Simply Lift

David Pollak

September 8, 2011

Contents

List of Figures

List of Listings

Part I

The Lift Web Framework

Chapter 1

Introduction

The Lift Web Framework provides web application developers tools to make writing security, interacting, scalable web applications easier than with any other web framework. After reading Part I of this book, you should understand Lift's core concepts and be able to write Lift applications. But with anything, practice is important. I have been writing Lift and Scala for 4 years, and even I learn new things about the language and the framework on a weekly basis. Please consider Lift an path and an exploration, rather than an end point.

"Yo, David, stop yer yappin'. I'm coming from Rails|Spring|Struts|Django and I want to get started super fast with Lift." See From MVC (13 on page 111).

Lift is built on top of the Scala programming language. Scala runs on the Java Virtual Machine. Lift applications are typically packaged as WAR files and run as a J/EE Servlets or Servlet Filters. This book will provide you with the core concepts you need to successfully write Lift web applications. The book assumes knowledge of Servlets and Servlet containers, the Scala Language (Chapters 1-6 of *Beginning Scala* gives you a good grounding in the language), build tools, program editors, web development including HTML and JavaScript, etc. Further, this book will not explore persistence. Lift has additional modules for persisting to relational and non-relational data stores. Lift doesn't distinguish as to how an object is materialized into the address space... Lift can treat any object any old way you want. There are many resources (including Exploring Lift) that cover ways to persist data from a JVM.

Lift is different from most web frameworks and it's likely that Lift's differences will present a challenge and a friction if you are familiar with the MVC school of web frameworks[1]. But Lift is different and Lift's differences give you more power to create interactive applications. Lift's differences lead to more concise web applications. Lift's differences result in more secure and scalable applications. Lift's differences let you be more productive and make maintaining applications easier for the future you or whoever is writing your applications. Please relax and work to understand Lift's differences... and see how you can make best use of Lift's features to build your web applications.

Lift creates abstractions that allow easier expression of business logic and then maps those abstractions to HTTP and HTML. This approach differs from traditional web frameworks which build abstractions on top of HTTP and HTML and require the developer to bridge between common business logic patterns and the underlying protocol. The difference means that you spend more time thinking about your application and less time thinking about the plumbing.

[1]This includes Ruby on Rails, Struts, Java Server Faces, Django, TurboGears, etc.

I am a "concept learner." I learn concepts and then apply them over and over again as situations come up. This book focuses a lot on the concepts. If you're a concept learner and like my stream on conciousness style, this book will likely suit you well. On the other hand, it may not.

Up to date versions of this book are available in PDF form at `http://simply.liftweb.net/Simply_Lift.pdf`. The source code for this book is available at https://github.com/dpp/simply_lift.

If you've got questions, feedback, or improvements to this document, please join the conversation on the Lift Google Group.

I'm a "roll up your sleaves and get your hands dirty with code" kinda guy... so let's build a simple Chat application in Lift. This application will allow us to demonstrate some of Lift's core features as well as giving a "smack in the face" demonstration of how Lift is different.

Chapter 2

The ubiquitous Chat app

Writing a multi-user chat application in Lift is super-simple and illustrates many of Lift's core concepts.

The Source Code can be found at https://github.com/dpp/simply_lift/tree/master/chat.

2.1 The View

When writing a Lift app, it's often best to start off with the user interface... build what the user will see and then add behavior to the HTML page. So, let's look at the Lift template that will make up our chat application.

Listing 2.1: index.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head><title>Home</title></head>
  <body class="lift:content_id=main">
    <div id="main" class="lift:surround?with=default;at=content">
      <!-- the behavior of the div -->
      <div class="lift:comet?type=Chat">
        Some chat messages
        <ul>
          <li>A message</li>
          <li class="clearable">Another message</li>
          <li class="clearable">A third message</li>
        </ul>
      </div>

      <div>
        <form class="lift:form.ajax">
          <input class="lift:ChatIn" id="chat_in"/>
          <input type="submit" value="Say Something"/>
        </form>
      </div>
    </div>
  </body>
</html>
```

It's a valid HTML page, but there are some hinky looking class attributes. The first one is `<body class="lift:content_id=main">`. The class in this case says "the actual page content is contained by the element with id='main'." This allows you to have valid HTML pages for each of your templates, but dynamically add "chrome" around the content based on one or more chrome templates.

Let's look at the `<div id="main">`. It's got a funky class as well: `lift:surround?with=default;at=content`. This class invokes a snippet which surrounds the `<div>` with the default template and inserts the `<div>` and its children at the element with id "content" in the default template. Or, it wraps the default chrome around the `<div>`. For more on snippets, see 7.1 on page 78.

Next, we define how we associate dynamic behavior with the list of chat elements: `<div class="lift:comet?type=Chat">`. The "comet" snippet looks for a class named `Chat` that extends `CometActor` and enables the mechanics of pushing content from the `CometActor` to the browser when the state of the `CometActor` changes.

2.2 The Chat Comet component

The Actor Model provides state in functional languages include Erlang. Lift has an Actor library and LiftActors (see 7.14) provides a powerful state and concurrency model. This may all seem abstract, so let's look at the `Chat` class.

Listing 2.2: Chat.scala

```scala
package code
package comet

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

/**
 * The screen real estate on the browser will be represented
 * by this component. When the component changes on the server
 * the changes are automatically reflected in the browser.
 */
class Chat extends CometActor with CometListener {
  private var msgs: Vector[String] = Vector() // private state

  /**
   * When the component is instantiated, register as
   * a listener with the ChatServer
   */
  def registerWith = ChatServer

  /**
```

```
   * The CometActor is an Actor, so it processes messages.
   * In this case, we're listening for Vector[String],
   * and when we get one, update our private state
   * and reRender() the component. reRender() will
   * cause changes to be sent to the browser.
   */
  override def lowPriority = {
    case v: Vector[String] => msgs = v; reRender()
  }

  /**
   * Put the messages in the li elements and clear
   * any elements that have the clearable class.
   */
  def render = "li *" #> msgs & ClearClearable
}
```

The `Chat` component has private state, registers with the `ChatServer`, handles incoming messages and can render itself. Let's look at each of those pieces.

The private state, like any private state in prototypical object oriented code, is the state that defines the object's behavior.

registerWith is a method that defines what component to register the Chat component with. Registration is a part of the Listener (or Observer) pattern. We'll look at the definition of the `ChatServer` in a minute.

The `lowPriority` method defines how to process incoming messages. In this case, we're Pattern Matching (see Section 7.15) the incoming message and if it's a `Vector[String]`, then we perform the action of setting our local state to the `Vector` and re-rendering the component. The re-rendering will force the changes out to any browser that is displaying the component.

We define how to `render` the component by defining the CSS to match and the replacement (See Section 7.10). We match all the `<li>` tags of the template and for each message, create an `<li>` tag with the child nodes set to the message. Additionally, we clear all the elements that have the `clearable` in the `class` attribute.

That's it for the Chat CometActor component.

2.3 The ChatServer

The ChatServer code is:

Listing 2.3: ChatServer.scala

```
package code
package comet

import net.liftweb._
import http._
import actor._

/**
```

```
 * A singleton that provides chat features to all clients.
 * It's an Actor so it's thread-safe because only one
 * message will be processed at once.
 */
object ChatServer extends LiftActor with ListenerManager {
  private var msgs = Vector("Welcome") // private state

  /**
   * When we update the listeners, what message do we send?
   * We send the msgs, which is an immutable data structure,
   * so it can be shared with lots of threads without any
   * danger or locking.
   */
  def createUpdate = msgs

  /**
   * process messages that are sent to the Actor. In
   * this case, we're looking for Strings that are sent
   * to the ChatServer. We append them to our Vector of
   * messages, and then update all the listeners.
   */
  override def lowPriority = {
    case s: String => msgs :+= s; updateListeners()
  }
}
```

The `ChatServer` is defined as an `object` rather than a `class`. This makes it a singleton which can be referenced by the name `ChatServer` anywhere in the application. Scala's singletons differ from Java's `static` in that the singleton is an instance of an object and that instance can be passed around like any other instance. This is why we can return the `ChatServer` instance from the `registerWith` method in that `Chat` component.

The ChatServer has private state, a `Vector[String]` representing the list of chat messages. Note that Scala's type inferencer infers the type of `msgs` so you do not have to explicitly define it.

The `createUpdate` method generates an update to send to listeners. This update is sent when a listener registers with the `ChatServer` or when the `updateListeners()` method is invoked.

Finally, the `lowPriority` method defines the messages that this component can handle. If the `ChatServer` receives a `String` as a message, it appends the `String` to the `Vector` of messages and updates listeners.

2.4 User Input

Let's go back to the view and see how the behavior is defined for adding lines to the chat.

`<form class="lift:form.ajax">` defines an input form and the `form.ajax` snippet turns a form into an Ajax (see Section 7.12) form that will be submitted back to the server without causing a full page load.

Next, we define the input form element: `<input class="lift:ChatIn" id="chat_in"/>`. It's a plain old input form, but we've told Lift to modify the `<input>`'s behavior by calling the `ChatIn` snippet.

2.5 Chat In

The `ChatIn` snippet (See Section 7.1) is defined as:

Listing 2.4: ChatIn.scala

```scala
package code
package snippet

import net.liftweb._
import http._
import js._
import JsCmds._
import JE._

import comet.ChatServer

/**
 * A snippet transforms input to output... it transforms
 * templates to dynamic content. Lift's templates can invoke
 * snippets and the snippets are resolved in many different
 * ways including "by convention". The snippet package
 * has named snippets and those snippets can be classes
 * that are instantiated when invoked or they can be
 * objects, singletons. Singletons are useful if there's
 * no explicit state managed in the snippet.
 */
object ChatIn {

  /**
   * The render method in this case returns a function
   * that transforms NodeSeq => NodeSeq. In this case,
   * the function transforms a form input element by attaching
   * behavior to the input. The behavior is to send a message
   * to the ChatServer and then returns JavaScript which
   * clears the input.
   */
  def render = SHtml.onSubmit(s => {
    ChatServer ! s
    SetValById("chat_in", "")
  })
}
```

The code is very simple. The snippet is defined as a method that associates a function with form element submission, `onSubmit`. When the element is submitted, be that normal form submission, Ajax, or whatever, the function is applied to the value of the form. In English, when the user submits the form, the function is called with the user's input.

The function sends the input as a message to the `ChatServer` and returns JavaScript that sets the value of the input box to a blank string.

2.6 Running it

Running the application is easy. Make sure you've got Java 1.6 or better installed on your machine. Change directories into the `chat` directory and type `sbt update ~jetty-run`. The Simple Build Tool will download all necessary dependencies, compile the program and run it.

You can point a couple of browsers to `http://localhost:8080` and start chatting.

Oh, and for fun, try entering `<script>alert('I ownz your browser');<script>` and see what happens. You'll note it's what you want to happen.

2.7 What you don't see

Excluding imports and comments, there are about 20 lines of Scala code to implement a multi-threaded, multi-user chat application. That's not a lot.

The first thing that's missing is synchronization or other explicit forms of thread locking. The application takes advantage of Actors and immutable data structures, thus the developer can focus on the business logic rather than the threading and locking primatives.

The next thing that's missing is routing and controllers and other stuff that you might have to do to wire up Ajax calls and polling for server-side changes (long or otherwise). In our application, we associated behavior with display and Lift took care of the rest (see Section 7.17).

We didn't do anything to explicitly to avoid cross-site scripting in our application. Because Lift takes advantage of Scala's strong typing and type safety (see Section 7.16), Lift knows the difference between a String that must be HTML encoded and an HTML element that's already properly encoded. By default, Lift applications are resistant to many of the OWASP top 10 security vulnerabilities (see Section 7.18).

This example shows many of Lift's strengths. Let's expand the application and see how Lift's strengths continue to support the development of the application.

Chapter 3

Snippets and SiteMap

Lift services HTTP request in three ways: generating HTML pages, low level HTTP responses (e.g., REST), and responding to Ajax/Comet requests. Lift treats each type of request differently to make the semantics for responding to each type of request most natural. Put another way, it's different to build a complex HTML page with lots of different components than to send back some JSON data that corresponds to a database record.

In this chapter, we're going to explore how Lift does dynamic HTML page generation based on the incoming HTTP request and URL including putting "chrome" around the HTML page (menus, etc.), placing dynamic content on each page, and site navigation including access control.

The code for this chapter can be found in the `samples/snippet_and_sitemap` directory of the *Simply Lift* distribution.

3.1 Starting at the beginning: `Boot.scala`

When your Lift application first starts up, it executes the code in Boot.scala:

Listing 3.1: Boot.scala

```
package bootstrap.liftweb

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

import common._
import http._
import sitemap._
import Loc._

import code.snippet._

/**
 * A class that's instantiated early and run. It allows the application
 * to modify lift's environment
 */
```

```
class Boot {
  /**
   * Calculate if the page should be displayed.
   * In this case, it will be visible every other minute
   */
  def displaySometimes_? : Boolean =
    (millis / 1000L / 60L) % 2 == 0

  def boot {
    // where to search snippet
    LiftRules.addToPackages("code")

    // Build SiteMap
    def sitemap(): SiteMap = SiteMap(
      Menu.i("Home") / "index", // the simple way to declare a menu

      Menu.i("Sometimes") / "sometimes" >> If(displaySometimes_? _,
                                  S ? "Can't view now"),

      // A menu with submenus
      Menu.i("Info") / "info" submenus(
        Menu.i("About") / "about" >> Hidden >> LocGroup("bottom"),
        Menu.i("Contact") / "contact",
        Menu.i("Feedback") / "feedback" >> LocGroup("bottom")
      ),


      Menu.i("Sitemap") / "sitemap" >> Hidden >> LocGroup("bottom"),

      Menu.i("Dynamic") / "dynamic", // a page with dynamic content

      Param.menu,

      Menu.param[Which]("Recurse", "Recurse",
                   {case "one" => Full(First())
                    case "two" => Full(Second())
                    case "both" => Full(Both())
                    case _ => Empty},
                   w => w.toString) / "recurse",

      // more complex because this menu allows anything in the
      // /static path to be visible
      Menu.i("Static") / "static" / **)

    // set the sitemap. Note if you don't want access control for
    // each page, just comment this line out.
    LiftRules.setSiteMapFunc(() => sitemap())

    //Show the spinny image when an Ajax call starts
    LiftRules.ajaxStart =
      Full(() => LiftRules.jsArtifacts.show("ajax-loader").cmd)

    // Make the spinny image go away when it ends
    LiftRules.ajaxEnd =
      Full(() => LiftRules.jsArtifacts.hide("ajax-loader").cmd)

    // Force the request to be UTF-8
    LiftRules.early.append(_.setCharacterEncoding("UTF-8"))

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

Rather than keeping configuration parameters in XML files, Lift keeps configuration parameters in code in `Boot`. Boot is executed once when the servlet container loads the Lift application. You can change many of Lift's execution rules in the `LiftRules` singleton during boot, but after boot, these parameters are frozen.

3.1.1 **`LiftRules`** rules

Most of the configuration parameters that define how Lift will convert an HTTP request into a response are contained in the `LiftRules` singleton. Some of the parameters for `LiftRules` are used commonly and some are very infrequently changed from their default. LiftRules can be changed during boot, but not at other times. So, set all your configuration in boot (or in methods that are called from boot).

3.1.2 Properties and Run modes

While many properties for your running application can be defined in `Boot.scala`, there are some properties that are best defined in a text file. Lift supports multiple properties files per project. The properties files are loaded based on the user, machine and run mode.

If you want to provide a configuration file for a subset of your application or for a specific environment, Lift expects configuration files to be named in a manner relating to the context in which they are being used. The standard name format is:

`modeName.userName.hostName.props`

examples:
`dpp.yak.props`
`test.dpp.yak.props`
`production.moose.props`
`staging.dpp.props`
`test.default.props`
`default.props`

with hostName and userName being optional, and modeName being one of "test", "staging", "production", "pilot", "profile", or blank (for development mode). The standard Lift properties file extension is "props".

Place properties files in the `src/main/resources/props` directory in your project and they will be packaged up as part of the build process.

When you're developing your Lift application, the run mode (see `net.liftweb.util.Props.mode`) will be `Development`. When you deploy your application, pass `-Drun.mode=production` to your web container. In production mode, Lift aggressively caches templates, snippet classes, etc.

3.1.3 By convention

Lift, like Rails, will look for items in certain locations by convention. For example, Lift will look for classes that implement snippets in the `xxx.snippet` package where the xxx part is the main package for your application. You define one or more packages for Lift to look in with:

```
  // where to search snippet
  LiftRules.addToPackages("code")
```

Here, we've added the `code` package to the list of packages that Lift will search through. You can also do `LiftRules.addToPackages("com.fruitbat.mydivision.myapplication")`.

3.1.4 Misc Rules

We'll skip the sitemap definition until the next section. This rule defines how to show a spinning icon during Ajax calls (Lift will automatically show the spinning icon if this function is enabled):

```
  //Show the spinny image when an Ajax call starts
  LiftRules.ajaxStart =
    Full(() => LiftRules.jsArtifacts.show("ajax-loader").cmd)
```

And this rule sets the default character encoding to UTF-8 rather than the default platform encoding:

```
  // Force the request to be UTF-8
  LiftRules.early.append(_.setCharacterEncoding("UTF-8"))
```

Okay... you get the idea... there are plenty of parameters to tune during boot.

3.1.5 Html5

Prior to Lift 2.2, Lift treated all templates as XHTML and emitted XHTML to the browser. When the Lift project started in early 2007, this seemed like a Really Good Idea™. Turns out the world has not adopted XHTML and some JavaScript libraries, e.g. Google Maps, doesn't work on XHTML pages. Lift 2.2 introduced optional Html5 support both in the parser (so it could read Html5 templates rather than requiring well formed XML in templates) and emits Html5 to the browser. Lift still processes pages as Scala `NodeSeq` elements, so no changes are required to the application.

In order to keep Lift 2.2 apps backward compatible with Lift's XHTML support, by default the XHTML parser/serializer are used. However, it's recommended to use the Html5 support which can be turned on in boot with:

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

3.2 SiteMap

Lift has an optional feature called SiteMap. You don't have to use it. But if you do set a sitemap in boot, then Lift will use the sitemap as a white list of HTML pages for your site (note that REST URLs do not need to be listed in the sitemap). SiteMap defines navigation and access control, allows you to create hierarchical menus, grouped menu items, display the entire sitemap, a relative sitemap, as well breadcrumbs. This section will discuss some of SiteMap's capabilities.

3.2.1 Defining the `SiteMap`

The SiteMap must be defined in boot and is only defined once[1]. Typically, you will define a function that returns a SiteMap instance:

```
// Build SiteMap
def sitemap(): SiteMap = ...
```

And then define the `SiteMap` in `LiftRules`:

```
// set the sitemap. Note if you don't want access control for
// each page, just comment this line out.
LiftRules.setSiteMapFunc(() => sitemap())
```

In development mode, the function will be called on each page load to rebuilt the SiteMap. In all other Lift run modes, the sitemap will be built once during boot.

A `SiteMap` is a collection of `Menu` instances. Each `Menu` has one `Loc[_]` and a set of `Menu` instances as submenus (zero or more). Each `Menu` instance has a unique name.

If an HTML page is not defined in the sitemap, Lift will not serve it. SiteMap is a white list of pages to serve. Further, the `Loc[_]` has parameters that can include multiple access control rules.

3.2.2 Simplest `SiteMap`

The simplest sitemap defines a single page:

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

[1]In development mode, the sitemap can be changed dynamically to support changes to site content without having to re-start your application each time navigation changes. This is a development-time feature only. There are significant performance penalties associated with rebuilding the sitemap on each page load including forcing the serialization of serving pages. There are plenty of features in SiteMap that allow you to enable/disable menu items and have dynamically generated submenus. Don't rely on Lift's development-mode menu reloading for your application design.

This is a SiteMap with a single menu item. The Menu has the name "Home" and will be displayed as the localized (see 8.1 on page 92) string "Home". The Menu.i method generates a `Menu` with a `Loc[Unit]`.

3.2.3 `Menu` and `Loc[_]`

You may be wondering why a `Menu` and a `Loc[_]` (short for location, pronouned "Loke") are separate and why the Loc takes a type parameter.

