need to make minor modifications to the EnumvType to make sure arguments match and we're set.

Finally, the Genre object and the associated GenreType is shown in listing 12.2. You can see that we create a singleton Genre object with specific member values for each enumeration value. The GenreType class is trivial now that we have the EnumvType class defined. To use the Genre type in our entity classes, we simply need to add the proper `var` and annotate it with the @Type

Listing 12.3: Using the @Type annotation

```
@Type{val 'type' = "com.foo.jpaweb.model.GenreType"}
var genre : Genre.Value = Genre.unknown
```

annotation, as shown in listing 12.3. We need to specify the type of the var due to the fact that the actual enumeration values are of the type Enumeration.Val, which doesn't match our valueOf method in the Enumv trait. We also want to make sure we set the enumeration to some reasonable default; in our example we have an *unknown* value to cover that case.

Now that we have our objects defined, let's start using them.

12.2 Obtaining a Per-Session EntityManager

Ideally, we would like our JPA access to be as seamless as possible, particularly when it comes to object lifecycle. In JPA, objects can be attached to a current persistence session, or they can be detached from a JPA session. This gives us a lot of flexibility (which we'll use later) in dealing with the objects themselves, but it also means that we need to be careful when we're accessing object properties. JPA can use lazy retrieval for instance properties; in particular, this is the default behavior for collection-based properties. What this means is that if we're working on a detached object and we attempt to access a collection contained in the instance, we're going to get an exception that the session that the object was loaded in is no longer live. What we're really like to do is have some hooks into Lift's request cycle that allows us to set up a session when the request starts and properly close it down when the request ends. We still have to be careful with objects that have been passed into our request (from form callbacks, for instance), but in general this will guarantee us that once we've loaded an object in our snippet code we have full access to all properties at any point within our snippets.

Fortunately for us, Lift provides just such a mechanism. In fact, Lift supports several related mechanisms for lifecycle management[5], but for now we're going to focus on just one: the RequestVar. A RequestVar represents a variable associated with the lifetime of the request. This is in contrast to SessionVar, which defines a variable for the lifetime of the user's session. RequestVar gives us several niceties over handling request parameters ourselves, including type safety and a default value. We go into more detail on Request- and SessionVars in section , but for now we're going to concentrate on the methods needed to do lifecycle management.

The initial setup of the entity manager is done through the RequestVar's constructor. We define a RequestVar as an object (as opposed to a class) as shown in listing 12.4 (the full listing is in section H.8 on page 128). In our example we're using an abstraction of the EntityManager (and Query) interfaces as a template for our concrete implementation; we'll cover these in section 12.3 on page 84. As you can see, we've defined two methods to handle the opening and closing of the EntityManager and we delegate to our impl for the actual setup. We utilize the constructor of the RequestVar to call the openEM() method, and we override the cleanupFunc method to call the closeEM() method. Note that the RequestVar is lazy, so the EntityManager won't be opened until it's actually requested.

[5]Notably, S.addAround with the LoanWrapper

Listing 12.4: Setting up an EntityManager via RequestVar

```
1 abstract class ScalaEntityManager(val persistanceName: String) {
2   // The concrete impl should provide these methods
3   protected def openEM () : EntityManager
4   protected def closeEM (em : EntityManager)
5
6   private object emVar extends RequestVar(openEM()) {
7     override def cleanupFunc : Can[() => Unit]= Full (() => closeEM(this.is))
8   }
```

Now that we've defined the mechanism for setting up and tearing down our EM, we need to actually obtain an EM. There are essentially two ways to do this

1. Construct the EntityManagerFactory directly
2. Obtain an EntityManager via JNDI

Of these two, JNDI offers a cleaner approach from the standpoint of deployment. Using JNDI acts as a form of IoC (Inversion of Control) to allow us to separate out the selection of the persistence module; it's also more in line with JEE standards and allows us to more easily move our app between full-blown (JBoss, GlassFish, etc) and lightweight (Jetty, Tomcat) containers.

12.2.1 Creating a factory directly

Creating a factory directly is fairly simple, and is the use case for apps that will run under the Java SE without access to the JEE facilicities. As shown in listing H.6 on page 126, we first set up the EntityManagerFactory by a call to the static javax.persistence.Persistence.createFactory(...) method. Once we have the factory, we define our openEM method to use the factory's createEntityManager() method to obtain the EM, and then we start a transaction. We're going to cover transactions in more detail a little later in this chapter; for now, just understand that in this particular configuration we have to begin and commit the transaction *within* the lifecycle of the EM. We also define our closeEM method as essentially the mirror of openEM.

12.2.2 Using JNDI

The standard binding for persistence modules in JNDI is `java:comp/env/persistence/<module name>`[6]. We'll cover how to set up JNDI on lightweight containers like Tomcat and Jetty in section G.1 on page 117, but for now we'll concentrate on the client side of things. We define our openEM method using the javax.naming.InitialContext class to obtain the EM from JNDI, as shown in listing H.7 on page 127. You can also see that we start a transaction (also via JNDI) just prior to looking up the EntityManager. When we use JNDI (really, JTA) for obtaining our transactions, notice that we open begin and end the transaction outsid of the EM's lifecycle. In a production application you would want to put these steps into a try/catch or use LiftRules.browserResponseToException (covered in section) to make sure that the user doesn't get a stack trace if something goes wrong.

[6] http://database.in2p3.fr/doc/oracle/Oracle_Application_Server_10_Release_3/web.1013/b28221/servjndi010.htm

Listing 12.5: Setting the transaction type

```
<persistence-unit name="jpaweb" transaction-type="RESOURCE_LOCAL">
  <non-jta-datasource>myDS</non-jta-datasource>

<persistence-unit name="jpaweb" transaction-type="JTA">
  <non-jta-datasource>myDS</jta-datasource>
```

You can also see in the listing that we've defined our closeEM method. We simply close the EM and then either commit or rollback the transaction depending on its state.

However you obtain the EM, the general principles of how you use the EM are the same. All of the entity operations work exactly the same way in either case and the EntityManager takes care of the underlying details, such as how to set rollback on the current transaction. This allows your code to stay portable; if you want to use the simpler user-managed factory you can start with that and when you change your mind later it requires only minor modifications to a single class.

12.2.3 Transactions

We're not going to go into too much detail here; there are better documents available[7] if you want to go into depth on how JTA or general transactions work. Essentially, a transaction is a set of operations that are performed atomically; that is, they either all complete successfully or none of them do. The classic example is transferring funds between two bank accounts: you subtract the amount from one account and add it to the other. If the addition fails and you're not operating in the context of a transaction, the client has lost money!

In JPA, transactions are required. If you don't perform your operations within the scope of a transaction you will either get an exception (if you're using JTA), or you will spend many hours trying to figure out why nothing is being saved to the database. There are two ways of handling transactions under JPA: resource local and JTA. Resource local transactions are what you use if you are managing the EM factory yourself (section H.6). Similarly, JTA is what you use when you obtain your EM via JNDI. Choosing between the two is as simple as setting a property in your persistence.xml file (and changing the code to open and close the EM). Listing 12.5 shows examples of setting the `transaction-type` attribute to `RESOURCE_LOCAL` and to JTA. If you want to use JTA, you can also omit the `transaction-type` attribute since JTA is the default.

