Lift Cookbook

Richard Dallaway

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Lift Cookbook

by Richard Dallaway

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Meghan Blanchette
Production Editor: Kara Ebrahim
Copyeditor:
Proofreader: FIX ME!

Indexer:
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Robert Romano

July 2013: First Edition

Revision History for the First Edition:

YYYY-MM-DD: First release

See *http://oreilly.com/catalog/errata.csp?isbn=9781449362683* for release details.

ISBN: 978-1-449-36268-3

[?]

Table of Contents

Preface

This is a collection of solutions to questions you might have while developing web applications with the Lift Web Framework.

The aim is to give a single, short answer to a specific question. When there's more than approach, or style, we'll give you one solution, but will point you at alternatives in the discussion.

Chapter 1 will get you up and running with Lift, but in other respects this cookbook is aimed at practitioners and the questions they have asked. If this is the first time you've heard of Lift, you'll want to look at:

- *Simply Lift*, online at *http://simply.liftweb.net*.
- Torsten Uhlmann's *Instant Lift Web Applications How-to*, PACKT Publishing, 2013.
- Timothy Perrett's *Lift in Action*, Manning Publications, 2011.

Contributors

I've mined the Lift mailing list for these recipes, but I'm not the only one. Recipes have been contributed by:

- Jono Ferguson
- Franz Bettag — Franz is an enthusiastic Scala Hacker for several years now. He joined the Lift team in January 2012 and actively tweets and blogs about his newest Scala adventures. Find him at *https://twitter.com/fbettag*
- Marek Żebrowski
- Peter Robinett — Peter is a web and mobile developer and a Lift committer. He can be found on the web at *http://www.bubblefoundry.com* and on Twitter at *http://twitter.com/pr1001*.

- Kevin Lau — Kevin is a founder of few web apps with a focus in AWS cloud, iOS and Lift.
- Tony Kay

You should join them: Recipe 11.5 tells you how.

Source

The text of this cookbook is at: *https://github.com/d6y/lift-cookbook*.

You'll find projects for each chapter at GitHub: *https://github.com/LiftCookbook/*.

Updates

Follow @LiftCookbook on Twitter.

Software Versions

Except where otherwise indicated, the examples use Lift 2.5 with SBT 0.12 and Scala 2.9.

Lift 2.5 is also available for Scala 2.10.

Conventions Used in This Book

The following typographical conventions are used in this book:

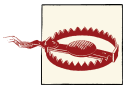
Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This icon signifies a tip, suggestion, or general note.



This icon indicates a warning or caution.

Using Code Examples

This book is here to help you get your job done. In general, if this book includes code examples, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Lift Cookbook* by Richard Dallaway (O'Reilly). Copyright 2013 Richard Dallaway, 978-1-4493-6268-3."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

Safari Books Online is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of product mixes and pricing programs for organizations, government agencies, and individuals. Subscribers have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technol-

ogy, and dozens more. For more information about Safari Books Online, please visit us online.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://www.oreilly.com/catalog/0636920029151*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

These recipes existing because of the many contributions on the Lift mailing list, where *Liftafarians*, as they are known, generously give their time to ask questions, put together example projects, give answers, share alternatives and chip in with comments. Thank you.

I also want to thank those who have provided corrections to the recipes, on the mailing list, Github, and Twitter. No matter what else it means, writing on the web and publishing on the web has meant I've received some prompt high-quality feedback.

I am indebted to the contributors who have taken the trouble to write new recipes for this book.

CHAPTER 1

Installing and Running Lift

This chapter covers questions regarding starting development with Lift: running a first Lift application and setting up a coding environment. You'll find answers regarding production deployment in Chapter 10.

1.1. Downloading and Running Lift

Problem

You want to install and run Lift on your computer.

Solution

The only prerequisite for installing and running Lift is to have Java 1.5 or later installed. Instructions for installing Java can be found at *http://java.com/*.

You can find out if you have Java from the shell or command prompt by asking for the version you have installed:

```
$ java -version
java version "1.7.0_13"
Java(TM) SE Runtime Environment (build 1.7.0_13-b20)
Java HotSpot(TM) 64-Bit Server VM (build 23.7-b01, mixed mode)
```

Once you have Java, the following instructions will download, build and start a basic Lift application.

For Mac and Linux

- Visit *http://liftweb.net/download* and download the Lift 2.5-RC5 ZIP file.
- Unzip the file.

- Start *Terminal* or your favourite shell tool.
- Navigate into the unzipped folder and into the *scala_29* sub-folder and then into the *lift_basic* folder.
- Run: `./sbt`.
- Required libraries will be downloaded automatically.
- At the SBT prompt (>) type: `container:start`.
- Open your browser and go to *http://127.0.0.1:8080/*.
- When you're done, type `exit` at the SBT prompt to stop your application from running.

For Windows

- Visit *http://liftweb.net/download* locate the link to the ZIP version of Lift 2.5-RC5 and save this to disk.
- Extract the contents of the ZIP file.
- Navigate in *Explorer* to the extracted folder, and inside navigate into *scala_29* and then *lift_basic*.
- Double click `sbt.bat` to run the build tool and a terminal window should open.
- Required libraries will be downloaded automatically.
- At the SBT prompt (>) type: `container:start`.
- You may find Windows Firewall blocking Java from running. If so, opt to "allow access".
- Start your browser and go to *http://127.0.0.1:8080/*.
- When you're done, type `exit` at the SBT prompt to stop your application from running.

Expected result

The result of the above commands should be a basic Lift application running on your computer as shown in Figure 1-1.

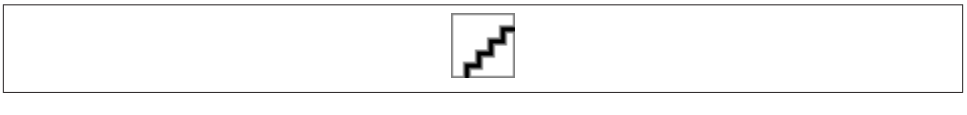


Figure 1-1. The basic Lift application home page.

Discussion

Lift isn't installed in the usual sense of "installing software". Instead you use standard build tools, such as SBT or Maven, to assemble your application with the Lift framework. In this recipe we downloaded a ZIP file containing four fairly minimal Lift applications, and then started one of them via the build tool.

Simple Build Tool

Typing `sbt` starts a *Simple Build Tool* used by Scala projects (it's not specific to Lift). SBT will check the project definition and download any libraries required, which will include the Lift framework.

This download happens once, and the downloaded files are stored on disk in *.ivy2* under your home folder.

Your application build is configured by *build.sbt*. Looking inside you'll see:

- basic information about your application, including a name and version;
- resolvers, which inform SBT where to fetch dependencies from;
- settings for plugins and the Scala compiler; and
- a list of dependencies required to run your application, which will include the Lift framework.

Running Your Application

The SBT command `container:start` starts the web server on the default port of 8080 and passes requests to your Lift application. The word *container* refers to the software you deploy your application into. There are a variety of containers (*Jetty* and *Tomcat* are probably the best known) all of which conform to a standard for deployment. The upshot is you can build your application and deploy to whichever one you prefer. The `container:start` command uses Jetty.

Source Code

The source code of the application resides in *src/main/webapp* and *src/main/scala*. If you take a look at *index.html* in the *webapp* folder you'll see mention of `lift:helloWorld`. That's a reference to the class defined in *scala/code/snippet/HelloWorld.scala*. This is a *snippet invocation* and an example of Lift's *view first* approach to web applications. That is, there's no routing set up for the index page to collect the data and forward it to the view. Instead, the view defines areas of the content that are replaced with functions, such as those functions defined in *HelloWorld.scala*.

Lift knows to look in the `code` package for snippets because that package is declared as a location for snippets in *scala/bootstrap/liftweb/Boot.scala*. The Boot class is run when starting your application, and it's where you can configure the behaviour of Lift.

See Also

The Simple Build Tool documentation is at *http://www.scala-sbt.org*.

Tutorials for Lift can be found in *Simply Lift* at *http://simply.liftweb.net/* and in *Lift in Action* (Tim Perrett, 2011, Manning Publications Co).

1.2. Creating a Lift Project from Scratch using SBT

Problem

You want want to create a Lift web project from scratch without using the ZIP files provided on the official Lift website.

Solution

You will need to configure SBT and the Lift project yourself. Luckily, only five small files are needed.

First, create an SBT plugin file at *project/plugins.sbt* (all file names are given relative to the project root directory):

```
libraryDependencies <+= sbtVersion(v => v match {
  case "0.11.0" => "com.github.siasia" %% "xsbt-web-plugin" % "0.11.0-0.2.8"
  case "0.11.1" => "com.github.siasia" %% "xsbt-web-plugin" % "0.11.1-0.2.10"
  case "0.11.2" => "com.github.siasia" %% "xsbt-web-plugin" % "0.11.2-0.2.11"
  case "0.11.3" => "com.github.siasia" %% "xsbt-web-plugin" % "0.11.3-0.2.11.1"
  case x if x startsWith "0.12" =>
    "com.github.siasia" %% "xsbt-web-plugin" % "0.12.0-0.2.11.1"
})
```

This file tells SBT that you will be using the xsbt-web-plugin and chooses the correct version based upon your version of SBT.

Next, create an SBT build file, *build.sbt*:

```
organization := "org.yourorganization"

name := "liftfromscratch"

version := "0.1-SNAPSHOT"

scalaVersion := "2.10.0"

seq(com.github.siasia.WebPlugin.webSettings :_*)
```

```
libraryDependencies ++= {
  val liftVersion = "2.5-RC5"
  Seq(
    "net.liftweb" %% "lift-webkit" % liftVersion % "compile",
      "org.eclipse.jetty"  %  "jetty-webapp"  %  "8.1.7.v20120910"    %  "contain
er,test",
    "org.eclipse.jetty.orbit" % "javax.servlet" % "3.0.0.v201112011016" %
      "container,compile" artifacts Artifact("javax.servlet", "jar", "jar")
  )
}
```

Feel free to change the various versions, though be aware that certain versions of Lift are only built for certain versions of Scala.

Now that you have the basics of an SBT project, you can launch the `sbt` console. It should load all the necessary dependencies, including the proper Scala version, and bring you to a prompt.

Next, create the following file at *src/main/webapp/WEB-INF/web.xml*:

```
<!DOCTYPE web-app SYSTEM "http://java.sun.com/dtd/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>
  </filter>
  <filter-mapping>
    <filter-name>LiftFilter</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>
</web-app>
```

The *web.xml* file tells web containers, such as Jetty as configured by xsbt-web-plugin, to pass all requests on to Lift.

Next, create a sample *index.html* file at *src/main/webapp/index.html* for our Lift app to load. For example:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Lift From Scratch</title>
  </head>
  <body>
    <h1>Welcome, you now have a working Lift installation</h1>
  </body>
</html>
```

Finally, setup the basic Lift boot settings by creating a *Boot.scala* file at *src/main/scala/bootstrap/Boot.scala*. The following contents will be sufficient:

```
package bootstrap.liftweb

import net.liftweb.http.{Html5Properties, LiftRules, Req}
import net.liftweb.sitemap.{Menu, SiteMap}

/**
 * A class that's instantiated early and run.  It allows the application
 * to modify lift's environment
 */
class Boot {
  def boot {
    // where to search snippet
    LiftRules.addToPackages("org.yourorganization.liftfromscratch")

    // Build SiteMap
    def sitemap(): SiteMap = SiteMap(
      Menu.i("Home") / "index"
    )

    // Use HTML5 for rendering
    LiftRules.htmlProperties.default.set((r: Req) =>
      new Html5Properties(r.userAgent))
  }
}
```

Congratulations, you now have a working Lift project!

You can verify that you have a working Lift project by launching the Jetty web container from the `sbt` console with the `container:start` command. First the *Boot.scala* file should be compiled and then you should be notified that Jetty has launched and is listening at *http://localhost:8080*. You should be able to go to the address in your web browser and see the rendered *index.html* file you created earlier.

Discussion

As shown above, creating a Lift project from scratch is a relatively simple process. However, it can be a tricky one for newcomers, especially if you are not used to the JVM ecosystem and its conventions for web containers. If you run into problems, make sure the files are in the correct locations and that their contents were not mistakenly modified. If all else fails, refer to the sample project below or ask for help on the Lift mailing list.

Lift projects using SBT or similar build tools follow a standard project layout, where Scala source code is in *src/main/scala* and web resources are in *src/main/webapp*. Your Scala files must be placed either directly at *src/main/scala* or in nested directories matching the organization and name you defined in *build.sbt*, in our case giving us *src/main/scala/org/yourorganization/liftfromscratch/*. Test files match the directory structure but are placed in *src/test/* instead of *src/main/*. Likewise, the *web.xml* file must be placed in *src/main/webapp/WEB-INF/* for it to be properly detected.

Given these conventions, you should have a directly structure looking quite, if not exactly, like this:

```
- project root directory
  | build.sbt
  - project/
    | plugins.sbt
  - src/
    - main/
      - scala/
        - bootstrap/
          | Boot.scala
        - org/
          - yourorganization/
            - liftfromscratch/
              | <your Scala code goes here>
      - webapp/
        | index.html
        | <any other web resources - images, HTML, JavaScript, etc - go here>
        - WEB-INF/
          | web.xml
    - test/
      - scala/
        - org/
          - yourorganization/
            - liftfromscratch/
              | <your tests go here>
```

See Also

There is a sample project created using this method at: *https://github.com/bubblefoundry/lift-from-scratch*.

1.3. Developing Using a Text Editor

Problem

You want to develop your Lift application using your favourite text editor, hitting reload in your browser to see changes.

Solution

Run SBT while you are editing, and ask it to detect and compile changes to Scala files. To do that, start `sbt` and enter the following to the SBT prompt:

```
~; container:start; container:reload /
```

When you save a source file in your editor, SBT will detect this change, compile the file, and reload the web container.

Discussion

An SBT command prefixed with ~ makes that command run when files change. The first semicolon introduces a sequence of commands, where if the first command succeeds, the second will run. The second semicolon means the `reload` command will run if the `start` command ran OK. The `start` command will recompile any Scala source files that have changed.

When you run SBT in this way, you'll notice the following output:

```
1. Waiting for source changes... (press enter to interrupt)
```

And indeed, if you do press enter in the SBT window you'll exit this *triggered execution* mode and SBT will no longer be looking for file changes. However, while SBT is watching for changes, the output will indicate when this happens with something that looks a little like this:

```
[info] Compiling 1 Scala source to target/scala-2.9.1/classes...
[success] Total time: 1 s, completed 15-Nov-2012 18:14:46
[pool-301-thread-4] DEBUG net.liftweb.http.LiftServlet - Destroyed Lift handler.
[info] stopped o.e.j.w.WebAppContext{/,[src/main/webapp/]}
[info] NO JSP Support for /, did not find org.apache.jasper.servlet.JspServlet
[info] started o.e.j.w.WebAppContext{/,[src/main/webapp/]}
[success] Total time: 0 s, completed 15-Nov-2012 18:14:46
2. Waiting for source changes... (press enter to interrupt)
```

Edits to HTML files don't trigger the SBT compile and reload commands. This is because SBT's default behaviour is to look for Scala and Java source file changes, and also changes to files in *src/main/resources/*. This works out just fine, because Jetty will use your modified HTML file when you reload the browser page.

Restarting the web container each time you edit a Scala file isn't ideal. You can reduce the need for restarts by integrating JRebel into your development environment, as described in Recipe 1.4.

However, if you are making a serious number of edits, you may prefer to issue a `container:stop` command until you're ready to run you application again with `container:start`. This will prevent SBT compiling and restarting your application over and over. The SBT console has a command history, and using the up and down keyboard arrows allows you to navigate to previous commands and run them by pressing the return key. That takes some of the tedium out of these long commands.

One error you may run into is:

```
java.lang.OutOfMemoryError: PermGen space
```

The *permanent generation* is a Java virtual machine concept. It's the area of memory used for storing classes (amongst other things). It's a fixed size and once it is full this PermGen error appears. As you might imagine, continually restarting a container causes

many classes to be loaded and unloaded, but the process is not perfect, effectively leaking memory. The best you can do is stop and then restart SBT. If you're seeing this error often, check the setting for `-XX:MaxPermSize` inside the *sbt* (or *sbt.bat*) script, and if you can, double it.

See Also

There's more about triggered execution at *http://www.scala-sbt.org/release/docs/Detailed-Topics/Triggered-Execution*.

Reference for the core SBT command line: *http://www.scala-sbt.org/release/docs/Detailed-Topics/Command-Line-Reference*.

Command reference for the web plugin for SBT is at: *https://github.com/JamesEarlDouglas/xsbt-web-plugin/wiki*.

1.4. Incorporating JRebel

Problem

You want to avoid application restarts when you change a Scala source file by using JRebel.

Solutions

There are three steps required: install JRebel once; each year request the free Scala license; and configure SBT to use JRebel.

First, visit the *https://my.jrebel.com/plans* and request the free Scala license.

Second, download the "Generic ZIP Archive" version of JRebel, unzip it to where you like. For this recipe I've chosen to use */opt/zt/jrebel/*.

When your have received your account confirmation email from JRebel, you can copy your "authentication token" from the "Active" area of ZeroTurnaround's site. To apply the token to your local install, run the JRebel configuration script:

```
$ /opt/zt/jrebel/bin/jrebel-config.sh
```

For Windows navigate to and launch *bin\jrebel-config.cmd*.

In the "Activation" setting select "I want to use myJRebel" and then in the "License" section paste in your activation token. Click the "Activate" button, and once you see the license status change to "You have a valid myJRebel token" click "Finish".

Finally, configure SBT by modifying the *sbt* script to enable JRebel. This means setting the `-javaagent` and `-noverify` flags for Java, and enabling the JRebel Lift plugin.

For Mac and Linux, the script that's included with the Lift downloads would become:

```
java -Drebel.lift_plugin=true -noverify -javaagent:/opt/zt/jrebel/jrebel.jar \
 -Xmx1024M -Xss2M -XX:MaxPermSize=512m -XX:+CMSClassUnloadingEnabled -jar \
 `dirname $0`/sbt-launch-0.12.jar "$@"
```

For Windows, modify *sbt.bat* to be:

```
set SCRIPT_DIR=%~dp0
java -Drebel.lift_plugin=true -noverify -javaagent:c:/opt/zt/jrebel/jrebel.jar \
 -XX:+CMSClassUnloadingEnabled -XX:MaxPermSize=256m -Xmx1024M -Xss2M \
 -jar "%SCRIPT_DIR%\sbt-launch-0.12.jar" %*
```

There's nothing else to do to use JRebel. When you start SBT you'll see a large banner starting something like this:

```
#############################################################

  JRebel 5.1.1 (201211271929)
  (c) Copyright ZeroTurnaround OU, Estonia, Tartu.

  Over the last 30 days JRebel prevented
  at least 335 redeploys/restarts saving you about 13.6 hours.
....
```

With JRebel installed, you can now `container:start` your application, modify and compile a Scala file and reload a page in your application. You'll see a notice that the class has been reloaded:

```
[2012-12-16 23:15:44] JRebel: Reloading class 'code.snippet.HelloWorld'.
```

That change is live, without having to restart the container.

Discussion

JRebel is very likely to speed up your development. It updates code in a running Java virtual machine, without having to stop and restart it. The effect is that, on the whole, you can compile a class, then hit reload in your browser to see the change in your Lift application.

Even with JRebel you will need to restart your applications from time to time, but JRebel usually reduces the number of restarts. For example, *Boot.scala* is run when your application starts, so if you modify something in your *Boot.scala* you'll need to start and start your application. JRebel can't help with that.

But there are also other situations that JRebel cannot help with, such as when a superclass changes. Generally, JRebel will emit a warning about this in the console window. If that happens, stop and start your application.

The `-Drebel.lift_plugin=true` setting adds Lift-specific functionality to JRebel. Specifically, it allows JRebel to reload changes to `LiftScreen`, `Wizard` and `RestHelper`. This means you can change fields or screens, and change REST `serve` code.

Purchased Licenses

This recipe uses a free Scala license for a service called myJRebel. This communicates with JRebel servers via the activation code. If you have purchased a license from Zero-Turnaround, the situation is slightly different. In this case, you will have a license key which you store in a file called *jrebel.lic*. You can place the file in a *.jrebel* folder in your home directory, or alongside *jrebel.jar* (e.g., in the */opt/zt/jrebel/* folder if that's where you installed JRebel), or you can specify some other location. For the latter option, modify the *sbt* script and specify the location of the file by adding another Java setting:

```
-Drebel.license=/path/to/jrebel.lic
```

See Also

You'll find details about how JRebel works in the FAQ at: *http://zeroturnaround.com/software/jrebel/resources/faq/*.

The Lift support was announced in a blog post in 2012 at *http://zeroturnaround.com/jrebel/lift-support-in-jrebel/*, where you'll find more about the capabilities of the plugin.

1.5. Developing using Eclipse

Problem

You want to develop your Lift application using the Eclipse IDE, hitting reload in your browser to see changes.

Solution

Use the "Scala IDE for Eclipse" plugin to Eclipse, and the *sbteclipse* plugin for SBT. This will give Eclipse the ability to understand Scala, and SBT the ability to create project files which Eclipse can load.

The instructions for the Eclipse plugin are given at *http://scala-ide.org*. There are a number of options to be aware of when picking an update site to use: there a different sites for Scala 2.9 and 2.10, and for different versions of Eclipse. Start with the *stable* version of the plugin rather than a nightly or milestone version. This will give you an Eclipse perspective that knows about Scala.

Once the Eclipse plugin is installed and restarted you need to create the project files to allow Eclipse to load your Lift project. Install "sbteclipse" by adding the following to *projects/plugins.sbt* in your Lift project:

```
addSbtPlugin("com.typesafe.sbteclipse" % "sbteclipse-plugin" % "2.1.2")
```

You can then create Eclipse project files (*.project* and *.classpath*) by entering the following to the SBT prompt:

```
eclipse
```

Open the project in Eclipse by navigating to "File > Import…" and selecting "General > Existing Projects into Workspace". Browse to, and pick, your Lift project.You are now set up to develop your application in Eclipse.

To see live changes as you edit and save your work, run SBT in a separate terminal window. That is, start *sbt* from a terminal window outside of Eclipse and enter the following:

```
~; container:start; container:reload /
```

This behaviour of this command is described in Recipe 1.3, but if you're using JRebel (see Recipe 1.4) then you just need to run `container:start` by itself.

You can then edit in Eclipse, save to compile, and in your web browser hit reload to see the changes.

Discussion

One of the great benefits of an IDE is the ability to navigate source, by cmd+click (Mac) or F3 (PC). You can ask the SBT `eclipse` command to download the Lift source and Scaladoc, allowing you to click through to the Lift source from methods and classes, which is a useful way to discover more about Lift.

To achieve this in a project, run `eclipse with-source=true` in SBT, but if you want this to be the default behaviour, add the following to your *build.sbt* file:

```
EclipseKeys.withSource := true
```

If you find yourself using the plugin frequently, you may wish to declare it in your global SBT configuration files so it applies to all projects. To do that, create a *~/.sbt/plugins/plugins.sbt* file containing:

```
resolvers += Classpaths.typesafeResolver

addSbtPlugin("com.typesafe.sbteclipse" % "sbteclipse-plugin" % "2.1.2")
```

Note the blank line between the `resolvers` and the `addSbtPlugin`. In *.sbt* files, a blank line is required between statements.

Finally, set any global configurations (such as `withSource`) in *~/.sbt/global.sbt*.

See Also

There are other useful settings for sbteclipse, described at *https://github.com/typesafehub/sbteclipse/wiki*. You'll also find the latest version number for the plugin on that site.

The SBT *~/.sbt/* structure is described in the guide to using plugins at *http://www.scala-sbt.org/release/docs/Getting-Started/Using-Plugins* and in the wiki page for global configuration at *http://www.scala-sbt.org/release/docs/Detailed-Topics/Global-Settings*.

1.6. Developing using Intellij IDEA

Problem

You want to use the Intellij IDEA development environment when writing your Lift application.

Solution

You need the Scala plugin for IntelliJ, and an SBT plugin to generate the IDEA project files.

The IntelliJ plugin you'll need to install once, and these instructions are for IntelliJ IDEA 12. The details may vary between releases of the IDE, but the basic idea is to find the JetBrains Scala plugin, and download and install it.

From the "Welcome to Intellij IDEA" screen, select "Configure" and then "Plugins". Select "Browse repositories…". In the search box, top right, type "Scala". You'll find on the left a number of matches: select "Scala". On the right, you'll see confirmation that this is the "Plugin for Scala language support" and the vendor is JetBrains Inc. Select the "Download and Install" icon from the top of the window, or right click to download and install. "Close" the dialog, and OK out of the plugins window. You'll be promoted to restart IntelliJ IDEA.

With the IDE configured, you now need to add the SBT plugin inside your Lift project by adding the following to the file *projects/plugins.sbt*:

```
addSbtPlugin("com.github.mpeltonen" % "sbt-idea" % "1.4.0")
```

Start SBT and at the SBT prompt create the IDEA project files by typing:

```
gen-idea
```

This will generate the *.idea* and *.iml* files that IntelliJ uses. Inside IntelliJ you can open the project from the "File" menu, picking "Open…" and then navigating to your project and selecting the directory.

To see live changes as you edit and save your work, run SBT in a separate terminal window. That is, start *sbt* from a terminal window outside of IntelliJ and enter the following:

```
~; container:start; container:reload /
```

This behaviour of this command is described in Recipe 1.3, but if you're using JRebel (see Recipe 1.4) then you just need to run `container:start` by itself.

Each time you compile or make the project, the container will pick up the changes, and you can see them by re-loading your browser window.

Discussion

By default the `gen-idea` command will fetch source for dependent libraries. That means out-of-the-box you can click though to Lift source code to explore it and learn more about the framework.

If you want to try the latest snapshot version of the plugin, you'll need to include the snapshot repository in your *plugin.sbt* file:

```
resolvers += "Sonatype snapshots" at
  "http://oss.sonatype.org/content/repositories/snapshots/"
```

Setting up the SBT IDEA plugin globally, for all SBT projects, is the same pattern as described for Eclipse in Recipe 1.5.

See Also

The sbt-idea plugin at *https://github.com/mpeltonen/sbt-idea* doesn't have a configuration guide yet. One way to discover the features is to browse the release notes in the *notes* folder of that project.

JetBrains have a blog for the Scala plugin with feature news and tips: *http://blog.jetbrains.com/scala/*.

1.7. Viewing the lift_proto H2 Database

Problem

You're developing using the default *lift_proto.db* H2 database, and you would like use a tool to look at the tables.

Solution

Use the web interface included as part of H2. Here are the steps:

- Locate the H2 JAR file. For me, this was: *~/.ivy2/cache/com.h2database/h2/jars/h2-1.2.147.jar*.
- Start the server from a terminal window using the JAR file you found: `java -cp /path/to/h2-version.jar org.h2.tools.Server`
- This should launch your web browser, asking you to login.
- Select "Generic H2 Server" in "Saved Settings".
- Enter *jdbc:h2:/path/to/youapp/lift_proto.db;AUTO_SERVER=TRUE* for "JDBC URL", adjusting the path for the location of your database, and adjusting the name of the database ("lift_proto.db") if different in your *Boot.scala*.
- Press "Connect" to view and edit your database.

Discussion

The default Lift projects that include a database, such as *lift_basic*, use the H2 relational database as it can be included as an SBT dependency and requires no external installation or configuration. It's a fine product, although production deployments typically use standalone databases, such as PostgreSQL or MySQL.

Even if you're deploying to a non-H2 database it may be useful to keep H2 around because it has an in-memory mode, which is great for unit tests. This means you can create a database in-memory, and throw it away when your unit tests ends.

If you don't like the web interface, the connection settings described in this recipe should give you the information you need to configure other SQL tools.

See Also

The properties of H2 are described at *http://www.h2database.com*.

If you're using the console frequently, consider making it accessible from your Lift application in development node. This is described by Diego Medina in a blog post at *https://fmpwizard.telegr.am/blog/lift-and-h2*.

The example Lift project for Chapter 7 has the H2 console enabled. The source is: *https://github.com/LiftCookbook/cookbook_squeryl*.

1.8. Using the Latest Lift build

Problem

You want to use the latest ("snapshot") build of Lift.

Solution

You need to make two changes to your *build.sbt* file. First, reference the snapshot repository:

```
resolvers += "snapshots" at
  "http://oss.sonatype.org/content/repositories/snapshots"
```

Second, change the `liftVersion` in your build to be the latest version. For this example, let's use the 2.6-SNAPSHOT version of Lift:

```
val liftVersion = "2.6-SNAPSHOT"
```

Restarting SBT (or issuing a `reload` command) will trigger a download of the latest build.

Discussion

Production releases of Lift (e.g., "2.4", "2.5"), as well as milestone releases (e.g., "2.5-M3") and release candidates (e.g., "2.5-RC1") are published into a releases repository. When SBT downloads them, they are downloaded once.

Snapshot releases are different: they are the result of an automated build, and change often. You can force SBT to resolve the latest versions by running the command `clean` and then `update`.

See Also

To learn the detail of SNAPSHOT versions, dig into the Maven Complete Reference at *http://www.sonatype.com/books/mvnref-book/reference/pom-relationships-sect-pom-syntax.html*.

1.9. Using a New Version of Scala

Problem

A new Scala version has just been released and you want to immediately use it in your Lift project.

Solution

You may find that the latest snapshot of Lift is built using the latest Scala version. Failing that, and assuming you cannot wait for a build, you may still be in luck. Providing that the Scala version is *binary compatible* with the latest version used by Lift, you can change your build file to force the Scala version.

For example, assuming your *build.sbt* file is set up to use Lift 2.5 with Scala 2.9.1:

```
scalaVersion := "2.9.1"

libraryDependencies ++= {
  val liftVersion = "2.5"
  Seq(
    "net.liftweb" %% "lift-webkit" % liftVersion % "compile->default"
  )
}
```

Let's assume that you now want to use Scala 2.9.3 but Lift 2.5 was only built against Scala 2.9.1. Replace `%%` with `%` for the `net.liftweb` resources and explicitly include the Scala version that Lift was built against for each Lift component:

```
scalaVersion := "2.9.3"

libraryDependencies ++= {
  val liftVersion = "2.5"
  Seq(
    "net.liftweb" % "lift-webkit_2.9.1" % liftVersion % "compile->default"
  )
}
```

What we've done here is change the `scalaVersion` to the new version we want to use, but explicitly specified we want the 2.9.1 Scala version for Lift. This works because the two different Scala versions are binary compatible.

Discussion

Dependencies have a particular naming convention. For example, the `lift-webkit` library for Lift 2.5-RC5 is called *lift-webkit_2.9.1-2.5-RC5.jar*. Normally in *build.sbt* we simply refer to `"net.liftweb" %% "lift-webkit"` and SBT turns that into the name of a file that can be downloaded.

However, in this recipe we have forced SBT to explicitly fetch the 2.9.1 version of the Lift resources rather than allow it to compute the URL to the Lift components. This is the difference between using `%%` and `%` in a dependency: with `%%` you do not specify the Scala version as SBT will append the `scalaVersion` number automatically; with `%` this automatic change is not made, so we have to manually specify more details for the name of the library.

Please note this only works for minor releases of Scala: major releases break compatibility. For example Scala 2.9.1 is compatible with Scala 2.9.0, but not 2.10.

See Also

Binary compatibly in Scala is discussed on the Scala user mailing list at *http://article.gmane.org/gmane.comp.lang.scala.user/39290*.

The SBT behaviour is described at: *http://www.scala-sbt.org/release/docs/Getting-Started/Library-Dependencies*.

Recipe 1.8 describes how to use a snapshot version of Lift.

CHAPTER 2
HTML

Generating HTML is often a major component of web applications. This chapter is concerned with Lift's *View First* approach and use of *CSS Selectors*. Later chapters focus more specifically on form processing, REST web services, JavaScript, Ajax and Comet.

Code for this chapter is at: *https://github.com/LiftCookbook/cookbook_html*.

2.1. Testing and Debugging CSS Selectors

Problem

You want to explore or debug CSS selectors interactively.

Solution

You can use the Scala REPL to run your CSS selectors.

Here's an example were we test out a CSS selector which adds an *href* attribute to a link. Start from within SBT and use the `console` command to get into the REPL:

```
> console
[info] Starting scala interpreter...
[info]
Welcome to Scala version 2.9.1.final
Type in expressions to have them evaluated.
Type :help for more information.

scala> import net.liftweb.util.Helpers._
import net.liftweb.util.Helpers._

scala> val f = "a [href]" #> "http://example.org"
f: net.liftweb.util.CssSel =
  (Full(a [href]), Full(ElemSelector(a,Full(AttrSubNode(href)))))
```

```
scala> val in = <a>click me</a>
in: scala.xml.Elem = <a>click me</a>

scala> f(in)
res0: scala.xml.NodeSeq =
  NodeSeq(<a href="http://example.org">click me</a>)
```

The `Helpers._` import brings in the CSS Selector functionality, which we then exercise by creating a selector, `f`, calling it with a very simple template, `in`, and observing the result, `res0`.

Discussion

CSS selector transforms are one of the distinguishing features of Lift. They succinctly describe a node in your template (left-hand side) and give a replacement (operation, the right-hand side). They do take a little while to get use to, so being able to test them at the Scala REPL is useful.

It may help to know that prior to CSS selectors Lift snippets were typically defined in terms of a function that took a `NodeSeq` and returned a `NodeSeq`, often via a method called `bind`. Lift would take your template, which would be the input `NodeSeq`, apply the function, and returns a new `NodeSeq`. You won't see that usage so often any more, but the principle is the same.

The CSS selector functionality in Lift gives you a `CssSel` function which is `NodeSeq => NodeSeq`. We exploit this in the above example by constructing an input `NodeSeq` (called `in`), then creating a CSS function (called `f`). Because we know that `CssSel` is defined as a `NodeSeq => NodeSeq` the natural way to execute the selector is to supply the `in` as a parameter, and this gives us the answer, `res0`.

If you use an IDE that supports a *worksheet*, which both Eclipse and IntelliJ IDEA do, then you can also run transformations in a worksheet.

See Also

The syntax for selectors is best described in *Simply Lift* at *http://simply.liftweb.net/*.

See Recipe 1.5 and Recipe 1.6 for how to work with Eclipse and Intellij IDEA.

2.2. Sequencing CSS Selector Operations

Problem

You want your CSS selector binding to apply to the results of earlier binding expressions.

Solution

Use `andThen` rather than `&` to compose your selector expressions.

For example, suppose we want to replace `<div id="foo"/>` with `<div id="bar">bar content</div>` but for some reason we needed to generate the `bar` div as a separate step in the selector expression:

```
sbt> console
[info] Starting scala interpreter...
[info]
Welcome to Scala version 2.9.1.final (Java 1.7.0_05).
Type in expressions to have them evaluated.
Type :help for more information.

scala> import net.liftweb.util.Helpers._
import net.liftweb.util.Helpers._

scala> def render = "#foo" #> <div id="bar"/> andThen "#bar *" #> "bar content"
render: scala.xml.NodeSeq => scala.xml.NodeSeq

scala> render(<div id="foo"/>)
res0: scala.xml.NodeSeq = NodeSeq(<div id="bar">bar content</div>)
```

Discussion

When using `&` think of the CSS selectors as always applying to the original template, no matter what other expressions you are combining. This is because `&` is aggregating the selectors together before applying them. In contrast, `andThen` is a method of all Scala functions that composes two functions together, with the first being called before the second.

Compare the example above if we change the `andThen` to `&`:

```
scala> def render = "#foo" #> <div id="bar" /> & "#bar *" #> "bar content"
render: net.liftweb.util.CssSel

scala> render(<div id="foo"/>)
res1: scala.xml.NodeSeq = NodeSeq(<div id="bar"></div>)
```

The second expression will not match as it is applied to the original input of `<div id="foo"/>` — the selector of `#bar` won't match on `id=foo`, and so adds nothing to the results of `render`.

See Also

The Lift Wiki page for CSS Selectors also describes this use of `andThen`: *https://www.assembla.com/spaces/liftweb/wiki/Binding_via_CSS_Selectors*.

2.3. Setting Meta Tag Contents

Problem

You want to set the content of an HTML meta tag from a snippet.

Solution

Use the `@` CSS binding name selector. For example, given:

```
<meta name="keywords" content="words, here, please" />
```

The following snippet code will update the value of the contents attribute:

```
"@keywords [content]" #> "words, we, really, want"
```

Discussion

The `@` selector selects all elements with the given name. It's useful in this case to change `<meta name="keyword">` tag, but you may also see it used with elsewhere. For example, in an HTML form you can select input fields such as `<input name="address">` with `"@address"`.

The `[content]` part is an example of a *replacement rule* that can follow a selector. That's to say, it's not specific to the `@` selector and can be used with other selectors. In this example it replaces the value of the attribute called "content". If the meta tag had no "content" attribute, it would be added.

There are two other replacement rules useful for manipulating attributes:

- `[content!]` to remove an attribute with a matching value.
- `[content+]` to append to the value.

Examples of these would be…

```
scala> import net.liftweb.util.Helpers._
import net.liftweb.util.Helpers._

scala> val in = <meta name="keywords" content="words, here, please" />
in: scala.xml.Elem = <meta name="keywords" content="words, here, please"></meta>

scala> val remove = "@keywords [content!]" #> "words, here, please"
remove: net.liftweb.util.CssSel = CssBind(Full(@keywords [content!]),
  Full(NameSelector(keywords,Full(AttrRemoveSubNode(content)))))

scala> remove(in)
res0: scala.xml.NodeSeq = NodeSeq(<meta name="keywords"></meta>)
```

…and…

```
scala> val add = "@keywords [content+]" #> ", thank you"
add: net.liftweb.util.CssSel = CssBind(Full(@keywords [content+]),
  Full(NameSelector(keywords,Full(AttrAppendSubNode(content)))))

scala> add(in)
res1: scala.xml.NodeSeq = NodeSeq(<meta content="words, here, please, thank you"
  name="keywords"></meta>)
```

Appending to a `class` Attribute

Although not directly relevant to `meta` tags, you should be aware of there is one convenient special case for appending to an attribute. If the attribute is `class`, a space is added together with your class value. As a demonstration of that, here's an example of appending a class called "btn-primary" to a `div`:

```
scala> def render = "div [class+]" #> "btn-primary"
render: net.liftweb.util.CssSel

scala> render(<div class="btn"/>)
res0: scala.xml.NodeSeq = NodeSeq(<div class="btn btn-primary"></div>)
```

See Also

The syntax for selectors is best described in *Simply Lift* at *http://simply.liftweb.net/*.

See Recipe 2.1 for how to run selectors from the REPL.

2.4. Setting the Page Title

Problem

You want to set the `<title>` of the page from a Lift snippet.

Solution

Select the content of the `title` element and replace it with the text you want:

```
"title *" #> "I am different"
```

Assuming you have a `<title>` tag in your template, the above will result in:

```
<title>I am different</title>
```

Discussion

This example uses a element selector, which picks out tags in the HTML template, and replaces the content. Notice that we are using "title **" to select the *content* of the title tag. If we had left off the ""** the entire title tag would have been replaced with text.

As an alternative, it is also possible to set the page title from the contents of `SiteMap`, meaning the title used will be the title you've assigned to the page in the site map. To do that, make use of `Menu.title` in your template directly:

```
<title data-lift="Menu.title"></title>
```

The `Menu.title` code appends to any existing text in the title. This means the following will have the phrase "Site Title - " in the title followed by the page title:

```
<title data-lift="Menu.title">Site Title - </title>
```

If you need more control, you can of course bind on `<title>` using a regular snippet. This example uses a custom snippet to put the site title after the page title:

```
<title data-lift="MyTitle"></title>

object MyTitle {
  def render = <title><lift:Menu.title /> - Site Title</title>
}
```

Notice that our snippet is returning another snippet, `<lift:Menu.title/>`. This is a perfectly normal thing to do in Lift, and snippet invocations returned from snippets will be processed by Lift as normal.

See Also

Recipe 2.7 describes the different ways to reference a snippet, such as `data-lift` and `<lift: ... />`.

At *https://www.assembla.com/spaces/liftweb/wiki/SiteMap* there's more about Site Map and the `Menu` snippets.

2.5. HTML Conditional Comments

Problem

You want to make use of Internet Explorer HTML conditional comments in your templates.

Solution

Put the markup in a snippet and include the snippet in your page or template.

For example, suppose we want to include the HTML5 Shiv (a.k.a. HTML5 Shim) JavaScript so we can use HTML5 elements with legacy IE browsers. To do that our snippet would be:

```
package code.snippet
```

```
import scala.xml.Unparsed

object Html5Shiv {
  def render = Unparsed("""<!--[if lt IE 9]>
    <script src="http://html5shim.googlecode.com/svn/trunk/html5.js">
    </script><![endif]-->""")
}
```

We would then reference the snippet in the <head> of a page, perhaps even in all pages via *templates-hidden/default.html*:

```
<script data-lift="Html5Shiv"></script>
```

Discussion

The HTML5 parser used by Lift does not carry comments from the source through to the rendered page. If you just tried to paste the *html5shim* markup into your template you'd find it missing from the rendered page.

We deal with this by generating unparsed markup from a snippet. If you're looking at `Unparsed` and worried, your instincts are correct. Normally Lift would cause the markup to be escaped, but in this case we really do want unparsed XML content (the comment tag) included in the output.

If you find you're using IE conditional comments frequently, you may want to create a more general version of the snippet. For example:

```
package code.snippet

import xml.{NodeSeq, Unparsed}
import net.liftweb.http.S

object IEOnly {

  private def condition : String =
    S.attr("cond") openOr "IE"

  def render(ns: NodeSeq) : NodeSeq =
    Unparsed("<!--[if " + condition + "]>") ++ ns ++ Unparsed("<![endif]-->")
}
```

It would be used like this…

```
<div data-lift="IEOnly">
  A div just for IE
</div>
```

…and produces output like this:

```
<!--[if IE]><div>
  A div just for IE
</div><![endif]-->
```

Notice that the `condition` test defaults to "IE", but first tries to look for an attribute called "cond". This allows you to write:

```
<div data-lift="IEOnly?cond=lt+IE+9">
  You're using IE 8 or earlier
</div>
```

The + symbol is the the URL encoding for a space, resulting in:

```
<!--[if lt IE 9]><div>
  You're using IE 8 or earlier
</div><![endif]-->
```

See Also

The `IEOnly` example is derived from a posting on the mailing list from Antonio Salazar Cardozo: *http://bit.ly/lift-ieonly*.

The html5shim project can be found at: *http://code.google.com/p/html5shim/*.

2.6. Returning Snippet Markup Unchanged

Problem

You want a snippet to return the original markup associated with the snippet invocation.

Solution

Use the `PassThru` transform.

Suppose you have a snippet which performs a transforms when some condition is met, but if the condition is not met, you want the snippet return the original markup.

Starting with the original markup…

```
<h2>Pass Thru Example</h2>

<p>There's a 50:50 chance of seeing "Try again" or "Congratulations!":</p>

<div data-lift="PassThruSnippet">
  Try again - this is the template content.
</div>
```

…we could leave it alone or change it with this snippet:

```
package code.snippet

import net.liftweb.util.Helpers._
import net.liftweb.util.PassThru

import scala.util.Random
```

```
import xml.Text

class PassThruSnippet {

  private def fiftyFifty = Random.nextBoolean

  def render =
    if (fiftyFifty) "*" #> Text("Congratulations! The content was changed")
    else PassThru

}
```

Discussion

`PassThru` is an *identity function* of type `NodeSeq => NodeSeq`. It returns the input it is given:

```
object PassThru extends Function1[NodeSeq, NodeSeq] {
  def apply(in: NodeSeq): NodeSeq = in
}
```

A related example is `ClearNodes`, defined as:

```
object ClearNodes extends Function1[NodeSeq, NodeSeq] {
  def apply(in: NodeSeq): NodeSeq = NodeSeq.Empty
}
```

The pattern of converting one `NodeSeq` to another is simple, but also powerful enough to get you out of most situations as you can always arbitrarily re-write the `NodeSeq`.

2.7. Snippet Not Found when using HTML 5

Problem

You're using Lift with the HTML 5 parser and one of your snippets is rendering with a "Class Not Found" error. It even happens for `<lift:HelloWorld.howdy />`.

Solution

Switch to the designer-friendly snippet invocation mechanism. E.g.,

```
<div data-lift="HellowWorld.howdy"></div>
```

Discussion

In this Cookbook we use the HTML 5 parser, which is set in *Boot.scala*:

```
// Use HTML5 for rendering
LiftRules.htmlProperties.default.set( (r: Req) =>
  new Html5Properties(r.userAgent) )
```

The HTML5 parser and the traditional Lift XHTML parser have different behaviours. In particular the HTML5 parser converts elements and attribute names to lower case when looking up snippets. This means Lift would take `<lift:HelloWorld.howdy />` and look for a class called "helloworld" rather than "HelloWorld", which would be the cause of the "Class Not Found Error".

Switching to the designer-friendly mechanism is the solution here, and you gain validating HTML as a bonus.

There are three popular ways of referencing a snippet:

- As an HTML 5 data attribute: `data-lift="MySnippet"`. This is style we use in this book, and is valid HTML 5 markup.
- Using the "lift" attribute, as in: `lift="MySnippet"`. This won't strictly validate against HTML 5, but you may see it.
- The XHTML namespace version: `<lift:MySnippet />`. You'll see the usage of this tag in templates declining because of the way it interacts with the HTML5 parser. However, it works just fine outside of a template, for example when embedding a snippet invocation in your server side code (Recipe 2.4 includes an example of this for "Menu.title").

See Also

The key differences between the XHTML and HTML5 parser are outlined on the mailing list at *http://bit.ly/lift-parsers*.

2.8. Avoiding CSS and JavaScript Caching

Problem

You've modified CSS or JavaScript in your application, but web browsers have cached your resources and are using the older versions. You'd like to avoid this browser caching.

Solution

Add the `with-resource-id` attribute to script or link tags:

```
<script data-lift="with-resource-id" src="/myscript.js"
 type="text/javascript"></script>
```

The addition of this attribute will cause Lift to append a *resource ID* to your `src` (or `href`), and as this resource ID changes each time Lift starts, it defeats browser caching.

The resultant HTML might be:

```
<script src="/myscript.js?F6197328978246UCAAN=_"
  type="text/javascript" ></script>
```

Discussion

The random value that is appended to the resource is computed when your Lift application boots. This means it should be stable between releases of your application.

If you need some other behaviour from `with-resource-id` you can assign a new function of type `String => String` to `LiftRules.attachResourceId`. The default implementation, shown above, takes the resource name, */myscript.js* in the example, and returns the resource name with an id appended.

You can also wrap a number of tags inside a `<lift:with-resource-id>...<lift:with-resource-id>` block. However, avoid doing this in the `<head>` of your page as the HTML5 parser will move the tags to be outside of the head section.

Note that some proxies may choose not to cache resources with query parameters at all. If that impacts you, it's possible to code a custom resource ID method to move the random resource ID out of the query parameter and into the path.

Here's one approach to doing this. Rather than generate JavaScript and CSS links that look like */assets/style.css?F61973*, we will generate */cache/F61973/assets/style.css*. We then will need to tell Lift to take requests that look like this new format, and render the correct content for the request. The code for this is:

```
package code.lib

import net.liftweb.util._
import net.liftweb.http._

object CustomResourceId {

 def init() : Unit = {
  // The random number we're using to avoid caching
  val resourceId = Helpers.nextFuncName

  // Prefix with-resource-id links with "/cache/{resouceId}"
  LiftRules.attachResourceId = (path: String) => {
   "/cache/" + resourceId + path
  }

  // Remove the cache/{resourceId} from the request if there is one
  LiftRules.statelessRewrite.prepend( NamedPF("BrowserCacheAssist") {
   case RewriteRequest(ParsePath("cache" :: id :: file, suffix, _, _), _, _) =>
    RewriteResponse(file, suffix)
  })

 }
}
```

This would be initialised in *Boot.scala*…

```
CustomResourceId.init()
```

…or you could just paste all the code into *Boot.scala*, if you prefer.

With the code in place, we can, for example, modify *templates-hidden/default.html* and add a resource ID class to jQuery:

```
<script id="jquery" data-lift="with-resource-id"
  src="/classpath/jquery.js" type="text/javascript"></script>
```

At run-time this would be rendered in HTML as:

```
<script type="text/javascript" id="jquery"
  src="/cache/F352555437877UHCNRW/classpath/jquery.js"></script>
```

Most of the work for this is happening in the `statelessRewrite`, which is working at a low-level inside Lift. The two parts to it are:

- A `RewriteRequest` which is the pattern we're matching on; and
- A `RewriteResponse` which is the result we want if the request matches.

Looking at the `RewriteRequest` first, this expects three arguments: the path, which we care about, and then the method (e.g., `GetRequest`, `PutRequest`, etc) and the `HTTPRequest` itself, neither of which concern us in this instance. In the path part we're matching on patterns that start with "cache", followed by something (we don't care what), and then the rest of the path, represented by the name `file`. In that situation, we rewrite to the original path, which is just the `file` with the `suffix`, effectively removing the */cache/F352555437877UHCNRW* part. This is the content that Lift will serve.

See Also

The source for `LiftRules` shows the default implementation of `attachResourceId`: *https://github.com/lift/framework/blob/master/web/webkit/src/main/scala/net/liftweb/http/LiftRules.scala*.

Google's *Optimize caching* notes are a good source of information about browser behaviour: *https://developers.google.com/speed/docs/best-practices/caching*.

You can learn more about URL re-writing at the Lift wiki: *https://www.assembla.com/spaces/liftweb/wiki/URL_Rewriting*.

2.9. Adding to the Head of a Page

Problem

You use a template for layout, but on one specific page you need to add something to the <head> section.

Solution

Use the head snippet so Lift knows to merge the contents with the <head> of your page. For example, suppose you have the following contents in *templates-hidden/default.html*:

```
<html lang="en" xmlns:lift="http://liftweb.net/">
  <head>
    <meta charset="utf-8"></meta>
    <title data-lift="Menu.title">App: </title>
    <script id="jquery" src="/classpath/jquery.js"
      type="text/javascript"></script>
    <script id="json" src="/classpath/json.js"
      type="text/javascript"></script>
 </head>
 <body>
     <div id="content">The main content will get bound here</div>
 </body>
</html>
```

Also suppose you have *index.html* on which you want to include *red-titles.css* to just change the style of just this page.

Do so by including the CSS in the part of the page that will get processed, and mark it with the head snippet:

```
<!DOCTYPE html>
<html>
 <head>
   <title>Special CSS</title>
 </head>
 <body data-lift-content-id="main">
  <div id="main" data-lift="surround?with=default;at=content">
    <link data-lift="head" rel="stylesheet"
       href="red-titles.css" type="text/css" />
    <h2>Hello</h2>
  </div>
 </body>
</html>
```

Note that this *index.html* page is validated HTML 5, and will produce a result with the custom CSS inside the <head> tag, something like this:

```
<!DOCTYPE html>
<html lang="en">
```

```
<head>
 <meta charset="utf-8">
 <title>App:  Special CSS</title>
 <script type="text/javascript"
   src="/classpath/jquery.js" id="jquery"></script>
 <script type="text/javascript"
   src="/classpath/json.js" id="json"></script>
 <link rel="stylesheet" href="red-titles.css" type="text/css">
</head>
<body>
  <div id="main">
    <h2>Hello</h2>
  </div>
 <script type="text/javascript" src="/ajax_request/liftAjax.js"></script>
 <script type="text/javascript">
 // <![CDATA[
 var lift_page = "F557573613430HI02U4";
 // ]]>
 </script>
</body>
</html>
```

Discussion

If you find your tags not appearing the the `<head>` section, check that the HTML in your template and page is valid HTML 5.

You can also use `<lift:head>...</lift:head>` to wrap a number of expressions, and will see `<head_merge>...</head_merge>` used in code example as an alternative to `<lift:head>`.

Another variant you may see is `class="lift:head"`, as an alternative to `data-lift="head"`.

The `head` snippet is a built-in snippet, but otherwise no different from any snippet you might write. What the snippet does is emit a `<head>` block, containing the elements you want in the head. These can be `<title>`, `<link>`, `<meta>`, `<style>`, `<script>` or `<base>` tags. How does this `<head>` block produced by the `head` snippet end up inside the main `<head>` section of the page? When Lift processes your template, it automatically merges all `<head>` tags into the main `<head>` section of the page.

You might suspect you can therefore put a plain old `<head>` section anywhere on your template. You can, but that would not necessarily be valid HTML 5 mark up.

There's also `tail` which works in a similar way, except anything marked with this snippet is moved to be just before the close of the body tag.

See Also

Recipe 5.7 describes how to move JavaScript to the end of the page with the `tail` snippet.

The W3C HTML validator is a useful tool for tracking down HTML markup issues that may cause problems with content being moved into the head of your page. *http://validator.w3.org/*.

2.10. Custom 404 Page

Problem

You want to show a customised "404" (not found) page.

Solution

In *Boot.scala* add the following:

```
import net.liftweb.util._
import net.liftweb.http._

LiftRules.uriNotFound.prepend(NamedPF("404handler"){
  case (req,failure) =>
    NotFoundAsTemplate(ParsePath(List("404"),"html",true,false))
})
```

The file *src/main/webapp/404.html* will now be served for requests to unknown resources.

Discussion

The `uriNotFound` Lift rule needs to return a `NotFound` in reply to a `Req` and `Box[Failure]`. This allows you to customise the response based on the request and the type of failure.

There are three types of `NotFound`:

- `NotFoundAsTemplate` — useful to invoke the Lift template processing facilities from a `ParsePath`.
- `NotFoundAsResponse` — allows you to return a specific `LiftResponse`.
- `NotFoundAsNode` — wrappers a `NodeSeq` for Lift to translate into a 404 response.

In the example we're matching any not found situation, regardless of the request and the failure, and evaluating this as a resource identified by `ParsePath`. The path we've used is */404.html*.

In case you're wondering, the last two `true` and `false` arguments to `ParsePath` indicates the path we've given is absolute, and doesn't end in a slash. `ParsePath` is a representation for a URI path, and exposing if the path is absolute or ends in a slash are useful flags for matching on, but in this case, they're not relevant.

Be aware that 404 pages, when rendered this way, won't have a location in the site map. That's because we've not included the *404.html* file in the site map, and we don't have to because we're rendering via `NotFoundAsTemplate` rather than sending a redirect to */404.html*. However, this means that if you display an error page using a template that contains `Menu.builder` or similar (as *templates-hidden/default.html* does), you'll see "No Navigation Defined". In that case, you'll probably want to use a different template on your 404 page.

As an alternative, you could include the 404 page in your site map but make it hidden when the site map is displayed via the `Menu.builder`:

```
Menu.i("404") / "404" >> Hidden
```

See Also

Recipe 6.5 for how to catch any exception thrown from your code.

2.11. Other Custom Status Pages

Problem

You want to show a customised page for certain HTTP status codes.

Solution

Use `LiftRules.responseTransformers` to match against the response and supply an alternative.

As an example, suppose we want to provide a custom page for 403 ("Forbidden") statuses created in our Lift application. Further suppose that this page might contain snippets so will need to pass through the Lift rendering flow.

To do this in *Boot.scala* we define the `LiftResponse` we want to generate and use the response when a 403 status is about to be produced by Lift:

```
def my403 : Box[LiftResponse] =
  for {
    session <- S.session
    req <- S.request
    template = Templates("403" :: Nil)
    response <- session.processTemplate(template, req, req.path, 403)
  } yield response
```

```
LiftRules.responseTransformers.append {
  case resp if resp.toResponse.code == 403 => my403 openOr resp
  case resp => resp
}
```

The file *src/main/webapp/403.html* will now be served for requests that generate 403 status codes. Other, non-403, responses are left untouched.

Discussion

`LiftRules.responseTransformers` allows you to supply `LiftResponse => LiftResponse` functions to change a response right at the end of the HTTP processing cycle. This is a very general mechanism: in this example we are matching on a status code, but we could match on anything exposed by `LiftResponse`.

In the recipe we respond with a template, but you may find situations where other kinds of response make sense, such as an `InMemoryResponse`.

You could even simplify the example to just this:

```
LiftRules.responseTransformers.append {
  case resp if resp.toResponse.code == 403 => RedirectResponse("/403.html")
  case resp => resp
}
```

In Lift 3 `responseTransformers` will be modified to be a partial function, meaning you'll be able to leave off the final `case r => r` part of this example.

That redirect will work just fine, with the only downside that the HTTP status code sent back to the web browser won't be a 403 code.

A more general approach, if you're customising a number of pages, would be to define the status codes you want to customise, create a page for each, and then only match on those pages:

```
LiftRules.responseTransformers.append {
  case Customised(resp) => resp
  case resp => resp
}

object Customised {

  // The pages we have customised: 403.html and 500.html
  val definedPages = 403 :: 500 :: Nil

  def unapply(resp: LiftResponse) : Option[LiftResponse] =
```

```
      definedPages.find(_ == resp.toResponse.code).flatMap(toResponse)

    def toResponse(status: Int) : Box[LiftResponse] =
      for {
        session <- S.session
        req <- S.request
        template = Templates(status.toString :: Nil)
        response <- session.processTemplate(template, req, req.path, status)
    } yield response

  }
```

The convention in `Customised` is that we have a HTML file in *src/main/webapp* which matches the status code we want to show, but of course you can change that by using a different pattern in the argument to `Templates`.

One way to test the above examples is to add the following to *Boot.scala* to make all requests to */secret* return a 403:

```
val Protected = If(() => false, () => ForbiddenResponse("No!"))

val entries = List(
  Menu.i("Home") / "index",
  Menu.i("secret") / "secret" >> Protected,
  // rest of your site map here...
)
```

If you request */secret*, a 403 response will be triggered, which will match the response transformer showing you the contents of the *403.html* template.

See Also

Recipe 2.10 explains the built-in support for custom 404 messages.

Recipe 6.5 for how to catch any exception thrown from your code.

2.12. Links in Notices

Problem

You want to include a clickable link in your `S.error`, `S.notice` or `S.warning` messages.

Solution

Include a `NodeSeq` containing a link in your notice:

```
S.error("checkPrivacyPolicy",
  <span>See our <a href="/policy">privacy policy</a></span>)
```

You might pair this with the following in your template…

```
<span data-lift="Msg?id=checkPrivacyPolicy"></span>
```

Discussion

You may be more familiar with the `S.error(String)` signature of Lift notices than the versions that take a `NodeSeq` as an argument, but the `String` versions just convert the `String` argument to a `scala.xml.Text` kind of `NodeSeq`.

See Also

Lift notices are described on the Wiki: *http://www.assembla.com/spaces/liftweb/wiki/Lift_Notices_and_Auto_Fadeout*.

2.13. Link to Download Data

Problem

You want a button or a link which, which clicked, will trigger a download in the browser rather than visiting a page.

Solution

Create a link using `SHtml.link`, provide a function to return a `LiftResponse` and wrap the response in a `ResponseShortcutException`.

As an example, we will create a snippet that shows the user a poem and provides a link to download the poem as a text file. The template for this snippet will present each line of the poem separated by a `<br>`:

```
<h1>A poem</h1>

<div data-lift="DownloadLink">
  <blockquote>
    <span class="poem">
        <span class="line">line goes here</span> <br />
    </span>
  </blockquote>
  <a href="">download link here</a>
</div>
```

The snippet itself will render the poem and replace the download link with one which will send a response that the browser will interpret as a file to download:

```
package code.snippet

import net.liftweb.util.Helpers._
import net.liftweb.http._
import xml.Text
```

```
class DownloadLink {

  val poem =
    "Roses are red," ::
    "Violets are blue," ::
    "Lift rocks!" ::
    "And so do you." :: Nil

  def render =
    ".poem" #> poem.map(line => ".line" #> line) &
    "a" #> downloadLink

  def downloadLink =
    SHtml.link("/notused",
      () => throw new ResponseShortcutException(poemTextFile),
      Text("Download") )

  def poemTextFile : LiftResponse =
    InMemoryResponse(
      poem.mkString("\n").getBytes("UTF-8"),
      "Content-Type" -> "text/plain; charset=utf8" ::
      "Content-Disposition" -> "attachment; filename=\"poem.txt\"" :: Nil,
      cookies=Nil, 200)
}
```

Recall that `SHtml.link` generates a link and executes a function you supply before following the link.

The trick here is that wrapping the `LiftResponse` in a `ResponseShortcutException` will indicate to Lift that the response is complete, so the page being linked to (`notused`) won't be processed. The browser is happy: it has a response to the link the user clicked on, and will render it how it wants to, which in this case will probably be by saving the file to disk.

Discussion

`SHtml.link` works by generating a URL which Lift associates with the function you give it. On a page called `downloadlink`, the URL will look something like:

```
downloadlink?F845451240716XSXE3G=_#notused
```

When that link is followed, Lift looks up the function and executes it, before processing the linked-to resource. However, in this case, we are short-cutting the Lift pipeline by throwing this particular exception. This is caught by Lift and the response wrapped by the exception is taken as the final response from the request.

This short-cutting is used by `S.redirectTo` via `ResponseShortcutException.redirect`. This companion object also defines `shortcutResponse` which you can use like this:

```
import net.liftweb.http.ResponseShortcutException._

def downloadLink =
  SHtml.link("/notused",
    () => {
      S.notice("The file was downloaded")
      throw shortcutResponse(poemTextFile)
    },
    Text("Download") )
```

We've included a `S.notice` to highlight that `throw shortcutResponse` will process Lift notices when the page next loads, whereas `throw new ResponseShortcutException` does not. In this case, the notice will not appear when the user downloads the file, but it will be included the next time notices are shown, such as when the user navigates to another page. For many situations, the difference is immaterial.

This recipe has used Lift's stateful features. You can see how useful it is to be able to close over state (the poem), and offer the data for download from memory. If you've created a report from a database, you can offer it as a download without having to re-generate the items from the database.

However, in other situations you might want to avoid holding this data as a function on a link. In that case, you'll want to create a REST service that returns a `LiftResponse`.

See Also

Chapter 4 looks at REST-based services in Lift.

Recipe 6.6 discusses `InMemoryResponse` and similar responses to return content to the browser

For reports, the Apache POI project, *http://poi.apache.org/*, includes libraries for generating Excel files; and OpenCSV, *http://opencsv.sourceforge.net*, is a library for generating CSV files.

2.14. Test on a Req

Problem

You want to be able to test a function that needs a `Req`.

Solution

Supply a mock request to Lift's `MockWeb.testReq`, and run your test in the context of the `Req` supplied by `testReq`.

The first step is to add Lift's Test Kit as dependency to your project in *build.sbt*:

```
libraryDependencies += "net.liftweb" %% "lift-testkit" % "2.5-RC5" % "test"
```

To demonstrate how to use `testReq` we will test a function that detects a Google crawler. Google identifies crawlers via various "User-Agent" header on a request, so the function we want to test would look like this:

```
package code.lib

import net.liftweb.http.Req

object RobotDetector {

  val botNames =
    "Googlebot" ::
    "Mediapartners-Google" ::
    "AdsBot-Google" :: Nil

  def known_?(ua: String) =
    botNames.exists(ua contains _)

  def googlebot_?(r: Req) : Boolean =
    r.header("User-Agent").exists(known_?)
}
```

We have the list of magic `botNames` that Google sends as a user agent, and we define a check, `known_?`, that takes the user agent string and looks to see if any robot satisfies the condition of being contained in that user agent string.

The `googlebot_?` method is given a Lift `Req` object and from this we look up the header. This evaluates to a `Box[String]` as it's possible there is no header. We find the answer by seeing if there exists in the `Box` a value that satisfies the `known_?` condition.

To test this, we create a user agent string, prepare a `MockHttpServletRequest` with the header, and use Lift's `MockWeb` to turn the low-level request into a Lift `Req` for us to test with:

```
package code.lib

import org.specs2.mutable._
import net.liftweb.mocks.MockHttpServletRequest
import net.liftweb.mockweb.MockWeb

class SingleRobotDetectorSpec extends Specification {

  "Google Bot Detector" should {

    "spot a web crawler" in {

      val userAgent = "Mozilla/5.0 (compatible; Googlebot/2.1)"

      // Mock a request with the right header:
```

```
        val http = new MockHttpServletRequest()
        http.headers = Map("User-Agent" -> List(userAgent))

        // Test with a Lift Req:
        MockWeb.testReq(http) { r =>
          RobotDetector.googlebot_?(r) must beTrue
        }
      }

    }

  }
```

Running this from SBT with the `test` command would produce:

```
[info] SingleRobotDetectorSpec
[info]
[info] Google Bot Detector should
[info] + spot a web crawler
[info]
[info] Total for specification SingleRobotDetectorSpec
[info] Finished in 18 ms
[info] 1 example, 0 failure, 0 error
```

Discussion

Although `MockWeb.testReq` is handling the creation of a `Req` for us, the environment for that `Req` is supplied by the `MockHttpServletRequest`. To configure a request, create an instance of the mock and mutate the state of it as required before using it with `testReq`.

Aside from HTTP headers, you can set cookies, content type, query parameters, the HTTP method, authentication type, and the body. There are variations on the `body` assignment, which conveniently set the content type depending on the value you assign:

- `JValue` will use content type of *application/json*.
- `NodeSeq` will use *text/xml* (or you can supply an alternative).
- `String` uses *text/plain* (unless you supply an alternative).
- `Array[Byte]` does not set the content type.

Data Table

In the example test above it would be tedious to have to set up the same code repeatedly for different user agents. Specs2's *Data Table* provides a compact way to run different example values through the same test:

```
package code.lib
```

```
import org.specs2._
import matcher._
import net.liftweb.mocks.MockHttpServletRequest
import net.liftweb.mockweb.MockWeb

class RobotDetectorSpec extends Specification with DataTables {

  def is = "Can detect Google robots" ^ {
    "Bot?" || "User Agent" |
    true   !! "Mozilla/5.0 (Googlebot/2.1)" |
    true   !! "Googlebot-Video/1.0" |
    true   !! "Mediapartners-Google" |
    true   !! "AdsBot-Google" |
    false  !! "Mozilla/5.0 (KHTML, like Gecko)" |> {
    (expectedResult, userAgent) => {
      val http = new MockHttpServletRequest()
      http.headers = Map("User-Agent" -> List(userAgent))
      MockWeb.testReq(http) { r =>
        RobotDetector.googlebot_?(r) must_== expectedResult
      }
     }
    }

  }

}
```

The core of this test is essentially unchanged: we create a mock, set the user agent, and check the result of `googlebot_?`. The difference is that Specs2 is providing a neat way to list out the various scenarios and pipe them through a function.

The output from running this under SBT would be:

```
[info] Can detect Google robots
[info] + Bot?  | User Agent
[info]   true  | Mozilla/5.0 (Googlebot/2.1)
[info]   true  | Googlebot-Video/1.0
[info]   true  | Mediapartners-Google
[info]   true  | AdsBot-Google
[info]   false | Mozilla/5.0 (KHTML, like Gecko)
[info]
[info] Total for specification RobotDetectorSpec
[info] Finished in 1 ms
[info] 1 example, 0 failure, 0 error
```

Although the expected value appears first in our table, there's no requirement to put it first.

See Also

The Lift wiki discusses this topic and also other approaches such as testing with Selenium. *https://www.assembla.com/spaces/liftweb/wiki/Testing_Lift_Applications*.

2.15. Rendering Textile Markup

Problem

You want to render Textile markup in your web application.

Solution

Install the Lift Textile module in your *build.sbt* file by adding the following to the list of dependencies:

```
"net.liftmodules" %% "textile_2.5" % "1.3"
```

You can then use the module to render Textile using the `toHtml` method.

For example, starting SBT after adding the module and running the SBT *console* command allows you to try out the module:

```
scala> import net.liftmodules.textile._
import net.liftmodules.textile._

scala> TextileParser.toHtml("""
 | h1. Hi!
 |
 | The module in "Lift":http://www.liftweb.net for turning Textile markup
 | into HTML is pretty easy to use.
 |
 | * As you can see.
 | * In this example.
 | """)
res0: scala.xml.NodeSeq =
NodeSeq(, <h1>Hi!</h1>,
, <p>The module in <a href="http://www.liftweb.net">Lift</a> for turning Textile
  markup<br></br>into HTML is pretty easy to use.</p>,
, <ul><li> As you can see.</li>
<li> In this example.</li>
</ul>,
, )
```

It's a little easier to see the output if we pretty print it:

```
scala> val pp = new PrettyPrinter(width=35, step=2)
pp: scala.xml.PrettyPrinter = scala.xml.PrettyPrinter@54c19de8

scala> pp.formatNodes(res0)
res1: String =
<h1>Hi!</h1><p>
  The module in
  <a href="http://www.liftweb.net">
    Lift
  </a>
  for turning Textile markup
```

```
    <br></br>
    into HTML is pretty easy to use.
  </p><ul>
    <li> As you can see.</li>
    <li> In this example.</li>
  </ul>
```

Discussion

There's nothing special code has to do to become a Lift module, although there are common conventions: they typically are packaged as *net.liftmodules*, but don't have to be; they usually depend on a version of Lift; they sometimes use the hooks provided by `LiftRules` to provide a particular behaviour. Anyone can create and publish a Lift module, and anyone can contribute to existing modules. In the end, they are declared as dependencies in SBT, and pulled into your code just like any other dependency.

The dependency name is made up of two elements: the name and the "edition" of Lift that the module is compatible with, as shown in Figure 2-1. By "edition" we just mean the first part of the Lift version number. This implies the module is compatible with any Lift release that starts "2.5".

Figure 2-1. The structure of a module version.

This structure has been adopted because modules have their own release cycle, independent of Lift. However, modules may also depend on certain features of Lift, and Lift may change APIs between major releases, hence the need to use part of the Lift version number to identify the module.

See Also

There's no real specification of what Textile is, but there are references available which cover the typical kinds of mark up to enter and what HTML you can expect to see: *http://redcloth.org/hobix.com/textile/*.

The home of the Textile module: *https://github.com/liftmodules/textile*.

The unit tests for the Textile module give you a good set of examples of what is supported: *https://github.com/liftmodules/textile/blob/master/src/test/scala/net/liftmodules/textile/TextileSpec.scala*.

Recipe 11.6 describes how to create modules.

CHAPTER 3

Forms Processing in Lift

This chapter looks at how to process form data with Lift: submitting forms, working with form elements. The end result of a form submission can be records being updated in a database, so you may also find Chapter 7 or Chapter 8 useful, as they discuss relational databases and MongoDB, respectively.

To the extend that form processing is passing data to a server, there are also recipes in Chapter 5 that are relevant to form processing.

You'll find many of the examples from this chapter as source code at: *https://github.com/LiftCookbook/cookbook_forms*.

3.1. Plain Old Form Processing

Problem

You want to process form data in a regular old-fashioned, non-Ajax, way.

Solution

Extract form values with `S.param`, process the values, and produce some output.

For example, we can show a form, process an input value, and give a message back as a notice. The template is a regular HTML form, with the addition of a snippet:

```
<form data-lift="Plain" action="/plain" method="post">
  <input type="text" name="name" placeholder="What's your name?">
  <input type="submit" value="Go">
</form>
```

In the snippet we can pick out the value of the field "name" with `S.param("name")`:

```
package code.snippet

import net.liftweb.common.Full
import net.liftweb.http.S
import net.liftweb.util.PassThru

object Plain {

  def render = S.param("name") match {
    case Full(name) =>
      S.notice("Hello "+name)
      S.redirectTo("/plain")
    case _ =>
      PassThru
  }

}
```

The first time through this snippet there will be no parameter so we just pass back the form unchanged to the browser, which is what `PassThru` is doing. You can then enter a value into the "name" field and submit the form. This will result in Lift processing the template again, but this time with a value for the "name" input. The result will be your browser redirected to a page with a message set for display.

Discussion

Manually plucking parameters from a request isn't making the best use of Lift, but sometimes you need to do it, and `S.param` is the way you can process request parameters.

The result of `S.param` is a `Box[String]`, and in the above example we pattern match on a value. With more than one parameter you're probably see `S.param` used in this way:

```
def render = {
  for {
    name <- S.param("name")
    pet <- S.param("petName")
  } {
    S.notice("Hello %s and %s".format(name,pet))
    S.redirectTo("/plain")
  }

 PassThru
}
```

If both "name" and "petName" are provided, the body of the `for` will be evaluated.

Related function on `S` include:

- `S.params(name)` — producing a `List[String]` for all the request parameters with the given name.

- `S.post_?`, `S.get_?` — to see if the request was a GET or POST.
- `S.getRequestHeader(name)` — giving the `Box[String]` for a header in the request with the given name.
- `S.request` — to access the `Box[Req]` which gives you access to further HTTP-specific information about the request.

As an example of using `S.request` we could produce a `List[String]` for the values of all request parameters that have "name" somewhere in their parameter name:

```
val names = for {
  req <- S.request.toList
  paramName <- req.paramNames
  if paramName.toLowerCase contains "name"
  value <- S.param(paramName)
} yield value
```

Note that by opening up `S.request` we can access all the parameter names via `paramNames` function on `Req`.

Screen or Wizard provide alternatives for form processing, but sometimes you just want to pull values from a request, as demonstrated in this recipe.

See Also

Simply Lift covers a variety of ways of processing forms. See: *http://simply.liftweb.net*.

3.2. Ajax Form Processing

Problem

You want to process a form on the server via Ajax, without reloading the whole page.

Solution

Mark your form as an Ajax form with `data-lift="form.ajax"` and supply a function to run on the server when the form is submitted.

Here's an example of a form that will collect our name, and submit it via Ajax to the server:

```
<form data-lift="form.ajax">
  <div data-lift="EchoForm">
    <input type="text" name="name" placeholder="What's your name?">
    <input type="submit">
  </div>
</form>
```

```
<div id="result">Your name will be echoed here</div>
```

The following snippet will echo back the name via Ajax:

```
package code.snippet

import net.liftweb.util.Helpers._
import net.liftweb.http.SHtml.{text,ajaxSubmit}
import net.liftweb.http.js.JsCmd
import net.liftweb.http.js.JsCmds.SetHtml
import xml.Text

object EchoForm extends {

  def render = {

    var name = ""

    def process() : JsCmd = SetHtml("result", Text(name))

    "@name" #> text(name, s => name = s) &
    "type=submit" #> ajaxSubmit("Click Me", process)
  }
}
```

The `render` method is binding the name input field to a function that will assign whatever the user enters to the variable `name`. Note you'll more typically see `s => name = s` written in the shorter form of `name = _`.

When the button is pressed, the `process` function is called, which will return the value of the `name` back to the element in the HTML with an ID of `result`.

Discussion

The `data-lift="form.ajax"` part of this recipe ensures that Lift adds the Ajax processing mechanics to the form. This means the `<form>` element in the output will end up as something like:

```
<form id="F2203365740CJME2G" action="javascript://"
  onsubmit="liftAjax.lift_ajaxHandler(
    jQuery('#'+&amp;quot;F2203365740CJME2G&amp;quot;).serialize(),
    null, null, &amp;quot;javascript&amp;quot;);return false;">
  ...
</form>
```

In other words, when the form is asked to submit, Lift will serialise the form via Ajax. This means you don't necessarily need the submit button at all. In this example with a single text field, if you omit the submit button you can trigger serialisation by pressing *return*. This will trigger the `s => name = s` function which was bound in our regular

data-lift="EchoForm" snippet. In other words, the value name will be assigned even without a submit button.

Adding in a submit button gives us a way to perform actions once all the field's functions have been executed.

Notice that Lift's approach is to serialise the form to the server, execute the functions associated with the fields, execute the submit function (if any), then return a JavaScript result to the client. The default serialisation process is to use jQuery's serialization method on the form. This serializes fields except submit buttons and file uploads.

Submit Styling

The SHtml.ajaxSubmit function generates a <input type="submit"> element for the page. You may prefer to use a styled button for submit. For example, with Twitter Bootstrap, a button with an icon would require the following markup:

```
<button id="submit" class="btn btn-primary btn-large">
  <i class="icon-white icon-ok"></i> Submit
</button>
```

Pressing a <button> inside a form triggers the submit. However, if you bound that button with Shtml.ajaxSubmit the content, and therefore the styling, would be lost.

To fix this you can assign a function to a hidden field. This function will be called when the form is submitted just like any other field. The only part of our snippet that changes is the CSS selector binding:

```
import net.liftweb.http.SHtml.hidden

"@name" #> text(name, s => name = s) &
"button *+" #> hidden(process)
```

The *+ replacement rule means append a value to the child node of the button. This will include a hidden field in the form, something like…

```
<input type="hidden" name="F1120202962828SOIEC2" value="true">
```

…and when the form is submitted, the hidden field is submitted, and like any field, Lift will call the function associated with it: process in this case.

The effect is something like ajaxSubmit, but not exactly the same. In this instance we're appending a hidden field after the <button>, but you could place it anywhere on the form you find convenient. However, there's one complication: when is process called? Is it before the name has been assigned or after? That depends on the order the fields are rendered. That's to say, in your HTML template, placing the button before the text field (and therefore moving the hidden field's position in this example), the process function is called before the name has been set.

There are a couple of ways around that. Either, ensure your hidden fields used in this way appear late in your form, or make sure the function is called late with a `formGroup`:

```
import net.liftweb.http.SHtml.hidden
import net.liftweb.http.S

"@name" #> text(name, s => name = s) &
"button *+" #> S.formGroup(1000) { hidden(process) }
```

The `formGroup` addition manipulates the function identifier to ensure it sorts later, resulting in the function `process` being called later than fields in the default group (0).

Lift 2.6 and 3.0 may contain `ajaxOnSubmit`, which will give the reliability of `ajaxSubmit` and the flexibility of the hidden-field approach. If you want to try it in Lift 2.5, Antonio Salazar Cardozo has created a helper you can include in your project: *https://gist.github.com/Shadowfiend/5042131*.

See Also

Function order is discussed in the Lift Cool Tips wiki page: *http://www.assembla.com/spaces/liftweb/wiki/cool_tips*.

Form serialization is described at *http://api.jquery.com/serialize/*.

Recipe 5.9 describes Ajax file uploads.

3.3. Ajax JSON Form Processing

Problem

You want to process a form via Ajax, sending the data in JSON format.

Solution

Make use of Lift's *jlift.js* JavaScript library and `JsonHandler` class.

As an example we can create a "motto server" that will accept an institution name and the institution's motto and perform some action on these values. We're just going to echo the name and motto back to the client.

Consider this HTML, which is not in a form, but includes *jlift.js*:

```
<html>
<head>
  <title>JSON Form</title>
</head>
```

```
<body data-lift-content-id="main">

<div id="main" data-lift="surround?with=default;at=content">

  <h1>Json Form example</h1>

  <!-- Required for JSON forms processing -->
  <script src="/classpath/jlift.js" data-lift="tail"></script>

  <div data-lift="JsonForm" >

    <script id="jsonScript" data-lift="tail"></script>

    <div id="jsonForm">

      <label for="name">
        Institution
        <input id="name" type="text" name="name" value="Royal Society" />
      </label>

      <label for="motto">
        Motto
        <input id="motto" type="text" name="motto" value="Nullius in verba" />
      </label>

      <input type="submit" value="Send" />

    </div>

    <div id="result">
      Result will appear here.
    </div>

  </div>

</div>
</body>
</html>
```

This HTML presents the user with two fields, a name and a motto, wrapped in a div called `jsonForm`. There's also a placeholder for some results, and you'll notice a `jsonScript` placeholder for some JavaScript code. The `jsonForm` will be manipulated to ensure it is sent via Ajax, and the `jsonScript` will be replaced with Lift's code to perform the serialisation. This happens in the snippet code:

```
package code.snippet

import scala.xml.{Text, NodeSeq}

import net.liftweb.util.Helpers._
import net.liftweb.util.JsonCmd
import net.liftweb.http.SHtml.jsonForm
```

```
import net.liftweb.http.JsonHandler
import net.liftweb.http.js.JsCmd
import net.liftweb.http.js.JsCmds.{SetHtml, Script}

object JsonForm {

  def render =
    "#jsonForm" #> ((ns:NodeSeq) => jsonForm(MottoServer, ns)) &
    "#jsonScript" #> Script(MottoServer.jsCmd)

  object MottoServer extends JsonHandler {

    def apply(in: Any): JsCmd = in match {
      case JsonCmd("processForm", target, params: Map[String, String], all) =>
        val name = params.getOrElse("name", "No Name")
        val motto = params.getOrElse("motto", "No Motto")
        SetHtml("result",
          Text("The motto of %s is %s".format(name,motto)) )
    }
  }
}
```

Like many snippets, this Scala code contains a `render` method which binds to elements on the page. Specifically, `jsonForm` is being replaced with `SHtml.jsonForm`, which will take a `NodeSeq` (which are the input fields) turns it into a form that will submit the values as JSON. The submission will be to our `MottoServer` code.

The `jsonScript` element is bound to JavaScript that will perform the transmission and encoding of the values to the server.

If you click the "Send" button and observe the network traffic, you'll see the following sent to the server:

```
{
  "command": "processForm",
  "params": {"name":"Royal Society","motto":"Nullius in verba"}
}
```

This is the value of the `all` parameter in the `JsonCmd` being pattern matched against in `MottoServer.apply`. Lift has taken care of the plumbing to make this happen.

The result of the pattern match in the example is to pick out the to field values, and send back JavaScript to update the `results` div with: "The motto of the Royal Society is Nullius in verba".

Discussion

The `JsonHandler` class and the `SHtml.jsonForm` method are together performing a lot of work for us. The `jsonForm` method is arranging for form fields to be encoded as JSON

and sent, via Ajax, to our `MottoServer` as a `JsonCmd`. In fact, it's a `JsonCmd` with a default command name of "processForm".

Our `MottoServer` class is looking for (matching on) this `JsonCmd`, extracting the values of the form fields, and echoing these back to the client as a `JsCmd` that updates a div on the page.

The `MottoServer.jsCmd` part is generating the JavaScript required to deliver the form fields to the server. As we will see later, this is providing a general purpose function we can use to send different JSON values and commands to the server.

Notice also, from the network traffic, that the form fields sent are serialised with the names they are given on the page. There are no "F…" values sent which map to function calls on the server. A consequence of this is that any fields dynamically added to the page will also be serialised to the server, where they can be picked up in the `MottoServer`.

The script *jlift.js* is providing the plumbing to make much of this happen.

Before going on, convince yourself that we're generating JavaScript on the server-side (`MottoServer.jsCmd`), which is executed on the client side when the form is submitted, to deliver results to the server.

Additional Commands

In the above example we match on a `JsonCmd` with a command name of "processForm". You may be wondering what other command can can supplied, or what the meaning of the "target" value is.

To demonstrate how you can implement other commands, we can add two additional buttons. These buttons will just convert the motto to upper case or lower case. The server side `render` method changes as follows:

```
def render =
  "#jsonForm" #> ((ns:NodeSeq) => jsonForm(MottoServer, ns)) &
  "#jsonScript" #> Script(
    MottoServer.jsCmd &
    Function("changeCase", List("direction"),
      MottoServer.call("processCase", JsVar("direction"),
        JsRaw("$('#motto').val()"))
    )
  )
```

The `JsonForm` is unchanged. We still include `MottoServer.jsCmd`, and we still want to wrap the fields and submit them as before. What we've added is an extra JavaScript function called "changeCase", which takes one argument called "direction" and as a body calls the `MottoServer` with various parameters. When it is rendered on the page it would appear as something like this:

```
function changeCase(direction) {
  F299202CYGIL({'command': "processCase", 'target': direction,
    'params': $('#motto').val() });
}
```

The `F299202CYGIL` function (or similar name) is generated by Lift as part of `MottoServer.jsCmd`, and it is responsible for delivering data to the server. The data it is delivering, in this case, is a JSON structure consisting of a different command ("processCase"), a target of whatever the JavaScript value `direction` evaluates to, and a parameter which is the result of the jQuery expression for the value of the "#motto" form field.

When is the `changeCase` function called? That's up to us, and one very simple way to call the function would be by this addition to the HTML:

```
<button onclick="javascript:changeCase('upper')">Upper case the Motto</button>
<button onclick="javascript:changeCase('lower')">Lower case the Motto</button>
```

When either of these buttons are pressed, the result will be a JSON value sent to the server with the command of "processCase" and the "direction" and "params" set accordingly. All that is left is to modify our `MottoServer` to pick up this `JsonCmd` on the server:

```
object MottoServer extends JsonHandler {

  def apply(in: Any): JsCmd = in match {

    case JsonCmd("processForm", target, params: Map[String, String], all) =>
      val name = params.getOrElse("name", "No Name")
      val motto = params.getOrElse("motto", "No Motto")
      SetHtml("result",
        Text("The motto of %s is %s".format(name,motto)) )

    case JsonCmd("processCase", direction, motto: String, all) =>
      val update =
        if (direction == "upper") motto.toUpperCase
        else motto.toLowerCase
      SetValById("motto", update)

  }
}
```

The first `JsonCmd` is unchanged. The second matches on the parameters sent and results in updating the form fields with an upper- or lower-case version of the motto.

See Also

There are further examples of `JsonHandler` at *http://demo.liftweb.net/json_more*.

If you want to process JSON via REST, take a look at the examples at *http://demo.liftweb.net/stateless_json*.

Lift in Action, section 9.1.4 discusses "Using JSON forms with AJAX", as does section 10.4 of *Exploring Lift* at *http://exploring.liftweb.net*.

3.4. Use a Date Picker for Input

Problem

You want to provide a date picker to make it easy for users to supply dates to your forms.

Solution

Use a standard Lift `SHtml.text` input field and attach a JavaScript date picker to it. In this example we will use the jQuery UI date picker.

Our form will include an input field called `birthday` to be used as a date picker, and also the jQuery UI JavaScript and CSS:

```
<!DOCTYPE html>
<head>
  <meta content="text/html; charset=UTF-8" http-equiv="content-type" />
  <title>jQuery Date Picker</title>
</head>
<body data-lift-content-id="main">
<div id="main" data-lift="surround?with=default;at=content">

  <h3>When's your birthday?</h3>

  <link data-lift="head" type="text/css" rel="stylesheet"
          href="//cdnjs.cloudflare.com/ajax/libs/jqueryui/1.10.2/css/smoothness/
jquery-ui-1.10.2.custom.min.css">
  </link>

  <script data-lift="tail"
    src="//cdnjs.cloudflare.com/ajax/libs/jqueryui/1.10.2/jquery-ui.min.js">
  </script>

  <div data-lift="JqDatePicker?form=post">
    <input type="text" id="birthday">
    <input type="submit" value="Submit">
  </div>

</div>
</body>
</html>
```

This would normally produce a regular text input field, but we can change that by adding JavaScript to attach the date picker to the input field. You could do this in the template, but in this example we're enhancing the text field as part of the snippet code:

```
package code.snippet

import java.util.Date
import java.text.SimpleDateFormat

import net.liftweb.util.Helpers._
import net.liftweb.http.{S, SHtml}
import net.liftweb.http.js.JsCmds.Run
import net.liftweb.common.Loggable

class JqDatePicker extends Loggable {

  val dateFormat = new SimpleDateFormat("yyyy-MM-dd")

  val default = (dateFormat format now)

  def logDate(s: String) : Unit = {
    val date : Date = tryo(dateFormat parse s) getOrElse now
    logger.info("Birthday: "+date)
  }

  def render = {
    S.appendJs(enhance)
    "#birthday" #> SHtml.text(default, logDate)
  }

  val enhance =
    Run("$('#birthday').datepicker({dateFormat: 'yy-mm-dd'});")
}
```

Notice in `render` we are binding a regular `SHtml.text` field to the element with the ID of "birthday", but also appending JavaScript to the page. That JavaScript selects the birthday input field and attaches a configured date picker to it.

When the field is submitted, the `logDate` method is called with the value of the text field. We parse the text into a `java.util.Date` object. The `tryo` Lift helper will catch any `ParseException` and return a `Box[Date]` which we open, or default to the current date if a bad date is supplied.

Running this code and submitting the form will produce a log message something like this:

```
INFO  code.snippet.DatePicker - Birthday: Sun Apr 21 00:00:00 BST 2013
```

Discussion

The approach outlined in this recipe can be used with other date picker libraries. The key point is to configure the date picker to submit a date in a format you can parse when the value is submitted to the server. This is the "wire format" of the date, and does not

have to necessarily be the same format the user sees in the browser, depending on the browser or the JavaScript library being used.

HTML 5 Date Pickers

The HTML 5 specification includes support for a variety of date input types: `date` `time`, `datetime-local`, `date`, `month`, `time`, `week`. For example:

```
<input type="date" name="birthday" value="2013-04-21">
```

This type of input field will submit a date in *yyyy-mm-dd* format which our snippet would be able to process.

As more browsers implement these types it will become possible to depend on them. However, you can default to the HTML 5 browser-native date pickers today and fall back to a JavaScript library as required. The difference is show in Figure 3-1.

Figure 3-1. An input field with the jQuery UI date picker attached, compared to the browser-native date picker in Chrome.

To detect if the browser supports `type="date"` inputs, we can use the *Modernizr* library. This is an additional script in our template:

```
<script data-lift="tail"
  src="//cdnjs.cloudflare.com/ajax/libs/modernizr/2.6.2/modernizr.min.js">
</script>
```

We will use this in our snippet. In fact, there are two changes we need to make to the snippet:

1. add the `type="date"` attribute to the input field; and
2. modify the JavaScript to only attach the jQuery UI date picker in browsers that don't support the `type="date"` input.

In code that becomes:

```
def render = {
  S.appendJs(enhance)
  "#birthday" #> SHtml.text(default, logDate, ("type"->"date"))
}

val enhance = Run(
  """
    |if (!Modernizr.inputtypes.date) {
    | $('input[type=date]').datepicker({dateFormat: 'yy-mm-dd'});
```

```
    |}
  """.stripMargin)
```

The `"type" -> "date"` parameter on `SHtml.text` is setting the attribute "type" to the value "date" on the resuting `<input>` field.

When this snippet runs, and the page is rendered, the jQuery UI date picker will be attached to input fields of `type="date"` only if the browser doesn't support that type already.

See Also

Dive into HTML5, at http://diveintohtml5.info , describes how to detect browser features.

jQuery UI's date picker has many configuration options, which are described at *http://api.jqueryui.com/datepicker/*.

The HTML5 date input types submit dates in RFC3339 format. These are specified at *http://tools.ietf.org/html/rfc3339*

3.5. Making Suggestions with Autocomplete

Problem

You want to provide an autocomplete widget, to give users suggestions as they type into a text field.

Solution

Use a JavaScript autocomplete widget, for example the jQuery UI autocomplete via the `AutoComplete` class from the Lift Widgets module.

Start by adding the Lift widgets module to your *build.sbt*:

```
libraryDependencies += "net.liftmodules" %% "widgets_2.5" % "1.3"
```

To enable the autocomplete widget, initialize it in *Boot.scala*:

```
import net.liftmodules.widgets.autocomplete.AutoComplete
AutoComplete.init()
```

We can create a regular form snippet…

```
<form data-lift="ProgrammingLanguages?form=post">
  <input id="autocomplete">
  <input type="submit">
</form>
```

…and connect the `AutoComplete` class to the element with the ID of "autocomplete":

```
package code.snippet

import net.liftweb.util.Helpers._
import net.liftweb.common.Loggable

import net.liftmodules.widgets.autocomplete.AutoComplete

class ProgrammingLanguages extends Loggable {

  val languages = List(
    "C", "C++", "Clojure", "CoffeeScript",
    "Java", "JavaScript",
    "POP-11", "Prolog", "Python", "Processing",
    "Scala", "Scheme", "Smalltalk", "SuperCollider"
  )

  def render = {

    val default = ""

    def suggest(value: String, limit: Int) =
      languages.filter(_.toLowerCase.startsWith(value))

    def submit(value: String) : Unit =
      logger.info("Value submitted: "+value)

    "#autocomplete" #> AutoComplete(default, suggest, submit)
  }

}
```

The last line of this snippet shows the binding of the `AutoComplete` class, which takes:

- a default value to show;
- a function which will produce suggestions from the text value entered — the result is a `Seq[String]` suggestions; and
- a function to call when the form is submitted.

Running this code renders as shown in Figure 3-2.

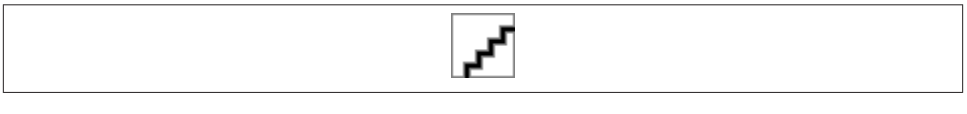


Figure 3-2. The rendering of the `ProgrammingLanguages` snippet.

When the form is submitted, the `submit` function will be passed the selected value. The `submit` function is simply logging this value:

```
INFO  code.snippet.ProgrammingLanguages - Value submitted: Scala
```

Discussion

The autocomplete widget uses jQuery autocomplete. This can be seen by examining the `NodeSeq` produced by the `AutoComplete.apply` method:

```
 <span>
  <head>
  <link type="text/css" rel="stylesheet"
    href="/classpath/autocomplete/jquery.autocomplete.css">
  </link>
  <script type="text/javascript"
    src="/classpath/autocomplete/jquery.autocomplete.js">
  </script>
  <script type="text/javascript">
// <![CDATA[
  jQuery(document).ready(function(){
    var data = "/ajax_request?F846528841915S2RBI0=foo";
    jQuery("#F846528841916S3QZ0V").
      autocomplete(data, {minChars:0,matchContains:true}).
      result(function(event, dt, formatted) {
       jQuery("#F846528841917CF4ZGL").val(formatted);
      }
     );
  });;
// ]]>
</script>
  </head>
  <input type="text" value="" id="F846528841916S3QZ0V"></input>
  <input name="F846528841917CF4ZGL" type="hidden" value=""
   id="F846528841917CF4ZGL"></input>
</span>
```

This chunk of markup is generated from the `AutoComplete(default, suggest, submit)` call. What's happening here is that the jQuery UI autocomplete JavaScript and CSS, which is bundled with the Lift Widgets module, is being included on the page. Recall from Recipe 2.9 that Lift will merge the `<head>` part of this markup into the `<head>` of the final HTML page.

When the page loads, the jQuery UI `autocomplete` function is bound to the input field, and configured with a URL which will deliver the user's input to our `suggest` function. All `suggest` need to do is return a `Seq[String]` of values for the jQuery autocomplete code to display to the user.

Submitting New Values

One peculiarity of the `AutoComplete` widget is that if you type in a new value — one not suggested — and press submit, it is not sent to the server. That is, you need to click on

one of the suggestions to select it. If that's not the behaviour you want, you can adjust it.

Inside the `render` method we can modify the behaviour by adding JavaScript to the page:

```
import net.liftweb.http.S
import net.liftweb.http.js.JsCmds.Run
S.appendJs(Run(
"""
  |$('#autocomplete input[type=text]').bind('blur',function() {
  |  $(this).next().val($(this).val());
  |});
""".stripMargin))
```

With this in place, when the input field loses focus, for example when the submit button is pressed, the value of the input field is stored as the value to be sent to the server.

Alternative Auto Complete JavaScript

Looking at the way the widget module builds autocomplete functionality may give you an insight into how you can incorporate other JavaScript autocomplete libraries into your Lift application. The idea is to include the JavaScript library, connect it to an element on the page, and then arrange for the server to be called to generate suggestions. Of course, if you only have a few items for the user to pick from, you could just include those items on the page, rather than making a round trip to the server.

As an example of server-generated suggestions, we can look at the *Typeahead* component that is included in Twitter Bootstrap.

To incorporate Typeahead, the template needs to change to include the library and mark the input field in the way Typeahead expects:

```
<link data-lift="head" rel="stylesheet"
          href="//netdna.bootstrapcdn.com/twitter-bootstrap/2.3.1/css/bootstrap-
combined.min.css">

<script data-lift="tail"
  src="//netdna.bootstrapcdn.com/twitter-bootstrap/2.3.1/js/bootstrap.min.js">
</script>

<form data-lift="ProgrammingLanguagesTypeAhead">
  <script id="js"></script>
  <input id="autocomplete" type="text"
    data-provide="typeahead" autocomplete="off">
  <input type="submit">
</form>
```

We've included a placeholder with an ID of "js" for the JavaScript that will call back to the server. We'll get to that in a moment.

The way Typeahead works is that we attach it to our input field, and tell it to call a JavaScript function when it need to make suggestions. That JavaScript function is going to be called `askServer`, and it is given two arguments: the input the user has typed so far (`query`), and a JavaScript function to call when the suggestions are available (`call back`). The Lift snippet needs to use the `query` value and then call the JavaScript `call back` function with whatever suggestions are made.

A snippet to implement this would be as follows:

```
package code.snippet

import net.liftweb.util.Helpers._
import net.liftweb.common.{Full, Empty, Loggable}

import net.liftweb.http._
import net.liftweb.http.js.JsCmds._
import net.liftweb.http.js.JsCmds.Run
import net.liftweb.http.js.JE.JsVar
import net.liftweb.json.JsonAST._
import net.liftweb.json.DefaultFormats

class ProgrammingLanguagesTypeAhead extends Loggable {

  val languages = List(
    "C", "C++", "Clojure", "CoffeeScript",
    "Java", "JavaScript",
    "POP-11", "Prolog", "Python", "Processing",
    "Scala", "Scheme", "Smalltalk", "SuperCollider"
  )

  def render = {

    implicit val formats = DefaultFormats

    def suggest(value: JValue) : JValue = {
      logger.info("Making suggestion for: "+value)

      val matches = for {
        q <- value.extractOpt[String].map(_.toLowerCase).toList
        lang <- languages.filter(_.toLowerCase startsWith q)
      } yield JString(lang)

      JArray(matches)
    }

    val callbackContext = new JsonContext(Full("callback"),Empty)

    val runSuggestion =
      SHtml.jsonCall(JsVar("query"), callbackContext, suggest _ )

    S.appendJs(Run(
```

```
        """
          |$('#autocomplete').typeahead({
          |  source: askServer
          |});
        """.stripMargin))

      "#js *" #> Function("askServer", "query" :: "callback" :: Nil,
                  Run(runSuggestion.toJsCmd)) &
      "#autocomplete" #> SHtml.text("", s => logger.info("Submitted: "+s))

    }
  }
```

Working from the bottom of the snippet, we bind a regular Lift `SHtml.text` input to the autocomplete field. This will receive the selected value when the form is submitted. We also bind the JavaScript placeholder to a JavaScript function definition called `askServer`. This is the function that Typeahead will call when it wants suggestions.

The JavaScript function we're defining takes two arguments: the `query` and `callback`. The body of `askServer` causes it to run something called `runSuggestion`, which is a `jsonCall` to the server, with the value of the `query`.

The suggestions are made by the `suggest` function which expects to be able to find a `String` in the passed-in JSON value. It uses this value to find matches in the list of `languages`. These are returned as a `JArray` of `JString`, which is treated as JSON data back on the client.

What does the client do with the data? It calls the `callback` function with the suggestions, which results in the display updating. We specify that it's `callback` via `JsonCon text`, which is a class that lets us specify a custom function to call on success of the request to the server.

It may help to understand this by looking at the JavaScript generated in the HTML page for `askServer`:

```
<script id="js">
function askServer(query, callback) {
  liftAjax.lift_ajaxHandler('F2689448437l7UZB5J0=' +
    encodeURIComponent(JSON.stringify(query)), callback, null, "json")
}
</script>
```

As the user types into the text field, Typeahead calls `askServer` with the input supplied. Lift's Ajax support arranges for that value, `query`, to be serialised to our `suggest` function on the server, and for the results to be passed to `callback`. At that point, Typeahead takes over again, and displays the suggestions.

Typing "Scala" to the text field and pressing submit will produce a sequence like this on the server:

```
INFO  c.s.ProgrammingLanguagesTypeAhead - Making suggestion for: JString(Sc)
INFO  c.s.ProgrammingLanguagesTypeAhead - Making suggestion for: JString(Sca)
INFO  c.s.ProgrammingLanguagesTypeAhead - Making suggestion for: JString(Sca)
INFO  c.s.ProgrammingLanguagesTypeAhead - Making suggestion for: JString(Scal)
INFO  c.s.ProgrammingLanguagesTypeAhead - Making suggestion for: JString(Scala)
INFO  c.s.ProgrammingLanguagesTypeAhead - Submitted: Scala
```

See Also

Recipe 5.1 describes `jsonCall`.

The behaviour of the widget module with respect to new values is described in a ticket on the module's GitHub page: *https://github.com/liftmodules/widgets/issues/4*. Enhancing modules is one route to get involved with Lift, and Chapter 11 describes other ways to contribute.

The jQuery UI Autocomplete widget is documented at: http://jqueryui.com/autocomplete/ . The version included with the Lift widgets module is version 1.0.2.

You can find documentation for Twitter Bootstrap Typeahead at *http://twitter.github.io/bootstrap/javascript.html#typeahead*.

3.6. Conditionally Disable a Checkbox

Problem

You want to add the `disabled` attribute to a `SHtml.checkbox` based on a conditional check.

Solution

Create a CSS selector transform to add the disabled attribute, and apply it to your checkbox transform.

For example, suppose you have a simple checkbox…

```
class Likes {
  var likeTurtles = false
  def render =
    "#like" #> SHtml.checkbox(likeTurtles, likeTurtles = _ )
}
```

…and a corresponding template:

```
<!DOCTYPE html>
<head>
  <meta content="text/html; charset=UTF-8" http-equiv="content-type" />
  <title>Disable Checkboxes</title>
</head>
<body data-lift-content-id="main">
```

```
<div id="main" data-lift="surround?with=default;at=content">

  <div>Select the things you like:</div>

  <form data-lift="Likes">
    <label for="like">Do you like turtles?</label>
    <input id="like" type="checkbox">
  </form>

</div>
</body>
</html>
```

Further suppose you want to disable it roughly 50% of the time. We could do that by adjusting the `NodeSeq` generated from `SHtml.checkbox`:

```
package code.snippet

import net.liftweb.util.Helpers._
import net.liftweb.util.PassThru
import net.liftweb.http.SHtml

class Likes {
  var likesTurtles = false

  def disable =
    if (math.random > 0.5d) "* [disabled]" #> "disabled"
    else PassThru

  def render =
    "#like" #> disable( SHtml.checkbox(likesTurtles, likesTurtles = _) )
}
```

When the checkbox is rendered, it will be disabled roughly half the time.

Discussion

The `disable` method returns a `NodeSeq => NodeSeq` function, meaning when we apply it we need to give it a `NodeSeq`, which is exactly what `SHtml.checkbox` provides.

The `[disabled]` part of the CSS selector is selecting the disabled attribute and replacing it with the value on the right of the `#>`, which is "disabled" in this example.

What this combination means is that half the time the disabled attribute will be set on the checkbox, and half the time the checkbox `NodeSeq` will be left untouched because `PassThru` does not change the `NodeSeq`.

See Also

Recipe 2.6 describes the `PassThru` function.

3.7. Use a Select Box with Multiple Options

Problem

You want to show the user a number of options in a select box, and allow them to select multiple values.

Solution

Use `SHtml.multiSelect(options, default, selection)`. Here's an example where a user can select up to three options:

```
package code.snippet

import net.liftweb.util.Helpers._
import net.liftweb.http.SHtml
import net.liftweb.common.Loggable

class MultiSelect extends Loggable {

  case class Item(id: String, name: String)

  val inventory =
    Item("a", "Coffee") ::
    Item("b", "Milk") ::
    Item("c", "Sugar") :: Nil

  val options : List[(String,String)] =
    inventory.map(i => (i.id -> i.name))

  val default = inventory.head.id :: Nil

  def render = {

    def selection(ids: List[String]) : Unit = {
      logger.info("Selected: "+ids)
    }

    "#opts *" #>
      SHtml.multiSelect(options, default, selection)
  }
}
```

In this example the user is being presented with a list of three items, with the first one selected, as shown in Figure 3-3. When the form is submitted the `selection` function is called, with a list of the selected option values.

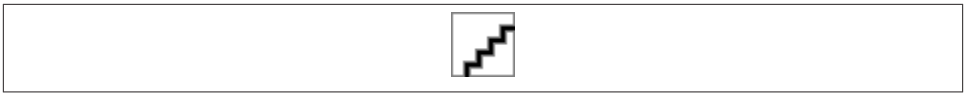


Figure 3-3. Selecting from multiple options.

The template to go with the snippet could be:

```
<div data-lift="MultiSelect?form=post">
  <p>What can I get you?</p>
  <div id="opts">options go here</div>
   <input type="submit" value="Place Order">
</div>
```

This will render as something like:

```
<form action="/" method="post"><div>
  <p>What can I get you?</p>
  <div id="opts">
   <select name="F25749422319ALP1BW" multiple="true">
     <option value="a" selected="selected">Coffee</option>
     <option value="b">Milk</option>
     <option value="c">Sugar</option>
   </select>
  </div>
  <input value="Place Order" type="submit">
</form>
```

Discussion

Recall that an HTML select consists of a set of options, each of which has a value and a name. To reflect this, the above examples takes our `inventory` of objects and turns it into a list of (value,name) string pairs, called `options`.

The function given to `SHtml.multiSelect` will receive the values (IDs), not the names, of the options. That is, if you ran the above code, and selected "Coffee" and "Milk" the function would see `List("a", "b")`.

Selected No Options

Be aware that if no options are selected, your handling function is not called. This is described in issue 1139.

One way to work around this is to add a hidden function to reset the list. For example, we could modify the above code to be a stateful snippet and remember the values we selected:

```
package code.snippet

import net.liftweb.util.Helpers._
import net.liftweb.http.{StatefulSnippet, SHtml}
```

```
import net.liftweb.common.Loggable

class MultiSelectStateful extends StatefulSnippet with Loggable {

  def dispatch = {
    case _ => render
  }

  case class Item(id: String, name: String)

  val inventory =
    Item("a", "Coffee") ::
    Item("b", "Milk") ::
    Item("c", "Sugar") :: Nil

  val options : List[(String,String)] =
    inventory.map(i => (i.id -> i.name))

  var current = inventory.head.id :: Nil

  def render = {

    def logSelected() =
      logger.info("Values selected: "+current)

    "#opts *" #> (
      SHtml.hidden( () => current = Nil) ++
      SHtml.multiSelect(options, current, current = _)
    ) &
    "type=submit" #> SHtml.onSubmitUnit(logSelected)

  }

}
```

The template is unchanged, and the snippet has been modified to introduce a `current` value and a hidden function to reset the value. We've bound the submit button to simply log the selected values when the form is submitted.

Each time the form is submited the `current` list of IDs is set to whatever you have selected in the browser. But note that we have started with a hidden function that resets `current` to the empty list. This means that if the receiving function in `multiSelect` is never called, because nothing is selected, that the value store in current would reflect this and be `Nil`.

That may be useful, depending on what behaviour you need in your application.

Type-safe Options

If you don't want to work in terms of `String` values for an option, you can use `multiSelectObj`. In this variation the list of options still provides a text name, but the value is in terms of a class. Likewise, the list of default values will be a list of class instances.

The only changes to the code are to produce a `List[(Item,String)]` for the options, and use an `Item` as a default:

```
val options : List[(Item,String)] =
  inventory.map(i => (i -> i.name))

val default = inventory.head :: Nil
```

The call to generate the multi-select from this data is similar, but note our `selection` function now receives a list of `Item`:

```
def render = {

  def selection(items: List[Item]) : Unit = {
    logger.info("Selected: "+items)
  }

  "#opts *" #>
    SHtml.multiSelectObj(options, default, selection)
  }
```

Enumerations

You can use `multiSelectObj` with enumerations:

```
package code.snippet

import net.liftweb.util.Helpers._
import net.liftweb.http.SHtml
import net.liftweb.common.Loggable

class MultiSelectEnum extends Loggable {

  object Item extends Enumeration {
    type Item = Value
    val Coffee, Milk, Sugar = Value
  }

  import Item._

  val options : List[(Item,String)] =
    Item.values.toList.map(i => (i -> i.toString))

  val default = Item.Coffee :: Nil

  def render = {
```

```
      def selection(items: List[Item]) : Unit = {
        logger.info("Selected: "+items)
      }

      "#opts *" #>
        SHtml.multiSelectObj(options, default, selection)
    }

  }
```

The enumeration version works in the same way as the type-safe version.

See Also

The "Submit Styling" discussion section in Recipe 3.2 discusses the use of hidden fields as function calls.

Recipe 5.2 describes how to trigger a server-side action when a selection changes in the browser.

Chapter 6 of *Exploring Lift*, "Forms in Lift", discusses multiselect and other types of form elements at *http://exploring.liftweb.net/*.

The issue relating to no options being selected is on Github: *https://github.com/lift/framework/issues/1139*

3.8. File Upload

Problem

You want a snippet to allow users to upload a file to your Lift application.

Solution

Use a `FileParamHolder` in your snippet, and extract file information from it when the form is submitted.

Start with a form which is marked as "multipart=true":

```
<html>
<head>
  <title>File Upload</title>
     <script  id="jquery"  src="/classpath/jquery.js"  type="text/javascript"></
script>
  <script id="json" src="/classpath/json.js" type="text/javascript"></script>
</head>
<body>
<form data-lift="FileUploadSnippet?form=post;multipart=true">
   <label for="file">
     Select a file: <input id="file"></input>
```

```
    </label>
    <input type="submit" value="Submit"></input>
</form>
</body>
</html>
```

We bind the file input to `SHtml.fileUpload` and the submit button to a function to process the uploaded file:

```
package code.snippet

import net.liftweb.util.Helpers._
import net.liftweb.http.SHtml._
import net.liftweb.http.FileParamHolder
import net.liftweb.common.{Loggable, Full, Empty, Box}


class FileUploadSnippet extends Loggable {

  def render = {

    var upload : Box[FileParamHolder] = Empty

    def processForm() = upload match {
      case Full(FileParamHolder(_, mimeType, fileName, file)) =>
        logger.info("%s of type %s is %d bytes long" format (
         fileName, mimeType, file.length) )

      case _ => logger.warn("No file?")
    }

    "#file" #> fileUpload(f => upload = Full(f)) &
      "type=submit" #> onSubmitUnit(processForm)
  }
}
```

The `fileUpload` binding ensures that the file is assigned to the `upload` variable. This allows us to access the `Array[Byte]` of the file in the `processForm` method when the form is submitted.

Discussion

HTTP includes an encoding type of "multipart/form-data" which was introduced to support binary data uploads. The `?form=post;multipart=true` parameters in the template mark the form with this encoding, and the HTML generated will look like this:

```
<form enctype="multipart/form-data" method="post" action="/fileupload">
```

When the browser submits the form, Lift detects the "multipart/form-data" encoding and extracts any files from the request. These are available as `uploadedFiles` on a `Req` object, for example:

```
val files : List[FileParamHolder] = S.request.map(_.uploadedFiles) openOr Nil
```

However, as we're dealing with a form with a single upload field it's easier to use `SHtml.fileUpload` to bind the input to our `upload` variable. Lift arranges for the function `f => upload = Full(f)` to be called when a file is selected and uploaded via this field. If the file is zero length, the function is not called.

The default behaviour for Lift is to read the file into memory and present it as a `FileParamHolder`. In this recipe we're pattern matching on the fields of the `FileParamHolder` and simply printing out what we know about the file. We're ignoring the first parameter which will be Lift's generated name for the field, but capturing the mime type, original filename and the raw data that was in the file.

You probably don't want to use this method for very large files. In fact, `LiftRules` provides a number of size restrictions which you can control:

- `LiftRules.maxMimeFileSize` — the maximum size of any single file uploaded (7MB by default).
- `LiftRules.maxMimeSize` — the maximum size of the multi-part upload in total (8MB by default)

Why two settings? Because when the form is submitted, there may be a number of fields on the form. For example, in the recipe the value of the submit button is send as one of the parts, and the file is sent as another. Hence, you might want to limit file size, but allow for some field values, or multiple files, to be submitted.

If you hit the size limit an exception will be thrown from the underlying file upload library. You can catch the exception, as described in Recipe 6.5:

```
LiftRules.exceptionHandler.prepend {
  case (_, _, x : FileUploadIOException) =>
    ResponseWithReason(BadResponse(), "Unable to process file. Too large?")
}
```

Be aware that the container (Jetty, Tomcat) or any web server (Apache, NGINX) may also have limits on file upload sizes.

Uploading a file into memory may be fine for some situations, but you may want to upload larger items to disk and then processes them in Lift as a stream. Lift supports this via the following setting:

```
LiftRules.handleMimeFile = OnDiskFileParamHolder.apply
```

The `handleMimeFile` variable expects to be given a function that takes a field name, mime type, filename and `InputStream` and returns a `FileParamHolder`. The default implementation of this is the `InMemFileParamHolder`, but changing to `OnDiskFileParamHolder` means Lift will write the file to disk first. You can of course implement your own handler in addition to using `OnDiskFileParamHolder` or `InMemFileParamHolder`.

With `OnDiskFileParamHolder`, the file will be written to a temporary location (`System.getProperty("java.io.tmpdir")`) but it's up to you to remove it when you're done with the file. For example, our snippet could change to:

```
def processForm() = upload match {

  case Full(content : OnDiskFileParamHolder) =>
    logger.info("File: "+content.localFile.getAbsolutePath)
    val in: InputStream = content.fileStream
    // ...do something with the stream here...
    val wasDeleted_? = content.localFile.delete()

  case _ => logger.warn("No file?")
}
```

Be aware that `OnDiskFileParamHolder` implements `FileParamHolder` so would match the original `FileParamHolder` pattern used in the recipe. However, if you access the `file` field of `OnDiskFileParamHolder`, you'll bring the file into memory, which would defeat the point of storing it on disk to process it as a stream.

If you want to monitor the progress of the upload on the server side, you can. There's a hook in `LiftRules` which is called as the upload is running:

```
def progressPrinter(bytesRead: Long, contentLength: Long, fieldIndex: Int) {
  println("Read %d of %d for %d" format (bytesRead, contentLength, fieldIndex))
}

LiftRules.progressListener = progressPrinter
```

This is the progress of the whole multi-part upload, not just the file being uploaded. In particular, the `contentLength` may not be known (in which case it will be `-1`), but if it is known it is the size of the complete multi-part upload. In the example in this recipe that would include the size of the file, but also the submit button value. This also explains the `fieldIndex`, which is a counter as to which part is being processed. It will take on the values of 0 and 1 for the two parts in this example.

See Also

The HTTP file upload mechanics are described in RFC 1867, *Form-based File Upload in HTML*: *http://tools.ietf.org/html/rfc1867*

Recipe 4.5 discusses file upload in the context of a REST service.

See Recipe 5.9 for an example of an Ajax file upload through integration with a JavaScript library, providing progress indicators and drag-and-drop support.

CHAPTER 4
REST

This chapter looks at recipes around REST web services, via Lift's `RestHelper` trait. For an introduction, take a look at the Lift wiki page at *https://www.assembla.com/spaces/liftweb/wiki/REST_Web_Services* and chapter 5 of *Simply Lift* at *http://simply.liftweb.net*.

The sample code from this chapter is at: *https://github.com/LiftCookbook/cookbook_rest*.

4.1. DRY URLs

Problem

You found yourself repeating parts of URL paths in your `RestHelper`, and you Don't want to Repeat Yourself.

Solution

Use `prefix` in your `RestHelper`:

```
package code.rest

import net.liftweb.http.rest.RestHelper
import net.liftweb.http.LiftRules

object IssuesService extends RestHelper {

  def init() : Unit = {
    LiftRules.statelessDispatch.append(IssuesService)
  }

  serve("issues" / "by-state" prefix {
    case "open" :: Nil XmlGet _ => <p>None open</p>
```

```
      case "closed" :: Nil XmlGet _ => <p>None closed</p>
      case "closed" :: Nil XmlDelete _ => <p>All deleted</p>
    })

}
```

This service responds to URLs of */issues/by-state/open* and */issues/by-state/closed* and we have factored out the common part as a `prefix`.

Wire this into *Boot.scala* with:

```
import code.rest.IssuesService
IssuesService.init()
```

We can test the service with cURL:

```
$ curl -H 'Content-Type: application/xml'
    http://localhost:8080/issues/by-state/open
<?xml version="1.0" encoding="UTF-8"?>
<p>None open</p>

$ curl -X DELETE -H 'Content-Type: application/xml'
    http://localhost:8080/issues/by-state/closed
<?xml version="1.0" encoding="UTF-8"?>
<p>All deleted</p>
```

Discussion

You can have many `serve` blocks in your `RestHelper`, which helps give your REST service structure.

In this example we've arbitrarily decided to return XML and to match on an XML request using `XmlGet` and `XmlDelete`. The test for an XML request requires a content-type of *text/xml* or *application/xml*, a request for a path that ends with *.xml*. This is why the cURL request includes a header with with the `-H` flag. If we hadn't included that, the request would not match any of our patterns, and the result would be a 404 response.

See Also

Recipe 4.6 gives an example of accepting and returning JSON.

4.2. Missing File Suffix

Problem

Your `RestHelper` expects a filename as part of the URL, but the suffix (extension) is missing, and you need it.

Solution

Access `req.path.suffix` to recover the suffix.

For example, when processing */download/123.png* you want to be able reconstruct *123.png*:

```
package code.rest

import net.liftweb.http.rest.RestHelper
import net.liftweb.http.LiftRules
import xml.Text

object Reunite extends RestHelper  {

  private def reunite(name: String, suffix: String) =
    if (suffix.isEmpty) name else name+"."+suffix

  serve {
    case "download" :: file :: Nil Get req =>
      Text("You requested "+reunite(file, req.path.suffix))
  }

  def init() : Unit = {
    LiftRules.statelessDispatch.append(Reunite)
  }

}
```

We are matching on *download* but rather than using the `file` value directly, we pass it through the `reunite` function first to attach the suffix back on (if any).

Requesting this URL with a command like cURL will show you the filename as expected:

```
$ curl http://127.0.0.1:8080/download/123.png
<?xml version="1.0" encoding="UTF-8"?>
You requested 123.png
```

Discussion

When Lift parses a request it splits the request into constituent parts (e.g., turning the path into a `List[String]`). This includes a separation of *some* suffixes. This is good for pattern matching when you want to change behaviour based on the suffix, but a hindrance in this particular situation.

Only those suffixes defined in `LiftRules.explicitlyParsedSuffixes` are split from the filename. This includes many of the common file suffixes (such as "png", "atom", "json") and also some you may not be so familiar with, such as "com".

Note that if the suffix is not in `explicitlyParsedSuffixes`, the suffix will be an empty String and the `name` (in the above example) will be the file name with the suffix still attached.

Depending on your needs, you could alternatively add a guard condition to check for the file suffix:

```
case "download" :: file :: Nil Get req if req.path.suffix == "png" =>
  Text("You requested PNG file called "+file)
```

Or rather than simply attaching the suffix back on, you could take the opportunity do some computation to decide what to send back. For example, if the client supports the WebP image format, you might prefer to send that:

```
package code.rest

import net.liftweb.http.rest.RestHelper
import net.liftweb.http.LiftRules
import xml.Text

object Reunite extends RestHelper  {

  def init() : Unit = {
    LiftRules.statelessDispatch.append(Reunite)
  }

  serve {
    case "negotiate" :: file :: Nil Get req =>
      val toSend =
        if (req.header("Accept").exists(_ == "image/webp")) file+".webp"
        else file+".png"

      Text("You requested "+file+", would send "+toSend)
  }

}
```

Calling this service would check the HTTP Accept header before deciding what resource to send:

```
$ curl http://localhost:8080/negotiate/123
<?xml version="1.0" encoding="UTF-8"?>
You requested 123, would send 123.png

$ curl http://localhost:8080/negotiate/123 -H "Accept: image/webp"
<?xml version="1.0" encoding="UTF-8"?>
You requested 123, would send 123.webp
```

See Also

Recipe 4.3 shows how to remove items from `explicitlyParsedSuffixes`.

The source for `HttpHelpers.scala` contains the `explicitlyParsedSuffixes` list, which is the default list of suffixes that Lift parses from a URL: https://github.com/lift/framework/blob/master/core/util/src/main/scala/net/liftweb/util/HttpHelpers.scala .

4.3. Missing .com from Email Addresses

When submitting an email address to a REST service, a domain ending ".com" is stripped before your REST service can handle the request.

Solution

Modify `LiftRules.explicitlyParsedSuffixes` so that Lift doesn't change URLs that end with ".com".

In *Boot.scala*:

```
import net.liftweb.util.Helpers
LiftRules.explicitlyParsedSuffixes = Helpers.knownSuffixes &~ (Set("com"))
```

Discussion

By default Lift will strip off file suffixes from URLs to make it easy to match on suffixes. An example would be needing to match on all requests ending in ".xml" or ".pdf". However, ".com" is also registered as one of those suffixes, but this is inconvenient if you have URLs that end with email addresses.

Note that this doesn't impact email addresses in the middle of URLs. For example, consider the following REST service:

```
package code.rest

import net.liftweb.http.rest.RestHelper
import net.liftweb.http.LiftRules
import xml.Text

object Suffix extends RestHelper {

  def init() : Unit = {
    LiftRules.statelessDispatch.append(Suffix)
  }

  serve {
    case "email" :: e :: "send" :: Nil Get req =>
      Text("In middle: "+e)

    case "email" :: e :: Nil Get req =>
      Text("At end: "+e)
  }
```

```
}
```

With this service `init` method called in *Boot.scala* we could then make requests and observe the issue:

```
$ curl http://localhost:8080/email/you@example.com/send
<?xml version="1.0" encoding="UTF-8"?>
In middle: you@example.com

$ curl http://localhost:8080/email/you@example.com
<?xml version="1.0" encoding="UTF-8"?>
At end: you@example
```

The ".com" is being treated as a file suffix, which is why the solution of removing it from the list of suffixes will resolve this problem.

Note that because other top-level domains, such as ".uk", ".nl", ".gov", are not in `explicitlyParsedSuffixes`, those email addresses are left untouched.

See Also

Recipe 4.2 describes the suffix processing in more detail.

4.4. Failing to Match on a File Suffix

Problem

You're trying to match on a file suffix (extension), but your match is failing.

Solution

Ensure the suffix you're matching on is included in `LiftRules.explicitlyParsedSuffixes`.

As an example, perhaps you want to match anything ending in *.csv* at your */reports/* URL:

```
case Req("reports" :: name :: Nil, "csv", GetRequest) =>
  Text("Here's your CSV report for "+name)
```

You're expecting */reports/foo.csv* to produce "Here's your CSV report for foo", but you get a 404.

To resolve this, include "csv" as a file suffix that Lift knows to split from URLs. In *Boot.scala* call:

```
LiftRules.explicitlyParsedSuffixes += "csv"
```

The pattern will now match.

Discussion

Without adding "csv" to the `explicitlyParsedSuffixes`, the example URL would match with:

```
case Req("reports" :: name :: Nil, "", GetRequest) =>
  Text("Here's your CSV report for "+name)
```

Here we're matching on no suffix (""). In this case `name` would be set to "foo.csv". This is because Lift separates file suffixes from the end of URLs only for file suffixes that are registered with `explicitlyParsedSuffixes`. Because "csv" is not one of the default registered suffixes, "foo.csv" is not split. That's why "csv" in the suffix position of `Req` pattern match won't match the request, but an empty string in that position will.

See Also

Recipe 4.2 explains more about the suffix removal in Lift.

4.5. Accept binary data in a REST service

Problem

You want to accept an image upload, or other binary data, in your RESTful service.

Solution

Access the request body in your REST helper:

```
package code.rest

import net.liftweb.http.rest.RestHelper
import net.liftweb.http.LiftRules

object Upload extends RestHelper {

  def init() : Unit = {
    LiftRules.statelessDispatch.append(Upload)
  }

  serve {
    case "upload" :: Nil Post req =>
      for {
        bodyBytes <- req.body
      } yield <info>Received {bodyBytes.length} bytes</info>
  }

}
```

Wire this into your application in *Boot.scala*:

```
import code.rest.Upload
Upload.init()
```

Test this service using a tool like cURL:

```
$ curl -X POST --data-binary "@dog.jpg" -H 'Content-Type: image/jpg'
    http://127.0.0.1:8080/upload
<?xml version="1.0" encoding="UTF-8"?>
<info>Received 1365418 bytes</info>
```

Discussion

In the above example the binary data is accessed via the `req.body`, which returns a `Box[Array[Byte]]`. We turn this into a `Box[Elem]` to send back to the client. Implicits in `RestHelper` turn this into an `XmlResponse` for Lift to handle.

Note that web containers, such as Jetty and Tomcat, may place limits on the size of an upload. You will recognise this situation by an error such as "java.lang.IllegalStateException: Form too large705784>200000". Check with documentation for the container for changing these limits.

To restrict the type of image you accept, you could add a *guard condition* to the match, but you may find you have more readable code by moving the logic into an `unapply` method on an object. For example, to restrict an upload to just a JPEG you could say:

```
serve {
  case "jpg" :: Nil Post JPeg(req) =>
    for {
      bodyBytes <- req.body
    } yield <info>Jpeg Received {bodyBytes.length} bytes</info>
  }

object JPeg {
  def unapply(req: Req): Option[Req] =
    req.contentType.filter(_ == "image/jpg").map(_ => req)
}
```

We have defined an extractor called `JPeg` which returns `Some[Req]` if the content type of the upload is *image/jpg*; otherwise the result will be `None`. This is used in the REST pattern match as `JPeg(req)`. Note that the `unapply` needs to return `Option[Req]` as this is what's expected by the `Post` extractor.

See Also

Odersky *et al.*, (2008), *Programming in Scala*, chapter 24, discusses extractors in detail: *http://www.artima.com/pins1ed/extractors.html*.

Recipe 3.8 describes form-based (multi-part) file uploads

4.6. Returning JSON

Problem

You want to return JSON from a REST call.

Solution

Use the Lift JSON domain specific language (DSL). For example:

```
package code.rest

import net.liftweb.http.rest.RestHelper
import net.liftweb.http.LiftRules
import net.liftweb.json.JsonAST._
import net.liftweb.json.JsonDSL._

object QuotationsAPI extends RestHelper {

  def init() : Unit = {
    LiftRules.statelessDispatch.append(QuotationsAPI)
  }

  serve {
    case "quotation" :: Nil JsonGet req =>
      ("text" -> "A beach house isn't just real estate. It's a state of mind.")
~
        ("by" -> "Douglas Adams") : JValue
  }

}
```

Wire this into *Boot.scala*:

```
import code.rest.QuotationsAPI
QuotationsAPI.init()
```

Running this example produces:

```
$ curl -H 'Content-type: text/json' http://127.0.0.1:8080/quotation
{
 "text":"A beach house isn't just real estate. It's a state of mind.",
 "by":"Douglas Adams"
}
```

Discussion

The *type ascription* at the end of the JSON expression (`: JValue`) tells the compiler that the expression is expected to be of type `JValue`. This is required to allow the DSL to apply. If would not be required if, for example, you were calling a function that was defined to return a `JValue`.

The JSON DSL allows you to created nested structures, lists and everything else you expect of JSON.

In addition to the DSL, you can also create JSON from a case class by using the `Extraction.decompose` method:

```
import net.liftweb.json.Extraction
import net.liftweb.json.DefaultFormats

case class Quote(by: String, text: String)
val quote = Quote(
  "A beach house isn't just real estate. It's a state of mind.",
  "Douglas Adams")

implicit val formats = DefaultFormats
val json : JValue = Extraction decompose quote
```

This will also produce a `JValue`, which when printed will be:

```
{
 "by":"A beach house isn't just real estate. It's a state of mind.",
 "text":"Douglas Adams"
}
```

See Also

The README file for the lift-json project is a great source of examples for using the JSON DSL: *https://github.com/lift/framework/tree/master/core/json*.

4.7. Google Sitemap

Problem

You want to make a Google Sitemap using Lift's rendering capabilities.

Solution

Create the sitemap structure, and bind to it as you would for any template in Lift.

Start with a *sitemap.html* in your *webapp* folder containing valid XML-Sitemap markup:

```
<?xml version="1.0" encoding="utf-8" ?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
    <url data-lift="SitemapContent.base">
        <loc></loc>
        <changefreq>daily</changefreq>
        <priority>1.0</priority>
        <lastmod></lastmod>
    </url>
    <url data-lift="SitemapContent.list">
```

```
        <loc></loc>
        <lastmod></lastmod>
    </url>
</urlset>
```

Make a snippet to fill the required gaps:

```
package code.snippet

import org.joda.time.DateTime
import net.liftweb.util.CssSel
import net.liftweb.http.S
import net.liftweb.util.Helpers._

class SitemapContent {

  case class Post(url: String, date: DateTime)

  lazy val entries =
    Post("/welcome", new DateTime) :: Post("/about", new DateTime) :: Nil

  val siteLastUdated = new DateTime

  def base: CssSel =
    "loc *" #> "http://%s/".format(S.hostName) &
      "lastmod *" #> siteLastUdated.toString("yyyy-MM-dd'T'HH:mm:ss.SSSZZ")

  def list: CssSel =
    "url *" #> entries.map(post =>
      "loc *" #> "http://%s%s".format(S.hostName, post.url) &
        "lastmod *" #> post.date.toString("yyyy-MM-dd'T'HH:mm:ss.SSSZZ"))

}
```

This example is using canned data for two pages.

Apply the template and snippet in a REST service at */sitemap*:

```
package code.rest

import net.liftweb.http._
import net.liftweb.http.rest.RestHelper

object Sitemap extends RestHelper {
  serve {
    case Req("sitemap" :: Nil, _, GetRequest) =>
      XmlResponse(
        S.render(<lift:embed what="sitemap" />,
        S.request.get.request).head)
  }
}
```

Wire this into your application in *Boot.scala*, for example:

```
LiftRules.statelessDispatch.append(code.rest.Sitemap)
```

Test this service using a tool like cURL:

```
$ curl http://127.0.0.1:8080/sitemap

<?xml version="1.0" encoding="UTF-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
    <url>
        <loc>http://127.0.0.1/</loc>
        <changefreq>daily</changefreq>
        <priority>1.0</priority>
        <lastmod>2013-02-10T19:16:12.433+00:00</lastmod>
    </url>
    <url>
        <loc>http://127.0.0.1/welcome</loc>
        <lastmod>2013-02-10T19:16:12.434+00:00</lastmod>
    </url><url>
        <loc>http://127.0.0.1/about</loc>
        <lastmod>2013-02-10T19:16:12.434+00:00</lastmod>
    </url>
</urlset>
```

Discussion

You may be wondering why we've used REST here, when we could have used a regular HTML template and snippet. The reason is that we want XML rather than HTML output. We use the same mechanism, but invoke it and wrap it in an `XmlResponse`.

The `S.render` method takes a `NodeSeq` and a `HTTPRequst`. The first we supply by running the *sitemap.html* snippet; the second is simply the current request. `XmlResponse` requires a `Node` rather than a `NodeSeq`, which is why we call `head` — as there's only one node in the response, this does what we need.

Note that Google Sitemaps need dates to be in ISO 8601 format. The built-in `java.text.SimpleDateFormat` does not support this format prior to Java 7. If you are using Java 6 you need to use `org.joda.time.DateTime` as we are in this example.

See Also

Sitemaps are described at: *http://support.google.com/webmasters/bin/answer.py?hl=en&answer=156184*.

4.8. Calling REST Service from a Native iOS Application

Problem

You want to make a HTTP POST from a native iOS device to a Lift REST service.

Solution

Use `NSURLConnection` ensuring you set the content-type to *application/json*.

For example, suppose we want to call this service:

```
package code.rest

import net.liftweb.http.rest.RestHelper
import net.liftweb.json.JsonDSL._
import net.liftweb.json.JsonAST._

object Shouty extends RestHelper {

  def greet(name: String) : JValue =
    "greeting" -> ("HELLO "+name.toUpperCase)

  serve {
    case "shout" :: Nil JsonPost json->request =>
      for { JString(name) <- (json \\ "name").toOpt }
      yield greet(name)
  }

}
```

The service expects a JSON post with a parameter of "name", and it returns a greeting as a JSON object. To demonstrate the data to and from the service, we can include the service in *Boot.scala*…

```
LiftRules.statelessDispatch.append(Shouty)
```

…and then call it from the command line:

```
$ curl -d '{ "name" : "Richard" }' -X POST -H 'Content-type: application/json'
   http://127.0.0.1:8080/shout
{
  "greeting":"HELLO RICHARD"
}
```

We can implement the POST request using `NSURLConnection`:

```
static NSString *url = @"http://localhost:8080/shout";

-(void) postAction {
  // JSON data:
  NSDictionary* dic = @{@"name": @"Richard"};
  NSData* jsonData =
    [NSJSONSerialization dataWithJSONObject:dic options:0 error:nil];
  NSMutableURLRequest *request = [
    NSMutableURLRequest requestWithURL:[NSURL URLWithString:url]
    cachePolicy:NSURLRequestUseProtocolCachePolicy timeoutInterval:60.0];

  // Construct HTTP request:
  [request setHTTPMethod:@"POST"];
```

```
    [request setValue:@"application/json" forHTTPHeaderField:@"Content-Type"];
    [request setValue:[NSString stringWithFormat:@"%d", [jsonData length]]
      forHTTPHeaderField:@"Content-Length"];
    [request setHTTPBody: jsonData];

    // Send the request:
    NSURLConnection *con = [[NSURLConnection alloc]
      initWithRequest:request delegate:self];
}

- (void)connection:(NSURLConnection *)connection
  didReceiveResponse:(NSURLResponse *)response {
   // Start off with new, empty, response data
   self.receivedJSONData = [NSMutableData data];
}

- (void)connection:(NSURLConnection *)connection
  didReceiveData:(NSData *)data {
   // append incoming data
   [self.receivedJSONData appendData:data];
}

- (void)connection:(NSURLConnection *)connection
  didFailWithError:(NSError *)error {
   NSLog(@"Error occurred ");
}

- (void)connectionDidFinishLoading:(NSURLConnection *)connection {
  NSError *e = nil;
  NSDictionary *JSON =
    [NSJSONSerialization JSONObjectWithData: self.receivedJSONData
    options: NSJSONReadingMutableContainers error: &e];
  NSLog(@"Return result: %@", [JSON objectForKey:@"greeting"]);
}
```

Obviously in this example we've used hard-coded values and URLs, but this will hopefully be a starting point for use in your application.

Discussion

There are many ways to do HTTP POST from iOS and it can be confusing to identify the correct way that works, especially without the aid of external library. The example above uses the iOS native API.

Another way is to use *AFNetworking*. This is a popular external library for iOS development, can cope with many scenarios and is simple to use:

```
#import "AFHTTPClient.h"
#import "AFNetworking.h"
#import "JSONKit.h"

static NSString *url = @"http://localhost:8080/shout";
```

```
-(void) postAction {
  // JSON data:
  NSDictionary* dic = @{@"name": @"Richard"};
  NSData* jsonData =
   [NSJSONSerialization dataWithJSONObject:dic options:0 error:nil];

  // Construct HTTP request:
  NSMutableURLRequest *request =
   [NSMutableURLRequest requestWithURL:[NSURL URLWithString:url]
    cachePolicy:NSURLRequestUseProtocolCachePolicy timeoutInterval:60.0];
  [request setHTTPMethod:@"POST"];
  [request setValue:@"application/json" forHTTPHeaderField:@"Content-Type"];
  [request setValue:[NSString stringWithFormat:@"%d", [jsonData length]]
    forHTTPHeaderField:@"Content-Length"];
  [request setHTTPBody: jsonData];

  // Send the request:
  AFJSONRequestOperation *operation =
    [[AFJSONRequestOperation alloc] initWithRequest: request];
  [operation setCompletionBlockWithSuccess:^(AFHTTPRequestOperation *operation,
    id responseObject)
  {
     NSString *response = [operation responseString];

     // Use JSONKit to deserialize the response into NSDictionary
     NSDictionary *deserializedJSON = [response objectFromJSONString];
     [deserializedJSON count];

     // The response object can be a NSDicionary or a NSArray:
      if([deserializedJSON count]> 0) {
         NSLog(@"Return value: %@",[deserializedJSON objectForKey:@"greeting"]);
      }
      else {
        NSArray *deserializedJSONArray = [response objectFromJSONString];
        NSLog(@"Return array value: %@", deserializedJSONArray );
      }
  }failure:^(AFHTTPRequestOperation *operation, NSError *error)
  {
    NSLog(@"Error: %@",error);
  }];
  [operation start];
}
```

The `NSURLConnection` approach is more versatile and gives you starting point to craft your own solution, such as by making the content type more specific. However, `AFNetworking` is popular and you may prefer that route.

See Also

You may find the "Complete REST Example" in *Simply Lift* to be a good test ground for your calls to Lift. *http://simply.liftweb.net/index-5.4.html*.

CHAPTER 5

JavaScript, Ajax, Comet

Lift is known for its great Ajax and comet support, and in this chapter we'll explore these.

For an introduction to Lift's Ajax and Comet features, see *Simply Lift* at *http://simply.liftweb.net*, chapter 9 of *Lift in Action* (Perrett, 2012, Manning Publications Co.), or watch Diego Medina's video presentation at *https://fmpwizard.telegr.am/blog/comet-actors-presentation*.

The source code for this chapter is: *https://github.com/LiftCookbook/cookbook_ajax*.

5.1. Trigger Server-Side Code from a Button

Problem

You want to trigger some server-side code when the user presses a button.

Solution

Use `ajaxInvoke`:

```
package code.snippet

import net.liftweb.util.Helpers._
import net.liftweb.http.SHtml
import net.liftweb.http.js.{JsCmd, JsCmds}
import net.liftweb.common.Loggable

object AjaxInvoke extends Loggable {

  def callback() : JsCmd = {
    logger.info("The button was pressed")
    JsCmds.Alert("You clicked it")
```

```
    }

    def button = "button [onclick]" #> SHtml.ajaxInvoke(callback)
  }
```

In this snippet we are binding the click event of a button to an `ajaxInvoke`: when the button is pressed, Lift arranges for the function you gave `ajaxInvoke` to be executed.

That function, `callback`, is just logging a message and returning a JavaScript alert to the browser. The corresponding HTML might include:

```
<div data-lift="AjaxInvoke.button">
  <button>Click Me</button>
</div>
```

Discussion

The signature of the function you pass to `ajaxInvoke` is `Unit => JsCmd`, meaning you can trigger a range of behaviours, from returning `Noop` if you want nothing to happen, through changing DOM element, all the way up to executing arbitrary JavaScript.

The example above is using a button, but will work on any element that has an event you can bind to. We're binding to *onclick* but it could be any event the DOM exposes.

Related to `ajaxInvoke` are the following functions:

- `SHtml.onEvent` which calls a function with the signature `String => JsCmd` because it is passed the `value` of the node it is attached to. In the above example, this would be the empty string as the button has no value.
- `SHtml.ajaxCall` is more general than `onEvent`, as you give it the expression you want passed to your server-side code.
- `SHtml.jsonCall` is even more general still: you give it a function that will return a JSON object when executed on the client, and this object will be passed to your server-side function.

Let's look at each of these in turn.

onEvent — Receiving the `value` of a DOM Element

You can use `onEvent` with any element that has a `value` attribute and responds to the event you choose to bind to. The function you supply to `onEvent` is called with the element's value. As an example, we can write a snippet which presents a challenge to the user, and validates the response:

```
package code.snippet

import scala.util.Random
import net.liftweb.util.Helpers._
```

```
import net.liftweb.http.SHtml
import net.liftweb.http.js.JsCmds.Alert

object OnEvent {

  def render = {
    val x, y = Random.nextInt(10)
    val sum = x + y

    "p *" #> "What is %d + %d?".format(x,y) &
    "input [onchange]" #> SHtml.onEvent( answer =>
      if (answer == sum.toString) Alert("Correct!")
      else Alert("Try again")
     )
  }

}
```

This snippet prompts the user to add the two random numbers presented in the `<p>` tag, and binds a validation function to the `<input>` on the page:

```
<div data-lift="OnEvent">
  <p>Problem appears here</p>
  <input placeholder="Type your answer"></input>
</div>
```

When `onchange` is triggered (by pressing return or the tab key, for example), the text entered is sent to our `onEvent` function as a `String`. On the server-side we chance the answer and send back "Correct!" or "Try again" as a JavaScript alert.

ajaxCall — Receiving an Arbitrary Client-Side String

Where `onEvent` sends `this.value` to your server-side code, `ajaxCall` allows you to specify the client-side expression used to produce a value.

To demonstrate this we can create a template that includes two elements: a button and a text field. We'll bind our function to the button, but read a value from the input field:

```
<div data-lift="AjaxCall">
  <input id="num" value="41"></input>
  <button>Increment</button>
</div>
```

We want to arrange for the button to read the `num` field, increment it, and return it back to the input field:

```
package code.snippet

import net.liftweb.util.Helpers._
import net.liftweb.http.SHtml
import net.liftweb.http.js.JE.ValById
import net.liftweb.http.js.JsCmds._
```

```
object AjaxCall {

 def increment(in: String) : String =
  asInt(in).map(_ + 1).map(_.toString) openOr in

 def render = "button [onclick]" #>
   SHtml.ajaxCall(ValById("num"), s => SetValById("num", increment(s)) )

 }
```

The first argument to `ajaxCall` is the expression that will produce a value for our function. It can be any `JsExp`, and we've used `ValById` which looks up the value of an element by the ID attribute. We could have used a regular jQuery expression to achieve the same effect with: `JsRaw("$('#num').val()")`.

Our second argument to `ajaxCall` takes the value of the `JsExp` expression as a `String`. We're using one of Lift's JavaScript command to replaces the value with a new value. The new value is the result of incrementing the number (providing it is a number).

The end result is that you press the button, and the number updates. It should go without saying that these are simple illustrations, and you probably don't want a server round-trip to add one to a number. The techniques come into their own when there is some action of value to perform on the server.

You may have guessed that `onEvent` is implemented as an `ajaxCall` for `JsRaw("this.value")`.

jsonCall — Receiving a JSON Value

Both `ajaxCall` and `onEvent` end up evaluating a `String => JsCmd` function. By contrast, `jsonCall` has the signature `JValue => JsCmd`, meaning you can pass more complex data structures from JavaScript to your Lift application.

To demonstrate this, we'll create a template that asks for input, has a function to convert the input into JSON, and a button to send the JSON to the server:

```
<div data-lift="JsonCall">
  <p>Enter an addition question:</p>
  <div>
    <input id="x"> + <input id="y"> = <input id="z">.
  </div>
  <button>Check</button>
</div>

<script type="text/javascript">
// <![CDATA[
function currentQuestion() {
  return {
    first:  parseInt($('#x').val()),
```

```
      second: parseInt($('#y').val()),
      answer: parseInt($('#z').val())
    };
  }
  // ]]>
```

The `currentQuestion` function is creating an object, which will be turned into a JSON string when sent to the server. On the server we'll check that this JSON represents a valid integer addition problem:

```
package code.snippet

import net.liftweb.util.Helpers._
import net.liftweb.http.SHtml
import net.liftweb.http.js.{JsCmd, JE}
import net.liftweb.common.Loggable
import net.liftweb.json.JsonAST._
import net.liftweb.http.js.JsCmds.Alert
import net.liftweb.json.DefaultFormats

object JsonCall extends Loggable {

  implicit val formats = DefaultFormats

  case class Question(first: Int, second: Int, answer: Int) {
    def valid_? = first + second == answer
  }

  def render = {

    def validate(value: JValue) : JsCmd = {
      logger.info(value)
      value.extractOpt[Question].map(_.valid_?) match {
        case Some(true) => Alert("Looks good")
        case Some(false) => Alert("That doesn't add up")
        case None => Alert("That doesn't make sense")
      }
    }

    "button [onclick]" #>
      SHtml.jsonCall( JE.Call("currentQuestion"), validate _ )
  }
}
```

Working from the bottom of this snippet up, we see a binding of the `<button>` to the `jsonCall`. The value we'll be working on is the value provided by the JavaScript function called `currentQuestion`. This was defined on the template page. When the button is clicked the JavaScript function is called and the resulting value will be presented to `validate`, which is our `JValue => JsCmd` function.

All `validate` does is log the JSON data and alert back if the question looks correct or not. To do this we use the Lift JSON ability to extract JSON to a case class and call the `valid_?` test to see if the numbers add up. This will evaluate to `Some(true)` if the addition works, `Some(false)` if the addition isn't correct or `None` if the input is missing or not a valid integer.

Running the code and entering 1, 2 and 3 into the text fields will produce the following in the server log:

```
JObject(List(JField(first,JInt(1)), JField(second,JInt(2)),
  JField(answer,JInt(3))))
```

This is the `JValue` representation of the JSON.

See Also

Recipe 5.2 includes an example of `SHtml.onEvents` which will bind a function to a number of events on a `NodeSeq`.

For another example of `AjaxInvoke` take a look at the *Call Scala code from JavaScript* section of Diego Medina's blog at: *http://blog.fmpwizard.com/scala-lift-custom-wizard*.

Exploring Lift, chapter 10, lists various `JsExp` classes you can use for `ajaxCall`: *http://exploring.liftweb.net/master/index-10.html*.

Recipe 3.3 using `JsonHandler` to send JSON data from a form to the server.

5.2. Call Server when Select Option Changes

Problem

When a HTML select option is selected, you want to trigger a function on the server.

Solution

Register a `String => JsCmd` function with `SHtml.ajaxSelect`.

In this example we will lookup the distance from Earth to the planet a user selects. This lookup will happen on the server and update the browser with the result. The interface is:

```
<div data-lift="HtmlSelectSnippet">
  <div>
    <label for="dropdown">Planet:</label>
    <select id="dropdown"></select>
  </div>
  <div id="distance">Distance will appear here</div>
</div>
```

The snippet code binds the `<select>` element to send the selected value to the server:

```
package code.snippet

import net.liftweb.common.Empty
import net.liftweb.util.Helpers._
import net.liftweb.http.SHtml.ajaxSelect
import net.liftweb.http.js.JsCmd
import net.liftweb.http.js.JsCmds.SetHtml
import xml.Text

class HtmlSelectSnippet {

  // Our "database" maps planet names to distances:
  type Planet = String
  type LightYears = Double

  val database = Map[Planet,LightYears](
    "Alpha Centauri Bb" -> 4.23,
    "Tau Ceti e" -> 11.90,
    "Tau Ceti f" -> 11.90,
    "Gliese 876 d" -> 15.00,
    "82 G Eridani b" -> 19.71
  )

  def render = {

    // To show the user a blank label and blank value option:
    val blankOption = ("" -> "")

    // The complete list of options includes everything in our database:
    val options : List[(String,String)] =
      blankOption ::
      database.keys.map(p => (p,p)).toList

    // Nothing is selected by default:
    val default = Empty

    // The function to call when an option is picked:
    def handler(selected: String) : JsCmd = {
      SetHtml("distance", Text(database(selected) + " light years"))
    }

    // Bind the <select> tag:
    "select" #> ajaxSelect(options, default, handler)
  }
}
```

The last line of the code is doing the work for us. It is generating the options and binding the selection to a function called `handler`. The handler function is called with the value of the selected item.

We're using the same `String` (the planet name) for the option label and value, but they could be different.

Discussion

To understand what's going on here, take a look at the HTML that Lift produces:

```
<select id="dropdown"
  onchange="liftAjax.lift_ajaxHandler('F470183993611Y15ZJU=' +
    this.options[this.selectedIndex].value, null, null, null)">
  <option value=""></option>
  <option value="Tau Ceti e">Tau Ceti e</option>
  ...
</select>
```

The `handler` function has been stored by Lift under the identifier of "F470183993611Y15ZJU" (in this particular rendering). An "onchange" event handler is attached to the select element and is responsible for transporting the selected value to the server, and bringing a value back. The `lift_ajaxHandler` JavaScript function is defined in *liftAjax.js* which is automatically added to your page.

Collecting the Value on Form Submission

If you need to additionally capture the selected value on a regular form submission, you can make use of `SHtml.onEvents`. This attaches event listeners to a `NodeSeq`, triggering a server-side function when the event occurs. We can use this with a regular form with a regular select box, but wire in Ajax calls to the server when the select changes.

To make use of this, our snippet changes very little:

```
var selectedValue : String = ""

"select" #> onEvents("onchange")(handler) {
  select(options, default, selectedValue = _)
} &
"type=submit" #> onSubmitUnit( () => S.notice("Destination "+selectedValue))
```

We are arranging for the same `handler` function to be called when an "onchange" event is triggered. This event binding is applied to a regular `SHtml.select`, which is storing the `selectedValue` when the form is submitted. We also bind a submit button to a function which generates a notice of which planet was selected.

The corresponding HTML also changes little. We need to add a button and make sure the snippet is marked as a form with `?form`:

```
<div data-lift="HtmlSelectFormSnippet?form=post">

  <div>
    <label for="dropdown">Planet:</label>
    <select id="dropdown"></select>
```

```
    </div>

    <div id="distance">Distance will appear here</div>

    <input type="submit" value="Book Ticket"/>

  </div>
```

Now when you change a selected value you see the dynamically updated distance calculation, but pressing the "Book Ticket" button also delivers the value to the server.

See Also

Recipe 3.7 describes how to use classes rather than `String` values for select boxes.

5.3. Creating Client-Side Actions in Your Scala Code

Problem

In your Lift code you want to set up an action that is run purely in client-side JavaScript.

Solution

Bind your JavaScript directly to the event handler you want to run.

Here's an example where we make a button slowly fade away when you press it, but notice that we're setting up this binding in our server-side Lift code:

```
package code.snippet

import net.liftweb.util.Helpers._

object ClientSide {
  def render = "button [onclick]" #> "$(this).fadeOut()"
}
```

In the template we'd perhaps say:

```
<div data-lift="ClientSide">
  <button>Click Me</button>
</div>
```

Lift will render the page as:

```
<button onclick="$(this).fadeOut()">Click Me</button>
```

Discussion

Lift includes a JavaScript abstraction which you can use to build up more elaborate expressions for the client-side. For example you can combine basic commands…

```
import net.liftweb.http.js.JsCmds.{Alert, RedirectTo}

def render = "button [onclick]" #>
  (Alert("Here we go...") & RedirectTo("http://liftweb.net"))
```

…which pops up an alert dialog and then sends you to *http://liftweb.net*. The HTML would be rendered as:

```
<button onclick="alert(&quot;Here we go...&quot;);
window.location = &quot;http://liftweb.net&quot;;">Click Me</button>
```

Another option is to use `JE.Call` to execute a JavaScript function with parameters. Suppose we have this function defined:

```
function greet(who, times) {
  for(i=0; i<times; i++)
    alert("Hello "+who);
}
```

We could bind a client-side button press to this client-side function like this:

```
import net.liftweb.http.js.JE

def render =
  "button [onclick]" #> JE.Call("greet", "World!", 3)
```

On the client-side, we'd see:

```
<button onclick="greet(&quot;World!&quot;,3)">Click Me For Greeting</button>
```

Note that the types `String` and `Int` have been preserved in the JavaScript syntax of the call. This has happened because `JE.Call` takes a variable number of `JsExp` arguments after the JavaScript function name. There are wrappers for JavaScript primitive types (`JE.Str`, `JE.Num`, `JsTrue`, `JsFalse`) and implicit conversions to save you having to wrap the Scala values yourself.

See Also

Chapter 10 of *Exploring Lift* at *http://exploring.liftweb.net/* gives a list of `JsCmds` and `JE` expressions.

5.4. Focus on a Field on Page Load

Problem

When a page loads you want the browser to select a particular field for input from the keyboard.

Solution

Wrap the input with a `FocusOnLoad` command:

```
package code.snippet

import net.liftweb.util.Helpers._
import net.liftweb.http.js.JsCmds.FocusOnLoad

class Focus {
  def render = "name=username" #> FocusOnLoad(<input type="text"/>)
}
```

The CSS transform in `render` will match against `name="username"` element in the HTML and replace it with a text input field that will be focused on automatically when the page loads.

Although we're focusing on in-line HTML, this could be any `NodeSeq`, such as the one produced by `SHtml.text`.

Discussion

`FocusOnLoad` is an example of a `NodeSeq => NodeSeq` transformation. It appends to the `NodeSeq` with the JavaScript required to set focus on that field.

The JavaScript that performs the focus simply looks up the node in the DOM by ID and calls `focus` on it. Although the example code above doesn't specify an ID, the `FocusOn` command is smart enough to add one automatically for us.

There are two related `JsCmd`:

- `Focus` — takes an element ID and sets focus on the element.
- `SetValueAndFocus` — which is like `Focus` but takes an additional `String` value to populate the element with.

These two are useful if you need to set focus from Ajax or Comet components dynamically.

See Also

The source for `FocusOnLoad` is worth checking out to understand how it, and related commands, are constructed. This may help you package your own JavaScript functionality up into commands that can be used in CSS binding expressions: *https://github.com/lift/framework/blob/master/web/webkit/src/main/scala/net/liftweb/http/js/JsCommands.scala*.

5.5. Add CSS Class to an Ajax Form

Problem

You want to set the CSS class of an Ajax form.

Solution

Name the class via `?class=` query parameter:

```
<form data-lift="form.ajax?class=boxed">
...
</form>
```

Discussion

If you need to set multiple CSS classes, encode a space between the class names, e.g., `class=boxed+primary`.

The `form.ajax` construction is a regular snippet call: the `Form` snippet is one of the handful of built-in snippets, and in this case we're calling the `ajax` method on that object. However, normally snippet calls do not copy attributes into the resulting markup, but this snippet is implemented to do exactly that.

See Also

For an example of accessing these query parameters in your own snippets, see Recipe 2.5.

Simply Lift, chapter 4, introduces Ajax forms at *http://simply.liftweb.net/*.

5.6. Running a Template via JavaScript

Problem

You want to load an entire page — a template with snippets — inside of the current page (i.e., without a browser refresh).

Solution

Use `Template` to load the template, and `SetHtml` to place the content on the page.

Let's populate a `<div>` with the site home page when a button is pressed:

```
<div data-lift="TemplateLoad">
  <div id="inject">Content will appear here</div>
  <button>Load Template</button>
</div>
```

The corresponding snippet would be:

```
package code.snippet

import net.liftweb.util.Helpers._
import net.liftweb.http.{SHtml, Templates}
import net.liftweb.http.js.JsCmds.{SetHtml, Noop}
import net.liftweb.http.js.JsCmd

object TemplateLoad {

  def content : JsCmd =
    Templates("index" :: Nil).map(ns => SetHtml("inject", ns)) openOr Noop

  def render = "button [onclick]" #> SHtml.ajaxInvoke(content _)
}
```

Clicking the button will cause the content of */index.html* to be loaded into the `inject` element.

Discussion

`Templates` produces a `Box[NodeSeq]`. In the example above, we map this content into a `JsCmd` which will populate the `inject` div.

If you write unit tests to access templates, be aware that you may need to modify your development or testing environment to include the *webapps* folder. To do this for SBT, add the following to *build.sbt*:

```
unmanagedResourceDirectories in Test <+= (baseDirectory) { _ / "src/main/
webapp" }
```

For this to work in your IDE you'll need to add *webapp* as a source folder to locate templates.

See Also

Recipe 5.1 describes `ajaxInvoke` and related methods.

5.7. Move JavaScript to End of Page

Problem

You want the JavaScript created in your snippet to be included at the end of the HTML page.

Solution

Use `S.appendJs` which places your JavaScript just before the closing `</body>` tag, along with other JavaScript produced by Lift.

In this HTML we have placed a `<script>` tag in the middle of the page, and marked it with a snippet called `JavaScriptTail`:

```
<!DOCTYPE html>
<head>
  <meta content="text/html; charset=UTF-8" http-equiv="content-type" />
  <title>JavaScript in Tail</title>
</head>
<body data-lift-content-id="main">
<div id="main" data-lift="surround?with=default;at=content">
  <h2>JavaScript in the tail of the page</h2>

  <script type="text/javascript" data-lift="JavaScriptTail">
  </script>

  <p>
    The JavaScript about to be run will have been moved
    to the end of this page, just before the closing
    body tag.
  </p>
</div>
</body>
</html>
```

The `<script>` content will be generated by a snippet. It doesn't need to be a `<script>` tag: the snippet just replaces the content with nothing, but hanging the snippet on the `<script>` tag is a reminder of the purpose of the snippet:

```
package code.snippet

import net.liftweb.util.Helpers._
import net.liftweb.http.js.JsCmds.Alert
import net.liftweb.http.S
import xml.NodeSeq

class JavaScriptTail {
  def render = {
    S.appendJs(Alert("Hi"))
    "*" #> NodeSeq.Empty
  }
}
```

Although the snippet is rendering nothing, it calls `S.appendJs` with a `JsCmd`. This will produce the following in the page just before the end of the body:

```
<script type="text/javascript">
// <![CDATA[
```

```
jQuery(document).ready(function() {
  alert("Hi");
});
// ]]>
</script>
```

Although the snippet was in the middle of the page, the JavaScript appears at the end of the page.

Discussion

There are three other ways you could tackle this problem. The first is to move your JavaScript to an external file, and simply include it on the page where you want it. For substantial JavaScript code, this might make sense.

The second is a variation on `S.appendJs`: `S.appendGlobalJs` works in the same way but does not include the jQuery `ready` around your JavaScript. This means you have no guarantee the DOM has loaded when your function is called.

A third option is wrap your JavaScript in a `<lift:tail>` snippet:

```
class JavascriptTail {
  def render =
    "*" #> <lift:tail>{Script(OnLoad(Alert("Hi")))}</lift:tail>
}
```

Note that `lift:tail` is a general-purpose Lift snippet and be used to move various kinds of content to the end of the page, not just JavaScript.

See Also

Recipe 2.9 discusses a related Lift snippet for moving content to the head of the page.

Recipe 2.7 describes the different ways of invoking a snippet, such as `<lift:tail>` versus `data-lift="tail"`.

5.8. Run JavaScript on Comet Session Loss

Problem

You're using a comet actor and you want to arrange for some JavaScript to be executed in the event of the session being lost.

Solution

Configure your JavaScript via `LiftRules.noCometSessionCmd`.

As an example we can modify the standard Lift chat demo to save the message being typed in the event of the session loss. In the style of the demo we would have a Ajax form for entering a message and the comet chat area for displaying messages received:

```
<form data-lift="form.ajax">
  <input type="text" data-lift="ChatSnippet" id="message"
    placeholder="Type a message" />
</form>

<div data-lift="comet?type=ChatClient">
  <ul>
    <li>A message</li>
  </ul>
</div>
```

To this we can add a function, `stash`, which we want to be called in the event of a comet session being lost:

```
<script type="text/javascript">
// <![CDATA[
function stash() {
  saveCookie("stashed", $('#message').val());
  location.reload();
}

jQuery(document).ready(function() {
  var stashedValue = readCookie("stashed") || "";
  $('#message').val(stashedValue);
  deleteCookie("stashed");
});

// Definition of saveCookie, readCookie, deleteCookie omitted.

</script>
```

Our `stash` function will save the current value of the input field in a cookie called "stashed". We arrange, on page load, to check for that cookie and insert the value into our message field.

The final part is to modify *Boot.scala* to register our `stash` function:

```
import net.liftweb.http.js.JsCmds.Run

LiftRules.noCometSessionCmd.default.set( () => Run("stash()") )
```

In this way, if a session is lost while composing a chat message, the browser will stash the message, and when the page re-loads the message will be recovered.

To test the example, type a message into the message field, then restart your Lift application. Wait 10 seconds, and you'll see the effect.

Discussion

Without changing `noCometSessionCmd`, the default behavior of Lift is to arrange for the browser to load the page `LiftRules.noCometSessionPage` — which will be / unless you change it. This is carried out via the JavaScript function `lift_sessionLost` in *comet-Ajax.js*.

By providing our own `() => JsCmd` function to `LiftRules.noCometSessionCmd`, Lift arranges to call this function and deliver the `JsCmd` to the browser, rather than `lift_sessionLost`. If you watch the HTTP traffic between your browser and Lift, you'll see the `stash` function call being returned in response to a comet request.

Factory

The `noCometSessionCmd.default.set` call is making use of Lift's dependency injection. Specifically, it's setting up the supply side of the dependency. Although we're setting a default here, it's possible in Lift to supply different behaviours with different scopes: request or session. See *https://www.assembla.com/spaces/liftweb/wiki/Dependency_Injection*.

This recipe has focused on the handling of loss of session for Comet, and for Ajax, there's a corresponding `LiftRules.noAjaxSessionCmd` setting.

See Also

You'll find the *The ubiquitous Chat app* in *Simply Lift*: *http://simply.liftweb.net/*.

Being able to debug HTTP traffic is a useful way to understand how your Comet or Ajax application is performing. There are many plugins and products to help with this, such as the *HttpFox* plugin for Firefox: *https://addons.mozilla.org/en-us/firefox/addon/httpfox/*.

5.9. Ajax File Upload

Problem

You want to offer your users an Ajax file upload tool, with progress bars and drag and drop support.

Solution

Add Sebastian Tschan's *jQuery File Upload* widget (*https://github.com/blueimp/jQuery-File-Upload*) to your project, and implement a REST end point to receive files.

The first step is to download the widget, and drag the *js* folder into your application as *src/main/webapp/js*. We can then use the JavaScript in a template:

```
<!DOCTYPE HTML>
<html>
<head>
  <meta charset="utf-8">
  <title>jQuery File Upload Example</title>
</head>
<body>

<h1>Drag files onto this page</h1>

<input id="fileupload" type="file" name="files[]" data-url="/upload" multiple>

<div id="progress" style="width:20em; border: 1pt solid silver; display: none">
  <div id="progress-bar" style="background: green; height: 1em; width:0%"></div>
</div>

<script src="//ajax.googleapis.com/ajax/libs/jquery/1.8.3/jquery.min.js">
</script>
<script src="js/vendor/jquery.ui.widget.js"></script>
<script src="js/jquery.iframe-transport.js"></script>
<script src="js/jquery.fileupload.js"></script>

<script>
  $(function () {
    $('#fileupload').fileupload({
      dataType: 'json',
      add: function (e,data) {
        $('#progress-bar').css('width', '0%');
        $('#progress').show();
        data.submit();
      },
      progressall: function (e, data) {
        var progress = parseInt(data.loaded / data.total * 100, 10) + '%';
        $('#progress-bar').css('width', progress);
      },
      done: function (e, data) {
        $.each(data.files, function (index, file) {
          $('<p/>').text(file.name).appendTo(document.body);
        });
        $('#progress').fadeOut();
      }
    });
  });
</script>

</body>
</html>
```

This template provides an input field for files, an area to use as a progress indicator, and configures the widget when the page loads in a jQuery `$( ... )` block. This is just regular usage of the JavaScript widget, and nothing particularly Lift-specific.

The final part is to implement a Lift REST service to receive the file or files. The URL of the service, */upload*, is set in `data-url` on the `input` field, and that's the address we match on:

```
package code.rest

import net.liftweb.http.rest.RestHelper
import net.liftweb.http.OkResponse

object AjaxFileUpload extends RestHelper {

  serve {

    case "upload" :: Nil Post req =>
      for (file <- req.uploadedFiles) {
        println("Received: "+file.fileName)
      }
      OkResponse()

  }

}
```

This implementation simply logs the name of the file received and acknowledges successful delivery with a 200 status code back to the widget.

As with all REST services, it needs to be registered in *Boot.scala*:

```
LiftRules.dispatch.append(code.rest.AjaxFileUpload)
```

By default the widget makes the whole HTML page a drop-target for files, meaning you can drag a file onto the page and it will immediately be uploaded to your Lift application.

Discussion

In this recipe we've shown just the basic integration of the widget with a Lift application. The demo site for the widget, *http://blueimp.github.com/jQuery-File-Upload/*, shows other capabilities, and provides documentation on how to integrate them.

Many of the features just require JavaScript configuration. For example, we've used the widget's `add`, `progressall`, and `done` handlers to show, update and then fade out a progress bar. When the upload is completed, the name of the uploaded file is appended to the page.

In the REST service the uploaded file are available via the `uploadedFiles` method on the request. When Lift receives a multi-part form it automatically extracts files as `up`

`loadedFiles`, each of which is a `FileParamHolder` which gives us access to the `fileName`, `length`, `mimeType` and `fileStream`.

By default uploaded files are held in memory, but that can be changed (see "Discussion" in Recipe 3.8).

In the recipe we return a 200 (`OkResponse`). If we wanted to signal to the widget that a file was rejected we can return another code. For example, perhaps we want to reject all files except PNG images. On the server-side we can do that by replacing the `OkResponse` with a test:

```
import net.liftweb.http.{ResponseWithReason, BadResponse, OkResponse}

if (req.uploadedFiles.exists( _.mimeType != "image/png" ))
  ResponseWithReason(BadResponse(), "Only PNGs")
else
  OkResponse()
```

We would mirror this with a `fail` handler in the client JavaScript:

```
fail: function (e, data) {
  alert(data.errorThrown);
}
```

If we uploaded, say a JPEG, the browser would show an alert dialog reporting "Only PNGs".

See Also

Diego Medina has posted a Gist of Lift REST code to integrate more fully with the image upload and image reviewing features of the widget, specifically implementing the JSON response that the widget expects for that functionality. You'll find it on GitHub via: *http://bit.ly/lift-restupload*.

Recipe 3.8 describes the basic file upload behaviour of Lift and how to control where files are stored.

Antonio Salazar Cardozo has posted example code for performing Ajax file upload using Lift's Ajax mechanisms. This avoids external JavaScript libraries. You can find a description and link to the code on the mailing list: *http://bit.ly/lift-ajaxupload*.

5.10. Format A Wired Cell

Problem

You want a wired UI element to have a different format than plain conversion to a string. For example, you'd like display a floating point value as a currency.

Solution

Use the `WiringUI.toNode` method to create a wiring node that can render the output formatted as you desire.

As an example consider a HTML template to display the quantity of an item being purchased and the subtotal:

```
<div data-lift="Wiring">

<table>
  <tbody>
    <tr><td>Quantity</td><td id="quantity">?</td></tr>
    <tr><td>Subtotal</td><td id="subtotal">?</td></tr>
  </tbody>
</table>

<button id="add">Add Another One</button>

</div>
```

We'd like the subtotal to display as US dollars. The snippet would be:

```
package code.snippet

import java.text.NumberFormat
import java.util.Locale

import scala.xml.{NodeSeq, Text}

import net.liftweb.util.Helpers._
import net.liftweb.util.{Cell, ValueCell}
import net.liftweb.http.{S, WiringUI}
import net.liftweb.http.SHtml.ajaxInvoke
import net.liftweb.http.js.JsCmd

class Wiring {

  val cost = ValueCell(1.99)
  val quantity = ValueCell(1)
  val subtotal = cost.lift(quantity)(_ * _)

  val formatter = NumberFormat.getCurrencyInstance(Locale.US)

  def currency(cell: Cell[Double]): NodeSeq => NodeSeq =
    WiringUI.toNode(cell)((value, ns) => Text(formatter format value))

  def increment(): JsCmd  = {
    quantity.atomicUpdate(_ + 1)
    S.notice("Added One")
  }

  def render =
```

```
      "#add [onclick]" #> ajaxInvoke(increment) &
      "#quantity *" #> WiringUI.asText(quantity) &
      "#subtotal *" #> currency(subtotal)

  }
```

We have defined a `currency` method to format the `subtotal` not as a `Double` but as currency amount using the Java number formatting capabilities. This will result in values like "$19.90" being shown rather than "19.9".

Discussion

The primary `WiringUI` class makes it easy to bind a cell as text. The `asText` method works by converting a value to a `String` and wrapping it in a `Text` node. This is done via `toNode`, however we can use the `toNode` method directly to generate a transform function that is both hooked into the wiring UI, and uses our code for the translation of the item.

The mechanism is type-safe. In this example `cost` is a `Double` cell, `quantity` is an `Int` cell, and `subtotal` is inferred as a `Cell[Double]`. This is why our formatting function is passed `value` as a `Double`.

Note that the function passed to `toNode` must return a node sequence. This gives a great deal of flexibility as you can return any kind of markup in a `NodeSeq`. Our example complies with this signature by wrapping a text value in a `Text` object.

The `WiringUI.toNode` requires a `(T, NodeSeq) => NodeSeq`. In the example above, we ignore the `NodeSeq`, but the value would be the contents of the element we've bound to. Given the input…

```
<td id="subtotal">?</td>
```

…this would mean the `NodeSeq` passed to us would just be the text node representing "?". With a richer template we can use CSS selectors. For example, we can modify the template…

```
<td>Subtotal</td><td id="subtotal">
  <i>The value is <b class="amount">?</b></i>
</td>
```

…and apply a CSS selector to change the "amount" element:

```
(value, ns) => (".amount *" #> Text(formatter format value)) apply ns)
```

See Also

Chapter 6 of *Simply Lift* describes Lift's Wiring mechanism, and gives a detailed shopping example. *http://simply.liftweb.net/*.

CHAPTER 6
Request Pipeline

When a request reaches Lift there are a number of points where you can jump in an control what Lift does, send back a different kind of response, or control access. This chapter looks at the pipeline through examples of different kinds of `LiftResponse` and configurations.

You can get a great overview of the pipeline, including diagrams, from the Lift pipeline Wiki page at *http://www.assembla.com/spaces/liftweb/wiki/HTTP_Pipeline*.

See *https://github.com/LiftCookbook/cookbook_pipeline* for the source code that accompanies this chapter.

6.1. Debugging a Request

Problem

You want to debug a request and see what's arriving to your Lift application.

Solution

Add an `onBeginServicing` function in *Boot.scala* to log the request. For example:

```
LiftRules.onBeginServicing.append {
  case r => println("Received: "+r)
}
```

Discussion

The `onBeginServicing` call is called quite early in the Lift pipeline, before `S` is set up, and before Lift has the chance to 404 your request. The function signature it expects is `Req => Unit`. We're just logging, but the functions could be used for other purposes.

If you want to select only certain paths, you can. For example, to track all requests starting */paypal*:

```
LiftRules.onBeginServicing.append {
  case r @ Req("paypal" :: _), _, _) => println(r)
}
```

This pattern will match any request starting */paypal*, and we're ignoring the suffix on the request if any, and the type of request (e.g., GET, POST or so on).

There's also `LiftRules.early` which is called before `onBeginServicing`. It expects a `HTTPRequest => Unit` function, so is a little lower-level than the `Req` used in `onBeginServicing`. However, it will be called by all requests that pass through Lift. To see the difference, you could mark a request as something that the container should handle by itself:

```
LiftRules.liftRequest.append {
  case Req("robots" :: _, _, _) => false
}
```

Which this in place a request for *robots.txt* will be logged by `LiftRules.early` but won't make it to any of the other methods described in this recipe.

If you need access to state (e.g., `S`), use `earlyInStateful`, which is based on a `Box[Req]` not a `Req`:

```
LiftRules.earlyInStateful.append {
  case Full(r) => // access S here
  case _ =>
}
```

It's possible for your `earlyInStateful` function to be called twice. This will happen when a new session is being set up. You can prevent this by only matching on requests in a running Lift session:

```
LiftRules.earlyInStateful.append {
  case Full(r) if LiftRules.getLiftSession(r).running_? => // access S here
  case _ =>
}
```

Finally, there's also `earlyInStateless` which like `earlyInStateful` works on a `Box[Req]` but in other respects is the same as `onBeginServicing`. It is triggered after `early` and before `earlyInStateful`.

As a summary, the functions described in this recipe are called in this order:

- `LiftRules.early`
- `LiftRules.onBeginServicing`
- `LiftRules.earlyInStateless`

- `LiftRules.earlyInStateful`

See Also

If you need to catch the end of a request, there is also an `onEndServicing` which can be given functions of type `(Req, Box[LiftResponse]) => Unit`.

Recipe 6.4 describe show to force requests to be stateless.

6.2. Running Code when Sessions are Created (or Destroyed)

Problem

You want to carry out actions when a session is created or destroyed.

Solution

Make use of the hooks in `LiftSession`. For example, in *Boot.scala*:

```
LiftSession.afterSessionCreate ::=
 ( (s:LiftSession, r:Req) => println("Session created") )

LiftSession.onBeginServicing ::=
 ( (s:LiftSession, r:Req) => println("Processing request") )

LiftSession.onShutdownSession ::=
 ( (s:LiftSession) => println("Session going away") )
```

If the request path has been marked as being stateless via `LiftRules.statelessReqTest`, the above example would only execute the `onBeginServicing` functions.

Discussion

The hooks in `LiftSession` allow you to insert code at various points in the session lifecycle: when the session is created, at the start of servicing the request, after servicing, when the session is about to shutdown, at shutdown… the pipeline diagrams mentioned at the start of this chapter are a useful guide to these stages.

The full list of session hooks is:

- `onSetupSession` — this will be the first hook called when a session is created.
- `afterSessionCreate` — called after all `onSetupSession` functions have been called.
- `onBeginServicing` — at the start of the request processing.

- `onEndServicing` — and the end of request processing.
- `onAboutToShutdownSession` — called just before a session is shutdown, for example when a session expires or the Lift application is being shutdown.
- `onShutdownSession` — called after all `onAboutToShutdownSession` functions have been run.

If you are testing testing these hooks, you might want to make session expire faster than the 30 minutes of inactivity used by default in Lift. To do this, supply a millisecond value to `LiftRules.sessionInactivityTimeout`:

```
// 30 second inactivity timeout
LiftRules.sessionInactivityTimeout.default.set(Full(1000L * 30))
```

There are two other hooks in `LiftSession`: `onSessionActivate` and `onSessionPassivate`. These may be of use if you are working with a servlet container in distributed mode, and want to be notified when the servlet HTTP session is about to be serialized (passivated) and de-serialized (activated) between container instances. These hooks are rarely used.

Note that the Lift session is not the same as the HTTP Session. Lift bridges from the HTTP session to its own session management. This is described in some detail in *Exploring Lift*.

See Also

Session management is discussed in section 9.5 of *Exploring Lift*: *http://exploring.liftweb.net/*.

Recipe 6.4 shows how to run without state.

6.3. Run Code when Lift Shuts Down

Problem

You want to have some code executed when your Lift application is shutting down.

Solution

Append to `LiftRules.unloadHooks`.