A `Menu` contains a location and many submenus. The original thought was that you could have a single `Loc[_]` that might be placed in different places in the menu hierarchy. So, historically, they are separated, but there's a one to one relation between them.

The `Loc[_]` takes a type parameter which defines a current value type for the `Loc`. For example, if the `Loc` refers to a page that will display a wiki page, then the type parameter of the `Loc` would be `WikiPage`: `Loc[WikiPage]`.

Each `Loc` can have many parameters (know as `LocParam`, "loke param") that define behavior for the `Loc[_]`. These parameters include access control testing, template definition, title, group, etc.

3.2.4 Access Control

You can control access to the URL/page represented by the `Loc` with the `If()` LocParam:

```
/**
 * Calculate if the page should be displayed.
 * In this case, it will be visible every other minute
 */
def displaySometimes_? : Boolean =
  (millis / 1000L / 60L) % 2 == 0

   Menu.i("Sometimes") / "sometimes" >> If(displaySometimes_? _,
                                     S ? "Can't view now")
```

We define a method that returns `true` if access is allowed. Adding the `If() LocParam` will restrict access to the page unless the function returns true. Menu items will not be visible for pages that do not pass the access control rules and even if the user types the URL into the browser, the page will not be displayed (by default, the user will be redirected by to the home page and an error will be displayed.)

3.2.5 `Hidden` and `Group`

Menu items can be hidden from the default menu hierarchy even if the page is accessible. The Hidden LocParam says "hide from default menu."

```
Menu.i("About") / "about" >> Hidden >> LocGroup("bottom")
```

Menu items can also be grouped together in a named group and then displayed:

```
<span class="lift:Menu.group?group=bottom"></span>
```

Which results in:

```
<a href="/about">About</a> <a href="/feedback">Feedback</a> <a href="/sitemap">Sitemap</a
```

3.2.6 Submenus

You can nest menus:

```
    // A menu with submenus
    Menu.i("Info") / "info" submenus(
      Menu.i("About") / "about" >> Hidden >> LocGroup("bottom"),
      Menu.i("Contact") / "contact",
      Menu.i("Feedback") / "feedback" >> LocGroup("bottom"))
```

The About, Contact and Feedback pages are nested under the Info page.

3.2.7 Parameters

You can parse the incoming URL and extract parameters from it into type-safe variables:

```
// capture the page parameter information
case class ParamInfo(theParam: String)

 // Create a menu for /param/somedata
 val menu = Menu.param[ParamInfo]("Param", "Param",
                                  s => Full(ParamInfo(s)),
                                  pi => pi.theParam) / "param"
```

The above code creates a menu called "Param". The menu is for the url `/param/xxx` where `xxx` can match anything.

When the URL `/param/dogfood` or `/param/fruitbat` is presented, it matches the `Loc` and the function (`s => Full(ParamInfo(s))`) is invoked. If it returns a `Full Box`, the value is placed in the `Loc`'s `currentValue`.

It's possible to hand-write Loc implementation that will match many URL parameters.

For information on accessing the captured parameters (in this case the `ParamInfo`), see 3.4.5 on page 23.

3.2.8 Wildcards

You can create menus that match all the contents of a given path. In this case, all the html files in `/static/` will be served. That includes `/static/index`, `/static/fruitbat`, and `/static/moose/frog/wombat/meow`.

```
    // more complex because this menu allows anything in the
    // /static path to be visible
    Menu.i("Static") / "static" / **
```

Note that Lift will not serve any files or directories that start with `.` (period) or _ (underscore) or end with `-hidden`.

3.2.9 Summary

We've demonstrated how to create a `SiteMap` with many different kinds of menu items. Next, let's look at the views.

3.3 View First

Once the access control is granted by SiteMap, Lift loads the view related to the URL. There are many mechanisms that Lift uses to resolve a path to a view, but the simplest is a one to one mapping between the URL path and the files in `/src/main/webapp`. If the URL is `/index`, then Lift will look for the localized (see 8.1 on page 92) version of `/src/main/webapp/index.html`. Once Lift loads the template, Lift processes it to transform it into the dynamic content you want to return in response to the URL input.

3.3.1 Page source

Let's look at the page source:

Listing 3.2: index.html

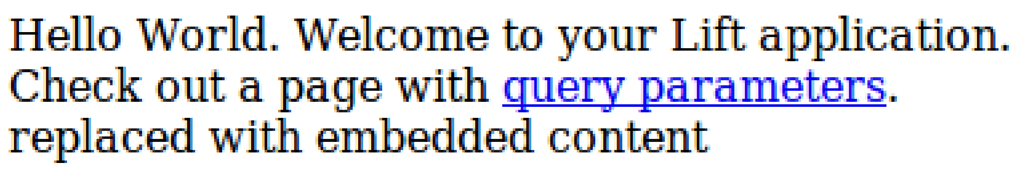
```
<!DOCTYPE html>
<html>
  <head>
    <meta content="text/html; charset=UTF-8" http-equiv="content-type" />
    <title>Home</title>
  </head>
  <body class="lift:content_id=main">
    <div id="main" class="lift:surround?with=default&at=content">
      <div>Hello World. Welcome to your Lift application.</div>
      <div>Check out a page with <a href="/param/foo">query parameters</a>.</div>

      <span class="lift:embed?what=_embedme">
        replaced with embedded content
      </span>

      <div>
BADTAB<ul>
BADTAB  <li>Recursive: <a href="/recurse/one">First snippet</a></li>
BADTAB  <li>Recursive: <a href="/recurse/two">Second snippet</a></li>
BADTAB  <li>Recursive: <a href="/recurse/both">Both snippets</a></li>
BADTAB</ul>
```

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

We can open the page in our browser:


Hello World. Welcome to your Lift application.
Check out a page with query parameters.
replaced with embedded content

3.3.2 Dynamic content

The template is a legal HTML page. But there are marker in the page to tell Lift how to interpret the HTML.

If the `<body>` tag contains a `class` attribute `lift:content_id=xxxx`, then Lift will find the element with the matching `id` and use that as the starting point for rendering the page. This allows your designers to edit and maintain the pages in the same hierarchy that you use for your application.

3.3.3 Surround and page chrome

The template processing starts with:

```
 <div id="main" class="lift:surround?with=default&at=content">
```

The `class` attribute `lift:surround?with=default;at=content` instructs Lift to surround the current Element with the template named default.html (typically located in the `/templates-hidden/` directory), and place the current page's content at the element with the "content" `id`.

This pattern allows us to wrap a common chrome around every page on our site. You can also specify different template to use for surrounding. Further, the template itself can choose different templates to use for surrounding.

3.3.4 Embed

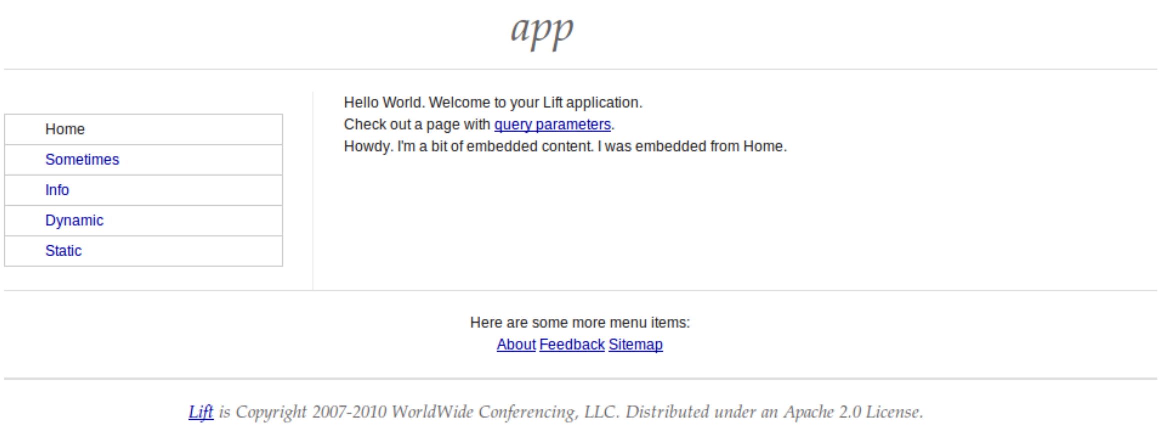
In addition to surrounding the page with chrome, you can also embed another file. For example, you could have a shopping cart component that is embedded in certain pages. We embed with:

```
<span class="lift:embed?what=_embedme">
  replaced with embedded content
</span>
```

Once again, the command is signalled with a `class` attribute that starts with `lift:`. In this case, we embed a template from the file `_embedme.html`.

3.3.5 Results

The resulting dynamically generated page looks like:


app

Hello World. Welcome to your Lift application.
Check out a page with query parameters.
Howdy. I'm a bit of embedded content. I was embedded from Home.

Here are some more menu items:
About Feedback Sitemap

Lift is Copyright 2007-2010 WorldWide Conferencing, LLC. Distributed under an Apache 2.0 License.

3.4 Snippets and Dynamic content

Lift templates contain no executable code. They are pure, raw, valid HTML.

Lift uses snippets to transform sections of the HTML page from static to dynamic. The key word is transform.

Lift's snippets are Scala functions: `NodeSeq => NodeSeq`. A `NodeSeq` is a collection of XML nodes. An snippet can only transform input `NodeSeq` to output `NodeSeq`. Well, not exactly... a snippet may also have side effects including setting cookies, doing database transactions, etc. But the core transformation concept is important. First, it isolates snippet functionality to discrete parts of the page. This means that each snippet, each `NodeSeq => NodeSeq`, is a component. Second, it means that pages are recursively built, but remain as valid HTML at all times. This means that the developer has to work hard to introduce a cross site scripting vulnerability. Third, the designers don't have to worry about learning to program anything in order to design HTML pages because the program execution is abstracted away from the HTML rather than embedded in the HTML.

3.4.1 Snippets in markup

In order to indicate that content is dynamic, the markup contains a snippet invocation. That typically takes the form `class="someclass someothercss lift:mysnippet"`. If a class attribute contains `lift:xxx`, the `xxx` will be resolved to a snippet. The snippet may take attributes. Attributes are encoded like URL parameters... offset by a `?` (question mark), then `name=value`, separted by `?` (question mark), `;` (semicolon) or `&` (ampersand). `name` and `value` are URL encoded.

You may also invoke snippets with XML tags:

```
<lift:my_snippet cat="foo">
  <div>xxxx</div>
</lift:my_snippet>
```

Note that the Html5 parser will force all tags to lower case so `<lift:MySnipet>` will become `<lift:mysnippet>`.

Lift 2.3 will also allow snippet invocation in the form `<div l="mysnippet?param=value">xxx</div>`.

The latter two mechanisms for invoking snippets will not result in valid Html5 templates.

3.4.2 Snippet resolution

Lift has a very complex set of rules to resolve from snippet name to NodeSeq => NodeSeq (see 23.1 on page 141). For now, the simplest mechanism is to have a `class` or `object` in the `snippet` package that matches the snippet name.

So `lift:HelloWorld` will look for the `code.snippet.HelloWorld` class and invoke the `render` method.

`lift:CatFood.fruitbat` will look for the `code.snippet.CatFood` class and invoke the `fruitbat` method.

3.4.3 Dynamic Example

Let's look at the `dynamic.html` page:

Listing 3.3: dynamic.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta content="text/html; charset=UTF-8" http-equiv="content-type" />
    <title>Dynamic</title>
  </head>
  <body class="lift:content_id=main">
    <div id="main" class="lift:surround?with=default;at=content">
      This page has dynamic content.
      The current time is <span class="lift:HelloWorld">now</span>.
    </div>
  </body>
</html>
```

This template invokes the `HelloWorld` snippet defined in `HelloWorld.scala`:

Listing 3.4: HelloWorld.scala

```
package code
package snippet

import lib._

import net.liftweb._
import util.Helpers._
import common._
import java.util.Date
```

```

class HelloWorld {
  lazy val date: Box[Date] = DependencyFactory.inject[Date] // inject the date

  def render = "* *" #> date.map(_.toString)
}
```

And the dynamic content becomes:

```
<span>Thu Dec 30 16:31:13 PST 2010</span>
```

The `HelloWorld` snippet code is simple.

```
lazy val date: Box[Date] = DependencyFactory.inject[Date]
```

Uses dependency injection (see 8.2 on page 94) to get a `Date` instance.

Then:

```
def render = "* *" #> date.map(_.toString)
```

Creates a CSS Selector Transform (see 7.10 on page 85) that inserts the `String` value of the injected `Date` into the markup, in this case the `<span>` that invoked the snippet.

3.4.4 Embedded Example

We've seen how we can embed a template using: `<div class="lift:embed?what=_embedme">xxx</div>`.

Let's look at the `_embedme.html` template:

```
<!DOCTYPE html>
<html>
  <head>
    <meta content="text/html; charset=UTF-8" http-equiv="content-type" />
    <title>I'm embeded</title>
  </head>
  <body class="lift:content_id=main">
    <div id="main">
      Howdy. I'm a bit of embedded content. I was
      embedded from <span class="lift:Embedded.from">???</span>.
    </div>
  </body>
</html>
```

And the invoked `Embedded.scala` program:

Listing 3.5: Embedded.scala

```
package code
package snippet

```

```
import lib._

import net.liftweb._
import http._
import util.Helpers._
import common._
import java.util.Date

/**
 * A snippet that lists the name of the current page
 */
object Embedded {
  def from = "*" #> S.location.map(_.name)
}
```

The template invokes the `from` method on the `Embedded` snippet. In this case, the snippet is an `object` singleton because it does not take any constructor parameters and has no instance variabled.

The `from` method:

```
  def from = "*" #> S.location.map(_.name)
```

Creates a CSS Selector Transform that replaces the contents with the `name` of the current `location`.

3.4.5 Param Example

Above, we saw how to create a `Loc[ParamInfo]` to capture URL parameters. Let's look at the `/param/xxx` page and see how we can access the parameters:

Listing 3.6: param.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta content="text/html; charset=UTF-8" http-equiv="content-type" />
    <title>Param</title>
  </head>
  <body class="lift:content_id=main">
    <div id="main" class="lift:surround?with=default;at=content">
      <div>
        Thanks for visiting this page. The parameter is
        <span class="lift:ShowParam">???</span>.
      </div>

      <div>
        Another way to get the param: <span class="lift:Param">???</span>.
      </div>

    </div>
  </body>
```

```
</html>
```

And let's look at two different snippets that can access the `ParamInfo` for the page:

Listing 3.7: Param.scala

```scala
package code
package snippet

import lib._

import net.liftweb._
import util.Helpers._
import common._
import http._
import sitemap._
import java.util.Date

// capture the page parameter information
case class ParamInfo(theParam: String)

// a snippet that takes the page parameter information
class ShowParam(pi: ParamInfo) {
  def render = "*" #> pi.theParam
}

object Param {
  // Create a menu for /param/somedata
  val menu = Menu.param[ParamInfo]("Param", "Param",
                                   s => Full(ParamInfo(s)),
                                   pi => pi.theParam) / "param"
  lazy val loc = menu.toLoc

  def render = "*" #> loc.currentValue.map(_.theParam)
}
```

Each snippet has a `render` method. However, the `ShowParam` class takes a constructor parameter which contains the `ParamInfo` from the current `Loc[_]`. If the current `Loc` does not have the type parameter `ParamInfo`, no instance of `ShowParam` would be created and the snippet could not be resolved. But we do have a Loc[ParamInfo], so Lift constructs a `ShowParam` with the `Loc`'s `currentValue` and then the `render` method is invoked and it returns a CSS Selector Transform which is a `NodeSeq => NodeSeq`.

The `object Param`'s `render` method accesses the `Loc[ParamInfo]` directly. The `render` method gets the `Loc`'s `currentValue` and uses that to calculate the return value, the CSS Selector Transform.

3.4.6 Recursive

Lift's snippets are evaluated lazily. This means that the body of the snippet is not executed until the outer snippet is executed which allows you to return markup from a snippet that itself contains a snippet or alternatively, choose part of the snippet body that itself contains a snippet invocation. For example, in this markup:

Listing 3.8: recurse.html

```
<div id="main" class="lift:surround?with=default&at=content">
  <div>
    This demonstrates Lift's recursive snippets
  </div>

  <div class="lift:Recurse">
    <div id="first" class="lift:FirstTemplate">
      The first template.
    </div>

    <div id="second" class="lift:SecondTemplate">
      The second template.
    </div>
  </div>

  <div>
    <ul>
      <li>Recursive: <a href="/recurse/one">First snippet</a></li>
      <li>Recursive: <a href="/recurse/two">Second snippet</a></li>
      <li>Recursive: <a href="/recurse/both">Both snippets</a></li>
    </ul>
  </div>
</div>
```

The `Recurse` snippet chooses one of both of the `<div>`'s, each of which invokes a snippet themselves. Here's the Scala:

Listing 3.9: Recurse.scala

```
package code
package snippet

import lib._

import net.liftweb._
import util._
import Helpers._
import http._
import scala.xml.NodeSeq

/**
 * The choices
 */
sealed trait Which
final case class First() extends Which
final case class Second() extends Which
final case class Both() extends Which

/**
 * Choose one or both of the templates
 */
class Recurse(which: Which) {
  // choose the template
  def render = which match {
    case First() => "#first ^^" #> "*" // choose only the first template
    case Second() => "#second ^^" #> "*" // choose only the second template
    case Both() => ClearClearable // it's a passthru
  }
}

/**
 * The first template snippet
 */
object FirstTemplate {
  // it's a passthru, but has the notice side effect
  def render(in: NodeSeq) = {
    S.notice("First Template Snippet executed")
    in
  }
}

/**
 * The second template snippet
 */
object SecondTemplate {
  // it's a passthru, but has the notice side effect
  def render(in: NodeSeq) = {
    S.notice("Second Template Snippet executed")
    in
  }
}
```

Depending on the value of `which`, one or both parts of the markup will be chosen. And each part of the markup itself invokes a snippet which displays a notice and passes the markup through.

Using this technique, you can have a snippet that chooses one or many different snippets or returns a `lift:embed` snippet, thus allowing for very dynamic markup generation.

3.4.7 Summary

We've seen some simple examples of Lift's snippet mechanism used to generate dynamic content. You can read more on snippets (see 7.1 on page 78).

3.5 Wrap up

In this chapter, we've seen how to define application behavior on `Boot.scala`. We've explored Lift's `SiteMap` which is used to generate navigation and enforce access control. We've seen how Lift's templating system works (well, there are actually a bunch of different ways to template in Lift, but we've explored to most common mechanism.) We've seen how snippets work.

In the next chapter, we'll take a dive into Lift's form handling.

Chapter 4

Forms

In this chapter, we'll see how Lift processes templates. We'll start with form processing the old fashioned way (where the designer names the inputs and the application maps those names to variables) through multi-page input forms and Ajax form support.

4.1 Old Fashioned Dumb Forms

Let's take a look at the HTML for a form:

Listing 4.1: dumb.html

```
<div id="main" class="lift:surround?with=default&at=content">
  <div>
    This is the simplest type of form processing... plain old
    mechanism of naming form elements and processing the form elements
    in a post-back.
  </div>

  <div>
    <form action="/dumb" method="post" class="lift:DumbForm">
      Name: <input name="name"><br>
      Age: <input name="age"><br>
      <input type="submit" value="Submit">
    </form>
  </div>
</div>
```

Okay... looks pretty normal... we define a form. The only thing we do is associate the behavior with the form with the `class="lift:DumbForm"` attribute on the `<form>` tag. The page is a post-back which means that the form is posted to the same URL that served the original content.

Let's see the code to process the form:

Listing 4.2: DumbForm.scala

```
package code
package snippet
```

```

import net.liftweb._
import http._
import scala.xml.NodeSeq

/**
 * A snippet that grabs the query parameters
 * from the form POST and processes them
 */
object DumbForm {
  def render(in: NodeSeq): NodeSeq = {

    // use a Scala for-comprehension to evaluate each parameter
    for {
      r <- S.request if r.post_? // make sure it's a post
      name <- S.param("name") // get the name field
      age <- S.param("age") // get the age field
    } {
      // if everything goes as expected,
      // display a notice and send the user
      // back to the home page
      S.notice("Name: "+name)
      S.notice("Age: "+age)
      S.redirectTo("/")
    }

    // pass through the HTML if we don't get a post and
    // all the parameters
    in
  }
}
```

It's pretty simple. If the request is a post and the query parameters exist, then display notices with the name and age and redirect back the application's home page.