You must make sure that your EM setup code matches what you have in your persistence.xml. Additionally, the database connection must match; with JTA, you *must* use a `jta-data-source` (obtained via JNDI) for your database connection. For resource-local, you can either use a `non-jta-datasource` element or you can set the provider properties, as shown in listing 12.6 on the following page. In this particular example we're setting the properties for Hibernate, but similar properties exist for TopLink[8], JPOX[9], and others.

In our opinion, using JTA to manage your transactions is well worth the extra effort. If you'll be deploying into a JEE container, such as JBoss or GlassFish, then you get JTA support almost for free since JTA is part of the JEE spec. Even if you want to deploy your application on a lightweight container like Jetty, the extra effort to set up JNDI and JTA in Jetty is small relative to the effort

[7] http://java.sun.com/developer/EJTechTips/2005/tt0125.html
[8] http://www.oracle.com/technology/products/ias/toplink/JPA/essentials/toplink-jpa-extensions.html
[9] http://www.jpox.org/docs/1_2/persistence_unit.html

Listing 12.6: Setting resource-local properties for Hibernate

```
<persistence>
  <persistence-unit name="jpaweb" transaction-type="RESOURCE_LOCAL">
    <properties>
      <property name="hibernate.dialect" value="org.hibernate.dialect.PostgreSQLDialect"/>
      <property name="hibernate.connection.driver_class" value="org.postgresql.Driver"/>
      <property name="hibernate.connection.username" value="somUser"/>
      <property name="hibernate.connection.password" value="somePass"/>
      <property name="hibernate.connection.url" value="jdbc:postgresql:jpaweb"/>
    </properties>
  </persistence-unit>
</persistence>
```

involved in coding and testing your app. As we've discussed before, you also get the benefit of separation of configuration when you use JNDI.

One final note in regard to transactions is how they're affected by Exceptions. Per the spec, any exceptions thrown during the scope of a transaction, other than `javax.persistence.NoResultException` or `javax.persistence.NonUniqueResultException`, will cause the transaction to be marked for rollback.

12.3 ScalaEntityManager and ScalaQuery

Now that we've gone through setting up our EntityManager, let's look at how we actually use them in an application. As a convenience, we will define two thin wrappers on the existing EntityManager[10] and Query[11] interfaces to provide more Scala-friendly methods. This means that we get Scala's collection types (i.e. List instead of java.util.List) and generic signatures so that we can avoid explicit casting. Listing H.8 on page 128 shows the ScalaEntityManager class. We start with the method and RequestVar definitions that we discussed in section 12.2. Following that, we have a few convenience methods for creating and executing named queries in a single method, as well combo methods to merge or remove an entity and flush to the database in one go. The rest of the methods essentially delegate to the underlying EM directly.

Next, we have the ScalaQuery class in listing H.9 on page 129. Like ScalaEntityManager, this is a thin wrapper on the Query interface. In particular, methods that return entities are typed against the ScalaQuery itself, so that you don't need to do any explicit casting in your client code. We also add some utility methods to simplify setting a parameter list as well as obtaining the result(s) of the query.

12.4 Examples

We find it helpful to solidify what we've discussed here with some concrete examples. Let's start with some basic operations

[10] http://java.sun.com/javaee/5/docs/api/javax/persistence/EntityManager.html
[11] http://java.sun.com/javaee/5/docs/api/javax/persistence/Query.html

12.4.1 Persisting, merging and removing

The first step to working with any persistent entities is to actually persist them. If you have a brand new object, you can do this with the `persist` method:

```
Model.persist(myNewAuthor)
```

This attaches the myNewAuthor object to the current persistence session. Once the object is attached it should be visible in any subsequent queries, although it may not be written to the database just yet (see section 12.4.6). Note that the persist method is only intended for brand new objects. If you have a detached object and you try to use persist you will most likely get an EntityExistsException as the instance you're merging is technically conflicting with itself. Instead, you want to use the merge method to re-attach detached objects:

```
val author = Model.merge(myOldAuthor)
```

An important thing to note is that the merge method doesn't actually attach the object passed to it; instead, it makes an attached *copy* of the passed object and returns the copy. If you mistakenly merge without using the returned value:

```
Model.merge(myOldAuthor)
myOldAuthor.name = "Fred"
```

you'll find that subsequent changes to the object won't be written to the database. One nice aspect of the merge method is that it intelligently detects whether the entity you're merging is a new object or a detached object. That means that you can use merge everywhere and let it sort out the semantics. For example, in our library application, using merge allows us to combine the add and edit functionality into a single snippet; if we want to edit an existing Author we pass it into the method. Otherwise, we pass a brand new Author instance into the method and the merge takes care of either case appropriately.

Removing an object is achieved by calling the `remove` method:

```
Model.remove(myAuthor)
```

The passed entity is detached from the session immediately and will be removed from the database at the appropriate time. If the entity has any associations on it (to collections or other entities), they will be cascaded as indicated by the entity mapping. An example of a cascade is shown in the Author listing on page 122. The books collection has the cascade set to REMOVE, which means that if an author is deleted, all of the books by that author will be removed as well. The default is to not cascade anything, so it's important that you properly set the cascade on collections to avoid constraint violations when you remove entities. It's also useful to point out that you don't actually need to have an entity loaded to remove it. You can use the `getReference` method to obtain a proxy that will cause the corresponding database entry to be removed:

```
Model.remove(Model.getReference(classOf[Author], someId))
```

12.4.2 Loading an Entity

There are actually three ways to load an entity object in your client code. The simplest is to use the `find` method:

```
val myBook = Model.find(classOf[Book], someId)
```

The `find` method takes two parameters: the class that you're trying to load and the value of the ID field of the entity. In our example, the Book class uses the Long type for its ID, so we would put a Long value here. It returns either a Full Can if the entity is found in the database, otherwise it returns Empty. With `find`, the entity is loaded immediately from the database and can be used in both attached and detached states. The next method you can use is the `getReference` method:

```
val myBook = Model.getReference(classOf[Book], someId)
```

This is very similar to the `find` method with a few key differences. First, the object that is returned is a lazy proxy for the entity. That means that no database load is required to occur when you execute the method, although providers may do at least a check on the existence of the ID. Because this is a lazy proxy, you usually don't want to use the returned object in a detached state unless you've accessed its fields while the session was open. The normal use of getReference is when you want to set up a relationship between two (or more) entities, since you don't need to query all of the fields just to set a foreign key. For example:

```
myBook.author = Model.getReference(classOf[Author], authorId)
```

When myBook is flushed to the database the EM will correctly set up the relationship. The final difference is in how unknown entities are handled. Recall that the `find` method returns Empty if the entity cannot be found; with `getReference`, however, we don't query the database until the reference is used. Because of this, the javax.persistence.EntityNotFoundException is thrown when you try to access an undefined entity for the first time (this also marks the transaction for rollback). The third method for loading an entity would be to use a query (named or otherwise) to fetch the entity. As an example, here's a query equivalent of the `find` method:

```
val myBook =
  Model.createQuery[Book](“from Book bk where bk.id = :id”)
       .setParams(“id” -> someId).findOne
```

The advantage here is that we have more control over what is selected by using the query language to specify other properties. One caveat is that when you use the findOne method you need to ensure that the query will actually result in a unique entity; otherwise, the EM will throw a `NonUniqueResultException`.