```
LiftRules.unloadHooks.append( () => println("Shutting down") )
```

Discussion

You append functions of type `() => Unit` to `unloadHooks`, and these functions are run right at the end of the Lift handler, after sessions have been destroyed, Lift actors have been shutdown, and requests have finished being handled.

This is triggered, in the words of the Java servlet specification, "by the web container to indicate to a filter that it is being taken out of service".

See Also

Recipe 9.7 includes an example of using a unload hook.

6.4. Running Stateless

Problem

You want to force your application to be stateless at the HTTP level.

Solution

In *Boot.scala*:

```
LiftRules.enableContainerSessions = false
LiftRules.statelessReqTest.append { case _ => true }
```

All requests will now be treated as stateless. Any attempt to use state, such as via `SessionVar` for example, will trigger a warning in developer mode: "Access to Lift's statefull features from Stateless mode. The operation on state will not complete."

Discussion

HTTP session creation is controlled via `enableContainerSessions`, and applies for all requests. Leaving this value at the default (`true`) allows more fine-grained control over which requests are stateless.

Using `statelessReqTest` allows you to decide, based on the `StatelessReqTest` case class, if a request should be stateless (`true`) or not (`false`). For example:

```
def asset(file: String) =
  List(".js", ".gif", ".css").exists(file.endsWith)

LiftRules.statelessReqTest.append {
  case StatelessReqTest("index" :: Nil, httpReq) => true
  case StatelessReqTest(List(_, file),  _) if asset(file) => true
}
```

This example would only make the index page and any GIFs, JavaScript and CSS files stateless. The `httpReq` part is a `HTTPRequest` instance, allowing you to base the decision on the content of the request (cookies, user agent, etc).

Another option is `LiftRules.statelessDispatch` which allows you to register a function which returns a `LiftResponse`. This will be executed without a session, and convenient for REST-based services.

If you just need to mark an entry in Sitemap as being stateless, you can:

```
Menu.i("Stateless Page") / "demo" >> Stateless
```

A request for */demo* would be processed without state.

See Also

Chapter 4 contains recipes for REST-based services in Lift.

The Lift Wiki gives further details on the processing of stateless requests: *http://www.assembla.com/wiki/show/liftweb/Stateless_Requests*.

This stateless request control was introduced in Lift 2.2. The announcement on the mailing list gives more details: *http://bit.ly/lift-stateless*.

6.5. Catch Any Exception

Problem

You want a wrapper around all requests to catch exceptions and display something to the user.

Solution

Declare an exception handler in *Boot.scala*:

```
LiftRules.exceptionHandler.prepend {
  case (runMode, request, exception) =>
    logger.error("Failed at: "+request.uri)
    InternalServerErrorResponse()
}
```

In the above example, all exceptions for all requests at all run modes are being matched, causing an error to be logged and a 500 (internal server error) to be returned to the browser.

Discussion

The partial function you add to `exceptionHandler` needs to return a `LiftResponse` (i.e., something to send to the browser). The default behaviour is to return an `XhtmlResponse`, which in `Props.RunModes.Development` gives details of the exception, and in all other run modes simply says: "Something unexpected happened".

You can return any kind of `LiftResponse`, including `RedirectResponse`, `JsonResponse`, `XmlResponse`, `JavaScriptResponse` and so on.

The example above just sends a standard 500 error. That won't be very helpful to your users. An alternative is to render a custom message, but retain the 500 status code which will be useful for external site monitoring services if you use them:

```
LiftRules.exceptionHandler.prepend {
  case (runMode, req, exception) =>
    logger.error("Failed at: "+req.uri)
    val content = S.render(<lift:embed what="500" />, req.request)
    XmlResponse(content.head, 500, "text/html", req.cookies)
}
```

Here we are sending back a response with a 500 status code, but the content is the `Node` that results from running *src/main/webapp/template-hidden/500.html*. Create that file with the message you want to show to users:

```
<html>
<head>
  <title>500</title>
</head>
<body data-lift-content-id="main">
<div id="main" data-lift="surround?with=default;at=content">
  <h1>Something is wrong!</h1>
  <p>It's our fault - sorry</p>
</div>
</body>
</html>
```

You can also control what to send to clients when processing Ajax requests. In the following example, we're matching just on Ajax POST requests, and returning custom JavaScript to the browser:

```
import net.liftweb.http.js.JsCmds._

val ajax = LiftRules.ajaxPath

LiftRules.exceptionHandler.prepend {
  case (mode, Req(ajax :: _, _, PostRequest), ex) =>
    logger.error("Error handing ajax")
    JavaScriptResponse(Alert("Boom!"))
}
```

You could test out this handling code by creating an Ajax button that always produces an exception:

```
package code.snippet

import net.liftweb.util.Helpers._
import net.liftweb.http.SHtml

class ThrowsException {
  private def fail = throw new Error("not implemented")

  def render = "*" #> SHtml.ajaxButton("Press Me", () => fail)
}
```

This Ajax example will jump in before Lift's default behaviour for Ajax errors. The default is to retry the Ajax command three times (`LiftRules.ajaxRetryCount`), and then execute `LiftRules.ajaxDefaultFailure`, which will pop up a dialog saying: "The server cannot be contacted at this time"

See Also

Recipe 2.10 for how to create a custom 404 (not found) page.

6.6. Streaming Content

Problem

You want to stream content back to the web client.

Solution

Use `OutputStreamResponse`, passing it a function that will write to the `OutputStream` that Lift supplies.

In this example we'll stream all the integers from one, via a REST service:

```
package code.rest

import net.liftweb.http.{Req,OutputStreamResponse}
import net.liftweb.http.rest._

object Numbers extends RestHelper {

  // Convert a number to a String, and then to UTF-8 bytes
  // to send down the output stream.
  def num2bytes(x: Int) = (x + "\n") getBytes("utf-8")

  // Generate numbers using a Scala stream:
  def infinite = Stream.from(1).map(num2bytes)
```

```
    serve {
      case Req("numbers" :: Nil, _, _) =>
        OutputStreamResponse( out => infinite.foreach(out.write) )
    }
  }
```

Scala's `Stream` class is a way to generate a sequence with lazy evaluation. The values being produced by `infinite` are used as example data to stream back to the client.

Wire this into Lift in *Boot.scala*:

```
LiftRules.dispatch.append(Numbers)
```

Visiting *http://127.0.0.1:8080/numbers* will generate a 200 status code and start producing the integers from 1. The numbers are produced quite quickly, so you probably don't want to try that in your web browser, but instead from something that is easier to stop, such as cURL.

Discussion

`OutputStreamResponse` expects a function of type `OutputStream => Unit`. The `OutputStream` argument is the output stream to the client. This means the bytes we write to the stream are written to the client. In the above example…

```
OutputStreamResponse(out => infinite.foreach(out.write))
```

…we are making use of the `write(byte[])` method on Java's `OutputStream` (`out`), and sending it the `Array[Byte]` being generated from our `infinite` stream.

Be aware that `OutputStreamResponse` is executed outside of the scope of `S`. This means if you need to access anything in the session, do so outside of the function you pass to `OutputStreamResponse`.

For more control over status codes, headers and cookies, there are a variety of signatures for the `OutputStreamResponse` object. For the most control, create an instance of the `OutputStreamResponse` class:

```
case class OutputStreamResponse(
  out: (OutputStream) => Unit,
  size: Long,
  headers: List[(String, String)],
  cookies: List[HTTPCookie],
  code: Int)
```

Note that setting `size` to `-1` causes the `Content-length` header to be skipped.

There are two related types of response: `InMemoryResponse` and `StreamingResponse`.

InMemoryResponse

`InMemoryResponse` is useful if you have already assembled the full content to send to the client. The signature is straightforward:

```
case class InMemoryResponse(
  data: Array[Byte],
  headers: List[(String, String)],
  cookies: List[HTTPCookie],
  code: Int)
```

As an example, we can modify the recipe and force our `infinite` sequence of numbers to produce the first few numbers as an `Array[Byte]` in memory:

```
import net.liftweb.util.Helpers._

serve {
  case Req(AsInt(n) :: Nil, _, _) =>
    InMemoryResponse(infinite.take(n).toArray.flatten, Nil, Nil, 200)
}
```

The `AsInt` helper in Lift matches on an integer, meaning that a request starting with a number matches and we'll return that many numbers from the infinite sequence. We're not setting headers or cookies, and this request produces what you'd expect:

```
$ curl http://127.0.0.1:8080/3
1
2
3
```

StreamingResponse

`StreamingResponse` pulls bytes into the output stream. This contrasts with `Output StreamResponse`, where you are pushing data to the client.

Construct this type of response by providing a class with a `read` method that can be read from:

```
case class StreamingResponse(
  data: {def read(buf: Array[Byte]): Int},
  onEnd: () => Unit,
  size: Long,
  headers: List[(String, String)],
  cookies: List[HTTPCookie],
  code: Int)
```

Notice the use of a structural type for the `data` parameter. Anything with a matching `read` method can be given here, including `java.io.InputStream`-like objects, meaning `StreamingResponse` can act as a pipe from input to output. Lift pulls 8k chunks from your `StreamingResponse` to send to the client.

Your `data read` function should follow the semantics of Java IO and return "the total number of bytes read into the buffer, or -1 is there is no more data because the end of the stream has been reached".

See Also

The contract for Java IO is described at *http://docs.oracle.com/javase/6/docs/api/java/io/InputStream.html*.

6.7. Serving a File with Access Control

Problem

You have a file on disk, you want to allow a user to download it, but only if they are allowed to. If they are not allowed to, you want to explain why.

Solution

Use `RestHelper` to serve the file or an explanation page.

For example, suppose we have the file */tmp/important* and we only want selected requests to download that file from the */download/important* URL. The structure for that would be:

```
package code.rest

import net.liftweb.util.Helpers._
import net.liftweb.http.rest.RestHelper
import net.liftweb.http.{StreamingResponse, LiftResponse, RedirectResponse}
import net.liftweb.common.{Box, Full}
import java.io.{FileInputStream, File}

object DownloadService extends RestHelper {

  // (code explained below to go here)

  serve {
    case "download" :: Known(fileId) :: Nil Get req =>
      if (permitted) fileResponse(fileId)
      else Full(RedirectResponse("/sorry"))
  }
}
```

We are allowing users to download "known" files. That is, files which we approve of for access. We do this because opening up the file system to any unfiltered end-user input pretty much means your server will be compromised.

For our example, `Known` is checking a static list of names:

```
val knownFiles = List("important")

object Known {
 def unapply(fileId: String): Option[String] = knownFiles.find(_ == fileId)
}
```

For requests to these known resources, we convert the REST request into a `Box[LiftResponse]`. For permitted access we serve up the file:

```
private def permitted = scala.math.random < 0.5d

private def fileResponse(fileId: String): Box[LiftResponse] = for {
    file <- Box !! new File("/tmp/"+fileId)
    input <- tryo(new FileInputStream(file))
 } yield StreamingResponse(input,
    () => input.close,
    file.length,
    headers=Nil,
    cookies=Nil,
    200)
```

If no permission is given, the user is redirected to */sorry.html*.

All of this is wired into Lift in *Boot.scala* with:

```
LiftRules.dispatch.append(DownloadService)
```

Discussion

By turning the request into a `Box[LiftResponse]` we are able to serve up the file, send the user to a different page, and also allow Lift to handle the 404 (`Empty`) cases.

If we added a test to see if the file existed on disk in `fileResponse` that would cause the method to evaluate to `Empty` for missing files, which triggers a 404. As the code stands, if the file does not exist, the `tryo` would give us a `Failure` which would turn into a 404 error with a body of "/tmp/important (No such file or directory)".

Because we are testing for known resources via the `Known` extractor as part of the pattern for */download/*, unknown resources will not be passed through to our `File` access code. Again, Lift will return a 404 for these.

Guard expressions can also be useful for these kinds of situations:

```
serve {
  case "download" :: Known(id) :: Nil Get _ if permitted => fileResponse(id)
  case "download" :: _ Get req => RedirectResponse("/sorry")
}
```

You can mix and match extractors, guards and conditions in your response to best fit the way you want the code to look and work.

See Also

Chatper 24: Extractors from *Programming in Scala*: *http://www.artima.com/pins1ed/extractors.html*.

6.8. Access Restriction by HTTP Header

Problem

You need to control access to a page based on the value of a HTTP header.

Solution

Use a custom `If` in SiteMap:

```
val HeaderRequired = If(
  () => S.request.map(_.header("ALLOWED") == Full("YES")) openOr false,
  "Access not allowed"
)

// Build SiteMap
val entries = List(
  Menu.i("Header Required") / "header-required" >> HeaderRequired
)
```

In this example *header-required.html* can only be viewed if the request includes a HTTP header called `ALLOWED` with a value of `YES`. Any other request for the page will be redirected with a Lift error notice of "Access not allowed".

This can be tested from the command line using a tool like cURL:

```
$ curl http://127.0.0.1:8080/header-required.html -H "ALLOWED:YES"
```

Discussion

The `If` test ensures the `() => Boolean` function you supply as a first argument returns `true` before the page it applies to is shown. In this example we'll get `true` if the request contains a header called "ALLOWED", and the optional value of that header is `Full("YES")`. This is a `LocParam` (location parameter) which modifies the site map item. It can be appended to any menu items you want using the `>>` method.

Note that without the header, the test will be false. This will mean with link to the page will not appear the menu generated by `Menu.builder`.

The second argument to the `If()` is what Lift does if the test isn't true when the user tries to access the page. It's a `() => LiftResponse` function. This means you can return whatever you like, including redirects to other pages. In the example we are making use

of a convenient implicit conversation from a `String` ("Access not allowed") to a notice with a redirection that will take the user to the home page.

If you visit the page without a header, you'll see a notice saying "Access not allowed". This will be the home page of the site, but that's just the default. You can request that Lift show a different page by setting `LiftRules.siteMapFailRedirectLocation` in *Boot.scala*:

```
LiftRules.siteMapFailRedirectLocation = "static" :: "permission" :: Nil
```

If you then try to access *header-required.html* without the header set, you'll be redirected to */static/permission* and shown the content of whatever you put in that page.

See Also

The Lift wiki gives a summary of Lift's Site Map and the tests you can include in site map entries: *https://www.assembla.com/wiki/show/liftweb/SiteMap*.

There are further details in chapter 7 of *Exploring Lift* at *http://exploring.liftweb.net*, and "SiteMap and access control", chapter 7 of *Lift in Action* (Perrett, 2012, Manning Publications Co.).

6.9. Accessing `HttpServletRequest`

Problem

To satisfy some API you need access to the `HttpServletRequest`.

Solution

Cast `S.request`:

```
import net.liftweb.http.S
import net.liftweb.http.provider.servlet.HTTPRequestServlet
import javax.servlet.http.HttpServletRequest

def servletRequest: Box[HttpServletRequest] = for {
  req <- S.request
  inner <- Box.asA[HTTPRequestServlet](req.request)
} yield inner.req
```

You can then make your API call:

```
servletRequest.foreach { r => yourApiCall(r) }
```

Discussion

Lift abstracts away from the low-level HTTP request, and from the details of the servlet container your application is running in. However, it's reassuring to know, if you absolutely need it, there is a way to get back down to the low-level.

Note that the results of `servletRequest` is a `Box` because there might not be a request when you evaluate `servletRequest` — or you might one day port to a different deployment environment and not be running on a standard Java servlet container.

As your code will have a direct dependency on the Java Servlet API, you'll need to include this dependency in your SBT build:

```
"javax.servlet" % "servlet-api" % "2.5" % "provided"
```

6.10. Force HTTPS Requests

Problem

You want to ensure clients are using HTTPs.

Solution

Add an `earlyResponse` function in *Boot.scala* redirecting HTTP requests to HTTPS equivalents. For example:

```
LiftRules.earlyResponse.append { (req: Req) =>
  if (req.request.scheme != "https") {
    val uriAndQuery = req.uri + (req.request.queryString.map(s => "?"+s) openOr
"")
    val uri = "https://%s%s".format(req.request.serverName, uriAndQuery)
    Full(PermRedirectResponse(uri, req, req.cookies: _*))
  }
  else Empty
}
```

Discussion

The `earlyResponse` call is called early on in the Lift pipeline. It is used to execute code before a request is handled and, if required, exit the pipeline and return a response. The function signature expected is `Req => Box[LiftResponse]`.

In this example we are testing for a request that is not "https", and then formulating a new URL that starts "https" and appends to it the rest of the original URL and any query parameters. With this created, we return a redirections to the new URL, along with any cookies that were set.

By evaluating to `Empty` for other requests (i.e., HTTPS requests), Lift will continue passing the request through the pipeline as usual.

The ideal method to ensure requests are served using the correct scheme would be via web server configuration, such as Apache or Nginx. This isn't possible in some cases, such as when your application is deployed to a PaaS such as CloudBees.

Amazon Load Balancer

For Amazon Elastic Load Balancer note that you need to use `X-Forwarded-Proto` header to detect HTTPS. As mentioned in their *Overview of Elastic Load Balancing* document, "Your server access logs contain only the protocol used between the server and the load balancer; they contain no information about the protocol used between the client and the load balancer."

In this situation modify the above test from `req.request.scheme != "https"` to:

```
req.header("X-Forwarded-Proto") != Full("https")
```

See Also

The *Overview of Elastic Load Balancing* can be found at: *http://docs.amazonwebservices.com/ElasticLoadBalancing/latest/DeveloperGuide/arch-loadbalancing.html*.

CHAPTER 7

Relational Database Persistence with Record and Squeryl

Squeryl is an object-relational mapping library. It converts Scala classes into tables, rows and columns in a relational database, and provides a way to write SQL-like queries that are type-checked by the Scala compiler. The Lift Squeryl Record module integrates Squeryl with Record, meaning your Lift application can use Squeryl to store and fetch data while making use of the features of record, such as data validation.

The code in this chapter can be found at: *https://github.com/LiftCookbook/cookbook_squeryl*.

7.1. Configuring Squeryl and Record

Problem

You want to configure your Lift application to use Squeryl and Record.

Solution

Include the Squeryl-Record dependency in your build, and in *Boot.scala* provide a database connection function to `SquerylRecord.initWithSquerylSession`.

For example, to configure Squeryl with PostgreSQL, modify *build.sbt* to add two dependencies, one for Squeryl-Record and one for the database driver:

```
libraryDependencies ++= {
  val liftVersion = "2.5-RC5"
  Seq(
    "net.liftweb" %% "lift-webkit" % liftVersion,
    "net.liftweb" %% "lift-squeryl-record" % liftVersion,
    "postgresql" % "postgresql" % "9.1-901.jdbc4"
```

```
    ...
    )
}
```

This will give you access to Squeryl version 0.9.5-6.

In *Boot.scala* we define a connection and register it with Squeryl:

```
Class.forName("org.postgresql.Driver")

def connection = DriverManager.getConnection(
  "jdbc:postgresql://localhost/mydb",
  "username", "password")

SquerylRecord.initWithSquerylSession(
  Session.create(connection, new PostgreSqlAdapter) )
```

All Squeryl queries need to run in the context of a transaction. One way to provide a transaction is to configure a transaction around all HTTP requests. This is also configured in *Boot.scala*:

```
import net.liftweb.squerylrecord.RecordTypeMode._
import net.liftweb.http.S
import net.liftweb.util.LoanWrapper

S.addAround(new LoanWrapper {
  override def apply[T](f: => T): T = {
    val result = inTransaction {
      try {
        Right(f)
      } catch {
        case e: LiftFlowOfControlException => Left(e)
      }
    }

    result match {
      case Right(r) => r
      case Left(exception) => throw exception
    }
  }
})
```

This arranges for requests to be handled in the `inTransaction` scope. As Lift uses an exception for redirects, we catch this exception and throw it after the transaction completes, avoiding rollbacks after a `S.redirectTo` or similar.

Discussion

You can use any JVM persistence mechanism with Lift. What Lift Record provides is a light interface around persistence with bindings to Lift's CSS transforms, screens and wizards. Squeryl-Record is a concrete implementation to connect Record with Squeryl.

This means you can use standard Record objects, which are effectively your schema, with Squeryl and write queries which are validated at compile time.

Plugging into Squeryl means initializing Squeryl's session management, which allows us to wrap queries in Squeryl's `transaction` and `inTransaction` functions. The difference between these two calls is that `inTransaction` will start a new transaction if one doesn't exist, whereas `transaction` always creates a new transaction.

By ensuring a transaction is available for all HTTP requests via `addAround`, we can write queries in Lift and for the most part not have to establish transactions ourselves unless we want to. For example:

```
import net.liftweb.squerylrecord.RecordTypeMode._
val r = myTable.insert(MyRecord.createRecord.myField(aValue))
```

In this recipe, the `PostgreSqlAdapter` is being used. Squeryl also supports: `OracleAdapter`, `MySQLInnoDBAdapter` and `MySQLAdapter`, `MSSQLServer`, `H2Adapter`, `DB2Adapter` and `DerbyAdapter`.

See Also

The Squeryl *Getting Started Guide* links to more information about session management and configuration: *http://squeryl.org/getting-started.html*.

See Recipe 7.2 for configuring connections via Java Naming and Directory Interface (JNDI).

7.2. Using a JNDI Datasource

Problem

You want to use a *Java Naming and Directory Interface* (JNDI) data source for your Record+Squeryl Lift application.

Solution

In *Boot.scala* call `initWithSquerylSession` with a `DataSource` looked up from the JDNI context:

```
import javax.sql.DataSource
val ds = new InitialContext().
  lookup("java:comp/env/jdbc/mydb").asInstanceOf[DataSource]

SquerylRecord.initWithSquerylSession(
  Session.create(ds.getConnection(), new MySQLAdapter) )
```

…replacing `mydb` with the name given to your database in your JNDI configuration, and replacing `MySQLAdapter` with the appropriate adapter for the database you are using.

Discussion

JNDI is a service provided by the web container (e.g., Jetty, Tomcat) which allows you to configure a database connection in the container and then refer the connection by name in your application. One advantage of this is that you can avoid including database credentials to your Lift source base.

The configuration of JNDI is different for each container, and may vary with versions of the container you use. The *See Also* section includes links to the documentation pages for popular containers.

Some environments may also require that you to reference the JNDI resource in your *src/main/webapp/WEB-INF/web.xml* file:

```
<resource-ref>
 <res-ref-name>jdbc/mydb</res-ref-name>
 <res-type>javax.sql.DataSource</res-type>
 <res-auth>Container</res-auth>
</resource-ref>
```

See Also

Resources for JDNI configuration include:

- An example on the Lift Wiki for Apache and Jetty configuration at:*http://www.assembla.com/spaces/liftweb/wiki/Apache_and_Jetty_Configuration*.
- The documentation for Jetty gives examples for various databases: *http://www.eclipse.org/jetty/documentation/current/jndi-datasource-examples.html*.
- For Tomcat, the JNDI configuration guide is: *http://tomcat.apache.org/tomcat-7.0-doc/jndi-resources-howto.html#JDBC_Data_Sources*.

7.3. One-to-Many Relationship

Problem

You want to model a one-to-many relationship, such as a satellite belonging to a single planet, but a planet possibly having many satellites.

Solution

Use Squeryl's `oneToManyRelation` in your schema, and on your Lift model include a reference from the satellite to the planet.

The objective is to model the relationship as shown in Figure 7-1.

Figure 7-1. One planet may have many satellites, but a satellite orbits just one planet.

In code:

```
package code.model

import org.squeryl.Schema
import net.liftweb.record.{MetaRecord, Record}
import net.liftweb.squerylrecord.KeyedRecord
import net.liftweb.record.field.{StringField, LongField}
import net.liftweb.squerylrecord.RecordTypeMode._

object MySchema extends Schema {

  val planets = table[Planet]
  val satellites = table[Satellite]

  val planetToSatellites = oneToManyRelation(planets, satellites).
    via((p,s) => p.id === s.planetId)

  on(satellites) { s =>
    declare(s.planetId defineAs indexed("planet_idx"))
  }

  class Planet extends Record[Planet] with KeyedRecord[Long] {
    override def meta = Planet
    override val idField = new LongField(this)
    val name = new StringField(this, 256)
    lazy val satellites = MySchema.planetToSatellites.left(this)
  }

  object Planet extends Planet with MetaRecord[Planet]

  class Satellite extends Record[Satellite] with KeyedRecord[Long] {
     override def meta = Satellite
     override val idField = new LongField(this)
     val name = new StringField(this, 256)
     val planetId = new LongField(this)
     lazy val planet = MySchema.planetToSatellites.right(this)
  }
```

```
    object Satellite extends Satellite with MetaRecord[Satellite]

}
```

This schema defines the two tables based on the Record classes, as `table[Planet]` and `table[Satellite]`. It establishes a `oneToManyRelation` based on (`via`) the `planetId` in the satellite table.

This gives Squeryl the information it needs to produce a foreign key to constrain the `planetId` to reference an existing record in the planet table. This can be seen in the schema generated by Squeryl. We can print the schema in *Boot.scala* with…

```
inTransaction {
  code.model.MySchema.printDdl
}
```

…which will print:

```
-- table declarations :
create table Planet (
    name varchar(256) not null,
    idField bigint not null primary key auto_increment
  );
create table Satellite (
    name varchar(256) not null,
    idField bigint not null primary key auto_increment,
    planetId bigint not null
  );
-- indexes on Satellite
create index planet_idx on Satellite (planetId);
-- foreign key constraints :
alter table Satellite add constraint SatelliteFK1 foreign key (planetId)
  references Planet(idField);
```

An index called `planet_idx` is declared on the `planetId` field to improve query performance during joins.

Finally we make use of the `planetToSatellites.left` and `right` methods to establish lookup queries as `Planet.satellites` and `Satellite.planet`. We can demonstrate their use by inserting example data and running the queries:

```
inTransaction {
  code.model.MySchema.create

  import code.model.MySchema._

  val earth = planets.insert(Planet.createRecord.name("Earth"))
  val mars = planets.insert(Planet.createRecord.name("Mars"))

  // .save as a short-hand for satellite.insert when we don't need
  // to immediately reference the record (save returns Unit).
```

```
    Satellite.createRecord.name("The Moon").planetId(earth.idField.is).save
    Satellite.createRecord.name("Phobos").planetId(mars.idField.is).save

    val deimos = satellites.insert(
      Satellite.createRecord.name("Deimos").planetId(mars.idField.is) )

    println("Deimos orbits: "+deimos.planet.single.name.is)
    println("Moons of Mars are: "+mars.satellites.map(_.name.is))

  }
```

Running this code produce the output:

```
Deimos orbits: Mars
Moons of Mars are: List(Phobos, Deimos)
```

In this example code we're calling `deimos.planet.single` which returns one result or will thrown an exception if the associated planet was not found. `headOption` is the safer way if there's a chance the record will not be found, as it will evaluate to `None` or `Some[Planet]`.

Discussion

The `planetToSatellites.left` method is not a simple collection of `Satellite` objects. It's a Squeryl `Query[Satellite]`, meaning you can treat it like any other kind of `Query able[Satellite]`. For example we could ask for those satellites of a planet that are alphabetically after "E", which for Mars would match "Phobos":

```
mars.satellites.where(s => s.name gt "E").map(_.name)
```

The `left` method result is also a `OneToMany[Satellite]` which adds the following methods:

- `assign` — adds a new relationship, but does not update the database.
- `associate` — which is like `assign` but does update the database.
- `deleteAll` — to remove the relationships.

The `assign` call gives the satellite the relationship to the planet:

```
val express = Satellite.createRecord.name("Mars Express")
mars.satellites.assign(express)
express.save
```

The next time we query `mars.satellites` we will find the *Mars Express* orbiter.

A call to `associate` would go one step further for us, making Squeryl insert or update the satellite automatically:

```
val express = Satellite.createRecord.name("Mars Express")
mars.satellites.associate(express)
```

The third method, `deleteAll` does what it sounds like it should do. It would execute the following SQL and return the number of rows removed:

```
delete from Satellite
```

The right side of the one-to-many also has additional methods added by `ManyToOne[Planet]` of `assign` and `delete`. Be aware that to delete the "one" side of a many-to-one, anything assigned to record will need to have been deleted already to avoid a database constraint error that would arise from, for example, leaving satellites referencing non-existent planets.

As `left` and `right` are queries, it means each time you use them you'll be sending a new query to the database. Squeryl refer to these forms as *stateless relations*.

The *stateful* versions of `left` and `right` look like this:

```
class Planet extends Record[Planet] with KeyedRecord[Long] {
 ...
 lazy val satellites : StatefulOneToMany[Satellite] =
   MySchema.planetToSatellites.leftStateful(this)
}

class Satellite extends Record[Satellite] with KeyedRecord[Long] {
  ...
  lazy val planet : StatefulManyToOne[Planet] =
    MySchema.planetToSatellites.rightStateful(this)
}
```

This change means the results of `mars.satellites` will be cached. Subsequent calls on that instance of a `Planet` won't trigger a round trip to the database. You can still `associate` new records or `deleteAll` records which will work as you expect, but if a relationship is added or changed elsewhere you'll need to call `refresh` on the relation to see the change.

Which version should you use? That will depend on your application, but you can use both in the same record if you need to.

See Also

Squeryl relations are documented at *http://squeryl.org/relations.html*.

7.4. Many-to-Many Relationship

Problem

You want to model a many-to-many relationship, such as a planet being visted by many space probes, but a space probe also visiting many planets.

Solution

Use Squeryl's `manyToManyRelation` in your schema, and implement a record to hold the join between the two sides of the relationship. Figure 7-2 shows the structure we will create in this recipe, where `Visit` is the record that will connect each many to the other many.

Figure 7-2. Many-to-many: Jupiter was visited by Juno and Voyager 1; Saturn was only visited by Voyager 1.

The schema is defined in terms of two tables, one for planets and one for space probes, plus a relationship between the two based on a third class, called `Visit`:

```
package code.model

import org.squeryl.Schema
import net.liftweb.record.{MetaRecord, Record}
import net.liftweb.squerylrecord.KeyedRecord
import net.liftweb.record.field.{IntField, StringField, LongField}
import net.liftweb.squerylrecord.RecordTypeMode._
import org.squeryl.dsl.ManyToMany

object MySchema extends Schema {

  val planets = table[Planet]
  val probes = table[Probe]

  val probeVisits = manyToManyRelation(probes, planets).via[Visit] {
    (probe, planet, visit) =>
      (visit.probeId === probe.id, visit.planetId === planet.id)
  }

  class Planet extends Record[Planet] with KeyedRecord[Long] {
    override def meta = Planet
    override val idField = new LongField(this)
    val name = new StringField(this, 256)
    lazy val probes : ManyToMany[Probe,Visit] =
      MySchema.probeVisits.right(this)
  }

  object Planet extends Planet with MetaRecord[Planet]

  class Probe extends Record[Probe] with KeyedRecord[Long] {
    override def meta = Probe
    override val idField = new LongField(this)
    val name = new StringField(this, 256)
```

```
    lazy val planets : ManyToMany[Planet,Visit] =
      MySchema.probeVisits.left(this)
  }

  object Probe extends Probe with MetaRecord[Probe]

  class Visit extends Record[Visit] with KeyedRecord[Long] {
    override def meta = Visit
    override val idField = new LongField(this)
    val planetId = new LongField(this)
    val probeId = new LongField(this)
  }

  object Visit extends Visit with MetaRecord[Visit]
}
```

In *Boot.scala* we can print out the schema…

```
inTransaction {
  code.model.MySchema.printDdl
}
```

…which will produce something like this, depending on the database in use:

```
-- table declarations :
create table Planet (
    name varchar(256) not null,
    idField bigint not null primary key auto_increment
  );
create table Probe (
    name varchar(256) not null,
    idField bigint not null primary key auto_increment
  );
create table Visit (
    idField bigint not null primary key auto_increment,
    planetId bigint not null,
    probeId bigint not null
  );
-- foreign key constraints :
alter table Visit add constraint VisitFK1 foreign key (probeId)
  references Probe(idField);
alter table Visit add constraint VisitFK2 foreign key (planetId)
  references Planet(idField);
```

Notice that the `visit` table will hold a row for each relationship between a `planetId` and `probeId`.

`Planet.probes` and `Probe.planets` provide an `associate` method to establish a new relationship. For example, we can establish a set of planets and probes…

```
val jupiter = planets.insert(Planet.createRecord.name("Jupiter"))
val saturn = planets.insert(Planet.createRecord.name("Saturn"))
val juno = probes.insert(Probe.createRecord.name("Juno"))
val voyager1 = probes.insert(Probe.createRecord.name("Voyager 1"))
```

…and then connect them:

```
juno.planets.associate(jupiter)
voyager1.planets.associate(jupiter)
voyager1.planets.associate(saturn)
```

We can also use `Probe.planets` and `Planet.probes` as a query to look up the associations. To access all the probes that had visited each planet in a snippet, we can write this:

```
package code.snippet

class ManyToManySnippet {
  def render =
    "#planet-visits" #> planets.map { planet =>
      ".planet-name *" #> planet.name.is &
      ".probe-name *" #> planet.probes.map(_.name.is)
    }
}
```

The snippet could be combined with a template like this:

```
<div data-lift="ManyToManySnippet">
  <h1>Planet facts</h1>
  <div id="planet-visits">
    <p>
      <span class="planet-name">Name will be here</span> was visited by:
    </p>
    <ul>
      <li class="probe-name">Probe name goes here</li>
    </ul>
  </div>
</div>
```

The top half of Figure 7-3 gives an example of the output from this snippet and template.

Discussion

The Squeryl DSL `manyToManyRelation(probes, planets).via[Visit]` is the core element here connecting our `Planet`, `Probe` and `Visit` records together. It allows us to access the "left" and "right" sides of the relationship in our model as `Probe.planets` and `Planet.probes`.

As with Recipe 7.3 for one-to-many relationships, the left and right sides are queries. When you ask for `Planet.probes` the database is queried appropriately with a join on the `Visit` records:

```
Select
  Probe.name,
  Probe.idField
From
  Visit,
  Probe
```

```
Where
  (Visit.probeId = Probe.idField) and (Visit.planetId = ?)
```

Also as described in Recipe 7.3 there are stateful variants of `left` and `right` to cache the query results.

In the data we inserted into the database, we did not have to mention `Visit`. The Squeryl `manyToManyRelation` has enough information to know how to insert a visit as the relationship. Incidentally, it doesn't matter which way round we make the calls in a many-to-many relationship. The following two expressions are equivalent and result in the same database structure:

```
juno.planets.associate(jupiter)
// ..or..
jupiter.probes.associate(juno)
```

You might even wonder why we had to bother with defining a `Visit` record at all, but there are benefits in doing so. For example, you can attach additional information onto the join table, such as the year the probe visited a planet.

To do this, we modify the record to include the additional field:

```
class Visit extends Record[Visit] with KeyedRecord[Long] {
  override def meta = Visit
  override val idField = new LongField(this)
  val planetId = new LongField(this)
  val probeId = new LongField(this)
  val year = new IntField(this)
}
```

`Visit` is still a container for the `planetId` and `probeId` references, but we also have a plain integer holder for the year of the visit.

To record a visit year, we need the `assign` method provided by `ManyToMany[T]`. This will establish the relationship, but not change the database. Instead, it returns the `Visit` instance which we can change and then store in the database:

```
probeVisits.insert(voyager1.planets.assign(saturn).year(1980))
```

The return type of `assign` in this case is `Visit`, and `Visit` has a `year` field. Inserting the `Visit` record via `probeVisits` will create a row in the table for visits.

To access this extra information on the `Visit` object, you can make use of a couple of methods provided by `ManyToMany[T]`:

- `associations` — a query returning the `Visit` objects related to the `Planet.probes` or `Probe.planets`.
- `associationMap` — a query returning pairs of `(Planet,Visit)` or `(Probe,Visit)` depending on which side of the join you call it on (probes or planets).

For example, in a snippet we could list all the space probes, and for each probe show the planet it visited and what year it was there. The snippet would look like this:

```
"#probe-visits" #> probes.map { probe =>
  ".probe-name *" #> probe.name.is &
  ".visit" #> probe.planets.associationMap.collect {
    case (planet, visit) =>
      ".planet-name *" #> planet.name.is &
      ".year" #> visit.year.is
    }
}
```

We are using `collect` here rather than `map` just to match the `(Planet,Visit)` tuple and give the values meaningful names. You could also use `(for { (planet, visit) <- probe.planets.associationMap } yield ...)` if you prefer.

The lower-half of Figure 7-3 demonstrates how this snippet would render when combined with the following template:

```
<h1>Probe facts</h1>

<div id="probe-visits">
  <p><span class="probe-name">Space craft name</span> visited:</p>
  <ul>
    <li class="visit">
      <span class="planet-name">Name here</span> in <span class="year">n</span>
    </li>
  </ul>
</div>
```

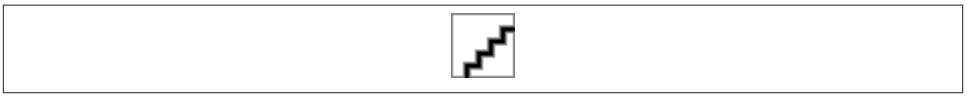


Figure 7-3. Example output from using the many-to-many features in this recipe.

To remove an association you have access to `dissociate` and `dissociateAll` on the `left` and `right` queries. To remove a single association:

```
val numRowsChanged = juno.planets.dissociate(jupiter)
```

This would be executed in SQL as:

```
delete from Visit
where
  probeId = ? and planetId = ?
```

To remove all the associations:

```
val numRowsChanged = jupiter.probes.dissociateAll
```

The SQL for this is:

```
delete from Visit
where
  Visit.planetId = ?
```

What you cannot do is delete a `Planet` or `Probe` if that record still has associations in the `Visit` relationship. What you'd get is a referential integrity exception thrown. Instead, you'll need to `dissociateAll` first:

```
jupiter.probes.dissociateAll
planets.delete(jupiter.id)
```

However, if you do want *cascading deletes* you can achieve this by overriding the default behaviour in your schema:

```
// To automatically remove probes when we remove planets:
probeVisits.rightForeignKeyDeclaration.constrainReference(onDelete cascade)

// To automatically remove planets when we remove probes:
probeVisits.leftForeignKeyDeclaration.constrainReference(onDelete cascade)
```

This is part of the schema, in that it will change the table constraints, with `printDdl` producing this (depending on the database you use):

```
alter table Visit add constraint VisitFK1 foreign key (probeId)
  references Probe(idField) on delete cascade;

alter table Visit add constraint VisitFK2 foreign key (planetId)
  references Planet(idField) on delete cascade;
```

See Also

Recipe 7.3, on one-to-many relationships, discusses `leftStateful` and `rightStateful` relations, which are also applicable for many-to-many relationships.

Foreign keys, cascading deletes, are described at: *http://squeryl.org/relations.html*.

7.5. Adding Validation to a Field

Problem

You want to add validation to a field in your model, so that users are informed of missing fields or fields that aren't acceptable to your application.

Solution

Override the `validations` method on your field and provide one or more validation functions.

As an example, imagine we have a database of planets and we want to ensure any new planets entered by users have names of at least five characters. We add this as a validation on our record:

```
class Planet extends Record[Planet] with KeyedRecord[Long]   {
   override def meta = Planet
   override val idField = new LongField(this)

   val name = new StringField(this, 256) {
     override def validations =
       valMinLen(5, "Name too short") _ :: super.validations
   }

 }
```

To check the validation, in our snippet we call `validate` on the record which will return all the errors for the record:

```
package code
package snippet

import net.liftweb.http.{S,SHtml}
import net.liftweb.util.Helpers._

import model.MySchema._

class ValidateSnippet {

  def render = {

    val newPlanet = Planet.createRecord

    def validateAndSave() : Unit = newPlanet.validate match {
      case Nil =>
        planets.insert(newPlanet)
        S.notice("Planet '%s' saved" format newPlanet.name.is)

      case errors =>
        S.error(errors)
    }

    "#planetName" #> newPlanet.name.toForm &
    "type=submit" #> SHtml.onSubmitUnit(validateAndSave)
  }
}
```

When the snippet runs we render the `Planet.name` field and wire up a submit button to call the `validateAndSave` method.

If the `newPlanet.validate` call indicates there are no errors (`Nil`), we can save the record and inform the user via a notice. If there are errors, we render all of them with `S.error`.

The corresponding template could be:

```
<html>
<head>
  <title>Planet Name Validation</title>
</head>
<body data-lift-content-id="main">
<div id="main" data-lift="surround?with=default;at=content">
  <h1>Add a planet</h1>

  <div data-lift="Msgs?showAll=false">
    <lift:notice_class>noticeBox</lift:notice_class>
  </div>

  <p>
    Planet names need to be at least 5 characters long.
  </p>

  <form class="ValidateSnippet?form">

    <div>
      <label for="planetName">Planet name:</label>
      <input id="planetName" type="text"></input>
      <span data-lift="Msg?id=name_id&errorClass=error">
        Msg to appear here
      </span>
    </div>

    <input type="submit"></input>

  </form>

</div>
</body>
</html>
```

In this template the error message is shown next to the `input` field, styled with a CSS class of `errorClass`. The success notice is shown near the top of the page, just below the `<h1>` heading, using a styled called `noticeBox`.

Discussion

The built-in validations are:

- `valMinLen` — validate that a string is at least a given length, as shown above.
- `valMaxLen` — validate that a string is not above a given length.
- `valRegex` — validate a string matches the given pattern.

An example of regular expression validation on a field would be:

```
import java.util.regex.Pattern

val url = new StringField(this, 1024) {
  override def validations =
    valRegex( Pattern.compile("^https?://.*"),
              "URLs should start http:// or https://") _ ::
    super.validations
}
```

The list of errors from `validate` are of type `List[FieldError]`. The `S.error` method accepts this list and registered each validation error message so it can be shown on the page. It does this by associating the message with an ID for the field, allowing you to pick out just the errors for an individual field, as we do in this recipe. The ID is stored on the field, and in the case of `Planet.name` it is available as `Planet.name.uniqueFieldId`. It's a `Box[String]` with a value of `Full("name_id")`. It is this `name_id` value that we used in the `lift:Msg?id=name_id&errorClass=error` markup to pick out just the error for this field.

You don't have to use `S.error` to display validation messages. You can roll your own display code, making use of the `FieldError` directly. As you can see from the source for `FieldError`, the error is available as a `msg` property:

```
case class FieldError(field: FieldIdentifier, msg: NodeSeq) {
  override def toString = field.uniqueFieldId + " : " + msg
}
```

See Also

The `BaseField.scala` class in the Lift source code contains the definition of the built-in `StringValidators`. Find the source at: *https://github.com/lift/framework/blob/master/core/util/src/main/scala/net/liftweb/util/BaseField.scala*.

Chapter 3 describes form processing, notices and errors.

7.6. Custom Validation Logic

Problem

You want to provide your own validation logic and apply it to a field in a record.

Solution

Implement a function from the type of the field to `List[FieldError]`, and reference the function in the `validations` on the field.

Here's an example: we have a database of planets, and when a user enters a new planet, we want the name to be unique. The name of the planet is a `String`, so we need to provide a function from `String => List[FieldError]`.

With the validation function defined (`valUnique`, below), we include it in the list of `validations` on the `name` field:

```
import net.liftweb.util.FieldError

class Planet extends Record[Planet] with KeyedRecord[Long] {
  override def meta = Planet
  override val idField = new LongField(this)

  val name = new StringField(this, 256) {
    override def validations =
      valUnique("Planet already exists") _ ::
      super.validations
  }

  private def valUnique(errorMsg: => String)(name: String): List[FieldError] =
    Planet.unique_?(name) match {
      case true => FieldError(this.name, errorMsg) :: Nil
      case false => Nil
    }
}

object Planet extends Planet with MetaRecord[Planet] {
  def unique_?(name: String) = from(planets) { p =>
    where(lower(p.name) === lower(name)) select(p)
  }.isEmpty
}
```

The validation is triggered just like any other validation, as described in Recipe 7.5.

Discussion

By convention validation functions have two argument lists: the first for the error message; the second to receive the value to validate. This allows you to easily re-use your validation function on other fields. For example, if you wanted to validate that satellites have a unique name, you could use exactly the same function but provide a different error message.

The `FieldError` you return needs to know the field it applies to as well as the message to display. In the example the field is `name`, but we've used `this.name` to avoid confusion with the `name` parameter passed into the `valUnique` function.

The example code has used text for the error message, but there is a variation of `FieldError` that accepts `NodeSeq`. This allows you to produce safe markup as part of the error if you need to. For example:

```
FieldError(this.name, <p>Please see <a href="/policy">our name policy</a></p>)
```

For internationalisation, you may prefer to pass in a key to the validation function, and resolve it via `S.?`:

```
val name = new StringField(this, 256) {
    override def validations =
      valUnique("validation.planet") _ ::
      super.validations
  }

// ...combined with...

private def valUnique(errorKey: => String)(name: String): List[FieldError] =
  Planet.unique_?(name) match {
    case false => FieldError(this.name, S ? errorKey) :: Nil
    case true => Nil
  }
```

See Also

Recipe 7.5 discusses field validation and the built-in validations.

Text localisation is discussed on the Lift wiki at: *https://www.assembla.com/wiki/show/liftweb/Localization*.

7.7. Modify a Field Value Before it is Set

Problem

You want to modify the value of a field before storing it, for example to clean a value by removing leading and trailing whitespace.

Solution

Override `setFilter` and provide a list of functions to apply to the field.

To remove leading and trailing whitespace entered by the user, the field would use the `trim` filter:

```
val name = new StringField(this, 256) {
   override def setFilter = trim _ :: super.setFilter
}
```

Discussion

The built-in filters are:

- `crop` — enforces the field's min and max length by truncation.

- `trim` — applies `String.trim` to the field value.
- `toUpper` and `toLower` — change the case of the field value.
- `removeRegExChars` — removes matching regular expression characters.
- `notNull` — coverts null values to an empty string.

Filters are run before validation. This means if you have a minimum length validation and the trim filter, for example, users cannot pass the validation test by just including spaces on the end of the value they enter.

A filter for a `String` field would be of type `String => String`, and the `setFilter` function expects a `List` of these. Knowing this, it's straight-forward to write custom filters. For example, here's is a filter that applies a simple form of title case on our `name` field:

```
def titleCase(in: String) =
 in.split("\\s").
 map(_.toList).
 collect {
   case x :: xs  => (Character.toUpperCase(x).toString :: xs).mkString
 }.mkString(" ")
```

This function is splitting the input string on spaces, converting each word into a list of characters, converting the first character into upper case, and then gluing the strings back together.

When installed as a filter…

```
val name = new StringField(this, 256) {
   override def setFilter =
    trim _ :: titleCase _ :: super.setFilter
}
```

…a user entering "jaglan beta" as a planet name would see it stored in the database as "Jaglan Beta".

See Also

The best place to understand the filters is the trait `StringValidators` in the source for `BaseField`: *https://github.com/lift/framework/blob/master/core/util/src/main/scala/net/liftweb/util/BaseField.scala*.

If you really do need to apply title case to a value, the Apache Commons `WordUtils` class provides ready-made functions for this. See: *http://commons.apache.org/lang/*.

7.8. Testing with Specs2

Problem

You want to write Specs2 unit tests which access your database model with Squeryl and Record.

Solution

Use an in-memory database, and arrange for it to be set up before your test and destroyed after it.

There are three parts to this: including a database in your project and connecting to it in an in-memory mode; creating a re-usable trait to set up the database; and then using the trait in your test.

The H2 database has an in memory mode, meaning it won't save data to disk. It needs to be included in *build.sbt* as a dependency. Whilst you are editing *build.sbt* also disable SBT's parallel test execution to prevent database tests from influencing each other:

```
libraryDependencies += "com.h2database" % "h2" % "1.3.170"

parallelExecution in Test := false
```

Create a trait to initialise the database and create the schema:

```
package code.model

import java.sql.DriverManager

import org.squeryl.Session
import org.squeryl.adapters.H2Adapter

import net.liftweb.util.StringHelpers
import net.liftweb.common._
import net.liftweb.http.{S, Req, LiftSession }
import net.liftweb.squerylrecord.SquerylRecord
import net.liftweb.squerylrecord.RecordTypeMode._

import org.specs2.mutable.Around
import org.specs2.execute.Result

trait TestLiftSession {
  def session = new LiftSession("", StringHelpers.randomString(20), Empty)
  def inSession[T](a: => T): T = S.init(Req.nil, session) { a }
}

trait DBTestKit extends Loggable {

  Class.forName("org.h2.Driver")
```

```
    Logger.setup = Full(net.liftweb.util.LoggingAutoConfigurer())
    Logger.setup.foreach { _.apply() }

    def configureH2() = {
      SquerylRecord.initWithSquerylSession(
        Session.create(
              DriverManager.getConnection("jdbc:h2:mem:dbname;DB_CLOSE_DELAY=-1",
"sa", ""),
          new H2Adapter)
      )
    }

    def createDb() {
      inTransaction {
        try {
          MySchema.drop
          MySchema.create
        } catch {
          case e : Throwable =>
            logger.error("DB Schema error", e)
            throw e
        }
      }
    }

}

case class InMemoryDB() extends Around with DBTestKit with TestLiftSession {
  def around[T <% Result](testToRun: =>T) = {
    configureH2
    createDb
    inSession {
      inTransaction {
        testToRun
      }
    }
  }
}
```

In summary, this trail provides an `InMemoryDB` *context* for Specs2. This context ensures that the database is configured, the schema created and a transaction is supplied around your test.

Finally, mix the trait into your test and execute in the scope of the `InMemoryDB` context.

As an example, using the schema from Recipe 7.3 we can test that the planet Mars has two moons:

```
package code.model

import org.specs2.mutable._
import net.liftweb.squerylrecord.RecordTypeMode._
```

```
import MySchema._

class PlanetsSpec extends Specification with DBTestKit {

  sequential

  "Planets" >> {

    "know that Mars has two moons" >> InMemoryDB() {

      val mars = planets.insert(Planet.createRecord.name("Mars"))
      Satellite.createRecord.name("Phobos").planetId(mars.idField.is).save
      Satellite.createRecord.name("Deimos").planetId(mars.idField.is).save

      mars.satellites.size must_== 2
    }

  }

}
```

Running this with SBT's `test` command would show a success:

```
> test
[info] PlanetsSpec
[info]
[info] Planets
[info] + know that Mars has two moons
[info]
[info]
[info] Total for specification PlanetsSpec
[info] Finished in 1 second, 274 ms
[info] 1 example, 0 failure, 0 error
[info]
[info] Passed: : Total 1, Failed 0, Errors 0, Passed 1, Skipped 0
[success] Total time: 3 s, completed 03-Feb-2013 11:31:16
```

Discussion

The `DBTestKit` trait has to do quite a lot of work for us. At the lowest level, it loads the H2 driver and configures Squeryl with an in-memory connection. The `mem` part of the JDBC connection string (`jdbc:h2:mem:dbname;DB_CLOSE_DELAY=-1`) means that H2 won't try to persist the data to disk. The database just resides in memory, so there are no files in disk to maintain, and it runs quickly.

By default, when a connection is closed, the in-memory database is destroyed. In this recipe we've disabled that by adding the `DB_CLOSE_DELAY=-1`, which will allow us to write unit tests that span connections if we want to.

The next step up from connection management is the creation of the database schema in memory. We do this in `createDb` by throwing away the schema and any data when

we start a test, and create it afresh. If you have very common test datasets, this might be a good place to insert that data before your test runs.

These steps are brought together at the `InMemoryDB` class, which implements a Specs2 interface for code to run `Around` a test. We've also wrapped the test around a `TestLiftSession`. This provides an empty session, which is useful if you are accessing state-related code (such as the `S` object). It's not necessary for running tests against Record and Squeryl, but it has been included here because you may want to do that at some point.

In our specification itself, we mix in the `DBTestKit` and reference the `InMemoryDB` context on the tests that access the database. You'll note that we've used `>>` rather than Specs2's `should` and `in` that you may have seen elsewhere. This is to avoid name conflicts between Specs2 and Squeryl that you might come across.

As we disabled parallel execution with SBT, we also disable parallel execution in Specs2 with `sequential`. We are doing this to prevent a situation where one test might be expecting data that another test is modifying at the same time.

If all the tests in a specification are going to use the database, you can use the Specs2 `AroundContextExample[T]` to avoid having to mention `InMemoryDB` on every test. To do that, mix in `AroundContextExample[InMemoryDB]` and define `aroundContext`:

```
package code.model

import MySchema._

import org.specs2.mutable._
import org.specs2.specification.AroundContextExample
import net.liftweb.squerylrecord.RecordTypeMode._

class AlternativePlanetsSpec extends Specification with
  AroundContextExample[InMemoryDB] {

  sequential

  def aroundContext = new InMemoryDB()

  "Solar System" >> {

    "know that Mars has two moons" >> {

      val mars = planets.insert(Planet.createRecord.name("Mars"))
      Satellite.createRecord.name("Phobos").planetId(mars.idField.is).save
      Satellite.createRecord.name("Deimos").planetId(mars.idField.is).save

      mars.satellites.size must_== 2
    }
```

```
    }
  }
```

All the tests in `AlternativePlanetsSpec` will now be run with an `InMemoryDB` around them.

We've used a database with an in-memory mode for the advantages of speed and no files to clean up. However, you could use any regular database: you'd need to change the driver and connection string.

See Also

See *http://www.h2database.com/html/features.html#in_memory_databases* for more about H2's in-memory database settings.

Recipe 8.9 discusses unit testing with MongoDB, but the comments on SBT's other testing commands and testing in an IDE would apply to this recipe too.

7.9. Store a Random Value in a Column

Problem

You need a column to hold a random value.

Solution

Use `UniqueIdField`:

```
import net.liftweb.record.field.UniqueIdField
val randomId = new UniqueIdField(this, 32) {}
```

Note the {} in the example: this is required as `UniqueIdField` is an abstract class.

The size value, 32, indicates how many random characters to create.

Discussion

The `UniqueIdField` field is a kind of `StringField` and the default value for the field comes from `StringHelpers.randomString`. The value is randomly generated, but not guaranteed to be unique in the database.

The database column backing the `UniqueIdField` in this recipe will be a `varchar(32) not null` or similar. The value stored will look like:

```
GOJFGQRLS5GVYGPH3L3HRNXTATG3RM5M
```

As the value is made up of just letters and numbers, it makes it easy to use in URLs as there are no characters to escape. For example, it could be used in a link to allow a user

to validate their account when sent the link over email, which is one of the uses in `ProtoUser`.

If you need to change the value, the `reset` method on the field will generate a new random string for the field.

If you need an automatic value that is even more likely to be unique per-row, you can add a field that wraps a *universally unique identifier* (UUID):

```
import java.util.UUID

val uuid = new StringField(this, 36) {
  override def defaultValue = UUID.randomUUID().toString
}
```

This will automatically insert values of the form "6481a844-460a-a4e0-9191-c808e3051519" in records you create.

See Also

Java's UUID support is described at *http://docs.oracle.com/javase/7/docs/api/java/util/UUID.html* and includes a link to RFC 4122 which defines UUIDs.

7.10. Automatic Created and Updated Timestamps

Problem

You want created and updated timestamps on your records and would like them automatically updated when a row is added or updated.

Solution

Define the following traits:

```
package code.model

import java.util.Calendar

import net.liftweb.record.field.DateTimeField
import net.liftweb.record.Record

trait Created[T <: Created[T]] extends Record[T] {
  self: T =>
  val created: DateTimeField[T] = new DateTimeField(this) {
    override def defaultValue = Calendar.getInstance
  }
}

trait Updated[T <: Updated[T]] extends Record[T] {
```

```
    self: T =>

    val updated = new DateTimeField(this) {
      override def defaultValue = Calendar.getInstance
    }

    def onUpdate = this.updated(Calendar.getInstance)

  }

  trait CreatedUpdated[T <: Updated[T] with Created[T]] extends
    Updated[T] with Created[T] {
      self: T =>
  }
```

Add the trait to the model. For example, we can modify a `Planet` record to include the time the record was created and updated:

```
class Planet private () extends Record[Planet]
  with KeyedRecord[Long] with CreatedUpdated[Planet] {
    override def meta = Planet
    // field entries as normal...
}
```

Finally, arrange for the `updated` field to be updated:

```
class MySchema extends Schema {
  ...
  override def callbacks = Seq(
    beforeUpdate[Planet] call {_.onUpdate}
  )
  ...
```

Discussion

Although there is a built-in `net.liftweb.record.LifecycleCallbacks` trait which allows you trigger behaviour `onUpdate`, `afterDelete` and so on, it is only for use on individual fields, rather than records. As our goal is to update the `updated` field when any part of the Record changes, we can't use the `LiftcycleCallbacks` here.

Instead, the `CreatedUpdated` trait simplifies adding an `updated` and `created` fields to a Record, but we do need to remember to add a hook into the schema to ensure the `updated` value is changed when a record is modified. This is why we set the `callbacks` on the Schema.

The schema for records with `CreatedUpdated` mixed in will include two additional columns:

```
updated timestamp not null,
created timestamp not null
```

The `timestamp` is used for the H2 database. For other databases the type may be different.

The values can be accessed like any other record field. Using the example data from Recipe 7.3 we could run the following:

```
val updated : Calendar = mars.updated.id
val created : Calendar = mars.created.is
```

If you only need created time, or updated time, just mix in the `Created[T]` or `Updated[T]` trait instead of `CreatedUpdated[T]`.

It should be noted that `onUpdate` is only called on full updates and not on partial updates with Squeryl. A full update is when the object is altered and then saved; a partial update is where you attempt to alter objects via a query.

If you're interested in other automations for Record, the Squeryl schema callbacks support these triggered behaviours:

- `beforeInsert` and `afterInsert`
- `afterSelect`
- `beforeUpdate` and `afterUpdate`
- `beforeDelete` and `afterDelete`

See Also

For a discussion of the difference between partial and full updates in Squeryl, see : *http://squeryl.org/inserts-updates-delete.html*.

7.11. Logging SQL

Problem

You want to see the SQL being executed by Squeryl.

Solution

Add the following anytime you have a Squeryl season, such as just before your query:

```
org.squeryl.Session.currentSession.setLogger( s => println(s) )
```

By providing a `String => Unit` function to `setLogger`, Squeryl will execute that function with the SQL it runs. In this example, we are simply printing the SQL to the console.

Discussion

You'll probably want to use the logging facilities in Lift to capture SQL. For example:

```
package code.snippet

import net.liftweb.common.Loggable
import org.squeryl.Session

class MySnippet extends Loggable {

  def render = {
    Session.currentSession.setLogger( s => logger.info(s) )
    // ...your snippet code here...
  }
}
```

This will log queries according to the settings for the logging system, typically the Logback project configured in *src/resources/props/default.logback.xml*.

It can be inconvenient to have to enable logging in each snippet during development. To trigger logging for all snippets, you can modify the `addAround` call in *Boot.scala* (Recipe 7.1) to include a `setLogger` call while `inTransaction`:

```
S.addAround(new LoanWrapper {
  override def apply[T](f: => T): T = {
    val result = inTransaction {
    Session.currentSession.setLogger( s => logger.info(s) )
    // ... rest of addAround as normal
```

See Also

Squeryl `setLogger` is documented at: *http://squeryl.org/miscellaneous.html*.

You can learn about logging in Lift from the Logging wiki page: *https://www.assembla.com/spaces/liftweb/wiki/Logging*.

7.12. Model a Column with MySQL MEDIUMTEXT

Problem

You want to use MySQL's `MEDIUMTEXT` for a column, but `StringField` doesn't have this option.

Solution

Use Squeryl's `dbType` in your schema:

```
object MySchema extends Schema {
  on(mytable)(t => declare(
```

```
    t.mycolumn defineAs dbType("MEDIUMTEXT")
  ))
}
```

This schema setting will give you the correct column type in MySQL:

```
create table mytable (
    mycolumn MEDIUMTEXT not null
);
```

On the record you can use `StringField` as usual.

Discussion

This recipe points towards the flexibility available with Squeryl's schema definition DSL. The column attribute in this example is just one of a variety of adjustments you can make to the default choices that Squeryl uses.

For example, you can use the syntax to chain column attributes for a single column, and also defined multiple columns at the same time:

```
object MySchema extends Schema {
  on(mytable)(t => declare(
    t.mycolumn defineAs(dbType("MEDIUMTEXT"),indexed),
    t.id definedAs(unique, named("MY_ID"))
  ))
}
```

See Also

The schema definition page for Squeryl gives examples of attributes you can apply to tables and columns: *http://squeryl.org/schema-definition.html*.

7.13. MySQL Character Set Encoding

Problem

Some characters stored in your MySQL database are appearing as `???`.

Solution

Ensure that:

- `LiftRules.early.append(_.setCharacterEncoding("UTF-8"))` is included in *Boot.scala*.
- `?useUnicode=true&characterEncoding=UTF-8` is included in your JDBC connections URL.

- Your MySQL database has been created using a UTF-8 character set.

Discussion

There are a number of interactions here which can impact characters going into, and coming out of, a MySQL database. The basic problem is that bytes transferred across networks have no meaning unless you know the encoding.

The `setCharacterEncoding("UTF-8")` call in *Boot.scala* is being applied to every `HTTPRequest` which ultimately, in a servlet container, is applied to a `ServletRequest`. This is how parameters in a request are going to be interpreted by the servlet container when received.

The flip side of this is that responses from Lift are encoded as UTF-8. You'll see this in a number of places. For example, `templates-hidden/default` includes:

```
<meta http-equiv="content-type" content="text/html; charset=UTF-8" />
```

Also, the `LiftResponse` classes set the encoding as UTF-8.

Another aspect is how character data from Lift is sent to the database over the network. This is controlled by the parameters to the JDBC driver. The default for MySQL is to detect the encoding, but it seems from experience that this is not a great option, so we force the UTF-8 encoding.

Finally, the MySQL database itself needs to store the data as UTF-8. The default character encoding is not UTF-8, so you'll need to specify the encoding when you create the database:

```
CREATE DATABASE myDb CHARACTER SET utf8
```

See Also

The MySQL JDBC configuration guide is at: *http://dev.mysql.com/doc/refman/5.6/en/connector-j-reference-configuration-properties.html*.

CHAPTER 8

MongoDB Persistence with Record

This section gives recipes for making use of MongoDB in your Lift application. Many of the code examples in this chapter can be found in: *https://github.com/LiftCookbook/cookbook_mongo*.

8.1. Connecting to a MongoDB Database

Problem

You want to connect to a MongoDB.

Solution

Add the Lift MongoDB dependencies to your build and configure a connection using `net.liftweb.mongodb` and `com.mongodb`.

In *build.sbt* add the following to `libraryDependencies`:

```
"net.liftweb" %% "lift-mongodb-record" % liftVersion
```

In *Boot.scala* add:

```
import com.mongodb.{ServerAddress, Mongo}
import net.liftweb.mongodb.{MongoDB,DefaultMongoIdentifier}

val server = new ServerAddress("127.0.0.1", 27017)
MongoDB.defineDb(DefaultMongoIdentifier, new Mongo(server), "mydb")
```

This will give you a connection to a local MongoDB database called "mydb".

Discussion

If your database needs authentication, use `MongoDB.defineDbAuth`:

```
MongoDB.defineDbAuth(DefaultMongoIdentifier, new Mongo(server),
  "mydb", "username", "password")
```

Some cloud services will give you a URL to connect to, such as *mongodb://alex.mongohq.com:10050/fglvBskrsdsdsDaGNs1*. In this case, the host and the port make up the first part, and the database name is the part after the /.

If you need to turn a URL like this into a connection, you can do so by using `java.net.URI` to parse the URL and make a connection:

```
object MongoUrl {

  def defineDb(id: MongoIdentifier, url: String) {

    val uri = new URI(url)

    val db = uri.getPath drop 1
    val server = new Mongo(new ServerAddress(uri.getHost, uri.getPort))

    Option(uri.getUserInfo).map(_.split(":")) match {
      case Some(Array(user,pass)) =>
        MongoDB.defineDbAuth(id, server, db, user, pass)
      case _ =>
        MongoDB.defineDb(id, server, db)
    }
  }

}

MongoUrl.defineDb(DefaultMongoIdentifier,
  "mongodb://user:pass@127.0.0.1:27017/myDb")
```

The full URL scheme for MongoDB is more complicated, allowing for multiple hosts and connection parameters, but the above code handles optional user name and password fields and may be enough to get you up and running with your MongoDB configuration.

The `DefaultMongoIdentifier` is a value used to identify a particular connection. Lift keeps a map of identifiers to connections, meaning you can connect to more than one database. The common case is a single database, and that is usually assigned to `DefaultMongoIdentifier`.

However, if you do need to access two MongoDB databases, you can create a new identifier and assign it as part of your record. For example:

```
object OtherMongoIdentifier extends MongoIdentifier {
  def jndiName: String = "other"
}

MongoUrl.defineDb(OtherMongoIdentifier, "mongodb://127.0.0.1:27017/other")
```

```
object Country extends Country with MongoMetaRecord[Country] {
  override def collectionName = "example.earth"
  override def mongoIdentifier = OtherMongoIdentifier
}
```

The `lift-mongodb-record` dependency itself depends on another Lift module, `lift-mongodb`, which provides connectivity and other other lower-level access to MongoDB. Both bottom out with the MongoDB Java driver.

See Also

Connection configuration that includes replica sets and MongoDB options, such as timeout settings, are described on the Lift wiki at: *https://www.assembla.com/wiki/show/liftweb/Mongo_Configuration*.

The full specification for MongoDB connection URLs is: *http://docs.mongodb.org/manual/reference/connection-string/*.

8.2. Storing a HashMap in a MongoDB Record

Problem

You want to store a hash map in MongoDB.

Solution

Create a MongoDB Record which contains a `MongoMapField`:

```
import net.liftweb.mongodb.record._
import net.liftweb.mongodb.record.field._

class Country private () extends MongoRecord[Country] with StringPk[Country] {
  override def meta = Country
  object population extends MongoMapField[Country,Int](this)
}

object Country extends Country with MongoMetaRecord[Country] {
  override def collectionName = "example.earth"
}
```

In this example we are creating a record for information about a country, and the `population` is a map from a `String` key to an `Integer` value.

We can use it in a snippet like this:

```
class Places {

  val uk = Country.find("uk") openOr {
    val info = Map(
```

```
        "Brighton" -> 134293,
        "Birmingham" -> 970892,
        "Liverpool" -> 469017)

      Country.createRecord.id("uk").population(info).save
    }

    def facts = "#facts" #> (
      for { (name,pop) <- uk.population.is } yield
        ".name *" #> name & ".pop *" #> pop
    )
  }
```

When this snippet is called, it looks up a record by `_id` of "uk" or creates it using some canned information. The template to go with the snippet could include:

```
<div data-lift="Places.facts">
 <table>
  <thead>
   <tr><th>City</th><th>Population</th></tr>
  </thead>
  <tbody>
   <tr id="facts">
    <td class="name">Name here</td><td class="pop">Population</td>
   </tr>
  </tbody>
 </table>
</div>
```

In MongoDB the resulting data structure would be:

```
$ mongo cookbook
MongoDB shell version: 2.0.6
connecting to: cookbook
> show collections
example.earth
system.indexes
> db.example.earth.find().pretty()
{
  "_id" : "uk",
  "population" : {
    "Brighton" : 134293,
    "Birmingham" : 970892,
    "Liverpool" : 469017
  }
}
```

Discussion

If you do not set a value for the map, the default will be an empty map, represented in MongoDB as:

```
({ "_id" : "uk", "population" : { } })
```

An alternative is to mark the field as optional:

```
object population extends MongoMapField[Country,Int](this) {
  override def optional_? = true
}
```

If you now write the document without a `population` set, the field will be omitted in MongoDB:

```
> db.example.earth.find();
{ "_id" : "uk" }
```

To append data to the map from your snippet, you can modify the record to supply a new `Map`:

```
uk.population(uk.population.is + ("Westminster"->81766)).update
```

Note that we are using `update` here, rather than `save`. The `save` method is pretty smart and will either insert a new document into a MongoDB collection or *replace* an existing document based on the `_id`. Update is different: it detects just the changed fields of the document and updates them. It will send this command to MongoDB for the document:

```
{ "$set" : { "population" : { "Brighton" : 134293 , "Liverpool" : 469017 ,
  "Birmingham" : 970892 , "Westminster" : 81766} }
```

You'll probably want to use `update` over `save` for changes to existing records.

To access an individual element of the map, you can use `get` (or `value`):

```
uk.population.get("San Francisco")
// will throw java.util.NoSuchElementException
```

…or you can access via the standard Scala map interface:

```
val sf : Option[Int] = uk.population.is.get("San Francisco")
```

What a `MongoMapField` Can Contain

You should be aware that `MongoMapField` supports only primitive types.

The mapped field used in this recipe is typed `String => Int`, but of course MongoDB will let you mix types such as putting a `String` or a `Boolean` as a population value. If you do modify the MongoDB record in the database outside of Lift and mix types, you'll get a `java.lang.ClassCastException` at runtime.

See Also

A discussion on the mailing list regarding the limited type support in `MongoMapField` and a possible way around it by overriding `asDBObject` can be found via: *http://bit.ly/lift-mongomap*.

8.3. Storing an Enumeration in MongoDB

Problem

You want to store an enumeration in a MongoDB document.

Solution

Use `EnumNameField` to store the string value of the enumeration. Here's an example using days of the week:

```
object DayOfWeek extends Enumeration {
  type DayOfWeek = Value
  val Mon, Tue, Wed, Thu, Fri, Sat, Sun = Value
}
```

We can use this to model someone's birth day-of-week:

```
package code.model

import net.liftweb.mongodb.record._
import net.liftweb.mongodb.record.field._
import net.liftweb.record.field.EnumNameField

class Birthday private () extends MongoRecord[Birthday] with StringPk[Birthday]
{
  override def meta = Birthday
  object dow extends EnumNameField(this, DayOfWeek)
}

object Birthday extends Birthday with MongoMetaRecord[Birthday]
```

When creating records `dow` field will expect a `DayOfWeek` value:

```
import DayOfWeek._

Birthday.createRecord.id("Albert Einstein").dow(Fri).save
Birthday.createRecord.id("Richard Feynman").dow(Sat).save
Birthday.createRecord.id("Isaac Newton").dow(Sun).save
```

Discussion

Take a look at what's stored in MongoDB:

```
> db.birthdays.find()
{ "_id" : "Albert Einstein", "dow" : "Fri" }
{ "_id" : "Richard Feynman", "dow" : "Sat" }
{ "_id" : "Isaac Newton", "dow" : "Sun" }
```

The `dow` value is the `toString` of the enumeration, not the `id` value:

```
Fri.toString // java.lang.String = Fri
Fri.id //  Int = 4
```

If you wanted to store the ID, use `EnumField` instead.

Be aware that other tools, notably Rogue, expect the string value, not the integer ID, of an enumeration, so you may prefer to use `EnumNameField` for that reason.

See Also

Recipe 8.6 introduces *Rogue*.

8.4. Embedding a Document Inside a MongoDB Record

Problem

You have a MongoDB record, and you want to embed another set of values inside it as a single entity.

Solution

Use `BsonRecord` to define the document to embed, and embed it using `BsonRecordField`. Here's an example of storing information about an image within a record:

```
import net.liftweb.record.field.{IntField,StringField}

class Image private () extends BsonRecord[Image] {
  def meta = Image
  object url extends StringField(this, 1024)
  object width extends IntField(this)
  object height extends IntField(this)
}

object Image extends Image with BsonMetaRecord[Image]
```

We can reference instances of the `Image` class via `BsonRecordField`:

```
class Country private () extends MongoRecord[Country] with StringPk[Country] {
  override def meta = Country
  object flag extends BsonRecordField(this, Image)
}

object Country extends Country with MongoMetaRecord[Country] {
  override def collectionName = "example.earth"
}
```

To associate a value:

```
val unionJack =
  Image.createRecord.url("http://bit.ly/unionflag200").width(200).height(100)
```

```
Country.createRecord.id("uk").flag(unionJack).save(true)
```

In MongoDB, the resulting data structure would be:

```
> db.example.earth.findOne()
{
  "_id" : "uk",
  "flag" : {
    "url" : "http://bit.ly/unionflag200",
    "width" : 200,
    "height" : 100
  }
}
```

Discussion

If you don't set a value on the embedded document, the default will be saved as:

```
"flag" : { "width" : 0, "height" : 0, "url" : "" }
```

You can prevent this by making the image optional:

```
object image extends BsonRecordField(this, Image) {
  override def optional_? = true
}
```

With `optional_?` set in this way the image part of the MongoDB document won't be saved if the value is not set. Within Scala you will then want to access the value with `valueBox` call:

```
val img : Box[Image] = uk.flag.valueBox
```

In fact, regardless of the setting of `optional_?` you can access the value using `valueBox`.

An alternative to optional values is to always provide a default value for the embedded document:

```
object image extends BsonRecordField(this, Image) {
 override def defaultValue =
  Image.createRecord.url("http://bit.ly/unionflag200").width(200).height(100)
}
```

See Also

The Lift Wiki describes BsonRecord in more detail at: *https://www.assembla.com/spaces/liftweb/wiki/Mongo_Record_Embedded_Objects*.

8.5. Linking Between MongoDB Records

Problem

You have a MongoDB record and want to include a link to another record.

Solution

Create a reference using a `MongoRefField` such as `ObjectIdRefField` or `StringRefField`, and dereference the record using the `obj` call.

As an example we can create records representing countries, where a country references the planet where you can find it:

```
class Planet private() extends MongoRecord[Planet] with StringPk[Planet] {
  override def meta = Planet
  object review extends StringField(this,1024)
}

object Planet extends Planet with MongoMetaRecord[Planet] {
  override def collectionName = "example.planet"
}

class Country private () extends MongoRecord[Country] with StringPk[Country] {
  override def meta = Country
  object planet extends StringRefField(this, Planet, 128)
}

object Country extends Country with MongoMetaRecord[Country] {
  override def collectionName = "example.country"
}
```

To make this example easier to follow, our model mixes in `StringPk[Planet]` to use strings as the primary key on our documents, rather than the more usual MongoDB object IDs. Consequently, the link is established with a `StringRefField`.

In a snippet we can make us of the `planet` reference by resolving it with `.obj`:

```
class HelloWorld {

  val uk = Country.find("uk") openOr {
    val earth = Planet.createRecord.id("earth").review("Harmless").save
    Country.createRecord.id("uk").planet(earth.id.is).save
  }

  def facts =
    ".country *" #> uk.id &
    ".planet" #> uk.planet.obj.map { p =>
      ".name *" #> p.id &
      ".review *" #> p.review }
  }
```

For the value `uk` we lookup an existing record, or create one if none is found. We create `earth` as a separate MongoDB record, and then reference it in the `planet` field with the ID of the planet.

Retrieving the reference is via the `obj` method, which returns a `Box[Planet]` in this example.

Discussion

Referenced records are fetched from MongoDB when you call the `obj` method on a `MongoRefField`. You can see this by turning on logging in the MongoDB driver. Do this by adding the following to the start of your *Boot.scala*:

```
System.setProperty("DEBUG.MONGO", "true")
System.setProperty("DB.TRACE", "true")
```

Having done this, the first time you run the snippet above your console will include:

```
INFO: find: cookbook.example.country { "_id" : "uk"}
INFO: update: cookbook.example.planet { "_id" : "earth"} { "_id" : "earth" ,
    "review" : "Harmless"}
INFO: update: cookbook.example.country { "_id" : "uk"} { "_id" : "uk" ,
    "planet" : "earth"}
INFO: find: cookbook.example.planet { "_id" : "earth"}
```

What you're seeing here is the initial look up for "uk", followed by the creation of the "earth" record and an update which is saving the "uk" record. Finally, there is a lookup of "earth" when `uk.obj` is called in the `facts` method.

The `obj` call will cache the `planet` reference. That means you could say…

```
".country *" #> uk.id &
".planet *" #> uk.planet.obj.map(_.id) &
".review *" #> uk.planet.obj.map(_.review)
```

…and you'd still only see one query for the "earth" record despite calling `obj` multiple times. The flip side of that is if the "earth" record was updated elsewhere in MongoDB after you called `obj` you would not see the change from a call to `uk.obj` unless you reloaded the `uk` record first.

Querying by Reference

Searching for records by a reference is straight-forward:

```
val earth : Planet = ...
val onEarth : List[Country]= Country.findAll(Country.planet.name, earth.id.is)
```

Or in this case, because we have `String` references, we could just say:

```
val onEarth : List[Country]= Country.findAll(Country.planet.name, "earth")
```

Updating and Deleting

Updating a reference is as you'd expect:

```
uk.planet.obj.foreach(_.review("Mostly harmless.").update)
```

This would result in the changed field being set:

```
INFO: update: cookbook.example.planet { "_id" : "earth"} { "$set" : {
   "review" : "Mostly harmless."}}
```

A `uk.planet.obj` call will now return a planet with the new review.

Or you could replace the reference with another:

```
uk.planet( Planet.createRecord.id("mars").save.id.is ).save
```

Again, note that the reference is via the ID of the record (`id.is`), not the record itself.

To remove the reference:

```
uk.planet(Empty).save
```

This removes the link, but the MongoDB record pointed to by the link will remain in the database. If you remove the object being referenced, a later call to `obj` will return an `Empty` box.

Types of Link

The example uses a `StringRefField` as the MongoDB records themselves use `String` as the `_id`. Other reference types are:

- `ObjectIdRefField` — possibly the most frequently used kind of reference, when you want to reference via the usual default `ObjectId` in MongoDB.
- `UUIDRefField` — for records with an ID based on `java.util.UUID`.
- `StringRefField` — as used in this example, where you control the ID as a `String`.
- `IntRefField` and `LongRefField` — for when you're using a numeric value as an ID.

See Also

10Gen Inc's *Data Modeling Decisions* describes embedding of documents compared to referencing objects. You'll find the article at: *http://docs.mongodb.org/manual/core/data-modeling/*.

8.6. Using Rogue

Problem

You want to use Foursquare's type-safe domain specific language (DSL), Rogue, for querying and updating MongoDB records.

Solution

You need to include the Rogue dependency in your build and import Rogue into your code.

For the first step, edit *build.sbt* and add:

```
"com.foursquare" %% "rogue" % "1.1.8" intransitive()
```

In your code `import com.foursquare.rogue._` and then start using Rogue. For example, using the Scala console (see Recipe 8.8):

```
scala> import com.foursquare.rogue.Rogue._
import com.foursquare.rogue.Rogue._

scala> import code.model._
import code.model._

scala> Country.where(_.id eqs "uk").fetch
res1: List[code.model.Country] = List(class code.model.Country={_id=uk,
  population=Map(Brighton->134293, Liverpool->469017, Birmingham->970892)})

scala> Country.where(_.id eqs "uk").count
res2: Long = 1

scala> Country.where(_.id eqs "uk").
  modify(_.population at "Brighton" inc 1).updateOne()
```

Discussion

Rogue is able to use information in your Lift Record to offer an elegant way to query and update records. It's type safe meaning, for example, if you try to use an `Int` where a `String` is expected in a query, MongoDB would allow that and fail to find results at runtime, but Rogue enables Scala to reject the query at compile time:

```
scala> Country.where(_.id eqs 7).fetch
<console>:20: error: type mismatch;
 found   : Int(7)
 required: String
              Country.where(_.id eqs 7).fetch
```

The DSL constructs a query which we then `fetch` to send the query to MongoDB. That last method, `fetch`, is just one of the ways to run the query. Others include:

- `count` — queries MongoDB for the size of the result set.
- `countDistinct` — the number of distinct values in the results.
- `exists` — true if there's any record that matches the query.
- `get` — returns an `Option[T]` from the query.
- `fetch(limit: Int)` — like `fetch` but returns at most `limit` results.
- `updateOne`, `updateMulti`, `upsertOne` and `upsertMulti` — modify a single document, or all documents, that match the query.
- `findAndDeleteOne` and `bulkDelete_!!` — to delete records.

The query language itself is expressive, and the best place to explore the variety of queries is in the `QueryTest` specification in the source for Rogue. You'll find a link to this in the README of the project on Github.

Rogue is working towards a v2 release which introduces a number of new concepts. If you want to give it a try, take a look at the instructions and comments on the Rogue mailing list via: *http://bit.ly/rogue2-announce*.

See Also

For geospacial queries, see Recipe 8.7.

The README page for Rogue is a great starting point, and includes a link to `QueryTest` giving plenty of example queries to crib from: *https://github.com/foursquare/rogue*.

The motivation for Rogue is described in a Foursquare engineering blog post: *http://engineering.foursquare.com/2011/01/21/rogue-a-type-safe-scala-dsl-for-querying-mongodb/*.

8.7. Storing Geospatial Values

Problem

You want to store latitude and longitude information in MongoDB.

Solution

Use Rogue's `LatLong` class to embed location information in your model. For example, we can store the location of a city like this:

```
import com.foursquare.rogue.Rogue._
import com.foursquare.rogue.LatLong

class City private () extends MongoRecord[City] with ObjectIdPk[City] {
  override def meta = City
  object name extends StringField(this, 60)
  object loc extends MongoCaseClassField[City, LatLong](this)
}

object City extends City with MongoMetaRecord[City] {
  import net.liftweb.mongodb.BsonDSL._
  ensureIndex(loc.name -> "2d", unique=true)
  override def collectionName = "example.city"
}
```

We can store values like this:

```
val place = LatLong(50.819059, -0.136642)
val city = City.createRecord.name("Brighton, UK").loc(pos).save(true)
```

This will produce data in MongoDB that looks like this:

```
{
  "_id" : ObjectId("50f2f9d43004ad90bbc06b83"),
  "name" : "Brighton, UK",
  "loc" : {
    "lat" : 50.819059,
    "long" : -0.136642
  }
}
```

Discussion

MongoDB supports *geospatial indexes*, and we're making use of this by doing two things. First, we are storing the location information in one of MongoDB's permitted formats. The format is an embedding document containing the coordinates. We could also have use a array of two values to represent the point.

Second, we're creating a index of type "2d", which allows us to use MongoDB's geospatial functions such as `$near` and `$within`. The `unique=true` in the `ensureIndex` highlights that you can control whether locations needs to be unique (`true`, no duplications) or not (`false`).

With regard to the unique index, you'll note that we're calling `save(true)` on the `City` in this example, rather than the plain `save` in most other recipes. We could use `save` here, and it would work fine, but difference is that `save(true)` raises the *write concern* level from "normal" to "safe".

With the normal write concern, the call to `save` would return as soon as the request has gone down the wire to the MongoDB server. This gives a certain degree of reliability in

that `save` would fail if the network had gone away. However, there's no indication that the server has processed the request. For example, if we tried to insert a city at the exact same location as one that was already in the database, the index uniqueness rule would be violated and the record would not be saved. With just `save` (or `save(false)`) our Lift application would not receive this error, and the call would fail silently. Raising the concern to "safe" causes `save(true)` to wait for an acknowledgment from the MongoDB server, which means the application will receive exceptions for some kinds of errors.

As an example, if we tried to insert a duplicate city, our call to `save(true)` would result in:

```
com.mongodb.MongoException$DuplicateKey: E11000 duplicate key
  error index: cookbook.example.city.$loc_2d
```

There are other levels of write concern, available via another variant of `save` which takes a `WriteConcern` as an argument.

If you ever need to drop an index, the MongoDB command is:

```
db.example.city.dropIndex( "loc_2d" )
```

Querying

The reason this recipe uses Rogue's `LatLong` class is to enable us to query using the Rogue DSL. Suppose we've inserted other cities into our collection:

```
> db.example.city.find({}, {_id:0} )
{"name": "London, UK", "loc": {"lat": 51.5, "long": -0.166667} }
{"name": "Brighton, UK", "loc": {"lat": 50.819059, "long": -0.136642} }
{"name": "Paris, France", "loc": {"lat": 48.866667, "long": 2.333333} }
{"name": "Berlin, Germany", "loc": {"lat": 52.533333, "long": 13.416667} }
{"name": "Sydney, Australia", "loc": {"lat": -33.867387, "long": 151.207629} }
{"name": "New York, USA", "loc": {"lat": 40.714623, "long": -74.006605} }
```

We can now find those cities within a 500km of London:

```
import com.foursquare.rogue.{LatLong, Degrees}

val centre = LatLong(51.5, -0.166667)
val radius = Degrees( (500 / 6378.137).toDegrees )
val nearby = City.where( _.loc near (centre.lat, centre.long, radius) ).fetch()
```

This would query MongoDB with this clause…

```
{ "loc" : { "$near" : [ 51.5 , -0.166667 , 4.491576420597608]}}
```

…which will identify London, Brighton and Paris as near to London.

The form of the query is a centre point and a spherical radius. Records falling inside that radius match the query and are returned closest first. We calculate the radius in radians: 500km divided by the radius of the Earth, approximately 6378km, gives us an angle in radians. We convert this to `Degrees` as required by Rogue.

See Also

The MongoDB manual discusses geospatial index at: *http://docs.mongodb.org/manual/core/geospatial-indexes/*.

You can learn more about write concerns at *http://docs.mongodb.org/manual/core/write-operations/*, and the various values to pass to `save` are described in the Java MongoDB driver: *http://api.mongodb.org/java/current/*.

8.8. Running Queries from the Scala Console

Problem

You want to try out a few queries interactively from the Scala console.

Solution

Start the console from your project, call `boot()`, and then interact with your model.

For example, using the MongoDB records developed as part of Recipe 8.1, we can perform a basic query:

```
$ sbt
...
> console
[info] Compiling 1 Scala source to /cookbook_mongo/target/scala-2.9.1/classes...
[info] Starting scala interpreter...
[info]
Welcome to Scala version 2.9.1.final ...
Type in expressions to have them evaluated.
Type :help for more information.

scala> import bootstrap.liftweb._
import bootstrap.liftweb._

scala> new Boot().boot

scala> import code.model._
import code.model._

scala> Country.findAll
res2: List[code.model.Country] = List(class code.model.Country={_id=uk,
  population=Map(Brighton -> 134293, Liverpool -> 469017,
  Birmingham -> 970892)})

scala> :q
```

Discussion

Running everything in `Boot` may be a little heavy handed, especially if you starting up various services and background tasks. All we need to do is define a database connection. For example, using the example code presented in Recipe 8.1, we could initialise a connection with:

```
scala> import bootstrap.liftweb._
import bootstrap.liftweb._

scala> import net.liftweb.mongodb._
import net.liftweb.mongodb._

scala> MongoUrl.defineDb(DefaultMongoIdentifier,
  "mongodb://127.0.0.1:27017/cookbook")

scala> Country.findAll
res2: List[code.model.Country] = List(class code.model.Country={_id=uk,
  population=Map(Brighton -> 134293, Liverpool -> 469017,
    Birmingham -> 970892)})
```

See Also

Recipe 8.1 for connecting to MongoDB and Recipe 8.6 for querying with Rogue.

8.9. Unit Testing Record with MongoDB

Problem

You want to write unit tests to run against your Lift Record code with MongoDB.

Solution

Using the Specs2 testing framework, surround your specification with a *context* which creates and connects to a database for each test and destroys it after the test runs.

First, create a Scala trait to set up and destroy a connection to MongoDB. We'll be mixing this trait into our specifications:

```
import net.liftweb.http.{Req, S, LiftSession}
import net.liftweb.util.StringHelpers
import net.liftweb.common.Empty
import net.liftweb.mongodb._
import com.mongodb.ServerAddress
import com.mongodb.Mongo
import org.specs2.mutable.Around
import org.specs2.execute.Result

trait MongoTestKit {
```

```
  val server = new Mongo(new ServerAddress("127.0.0.1", 27017))

  def dbName = "test_"+this.getClass.getName
    .replace(".", "_")
    .toLowerCase

  def initDb() : Unit = MongoDB.defineDb(DefaultMongoIdentifier, server, dbName)

  def destroyDb() : Unit = {
    MongoDB.use(DefaultMongoIdentifier) { d => d.dropDatabase() }
    MongoDB.close
  }

  trait TestLiftSession {
    def session = new LiftSession("", StringHelpers.randomString(20), Empty)
    def inSession[T](a: => T): T = S.init(Req.nil, session) { a }
  }

  object MongoContext extends Around with TestLiftSession {
    def around[T <% Result](testToRun: =>T) = {
      initDb()
      try {
        inSession {
          testToRun
        }
      } finally {
        destroyDb()
      }
    }
  }

}
```

This trait provides the plumbing for connection to a MongoDB server running locally, and creates a database based on the name of the class it is mixed into. The important part is the `MongoContext` which ensures that `around` your specification the database is initialized, and that after your specification is run, it is cleaned up.

To use this in a specification, mix in the trait and then add the context:

```
import org.specs2.mutable._

class MySpec extends Specification with MongoTestKit {

  sequential

  "My Record" should {

    "be able to create records" in MongoContext {
      val r = MyRecord.createRecord
      // ...your useful test here...
```

```
        r.valueBox.isDefined must beTrue
      }

    }
  }
```

You can now run the test in SBT by typing `test`:

```
> test
[info] Compiling 1 Scala source to target/scala-2.9.1/test-classes...
[info] My Record should
[info] + be able to create records
[info]
[info]
[info] Total for specification MySpec
[info] Finished in 1 second, 199 ms
[info] 1 example, 0 failure, 0 error
[info]
[info] Passed: : Total 1, Failed 0, Errors 0, Passed 0, Skipped 0
[success] Total time: 1 s, completed 03-Jan-2013 22:47:54
```

Discussion

Lift normally provides all the scaffolding you need to connect and run against MongoDB. Without a running Lift application, we need to ensure MongoDB is configured when our tests run outside of Lift, and that's what the `MongoTestKit` trait is providing for us.

The one unusual part of the test set up is including a `TestLiftSession`. This provides an empty session around your test, which is useful if you are accessing or testing state-related code (e.g., access to `S`). It's not strictly necessary for running tests against Record, but it has been included here because you may want to do that at some point, for example if you are testing user login via MongoDB records.

There are a few nice tricks in SBT to help you run tests. Running `test` will run all the tests in your project. If you want to focus on just one test, you can:

```
> test-only org.example.code.MySpec
```

This command also supports wildcards, so if we only wanted to run tests that start with the word "Mongo" we could:

```
> test-only org.example.code.Mongo*
```

There's also `test-quick` (in SBT 0.12) which will only run tests that have not been run, have changed, or failed last time and `~test` to watch for changes in tests and run them.

`test-only` together with modifications to `around` in `MongoTestKit` can be a good way to track down any issues you have with a test. By disabling the call to `destroyDb()` you

can jump into the MongoDB shell and examine the state of the database after a test has run.

Database Cleanup

Around each test we've simply deleted the database so the next time we try to use it, it'll be empty. In some situations you may not be able to do this. For example, if you're running tests against a database hosted with companies such as MongoLabs or MongoHQ, then deleting the database will mean you won't be able to connect to it next time you run.

One way to resolve that is to clean up each individual collection, by defining the collections you need to clean up, and replacing `destroyDb` with a method that will remove all entries in those collections:

```
lazy val collections : List[MongoMetaRecord[_]] = List(MyRecord)

def destroyDb() : Unit = {
  collections.foreach(_ bulkDelete_!! new BasicDBObject)
  MongoDB.close
}
```

Note that the collection list is `lazy` to avoid start up of the Record system before we've initialized our database connections.

Parallel Tests

If your tests are modifying data and have the potential to interact, you'll want to stop SBT from running your tests in parallel. A symptom of this would be tests that fail apparently randomly, or working tests that stop working when you add a new test, or tests that seem to lock up. Disable by adding the following to *build.sbt*:

```
parallelExecution in Test := false
```

You'll notice that the example specification includes the line: `sequential`. This disables the default behaviour in Specs2 of running all tests concurrently.

Running tests in IDEs

IntelliJ IDEA detects and allows you to runs Specs2 tests automatically. With Eclipse, you'll need to include the JUnit runner annotation at the start of your specification:

```
import org.junit.runner.RunWith
import org.specs2.runner.JUnitRunner

@RunWith(classOf[JUnitRunner])
class MySpec extends Specification with MongoTestKit  {
...
```

You can then "Run As…" the class in Eclipse.

See Also

Specs2 is documented at: *http://specs2.org/*.

If you prefer to use the Scala Test framework (*http://www.scalatest.org*), take a look at Tim Nelson's *Mongo Auth* Lift module at *https://github.com/eltimn/lift-mongoauth*. It includes tests using that framework that run against MongoDB. Much of what Tim has written there has been adapted to produce this recipe for Specs2.

The Lift MongoDB Record library includes a variation on testing with Specs2, using just `Before` and `After` rather than the `around` example used in this recipe. If you prefer that approach, you'll find the code in: *https://github.com/lift/framework/tree/master/persistence/mongodb-record/src/test/scala/net/liftweb/mongodb/record*.

Flapdoodle (*https://github.com/flapdoodle-oss/embedmongo.flapdoodle.de*) provides a way to automate the download, install, set up and clean up of a MongoDB database. This automation is something you can wrap around your unit tests, and a Specs2 integration is included using the same `Before` and `After` approach to testing used by Lift MongoDB Record: *https://github.com/athieriot/specs2-embedmongo*.

The test interface provided by SBT, such as the `test` command, also supports the ability to fork tests, set specific configurations for test cases, and ways to select which tests are run. You'll find it at: *http://www.scala-sbt.org/release/docs/Detailed-Topics/Testing*.

The Lift Wiki describes more about unit testing and Lift sessions: *https://www.assembla.com/wiki/show/liftweb/Unit_Testing_Snippets_With_A_Logged_In_User*.

CHAPTER 9

Around Lift

This chapter looks at interacting with other systems from within Lift, such as sending email, calling URLs or scheduling tasks.

Many of the recipes in this chapter have code examples in a project at Github: *https://github.com/LiftCookbook/cookbook_around*.

9.1. Sending Plain Text Email

Problem

You want to send a plain text email from your Lift application.

Solution

Use the `Mailer`:

```
import net.liftweb.util.Mailer
import net.liftweb.util.Mailer._

Mailer.sendMail(
  From("you@example.org"),
  Subject("Hello"),
  To("other@example.org"),
  PlainMailBodyType("Hello from Lift") )
```

Discussion

`Mailer` sends the message asynchronously, meaning `sendMail` will return immediately, so you don't have to worry about the time costs of negotiating with an SMTP server. There's also a `blockingSendMail` method if you need to wait.

By default, the SMTP server used will be *localhost*. You can change this by setting the `mail.smtp.host` property. For example, edit *src/mail/resources/props/default.props* and add the line:

```
mail.smtp.host=smtp.example.org
```

The signature of `sendMail` requires a `From`, `Subject` and then any number of `MailTypes`:

- `To`, `CC` and `BCC` — the recipient email address.
- `ReplyTo` — the address mail clients should use for replies.
- `MessageHeader` — key/value pairs to include as headers in the message.
- `PlainMailBodyType` — a plain text email sent with UTF-8 encoding.
- `PlainPlusBodyType` — a plain text email, where you specify the encoding.
- `XHTMLMailBodyType` — for HTML email (Recipe 9.3).
- `XHTMLPlusImages` — for attachments (Recipe 9.5).

In the example above we added two types:`PlainMailBodyType` and `To`. Adding more is as you'd expect:

```
Mailer.sendMail(
  From("you@example.org"),
  Subject("Hello"),
  To("other@example.org"),
  To("someone@example.org"),
  MessageHeader("X-Ignore-This", "true"),
  PlainMailBodyType("Hello from Lift") )
```

The address-like `MailTypes` (`To`, `CC`, `BCC`, `ReplyTo`) can be given an optional "personal name":

```
From("you@example.org", Full("Example Corporation"))
```

This would appear in your mailbox as:

```
From: Example Corporation <you@example.org>
```

The default character set is UTF-8. If you need to change this replace the use of `PlainMailBodyType` with `PlainPlusBodyType("Hello from Lift", "ISO8859_1")`.

See Also

Recipe 9.5 describes email with attachments.

For HTML email, see Recipe 9.3.

9.2. Logging Email Rather Than Sending

Problem

You don't want email sent when developing your Lift application locally, but you do want to see what would have been sent.

Solution

Assign a logging function to `Mailer.devModeSend` in *Boot.scala*:

```
import net.liftweb.util.Mailer._
import javax.mail.internet.{MimeMessage,MimeMultipart}

Mailer.devModeSend.default.set( (m: MimeMessage) =>
  logger.info("Would have sent: "+m.getContent)
)
```

When you send an email with `Mailer`, no SMTP server will be contacted, and instead you'll see output to your log:

```
Would have sent: Hello from Lift
```

Discussion

The key part of this recipe is setting a `MimeMessage => Unit` function on `Mailer.devModeSend`. We happen to be logging, but you can use this function to handle the email any way you want.

The Lift Mailer allows you to control how email is sent at each run mode:

- `devModeSend` — will send email by default.
- `testModeSend` — only log that a message would have been sent.
- `stagingModeSend` — will send email by default.
- `productionModeSend` — will send email by default.
- `pilotModeSend` — will send email by default.
- `profileModeSend` — will send email by default.

The `testModeSend` logs a reference to the `MimeMessage`, meaning your log would show a message like:

```
Sending javax.mail.internet.MimeMessage@4a91a883
```

This recipe has changed the behaviour of `Mailer` when your Lift application is in developer mode (which it is by default). We're logging just the body part of the message.

Java Mail doesn't include a utility to display all the parts of an email, so if you want more information you'll need to roll your own function. For example:

```
def display(m: MimeMessage) : String = {

  val nl = System.getProperty("line.separator")

  val from = "From: "+m.getFrom.map(_.toString).mkString(",")

  val subj = "Subject: "+m.getSubject

  def parts(mm: MimeMultipart) = (0 until mm.getCount).map(mm.getBodyPart)

  val body = m.getContent match {
    case mm: MimeMultipart =>
      val bodyParts = for (part <- parts(mm)) yield part.getContent.toString
      bodyParts.mkString(nl)

    case otherwise => otherwise.toString
  }

  val to = for {
    rt <- List(RecipientType.TO, RecipientType.CC, RecipientType.BCC)
    address <- Option(m.getRecipients(rt)) getOrElse Array()
  } yield rt.toString + ": " + address.toString

  List(from, to.mkString(nl), subj, body) mkString nl
}

Mailer.devModeSend.default.set( (m: MimeMessage) =>
  logger.info("Would have sent: "+display(m))
)
```

This would produce output of the form:

```
Would have sent: From: you@example.org
To: other@example.org
To: someone@example.org
Subject: Hello
Hello from Lift
```

This example `display` function is long mostly straight-forward. The `body` value handles multi-part messages by extracting the each body part. This is triggered when sending more structured emails, such as the HTML emails described in Recipe 9.3.

If you want to debug the mail system while it's actually sending the email, enable the Java Mail debug mode. In *default.props* add:

```
mail.debug=true
```

This produces low-level output from the Java Mail system when email is sent:

```
DEBUG: JavaMail version 1.4.4
DEBUG: successfully loaded resource: /META-INF/javamail.default.providers
DEBUG SMTP: useEhlo true, useAuth false
DEBUG SMTP: trying to connect to host "localhost", port 25, isSSL false
...
```

See Also

Run modes are described at: *https://www.assembla.com/spaces/liftweb/wiki/Run_Modes*.

9.3. Sending HTML email

Problem

You want to send an HTML email from your Lift application.

Solution

Give `Mailer` a `NodeSeq` containing your HTML message:

```
import net.liftweb.util.Mailer
import net.liftweb.util.Mailer._

val msg = <html>
   <head>
     <title>Hello</title>
   </head>
   <body>
    <h1>Hello</h1>
   </body>
  </html>

Mailer.sendMail(
  From("me@example.org"),
  Subject("Hello"),
  To("you@example.org"),
  msg)
```

Discussion

An implicit converts the `NodeSeq` into a `XHTMLMailBodyType`. This ensures the mime type of the email is *text/html*. Despite the name of "XHTML", the message is to converted for transmission using HTML5 semantics.

The character encoding for HTML email, UTF-8, can be changed by setting `mail.charset` in your Lift properties file.

If you want to set both the text and HTML version of a message, supply each body wrapped in the appropriate `BodyType` class:

```
val html = <html>
  <head>
    <title>Hello</title>
  </head>
  <body>
    <h1>Hello!</h1>
  </body>
</html>

var text = "Hello!"

Mailer.sendMail(
  From("me@example.org"),
  Subject("Hello"),
  To("you@example.org"),
  PlainMailBodyType(text),
  XHTMLMailBodyType(html)
)
```

This message would be sent as a *multipart/alternative*:

```
Content-Type: multipart/alternative;
  boundary="----=_Part_1_1197390963.1360226660982"
Date: Thu, 07 Feb 2013 02:44:22 -0600 (CST)

------=_Part_1_1197390963.1360226660982
Content-Type: text/plain; charset=UTF-8
Content-Transfer-Encoding: 7bit

Hello!
------=_Part_1_1197390963.1360226660982
Content-Type: text/html; charset=UTF-8
Content-Transfer-Encoding: 7bit

<html>
      <head>
        <title>Hello</title>
      </head>
      <body>
        <h1>Hello!</h1>
      </body>
    </html>
------=_Part_1_1197390963.1360226660982--
```

When receiving a message with this content, it is up to the mail client to decide which version to show (text or HTML).

See Also

For sending with attachments, see Recipe 9.5.

9.4. Sending Authenticated Email

Problem

You need to authenticate with an SMTP server to send email.

Solution

Set the `Mailer.authenticator` in `Boot` with the credentials for your SMTP server, and enable the `mail.smtp.auth` flag in your Lift properties file.

Modify *Boot.scala* to include:

```
import net.liftweb.util.{Props, Mailer}
import javax.mail.{Authenticator,PasswordAuthentication}

Mailer.authenticator = for {
  user <- Props.get("mail.user")
  pass <- Props.get("mail.password")
} yield new Authenticator {
  override def getPasswordAuthentication =
    new PasswordAuthentication(user,pass)
}
```

In this example we expect the username and password to come from Lift properties, so we need to modify *src/main/resources/props/default.props* to include them:

```
mail.smtp.auth=true
mail.user=me@example.org
mail.password=correct horse battery staple
mail.smtp.host=smtp.sendgrid.net
```

When you send email, the credentials in *default.props* will be used to authenticate with the SMTP server.

Discussion

We've used Lift properties as a way to configure SMTP authentication. This has the benefit of allowing us to enable authentication for just some run modes. For example, if our *default.props* did not contain authentication settings, but our *production.default.props* did, then no authentication would happen in development mode, ensuring we can't accidentally send email outside of a production environment.

You don't have to use a properties file for this: the Lift Mailer also supports JNDI, or you could lookup a username and password some other way and set `Mailer.authenticator` when you have the values.

However, some mail services such as SendGrid do require `mail.smtp.auth=true` to be set, and that should go into your Lift properties file or set as a JVM argument: `-Dmail.smtp.auth=true`.

See Also

As well as `mail.smtp.auth` there are a range of settings to control the Java Mail API. examples include controlling port numbers and timeouts. These are listed at: *http://javamail.kenai.com/nonav/javadocs/com/sun/mail/smtp/package-summary.html*.

9.5. Sending Email with Attachments

Problem

You want to send an email with one or more attachments.

Solution

Use the `Mailer` `XHTMLPlusImages` to package a message with attachments.

Suppose we want to construct a CSV file and send it via email:

```
val content = "Planet,Discoverer\r\n" +
  "HR 8799 c, Marois et al\r\n" +
  "Kepler-22b, Kepler Science Team\r\n"

case class CSVFile(bytes: Array[Byte],
  filename: String = "file.csv",
  mime: String = "text/csv; charset=utf8; header=present" )

val attach = CSVFile(content.mkString.getBytes("utf8"))

val body = <p>Please research the enclosed.</p>

val msg = XHTMLPlusImages(body,
  PlusImageHolder(attach.filename, attach.mime, attach.bytes))

Mailer.sendMail(
  From("me@example.org",
  Subject("Planets"),
  To("you@example.org"),
  msg)
```

What's happening here is that our message is an `XHTMLPlusImages` instance, which accepts a body message and attachment. The attachment, the `PlusImageHolder`, is an `Array[Byte]`, mime-type and a filename.

Discussion

`XHTMLPlusImages` can also accept more than one `PlusImageHolder` if you have more than one file to attach. Although the name `PlusImageHolder` may suggest it is for attachment images, you can attach any kind of data as an `Array[Byte]` with an appropriate mime type.

By default the attachment is sent with an "inline" disposition. This controls the `Content-Disposition` header in the message, and "inline" means the content is intended for display automatically when the message is shown. The alternative is "attachment", and this can be indicated with a option final parameter to `PlusImageHolder`:

```
PlusImageHolder(attach.filename, attach.mime, attach.bytes, attachment=true)
```

In reality, the mail client will display the message how it wants to, but this extra parameter may give you a little more control.

To attach a pre-made file, you can use `LiftRules.loadResource` to fetch content from the classpath. As an example, if our project contained a file called *Kepler-22b_System_Diagram.jpg* in the *src/main/resources/* folder, we could load and attach it like this:

```
val filename = "Kepler-22b_System_Diagram.jpg"

val msg =
  for ( bytes <- LiftRules.loadResource("/"+filename) )
  yield XHTMLPlusImages(
    <p>Please research this planet.</p>,
    PlusImageHolder(filename, "image/jpg", bytes) )

msg match {
  case Full(m) =>
    Mailer.sendMail(
      From("me@example.org"),
      Subject("Planet attachment"),
      To("you@example.org"),
      m)

  case _ =>
    logger.error("Planet file not found")
}
```

As the contents of *src/main/resources* is included on the classpath, we pass the filename to `loadResource` with a leading `/` character so the file can be found at the right place on the classpath.

The `loadResource` returns a `Box[Array[Byte]]` as we have no guarantee the file will exist. We map this to a `Box[XHTMLPlusImages]` and match on that result to either send the email, or log that the file wasn't found.

See Also

Messages are sent using the *multipart/related* mime heading, with an "inline" disposition. Lift ticket #1197 links to a discussion regarding "multipart/mixed" which may be preferable for working around issues with Microsoft Exchange. See: *https://github.com/lift/framework/issues/1197*.

RFC 2183 describes the "Content-Disposition" header: *http://www.ietf.org/rfc/rfc2183.txt*.

9.6. Run a Task Later

Problem

You want to schedule code to run at some future time.

Solution

Use `net.liftweb.util.Schedule`:

```
import net.liftweb.util.Schedule
import net.liftweb.util.Helpers._

Schedule(() => println("doing it"), 30 seconds)
```

This would cause "doing it" to be printed on the console 30 seconds from now.

Discussion

The signature for `Schedule` used above expects a function of type `() => Unit`, which is the thing we want to happen in the future, and a `TimeSpan` from Lift's `TimeHelpers` which is when we want it to happen. The `30 seconds` value gives us a `TimeSpan` via the `Helpers._` import, but there's a variation called `perform` which accepts a `Long` millisecond value if you prefer that:

```
Schedule.perform(() => println("doing it"), 30*1000L)
```

Behind the scenes, Lift is making use of the `ScheduledExecutorService` from `java.util.concurrent`, and as such returns a `ScheduledFuture[Unit]`. You can use this future to `cancel` the operation before it runs.

It may be a surprise to find that you can call `Schedule` with just a function as an argument, and not a delay value. This version runs the function immediately, but on a worker

thread. This is a convenient way to asynchronously run other tasks without going to the trouble of creating an actor for the purpose.

There is also a `Schedule.schedule` method which will send a specified actor a specified message after a given delay. This takes a `TimeSpan` delay, but again there's also a `Schedule.perform` version that accepts a `Long` as a delay.

See Also

Recipe 9.7 includes an example of scheduling with actors.

`ScheduledFuture` is documented via the Java Doc for `Future` at: *http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/Future.html*. If you're building complex, low-level, cancellable concurrency functions, it's advisable to have a copy of *Java Concurrency in Practice* close by (Goetz *et al.*, 2006, Addison-Wesley Professional).

9.7. Run Tasks Periodically

Problem

You want a scheduled task to run periodically (repeatedly).

Solution

Use `net.liftweb.util.Schedule` ensuring that you call `schedule` again during your task to re-schedule it. For example, using an actor:

```
import net.liftweb.util.Schedule
import net.liftweb.actor.LiftActor
import net.liftweb.util.Helpers._

object MyScheduledTask extends LiftActor {

  case class DoIt()
  case class Stop()

  private var stopped = false

   def messageHandler = {
     case DoIt if !stopped =>
        Schedule.schedule(this, DoIt, 10 minutes)
       // ... do useful work here

     case Stop =>
       stopped = true
   }
}
```

The example creates a `LiftActor` for the work to be done. On receipt of a `DoIt` message, the actor re-schedules itself before doing whatever useful work needs to be done. In this way, the actor will be called every 10 minutes.

Discussion

The `Schedule.schedule` call is ensuring that `this` actor is sent the `DoIt` message after 10 minutes.

To start this process off, possibly in *Boot.scala*, just send the `DoIt` message to the actor:

```
MyScheduledTask ! MyScheduledTask.DoIt
```

To ensure the process stops correctly when Lift shuts down, we register a shutdown hook in *Boot.scala* to send the `Stop` message to prevent future re-schedules:

```
LiftRules.unloadHooks.append( () => MyScheduledTask ! MyScheduledTask.Stop )
```

Without the `Stop` message the actor would continue to be rescheduled until the JVM exits. This may be acceptable, but note that during development with SBT, without the `Stop` message, you will continue to schedule tasks after issuing the `container:stop` command.

Schedule returns a `ScheduledFuture[Unit]` from the Java concurrency library, which allows you to `cancel` the activity.

See Also

Chapter 1 of *Lift in Action* (Perrett, 2011, Manning Publications Co) includes a Comet Actor clock example that uses `Schedule`.

9.8. Fetching URLs

Problem

You want your Lift application to fetch a URL, and process it as text, JSON, XML or HTML.

Solution

Use *Dispatch*, "a library for asynchronous HTTP interaction".

Before you start, include Dispatch dependency in *build.sbt*:

```
libraryDependencies += "net.databinder.dispatch" %% "dispatch-core" % "0.9.5"
```

Using the example from the Dispatch documentation, we can make a HTTP request to try to determine the country from the service at *http://www.hostip.info/use.html*:

```
import dispatch._
val svc = url("http://api.hostip.info/country.php")
val country : Promise[String] = Http(svc OK as.String)

println(country())
```

Note that the result `country` is not a `String` but a `Promise[String]`, and we `apply` to wait for the resulting value.

The result printed will be a country code such as `GB`, or `XX` if the country cannot be determined from your IP address.

Discussion

This short example expects a 200 (OK) status result and turns the result into a `String`, but that's a tiny part of what Dispatch is capable of. We'll explore further in this section.

What if the request doesn't return a 200? In that case, with the code we have, we'd get an exception such as: "Unexpected response status: 404". There are a few ways to change that.

We can ask for an `Option`:

```
val result : Option[String] = country.option()
```

As you'd expect, this will give a `None` or `Some[String]`. However, if you have debug level logging enabled in your application you'll see the request and response and error messages from the underlying Netty library. You can tune these messages by adding a logger setting to *default.logback.xml* file:

```
<logger name="com.ning.http.client" level="WARN"/>
```

A second possibility is to use `either` with the usual convention that the `Right` is the expected result and `Left` signifies a failure:

```
country.either() match {
  case Left(status) => println(status.getMessage)
  case Right(cc) => println(cc)
}
```

This will print a result as we are forcing the evaluation with an apply via `either()`.

`Promise[T]` implements `map`, `flatMap`, `filter`, `fold` and all the usual methods you'd expect to allow you to compose. This means you can use the promise with a for comprehension:

```
val codeLength = for (cc <- country) yield cc.length
```

Note that `codeLength` is a `Promise[Int]`. To get the value you can evaluate `codeLength()` and you'll get a result of `2`.

As well as extracting string values with `as.String` there are other options, including…

- `as.Bytes` — to work with `Promise[Array[Byte]]`.
- `as.File` — to write to a file, as in `Http(svc > as.File(new File("/tmp/cc")) )`.
- `as.Response` — to allow you to provide a `client.Response => T` function to use on the response.
- `as.xml.Elem` — to parse XML response.

As an example of `as.xml.Elem`:

```
val svc = url("http://api.hostip.info/?ip=12.215.42.19")
val country  = Http(svc > as.xml.Elem)
println(country.map(_ \\ "description")())
```

This example is parsing the XML response to the request, which returns a `Promise[scala.xml.Elem]`. We're picking out the description node of the XML via a `map`, which will be a `Promise[NodeSeq]` which we then force to evaluate. The output is something like:

```
<gml:description
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:gml="http://www.opengis.net/gml">
     This is the Hostip Lookup Service
</gml:description>
```

That example assumes the request is going to be well-formed. In addition to the core Databinder library, there are extensions for JSoup and TagSoup to assist in parsing HTML that isn't necessarily well-formed.

For example, to use JSoup, include the dependency:

```
libraryDependencies += "net.databinder.dispatch" %% "dispatch-jsoup" % "0.9.5"
```

You can then use the features of JSoup, such as picking out elements of a page using CSS selectors:

```
import org.jsoup.nodes.Document

val svc = url("http://www.example.org").setFollowRedirects(true)
val title = Http(svc > as.jsoup.Document).map(_.select("h1").text).option
println( title() getOrElse "unknown title" )
```

Here we are applying JSoup's `select` function to pick out the `<h1>` element on the page, taking the text of the element which we turn into a `Promise[Option[String]]`. The result, unless *example.org* has changed, will be "Example Domain".

As a final example of using Dispatch, we can pipe a request into Lift's JSON library:

```
import net.liftweb.json._
import com.ning.http.client

object asJson extends (client.Response => JValue) {
  def apply(r: client.Response) = JsonParser.parse(r.getResponseBody)
```

```
}

val svc = url("http://api.hostip.info/get_json.php?ip=212.58.241.131")
val json : Promise[JValue] = Http(svc > asJson)

case class HostInfo(country_name: String, country_code: String)
implicit val formats = DefaultFormats

val hostInfo = json.map(_.extract[HostInfo])()
```

The URL we're calling returns a JSON representation for location information of the IP address we've passed.

By providing a `Response => JValue` to Dispatch we're able to pass the response body through to the JSON parser. We can then map on the `Promise[JValue]` to apply whatever Lift JSON functions we want to. In this case, we're extracting a simple case class.

The result from the above would show `hostInfo` as:

```
HostInfo(UNITED KINGDOM,GB)
```

See Also

The Dispatch documentation is well-written and guides you through the way Dispatch approaches HTTP. Do spend some time with it at: *http://dispatch.databinder.net/Dispatch.html*.

To see what's available on Dispatch's `Promise`, browse the source: *https://github.com/dispatch/reboot/blob/master/core/src/main/scala/promise.scala*.

For questions about Dispatch, the best place is the Dispatch Google Group: *https://groups.google.com/forum/!forum/dispatch-scala*.

The previous major version of Dispatch, 0.8.x ("Dispatch Classic"), is quite different from the "reboot" of the project as version 0.9. Consequently, examples you may see that use 0.8.x will need some conversion to run with 0.9.x. Nathan Hamblen's blog describes the change: *http://code.technically.us/post/17038250904/fables-of-the-reconstruction-part-2-have-you-tried*.

For working with JSoup, take a look at the JSoup Cookbook at: *http://jsoup.org/cookbook/*.

CHAPTER 10

Production Deployment

Deploying a Lift application to production means little more than packaging it and ensuring you set the *run mode* to `production`. The recipes in this chapter show how to do this for various hosted services.

You can also install and run a *container* such as Tomcat or Jetty on your own servers. Containers were introduced in "Running Your Application" on page 3. This brings with it the need to understand how to install, configure, start, stop and manage each container, and how to integrate it with load balancers or other front-ends. These are large topics, and you can find out more from such sources as:

- The deployment section of the Lift Wiki at *https://www.assembla.com/spaces/liftweb/wiki/Deployment*.
- Timothy Perrett (2012), *Lift in Action*, Chapter 15, "Deployment and scaling", Manning Publications Co.
- Jason Brittain and Ian F. Darwin (2007), *Tomcat: The Definitive Guide*, O'Reilly Media, Inc.
- Tanuj Khare (2012) *Apache Tomcat 7 Essentials*, Packt Publishing.

The Lift wiki includes a page on Tomcat configuration options relevant to Lift: *https://www.assembla.com/spaces/liftweb/wiki/Tomcat*.

10.1. Deploying to CloudBees

Problem

You have an account with the CloudBees PaaS hosting environment, and you want to deploy your Lift application there.

Solution

Use the SBT `package` command to produce a WAR file that can be deployed to CloudBees, and then use the CloudBees SDK to configure and deploy your application.

From within the CloudBees "Grand Central" console, create a new application under your account. In what follows we'll assume your account is called *myaccount* and your application is called *myapp*.

For the best performance you will want to ensure the Lift run mode is set to "production". Do this from the CloudBees SDK command line:

```
$ bees config:set -a myaccount/myapp run.mode=production
```

This will set the run mode to production for your CloudBees applications identified as "myaccount/myapp". Omitting the `-a` will set it for your whole CloudBees account.

CloudBees will remember this setting, so you only need to do it once.

You can then deploy:

```
$ sbt package
...
[info] Packaging /Users/richard/myapp/target/scala-2.9.1/myapp.war...
...
$ bees app:deploy -a myaccount/myapp ./target/scala-2.9.1/myapp.war
```

This will send your WAR file to CloudBees and deploy it. You'll see the location (URL) of your application output from the bees `app:deploy` command when it completes.

If you change a configuration setting, you will need to restart the application for setting to take effect. Deploying the application will do this, otherwise run the `bees app:restart` command:

```
$ bees app:restart -a myaccount/myapp
```

Discussion

If you are deploying an application to multiple CloudBees instances, be aware that by default CloudBees will round robin requests to each instance. If you use any of Lift's state features you'll want to enable session affinity (sticky sessions):

```
$ bees app:update -a myaccount/myapp stickySession=true
```

If you are using comet, it'll work fine, but the CloudBees default is to enable *request buffering*. This allows CloudBees to do smart things, such as re-routing requests in a cluster if one machine does not respond. A consequence of request buffering is that long-polling comet requests will timeout more often. To turn this feature off, run the following:

```
$ bees app:update -a myaccount/myapp disableProxyBuffering=true
```

As with the run mode setting, CloudBees will remember these settings, so you only need to set them once.

Finally, you may want to increase the *permanent generation* memory setting of the JVM. By default, an application has 64M assigned for the permgen. To increase this to 128M, run the `bees app:update` commannd:

```
$ bees app:update -a myaccount/myapp jvmPermSize=128
```

The commands `bees app:info` and `bees config:list` will report back the settings for your application.

RDBMS Configuration

If you are using a SQL database in your application, you'll want to configure *src/main/webapp/WEB-INF/cloudbees-web.xml*. For example:

```
<?xml version="1.0"?>
<cloudbees-web-app xmlns="http://www.cloudbees.com/xml/webapp/1">

<appid>myaccount/myapp</appid>

<resource name="jdbc/mydb" auth="Container" type="javax.sql.DataSource">
  <param name="username" value="dbuser" />
  <param name="password" value="dbpassword" />
  <param name="url" value="jdbc:cloudbees://mydb" />

  <!-- For these connections settings, see:
   http://commons.apache.org/dbcp/configuration.html
  -->
  <param name="maxActive" value="10" />
  <param name="maxIdle" value="2" />
  <param name="maxWait" value="15000" />
  <param name="removeAbandoned" value="true" />
  <param name="removeAbandonedTimeout" value="300" />
  <param name="logAbandoned" value="true" />

  <!-- Avoid idle timeouts -->
  <param name="validationQuery" value="SELECT 1" />
  <param name="testOnBorrow" value="true" />

 </resource>

</cloudbees-web-app>
```

The above is a JNDI database configuration, defining a connection to a CloudBees database called "mydb". This will be used by Lift if the JNDI name is referenced in *Boot.scala*:

```
DefaultConnectionIdentifier.jndiName = "jdbc/mydb"

if (!DB.jndiJdbcConnAvailable_?) {
```

```
    // set up alternative local database connection here
}
```

Because the JDNI setting is only defined in *cloudbees-web.xml* it will only be available in a CloudBees environment. This means you can develop against a different database locally, and use your CloudBees database when deploying.

Host IP and Port Number

Generally you don't need to know about your deployed instance's public host name and port number. Requests to your application URL are routed to the specific instance by CloudBees. However there are situations, especially when you have multiple instances, where you do need to find this out. For example, if you want to receive messages from Amazon's Simple Notification Service (SNS), then each instance will need to give a direct URL to SNS when the application boots.

To get the public hostname, you need to make a HTTP request to *http://instance-data/latest/meta-data/public-hostname*, as documented at *https://developer.cloudbees.com/bin/view/Main/Finding+out+app+port+and+hostname*. For example:

```
import io.Source

val beesPublicHostname : Box[String] = tryo {
  Source.fromURL("http://instance-data/latest/meta-data/public-hostname").
    getLines().toStream.head
}
```

This will return a `Full` hostname on the CloudBees environment, but when running locally will fail and return a `Failure`. For example:

```
Failure(instance-data,Full(java.net.UnknownHostException: instance-data),Empty)
```

The port number can be found from the name of a file in the *.genapps/ports* folder of your application deployment:

```
val beesPort : Option[Int] = {
  val portsDir = new File(System.getenv("PWD"), ".genapp/ports")
  for {
    files <- Option(portsDir.list)
    port <- files.flatMap(asInt).headOption
  } yield port
}
```

The `java.io.File#list` method returns a list of filenames in a directory, but will return `null` if the directory doesn't exist or if there are any IO errors. For this reason, we wrap it in `Option` to convert `null` values to `None`.

Running locally this will return a `None`, but on CloudBees you'll see a `Full[Int]` port number.

You might put these two values together as follows:

```
import java.net.InetAddress

val hostAndPort : String =
  (beesPublicHostname openOr InetAddress.getLocalHost.getHostAddress) +
  ":" + (beesPort getOrElse 8080).toString
```

Running locally `hostAndPort` might be "192.168.1.60:8080" and running on CloudBees it would be something like "ec2-204-236-222-252.compute-1.amazonaws.com:8520".

Java Version

Currently the default JVM provided by CloudBees is JDK 7, but you can select 6, 7 and 8. To change the default Java Virtual Machine, use the `bees config:set` command:

```
$ bees config:set -a myaccount/myapp -Rjava_version=1.8
```

Excluding the application identifier `-a myaccount/myapp` from the command will set the JVM as the default for all applications in the account. The `bees config:set` command will update the configuration, but not take effect until the application(s) have been updated or restarted.

The JVM can also be changed when an application is deployed or updated via the following commands:

```
$ bees app:deploy -a myaccount/myapp sample.war -Rjava_version=1.6
$ bees app:update -a myaccount/myapp -Rjava_version=1.7
```

To confirm which JVM an application is currently running, use the `bees config:list` command, which will display Java version. As can be seen below:

```
$ bees config:list -a myaccount/myapp
Runtime Parameters:
  java_version=1.6
```

Container Version

CloudBees offer several containers: Tomcat 6.0.32 (the default), Tomcat 7, JBoss 7.02, JBoss 7.1 and GlassFish 3.

To change containers the application will need to be redeployed as CloudBees uses different file configurations for the various containers. Hence we use the `bees app:deploy` command. The following example updates to Tomcat 7:

```
$ bees app:deploy -t tomcat7 -a myaccount/myapp sample.war
```

The JVM and container commands can be run as a single `bees app:deploy` as follows:

```
$ bees app:deploy -t tomcat -a myaccount/myapp sample.war -Rjava_version=1.6
```

This would deploy *sample.war* to the "myapp" application on "myaccount" with Tomcat 6.0.32 and JDK 6.

To determine which container an application is deployed to, use the command `bees app:info`:

```
$ bees app:info -a myaccount/myapp
Application     : myaccount/myapp
Title           : myapp
Created         : Wed Mar 20 11:02:40 EST 2013
Status          : active
URL             : myapp.myaccount.cloudbees.net
clusterSize     : 1
container       : java_free
containerType   : tomcat
idleTimeout     : 21600
maxMemory       : 256
proxyBuffering  : false
securityMode    : PUBLIC
serverPool      : stax-global (Stax Global Pool)
```

ClickStarts

ClickStart Applications are templates to quickly get an application, and automated build, up and running at CloudBees. The Lift ClickStart creates a private Git source repository at CloudBees which contains a Lift 2.4 application, provisions a MySQL database, creates a Maven-based Jenkins build, and deploys the application. All you need to do is provide a name for the application (without whitespace).

To access the Git source repository created for you, you'll need to upload a SSH public key. You can do this in the "My Keys" section of your account settings on the CloudBees web site.

The build that's created for you will automatically build and deploy your application to CloudBees when you push changes to your Git repository.

If all of that's a good match to the technologies and services you want to use, ClickStart is a great way to deploy your application. Alternatively, it gives you a starting point from which you can modify elements; or you could fork the CloudBees Lift template and create your own from *https://github.com/CloudBees-community/lift_template*.

See Also

The CloudBees SDK provides command line tools for configuring and controlling applications. It can be found at *https://wiki.cloudbees.com/bin/view/RUN/BeesSDK*.

The CloudBees developer portal (*https://developer.cloudbees.com*) contains a "Resources" section which provides details of the CloudBees services.

The JVM PermGen settings for CloudBees are described at *https://wiki.cloudbees.com/bin/view/RUN/JVM+PermGen+Space*, and settings for which JVM is used can be found at *https://developer.cloudbees.com/bin/view/RUN/JVMVersion*. For information about the containers, see: *https://developer.cloudbees.com/bin/view/RUN/ClickStack*.

10.2. Deploying to Amazon Elastic Beanstalk

Problem

You want to run your Lift application on Amazon Web Services (AWS) Elastic Beanstalk.

Solution

Create a new Tomcat 7 *environment*, use SBT to package your Lift application as a WAR file, and then deploy the application to your environment.

To create a new environment, visit the AWS console, navigate to Elastic Beanstalk and select "Apache Tomcat 7" as your environment. This will create and launch a default Beanstalk application. This will take a few minutes, but eventually report "Successfully running version Sample Application". You'll be shown the URL of the application (something like *http://default-environment-nsdmixm7ja.elasticbeanstalk.com*) and visiting the URL you're given will show the running default Amazon application.

Prepare your WAR file by running:

```
$ sbt package
```

This will write a WAR file into the *target* folder. To deploy this WAR file from the AWS Beanstalk web console (see Figure 10-1), select the "Versions" tab under the "Elastic Beanstalk Application Details" and click the "Upload new version" button. You'll be given a dialog where you give a version label and use the "Choose file" button to select the WAR file you just built. You can either upload and deploy in one step, or upload first and then select the version in the console and hit the "Deploy" button.

The Beanstalk console will show "Environment updating…" and after some minutes it'll report "Successfully running". Your Lift application is now deployed and running on Beanstalk.

A final step is to enable Lift's production run mode. From the environment in the AWS Beanstalk web console, follow the "Edit Configuration" link. A dialog will appear, and under the "Container" tab add `-Drun.mode=production` to the "JVM Command Line Options" and hit "Apply Changes" to redeploy your application.

Figure 10-1. AWS Console, with Elastic Beanstalk service selected.

Discussion

Elastic Beanstalk provides a pre-built stack of software and infrastructure, in this case: Linux, Tomcat 7, a 64 bit "t1.micro" EC2 instance, load balancing, and an S3 bucket. That's the *environment* and it has reasonable default settings. Beanstalk also provides an easy way to deploy your Lift application. As we've seen in this recipe, you upload an application (WAR file) to Beanstalk and deploy it to the environment.

As with many cloud providers keep in mind that you want to avoid local file storage. The reason for this is to allow instances to be terminated or restarted without data loss. With your Beanstalk application you do have a file system and you can write to it, but it is lost if the image is restarted. You can get persistent local file storage, for example using Amazon Elastic Block Storage, but you're fighting against the nature of the platform.

Log files are written to the local file system. To access them, from the AWS console, navigate to your environment, into the "Logs" tab and hit the "Snapshot" button. This will take a copy of the logs and store them in an S3 bucket, and give you a link to the file contents. This is a single file showing the content of variety of log files, and *catalina.out* will be the one showing any output from your Lift application. If you want to try to keep these log files around, you can configure the environment to rotate the logs to S3 every hour from the "Container" tab under "Edit Configuration".

The Lift application WAR files are stored in the same S3 bucket that the logs are stored in. From the AWS console, you'll find it under the S3 page listed with a name like "elasticbeanstalk-us-east-1-5989673916964". You'll note that the AWS uploads makes your WAR filename unique by adding a prefix to each filename. If you need to be able to tell the difference between these files in S3, one good approach is to is to bump the `version` value in your *build.sbt* file. This version number is included in the WAR filename.

Multiple Instances

Beanstalks enables *auto scaling* by default. That is, it launches a single instance of your Lift application, but if the load increases above a threshold, up to four instances may be running.

If you're making use of Lift's state features, you'll need to enable sticky sessions from the "Load Balancer" tab of the environment configuration. It's a check box named "En-

able Session Stickiness" — it's easy to miss, but that tab does scroll to show more options if you don't see it first time.

Working with a Database

There's nothing unusual you have to do to use Lift and a database from Beanstalk. However, Beanstalk does try to make it easy for you to work with Amazon's Relational Database Service (RDS). Either when creating your Beanstalk environment, or from the configuration options later, you can add an RDS instance, which can be an Oracle, SQL-Server or MySQL database.

The MySQL option will create a MySQL 5.5 InnoDB database. The database will be accessible from Beanstalk, but not from elsewhere on the Internet. To change that, modify the security groups for the RDS instance from the AWS web console. For example, you might permit access from your IP address.

When your application launches with an associated RDS instance, the JVM system properties include settings for the database name, host, port, user and password. You could pull them together like this in *Boot.scala*:

```
Class.forName("com.mysql.jdbc.Driver")

val connection = for {
  host <- Box !! System.getProperty("RDS_HOSTNAME")
  port <- Box !! System.getProperty("RDS_PORT")
  db   <- Box !! System.getProperty("RDS_DB_NAME")
  user <- Box !! System.getProperty("RDS_USERNAME")
  pass <- Box !! System.getProperty("RDS_PASSWORD")
} yield DriverManager.getConnection(
    "jdbc:mysql://%s:%s/%s" format (host,port,db),
    user, pass)
```

That would give you a `Box[Connection]` which, if `Full`, you could use in a `SquerylRecord.initWithSquerylSession` call for example (see Chapter 7).

Alternatively you might want to guarantee a connection by supplying defaults for all the values with something like this:

```
Class.forName("com.mysql.jdbc.Driver")

val connection = {
  val host = System.getProperty("RDS_HOSTNAME", "localhost")
  val port = System.getProperty("RDS_PORT", "3306")
  val db = System.getProperty("RDS_DB_NAME", "db")
  val user = System.getProperty("RDS_USERNAME", "sa")
  val pass = System.getProperty("RDS_PASSWORD", "")

  DriverManager.getConnection(
    "jdbc:mysql://%s:%s/%s" format (host,port,db),
    user, pass)
}
```

See Also

Amazon provide a walk-through with screen shots, showing how to create a Beanstalk application. It's at: *http://docs.amazonwebservices.com/elasticbeanstalk/latest/dg/GettingStarted.Walkthrough.html*.

Elastic Beanstalk, by van Villet *et al* (2011, O'Reilly Media, Inc) goes into the details of the Beanstalk infrastructure, how to work with Eclipse, enabling continuous integration, and how to hack the instance, for example to use NGINX as a front-end to Beanstalk.

The Amazon documentation for "Configuring Databases with AWS Elastic Beanstalk" describes the RDS settings in more detail: *http://docs.amazonwebservices.com/elasticbeanstalk/latest/dg/using-features.managing.db.html*.

10.3. Deploying to Heroku

Problem

You want to deploy your Lift application to your account on the Heroku cloud platform.

Solution

Package your Lift application as a WAR file and use the Heroku deploy plugin to send and run your application. This will give you an application running under Tomcat 7. Anyone can use this method to deploy an application but Heroku only provide support for it for Enterprise Java customers.

This recipe walks through the process in three stages: one-time set up; deployment of the WAR; and configuration of your Lift application for production performance.

If you've not already done so, download and install the Heroku command line tools ("Toolbelt") and login using your Heuroku credentials and upload an SSH key:

```
$ heroku login
Enter your Heroku credentials.
Email: you@example.org
Password (typing will be hidden):
Found the following SSH public keys:
1) github.pub
2) id_rsa.pub
Which would you like to use with your Heroku account? 2
Uploading SSH public key ~/.ssh/id_rsa.pub... done
Authentication successful.
```

Install the deploy plugin:

```
$ heroku plugins:install https://github.com/heroku/heroku-deploy
Installing heroku-deploy... done
```

With that one-time set up complete, you can create an application on Heroku. Here we've not specified a name so we given a random name of "glacial-waters-6292" which we will use throughout this recipe:

```
$ heroku create
Creating glacial-waters-6292... done, stack is cedar
http://glacial-waters-6292.herokuapp.com/       |       git@heroku.com:glacial-
waters-6292.git
```

Before deploying, we set the Lift run mode to production. This is done via the `config:set` command. First check the current settings for `JAVA_OPTS` and then modify the options by adding `-Drun.mode=production`:

```
$ heroku config:get JAVA_OPTS --app glacial-waters-6292
-Xmx384m -Xss512k -XX:+UseCompressedOops

$ heroku config:set JAVA_OPTS="-Drun.mode=production -Xmx384m -Xss512k
  -XX:+UseCompressedOops" --app glacial-waters-6292
```

We can deploy to Heroku by packaging the application as a WAR file, and then running the Heroku `deploy:war` command:

```
$ sbt package
....
[info] Packaging target/scala-2.9.1/myapp-0.0.1.war ...
....
$ heroku deploy:war --war target/scala-2.9.1/myapp-0.0.1.war
  --app glacial-waters-6292
Uploading target/scala-2.9.1/myapp-0.0.1.war............done
Deploying to glacial-waters-6292.........done
Created release v6
```

Your Lift application is now running on Heroku.

Discussion

There are a few important comments regarding Lift applications on Heroku. First, note that there's no support for session affinity. This means if you deploy to multiple *dynos* (Heroku terminology for instances), there is no co-ordination over which requests go to which servers. As a consequence, you won't be able to make use of Lift's stateful features and will want to turn them off (Recipe 6.4 describes for how to do that).

Second, if you are using Lift comet features, there's an adjustment to make in *Boot.scala* to work a little better in the Heroku environment:

```
LiftRules.cometRequestTimeout = Full(25)
```

This setting controls how long Lift waits before testing a comet connection. We're replacing the Lift default of 120 seconds with 25 seconds because Heroku terminates connections after 30 seconds. Although Lift recovers from this, the user experience may be to see a delay when interacting with a page.

A third important point to note is that the dyno will be restarted every day. Additionally, if you are only running one web dyno, it will be idled after an hour of inactivity. You can see this happening by tailing your application log:

```
$ heroku logs -t --app glacial-waters-6292
...
2012-12-31T11:31:39+00:00 heroku[web.1]: Idling
2012-12-31T11:31:41+00:00 heroku[web.1]: Stopping all processes with SIGTERM
2012-12-31T11:31:43+00:00 heroku[web.1]: Process exited with status 143
2012-12-31T11:31:43+00:00 heroku[web.1]: State changed from up to down
```

Anyone visiting your Lift application will cause Heroku to unidle your application.

Note, though, that the application was stopped with a *SIGTERM*. This is a Unix signal sent to a process, the JVM in this case, to request it to stop. Unfortunately the Tomcat application on Heroku does not use this signal to request Lift to shutdown. This may be of little consequence to you, but if you do have external resources you want to release to other actions to take at shutdown, you need to register a shutdown hook with the JVM.

For example, you might add this to *Boot.scala* if you're running on Heroku:

```
Runtime.getRuntime().addShutdownHook(new Thread {
  override def run() {
    println("Shutdown hook being called")
    // Do useful clean up here
  }
})
```

Do not count on being able to do much during shutdown. Heroku allows around 10 seconds before killing the JVM after issuing the *SIGTERM*.

Possibly a more general approach is to perform clean up using Lift's unload hooks (see Recipe 6.3) and then arrange the hooks to be called when Heroku sends the signal to terminate:

```
Runtime.getRuntime().addShutdownHook(new Thread {
  override def run() {
    LiftRules.unloadHooks.toList.foreach{ f => tryo { f() } }
  }
})
```

This handling of `SIGTERM` may be a surprise, but if we look at how the application is running on Heroku, things become clearer. The dyno is an allocation of resources (512m of memory) and allows an arbitrary command to run. The command being run is a Java process starting a "webapp runner" package. You can see this in two ways. First, if you shell to your dyno, you'll see a WAR file as well as a JAR file:

```
$ heroku run bash --app glacial-waters-6292
Running `bash` attached to terminal... up, run.8802
```

```
~ $ ls
Procfile  myapp-0.0.1.war  webapp-runner-7.0.29.3.jar
```

Second, by looking at the processes executing:

```
$ heroku ps --app glacial-waters-6292
=== web: `${PRE_JAVA}java ${JAVA_OPTS} -jar webapp-runner-7.0.29.3.jar
 --port ${PORT} ${WEBAPP_RUNNER_OPTS} myapp-0.0.1.war`
web.1: up 2013/01/01 22:37:35 (~ 31s ago)
```

Here we see a Java process executing a JAR file called *webapp-runner-7.0.29.3.jar* which is passed our WAR file as an argument. This is not identical to the Tomcat *catalina.sh* script you may be more familiar with, but instead is this launcher process: *https://github.com/jsimone/webapp-runner*. As it does not register a handler to deal with *SIGTERM*, so we will have to if we need to release any resources during shutdown.

All of this means that if you want to launch a Lift application in a different way, you can. You'd need to wrap an appropriate container (Jetty or Tomcat for example), and provide a `main` method for Heroku to call. This is sometimes called *containerless deployment*.

If you are not a Heruoku Enterprise Java customer, and you're uncomfortable with the unsupported nature of the `deploy:war` plugin, you now know what you need to do to run in a supported way: provide a `main` method that launches you application and listen for connections. The *See Also* section gives pointers for how to do this.

Database Access in Heroku

Heroku make no restrictions on which databases you can connect to from your Lift application, but they try to make it easy to use their PostgreSQL service by attaching a free database to applications you create.

You can find out if you have a database by running the `pg` command:

```
$ heroku pg --app glacial-waters-6292
=== HEROKU_POSTGRESQL_BLACK_URL (DATABASE_URL)
Plan:        Dev
Status:      available
Connections: 0
PG Version:  9.1.6
Created:     2012-12-31 10:02 UTC
Data Size:   5.9 MB
Tables:      0
Rows:        0/10000 (In compliance)
Fork/Follow: Unsupported
```

The URL of the database is provided to your Lift application as the `DATABASE_URL` environment variable. It will have a value something like this:

```
postgres://gghetjutddgr:RNC_lINakkk899HHYEFUppwG@ec2-54-243-230-119.compute-1.
 amazonaws.com:5432/d44nsahps11hda
```

This URL contains a user name, password, host and database name, but needs to be manipulated to be used by JDBC. To do so, you might include the following in *Boot.scala*:

```
 Box !! System.getenv("DATABASE_URL") match {
  case Full(url) => initHerokuDb(url)
  case _ => // configure local database perhaps
}

def initHerokuDb(dbInfo: String) {
  Class.forName("org.postgresql.Driver")

  // Extract credentials from Heroku database URL:
  val dbUri = new URI(dbInfo)
  val Array(user, pass) = dbUri.getUserInfo.split(":")

  // Construct JDBC connection string from the URI:
  def connection = DriverManager.getConnection(
    "jdbc:postgresql://" + dbUri.getHost + ':' + dbUri.getPort +
      dbUri.getPath, user, pass)

  SquerylRecord.initWithSquerylSession(
    Session.create(connection, new PostgreSqlAdapter))

}
```

Here we are testing for the presence of the `DATABASE_URL` environment variable, which would indicate that we are in the Heroku environment. We can then extract out the connection information to use in `Session.create`. We would additionally need to complete the usual `addAround` configuration described in Recipe 7.1.

For it to run *build.sbt* needs the appropriate dependencies for Record and PostgresSQL:

```
...
"postgresql" % "postgresql" % "9.1-901.jdbc4",
"net.liftweb" %% "lift-record" % liftVersion,
"net.liftweb" %% "lift-squeryl-record" % liftVersion,
...
```

With this in place, your Lift application can make use of the Heroku database. You can also access the database from the shell, e.g.,:

```
$ pg:psql --app glacial-waters-6292
psql (9.1.4, server 9.1.6)
SSL connection (cipher: DHE-RSA-AES256-SHA, bits: 256)
Type "help" for help.

d44nsahps11hda=> \d
No relations found.
d44nsahps11hda=> \q
$
```

To access via a JDBC tool outside of the Heroku environment, you'll need to include parameters to force SSL. For example:

```
jdbc:postgresql://ec2-54-243-230-119.compute-1.amazonaws.com:5432/
d44nsahps11hda?
  username=gghetjutddgr&password=RNC_lINakkk899HHYEFUppwG&ssl=true&sslfactory=
  org.postgresql.ssl.NonValidatingFactory
```

See Also

Both the Scala and Java articles at Heroku are useful to learn more of the details described in this recipe: *https://devcenter.heroku.com/categories/scala* and *https://devcenter.heroku.com/categories/java*.

Dynos and the Dyno Manifold are described at: *https://devcenter.heroku.com/articles/dynos*.

The JVM shutdown hooks are described in the JDK documentation: *http://docs.oracle.com/javase/7/docs/api/java/lang/Runtime.html*.

Heroku's guide to containerless deployment makes use of Maven to package your application, as is documented at *https://devcenter.heroku.com/articles/java-webapp-runner*. There are also a template SBT project from Matthew Henderson that includes a `JettyLauncher` class: *https://github.com/ghostm/lift_blank_heroku*.

The way Heroku deal with comet long-poling is described at *https://devcenter.heroku.com/articles/request-timeout*.

10.4. Distributing Comet Across Multiple Servers

Problem

You use Lift's Comet support, and want to run across multiple servers for increased redundancy or to handle increased load.

Solution

Use the *publish/subscribe* (pubsub) model to connect each server to a *topic* and route comet message out to the topic where it can be broadcast to all servers that are part of your application.

There are a variety of technologies you can use to accomplish this, such as databases, message systems, actor systems. For this recipe we will use the RabitMQ message service, but there are examples using CouchDB and Amazon's Simple Notification Service in the *See Also* section.

Regardless of the technology, the principle is that illustrated in Figure 10-2. A comet event originating on one Lift application, is sent to a service for redistribution. It is the responsibility of this service (labeled as "topic" in the figure) to ensure all the participating Lift applications receive the event.

Figure 10-2. Comet events originating on one server are distributed via a topic.

The first step is to download and install RabbitMQ from *http://rabbitmq.com/*. Then start the server:

```
$ ./sbin/rabbitmq-server -detatched
```

This command will produce various messages as it starts but will eventually say: "broker running".

The Lift application we'll use to demonstrate the pubsub pattern is the real-time chat application, described in *Simply Lift*. The first modification is to include the Lift module to talk to RabbitMQ. This is a one-line addition to the `libraryDependencies` in *build.sbt*:

```
"net.liftmodules" %% "amqp_2.5" % "1.3"
```

AMQP stands for Advanced Message Queuing Protocol, a protocol that RabbitMQ talks. The AMQP module provides abstract actors to send and receive messages, and we will implement these actors as `RemoteSend` and `RemoteReceiver`:

```
package code.comet

import net.liftmodules.amqp._
import com.rabbitmq.client._

object Rabbit {

  val factory = new ConnectionFactory {
    import ConnectionFactory._
    setHost("127.0.0.1")
    setPort(DEFAULT_AMQP_PORT)
  }

  val exchange = "lift.chat"
  val routing = ""
  val durable = true
  val autoAck = false

  object RemoteSend extends AMQPSender[String](factory, exchange, routing) {
    def configure(channel: Channel) =
```

```
        channel.exchangeDeclare(exchange, "fanout", durable)
    }

    object RemoteReceiver extends AMQPDispatcher[String](factory) {
      def configure(channel: Channel) = {

        channel.exchangeDeclare(exchange, "fanout", durable)
        val queueName = channel.queueDeclare().getQueue()

        channel.queueBind(queueName, exchange, routing)

        channel.basicConsume(queueName, autoAck,
          new SerializedConsumer(channel, this) )
      }
    }

  }
```

This code is establishing `RemoteSend` and `RemoteReceiver` actors that serialise `String` values via RabbitMQ. This code is explored in the *Discussion* section below.

To make use of this and route comet messages over RabbitMQ we need to make two changes. In *Boot.scala* we need to start listening for messages from RabbitMQ:

```
RemoteReceiver ! AMQPAddListener(ChatServer)
```

This is attaching the `ChatServer` as a listener for AMQP messages from the `RemoteReciver`.

The final change is to the `ChatServer` itself. The regular behaviour of the `ChatServer` is to receive a `String` message from a client and update all the screens attached to the comet server:

```
override def lowPriority = {
  case s : String => msgs :+= s; updateListeners()
}
```

The change to route messages over RabbitMQ is to redirect any `String` from clients to RabbitMQ, and handle any AMQP messages from RabbitMQ and update all clients:

```
override def lowPriority = {
  case AMQPMessage(s: String) => msgs :+= s; updateListeners()
  case s: String => RemoteSend ! AMQPMessage(s)
}
```

This change means all our comet chat messages go out to RabbitMQ where they are distributed to all the instances of our Lift application, and all the instances receive the messages back as `AMQPMessage` instances and update chat clients as normal.

Discussion

To run more than one instance of your Lift application locally, you'll want to start SBT as normal, and then in another terminal start again but on a different port number:

```
$ sbt
...
> set port in container.Configuration := 9090
[info] Reapplying settings...
[info] Set current project to RabbitMQ Chat (in build file:rabbitmq_chat/)
> container:start
```

You can then visit one application at *http://127.0.0.1:8080* and another at *http://127.0.0.1:9090*.

In the example code, you can see that `AMQPSender[T]` and `AMQPDispatcher[T]` take care of most of the work for us, and we provide some configuration. In the case of `RemoteSend` we're configuring the `AMQPSender` to work with `String` messages and to work with an *exchange* called "lift.chat". In RabbitMQ the exchange is the entity we send messages to, and the exchange has the responsibility for passing on the message. In this case the exchange is a *fanout* (a simple kind of topic) where each subscriber receives a copy of any messages sent to the exchange. This is clearly what we want to get our chat messages sent to all connected Lift instances of the chat application.

The `RemoteReceiver` is also configured to receive `String` messages, although the configuration is a little longer. Here, as well as indicating the exchange to be used, we declare a *temporary queue* for our Lift instance. The queue is the place where RabbitMQ sends messages, and what we're saying here is that each receiver has its own queue. The fanout exchange will ensure any message sent to the exchange is placed into every queue. The queue has a random name assigned by RabbitMQ and is destroyed we disconnect from it.

The final part of the `RemoteReceiver` is to specify how we consume messages. The default behaviour of `RemoteSend` is to serialise objects, so we mirror that in the receiver by using the `SerializedConsumer` class provided by the AMQP module.

To see the behaviour of RabbitMQ, it's useful to install the management web console. From the directory where you installed RabbitMQ:

```
$ ./sbin/rabbitmq-plugins enable rabbitmq_management
```

Visit the administrative web interface at *http://127.0.0.1:15672/* and login. The default username and password is "guest".

Having to have RabbitMQ (or other type of pubsub solution) running during development may be inconvenient. In that case, you can simply not initialize the service in *Boot.scala*:

```
if (Props.productionMode)
  RemoteReceiver ! AMQPAddListener(ChatServer)
```

And in the chat server, only send to local clients:

```
override def lowPriority = {
  case AMQPMessage(s: String) => msgs :+= s; updateListeners()
  case s: String =>
    if (Props.productionMode) RemoteSend ! AMQPMessage(s)
    else { msgs :+= s; updateListeners() }
  }
```

Note that `Props.productionMode` is `true` for the run modes of `Production`, `Staging` and `Pilot`.

See Also

The Lift Chat example is described in *Simply Lift* at *http://simply.liftweb.net/*. The source code used in this recipe can be found at: *https://github.com/LiftCookbook/rabbitmq_chat*.

The Lift AMQP module is: *https://github.com/liftmodules/amqp*.

If you want to learn more about RabbitMQ, take a look at the tutorials (*http://www.rabbitmq.com/tutorials/tutorial-five-java.html*) or Alvaro Videla and Jason J.W. Williams (2012), *RabbitMQ in Action: Distributed Messaging for Everyone*, Manning Publications.

Diego Medina has implemented a distributed comet solution using CouchDB, and has described it in a blog post at *https://fmpwizard.telegr.am/blog/distributed-comet-chat-lift*.

Amazon's Simple Notification Service (SNS) is a fanout facility so can also be used to implement this pattern. You can find a Lift module for SNS at *https://github.com/SpiralArm/liftmodules-aws-sns*.

CHAPTER 11

Contributing, Bug Reports & Getting Help

11.1. You'd Like Some Help

Problem

You're stuck on something in Lift and you'd like some help.

Solution

Ask a question on the Lift mailing list at *https://groups.google.com/group/liftweb*.

Discussion

You will find some information about Lift on StackOverflow, Quora and elsewhere, but the mailing list is *the* place to go to get help and support. You can search the archive to see if your question has already been addressed, but questions are very welcome as it helps the Lift community understand what users need and what might be causing problems or need explanation. Much of what you're reading here originates from questions that have been asked on the mailing list.

New members to the mailing list are moderated to help reduce spam. This means the first time you post your message may take a few hours to show up.

See Also

The Lift community is one of the things that makes Lift what it is. Please take a look at *http://liftweb.net/community* before posting: you'll get the best response with a polite question.

If you need paid consulting, development or SLA-backed support, there's a list of organizations on the Wiki at *https://www.assembla.com/spaces/liftweb/wiki/Commercial_Support*.

11.2. How to Report Bugs

Problem

You've found a bug and want to report it.

Solution

Discuss your findings on the Lift mailing list, saying what you're seeing, and what you'd expect to see.

By all means look at the existing tickets to see if your issue is there or recently fixed, but please do not raise a ticket unless asked to do so by a Lift committer on the mailing list.

Discussion

> If Lift is not behaving as you expect, please ask questions about what you're seeing. The ideal form of these questions is "When I do X, my Lift app does Y, but I expect it to do Z, why?" This provides a set of language to discuss your application and the way that Lift responds to requests. Perhaps there's a way of improving Lift. Perhaps there's a concept that's different in Lift than you might be used to. Perhaps there's a documentation issue that can help bridge the gap between what Lift is doing and what you expect it to do. Most importantly, just because Lift is behaving differently than you expect it to, it's not necessarily a bug in Lift.
>
> — David Pollack
> *http://lift.la/expected-behavior-in-the-lift-community*

A great way to get help with the issue you have, or getting a bug fixed, is to produce a small example to illustrate the problem and posting it on GitHub. The key is to provide instructions so someone can run the example and see exactly what you see without having to jump though hoops.

See Also

You'll find a list of tickets at *http://ticket.liftweb.net/*.

If you're asked to, or want to, post example code there's a guide at *http://www.assembla.com/wiki/show/liftweb/Posting_example_code*.

If you've found a bug, and you're asked to create a ticket, the main thing to do is to include a link to the mailing list discussion of the issue, as described at *http://www.assembla.com/wiki/show/liftweb/Creating_tickets*.

11.3. Contributing Small Code Changes and ScalaDoc

Problem

You have a small change or ScalaDoc improvement you'd like incorporated into Lift.

Solution

You can issue pull requests against the Lift source providing your change meets one of the following requirements:

- It's change to a comment.
- It's example code.
- It's a *small* change, enhancements, or bug fix to Lift.

Your pull request must include an edit to add your signature to the bottom of the *contributors.md* file.

Discussion

Historically Lift had a strict contributor policy of simply not accepting any code contributions except from committers who had signed an agreement to assign copyright to the Lift project. This allowed corporations wanting to adopt Lift to do so without litigation concerns.

The safety is still there, now via the requirement for a signature on the contributors file, which reads:

> By submitting this pull request which includes my name and email address (the email address may be in a non-robot readable format), I agree that the entirety of the contribution is my own original work, that there are no prior claims on this work including, but not limited to, any agreements I may have with my employer or other contracts, and that I license this work under an Apache 2.0 license.

What's a small change? That's a good question, and if you're unsure, talk about your proposed change on the mailing list.

See Also

This contribution policy was introduced in November 2012, and can be read about at *http://www.lift.la/blog/new_contribution_policy*.

The *contributors.md* file is found at: *https://github.com/lift/framework/blob/master/contributors.md*.

The Lift source is on GitHub at *https://github.com/lift/*. The *framework* project is probably the one you want, although you'll also find Git repositories for examples and Lift web sites there.

GitHub provide an introduction to pull requests at *https://help.github.com/articles/using-pull-requests*.

Recipe 11.6 describes how to share code of any size via Lift modules.

11.4. Contributing Documentation

Problem

You'd like to contribute documentation to Lift.

Solution

Update or add to the Lift wiki at *https://www.assembla.com/wiki/show/liftweb*.

You'll need to sign in, or create a free account, with Assembla, the company that hosts the wiki. You then need to become a *watcher* of the Lift wiki, which is offered as a link on the top right of the Lift wiki page. As a watcher you can edit pages and create new pages.

Discussion

If you're unsure about a change you'd like to make, just ask for feedback on the Lift mailing list.

One limitation of the watcher role on Assembla is that you cannot move pages. If you create a new page in the wrong section, or want to reorganise pages, you'll need to ask on the Lift mailing list for someone with permissions to do that for you.

See Also

The markup format for the Wiki pages is Textile. A reference guide can be found at: *http://redcloth.org/hobix.com/textile/*.

11.5. How to Add a New Recipe to this Cookbook

Problem

You'd like to add a section or chapter to this cookbook.

Solution

If you're comfortable using Git, you can fork the repository and send a pull request.

Alternatively, download a template file, write your recipe, and email to the Lift mailing list at *https://groups.google.com/group/liftweb*.

You can find the template file at: *https://raw.github.com/d6y/lift-cookbook/master/template.asciidoc*.

Discussion

Anything you've puzzled over, or things which have surprised you, impressed you or are non-obvious are great topics for recipes. Improvements, discussions and clarifications of existing recipes are welcome too.

The cookbook is structured using a markup language called Asciidoc. If you're familiar with Markdown or Textile, you'll find similarities. For the cookbook you only need to know about section headings, source code formatting and links. Examples of all of these are in the `template.asciidoc` file.

To find out where to make a change you need to know that each chapter is a separate file, and each recipe is a section in that file.

Licensing

We ask contributors the following:

- You agree to license your work (including the words you write, the code you use and any images) to us under the Creative Commons Attribution, Non Commercial, No Derivatives license.
- You assert that the work is your own, or you have the necessary permission for the work.

To keep things simple, all author royalties from this book are given to charity.

See Also

The source to this book is at: *https://github.com/d6y/lift-cookbook/*.

The AsciiDoc cheetsheet at *http://powerman.name/doc/asciidoc* is a quick way to get into AsciiDoc, but if you need more, the AsciiDoc home page has the details: *http://www.methods.co.nz/asciidoc/*.

GitHub provide an introduction to pull requests at *https://help.github.com/articles/using-pull-requests*.

Recipe 11.4 describes other ways to contribution documentation to Lift.

11.6. Sharing Code in Modules

Problem

You have code you'd like to share between Lift projects or with the community.

Solution

Create a Lift module, and then reference the module from your Lift projects.

As an example, let's create a module to embed the *Snowstorm snow fall effect* on every page in your Lift web application (please don't do this).

There's nothing special about modules: they are code, packaged and used like any other dependency. What makes them possible is the exposure of extension points via `LiftRules`. The main convention is to have an `init` method that Lift applications can use to initialize your module.

For our snow storm we're going to package some JavaScript and inject the script onto every page.

Starting with the *lift_blank* template downloaded from *liftweb.net*, we can remove all the source and HTML files as this won't be a runnable Lift application in itself. However, it will leave us with the regular Lift structure and build configuration.

Our module will need the Snowstorm JavaScript file from *https://github.com/scottschiller/snowstorm/* copied as *resources/toserve/snowstorm.js*. This will place the JavaScript file on the classpath of our Lift application.

The final piece of the module is to ensure the JavaScript is included on every page:

```
package net.liftmodules.snowstorm

import net.liftweb.http._

object Snowstorm {

 def init() : Unit = {

  ResourceServer.allow {
```

```
      case "snowstorm.js" :: Nil => true
    }

    def addSnow(s: LiftSession, r: Req) = S.putInHead(
      <script type="text/javascript" src="/classpath/snowstorm.js"></script> )

    LiftSession.onBeginServicing = addSnow _ :: LiftSession.onBeginServicing

  }

}
```

Here we are plugging into Lift's processing pipeline and adding the required JavaScript to the head of every page.

We modify *build.sbt* to give the module a name, organisation and version number. We also can remove many of the dependencies and the web plugin as we only depend on the web API elements of Lift:

```
name := "snowstorm"

version := "1.0.0"

organization := "net.liftmodules"

scalaVersion := "2.9.1"

resolvers ++= Seq(
   "snapshots" at "http://oss.sonatype.org/content/repositories/snapshots",
   "releases" at "http://oss.sonatype.org/content/repositories/releases"
)

scalacOptions ++= Seq("-deprecation", "-unchecked")

libraryDependencies ++= {
  val liftVersion = "2.5-RC5"
  Seq(
    "net.liftweb" %% "lift-webkit"  % liftVersion  % "compile"
  )
}
```

We can publish this plugin to the repository on disk by starting SBT and typing:

```
publish-local
```

With our module built and published, we can now include it in our Lift applications. To do that, modify the Lift application's *build.sbt* to reference this new "snowstorm" dependency:

```
libraryDependencies ++= {
  val liftVersion = "2.5-RC5"
  Seq(
  ...
```

```
"net.liftmodules" %% "snowstorm" % "1.0.0",
...
```

In our Lift application's *Boot.scala* we finally initialise the plugin:

```
import net.liftmodules.snowstorm.Snowstorm
Snowstorm.init()
```

When we run our Lift application, white snow will be falling on every page, supplied by the module.

Discussion

The module is self contained: there's no need for users to copy JavaScript files around or modify their templates. To achieve that we've made use of `ResourceServer`. When we reference the JavaScript file via */classpath/snowstorm.js*, Lift will attempt to locate *snowstorm.js* from the classpath. This is what we want for our Lift application because *snowstorm.js* will be inside the module JAR file.

However, we do not want to expose all files on the classpath to anyone visiting our application. To avoid that, Lift looks for resources inside a *toserve* folder, which for our purposes means files and folders inside *src/main/resources/toserve*. You can think of */classpath* meaning *toserve* (although, you can change those values via `LiftRules.resourceServerPath` and `ResourceServer.baseResourceLocation`).

As a further precaution you need to explicitly allow access to these resources. That's done with:

```
ResourceServer.allow {
  case "snowstorm.js" :: Nil => true
}
```

We're just always returning `true` for anyone who asks for this resource, but we could dynamically control access here if we wanted.

`S.putInHead` adds the JavaScript to the head of a page, and is triggered on every page by `LiftSession.onBeginServicing` (also discussed in Recipe 6.2). We could make use of `Req` here to restrict the snow storm to particular pages, but we're adding it to every page.

Hopefully you can see that anything you can do in a Lift application you can probably turn into a Lift module. A typical approach might be to have functionality in a Lift application, and then factor out settings in `Boot` into a module `init` method. For example, if you wanted to provide REST services as a module, that would be possible and is an approach taken by the Lift PayPal module.

Making your Module Available

If you do want your module to be used by a wider audience, you need to publish it to a public repository, such as Sonatype or CloudBees. You'll also want to keep your module up-to-date with Lift releases. There are a few conventions around this.

One convention is to include the Lift "edition" as part of the module name. For example, version 1.0.0 of your "foo" module for Lift 2.5 would have the name "foo_2.5". This makes it clear your module is compatible with Lift for all of 2.5, including milestones, release candidates, snapshots and final releases. It also means you only need publish your module once, at least until Lift 2.6 or 3.0 is available.

One way to ease the above is to make a modification to your module build to allow the Lift version number to change. This makes it possible to automate the build when new versions of Lift are released. To do that, create *project/LiftModule.scala* in your module:

```
import sbt._
import sbt.Keys._

object LiftModuleBuild extends Build {

  val liftVersion = SettingKey[String]("liftVersion",
    "Full version number of the Lift Web Framework")

  val liftEdition = SettingKey[String]("liftEdition",
    "Lift Edition (short version number to append to artifact name)")

  val project = Project("LiftModule", file("."))
}
```

This defines a setting to control the Lift version number. You use it in your module *build.sbt* like this:

```
name := "snowstorm"

organization := "net.liftmodules"

version := "1.0.0-SNAPSHOT"

liftVersion <<= liftVersion ?? "2.5-SNAPSHOT"

liftEdition <<= liftVersion apply { _.substring(0,3) }

name <<= (name, liftEdition) { (n, e) =>  n + "_" + e }

...

libraryDependencies <++= liftVersion { v =>
  "net.liftweb" %% "lift-webkit" % v % "provided" ::
  Nil
}
```

Note the "provided" configuration for Lift in the build file. This means that when your module is used the version of Lift's webkit used be the version provided by the application being built by the person using your module.

What the above gives you is a way for your module to depend on Lift ("2.5"), without locking your module into a specific release it happens to be built with "2.5-SNAPSHOT". By using a setting of `liftVersion`, we can control the version via a script for all modules. This is what we do to publish a range of Lift Modules after each Lift release, as described at *https://www.assembla.com/spaces/liftweb/wiki/Releasing_the_modules*.

When your module is built, don't forget to announce it on the Lift mailing list.

Debugging Your Module

When working on a module and testing it in a Lift application, it would be a chore to have to publish your module each time you changed it. Fortunately, SBT allows your Lift application to depend on the source of a module. To use this, change your Lift project to remove the dependency on the published module and instead add a local dependency by creating *project/LocalModuleDev.scala*:

```
import sbt._
object LocalModuleDev extends Build {
  lazy val root = Project("", file(".")) dependsOn(snow)
  lazy val snow = ProjectRef(uri("../snowstorm"), "LiftModule")
}
```

We are assuming that the snowstorm source can be found at *../snowstorm* relative to the Lift application we are using. With this in place, when you build your Lift project, SBT will automatically compile and depend on changes in the local `snowstorm` module.

See Also

Originally Lift included a set of modules, but these have been separated out to individual projects at *https://github.com/liftmodules/*. The Lift contributor policy outlined in Recipe 11.3 doesn't apply to Lift modules: you're free to contribute to these modules as you would any other open source project.

The Lift wiki pages for modules can be reached via *http://liftmodules.net*.

The Snowstorm project ("setting CPUs on fire worldwide every winter since 2003") is at: *https://github.com/scottschiller/snowstorm/* and the module developed for this recipe is at *https://github.com/LiftCookbook/snowstorm-example-module*.

To publish to Sonatype, take a look at their guide: *https://docs.sonatype.org/display/Repository/Sonatype+OSS+Maven+Repository+Usage+Guide*. CloudBees offer an open source repository, which you can learn about via *http://www.cloudbees.com/foss/foss-dev.cb*.

There are other ways to structure common code between Lift applications. For an example that uses SBT modules and Git, see the mailing list discussion on "Modularize Lift Applications" via *http://bit.ly/lift-moduleapp*.

About the Author

Richard Dallaway is a partner at Underscore Consulting, working on client software projects. He's a Lift committer, where he has focused on the module system. Prior to Underscore, Richard gained his PhD from the School of Cognitive Science at the University of Sussex, worked on machine learning projects, before moving to consultancy and software engineering. He lives with his wife and one small terrier in Brighton, UK.

Colophon

The animal on the cover of *FILL IN TITLE* is FILL IN DESCRIPTION.

The cover image is from FILL IN CREDITS. The cover font is Adobe ITC Garamond. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.