There are plenty of reasons not to do things this way.

First, if there's a naming mis-match between the HTML and the Scala code, you might miss a form field... and keeping the naming aligned is not always easy.

Second, forms with predictable names lead to replay attacks. If an attacker can capture the form submits you've made and substitute new values for import fields, they can more easily hack your application.

Third, keeping state around becomes very difficult with manual forms. You have to resort to hidden fields that contain primary keys or other information that can be tampered with.

Lift provides you with much more powereful and secure mechanisms for dealing with HTML forms.

4.2 OnSubmit

Some of Lift's design reflects VisualBasic... associating user behavior with a user interface element. It's a simple, yet very powerful concept. Each form element is associated with a function on the server[1]. Further, because functions in Scala close over scope (capture the variables currently in scope), it's both easy and secure to keep state around without exposing that state to the web client.

So, let's see how it works. First, the HTML:

Listing 4.3: onsubmit.html

```
<div id="main" class="lift:surround?with=default&at=content">
  <div>
    Using Lift's SHtml.onSubmit, we've got better control
    over the form processing.
  </div>

  <div>
    <form class="lift:OnSubmit?form=post">
      Name: <input name="name"><br>
      Age: <input name="age" value="0"><br>
      <input type="submit" value="Submit">
    </form>
  </div>
</div>
```

The only different thing in this HTML is `<form class="lift:OnSubmit?form=post">`. The snippet, behavior, of the form is to invoke `OnSubmit.render`. The `form=post` attribute makes the form into a post-back. It sets the `method` and `action` attributes on the `<form>` tag: `<form method="post" action="/onsubmit">`.

Let's look at the snippet:

Listing 4.4: OnSubmit.scala

```
package code
package snippet

import net.liftweb._
import http._
import util.Helpers._
import scala.xml.NodeSeq

/**
 * A snippet that binds behavior, functions,
 * to HTML elements
 */
object OnSubmit {
  def render = {
    // define some variables to put our values into
    var name = ""
```

[1]Before you get all upset about statefulness and such, please read about Lift and State (see 20 on page 131).

```
    var age = 0

    // process the form
    def process() {
      // if the age is < 13, display an error
      if (age < 13) S.error("Too young!")
      else {
        // otherwise give the user feedback and
        // redirect to the home page
        S.notice("Name: "+name)
        S.notice("Age: "+age)
        S.redirectTo("/")
      }
    }

    // associate each of the form elements
    // with a function... behavior to perform when the
    // for element is submitted
    "name=name" #> SHtml.onSubmit(name = _) & // set the name
    // set the age variable if we can convert to an Int
    "name=age" #> SHtml.onSubmit(s => asInt(s).foreach(age = _)) &
    // when the form is submitted, process the variable
    "type=submit" #> SHtml.onSubmitUnit(process)
  }
}
```

Like `DumbForm.scala`, the snippet is implemented as a singleton. The render method declares two variables: `name` and `age`. Let's skip the `process()` method and look at the was we're associating behavior with the form elements.

`"name=name" #> SHtml.onSubmit(name = _)` takes the incoming HTML elements with the `name` attribute equal to "name" and, via the `SHtml.onSubmit` method, associating a function with the form element. The function takes a `String` parameter and sets the value of the `name` variable to the `String`. The resulting HTML is `<input name="F10714412223674KM">`. The new `name` attribute is a GUID (globally unique identifier) that associates the function (set the name to the input) with the form element. When the form is submitted, via normal HTTP post or via Ajax, the function will be executed with the value of the form element. On form submit, perform this function.

Let's see about the age form field: `"name=age" #> SHtml.onSubmit(s => asInt(s).foreach(age = _))`. The function that's executed uses `Helpers.asInt` to try to parse the `String` to an `Int`. If the parsing is successful, the `age` variable is set to the parsed `Int`.

Finally, we associate a function with the submit button: `"type=submit" #> SHtml.onSubmitUnit(process)`. `SHtml.onSubmitUnit` method takes a function that takes no parameters (rather than a function that takes a single `String` as a parameter) and applies that function when the form is submitted.

The `process()` method closes over the scope of the `name` and `age` variables and when that method is lifted to a function, it still closes over the variables... that means that when the function is applied, it refers to the same instances of the `name` and `age` variables as the other functions in this method. However, if we had 85 copies of the form open in 85 browsers, each would be closing over different instances of the `name` and `age` variables. In this way, Lift allows your application to contain complex state without exposing that complex state to the browser.

The problem with this form example is that if you type an incorrect age, the whole form is reset. Let's see how we can do better error handling.

4.3 Stateful Snippets

In order for us to give the user a better experience, we need to capture the state of the name and age variables across the multiple form submissions. The mechanism that Lift has for doing this is the Stateful Snippet[2]. A snippet that subclasses `StatefulSnippet` has an extra hidden parameter automatically inserted into the form which ensures that during processing of that form, the same instance of the StatefulSnippet will be used[3].

Let's look at the HTML template:

Listing 4.5: stateful.html

```
<div id="main" class="lift:surround?with=default&at=content">
  <div>
    Using stateful snippets for a better
    user experience
  </div>

  <div>
    <div class="lift:Stateful?form=post">
      Name: <input name="name"><br>
      Age: <input name="age" value="0"><br>
      <input type="submit" value="Submit">
    </div>
  </div>
</div>
```

The template looks pretty much like the template in `onsubmit.html`. Let's look at the snippet itself:

Listing 4.6: Stateful.scala

```
package code
package snippet

import net.liftweb._
import http._
import common._
import util.Helpers._
import scala.xml.NodeSeq
```

[2]There are no stateless snippets. A Stateful Snippet doesn't consume any more server-side resources than does a form composed via `SHtml.onSubmit()`. Oh, and state is not a barier to scalaing. See Chapter 20.

[3]Earlier I talked about the security implications of hidden form parameters. The hidden parameter mechanism is not vulnerable to the same issues because the hidden parameter itself is just a GUID that causes a function to be invoked on the server. No state is exposed to the client, so there's nothing for a hacker to capture or mutate that would allow for the exploitation of a vulnerability.

```

/**
 * A stateful snippet. The state associated with this
 * snippet is in instance variables
 */
class Stateful extends StatefulSnippet {
  // state unique to this instance of the stateful snippet
  private var name = ""
  private var age = "0"

  // capture from whence the user came so we
  // can send them back
  private val whence = S.referer openOr "/"

  // StatefulSnippet requires an explicit dispatch
  // to the method.
  def dispatch = {case "render" => render}

  // associate behavior with each HTML element
  def render =
    "name=name" #> SHtml.text(name, name = _, "id" -> "the_name") &
    "name=age" #> SHtml.text(age, age = _) &
    "type=submit" #> SHtml.onSubmitUnit(process)

  // process the form
  private def process() =
    asInt(age) match {
      case Full(a) if a < 13 => S.error("Too young!")
      case Full(a) => {
        S.notice("Name: "+name)
        S.notice("Age: "+a)
        S.redirectTo(whence)
      }

      case _ => S.error("Age doesn't parse as a number")
    }
}
```

There's a fair amount different here. First, the class definition: `class Stateful extends StatefulSnippet`. Because the snippet instance itself contains state, it can't be an object singleton. It must be declared as a class so there are multiple instances.

We capture state (`name`, `age` and `from` whence the user came), in instance variables.

`StatefulSnippets` require a `dispatch` method which does method dispatching explicitly rather than "by-convention."

The render method uses familiar CSS Selector Transforms to associate markup with behavior. However, rather than using `SHtml.onSubmit`, we're using `SHtml.text` to explicitly generate an HTML `<input>` element with both the `name` and `value` attributes set. In the case of the first input, we're also explicitly setting the `id` attribute. We're not using it in the application, but it's a way to demonstrate how to add extra attributes.

Finally, the `process()` method attempts to covert the age `String` into an `Int`. If it's an `Int`, but less than 13, we present an error. If the `String` cannot be parsed to an `Int`, we present an error, otherwise we do notify the user and go back to the page the user came from.

Note in this example, we preserve the form values, so if you type something wrong in the `name` or `age` fields, what you typed is presented to you again.

The big difference between the resulting HTML for `StatefulSnippets` and other snippets is the insertion of `<input name="F1071441222401LO3" type="hidden" value="true">` in the form. This hidden field associates the snippet named “Stateful” with the instance of `Stateful` that was used to initially generate the form.

Let’s look at an alternative mechanism for creating a nice user experience.

4.4 RequestVars

In this example, we’re going to preserve state during the request by placing state in `RequestVars` (see 7.8 on page 84).

Lift has type-safe containers for state called `XXXVars`. There are `SessionVars` that have session scope, `WizardVars` that are scoped to a Wizard and `RequestVars` that are scoped to the current request[4]. `Vars` are defined as singletons: `private object name extends RequestVar("")`. They are typed (in this case, the type is `String`) and they have a default value.

So, let’s look at the HTML which looks shockingly like the HTML in the last two examples:

Listing 4.7: requestvar.html

```
<div id="main" class="lift:surround?with=default&at=content">
  <div>
    Using RequestVars to store state
  </div>

  <div>
    <form class="lift:ReqVar?form=post">
      Name: <input name="name"><br>
      Age: <input name="age" id="the_age" value="0"><br>
      <input type="submit" value="Submit">
    </form>
  </div>
</div>
```

Now, let’s look at the snippet code:

Listing 4.8: ReqVar.scala

```
package code
package snippet

import net.liftweb._
import http._
import common._
```

[4]In this case, “request” means full HTML page load and all subsquent Ajax operations on that page. There’s also a `TransientRequestVar` that has the scope of the current HTTP request.

```
import util.Helpers._
import scala.xml.NodeSeq

/**
 * A RequestVar-based snippet
 */
object ReqVar {
  // define RequestVar holders for name, age, and whence
  private object name extends RequestVar("")
  private object age extends RequestVar("0")
  private object whence extends RequestVar(S.referer openOr "/")

  def render = {
    // capture the whence... which forces evaluation of
    // the whence RequestVar unless it's already been set
    val w = whence.is

    // we don't need an explicit function because RequestVar
    // extends Settable{type=String}, so Lift knows how to
    // get/set the RequestVar for text element creation
    "name=name" #> SHtml.textElem(name) &
    // add a hidden field that sets whence so we
    // know where to go
    "name=age" #> (SHtml.textElem(age) ++
                   SHtml.hidden(() => whence.set(w))) &
    "type=submit" #> SHtml.onSubmitUnit(process)
  }

  // process the same way as
  // in Stateful
  private def process() =
    asInt(age.is) match {
      case Full(a) if a < 13 => S.error("Too young!")
      case Full(a) => {
        S.notice("Name: "+name)
        S.notice("Age: "+a)
        S.redirectTo(whence)
      }

      case _ => S.error("Age doesn't parse as a number")
    }
}
```

The snippet is a singleton because the state is kept in the `RequestVars`.

We use `SHtml.textElem()` to generate the `<input>` tag. We can pass the `RequestVar` into the method and the function that gets/sets the `RequestVar` is generated for us.

The use of this mechanism for doing stateful forms versus the StatefulSnippet mechanism is one of personal choice. Neither one is better, they are just different.

Next, let's look at how to get more granular with error messages.

4.5 Field Errors

In the prior examples, we displayed an error to the user. However, we didn't tell the user what field resulted in the error. Let's be a little more granular about error reporting.

First, let's look at the HTML:

Listing 4.9: fielderror.html

```html
<div id="main" class="lift:surround?with=default&at=content">
  <div>
    Let's get granular about error messages
  </div>

  <div>
    <div class="lift:FieldErrorExample?form=post">
      Name: <input name="name"><br>
      Age: <span class="lift:Msg?id=age&errorClass=error">error</span>
      <input name="age" id="the_age" value="0"><br>
      <input type="submit" value="Submit">
    </div>
  </div>
</div>
```

This HTML is different. Note: `Age: <span class="lift:Msg?id=age&errorClass=error">error</s` We mark an area in the markup to put the error message.

Let's look at our snippet code which is very similar to `Stateful.scala` with a small, but important difference:

Listing 4.10: FieldErrorExample.scala

```scala
package code
package snippet

import net.liftweb._
import http._
import common._
import util.Helpers._
import scala.xml.NodeSeq

/**
 * A StatefulSnippet like Stateful.scala
 */
class FieldErrorExample extends StatefulSnippet {
  private var name = ""
  private var age = "0"
  private val whence = S.referer openOr "/"

  def dispatch = {case _ => render}

  def render =
    "name=name" #> SHtml.text(name, name = _) &
    "name=age" #> SHtml.text(age, age = _) &
    "type=submit" #> SHtml.onSubmitUnit(process)

  // like Stateful
  private def process() =
    asInt(age) match {
      // notice the parameter for error corresponds to
      // the id in the Msg span
      case Full(a) if a < 13 => S.error("age", "Too young!")
      case Full(a) => {
        S.notice("Name: "+name)
        S.notice("Age: "+a)
        S.redirectTo(whence)
      }

      // notice the parameter for error corresponds to
      // the id in the Msg span
      case _ => S.error("age", "Age doesn't parse as a number")
    }
}
```

The key difference is: `case Full(a) if a < 13 => S.error("age", "Too young!")`. Note that we pass "age" to `S.error` and this corresponds to the `id` in the `Msg` snippet in markup. This tells Lift how to associate the error message and the markup.

But there's a better way to do complex forms in Lift: `LiftScreen`.

4.6 LiftScreen

Much of what we do to build web applications is generating screens that associate input with dynamic content. Lift provides Screen and Wizard for building single page and multi-page input forms with validation, back-button support, etc.

So, let's look at the HTML for a screen:

Listing 4.11: screen.html

```
<div id="main" class="lift:surround?with=default&at=content">
  <div>
    Let's use Lift's LiftScreen to build complex
    simple screen input forms.
  </div>

  <div class="lift:ScreenExample">
    Put your form here
  </div>
</div>
```

We don't explicitly declare the form elements. We just point to the snippet which looks like:

Listing 4.12: ScreenExample.scala

```
package code
package snippet
```

```

import net.liftweb._
import http._

/**
 * Declare the fields on the screen
 */
object ScreenExample extends LiftScreen {
  // here are the fields and default values
  val name = field("Name", "")

  // the age has validation rules
  val age = field("Age", 0, minVal(13, "Too Young"))

  def finish() {
    S.notice("Name: "+name)
    S.notice("Age: "+age)
  }
}
```

In the screen, we define the fields and their validation rules and then what to do when the screen is finished. Lift takes care of the rest.

The markup for generating the form, by default, is found in `/templates-hidden/wizard-all.html`. You can also select templates on a screen-by-screen basis.

4.7 Wizard

`LiftScreen` is great for single screen applications. If you've got input and validation that requires multiple screens, Wizard is what you want. We'll skip the markup 'cause it's just a snippet invocation. Here's the wizard code:

Listing 4.13: WizardExample.scala

```
package code
package snippet

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

/**
 * Define the multi-page input screen
 */
object WizardExample extends Wizard {

  // define the first screen
  val screen1 = new Screen {
    val name = field("Name", "")
    val age = field("Age", 0, minVal(13, "Too Young"))
  }

  // define the second screen
  val screen2 = new Screen {

    // a radio button
    val rad = radio("Radio", "Red", List("Red", "Green", "Blue"))

    // a select
    val sel = select("Select", "Archer", List("Elwood", "Archer", "Madeline"))

    // want a text area... yeah, we got that
    val ta = textarea("Text Area", "")

    // here are password inputs with minimum lenght
    val pwd1 = password("Password", "", valMinLen(6, "Password too short"))

    // and a custom validator
    val pwd2 = password("Password (re-enter)", "", mustMatch _)

    // return a List[FieldError]... there's an implicit conversion
    // from String to List[FieldError] that inserts the field's ID
    def mustMatch(s: String): List[FieldError] =
      if (s != pwd1.is) "Passwords do not match" else Nil

  }

  def finish() {
    S.notice("Name: "+screen1.name)
    S.notice("Age: "+screen1.age)
  }
}
```

It's declarative just like the screen example above. The back button works. You can have multiple wizards active in multiple tabs in your browser and they don't intefer with each other.

4.8 Ajax

In addition to full-page HTML, Lift support Ajax forms. Because Lift's forms are functions on the server-side associated with GUIDs in the browser, switching a form from full page load to Ajax is, well, pretty trivial. Let's look at the markup:

Listing 4.14: ajax.html

```
<div id="main" class="lift:surround?with=default&at=content">
  <div>
    An example of doing forms with Ajax.
  </div>

  <form class="lift:form.ajax">
    <div class="lift:AjaxExample">
      Name: <input name="name"><br>
      Age: <span class="lift:Msg?id=age&errorClass=error">error</span><input name="age" id="
    <input type="submit" value="Submit">
   </div>
  </form>
</div>
```

The key difference is: `<form class="lift:form.ajax">`. This invokes Lift's built-in `form` snippet and designates the current form as an Ajax form. Then the snippet does the following:

Listing 4.15: AjaxExample.scala

```scala
package code
package snippet

import net.liftweb._
import http._
import common._
import util.Helpers._
import js._
import JsCmds._
import JE._
import scala.xml.NodeSeq

/**
 * Ajax for processing... it looks a lot like the Stateful example
 */
object AjaxExample {
  def render = {
    // state
    var name = ""
    var age = "0"
    val whence = S.referer openOr "/"

    // our process method returns a
    // JsCmd which will be sent back to the browser
    // as part of the response
    def process(): JsCmd= {

      // sleep for 400 millis to allow the user to
      // see the spinning icon
      Thread.sleep(400)

      // do the matching
      asInt(age) match {
        // display an error and otherwise do nothing
        case Full(a) if a < 13 => S.error("age", "Too young!"); Noop

        // redirect to the page that the user came from
        // and display notices on that page
        case Full(a) => {
          RedirectTo(whence, () => {
            S.notice("Name: "+name)
            S.notice("Age: "+a)
          })
        }

        // more errors
        case _ => S.error("age", "Age doesn't parse as a number"); Noop
      }
    }

    // binding looks normal
    "name=name" #> SHtml.text(name, name = _, "id" -> "the_name") &
    "name=age" #> (SHtml.text(age, age = _) ++ SHtml.hidden(process))
   }
}
```

The code looks a lot like the Stateful code. Except that we don't bind to the submit button (the submit button is not serialized over Ajax), so we have to add a hidden field to the age field which does the processing.

The `process()` method returns a `JsCmd` which is the JavaScript command to send back to the browser in response to the Ajax form submission. In this case, we're either using `S.error` to display error notices followed by a `Noop` or we're doing a redirect.

We pause for 400 milliseconds in the `process()` method so that the user can see the spinner in the browser indicating that an Ajax operation is taking place.

But the core take-away is that normal HTML processing and Ajax processing are almost identical and both are super-easy.

4.9 But sometimes Old Fashioned is good

In this chapter, we've explored Lift's form building and processing features and demonstrated the power and value of associating GUIDs on the client with functions on the server. However, sometimes it's nice to have parameter processing via URL parameters... and that's easy to do with Lift as well.

Every page in the examples for this chapter contain:

```
BADTAB<form action="/query">
BADTAB  <input name="q">
BADTAB  <input type="submit" value="Search">
BADTAB</form>
```

This is a plain old form that generates a URL like: `http://localhost:8080/query?q=catfood` This URL can be copied, pasted, shared, etc.

Processing this URL is easy:

Listing 4.16: Query.scala

```
package code
package snippet

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

object Query {
  def results = ClearClearable andThen
  "li *" #> S.param("q"). // get the query parameter
  toList. // convert the Box to a List
  flatMap(q => {
    ("You asked: "+q) :: // prepend the query
    (1 to toInt(q)).toList.map(_.toString) // if it can be converted to an Int
    // convert it and return a sequence of Ints
  })
}
```

Using `S.param("param_name")` we can extract the query parameter and do something with it.

4.10 Conclusion

Lift's form generation and processing tools offer a wide variety of mechanisms to securely, simply and powerfully generate and process HTML forms either as part of full HTTP requests or via Ajax requests.