12.4.3 Loading many entities

Corresponding to the findOne method is the findAll method, which returns all entities based on a query. There are two ways to use findAll; the first is to use the convenience findAll method defined in the ScalaEntityManager class:

```
val myBooks = Model.findAll(“booksByYear”, “year” -> myYear)
```

This requires the use of a named query for the first arg, and subsequent args are of the form ("paramName" -> value). Named queries can be defined in your orm.xml, as shown in section 12.1.2 on page 78. Named queries are highly recommended over ad-hoc queries since they allow you to keep the queries in one location instead of being scattered all over your code. Named queries can also be pre-compiled by the JPA provider, which will catch errors at startup (or in your unit tests, hint hint) instead of when the query is run inside your code.

The second method is to create a ScalaQuery instance directly and then set parameters and execute it. In reality this is exactly what the Model.findAll method is doing. The advantage here is that with the ScalaQuery instance you can do things like set hinting, paging, etc. For instance, if you wanted to do paging on the books query, you could do

```
val myBooks = Model.createNamedQuery("booksByYear")
                   .setParams("year" -> myYear)
                   .setMaxResults(20)
                   .setFirstResult(pageOffset).findAll
```

12.4.4 Tips for using queries

In general we recommend that you use named queries throughout your code. In our experience, the extra effort involved in adding a named query is more than offset by the time it saves you if you ever need to modify the query. Additionally, we recommend that you use named parameters in your queries. Named parameters are just that: parameters that are inserted into your query by name, in contrast to positional parameters. As an example, here is the same query using named and positional parameters:

Named parameters `select user from User where (user.name like :searchString or user.email like :searchString) and user.widgets > :widgetCount`

Positional parameters `select user from User where (user.name like ?  or user.email like ?)  and user.widgets > :widgetCount`

From this example we can show several advantages of named params over positional params

1. You can reuse the same parameter within the same query and you only set it once. In the example about we would set the same parameter twice using positional params

2. The parameters can have meaningful names.

3. With positional params you may have to edit your code if you need to alter your query to add or remove parameters

In any case, you should generally use the parameterized query types as opposed to hand constructing your queries; using things like string concatenation opens up your site to SQL injection attacks unless you're very careful. For more information on queries there's an excellent reference for the EJBQL on the Hibernate website at http://www.hibernate.org/hib_docs/entitymanager/reference/en/html/queryhql.html.

12.4.5 Converting collection properties

The ScalaEntityManager and ScalaQuery methods are already defined so that they return Scala-friendly collections such as `scala.collection.jcl.BufferWrapper` or SetWrapper. We have to use Java Collections[12] "under the hood" and then wrap them because JPA doesn't understand Scala collections. For the same reason, collections in your entity classes must also use the Java Collections classes. Fortunately, Scala has a very nice framework for wrapping Java collections. In particular, the scala.collection.jcl.Conversions class contains a number of implicit conversions; all you have to do is import them at the top of your source file like so:

```
import scala.collection.jcl.Conversions._
```

Once you've done that the methods are automatically in scope and you can use collections in your entities as if they were real Scala collections. For example, we may want to see if our Author has written any mysteries:

```
val suspenseful = author.books.exists(_.genre = Genre.Mystery)
```

12.4.6 The importance of flush()

It's important to understand that in JPA the provider isn't required to write to the database until the session closes or is flushed. That means that constraint violations aren't necessarily checked at the time that you persist, merge or remove and object. Using the flush method forces the provider to write any pending changes to the database and immediately throw any exceptions resulting from any violations. As a convenience, we've written methods to do persist, merge and remove with a subsequent flush, as shown in listing H.8 on page 128. We do, however, recommend that if you will be doing multiple operations in one session cycle that you use a single flush at the end:

```
val container = Model.find(classOf[Container], containerId)
Model.remove(container.widget)
container.widget = new Widget("Foo!")
// next line only required if container.widget doesn't cascade PERSIST
Model.persist(container.widget)
Model.flush()
```

[12] http://java.sun.com/docs/books/tutorial/collections/index.html

Chapter 13

Using Scala Actors

- Intro to Concurrency
- Intro to Actors as a potential solution
- Examples of Actor usage within Lift base
- Examples of Actor usage on top of lift
    - Simple: Mailer
    - Complex: Pulling RSS feeds?

Chapter 14

OpenID Integration

- OpenID
  - Rolled into lift-proper a while ago
  - Overview of OpenID, no politics :)
  - Request cycle of an OpenID auth
  - How to integrate into Lift
  - OpenIDProtoUser
- OAuth
  - What?
  - Why?
  - How?

14.1 What is OpenID?

The OpenID Foundation[1] as:

> "OpenID eliminates the need for multiple usernames across different websites, simplifying your online experience.
>
> You get to choose the OpenID Provider that best meets your needs and most importantly that you trust. At the same time, your OpenID can stay with you, no matter which Provider you move to. And best of all, the OpenID technology is not proprietary and is completely free.
>
> For businesses, this means a lower cost of password and account management, while drawing new web traffic. OpenID lowers user frustration by letting users have control of their login.

[1] http://openid.net/

For geeks, OpenID is an open, decentralized, free framework for user-centric digital identity. OpenID takes advantage of already existing internet technology (URI, HTTP, SSL, Diffie-Hellman) and realizes that people are already creating identities for themselves whether it be at their blog, photostream, profile page, etc. With OpenID you can easily transform one of these existing URIs into an account which can be used at sites which support OpenID logins.

OpenID is still in the adoption phase and is becoming more and more popular, as large organizations like AOL, Microsoft, Sun, Novell, etc. begin to accept and provide OpenIDs. Today it is estimated that there are over 160-million OpenID enabled URIs with nearly ten-thousand sites supporting OpenID logins."

Chapter 15

Lucene/Compass Integration

Search is a requirement for any app

Dave has some code in ESME for this, I'll ask if we can pinch it as a starting point.

Chapter 16

Tagging Support

Planning:
- What are tags?
- Why Tags?
- Go other the tag architecture
- Add Tagging to PocketChange

- Tyler has lots of code for this
- Use the Delicious style of tagmap
    - Table of ThingsWithTags - id | name
    - Table of Tags - id | name
    - Table of TagMap - id | thingId | tagId
- Show an implementation that does it. “AfterSave”

Chapter 17

Lift Widgets

Planning:

- Create and extend the Sparklines widget from Marius
    - General Strategy for developing widgets
    - Pushing the right <head> includes (location of the JS file)
    - Figure out how to get your data in there and what you can about controlling and wrap it the loving embrace of lift.