Chapter 5

HTTP and REST

We explored Lift's HTML generation features. Let's dive down to a lower level and handle HTTP requests REST-style. The code for this chapter can be found at https://github.com/dpp/simply_-lift/tree/master/samples/http_rest

5.1 Introduction

Lift gives you access to low level HTTP requests, either within the scope of an session or outside the scope of a session. In sessionless or stateless mode, Lift does not use the container's session management machinery to add a cookie to the `HTTP` response and does not make `SessionVar` or `ContainerVar` available during the request. Stateless REST requests do not require session affinity. Authentication for stateless REST handling can be done via OAuth. If the requests are handled statefully, a container session will be created if the `JSESSIONID` cookie is not supplied as part of the request and the `JSESSIONID` cookie will be included with the response.

Lift makes use of Scala's pattern matching to allow you match incoming HTTP requests, extract values as part of the pattern matching process and return the results. Scala's pattern matching is very, very powerful. It allows both the declaration of a pattern that must be matched, wildcard values (a sub-expression may match any supplied value), wildcard values extracted into variables, and explicit extractors (imperative logic applied to a value to determine if it should match and if it does, extract it into a variable). Lift tests a Scala `PartialFunction[Req, () => Box[LiftResponse]]` to see if it is defined for a given `Req`, which represents an HTTP request. If there is a match, Lift will take the resulting function, apply it to get a `Box[LiftResponse]` and if the `Box` is full, the response will be sent back to the browser. That's a mouth-full. Let's look at examples.

5.2 REST the hard way

Let's take a look at the raw level of doing REST with Lift: taking an incoming HTTP request and transforming it into a function that returns a `Box[LiftResponse]` (and don't worry, it gets easier, but we're starting with the ugly verbose stuff so you get an idea of what's happening under the covers):

Listing 5.1: BasicExample.scala

```
package code
package lib

import model._

import net.liftweb._
import common._
import http._

/**
 * A simple example of a REST style interface
 * using the basic Lift tools
 */
object BasicExample {
  /*
   * Given a suffix and an item, make a LiftResponse
   */
  private def toResponse(suffix: String, item: Item) =
    suffix match {
      case "xml" => XmlResponse(item)
      case _ => JsonResponse(item)
    }

  /**
   * Find /simple/item/1234.json
   * Find /simple/item/1234.xml
   */
  lazy val findItem: LiftRules.DispatchPF = {
    case Req("simple" :: "item" :: itemId :: Nil, // path
             suffix, // suffix
             GetRequest) =>
               () => Item.find(itemId).map(toResponse(suffix, _))
  }

  /**
   * Find /simple2/item/1234.json
   */
  lazy val extractFindItem: LiftRules.DispatchPF = {
    // path with extractor
    case Req("simple2" :: "item" :: Item(item) :: Nil,
             suffix, GetRequest) =>
               // a function that returns the response
               () => Full(toResponse(suffix, item))
  }
}
```

One additional piece of the puzzle is hooking up the handlers to Lift. This is done in `Boot.scala` with the following lines:

```
    // the stateless REST handlers
    LiftRules.statelessDispatchTable.append(BasicExample.findItem)
    LiftRules.statelessDispatchTable.append(BasicExample.extractFindItem)
```

```

  // stateful versions of the same
  // LiftRules.dispatch.append(BasicExample.findItem)
  // LiftRules.dispatch.append(BasicExample.extractFindItem)
```

Let's break down the code. First, each handler is a `PartialFunction[Req, () => Box[LiftResponse]]`, but we can use a shorthand of `LiftRules.dispatchPF` which is a Scala type that aliases the partial function.

```
lazy val findItem: LiftRules.DispatchPF =
```

defines `findItem` which has the type signature of a request dispatch handler.

```
  case Req("simple" :: "item" :: itemId :: Nil, // path
           suffix, // suffix
           GetRequest) =>
```

Defines a pattern to match. In this case, any 3 part path that has the first two parts `/simple/item` will be matched. The third part of the path will be extracted to the variable `itemId`. The suffix of the last path item will be extracted to the variable `suffix` and the request must be a GET.

If the above criteria is met, then the partial function is defined and Lift will apply the partial function to get the resulting `() => Box[LiftResponse]`.

```
            () => Item.find(itemId).map(toResponse(suffix, _))
```

This is a function that finds the `itemId` and converts the resulting `Item` to a response based on the request suffix. The `toResponse` method looks like:

```
 /*
  * Given a suffix and an item, make a LiftResponse
  */
 private def toResponse(suffix: String, item: Item) =
   suffix match {
     case "xml" => XmlResponse(item)
     case _ => JsonResponse(item)
   }
```

That's all pretty straight forward, if a little verbose. Let's look at the other example in this file. It uses an extractor to convert the `String` of the third element of the request path to an `Item`:

```
  // path with extractor
  case Req("simple2" :: "item" :: Item(item) :: Nil,
           suffix, GetRequest) =>
```

In this case, the pattern will not be matched unless that third element of the path is a valid `Item`. If it is, the variable `item` will contain the `Item` for processing. Converting this to a valid response looks like:

```
            // a function that returns the response
            () => Full(toResponse(suffix, item))
```

Let's look at the `object Item`'s unapply method to see how the extraction works:

```
 /**
  * Extract a String (id) to an Item
  */
 def unapply(id: String): Option[Item] = Item.find(id)
```

In fact, let's look at the entire `Item` code listing. As promised, *Simply Lift*, does not explicitly cover persistence. This class is an in-memory mock persistence class, but it behaves like any other persistence mechanism in Lift.

Listing 5.2: Item.scala

```
package code
package model

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

import scala.xml.Node

/**
 * An item in inventory
 */
case class Item(id: String, name: String,
                description: String,
                price: BigDecimal, taxable: Boolean,
                weightInGrams: Int, qnty: Int)

/**
 * The Item companion object
 */
object Item {
  private implicit val formats =
    net.liftweb.json.DefaultFormats + BigDecimalSerializer

  private var items: List[Item] = parse(data).extract[List[Item]]

  private var listeners: List[Item => Unit] = Nil

  /**
   * Convert a JValue to an Item if possible
   */
  def apply(in: JValue): Box[Item] = Helpers.tryo{in.extract[Item]}

  /**
   * Extract a String (id) to an Item
   */
  def unapply(id: String): Option[Item] = Item.find(id)

  /**
```

```
   * Extract a JValue to an Item
   */
  def unapply(in: JValue): Option[Item] = apply(in)

  /**
   * The default unapply method for the case class.
   * We needed to replicate it here because we
   * have overloaded unapply methods
   */
  def unapply(in: Any): Option[(String, String,
                               String,
                               BigDecimal, Boolean,
                               Int, Int)] = {
    in match {
      case i: Item => Some((i.id, i.name, i.description,
                            i.price, i.taxable,
                            i.weightInGrams, i.qnty))
      case _ => None
    }
  }

  /**
   * Convert an item to XML
   */
  implicit def toXml(item: Item): Node =
    <item>{Xml.toXml(item)}</item>


  /**
   * Convert the item to JSON format. This is
   * implicit and in the companion object, so
   * an Item can be returned easily from a JSON call
   */
  implicit def toJson(item: Item): JValue =
    Extraction.decompose(item)

  /**
   * Convert a Seq[Item] to JSON format. This is
   * implicit and in the companion object, so
   * an Item can be returned easily from a JSON call
   */
  implicit def toJson(items: Seq[Item]): JValue =
    Extraction.decompose(items)

  /**
   * Convert a Seq[Item] to XML format. This is
   * implicit and in the companion object, so
   * an Item can be returned easily from an XML REST call
   */
  implicit def toXml(items: Seq[Item]): Node =
    <items>{
      items.map(toXml)
    }</items>

  /**
   * Get all the items in inventory
   */
  def inventoryItems: Seq[Item] = items

  // The raw data
  private def data =
"""[
  {"id": "1234", "name": "Cat Food",
  "description": "Yummy, tasty cat food",
  "price": 4.25,
  "taxable": true,
  "weightInGrams": 1000,
  "qnty": 4
  },
  {"id": "1235", "name": "Dog Food",
  "description": "Yummy, tasty dog food",
  "price": 7.25,
  "taxable": true,
  "weightInGrams": 5000,
  "qnty": 72
  },
  {"id": "1236", "name": "Fish Food",
  "description": "Yummy, tasty fish food",
  "price": 2,
  "taxable": false,
  "weightInGrams": 200,
  "qnty": 45
  },
  {"id": "1237", "name": "Sloth Food",
  "description": "Slow, slow sloth food",
  "price": 18.33,
  "taxable": true,
  "weightInGrams": 750,
  "qnty": 62
  },
]
"""

  /**
   * Select a random Item
   */
  def randomItem: Item = synchronized {
    items(Helpers.randomInt(items.length))
  }

  /**
   * Find an item by id
   */
  def find(id: String): Box[Item] = synchronized {
    items.find(_.id == id)
  }

  /**
```

```
   * Add an item to inventory
   */
  def add(item: Item): Item = {
    synchronized {
      items = item :: items.filterNot(_.id == item.id)
      updateListeners(item)
    }
  }

  /**
   * Find all the items with the string in their name or
   * description
   */
  def search(str: String): List[Item] = {
    val strLC = str.toLowerCase()

    items.filter(i =>
      i.name.toLowerCase.indexOf(strLC) >= 0 ||
                 i.description.toLowerCase.indexOf(strLC) >= 0)
  }

  /**
   * Deletes the item with id and returns the
   * deleted item or Empty if there's no match
   */
  def delete(id: String): Box[Item] = synchronized {
    var ret: Box[Item] = Empty

    val Id = id // an upper case stable ID for pattern matching

    items = items.filter {
      case i@Item(Id, _, _, _, _, _, _) =>
        ret = Full(i) // side effect
        false
      case _ => true
    }

    ret.map(updateListeners)
  }

  /**
   * Update listeners when the data changes
   */
  private def updateListeners(item: Item): Item = {
    synchronized {
      listeners.foreach(f =>
        Schedule.schedule(() => f(item), 0 seconds))

      listeners = Nil
    }
    item
  }

  /**
```

```
   * Add an onChange listener
   */
  def onChange(f: Item => Unit) {
    synchronized {
      // prepend the function to the list of listeners
      listeners ::= f
    }
  }

}

/**
 * A helper that will JSON serialize BigDecimal
 */
object BigDecimalSerializer extends Serializer[BigDecimal] {
  private val Class = classOf[BigDecimal]

  def deserialize(implicit format: Formats): PartialFunction[(TypeInfo, JValue), BigDecimal
    case (TypeInfo(Class, _), json) => json match {
      case JInt(iv) => BigDecimal(iv)
      case JDouble(dv) => BigDecimal(dv)
      case value => throw new MappingException("Can't convert " + value + " to " + Class)
    }
  }

  def serialize(implicit format: Formats): PartialFunction[Any, JValue] = {
    case d: BigDecimal => JDouble(d.doubleValue)
  }
}
```

Let's take a look at what the resulting output is:

```
dpp@raptor:~/proj/simply_lift/samples/http_rest$ curl http://localhost:8080/simple/item/123
{
  "id":"1234",
  "name":"Cat Food",
  "description":"Yummy, tasty cat food",
  "price":4.25,
  "taxable":true,
  "weightInGrams":1000,
  "qnty":4
}

dpp@raptor:~/proj/simply_lift/samples/http_rest$ curl http://localhost:8080/simple/item/123
<?xml version="1.0" encoding="UTF-8"?>
<item>
  <id>1234</id>
  <name>Cat Food</name>
  <description>Yummy, tasty cat food</description>
  <price>4.25</price>
  <taxable>true</taxable>
  <weightInGrams>1000</weightInGrams>
  <qnty>4</qnty>
</item>
```

```
dpp@raptor:~/proj/simply_lift/samples/http_rest$
```

5.3 Making it easier with `RestHelper`

The above example shows you how Lift deals with REST calls. However, it's a tad verbose. Lift's `RestHelper` trait contains a lot of very helpful shortcuts that make code more concise, easier to read and easier to maintain. Let's look at a bunch of examples and then we'll work through each one:

Listing 5.3: BasicWithHelper.scala

```scala
package code
package lib

import model._

import net.liftweb._
import common._
import http._
import rest._
import json._
import scala.xml._

/**
 * A simple example of a REST style interface
 * using the basic Lift tools
 */
object BasicWithHelper extends RestHelper {
  /*
   * Serve the URL, but have a helpful error message when you
   * return a 404 if the item is not found
   */
  serve {
    case "simple3" :: "item" :: itemId :: Nil JsonGet _ =>
      for {
        // find the item, and if it's not found,
        // return a nice message for the 404
        item <- Item.find(itemId) ?~ "Item Not Found"
      } yield item: JValue

    case "simple3" :: "item" :: itemId :: Nil XmlGet _ =>
      for {
        item <- Item.find(itemId) ?~ "Item Not Found"
      } yield item: Node
  }



  serve {
    // Prefix notation
    case JsonGet("simple4" :: "item" :: Item(item) :: Nil, _) =>
```

```
      // no need to explicitly create a LiftResponse
      // Just make it JSON and RestHelper does the rest
      item: JValue

    // infix notation
    case "simple4" :: "item" :: Item(item) :: Nil XmlGet _ =>
      item: Node
  }

  // serve a bunch of items given a single prefix
  serve ( "simple5" / "item" prefix {
    // all the inventory
    case Nil JsonGet _ => Item.inventoryItems: JValue
    case Nil XmlGet _ => Item.inventoryItems: Node

    // a particular item
    case Item(item) :: Nil JsonGet _ => item: JValue
    case Item(item) :: Nil XmlGet _ => item: Node
  })

  /**
   * Here's how we convert from an Item
   * to JSON or XML depending on the request's
   * Accepts header
   */
  implicit def itemToResponseByAccepts: JxCvtPF[Item] = {
    case (JsonSelect, c, _) => c: JValue
    case (XmlSelect, c, _) => c: Node
  }

  /**
   * serve the response by returning an item
   * (or a Box[Item]) and let RestHelper determine
   * the conversion to a LiftResponse using
   * the itemToResponseByAccepts partial function
   */
  serveJx[Item] {
    case "simple6" :: "item" :: Item(item) :: Nil Get _ => item
    case "simple6" :: "item" :: "other" :: item :: Nil Get _ =>
      Item.find(item) ?~ "The item you're looking for isn't here"
  }

  /**
   * Same as the serveJx example above, but we've
   * used prefixJx to avoid having to copy the path
   * prefix over and over again
   */
  serveJx[Item] {
    "simple7" / "item" prefixJx {
      case Item(item) :: Nil Get _ => item
      case "other" :: item :: Nil Get _ =>
        Item.find(item) ?~ "The item you're looking for isn't here"
    }
  }
```

```

}
```

The first thing is how we declare and register the RestHelper-based service:

```
/**
 * A simple example of a REST style interface
 * using the basic Lift tools
 */
object BasicWithHelper extends RestHelper {
```

Our BaseicWithHelper singleton extends the net.liftweb.http.rest.RestHelper trait. We register the dispatch in Boot.scala:

```
    LiftRules.statelessDispatchTable.append(BasicWithHelper)
```

This means that the whole BasicWithHelper singleton is a PartialFunction[Req, () => Box[LiftResponse]] that aggregates all the sub-patterns contained inside it. We defined the sub-patterns in a serve block which contains the pattern to match. For example:

```
  serve {
    case "simple3" :: "item" :: itemId :: Nil JsonGet _ =>
      for {
        // find the item, and if it's not found,
        // return a nice message for the 404
        item <- Item.find(itemId) ?~ "Item Not Found"
      } yield item: JValue

    case "simple3" :: "item" :: itemId :: Nil XmlGet _ =>
      for {
        item <- Item.find(itemId) ?~ "Item Not Found"
      } yield item: Node
  }
```

Let's break this down further:

```
case "simple3" :: "item" :: itemId :: Nil JsonGet _ =>
```

The above matches /simple3/item/xxx where xxx is extracted to the itemId variable. The request must also have an Accepts header that calls for JSON.

If the pattern matches, execute the following code:

```
      for {
        // find the item, and if it's not found,
        // return a nice message for the 404
        item <- Item.find(itemId) ?~ "Item Not Found"
      } yield item: JValue
```

Some things to notice, we didn't explicitly create a function that returns a Box[LiftResponse]. Instead, the type is Box[JValue]. RestHelper provides implicit conversions from

`Box[JValue]` to `() => Box[LiftResponse]`. Specifically, if the `Box` is a `Failure`, `RestHelper` will generate a 404 response with the `Failure` message as the 404's body. If the `Box` is `Full`, `RestHelper` will create a `JsonResponse` with the value in the payload. Let's take a look at the two cases:

```
dpp@raptor:~/proj/simply_lift/samples/http_rest$ curl http://localhost:8080/simple3/item/12
Item Not Found

dpp@raptor:~/proj/simply_lift/samples/http_rest$ curl http://localhost:8080/simple3/item/12
{
  "id":"1234",
  "name":"Cat Food",
  "description":"Yummy, tasty cat food",
  "price":4.25,
  "taxable":true,
  "weightInGrams":1000,
  "qnty":4
}
```

The XML example is pretty much the same, except we coerse the response to `Box[Node]` which `RestHelper` converts into an `XmlResponse`:

```
    case "simple3" :: "item" :: itemId :: Nil XmlGet _ =>
      for {
        item <- Item.find(itemId) ?~ "Item Not Found"
      } yield item: Node
```

Which results in the following:

```
dpp@raptor:~/proj/simply_lift/samples/http_rest$ curl -i -H "Accept: application/xml" http
HTTP/1.1 200 OK
Expires: Wed, 9 Mar 2011 01:48:38 UTC
Content-Length: 230
Cache-Control: no-cache; private; no-store
Content-Type: text/xml; charset=utf-8
Pragma: no-cache
Date: Wed, 9 Mar 2011 01:48:38 UTC
X-Lift-Version: Unknown Lift Version
Server: Jetty(6.1.22)

<?xml version="1.0" encoding="UTF-8"?>
<item>
  <id>1234</id>
  <name>Cat Food</name>
  <description>Yummy, tasty cat food</description>
  <price>4.25</price>
  <taxable>true</taxable>
  <weightInGrams>1000</weightInGrams>
  <qnty>4</qnty>
</item>
```

Okay... that's simpler because we define stuff in the `serve` block and the conversions from `JValue` and `Node` to the right response types is taken care of. Just to be explicit about where the implicit conversions are defined, they're in the `Item` singleton:

```
/**
 * Convert an item to XML
 */
implicit def toXml(item: Item): Node =
  <item>{Xml.toXml(item)}</item>


/**
 * Convert the item to JSON format. This is
 * implicit and in the companion object, so
 * an Item can be returned easily from a JSON call
 */
implicit def toJson(item: Item): JValue =
  Extraction.decompose(item)
```

Okay, so, yippee skippy, we can do simpler REST. Let's keep looking at examples of how we can make it even simpler. This example uses extractors rather than doing the explicit `Item.find`:

```
serve {
  // Prefix notation
  case JsonGet("simple4" :: "item" :: Item(item) :: Nil, _) =>
    // no need to explicitly create a LiftResponse
    // Just make it JSON and RestHelper does the rest
    item: JValue

  // infix notation
  case "simple4" :: "item" :: Item(item) :: Nil XmlGet _ =>
    item: Node
}
```

If you like DRY and don't want to keep repeating the same path prefixes, you can use `prefix`, for example:

```
// serve a bunch of items given a single prefix
serve ( "simple5" / "item" prefix {
  // all the inventory
  case Nil JsonGet _ => Item.inventoryItems: JValue
  case Nil XmlGet _ => Item.inventoryItems: Node

  // a particular item
  case Item(item) :: Nil JsonGet _ => item: JValue
  case Item(item) :: Nil XmlGet _ => item: Node
})
```

The above code will list all the items in response to `/simple5/item` and will serve a specific item in response to `/simple5/item/1234`, as we see in:

```
dpp@raptor:~/proj/simply_lift/samples/http_rest$ curl http://localhost:8080/simple5/item
[{
  "id":"1234",
  "name":"Cat Food",
  "description":"Yummy, tasty cat food",
  "price":4.25,
  "taxable":true,
  "weightInGrams":1000,
  "qnty":4
},
...
,{
  "id":"1237",
  "name":"Sloth Food",
  "description":"Slow, slow sloth food",
  "price":18.33,
  "taxable":true,
  "weightInGrams":750,
  "qnty":62
}]

dpp@raptor:~/proj/simply_lift/samples/http_rest$ curl http://localhost:8080/simple5/item/1
{
  "id":"1237",
  "name":"Sloth Food",
  "description":"Slow, slow sloth food",
  "price":18.33,
  "taxable":true,
  "weightInGrams":750,
  "qnty":62
}
```

In the above examples, we've explicitly coersed the results into a `JValue` or `Node` depending on the request type. With Lift, it's possible to define a conversion from a given type to response types (the default response types are JSON and XML) based on the request type and then define the request patterns to match and `RestHelper` takes care of the rest (so to speak.) Let's define the conversion from `Item` to `JValue` and `Node` (note the `implicit` keyword, that says that the conversion is available to `serveJx` statements:

```
  implicit def itemToResponseByAccepts: JxCvtPF[Item] = {
    case (JsonSelect, c, _) => c: JValue
    case (XmlSelect, c, _) => c: Node
  }
```

This is pretty straight forward. If it's a `JsonSelect`, return a `JValue` and if it's an `XmlSelect`, convert to a `Node`.

This is used in the `serveJx` statement:

```
  serveJx[Item] {
    case "simple6" :: "item" :: Item(item) :: Nil Get _ => item
    case "simple6" :: "item" :: "other" :: item :: Nil Get _ =>
      Item.find(item) ?~ "The item you're looking for isn't here"
  }
```

So /simple6/item/1234 will match and result in an Item being returned and based on the above implicit conversion, we turn the Item into a JValue or Node depending on the Accepts header and then convert that to a () => Box[LiftResponse]. Let's see what curl has to say about it:

```
dpp@raptor:~/proj/simply_lift/samples/http_rest$ curl http://localhost:8080/simple6/item/12
{
  "id":"1237",
  "name":"Sloth Food",
  "description":"Slow, slow sloth food",
  "price":18.33,
  "taxable":true,
  "weightInGrams":750,
  "qnty":62
}

dpp@raptor:~/proj/simply_lift/samples/http_rest$ curl -H "Accept: application/xml" http://l
<?xml version="1.0" encoding="UTF-8"?>
<item>
  <id>1234</id>
  <name>Cat Food</name>
  <description>Yummy, tasty cat food</description>
  <price>4.25</price>
  <taxable>true</taxable>
  <weightInGrams>1000</weightInGrams>
  <qnty>4</qnty>
</item>
```

Note also that /simple6/item/other/1234 does the right thing. This is because the path is 4 elements long, so it won't match the first part of the pattern, but does match the second part of the pattern.

Finally, let's combine serveJx and it's DRY helper, prefixJx.

```
  serveJx[Item] {
    "simple7" / "item" prefixJx {
      case Item(item) :: Nil Get _ => item
      case "other" :: item :: Nil Get _ =>
        Item.find(item) ?~ "The item you're looking for isn't here"
    }
  }
```

5.4 A complete REST example

The above code gives us the bits and pieces that we can combine into a full fledged REST service. Let's do that combination and see what such a service looks like:

Listing 5.4: FullRest.scala

```
package code
package lib
```

```

import model._

import net.liftweb._
import common._
import http._
import rest._
import util._
import Helpers._
import json._
import scala.xml._

/**
 * A full REST example
 */
object FullRest extends RestHelper {

  // Serve /api/item and friends
  serve( "api" / "item" prefix {

    // /api/item returns all the items
    case Nil JsonGet _ => Item.inventoryItems: JValue

    // /api/item/count gets the item count
    case "count" :: Nil JsonGet _ => JInt(Item.inventoryItems.length)

    // /api/item/item_id gets the specified item (or a 404)
    case Item(item) :: Nil JsonGet _ => item: JValue

    // /api/item/search/foo or /api/item/search?q=foo
    case "search" :: q JsonGet _ =>
      (for {
        searchString <- q ::: S.params("q")
        item <- Item.search(searchString)
      } yield item).distinct: JValue

    // DELETE the item in question
    case Item(item) :: Nil JsonDelete _ =>
      Item.delete(item.id).map(a => a: JValue)

    // PUT adds the item if the JSON is parsable
    case Nil JsonPut Item(item) -> _ => Item.add(item): JValue

    // POST if we find the item, merge the fields from the
    // the POST body and update the item
    case Item(item) :: Nil JsonPost json -> _ =>
      Item(mergeJson(item, json)).map(Item.add(_): JValue)

    // Wait for a change to the Items
    // But do it asynchronously
    case "change" :: Nil JsonGet _ =>
      RestContinuation.async {
        satisfyRequest => {
          // schedule a "Null" return if there's no other answer
          // after 110 seconds
          Schedule.schedule(() => satisfyRequest(JNull), 110 seconds)

          // register for an "onChange" event. When it
          // fires, return the changed item as a response
          Item.onChange(item => satisfyRequest(item: JValue))
        }
      }
  })
}
```

The whole service is JSON only and contained in a single `serve` block and uses the `prefix` helper to define all the requests under `/api/item` as part of the service.

The first couple of patterns are a re-hash of what we've already covered:

```
    // /api/item returns all the items
    case Nil JsonGet _ => Item.inventoryItems: JValue

    // /api/item/count gets the item count
    case "count" :: Nil JsonGet _ => JInt(Item.inventoryItems.length)

    // /api/item/item_id gets the specified item (or a 404)
    case Item(item) :: Nil JsonGet _ => item: JValue
```

The next is a search feature at `/api/item/search`. Using a little Scala library fun, we create a list of the request path elements that come after the `search` element and all the query parameters named `q`. Based on these, we search for all the `Items` that match the search term. We wind up with a `List[Item]` and we remove duplicates with `distinct` and finally coerse the `List[Item]` to a `JValue`:

```
    // /api/item/search/foo or /api/item/search?q=foo
    case "search" :: q JsonGet _ =>
      (for {
        searchString <- q ::: S.params("q")
        item <- Item.search(searchString)
      } yield item).distinct: JValue
```

Next, let's see how to delete an `Item`:

```
    // DELETE the item in question
    case Item(item) :: Nil JsonDelete _ =>
      Item.delete(item.id).map(a => a: JValue)
```

The only real difference is we're looking for a `JsonDelete` HTTP request.

Let's see how we add an `Item` with a PUT:

```
    // PUT adds the item if the JSON is parsable
    case Nil JsonPut Item(item) -> _ => Item.add(item): JValue
```

Note the `Item(item) -> _` after `JsonPut`. The extraction signature for `JsonPut` is `(List[String], (JValue, Req))`. The `List[String]` part is simple... it's a `List` that contains the request path. The second part of the Pair is a Pair itself that contains the `JValue` and the underlying `Req` (in case you need to do something with the request itself). Because there's a `def unapply(in: JValue): Option[Item]` method in the `Item` singleton, we can extract (pattern match) the `JValue` that is built from the PUT request body. This means if the user PUTs a JSON blob that can be turned into an `Item` the pattern will match and we'll evaluate the right hand side of the case statement which adds the `Item` to inventory. That's a big ole dense pile of information. So, we'll try it again with POST.

```
  case Item(item) :: Nil JsonPost json -> _ =>
   Item(mergeJson(item, json)).map(Item.add(_): JValue)
```

In this case, we're match a POST on `/api/item/1234` that has some parsable JSON in the POST body. The `mergeJson` method takes all the fields in the found `Item` and replaces them with any of the fields in the JSON in the POST body. So a POST body of `{"qnty": 123}` would replace the `qnty` field in the `Item`. The `Item` is then added back into the backing store.

Cool. So, we've got a variety of GET support in our REST service, a DELETE, PUT and POST. All using the patterns that `RestHelper` gives us.

Now we have some fun.

One of the features of Lift's HTML side is support for Comet (server push via long-polling.) If the web container supports it, Lift will automatically use asynchronous support. That means that during a long poll, while no computations are being performed related to the servicing of the request, no threads will be consumed. This allows lots and lots of open long polling clients. Lift's REST support includes asynchronous support. In this case, we'll demonstrate opening an HTTP request to `/api/item/change` and wait for a change to the backing store. The request will be satisfied with a change to the backing store or a JSON JNull after 110 seconds:

```
  case "change" :: Nil JsonGet _ =>
   RestContinuation.async {
    satisfyRequest => {
     // schedule a "Null" return if there's no other answer
     // after 110 seconds
     Schedule.schedule(() => satisfyRequest(JNull), 110 seconds)

     // register for an "onChange" event. When it
     // fires, return the changed item as a response
     Item.onChange(item => satisfyRequest(item: JValue))
    }
   }
```

If we receive a GET request to `/api/item/change`, invoke `RestContinuation.async`. We pass a closure that sets up the call. We set up the call by scheduling a `JNull` to be sent after 110 seconds. We also register a function which is invoked when the backing store is changed. When either event (110 seconds elapses or the backing store changes), the functions will be invoked and they will apply the `satifyRequest` function which will invoke the continuation and send the response back to the client. Using this mechanism, you can create long polling services that do not consume threads on the server. Note too that the `satisfyRequest` function is fire-once so you can call it lots of times, but only the first time counts.

5.5 Wrap Up

In this chapter, we've covered how you create web services in Lift. While there is a lot of implicit conversion stuff going on under the covers in `RestHelper`, the resulting code is pretty easy to read, create, and maintain. At the core, you match an incoming request against a pattern, if the pattern matches, evaluate the expression on the right hand side of the pattern.

Chapter 6

Wiring

Interactive web applications have many interdependent components on a single web page. For example (and this is the example we'll use for this chapter), you may have a shopping cart in your application. The shopping cart will contain items and quantities. As you add/remove items from the cart, the cart should update, along with the sub-total, the tax, the shipping and the grand total. Plus, the count of the items in the cart may be displayed on some pages without the cart contents. Keeping track of all of these dependencies for all the different page layouts is pretty tough work. When it comes to updating the site, the team must remember where all of the items are and how to update them and if they get one wrong, the site looks broken.

Lift's Wiring provides a simple solution to managing complex dependencies on a single page and on multiple tabs. Lift's Wiring allows you to declare the formulaic relationships among cells (like a spreadsheet) and then the user interface components (yes, there can be more than one component) associated with each cell. Lift will automatically update the dependent user interface components based on change in the predicates. Lift will do this on initial page render and with each Ajax or Comet update to the page. Put another way, Wiring is like a spreadsheet and the page will automatically get updated when any of the predicate values change such that the change results in a change in the display value.

6.1 Cells

Like a spreadsheet, Lift's Wiring is based on Cells. Cells come in three types: `ValueCell`, `DynamicCell`, and `FuncCell`.

A `ValueCell` contains a value that is entered by a user or depends on some user action. A `ValueCell` may represent the items in our shopping cart or the tax rate.

A `DynamicCell` contains a value that changes every time the cell is accessed. For example, a random number or the current time.

A `FuncCell` has a value based on a formula applied to the value or other cells.

Let's see some code that demonstrates this:

```scala
val quantity = ValueCell(0)
val price = ValueCell(1d)
val total = price.lift(_ * quantity)
```

We define two `ValueCells`, one for `quantity` and the other for `price`. Next, define the `total` by "lifting" the `price` in a formula that multiplies it by `quantity`. Let's see how it works in the console:

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

scala> import util._
import util._

scala> val quantity = ValueCell(0)
quantity: net.liftweb.util.ValueCell[Int] = ValueCell(0)

scala> val price = ValueCell(0d)
price: net.liftweb.util.ValueCell[Double] = ValueCell(0.0)

scala> val total = price.lift(_ * quantity)
total: net.liftweb.util.Cell[Double] = FuncCell1(ValueCell(0.0),<function1>)

scala> total.get
res1: Double = 0.0

scala> quantity.set(10)
res2: Int = 10

scala> price.set(0.5d)
res3: Double = 0.5

scala> total.get
res4: Double = 5.0
```

Okay... pretty nifty... we can define relationships that are arbitrarily complex between `Cells` and they know how to calculate themselves.

6.2 Hooking it up to the UI

Now that we can declare relationships among cells, how do we associate the value of `Cells` with the user interface?

Turns out that it's pretty simple:

```
 "#total" #> WiringUI.asText(total)
```

We associate the element with `id="total"` with a function that displays the value in `total`. Here's the method definition:

```
  /**
   * Given a Cell register the
   * postPageJavaScript that will update the element with
   * a new value.
   *
```

```
 * @param cell the cell to associate with
 *
 * @return a function that will mutate the NodeSeq (an id attribute may be added if
 * there's none already defined)
 */
def asText[T](cell: Cell[T]): NodeSeq => NodeSeq =
```

Huh? that's a lot of mumbo-jumbo... what's a `postPageJavaScript`?

So, here's the magic of `WiringUI`: Most web frameworks treat a page rendering as an event in time. Maybe (in the case of Seaside), there are some side effects of rendering that close over page rendering state such that when forms are submitted back, you get page state back. Lift treats a full HTML page render and subsequent Ajax requests on the page as a single event that has a single scope. This means that `RequestVar`s populated during a page render are available during subsequent Ajax requests on that page. Part of the state that results in a page render is the `postPageJavaScript` which is a bucket of `() => JsCmd` or a collection of functions that return JavaScript. Before responding to any HTTP request associated with the page, Lift runs all these functions and appends the resulting JavaScript to the response sent back to the browser. HTTP requests associated with the page include the initial page render, subsequent Ajax request associated with the page and associated Comet (long poll) requests generated by the page.

For each `Cell` that you wire up to the user interface, Lift captures the id of the DOM node (and if there's no id, Lift will assign one) and the current value of the `Cell`. Lift generates a function that looks at the current `Cell` value and if it's changed, Lift generates JavaScript that updates the DOM node with the `Cell`'s current value.

The result is that if an Ajax operation changes the value of a `ValueCell`, then all the dependent cells will update and the associated DOM updates will be carried back with the HTTP response.

You have a lot of control over the display of the value. The `asText` method creates a `Text(cell.toString)`. However, `WiringUI.apply` allows you to associate a function that converts the `Cell`'s type `T` to a `NodeSeq`. Further, you can control the transition in the browser with a `jsEffect` (type signiture `(String, Boolean, JsCmd) => JsCmd`). There are pre-build `jsEffects` based on jQuery including my favorite, `fade`:

```
/**
 * Fade out the old value and fade in the new value
 * using jQuery fast fade.
 */
def fade: (String, Boolean, JsCmd) => JsCmd = {
  (id: String, first: Boolean, cmd: JsCmd) => {
    if (first) cmd
    else {
      val sel = "jQuery('#'+"+id.encJs+")"
      Run(sel+".fadeOut('fast', function() {"+
          cmd.toJsCmd+" "+sel+".fadeIn('fast');})")
    }
  }
}
```

Which you can use as:

```
"#total" #> WiringUI.asText(total, JqWiringSupport.fade)
```

Now, when the total field updates, the old value will fade out and the new value will fade in... cool.

6.3 Shared Shopping

Let's move onto a real code example. You can find this code at Shop with Me source.

The example is going to be a simple shopping site. There are a bunch of items that you can view. You have a shopping cart. You can add items to the cart. If you're viewing the cart in multiple tabs or browser windows, the cart in all tabs/windows will update when you change the cart. Further, you can share your cart with someone else and any changes to the cart will be propagated to all the different browsers sharing the same cart.

The data model is the same that we used in the REST chapter (see 5.2 on page 46).

Let's look at the shopping cart definition:

Listing 6.1: Cart.scala

```scala
package code
package lib

import model.Item

import net.liftweb._
import util._

/**
 * The shopping cart
 */
class Cart {
  /**
   * The contents of the cart
   */
  val contents = ValueCell[Vector[CartItem]](Vector())

  /**
   * The subtotal
   */
  val subtotal = contents.lift(_.foldLeft(zero)(_ +
                                              _.qMult(_.price)))

  /**
   * The taxable subtotal
   */
  val taxableSubtotal = contents.lift(_.filter(_.taxable).
                                     foldLeft(zero)(_ +
                                                   _.qMult(_.price)))

  /**
   * The current tax rate
   */
  val taxRate = ValueCell(BigDecimal("0.07"))
```

```

/**
 * The computed tax
 */
val tax = taxableSubtotal.lift(taxRate)(_ * _)

/**
 * The total
 */
val total = subtotal.lift(tax)(_ + _)

/**
 * The weight of the cart
 */
val weight = contents.lift(_.foldLeft(zero)(_ +
                                  _.qMult(_.weightInGrams)))

// Helper methods

/**
 * A nice constant zero
 */
def zero = BigDecimal(0)

/**
 * Add an item to the cart. If it's already in the cart,
 * then increment the quantity
 */
def addItem(item: Item) {
  contents.atomicUpdate(v => v.find(_.item == item) match {
    case Some(ci) => v.map(ci => ci.copy(qnty = ci.qnty +
                                (if (ci.item == item) 1 else 0)))
    case _ => v :+ CartItem(item, 1)
  })
}

/**
 * Set the item quantity. If zero or negative, remove
 */
def setItemCnt(item: Item, qnty: Int) {
  if (qnty <= 0) removeItem(item)
  else contents.atomicUpdate(v => v.find(_.item == item) match {
    case Some(ci) => v.map(ci => ci.copy(qnty =
                                (if (ci.item == item) qnty
                                 else ci.qnty)))
    case _ => v :+ CartItem(item, qnty)
  })

}

/**
 * Removes an item from the cart
 */
def removeItem(item: Item) {
    contents.atomicUpdate(_.filterNot(_.item == item))
  }
}

/**
 * An item in the cart
 */
case class CartItem(item: Item, qnty: Int,
                    id: String = Helpers.nextFuncName) {

  /**
   * Multiply the quantity times some calculation on the
   * contained Item (e.g., getting its weight)
   */
  def qMult(f: Item => BigDecimal): BigDecimal = f(item) * qnty
}

/**
 * The CartItem companion object
 */
object CartItem {
  implicit def cartItemToItem(in: CartItem): Item = in.item
}
```

Looks pretty straight forward. You've got 2 `ValueCell`s, the cart contents and the tax rate. You've gota bunch of calculated `Cell`s. At the bottom of the `Cart` class definition are some helper methods that allow you to add, remove and update cart contents. We also define the `CartItem` case class that contains the `Item` and the `qnty` (quantity).

So far, so good. Next, let's look at the way we display all the items:

Listing 6.2: AllItemsPage.scala

```
package code
package snippet

import model.Item
import comet._

import net.liftweb._
import http._
import sitemap._
import util._
import Helpers._

object AllItemsPage {
  // define the menu item for the page that
  // will display all items
  lazy val menu = Menu.i("Items") / "item" >>
  Loc.Snippet("Items", render)

  // display the items
  def render =
    "tbody *" #> renderItems(Item.inventoryItems)

  // for a list of items, display those items
  def renderItems(in: Seq[Item]) =
    "tr" #> in.map(item => {
      "a *" #> item.name &
      "a [href]" #> AnItemPage.menu.calcHref(item) &
      "@description *" #> item.description &
      "@price *" #> item.price.toString &
      "@add_to_cart [onclick]" #>
      SHtml.ajaxInvoke(() => TheCart.addItem(item))})
}
```

We define our `SiteMap` entry:

```
  lazy val menu = Menu.i("Items") / "item" >>
    Loc.Snippet("Items", render)
```

So, when the user browses to `/item`, they're presented with all the items in inventory.

The template for displaying `Items` looks like:

Listing 6.3: items.html

```
<table class="lift:Items">
  <tbody>
    <tr>
      <td name="name"><a href="#">Name</a></td>
      <td name="description">Desc</td>
      <td name="price">$50.00</td>
      <td><button name="add_to_cart">Add to Cart</button></td>
    </tr>
  </tbody>
</table>
```

Next, let's look at the code for displaying an `Item`:

Listing 6.4: AnItemPage.scala

```
package code
package snippet

import model.Item
import comet._

import net.liftweb._
import util._
import Helpers._
import http._
import sitemap._

import scala.xml.Text

object AnItemPage {
  // create a parameterized page
  def menu = Menu.param[Item]("Item", Loc.LinkText(i => Text(i.name)),
                          Item.find _, _.id) / "item" / *
}

class AnItemPage(item: Item) {
  def render = "@name *" #> item.name &
  "@description *" #> item.description &
  "@price *" #> item.price.toString &
  "@add_to_cart [onclick]" #> SHtml.ajaxInvoke(() => TheCart.addItem(item))
}
```

This defines what happens when the user goes to `/item/1234`. This is more “controller-like” than most of the other Lift code. Let’s look at the menu item definition:

```
  def menu = Menu.param[Item]("Item", Loc.LinkText(i => Text(i.name)),
                          Item.find _, _.id) / "item" / *
```

We are defining a parameterized `Menu` entry. The parameter type is `Item`. That means that the page will display an `Item` and that we must be able to calculate the `Item` based on the request.

`"Item"` is the name of the menu entry.

`Loc.LinkText(i => Text(i.name))` takes an item and generates the display text for the menu entry.

`Item.find _` is a function that takes a `String` and converts it to `Box[Item]`. It looks up the Item based on the parameter in the request that we’re interested in.

`_.id` is a function (`Item => String`) that takes an `Item` and returns a `String` that represents how to build a URL that represents the Item page. This is used by `"a [href]" #> AnItemPage.menu.calcHref(item)` to convert an `Item` to the HREF for the page that display the `Item`.

Finally, the URL is defined by `/ "item" / *` which is pretty much what it looks like. It’ll match an incoming request of the form `/item/xxx` and `xxx` is passed to the `String => Box[Item]` function to determine the `Item` associated with the URL.

So, we can display all the items. Navigate from all the items to a single item. Each item has a button that allows you to add the `Item` to the shopping cart. The `Item` is added to the cart with this code: `SHtml.ajaxInvoke(() => TheCart.addItem(item))})`. The `TheCart.addItem(item)` can be called from anywhere in the application without regard for what needs to be updated when the cart is changed.

Let’s look at how the cart is displayed and managed:

Listing 6.5: CometCart.scala

```
package code
package comet

import lib._

import net.liftweb._
import common._
import http._
import util._
import js._
import js.jquery._
import JsCmds._
import scala.xml.NodeSeq
import Helpers._

/**
 * What's the current cart for this session
 */
object TheCart extends SessionVar(new Cart())

/**
 * The CometCart is the CometActor the represents the shopping cart
 */
class CometCart extends CometActor {
  // our current cart
  private var cart = TheCart.get

  /**
   * Draw yourself
   */
  def render = {
    "#contents" #> (
      "tbody" #>
      Helpers.findOrCreateId(id => // make sure tbody has an id
        // when the cart contents updates
        WiringUI.history(cart.contents) {
          (old, nw, ns) => {
            // capture the tr part of the template
            val theTR = ("tr ^^" #> "**")(ns)

            def ciToId(ci: CartItem): String = ci.id + "_" + ci.qnty

            // build a row out of a cart item
            def html(ci: CartItem): NodeSeq = {
              ("tr [id]" #> ciToId(ci) &
               "@name *" #> ci.name &
               "@qnty *" #> SHtml.
               ajaxText(ci.qnty.toString,
                        s => {
                          TheCart.
                          setItemCnt(ci,
                                     Helpers.toInt(s))
                        }, "style" -> "width: 20px;") &
               "@del [onclick]" #> SHtml.
             ajaxInvoke(() => TheCart.removeItem(ci)))(theTR)
            }

            // calculate the delta between the lists and
            // based on the deltas, emit the current jQuery
            // stuff to update the display
            JqWiringSupport.calculateDeltas(old, nw, id)(ciToId _, html _)
          }
      })) &
    "#subtotal" #> WiringUI.asText(cart.subtotal) & // display the subttotal
    "#tax" #> WiringUI.asText(cart.tax) & // display the tax
    "#total" #> WiringUI.asText(cart.total) // display the total
  }

  /**
   * Process messages from external sources
   */
  override def lowPriority = {
    // if someone sends us a new cart
    case SetNewCart(newCart) => {
      // unregister from the old cart
      unregisterFromAllDepenencies()

      // remove all the dependencies for the old cart
      // from the postPageJavaScript
      theSession.clearPostPageJavaScriptForThisPage()

      // set the new cart
      cart = newCart

      // do a full reRender including the fixed render piece
      reRender(true)
    }
  }
}

/**
 * Set a new cart for the CometCart
 */
case class SetNewCart(cart: Cart)
```

Let's walk through the code:

```
object TheCart extends SessionVar(new Cart())
```

We define a `SessionVar` that holds the shopping cart.

Our `CometActor` captures the the current cart from the `SessionVar`:

```
class CometCart extends CometActor {
  // our current cart
  private var cart = TheCart.get
```

Next, let's see how to draw the `cart.total`:

```
"#total" #> WiringUI.asText(cart.total) // display the total
```

That's pretty much the way it should be.

Let's look at the gnarly piece... how to draw or redraw the cart contents based on changes and only send the JavaScript the will manipulate the browser DOM to add or remove items from the cart:

```
"#contents" #> (
     "tbody" #>
     Helpers.findOrCreateId(id => // make sure tbody has an id
       // when the cart contents updates
       WiringUI.history(cart.contents) {
         (old, nw, ns) => {
           // capture the tr part of the template
           val theTR = ("tr ^^" #> "**")(ns)

           def ciToId(ci: CartItem): String = ci.id + "_" + ci.qnty

           // build a row out of a cart item
           def html(ci: CartItem): NodeSeq = {
             ("tr [id]" #> ciToId(ci) &
              "@name *" #> ci.name &
              "@qnty *" #> SHtml.
              ajaxText(ci.qnty.toString,
                      s => {
                        TheCart.
                        setItemCnt(ci,
                                   Helpers.toInt(s))
                      }, "style" -> "width: 20px;") &
              "@del [onclick]" #> SHtml.
            ajaxInvoke(() => TheCart.removeItem(ci)))(theTR)
           }

           // calculate the delta between the lists and
           // based on the deltas, emit the current jQuery
           // stuff to update the display
           JqWiringSupport.calculateDeltas(old, nw, id)(ciToId _, html _)
         }
       }))
```

First, we make sure we know the `id` of the `<tbody>` element: `"tbody" #> Helpers.findOrCreateId(id =>`

Next, wire the `CometCart` up to the `cart.contents` such that when the contents change, we get the old value (`old`), the new value (`nw`) and the memoized `NodeSeq` (the template used to do the rendering): `WiringUI.history(cart.contents) { (old, nw, ns) => {`

Capture the part of the template associated with the `<tr>` element in the `theTR` variable: `val theTR = ("tr ^^" #> "**")(ns)`

Based on a `CartItem`, return a stable id for the DOM node the represents the `CartItem`:

The `html` method converts a `CartItem` to a `NodeSeq` including Ajax controls for changing quantity and removing the item from the cart.

Finally, based on the deltas between the old list of `CartItem` and the new list, generate the JavaScript that will manipulate the DOM by inserting and removing the appropriate DOM elements: `JqWiringSupport.calculateDeltas(old, nw, id)(ciToId _, html _)`

Next, let's see how to change the cart. If we want to share the shopping cart between two browser sessions... two people shopping at their browser, but putting things in a single cart, we need a way to change the cart. We process the `SetNewCart` message to `CometCart`:

```scala
  // if someone sends us a new cart
  case SetNewCart(newCart) => {
    // unregister from the old cart
    unregisterFromAllDepenencies()

    // remove all the dependencies for the old cart
    // from the postPageJavaScript
    theSession.clearPostPageJavaScriptForThisPage()

    // set the new cart
    cart = newCart

    // do a full reRender including the fixed render piece
    reRender(true)
  }
```

There are two lines in the above code that hint at how Wiring interacts with Lift's Comet support: `unregisterFromAllDepenencies()` and `theSession.clearPostPageJavaScriptForThisPage()`

When a `CometActor` depends on something in `WiringUI`, Lift generates a weak reference between the `Cell` and the `CometActor`. When the `Cell` changes value, it pokes the `CometActor`. The `CometActor` then updates the browser's screen real estate associated with changes to `Cells`. `unregisterFromAllDepenencies()` disconnects the `CometActor` from the `Cells`. `theSession.clearPostPageJavaScriptForThisPage()` removes all the `postPageJavaScript` associated with the `CometActor`. Because the `CometActor` is not associated with a single page, but can appear on many pages, it has its own `postPageJavaScript` context.

The final piece of the puzzle is how we share a `Cart` across sessions. From the UI perspective, here's how we display the modal dialog when the user presses the "Share Cart" button:

Listing 6.6: Link.scala

```scala
package code
package snippet

import model._
import comet._
import lib._

import net.liftweb._
import http._
import util.Helpers._
import js._
import JsCmds._
import js.jquery.JqJsCmds._

class Link {
 // open a modal dialog based on the _share_link.html template
 def request = "* [onclick]" #> SHtml.ajaxInvoke(() => {
   (for {
      template <- TemplateFinder.findAnyTemplate(List("_share_link"))
    } yield ModalDialog(template)) openOr Noop

  })

  // close the modal dialog
  def close = "* [onclick]" #> SHtml.ajaxInvoke(() => Unblock)

  // Generate the href and link for sharing
  def generate = {
    val s = ShareCart.generateLink(TheCart)
    "a [href]" #> s & "a *" #> s
  }
}
```

Basically, we use jQuery's ModalDialog plugin to put a dialog up that contains a link generated by the `ShareCart` object. Let's look at ShareCart.scala:

Listing 6.7: ShareCart.scala

```
package code
package lib

import comet._

import net.liftweb._
import common._
import http._
import rest.RestHelper
import util._
import Helpers._

// it's a RestHelper
object ShareCart extends RestHelper {
  // private state
  private var carts: Map[String, (Long, Cart)] = Map()

  // given a Cart, generate a unique sharing code
  def codeForCart(cart: Cart): String = synchronized {
    val ret = Helpers.randomString(12)

    carts += ret -> (10.minutes.later.millis -> cart)

    ret
  }

  /**
   * Generate the right link to this cart
   */
  def generateLink(cart: Cart): String = {
    S.hostAndPath + "/co_shop/"+codeForCart(cart)
  }

  // An extractor that converts a String to a Cart, if
  // possible
  def unapply(code: String): Option[Cart] = synchronized {
    carts.get(code).map(_._2)
  }

  // remove any carts that are 10+ minutes old
  private def cleanup() {
    val now = Helpers.millis
    synchronized{
      carts = carts.filter{
        case (_, (time, _)) => time > now
      }
    }
    Schedule.schedule(() => cleanup(), 5 seconds)
  }

  // clean up every 5 seconds
  cleanup()

  // the REST part of the code
  serve {
    // match the incoming URL
    case "co_shop" :: ShareCart(cart) :: Nil Get _ => {
      // set the cart
      TheCart.set(cart)

      // send the SetNewCart message to the CometCart
      S.session.foreach(
        _.sendCometActorMessage("CometCart", Empty,
                                SetNewCart(cart)))

      // redirect the browser to /
      RedirectResponse("/")
    }
  }
}
```

The code manages the association between random IDs and Carts. If the user browses to /co_-shop/share_cart_id, ShareCart will set TheCart to the shared Cart and send a SetNew-Cart message to the CometCart instance associated with the session.

6.4 Wrap up

In this chapter we've seen how Lift's Wiring can be used to create complex inter-relationships among values and then surface those relationships in the web user interface. Wiring can be used with Ajax or Comet. Wiring makes it simple to build complex web pages that are user friendly and easy to maintain.

Chapter 7

Core Concepts

7.1 Snippets

Lift is built on the Scala programming language. Scala is a hybrid of Functional and Object Oriented. Two core principles of functional programming languages are immutability and transformation.

Immutability means that once a data structure is instantiated, it will not change for its life. More concretely, once you instantiate an object, you can freely pass the object around and the object will always return the same values for all its methods. Java's `String` class is immutable. Python requires immutable classes as indexes to dictionaries. Immutability is also very powerful for multithreaded applications because you can pass references to immutable objects across thread boundaries without having to worry about locking or synchronization because you are guaranteed that the objects will not change state.

7.1.1 Snippet `NodeSeq => NodeSeq`

Transformation provides an alternative to "writing to a stream" for composing web pages. Rather than having tags that cause characters to be streamed as part of the response, Lift loads the view and for each "snippet" encountered in the view, Lift transforms just the markup associated with the snippet invocation into a new set of HTML.

Let's make it more concrete, here's some markup:

```
<span class="foo lift:WhatTime">The time is <span id="current_time">currentTime</span></spa
```

And the associated snippet:

```
object WhatTime {
  def render = "#current_time" #> (new Date).toString
}
```

The resulting markup will look like:

```
<span class="foo">The time is Mon Dec 06 21:01:36 PST 2010</span>
```

Let's walk through how this works. First, the class attribute in the `<span>` has two classes, `foo` and `lift:WhatTime`. Any class attribute that starts with `lift:` indicates a snippet invocation. A snippet is a function that transforms HTML to HTML, or in Scala, `NodeSeq => NodeSeq`.

Lift looks up the snippet named `WhatTime` (See Section 23.1) which in this case resolves to a singleton and invokes the `render` method. The render method returns a `NodeSeq => NodeSeq` built using Lift's CSS Selector Transforms (See Section 7.10). The parameter to the function is the `Element` that caused the snippet invocation with the actual snippet invocation removed from the `class` attribute:

```
<span class="foo">The time is <span id="current_time">currentTime</span></span>
```

The function is then applied and the resulting `NodeSeq` is inserted in the page where the original `Element` was. Because the page is composed of immutable XML objects, we can transform

`NodeSeq => NodeSeq` and not worry about anything getting changed out from under us. We also know that we've got valid markup through the entire page transformation process.

Further, retaining the page as a well formed XML document allows certain tags to be put in the `<head>` tag and other tags to be inserted just before the close of the `</body>` tag (See Section 7.17).

But the simplicity of the transform is simulateously easy to understand and very powerful.

7.1.2 Snippet instances

The snippet could also be defined as:

```
class WhatTime {
 private var x = 0

 def render = {
   x += 1
   "#current_time" #> ((new Date).toString + " and you've seen this message "+x+" times)
 }
}
```

7.1.3 Multiple methods on a snippet class

7.1.4 Inter-snippet communication

7.1.5 Recursive Snippets

7.1.6 Snippet parameters

7.2 Box/Option

Scala has a ton of nice features. One of the features that I was slow to adopt, until Burak Emir gently reminded me a bunch of times, is "Options". Read on about Options, Boxes, and how Lift makes good use of them to make clean, error resistant code. If you come from an imperative (Java, Ruby) background, you'll probably recognize the following code:

```
x = someOperation
if !x.nil?
  y = someOtherOperation
  if !y.nil?
    doSomething(x,y) return "it worked"
  end
end
return "it failed"
```

Okay, so that's pseudo-code, but there are tons of operation, guard, operation, guard, blah blah constructs.

Further, null/nil are passed around as failures. This is especially bad when it's null, but it's pretty bad when it's nil because it's not clear to the consumer of the API that there can be a "call failed" return value.

In Java, `null` is a non-object. It has no methods. It is the exception to the statically typed rule (`null` has no class, but any reference of any class can be set to `null`.) Invoking a method on `null` has one and only one result: an exception is thrown. `null` is often returned from methods as a flag indicating that the method ran successfully, but yielded no meaningful value. For example, `CardHolder.findByCreditCardNumber("2222222222")` In fact, the guy who invented `null` called it a billion dollar mistake.

Ruby has `nil` which is marginally better than `null`. `nil` is a real, singleton object. There's only one instance of `nil` in the whole system. It has methods. It is a subclass of `Object`. `Object` has a method called "`nil?`" which returns `false`, except the `nil` singleton overrides this method to return `true`. `nil` is returned much like `null` in Java. It's the "no valid answer" answer.

Scala does something different.

There's an abstract class, called `Option`. Options are strongly typed. They are declared `Option[T]`. This means an `Option` can be of any type, but once its type is defined, it does not change. There are two subclasses of `Option`: `Some` and `None`. `None` is a singleton (like `nil`). `Some` is a container around the actual answer. So, you might have a method that looks like:

```
def findUser(name: String): Option[User] = {
  val query = buildQuery(name)
  val resultSet = performQuery(query)
  val retVal = if (resultSet.next) Some(createUser(resultSet)) else None
  resultSet.close
  retVal
}
```

Some, you've got a `findUser` method that returns either `Some(User)` or `None`. So far, it doesn't look a lot different than our example above. So, to confuse everyone, I'm going to talk about collections for a minute.

A really nice thing in Scala (yes, Ruby has this too) is rich list operations. Rather than creating a counter and pulling list (array) elements out one by one, you write a little function and pass that function to the list. The list calls the function with each element and returns a new list with the values returned from each call. It's easier to see it in code:

```
scala> List(1,2,3).map(x => x * 2)
line0: scala.List[scala.Int] = List(2,4,6)
```

The above code multiplies each list item by two and "map" returns the resulting list. Oh, and you can be more terse, if you want:

```
scala> List(1,2,3).map(_ * 2)
line2: scala.List[scala.Int] = List(2,4,6)
```

You can nest map operations:

```
scala> List(1,2,3).map(x => List(4,5,6).map(y => x * y))
line13: scala.List[scala.List[scala.Int]] = List(List(4,5,6),List(8,10,12),List(12,15,18))
```

And, you can "flatten" the inner list:

```
scala> List(1,2,3).flatMap(x => List(4,5,6).map(y => x * y))
line14: scala.List[scala.Int] = List(4,5,6,8,10,12,12,15,18)
```

Finally, you can "filter" only the even numbers from the first list:

```
scala> List(1,2,3).filter(_ % 2 == 0). flatMap(x => List(4,5,6).map(y => x * y))
line16: scala.List[scala.Int] = List(8,10,12)
```

But, as you can see, the `map/flatMap/filter` stuff gets pretty verbose. Scala introduced a "for" comprehension to make the code more readable:

```
scala> for {
  x <- List(1,2,3) if x % 2 == 0
  y <- List(4,5,6)} yield x * y
res0: List[Int] = List(8, 10, 12)
```

Okay, but what does this have to do with `Option[T]`?

Turns out that `Option` implements `map`, `flatMap`, and `filter` (the methods necessary for the Scala compiler to use in the 'for' comprehension). Just as a side note, when I first encountered the phrase "'for' comprehension", I got scared. I've been doing programming for years and never heard of a "comprenhension" let alone a 'for' one. Turns out, that there's nothing fancy going on, but "'for' comprehension" is just a term of art for the above construct.

So, the cool thing is that you can use this construct very effectively. The first example is simple:

```
scala> for {x <- Some(3); y <- Some(4)} yield x * y
res1: Option[Int] = Some(12)
```

"That's nice, you just wrote a lot of code to multiply 3 by 4."

Let's see what happens if we have a "None" in there:

```
scala> val yOpt: Option[Int] = None
yOpt: Option[Int] = None
scala> for {x <- Some(3); y <- yOpt} yield x * y
res3: Option[Int] = None
```

So, we get a "None" back. How do we turn this into a default value?