Chapter 18

Web Services

Planning:

- Explain REST - http://en.wikipedia.org/wiki/REST
- Build off of the URL rewriting comment
- api demo from the wiki
- Talk to SteveJ about using his demo from LiftOff: http://liftweb.net/index.php/HowTo_do_Web_Services
    - Add a dispatcher to sit on the right urls
    - Add a class that has the methods that correspond to your API/Service
    - That's about it
- SteveJ gave us the go-ahead to use his stuff. It will have to be updated to the latest.

Part III

Appendices

Appendix A

Message Handling

A.1 Planning:

- Introduce message concept
- Introduce message types
  - error
  - notice
  - warning
- Talk about how they get to the screen
  - There is a requestVar that handles a list of messages
  - LiftSession spills those to the screen

For example, if a user successfully saves, notify with S.notice(“Save was sucessful”) and then redirect/sit on the page/etc

```
def saveAuthor(a: Author): NodeSeq = {
  a.save
  S.notice(“Author “ + a.niceName + “ saved successfully.”)
  S.redirect(“/”)
}
```

A.2 builtin/snippet/Msgs.scala

Marius: Comet push to msgs: http://groups.google.com/group/liftweb/browse_thread/thread/cfab4a332bb7cf81/32c04a

You can send Errors, Warnings and Notices and out of the box you can use:

```
<lift:Msgs />
<lift:Msgs id=”msg_area” />
<lift:snippet type="error_report">
  <lift:error_msg>Error!  The details are:</lift:error_msg>
  <lift:error_class>errorBox</lift:error_class>
  <lift:warning_msg>Whoops, I had a problem:</lift:warning_msg>
  <lift:warning_class>warningBox</lift:warning_class>
  <lift:notice_msg>Note:</lift:notice_msg>
  <lift:notice_class>noticeBox</lift:notice_class>
</lift:snippet>
```

Talk to the Msgs area via:

```
S.error, S.warning and S.notice
```

You can style them using CSS, for example:

```
.error {
  color: red;
  font-size: bigger;
}
.warning {
  color: yellow;
  font-size: bigger;
}
.notice {
  color: green;
  font-size: bigger;
}
```

Handing exceptions and redirecting the user

Appendix B

Helper Methods

B.1 Planning

- Can/Empty/Full/Failure
- Good description with examples: http://groups.google.com/group/liftweb/browse_thread/thread/1a0808688d792
  - openOr examples
  - map examples
  - pass - function with side effects
  - run - function with default
  - Using and/or chaining Failure objects
  - OR
  - object passedAuthor extends RequestVar[Can[Author]](Empty)
  - def view (xhtml : NodeSeq) : NodeSeq = passedAuthor.map({ author => // do bind, etc here and return a NodeSeq
  - }) openOr Text("Invalid author")
- Time/date formatting
- String formatting and utilities
- Encryption and hashing
- List helpers
- Binding
- URL modification
- IO Helpers
- tryo wrappers

B.2 Writing

B.2.1 Can (or Scala's Option class on steroids)

The Can Type is a utility that is built on top of Scala's Option Type, so let's take a quick look at Option first.

The Option class allows for typesafe method of dealing with a situation where you may or may not have a result. Option has two values, either Some(value) where value is actually the value and None which is used to represent nothing.

A typical example for Option is outlined using Scala's Map type. Below you'll see a definition of a Map, a successful attempt to get the value of key "a" and an attempt to get the value of key "i".

Notice that when there was an existing key-value pair for "a" the value was returned as Some(A) and when we asked for the value of key "i" we received None.

```
scala> val cap = Map("a" -> "A", "b" -> "B")
cap: scala.collection.immutable.Map[java.lang.String,java.lang.String] = Map(a -> A, b -> B
scala> cap.get("a")
res1: Option[java.lang.String] = Some(A)
scala> cap.get("i")
res2: Option[java.lang.String] = None
```

Getting the value out of an Option is usually handled via map, for example:

```
def prettyPrint(foo: Option[String]): String = foo match {
  case Some(x) => x
  case None => ‘‘Nothing found.’’
}
```

Which would be used in conjunction with the previous code:

```
scala> prettyPrint(cap.get("a"))
res7: String = A
scala> prettyPrint(cap.get("i"))
res8: String = Nothing found.
```

Lift's Can extends Option with a few ideas, mainly the fact that you can add a message about why a Can is Empty. Empty corresponds to Option's None and Full to Option's Some. So you can pattern match against a Can like so:

```
a match {
  Full(author) => Text(‘‘I found the author ‘‘ + author.niceName)
  Empty => Text(‘‘No author by that name.’’)
  Failure(message, _, _) => Text(‘‘Nothing found due to ‘‘ + message) // message may be someth
}
def confirmDelete {
  (for (val id <- param("id");     // get the ID
        val user <- User.find(id)) // find the user
```

```
      yield {
        user.delete_!
        notice("User deleted")
        redirectTo("/simple/index.html")
      }) getOrElse {error("User not found"); redirectTo("/simple/index.html")}
    }
```

openOr example

```
lazy val UserBio = UserBio.find(By(UserBio.id, id)) openOr (new UserBio)
def view (xhtml: NodeSeq): NodeSeq = passedAuthor.map({ author => // do bind, etc here and return
```

B.2.2 Time/date formatting

Various time span and general convenience functions.

B.2.3 String formatting and utilities

Capitailize, spliting, parsing strings to numbers, cleaning non-standard chars.

B.2.4 Encryption and hashing

Blowfish en/decrypt, md5, sha256, hexEncode

B.2.5 List helpers

Rotation, enumsToList, join, or

B.2.6 Binding

Binding for templates

B.2.7 URL modification

encode, decode, add params, check appropriate file extensions.

B.2.8 IO Helpers

Reading files and Streams, executing files

B.2.9 tryo wrappers

Addition to try blocks, allows the developer to ignore certain exceptions.

B.2.10 Regular Expressions

Appendix C

Internationalization

- S.?(String)
- lift:loc tag
- Proper placement of language resource

Appendix D

Logging in Lift

- Configuring built-in log4j
- Using slf4j instead
- Query Logger
- Maybe some recommendations?

Appendix E

Sending Email

- Configuring the mailer lib
- Sending email
- System.properties

Appendix F

Development Tips

- .props for Dev and Prod
- JavaRebel
- Maven's java:cc mode

Appendix G

Deployment Notes and Tips

Ask the list what they are deploying with. Generally it shouldn't be an issue where they deploy since Lift is essentially self-contained. We can go into specifics for special cases, I suppose

- Jetty is required for continuations
- Basic deployment via Jetty
- Brief overview of Jetty
- Capabilities
- Configuration
- SSL
- How jetty is configured in maven for testing
- Changing the default port (8080)
- Tips and tricks
- Deployment in Tomcat
- Packaging options (pom dependency scope)
- Deployment in JBoss
- Deployment in GlassFish

G.1 Setting up JNDI in Jetty

Appendix H

Code Listings

To conserve space and preserve flow in the main text, we've placed full code listings in this appendix. Each listing is cross-referenced back to the section where it's used.

H.1 JPA Library Demo

The full library demo is available under the main Lift Git repository at `http://github.com/dpp/liftweb/tree/master`. To illustrate some points, we've included selected listings from the project.

H.1.1 Master pom.xml

Listing H.1: Master pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project>
  <modelVersion>4.0.0</modelVersion>

  <parent>
    <groupId>net.liftweb</groupId>
    <artifactId>lift-examples-parent</artifactId>
    <version>0.10-SNAPSHOT</version>
    <relativePath>../pom.xml</relativePath>
  </parent>

  <artifactId>JPADemo-Master</artifactId>
  <description>JPA tutorial master project</description>
  <name>JPA Demo Master</name>
  <inceptionYear>2007</inceptionYear>
  <packaging>pom</packaging>

  <modules>
    <module>JPADemo-web</module>
    <module>JPADemo-spa</module>
  </modules>

  <repositories>
    <!-- Add this so that we get the latest version of Hibernate -->
    <repository>
      <id>JBoss</id>
      <name>JBoss Maven Repo</name>
      <layout>default</layout>
      <url>http://repository.jboss.com/maven2/</url>
      <snapshots>
        <enabled>false</enabled>
      </snapshots>
    </repository>
  </repositories>

  <!-- Master properties (inherited by modules) -->
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.4</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>geronimo-spec</groupId>
      <artifactId>geronimo-spec-jta</artifactId>
      <version>1.0.1B-rc4</version>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>javax.persistence</groupId>
      <artifactId>persistence-api</artifactId>
      <version>1.0</version>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>org.apache.derby</groupId>
      <artifactId>derby</artifactId>
      <version>10.2.2.0</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-core</artifactId>
```

```
      <version>3.3.0.SP1</version>
      <exclusions>
        <exclusion>
          <groupId>javax.transaction</groupId>
          <artifactId>jta</artifactId>
        </exclusion>
      </exclusions>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-annotations</artifactId>
      <version>3.4.0.GA</version>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-log4j12</artifactId>
      <version>1.4.2</version>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>log4j</groupId>
      <artifactId>log4j</artifactId>
      <version>1.2.15</version>
      <scope>provided</scope>
      <exclusions>
        <exclusion>
          <groupId>com.sun.jmx</groupId>
          <artifactId>jmxri</artifactId>
        </exclusion>
        <exclusion>
          <groupId>com.sun.jdmk</groupId>
          <artifactId>jmxtools</artifactId>
        </exclusion>
      </exclusions>
    </dependency>
  </dependencies>

</project>
```

H.1.2 Author Entity

Listing H.2: Author.scala

```
package com.foo.jpaweb.model

import javax.persistence._

/**
  An author is someone who writes books.
*/
@Entity
class Author {
  @Id
  @GeneratedValue{val strategy = GenerationType.AUTO}
  var id : Long = _

  @Column{val unique = true, val nullable = false}
  var name : String = ""

  @OneToMany{val mappedBy = "author",
             val targetEntity = classOf[Book],
             val cascade = Array(CascadeType.REMOVE)}
  var books : java.util.Set[Book] = new java.util.HashSet[Book]()
}
```

H.1.3 orm.xml Mapping

Listing H.3: orm.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm
        http://java.sun.com/xml/ns/persistence/orm_1_0.xsd" version="1.0">

  <package>com.foo.jpaweb.model</package>

  <entity class="Book">
    <named-query name="findBooksByAuthor">
      <query><![CDATA[from Book b where b.author.id = :id order by b.title]]></query>
    </named-query>
    <named-query name="findBooksByDate">
      <query><![CDATA[from Book b where b.published between :startDate and :endDate]]></query>
    </named-query>
    <named-query name="findBooksByTitle">
      <query><![CDATA[from Book b where lower(b.title) like :title order by b. title ]]></query>
    </named-query>
    <named-query name="findAllBooks">
      <query><![CDATA[from Book b order by b.title]]></query>
    </named-query>
  </entity>

  <entity class="Author">
    <named-query name="findAllAuthors">
      <query><![CDATA[from Author a order by a.name]]></query>
    </named-query>
  </entity>

</entity-mappings>
```

H.1.4 Enumv Trait

Listing H.4: Enumv Trait

```
1  trait Enumv {
2
3    this: Enumeration =>
4
5    private var nameDescriptionMap = scala.collection.mutable.Map[String, String]()
6
7    /* store a name and description for forms */
8    def Value(name: String, desc: String) : Value = {
9      nameDescriptionMap += (name -> desc)
10     new Val(name)
11   }
12
13     /* get description if it exists else name */
14   def getDescriptionOrName(ev: this.Value) = {
15     try {
16       nameDescriptionMap(""+ev)
17     } catch {
18       case e: NoSuchElementException => ev.toString
19     }
20   }
21
22   /* get name description pair list for forms */
23   def getNameDescriptionList = this.elements.toList.map(v => (v.toString, getDescriptionOrName(v) ) ).toList
24
25   /* get the enum given a string */
26   def valueOf(str: String) = this.elements.toList. filter (_.toString == str) match {
27     case Nil => null
28     case x => x.head
29   }
30 }
```

H.1.5 EnumerationType

Listing H.5: EnumvType class

```
1  abstract class EnumvType(val et: Enumeration with Enumv) extends UserType {
2
3    val SQL_TYPES = Array({Types.VARCHAR})
4
5    override def sqlTypes() = SQL_TYPES
6
7    override def returnedClass = classOf[et.Value]
8
9    override def equals(x: Object, y: Object): Boolean = {
10     return x == y
11   }
12
13   override def hashCode(x: Object) = x.hashCode
14
15   override def nullSafeGet(resultSet: ResultSet, names: Array[String], owner: Object): Object = {
16     val value = resultSet.getString(names(0))
17     if (resultSet.wasNull()) return null
18     else {
19       return et.valueOf(value)
20     }
21   }
```

H.1.6 Constructing an EM Directly

Listing H.6: Creating the EM directly

```
1  object Model extends ScalaEntityManager("jpaweb") {
2    lazy val factory = Persistence.createEntityManagerFactory(persistenceName)
3    def tx = getEM.getTransaction()
4
5    override def openEM() = {
6      val em = factory.createEntityManager()
7      em.getTransaction().begin()
8      em
9    }
10
11   override def closeEM(em : EntityManager) = {
12     val tx = em.getTransaction()
13     if (tx.isActive() && !tx.getRollbackOnly()) {
14       tx.commit()
15     } else if (tx.getRollbackOnly()) {
16       tx.rollback()
17     }
18     em.close()
19   }
```