```
scala> (for {x <- Some(3); y <- yOpt} yield x * y) getOrElse -1
res4: Int = -1
```

```
scala> (for {x <- Some(3); y <- Some(4)} yield x * y) getOrElse -1
res5: Int = 12
```

Note that the "getOrElse" code is "passed by name". Put another way, that code is only executed if the "else" clause is valid.

Lift has an analogous construct called Box.

A Box can be Full or not. A non-Full Box can be the Empty singleton or a Failure. A Failure carries around information about why the Box contains no value.

Failure is very helpful because you can carry around information to display an error... an HTTP response code, a message, what have you.

In Lift, I put this all together in the following way:

- methods that return request parameters return Box[String]
- finder methods on models (not find all, just the ones that return a single instance) return Box[Model]
- any method that would have returned a null if I was writing in Java returns a Box[T] in Lift

That means you get code that looks like:

```
scala> for {id <- S.param("id") ?~ "id param missing"
u <- getUser(id) ?~ "User not found"
} yield u.toXml
res6: net.liftweb.common.Box[scala.xml.Elem] = Failure(id param missing,Empty,Empty)
```

There's no explicit guard/test to see if the "id" parameter was passed in and there's no explicit test to see if the user was found.

Note also that this code is completely type-safe. While there was no explicit type declarations, the compiler was able to figure out what types the various objects were.

So, let's look at the code inside a REST handler:

```
serve {
 case "user" :: "info" :: _ XmlGet _ =>
   for {
    id <- S.param("id") ?~ "id param missing" ~> 401
    u <- User.find(id) ?~ "User not found"
  } yield u.toXml
}
```

If the id parameter is missing, present a nice error message and return a 401 (okay... this is random, but you get the point). And by default, if the user isn't found, return a 404 with the error that the user isn't found.

Here's what it looks like using wget:

```
dpp@bison:~/lift_sbt_prototype$ wget http://localhost:8080/user/info.xml
--2010-06-01 15:07:27-- http://localhost:8080/user/info.xml
Resolving localhost... ::1, 127.0.0.1
Connecting to localhost|::1|:8080... connected.
HTTP request sent, awaiting response... 401 Unauthorized
Authorization failed.

dpp@bison:~/lift_sbt_prototype$ wget http://localhost:8080/user/info.xml?id=2
--2010-06-01 15:07:44-- http://localhost:8080/user/info.xml?id=2
Resolving localhost... ::1, 127.0.0.1
Connecting to localhost|::1|:8080... connected.
HTTP request sent, awaiting response... 404 Not Found
2010-06-01 15:07:44 ERROR 404: Not Found.

dpp@bison:~/lift_sbt_prototype$ wget http://localhost:8080/user/info.xml?id=1
--2010-06-01 15:24:12-- http://localhost:8080/user/info.xml?id=1
Resolving localhost... ::1, 127.0.0.1
Connecting to localhost|::1|:8080... connected.
HTTP request sent, awaiting response...
200 OK Length: 274 [text/xml] Saving to: `info.xml?id=1'

dpp@bison:~/lift_sbt_prototype$ cat info.xml\?id\=1
<?xml version="1.0" encoding="UTF-8"?>
 <User id="1" firstName="Elwood" ... validated="true" superUser="false">
</User>
```

One more thing about Box and Option... they lead to less complex, more maintainable code. Even if you didn't know anything about Scala or Lift, you can read the XML serving code and the console exchange and figure out what happened any why it happened. This is a lot more readable than deeply nested if statements. And if it's readable, it's maintainable.

I hope this is an understandable introduction to Scala's Option class and 'for' comprehension and how Lift makes use of these tools.

7.3 S/SHtml

7.4 Boot

7.5 SiteMap

7.6 GUIDs

A core concept in Lift is GUIDs. GUIDs are globally unique identifiers used to associate something in the browser with a function on the server. GUIDs make Lift more secure because they make replay attacks very difficult and GUIDs make it easier to develop complex, stateful, interactive applications because the developer spends more time on business logic and less time on the plumbing of it.

7.6.1 How GUIDs are generated

7.6.2 Where they are used

7.7 LiftRules

7.8 SessionVars and RequestVars

7.9 Helpers

7.10 CSS Selector Transforms

Lift 2.2-M1 introduced a new mechanism for transforming XHTML: CSS Selector Transforms (`CssBindFunc`). The new mechanism provides a subset of CSS selectors that can be used to transform `NodeSeq => NodeSeq`. Examples of this feature include:

- `"#name" #> userName` // replace the element with the id name with the variable userName
- `"#chat_lines *" #> listOfChats` // replace the content of chat_lines with each element of listOfChats
- `".pretty *" #> <b>Unicorn</b>` // each element with CSS class pretty, replace content with <b>Unicorn</b>
- `"dog=cat [href]" #> "http://dogscape.com"` // set the href attribute of all elements with the dog attribute set to cat
- `"#name" #> userName & "#age" #> userAge` // set name to userName and age to userAge
- `"li *" #> userName & "li [class]" #> "foo"` // set the contents of all <li> element with username and class to foo
- `"li *" #> userName & "li [class+]" #> "foo"` // set the contents of all <li> element with username and append foo to the class
- `"*" #> <span>{userName}</span>` // set all the elements to <span>{userName}</span>

CSS Selector Transforms extends `NodeSeq => NodeSeq`... they are quite literally functions and can be passes as a parameter to anything expecting `NodeSeq => NodeSeq` or returned as a result for any method that returns `NodeSeq => NodeSeq`.

Let's look at each of the pieces to see how they work.

First, you must `import net.liftweb.util._` and `import Helpers._` These packages include the classes and the implicit conversions that make the CSS Selector Tranforms work.

The transform is defined by: String representing selector #> transform value.

The selector is a String constant which implements the following subset of CSS Selectors:

- `#id` - selects the element with the specified id
- `.class` - selects all elements have a class attribute where one of the space-separated values equals class
- `attr_name=attr_value` - selects all elements where the given attribute equals the given value
- `element_name` - selects all the elements matching the name
- `*` - selects all elements

- `@name` - selects all elements with `the specified name`
- `:button` - selects all the elements with `type="button"`
- `:checkbox` - selects all the elements with `type="checkbox"`
- `:file` - selects all the elements with `type="file"`
- `:password` - selects all the elements with `type="password"`
- `:radio` - selects all the elements with `type="radio"`
- `:reset` - selects all the elements with `type="reset"`
- `:submit` - selects all the elements with `type="submit"`
- `:text` - selects all the elements with `type="text"`

You can put replacement rules after the selector:

- none (e.g., `"#id"`) replaces all matching elements with the values
  `"#name" #> "David" // <span><span id="name"/></span> -> <span>David</span>`
- * (e.g., `"#id *"`) replaces the content children of the matching elements with the values
  `"#name *" #> "David" // <span><span id="name"/></span> -> <span><span id="name>David</span></span>`
- *+ or `*<` (e.g., `"#id *+"`) appends the value to the the content children nodes
  `"#name *+" #> "David" // <span><span id="name">Name:  </s-pan></span> -> <span><span id="name>Name:  David</span></span>`
- `-*` or `>*` (e.g., `"#id -*"`) prepends the value to the the content children nodes
  `"#name -*" #> "David" // <span><span id="name"> Pol-lak</span></span> -> <span><span id="name>David Pol-lak</span></span>`
- `[attr]` (e.g., "#id `[href]`") replaces the matching attribute's value with the values.
  `"#link [href]" #> "http://dogscape.com"`
  `// <a href="#" id="link">Dogscape</a> -> <a href="http://dogscape.com" id="link">Dogscape</a>`
- `[attr+]` (e.g., "#id `[class+]`") appends the value to the existing attribute.
  `"span [class+]" #> "error"`
  `// <span class"foo">Dogscape</span> -> <span class"foo er-ror">Dogscape</span>`
- `[attr!]` (e.g., "#id `[class!]`") removes the matching value to the existing from.
  `"span [class!]" #> "error"`
  `// <span class"error foo">Dogscape</span> -> <span class"foo">Dogscape</span>`

- `^^` - lifts the selected element to the root of the elements that are returned making it possible to choose an element from a template
- `^*` - lifts the selected element's children to the root of the elements that are returned making it possible to choose an element's children from a template

The right hand side of the CSS Selector Transform can be one of the following:

- `String` – a `String` constant, for example:
  "#name *" #> "David" // <span id="name"/> -> <span id="name">David</span>
  "#name *" #> getUserNameAsString
- `NodeSeq` - a `NodeSeq` constant, for example:
  "#name *" #> <i>David</i> // <span id="name"/> -> <span id="name"><i>David</i></span>
  "#name *" #> getUserNameAsHtml
- `NodeSeq => NodeSeq` – a function that transforms the node (yes, it can be a CssBindFunc):
  `"#name" #> ((n: NodeSeq) => n % ("class" -> "dog"))` // <span id="name"/> -> <span id="name" class="dog"/>
- `Bindable` – something that implements the `Bindable` trait (e.g., `MappedField` and `Record.Field`)
- `StringPromotable` – A constant that can be promoted to a `String` (`Int`, `Symbol`, `Long` or `Boolean`). There is an automatic (implicit) conversion from Int, Symbol, Long or Boolean to StringPromotable.
  `"#id_like_cats" #> true & "#number_of_cats" #> 2`
- `IterableConst` – A `Box`, `Seq`, or `Option` of `NodeSeq => NodeSeq`, `String`, `NodeSeq`, or `Bindable`. Implicit conversions automatically promote the likes of `Box[String]`, `List[String]`, `List[NodeSeq]`, etc. to `IterableConst`.
  `"#id" #> (Empty: Box[String]) // <span><span id="id">Hi</span></span> -> <span/>`
  `"#id" #> List("a", "b", "c") // <span><span id="id"/></span> -> <span>abc</span>`
  `"#id [href]" #> (None: Option[String]) <a id="id" href="dog"/> -> <a id="id"/>`

Note that if you bind to the children of a selected element, multiple copies of the element result from bind to an `IterableConst` (if the element has an id attribute, the id attribute will be stripped after the first element):

```
"#line *" #> List("a", "b", "c") // <li id="line>sample</li> ->
                            // <li id="line">a</li><li>b</li><li>c</li>

"#age *" #> (None: Option[NodeSeq]) // <span><span id="age">Dunno</span></span> ->
                                  // <span/>
```

The above use cases may seem a little strange (they are not quite orthogonal), but they address common use cases in Lift. * IterableFunc – A Box, Seq, or Option of functions that transform NodeSeq => String, NodeSeq, Seq[String], Seq[NodeSeq], Box[String], Box[NodeSeq], Option[String] or Option[NodeSeq]. The same rules for handling multiple values in IterableConst apply to IterableFunc. Implicit conversions automatically promote the functions with the appropriate signature to an IterableFunc.

You can chain CSS Selector Transforms with the `&` method:

```
"#id" #> "33" & "#name" #> "David" & "#chat_line" #> List("a", "b",
"c") & ClearClearable
```

CSS Selector Transforms offer an alternative to Lift's traditional binding (See `Helpers.bind()`).

7.11 Client-side behavior invoking server-side functions

7.12 Ajax

7.13 Comet

7.14 LiftActor

7.15 Pattern Matching

7.16 Type safety

7.17 Page rewriting

7.18 Security

Chapter 8

Common Patterns

8.1 Localization

Lift has broad support for localization at the page and element level.

8.1.1 Localizing Templates

The locale for the current request is calculated based on the function in `LiftRules.localeCalculator`. By default, the function looks at the Locale in the HTTP request. But you can change this function to look at the Locale for the current user by changing `LiftRules.localeCalculator`.

When a template is requested, Lift's `TemplateFinder` looks for a template with the suffix `_langCOUNTRY.html`, then `_lang.html`, then `.html`. So, if you're loading `/frog` and your Locale is `enUS`, then Lift will look for `/frog_enUS.html`, then `/frog_en.html`, then `/frog.html`. But if your Locale is Czech, then Lift would look for `/frog_csCZ.html`, `/frog_cs.html`, and `/frog.html`. The same lookup mechanism is used for templates accessed via the Surround (See Section 9.14) and Embed (See Section 9.13) snippets. So, at the template level, Lift offers very flexible templating.

Note: Lift parses all templates in UTF-8. Please make sure your text editor is set to UTF-8 encoding.

8.1.2 Resource Lookup

Lift uses the following mechanism to look up resources. Localized resources are stored in template files along-side your HTML pages. The same parser is used to load resources and the pages themselves. A global set of resources is searched for in the following files: `/_resources.html`, `/templates-hidden/_resources.html`, and `/resources-hidden/_resources.html`. Keep in mind that Lift will look for the _resources file using the suffixes based on the Locale.

The resource file should be in the following format:

```
<resources>
  <res name="welcome">Benvenuto</res>
  <res name="thank.you">Grazie</res>
  <res name="locale">Località</res>
  <res name="change">Cambia</res>
</resources>
```

In addition to global resource files, there are per-page resource files (based on the current `Req`.) If you are currently requesting page `/foo/bar`, the following resource files will also be consulted: `/foo/_resources_bar.html`, `/templates-hidden/foo/_resources_bar.html`, and `/foo/resources-hidden/_resources_bar.html` (and all Locale-specific suffixes.) You can choose to create a separate resource file for each locale, or lump multiple locales into the `_resources_bar.html` file itself using the following format:

```
<resources>
  <res name="hello" lang="en" default="true">Hello</res>
  <res name="hello" lang="en" country="US">Howdy, dude!</res>
  <res name="hello" lang="it">Benvenuto</res>
  <res name="thank.you" lang="en" default="true">Thank You</res>
  <res name="thank.you" lang="it">Grazie</res>
  <res name="locale" lang="en" default="true">Locale</res>
  <res name="locale" lang="it">Località</res>
  <res name="change" lang="en" default="true">Change</res>
  <res name="change" lang="it">Cambia</res>
</resources>
```

8.1.3 Accessing Resources

Lift makes it easy to access resources.

From snippets: `<span class="lift:Loc.hello">This Hello will be replaced if possible</span>` Note that the value after the . in the snippet invocation is used to look up the resource name.

From code:

- `S.loc("hello")` - return a `Box[NodeSeq]` containing the localized value for the resource named “hello”.
- `S.??("Hello World")` - look for a resource named “Hello World” and return the String value for that resource. If the resource is not found, return “Hello World”.

8.1.4 Conclusion

Lift offers a broad range of mechanisms for localizing your application on a page-by-page and resource-by-resource by-resource basis.

8.2 Dependency Injection

Dependency injection is an important topic in the Java world. It's important because Java lacks certain basic features (e.g., functions) that tend to bind abstract interfaces to concrete implementations. Basically, it's so much easier to do `MyInterface thing = new MyInterfaceImpl()`, so most developers do just that.

Scala's cake pattern goes a long way to help developers compose complex behaviors by combining Scala traits. Jonas Bonér wrote an excellent piece on Dependency Injection.

The cake pattern only goes half way to giving a Java developer complete dependency injection functionality. The cake pattern allows you to compose the complex classes out of Scala traits, but the cake pattern is less helpful in terms of allowing you to make dynamic choices about which combination of cake to vend in a given situation. Lift provides extra features that complete the dependency injection puzzle.

8.2.1 Lift Libraries and Injector

Lift is both a web framework and a set of Scala libraries. Lift's `common`, `actor`, `json`, and `util` packages provide common libraries for Scala developers to build their application. Lift's libraries are well tested, widely used, well supported, and released on a well defined schedule (montly milestones, quarterly releases).

Lift's `Injector` trait forms the basis of dependency injection:

```
/**
 * A trait that does basic dependency injection.
 */
trait Injector {
  implicit def inject[T](implicit man: Manifest[T]): Box[T]
}
```

You can use this trait as follows:

```
object MyInjector extends Injector {...}

val myThing: Box[Thing] = MyInjector.inject
```

The reason that the instance of `MyThing` is in a `Box` is because we're not guaranteed that `MyInjector` knows how to create an instance of `Thing`. Lift provides an implementation of `Injector` called `SimpleInjector` that allows you to register (and re-register) functions for injection:

```
object MyInjector extends SimpleInjector

def buildOne(): Thing = if (testMode) new Thing with TestThingy {} else new Thing with Runt

MyInjector.registerInjection(buildOne _) // register the function that builds Thing

val myThing: Box[Thing] = MyInjector.inject
```

This isn't bad... it allows us to define a function that makes the injection-time decision, and we can change the function out during runtime (or test-time.) However, there are two problems: getting Boxes for each injection is less than optimal. Further, globally scoped functions mean you have to put a whole bunch of logic (test vs. production vs. xxx) into the function. `SimpleInjector` has lots of ways to help out.

```
object MyInjector extends SimpleInjector {
  val thing = new Inject(buildOne _) {} // define a thing, has to be a val so it's eagerly
}

def buildOne(): Thing = if (testMode) new Thing with TestThingy {} else new Thing with Runt

val myThingBox: Box[Thing] = MyInjector.injectval

myThing = MyInjector.thing.vend // vend an instance of Thing
```

`Inject` has a futher trick up its sleave... with `Inject`, you can scope the function... this is helpful for testing and if you need to change behavior for a particular call scope:

```
MyInjector.thing.doWith(new Thing with SpecialThing {}) {
  val t = MyInjector.thing.vend // an instance of SpecialThing
  val bt: Box[Thing] = MyInjector.inject // Full(SpecialThing)
}

MyInjector.thing.default.set(() => new Thing with YetAnotherThing {}) // set the global sco
```

Within the scope of the `doWith` call, `MyInjector.thing` will vend instances of `SpecialThing`. This is useful for testing as well as changing behavior within the scope of the call or globally. This gives us much of the functionality we get with dependency injection packages for Java. But within Lift WebKit, it gets better.

8.2.2 Lift WebKit and enhanced injection scoping

Lift's WebKit offers broad ranging tools for handling HTTP requests as well as HTML manipulation.

Lift WebKit's `Factory` extends `SimpleInjector`, but adds the ability to scope the function based on current HTTP request or the current container session:

```
object MyInjector extends Factory {
  val thing = new FactoryMaker(buildOne _) {} // define a thing, has to be a val so it's ea
                                      // evaluated and registered
}

MyInjector.thing.session.set(new Thing with ThingForSession {}) // set the instance that wi
                                                                // for the duration of the session

MyInjector.thing.request.set(new Thing with ThingForRequest {}) // set the instance that wi
                                                                // for the duration of the request
```

WebKit's `LiftRules` is a `Factory` and many of the properties that `LiftRules` contains are `FactoryMakers`. This means that you can change behavior during call scope (useful for testing):

```
LiftRules.convertToEntity.doWith(true) { ... test that we convert certain characters to ent
```

Or based on the current request (for example you can change the rules for calculating the docType during the current request):

```
if (isMobileReqest) LiftRules.docType.request.set((r: Req) => Full(DocType.xhtmlMobile))
```

Or based on the current session (for example, changing maxConcurrentRequests based on some rules when a session is created):

```
if (browserIsSomethingElse) LiftRules.maxConcurrentRequests.session.set((r: Req) => 32)
          // for this session, we allow 32 concurrent requests
```

8.2.3 Conclusion

Lift's `SimpleInjector/Factory` facilities provide a powerful and flexible mechanism for vending instances based on a global function, call stack scoping, request and session scoping and provides more flexible features than most Java-based dependency injection frameworks without resorting to XML for configuration or byte-code rewriting magic.

8.3 Modules

Lift has supported modules from the first version of the project in 2007. Lift's entire handling of the HTTP request/response cycle is open to hooks. Further, Lift's templating mechanism where resulting HTML pages are composed by transforming page content via snippets (See Section 7.1) which are simply functions that take HTML and return HTML: `NodeSeq => NodeSeq`. Because Lift's snippet resolution mechanism is open and any code referenced in Boot (See Section 7.4), any code can be a Lift "module" by virtue of registering its snippets and other resources in `LiftRules`. Many Lift modules already exist including PayPal, OAuth, OpenID, LDAP, and even a module containing many jQuery widgets.

The most difficult issue relating to integration of external modules into Lift is how to properly insert the module's menu items into a SiteMap (See Section 3.2) menu hierarchy. Lift 2.2 introduces a more flexible mechanism for mutating the `SiteMap`: `SiteMap` mutators. `SiteMap` mutators are functions that rewrite the SiteMap based on rules for where to insert the module's menus in the menu hierarchy. Each module may publish markers. For example, here are the markers for ProtoUser:

```
/**
* Insert this LocParam into your menu if you want the
* User's menu items to be inserted at the same level
* and after the item
*/
final case object AddUserMenusAfter extends Loc.LocParam[Any]
```

```
/**
* replace the menu that has this LocParam with the User's menu
* items
*/
final case object AddUserMenusHere extends Loc.LocParam[Any]
/**
* Insert this LocParam into your menu if you want the
* User's menu items to be children of that menu
*/
final case object AddUserMenusUnder extends Loc.LocParam[Any]
```

The module also makes a `SiteMap` mutator available, this can either be returned from the module's `init` method or via some other method on the module. ProtoUser makes the `sitemapMutator` method available which returns a `SiteMap => SiteMap`.

The application can add the marker to the appropriate menu item:

```
Menu("Home") / "index" >> User.AddUserMenusAfter
```

And when the application registers the `SiteMap` with `LiftRules`, it applies the mutator:

```
LiftRules.setSiteMapFunc(() => User.sitemapMutator(sitemap()))
```

Because the mutators are composable:

```
val allMutators = User.sitemapMutator andThen FruitBat.sitemapMutator
LiftRules.setSiteMapFunc(() => allMutators(sitemap()))
```

For each module, the implementation of the mutators is pretty simple:

```
 private lazy val AfterUnapply = SiteMap.buildMenuMatcher(_ == AddUserMenusAfter)
 private lazy val HereUnapply = SiteMap.buildMenuMatcher(_ == AddUserMenusHere)
 private lazy val UnderUnapply = SiteMap.buildMenuMatcher(_ == AddUserMenusUnder)