H.1.7 Obtaining an EM Via JNDI

Listing H.7: Obtaining the EM via JNDI

```
1  object Model extends ScalaEntityManager("jpaweb") {
2    lazy val ctxt = new InitialContext()
3    def tx = ctxt.lookup("java:comp/UserTransaction").asInstanceOf[UserTransaction]
4
5    override def openEM () = {
6      tx.begin()
7      ctxt.lookup("java:comp/env/persistence/" + persistanceName).asInstanceOf[EntityManager]
8    }
9
10   override def closeEM (em : EntityManager) = {
11     em.close()
12
13     /*
14      * We only want to commit if we haven't already
15      * thrown an exception (due to a constraint violation , etc)
16      */
17     try {
18       if (tx.getStatus() == Status.STATUS_MARKED_ROLLBACK) {
19         tx.rollback()
20       } else if (tx.getStatus() == Status.STATUS_ACTIVE) {
21         tx.commit()
22         Log.debug("Committed TX")
23       } else {
24         Log.debug("TX status = " + txStatus(tx.getStatus()))
25       }
26     }
27   }
```

H.1.8 ScalaEntityManager Trait

Listing H.8: ScalaEntityManager

```
abstract class ScalaEntityManager(val persistanceName: String) {
  // The concrete impl should provide these methods
  protected def openEM () : EntityManager
  protected def closeEM (em : EntityManager)

  private object emVar extends RequestVar(openEM()) {
    override def cleanupFunc : Can[() => Unit]= Full (() => closeEM(this.is))
  }

  // dont encourage use of the entity manager directly
  def getEM = em
  private def em = emVar.is

  // value added methods
  def findAll[A](queryName : String, params : Pair[String,Any]*) = createAndParamify[A](queryName, params).findAll
  def createNamedQuery[A](queryName : String, params : Pair[String,Any]*) : ScalaQuery[A] = createAndParamify[A](queryName,
      params)

  // Worker for the previous two methods
  private def createAndParamify[A](queryName : String, params : Seq[Pair[String,Any]]) : ScalaQuery[A] = {
    val q = createNamedQuery[A](queryName)
    params.foreach(param => q.setParameter(param._1, param._2))
    q
  }

  // Common enough to combine into one op
  def persistAndFlush(entity : AnyRef) = { em.persist(entity ); em.flush() }
  def mergeAndFlush[T](entity : T) : T = {val e = merge(entity); flush () ; e}
  def removeAndFlush(entity : AnyRef) = { em.remove(entity); em.flush() }

  // methods defined on Entity Manager
  def persist ( entity : AnyRef) = em.persist(entity)
  def merge[T](entity: T): T = em.merge(entity)
  def remove(entity: AnyRef) = em.remove(entity);
  def find[A](clazz: Class[A], id: Any) = JPA.findToCan(em.find[A](clazz, id).asInstanceOf[A])
  def flush () = em.flush()
  def setFlushMode(flushModeType: FlushModeType) = em.setFlushMode(flushModeType)
  def refresh ( entity : AnyRef) = em.refresh(entity)
  def getFlushMode() = em.getFlushMode()
  def createQuery[A](queryString: String) = new ScalaQuery[A](em.createQuery(queryString))
  def createNamedQuery[A](queryName: String) = new ScalaQuery[A](em.createNamedQuery(queryName))
  def createNativeQuery[A](sqlString: String) = new ScalaQuery[A](em.createNativeQuery(sqlString))
  def createNativeQuery[A](sqlString: String, clazz: Class[A]) = new ScalaQuery[A](em.createNativeQuery(sqlString, clazz))
  def createNativeQuery[A](sqlString: String, resultSetMapping: String) = new ScalaQuery[A](em.createNativeQuery(sqlString,
      resultSetMapping))
  def close () = em.close()
  def isOpen() = em.isOpen()
  def getTransaction() = em.getTransaction()
  def joinTransaction () = em.joinTransaction()
  def clear () = em.clear()
  def getDelegate() = em.getDelegate()
  def getReference[A](clazz: Class[A], primaryKey: Any) = em.getReference[A](clazz, primaryKey)
  def lock( entity : AnyRef, lockMode: LockModeType) = em.lock(entity, lockMode)
  def contains(entity : AnyRef) = em.contains(entity)
}
```

H.1.9 ScalaQuery Trait

Listing H.9: ScalaQuery

```
1  class ScalaQuery[A](val query: Query) {
2    // value added methods
3    def findAll = getResultList()
4    def findOne = JPA.findToCan[A](query.getSingleResult.asInstanceOf[A])
5    def setParams(params : Pair[String,Any]*) = {params.foreach(param => query.setParameter(param._1, param._2)); this}
6
7    // methods defined on Query
8    def getResultList() = Conversions.convertList[A](query.getResultList.asInstanceOf[java.util.List[A]])
9    def getSingleResult() = query.getSingleResult.asInstanceOf[A]
10   def executeUpdate() = query.executeUpdate()
11   def setMaxResults(maxResult: Int) = {query.setMaxResults(maxResult);this}
12   def setFirstResult(startPosition: Int) = {query.setFirstResult(startPosition); this}
13   def setHint(hintName: String, value: Any) = {query.setHint(hintName, value); this}
14   def setParameter(name: String, value: Any) = {query.setParameter(name, value); this}
15   def setParameter(position: Int, value: Any) = {query.setParameter(position, value); this}
16   def setParameter(name: String, value: Date, temporalType: TemporalType) = {query.setParameter(name, value, temporalType); this}
17   def setParameter(position: Int, value: Date, temporalType: TemporalType) = {query.setParameter(position, value, temporalType); this}
18   def setParameter(name: String, value: Calendar, temporalType: TemporalType) = {query.setParameter(name, value, temporalType); this}
19   def setParameter(position: Int, value: Calendar, temporalType: TemporalType) = {query.setParameter(position, value, temporalType);
         this}
20   def setFlushMode(flushMode: FlushModeType) = {query.setFlushMode(flushMode); this}
21 }
```

H.1.10 JPA web.xml

This shows the LiftFilter setup as well as the persistence-context-ref.

Listing H.10: JPA web.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/j2ee/dtds/web-app_2_3.dtd">

<web-app>
<filter>
  <filter-name>LiftFilter</filter-name>
  <display-name>Lift Filter</display-name>
  <description>The Filter that intercepts lift calls</description>
  <filter-class>net.liftweb.http.LiftFilter</filter-class>
  <persistence-context-ref>
    <description>
      Persistence context for the library app
    </description>
    <persistence-context-ref-name>
      persistence/jpaweb
    </persistence-context-ref-name>
    <persistence-unit-name>
      jpaweb
    </persistence-unit-name>
  </persistence-context-ref>
</filter>


<filter-mapping>
  <filter-name>LiftFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>

</web-app>
```

Appendix I

Lift API

I.1 LiftRules

LiftRules object represents the backbone of Lift's flaxibility in terms of modeling the behavior to meet your application needs. It contains a handfull of variables and functions that allows you to hook in your own functions. LiftRules is typically used by the Lift application from Boot such as the application is properly setup at startup.

I.1.1 Declared types

Listing I.1: LiftRules types

```
  /**
* A partial function that will be called by lift when processing requests.
*/
  type DispatchPF = PartialFunction[Req, () => Can[LiftResponse]];

  /**
* A partial function that will be called very early by lift so that
   * you can transform an URI path into something else.
*/
  type RewritePF = PartialFunction[RewriteRequest, RewriteResponse]

  /**
* A partial function that allows to specify a dynamic snippet
*/
  type SnippetPF = PartialFunction[List[String], NodeSeq => NodeSeq]

  /**
* A partial function that allows to specify application wide custom lift tags.
   * They can pe used in markup like <lift:xxx/>
*/
```

```
    type LiftTagPF = PartialFunction[(String, Elem, MetaData, NodeSeq, String), NodeSeq]

    /**
  * A partial function that allows you to specify your own function when lift can not
    * find a certain path.
  */
    type URINotFoundPF = PartialFunction[(Req, Can[Failure]), LiftResponse]

    /**
  * A partial function that allows you decorate the URL's used for links, form action,
    * Comet and Ajax requests. This is useful when you want to pass certain parameters for
  * instance for load balancing rules. There are practices where sticky sessions needs to
    * be used and Load balancer need to dispatch the requests of a certain session to the same
    * cluster node. Not always jsessionid cookie or url rewriting. Thus load balancers support
    * to do balancing based on certain query parameters (i.e. srvid=1) which internally are mapped
    * with the actual node IP address.
  */
    type URLDecorator = PartialFunction[String, String]

    /**
  * A partial function that allows you to map a DispatchSnippet with a name. DispatchSnippet
    * is a trait that allows you to dispatch to various snippet functions without the use
    * of reflection.
  */
    type SnippetDispatchPF = PartialFunction[String, DispatchSnippet]

    /**
  * A partial function that allows you to map a DispatchSnippet with a name
  */
    type ViewDispatchPF = PartialFunction[List[String], Either[() => Can[NodeSeq], LiftView]]

 /**
  * A partial function that allows the application to define requests that should be
  * handled by lift rather than the default servlet handler
  */
 type LiftRequestPF = PartialFunction[Req, Boolean]
```

I.1.2 LiftRules variables

Listing I.2: LiftRules variables

```
/**
 * A function that takes the HTTPSession and the contextPath as parameters
 * and returns a LiftSession reference. This can be used in cases subclassing
 * LiftSession is necessary.
 */
var sessionCreator: (HttpSession, String, List[(String, String)]) => LiftSession

/**
```

```
 * The path to handle served resources. By default it is set to classpath
 */
var ResourceServerPath

/**
 * Allows you to switch between JavaScript framework that Lift is using. By default it
 * points to JQueryArtifacts. Lift also support YUIArtifacts
 */
var jsArtifacts: JSArtifacts

/**
 * Use this PartialFunction to to automatically add static URL parameters
 * to any URL reference from the markup of Ajax request.
 */
var urlDecorate: List[URLDecorator]

/**
 * Calculate the Comet Server (by default, the server that
 * the request was made on, but can do the multi-server thing
 * as well)
 */
var cometServer: () => String

/**
 * The maximum concurrent requests. If this number of
 * requests are being serviced for a given session, messages
 * will be sent to all Comet requests to terminate
 */
var maxConcurrentRequests = 2

/**
 * A partial function that determines content type based on an incoming
 * Req and Accept header
 */
var determineContentType: PartialFunction[(Can[Req], Can[String]), String]

/**
 * Contains functions that willbe executed when LiftServlet.destroy is called.
 */
val unloadHooks: ListBuffer[() => Unit]

/**
 * The maximum allowed size of a complete mime multi-part POST. Default
 * 8MB
 */
var maxMimeSize: Long

/**
 * Should pages that are not found be passed along the servlet chain to the
 * next handler?
```

```
 */
var passNotFoundToChain = false

/**
 * The maximum allowed size of a single file in a mime multi-part POST.
 * Default 7MB
 */
var maxMimeFileSize: Long

/**
 * The function referenced here is called if there's a localization lookup failure
 */
var localizationLookupFailureNotice: Can[(String, Locale) => Unit]

/**
 * The default location to send people if SiteMap access control fails
 */
var siteMapFailRedirectLocation: List[String]

/**
 * If you don't want lift to send the application/xhtml+xml mime type to those browsers
 * that understand it, then set this to {@code false}
 */
var useXhtmlMimeType: Boolean = true

/**
 * A function that defines how a String should be converted to XML
 * for the localization stuff. By default, Text(s) is returned,
 * but you can change this to attempt to parse the XML in the String and
 * return the NodeSeq.
 */
var localizeStringToXml: String => NodeSeq

/**
 * The base name of the resource bundle. Default set to lift
 */
var resourceName = "lift"

/**
 * The base name of the resource bundle of the lift core code
 * Default set to i18n.lift-core
 */
var liftCoreResourceName

/**
 * Where to send the user if there's no comet session. Default set to /
 */
var noCometSessionPage

/**
```

```
 * Put a function that will calculate the request timeout based on the
 * incoming request.
 */
var calcRequestTimeout: Can[Req => Int]

/**
 * If you want the standard (non-AJAX) request timeout to be something other than
 * 10 seconds, put the value here
 */
var stdRequestTimeout: Can[Int]

/**
 * If you want the AJAX request timeout to be something other than 120 seconds, put the value here
 */
var cometRequestTimeout: Can[Int]

/**
 * Meta information for the notices that are applied via Ajax response
 */
var ajaxNoticeMeta: Can[AjaxMessageMeta]

/**
 * Meta information for the warnings that are applied via Ajax response
 */
var ajaxWarningMeta: Can[AjaxMessageMeta]

/**
 * Meta information for the errors that are applied via Ajax response
 */
var ajaxErrorMeta: Can[AjaxMessageMeta]

/**
 * A function that takes the current HTTP request and returns the current
 */
var timeZoneCalculator: Can[HttpServletRequest] => TimeZone

/**
 * How many times do we retry an Ajax command before calling it a failure?
 */
var ajaxRetryCount: Can[Int]

/**
 * The JavaScript to execute at the begining of an
 * Ajax request (for example, showing the spinning working thingy)
 */
var ajaxStart: Can[() => JsCmd]

/**
 * The JavaScript to execute at the end of an
 * Ajax request (for example, removing the spinning working thingy)
```

```
 */
var ajaxEnd: Can[() => JsCmd]

/**
 * The default action to take when the JavaScript action fails
 */
var ajaxDefaultFailure: Can[() => JsCmd]

/**
 * A function that takes the current HTTP request and returns the current
 */
var localeCalculator: Can[HttpServletRequest] => Locale

/**
 * The Ajax path
 */
var ajaxPath = "ajax_request"

/**
 * The Comet path
 */
var cometPath = "comet_request"

/**
 * Determins the JsExp from the path. By default Lift also adds some unique sequences.
 */
var calcCometPath: String => JsExp

/**
 * If there is an alternative way of calculating the context path
 * (by default inspecting the X-Lift-ContextPath header)
 */
var calculateContextPath: HttpServletRequest => Can[String]

/**
 * The function that returns a LiftLogger
 */
var cometLoggerBuilder: () => LiftLogger

/**
 * The partial function determining the default HTTP headers to be sent to client
 */
var defaultHeaders: PartialFunction[(NodeSeq, Req), List[(String, String)]]

/**
 * A lift of functions that allows tranforming the responses
 */
var responseTransformers: List[LiftResponse => LiftResponse]

/**
```

```
 * convertResponse is a PartialFunction that reduces a given Tuple4 into a
 * LiftResponse that can then be sent to the browser. The tuple is formed by the xhtml reaponse,
 * HTTP headers list, HTTP cookies and the request object.
 */
var convertResponse: PartialFunction[(Any, List[(String, String)], List[Cookie], Req), LiftResponse]

/**
 * The functions to be executed when a snippet is not found
 */
var snippetFailedFunc: List[SnippetFailure => Unit]

/**
 * The function that deals with how exceptions are presented to the user during processing
 * of an HTTP request. Put a new function here to change the behavior.
 *
 * The function takes the Req and the Exception and returns a LiftResponse that's
 * sent to the browser.
 */
var logAndReturnExceptionToBrowser: (Req, Throwable) => LiftResponse

/**
 * The partial function (pattern matching) for handling converting an exception to something to
 * be sent to the browser depending on the current RunMode (development, etc.)
 *
 * The best thing to do is browserResponseToException = { case (...) => } orElse browserResponseToException
 * so that your response over-rides the default, but the processing falls through to the default.
 */
var browserResponseToException: PartialFunction[(Props.RunModes.Value, Req, Throwable), LiftResponse]

/**
 * The list of partial function for defining the behavior of what happens when
 * URI is invalid and you're not using a site map
 */
def uriNotFound: List[URINotFoundPF]

/**
 * Hooks to be called when Lift start processing a request
 */
var onBeginServicing: List[Req => Unit]

/**
 * Hooks to be called when Lift ends processing a request
 */
var onEndServicing: List[(Req, Can[LiftResponse]) => Unit]

/**
 * By default it is set to true
 */
var autoIncludeComet: LiftSession => Boolean
```

```
/**
 * By default it is set to true
 */
var autoIncludeAjax: LiftSession => Boolean

/**
 * The deault Ajax script.
 */
var renderAjaxScript: LiftSession => JsCmd

/**
 * The deault Comet script.
 */
var renderCometScript: LiftSession => JsCmd

/**
 * Renders commet versioning
 */
var renderCometPageContents: (LiftSession, Seq[CometVersionPair]) => JsCmd

/**
 * If this time does not elapses Lift will send HTTP 304 status to client
 * indicating that the cached content should be used.
 */
var ajaxScriptUpdateTime: LiftSession => Long

/**
 * If this time does not elapses Lift will send HTTP 304 status to client
 * indicating that the cached content should be used.
 */
var cometScriptUpdateTime: LiftSession => Long

/**
 * The name of the Ajax script
 */
var ajaxScriptName: () => String

/**
 * The name of the Comet script
 */
var cometScriptName: () => String

/**
 * Return the Comet JavaScript code
 */
var serveCometScript: (LiftSession, Req) => Can[LiftResponse]

/**
 * Return the Ajax JavaScript code
 */
```

```
var serveAjaxScript: (LiftSession, Req) => Can[LiftResponse]
```

I.1.3 LiftRules useful functions

Listing I.3: LiftRules functions

```
/**
 * Add functions to be called before sending response to client
 */
def appendBeforeSend(f: (BasicResponse, HttpServletResponse, List[(String, String)], Can[Req]) => Any)

/**
 * Add functions to be called before sending response to client
 */
def appendAfterSend(f: (BasicResponse, HttpServletResponse, List[(String, String)], Can[Req]) => Any)

/**
 * Add unload hook functions
 */
def addUnloadHook(f: () => Unit)

/**
 * Append a named partial function defining application-wide
 * &lt;lift:xxx/&gt; tags
 */
def appendLiftTagProcessing(in: LiftTagPF)

/**
 * Prepend a named partial function defining application-wide
 * &lt;lift:xxx/&gt; tags
 */
def prependLiftTagProcessing(in: LiftTagPF)

/**
 * Append a partial function to look up snippets to
 * the rules
 */
def appendSnippetDispatch(in: SnippetDispatchPF)

/**
 * Prepend a partial function to look up snippets to
 * the rules
 */
def prependSnippetDispatch(in: SnippetDispatchPF)

/**
 * Prepend a partial function to the list of partial functions
 * the define views
 */
```

```
def prependViewDispatch(in: ViewDispatchPF)

/**
 * Append a partial function to the list of partial functions
 * that define views
 */
def appendViewDispatch(in: ViewDispatchPF)

/**
 * Append function to be executed very early when receiving the HTTP request
 */
def appendEarly(f: HttpServletRequest => Any)

/**
 * Prepend a request handler to the stateless request handler
 */
def prependStatelessDispatch(in: DispatchPF)

/**
 * Postpend a request handler to the stateless request handler
 */
def appendStatelessDispatch(in: DispatchPF)

/**
 * Set the ServletContext in LiftRules
 */
def setContext(in: ServletContext): Unit

/**
 * Add package name for components lookup such as snippets, LiftView-s etc
 */
def addToPackages(what: String)

/**
 * Obtains the resource using the ServletContext
 */
def getResource(name: String): Can[_root_.java.net.URL]

/**
 * Obtains the resource using the ServletContext as an InputStream
 */
def getResourceAsStream(name: String): Can[_root_.java.io.InputStream]

/**
 * Obtains the resource using the ServletContext as an Array[Byte]
 */
def loadResource(name: String): Can[Array[Byte]]

/**
 * Loads a resource as a NodeSeq
```

```
 */
def loadResourceAsXml(name: String): Can[NodeSeq]

/**
 * Loads a resource as a String
 */
def loadResourceAsString(name: String): Can[String]

/**
 * Append a partial function to the list of interceptors to test
 * if the request should be handled by lift
 */
def appendLiftRequest(what: LiftRequestPF)

/**
 * Adds a SnipetPF at the begining of the functions list
 */
def prependSnippet(pf: SnippetPF)

/**
 * Adds a SnipetPF at the end of the functions list
 */
def appendSnippet(pf: SnippetPF)

/**
 * Adds a RewritePF at the begining of the functions list
 */
def prependRewrite(pf: RewritePF)

/**
 * Adds a RewritePF at the end of the functions list
 */
def appendRewrite(pf: RewritePF)

/**
 * Adds a DispatchPF at the begining of the functions list
 */
def prependDispatch(pf: DispatchPF)

/**
 * Adds a DispatchPF at the end of the functions list
 */
def appendDispatch(pf: DispatchPF)

/**
 * Prepend the URINotFound handler to the existing list.
 * Because the default Lift URI Not Found handler handles
 * The default case, you need only handle special cases.
 */
def prependUriNotFound(in: URINotFoundPF)
```

```
/**
 * Modifies the root relative paths from the css url-s
 *
 * @param path - the path of the css resource
 * @prefix - the prefix to be added on the root relative paths. If this is Empty
 *            the prefix will be the application context path.
 */
def fixCSS(path: List[String], prefix: Can[String])
```

I.2 S

I.3 SHtml

I.4 Foo

Index