 /**
  * The SiteMap mutator function
  */
 def sitemapMutator: SiteMap => SiteMap = SiteMap.sitemapMutator {
  case AfterUnapply(menu) => menu :: sitemap
  case HereUnapply(_) => sitemap
  case UnderUnapply(menu) => List(menu.rebuild(_ ::: sitemap))
 }(SiteMap.addMenusAtEndMutator(sitemap))
```

We've defined some extractors that help with pattern matching. `SiteMap.buildMenuMatcher` is a helper method to make building the extractors super-simple. Then we supply a `PartialFunction[Menu, List[Menu]]` which looks for the marker `LocParam` and re-writes the menu based on the marker. If there are no matches, the additional rule is fired, in this case, we append the menus at the end of the `SiteMap`.

8.4 HtmlProperties, XHTML and HTML5

Lift unifies many aspects of parsing and displaying the HTML page in a single trait, `HtmlProperties`.

HtmlProperties defines, on a session-by-session (and even a request-by-request) basis, the way that templates are parsed and the way that Scala's `NodeSeq` is converted into valid HTML output. The properties on `HtmlProperties` are:

- `docType` - the DocType for the HTML page, e.g., `<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">` or `<!DOCTYPE html>`
- `encoding` - the page's encoding, e.g., `<?xml version="1.0" encoding="UTF-8"?>`
- contentType - the setting of the Content-Type response header, e.g., `application/xhtml+xml; charset=utf-8` or `text/html; charset=utf-8`
- `htmlOutputHeader` - calculates the way to combine the `docType` and `encoding` (this is important for IE6 support where encoding goes after docType).
- `htmlParser` - a function that converts an `InputStream` to a `Box[NodeSeq]`. This is used by Lift to parse templates.
- `htmlWriter` - a function that writes a `NodeSeq` to a `Writer`. This is used by Lift to convert the internal XML representation of a page to a stream of bytes representing an HTML page.
- `html5FormsSupport` - a flag indicating whether the current browser supports HTML5 forms.
- `maxOpenRequests` - the maximum number of concurrent HTTP requests the browser supports to a named host.
- `userAgent` - the User-Agent string sent from the browser.

8.4.1 XHTML via `OldHtmlProperties`

The default properties that keep compability with the disparate LiftRules used to calculate DocType and Encoding. Uses the `PCDataXmlParser` parser which requires well-formed XML files. Output is generally XHTML via `AltXML.toXML`, but cerain tags (e.g., `<br>`) are written in IE6/IE7 friendly ways.

8.4.2 HTML5 via `Html5Properties`

Prior to Lift 2.2, Lift always emitted XHTML and by default set the Content-Type header to `application/xhtml+xml; charset=utf-8`. This continues to be Lift's default behavior. It turns out that most browsers, even modern ones (Firefox, Chrome and Safari) had issues with XHTML. Further, XHTML limited the behavior of certain JavaScript libraries.

By invoking `LiftRules.htmlProperties.default.set((r: Req) => new Html5Properties(r.userAgent))` in Boot.scala, you can set Lift to full HTML5 support. Lift uses the nu.validator HTML parser and emits the correct DocType and response headers such that all tested browsers (IE6+, Firefox 2+, Safari 2+, Chrome 1+) render pages correctly.

Because the HTML5 parser is different from the standard XML parser, you will need to adjust your existing templates in the following ways:

- All tags are converted to lower case. This means the `<lift:FooBar/>` gets converted to `<lift:foobar/>` I advise converting to designer friendly where possible (e.g., `<div class="lift:FooBar"></div>`).
- Tags of the format `<div/>` and `<my_thing:bind/>` are not legal. They must be converted to `<div></div>` and `<my_thing:bind></my_thing:bind>`. Unfortunately, the parser is very forgiving so rather than barking about the lack of closing tag, the parser will nest things in unexpected ways.
- There are some tags that the parser "ensures". For example a `<tr>`, `<thead>`, or `<tbody>` tag **must** be the first tag inside `<table>`. This breaks the
  `<table><mysnippet:line>`
  `<tr><td><mysnippet:bind_here></mysnippet:bind_here></td></tr>`
  `</mysnippet:line><table>`
  paradigm. You can get the desired behavior with
  `<table><tr lift:bind="mysnippet:line"><td><mysnippet:bind_-here></mysnippet:bind_here></td></tr><table>`.

8.4.3 Changing behavior mid-session or mid-request

You can change the behavior of HtmlProperties mid-session or mid-request. `LiftSession.sessionHtmlProperties` is a `SessionVar` that contains the `HtmlProperties` for the session. `LiftSession.requestHtmlProperties` is a `TranientRequestVar` containing the `HtmlProperties` for the request. At the begining of the request, `requestHtmlProperties` is set to the value of `sessionHtmlProperties`. You can alter a property for the duration of the request using:

```
for {
 session <- S.session
} session.requestHtmlProperties.set(session.
       requestHtmlProperties.is.setDocType(() =>
                             Full("<!DOCTYPE moose>")))
```

Chapter 9

Built-in Snippets

9.1 CSS

9.2 Msgs

9.3 Msg

9.4 Menu

9.5 A

9.6 Children

9.7 Comet

9.8 Form

9.9 Ignore

9.10 Loc

9.11 Surround

9.12 TestCond

9.13 Embed

9.14 Tail

9.15 WithParam

9.16 VersionInfo

Chapter 10

SiteMap

Chapter 11

REST

Lift makes providing REST-style web services very simple.

First, create an object that extends `RestHelper`:

```
import net.liftweb.http._
import net.liftweb.http.rest._

object MyRest extends RestHelper {

}
```

And hook your changes up to Lift in `Boot.scala`:

```
LiftRules.dispatch.append(MyRest) // stateful -- associated with a servlet container sessio
LiftRules.statelessDispatchTable.append(MyRest) // stateless -- no session created
```

Within your MyRest object, you can define which URLs to serve:

```
serve {
  case Req("api" :: "static" :: _, "xml", GetRequest) => <b>Static</b>
  case Req("api" :: "static" :: _, "json", GetRequest) => JString("Static")
}
```

The above code uses the suffix of the request to determine the response type. Lift supports testing the `Accept` header for a response type:

```
serve {
  case XmlGet("api" :: "static" :: _, _) => <b>Static</b>
  case JsonGet("api" :: "static" :: _, _) => JString("Static")
}
```

The above can also be written:

```
serve {
  case "api" :: "static" :: _ XmlGet _=> <b>Static</b>
  case "api" :: "static" :: _ JsonGet _ => JString("Static")
}
```

Note: If you want to navigate your Web Service, you must remember to add a `*.xml` or `*.json` (depending in what you have implemented) at the end of the URL: `http://localhost:8080/XXX/api/static/call.json` `http://localhost:8080/XXX/api/static/call.xml`

Because the REST dispatch code is based on Scala's pattern matching, we can extract elements from the request (in this case the third element will be extracted into the id variable which is a String:

```
serve {
  case "api" :: "user" :: id :: _ XmlGet _ => <b>ID: {id}</b>
  case "api" :: "user" :: id :: _ JsonGet _ => JString(id)
}
```

And with extractors, we convert an element to a particular type and only succeed with the pattern match (and the dispatch) if the parameter can be converted. For example:

```
serve {
  case "api" :: "user" :: AsLong(id) :: _ XmlGet _ => <b>ID: {id}</b>
  case "api" :: "user" :: AsLong(id) :: _ JsonGet _ => JInt(id)
}
```

In the above example, id is extracted if it can be converted to a Long.

Lift's REST helper can also extract `XML` or `JSON` from a `POST` or `PUT` request and only dispatch the request if the XML or JSON is valid:

```
serve {
  case "api" :: "user" :: _ XmlPut xml -> _ => // xml is a scala.xml.Node
    User.createFromXml(xml).map { u => u.save; u.toXml}

  case "api" :: "user" :: _ JsonPut json -> _ => // json is a net.liftweb.json.JsonAST.JVal
    User.createFromJson(json).map { u => u.save; u.toJson}
}
```

There may be cases when you want to have a single piece of business logic to calculate a value, but then convert the value to a result based on the request type. That's where `serveJx` comes in … it'll serve a response for `JSON` and `XML` requests. If you define a trait called `Convertable`:

```
trait Convertable {
  def toXml: Elem
  def toJson: JValue
}
```

Then define a pattern that will convert from a `Convertable` to a `JSON` or `XML`:

implicit def cvt: JxCvtPF[Convertable] = { case (JsonSelect, c, _) => c.toJson case (XmlSelect, c, _) => c.toXml }

And anywhere you use `serveJx` and your pattern results in a `Box[Convertable]`, the `cvt` pattern is used to generate the appropriate response:

```
serveJx {
  case Get("api" :: "info" :: Info(info) :: _, _) => Full(info)
}
```

Or:

```
// extract the parameters, create a user
// return the appropriate response

def addUser(): Box[UserInfo] =
  for {
    firstname <- S.param("firstname") ?~ "firstname parameter missing" ~> 400
    lastname <- S.param("lastname") ?~ "lastname parameter missing"
    email <- S.param("email") ?~ "email parameter missing"
  } yield {
    val u = User.create.firstName(firstname).
      lastName(lastname).email(email)

    S.param("password") foreach u.password.set
    u.saveMe
  }

serveJx {
  case Post("api" :: "add_user" :: _, _) => addUser()
}
```

In the above example, if the `firstname` parameter is missing, the response will be a 400 with the response body “firstname parameter missing”. If the `lastname` parameter is missing, the response will be a 404 with the response body “lastname parameter missing”.

Chapter 12

MVC (If you really want it)

Chapter 13

From MVC

Okay, so you're coming from MVC-land and you're used to defining routes, defining controlers and defining views.

Lift is different. For HTML requests, Lift loads the view first and builds your page from the view. Lift also supports REST style requests for non-HTML data. (See 11 on page 105)

"Why?" Because complex HTML pages rarely contain a dominant piece of logic... a single controller... but contain many different components. Some of those components interact and some do not. In Lift, you define the collection of components to be rendered in the resulting HTML page in the view.

So, to create a page that has dynamic content, we need to do three things:

- Make a SiteMap entry for the page
- Create the view (the HTML)
- Create the behavior (the Snippet that transforms the incoming HTML to the dynamically generated HTML)

You can find the source for this project at https://github.com/dpp/simply_lift/tree/master/samples/from_mvc.

13.1 First things first

The first step to using Lift is to make sure you've got Java 1.6 or better installed on your machine... you'll need tar or zip as well.

Download the TAR or Zip version of the Lift templates and extract the files.

Copy the `lift_basic` project into another directory called `first_lift`.

`cd` into `first_lift` and type `sbt`. It will take a few minutes for `sbt`, the Simple Build Tool, to download all the depedencies. At the > prompt type `update` which will download Lift and everything else you need to get started. Once all that stuff is downloaded, type `jetty-run` and point your browser to `http://localhost:8080` and you'll see a live application. To continuously update your running application as you to code, enter `~prepare-webapp` at the sbt prompt.

13.2 Making a `SiteMap` entry

Every page on the site needs a `SiteMap` entry. For more on `SiteMap`, see 3.2 on page 15 and 7.5 on page 84.

Open the `Boot.scala` file (`src/main/scala/bootstrap/liftweb/Boot.scala`) and update the `SiteMap` definition:

```scala
    // Build SiteMap
    def sitemap(): SiteMap = SiteMap(
      Menu("Home") / "index",
      Menu("Second Page") / "second"
    )
```

13.3 Creating the view

Next you have to create a file that corresponds to the path defined in the `SiteMap`. So, let's look at the `src/main/webapp/index.html` file:

Listing 13.1: index.html

```html
<!DOCTYPE html>
<html>
  <head>
    <meta content="text/html; charset=UTF-8" http-equiv="content-type" />
    <title>Home</title>
  </head>
  <body class="lift:content_id=main">
    <div id="main" class="lift:surround?with=default&at=content">
      <div>
        Hi, I'm a page that contains the time:
        <span class="lift:TimeNow">??? some time</span>.
      </div>

      <div>
        And a button: <button class="lift:ClickMe">Click Me</button>.
      </div>
    </div>
  </body>
</html>
```

The page is valid HTML5. `<body class="lift:content_id=main">` says "ignore everything on this page except the Element with id 'main'."

`<div id="main" class="lift:surround?with=default&at=content">` says "Wrap the default page chrome around this Element."

`<span class="lift:TimeNow">??? some time</span>` says "Find the `TimeNow` snippet and transform this Element with the rules contained in that snippet." See 7.1 on page 78. The result will be `<span>Fri Jan 21 11:30:34 PST 2011</span>`

So, that's simple. You tell Lift what Snippet to use to transform your static content into dynamic content.

13.4 Creating the Snippet

Next you have to tell Lift what the rules are for transforming the section of your template based on dynamic rules. This is a Snippet... it's a function that transforms `NodeSeq => NodeSeq`. Let's look at the `TimeNow` snippet:

Listing 13.2: TimeNow.scala

```scala
// make sure this is the snippet package so Lift
// can find the snippet
package code
package snippet

// some inputs
import net.liftweb._
import util._
import Helpers._

// our snippet
object TimeNow {
 // create a function (NodeSeq => NodeSeq)
 // that puts the current time into the
 // body of the incoming Elem
 def render = "* *" #> now.toString
}
```

This snippet must be in the `snippet` package so Lift knows how to find it by convention.

It is an `object` which is a singleton because the snippet has no state.

Lift calls the `render` method on a snippet unless you specify another method when you invoke your snippet.

The snippet generates a function, `NodeSeq => NodeSeq`, that uses Lift's CSS Selector Transforms (See 7.10 on page 85) to insert the current time into the body of all HTML Elements: `def render = "* *" #> now.toString`

13.5 Getting Ajaxy

The `ClickMe` snippet is a little more complex, but it demonstrates, especially on the "Second Page" the power of Lift's View First in which no particular component on the page is the dominant component. Here's the `ClickMe` code:

Listing 13.3: ClickMe.scala

```scala
// make sure this is the snippet package so Lift
// can find the snippet
package code
package snippet

// some inputs
import net.liftweb._
import util._
import Helpers._
import http._
import js.JsCmds._

// our snippet
object ClickMe {
  // variables associated with the request
  private object pos extends RequestVar(0)
  private object cnt extends RequestVar(0)

  // create a function (NodeSeq => NodeSeq)
  // set the onClick method of the button
  def render = {
    // capture our position on the page
    val posOnPage = pos.set(pos.is + 1)

    "button [onclick]" #>
    SHtml.ajaxInvoke(() => {
      cnt.set(cnt.is + 1) // increment the click count
      Alert("Thanks pos: "+posOnPage+
          " click count "+cnt)
    })
  }
}
```

We define two `RequestVar`s that hold request-scoped values. For Lift, the scope of a request is the initial full HTML page load plus any Ajax requests associated with that page.

When the snippet's `render` method is called, we capture the current value for the `pos` `Request-Var`.

The snippet associates the invocation of an Ajax call with the button's `onclick` method. When the button is clicked, the function is invoked.

The function closed over the scope of the position of the button on the page. The buttons all share the `cnt` `RequestVar` and thus for a single page load, the number of button-presses are counted. If you have 5 different browser tabs open to the same page, each tab will have a unique page count.

This demonstrates the component nature of Lift and why having complex items on a page means not having a front-controller, but having lots of behaviors associated with lots of HTML elements.

13.6 Next Steps

If you want to see more of Lift's snazzy Ajax and Comet, check out 2 on page 5. If you want to see more of the basics of `SiteMap` and snippets, check out 3 on page 11. If you want to see how Lift does forms, check out 4 on page 27.

Part II

Recipes

Chapter 14

Dynamic html tables created from `DB.runQuery()`

14.1 Problem

What I'm trying is:

1. query the SQL server via `DB.runQuery()`

2. put the result (multiple, rows and columns) into a Table structure like this:

```
<table>
  <thead>
    <tr><th></th></tr>
  </thead>
  <tbody>
  <tr><td></td></tr>
  </tbody>
</table>
```

14.2 Solution

The DB.runQuery(sql_query_string) method returns (List[String], List[List[String]]), to put that in a table, your view looks like:

```
<table class="lift:MySnippet">
  <thead>
    <tr><th id="my_th">Field Name</th></tr>
  </thead>
  <tbody>
  <tr id="my_tr"><td>An item</td></tr>
  </tbody>
</table>
```

And your snippet uses CSS Selector Transforms (See Section 7.10) and looks like:

```
object MySnippet {
 def render = {
  val (fieldNames: List[String], fieldValues: List[List[String]]) = DB.runQuery(...)

  "#my_th *" #> fieldNames &
  "#my_tr *" #> fieldValues.map(values => "td *" #> values)
 }
}
```

Chapter 15

Dynamically choosing content

15.1 Problem

I want to to keep design completely separated from logic and I am bit stuck. I have a page that loads different pieces of html depending on some variables and it also has some ajax code so it may load new pieces of html. So far, that page uses only one snippet that has the logic to decide what html should be loaded. So here is the question, how should the snippet get an only-with-design piece of html to bind data to it.

15.2 Solution

Snippets are evaluated recursively... this means that you can return markup from a snippet that contains other snippets.

The other thing to pay attention to is the `<lift:embed>` snippet (See Section 9.13).

Combining the two:

Main page:

```
<html><body> Stuff here
<div class="lift:ChooseBehavior">Different behavior will go here</div>
</body></html>
```

The snippet:

```
object ChooseBehavior {
 def render = someState match {
   case ShowData => <lift:embed what="_showData" />
   case EditData => <lift:embed what="_editData" />
   case AjaxThing => <lift:embed what="_ajaxThing" />
 }
}
```

Then your designer need only edit the main page and each of the templates, and then you wire them together.

Chapter 16

Ajax Forms

Chapter 17

Protecting REST APIs

17.1 Problem

I want to expose part of my site as authenticated REST, but with custom authentication (not the HTTP based authentication).

Right now, I'm thinking of using a custom dispatch, but that means I'll have to check every request in the request handler itself to see if it is authenticated, right?

Authentication is just a SessionVar on the server, so it also implies I need a way to pass the session identifier back and forth between the REST client and the service. If it were a cookie I think it would be transparent, but I think Lift adds te session ids to the URLs (at least that's what I see in my address bar).

So, assuming I have a public "login" REST call that sets a SessionVar, how do I pass this transarently to the REST client? I have thought about a token system as well, but that seems like copying the session system.

Any suggestions?

17.2 Solution

If you've got a:

```
object MyService extends RestHelper {
....
}
```

And:

```
val ensureSession: PartialFunction[Req, Unit] = {
  case _ if isLoggedIn =>
}
```

then in Boot:

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

LiftRules.dispatch.append(ensureSession guard MyService)
```

This is a simple way to compose `PartialFunctions` and put a guard around all the elements of a `PartialFunction`.

Chapter 18

URI-based locale selection

18.1 Problem

I'm evaluating Lift and one thing I miss, or cannot see how toimplement, is the ability to have the locale determined from an URI-pattern. In Struts2 I have:

```
namespace="/{request_locale}"
```

So I can have an action (restful) invoked on an `URI=/no/companies/company/1` and it will call my CompanyAction with `id=1` and the locale

set to `no` If called from `URI=/en/companies/company/1` it will callthe same `CompanyAction` but the locale will be set to "`en`".

So my question is: Is it possible to teach Lift to retrieve the locale based on some uri-pattern, so that it will try to resolve my `*.xhtml` after the `/{request_locale}` part?

`/no/index.xhtml`

`/en/index.xhtml`

Should then map to the same templates but with different locale.

18.2 Solution

This is an ideal use of URL rewriting.

You have to hook up the module in `Boot.scala` with: `UrlLocalizer.init()`.

You can see a complete runnable example at DPP's GitHub Starting Point.

Here's the code:

```scala
package code
package lib

import net.liftweb._
import http._
import provider._
import common._

import java.util.Locale

object UrlLocalizer {
  // capture the old localization function
  val oldLocalizeFunc = LiftRules.localeCalculator

  /**
   * What are the available locales?
   */
  val locales: Map[String, Locale] =
    Map(Locale.getAvailableLocales.map(l => l.toString -> l) :_*)

  object currentLocale extends RequestVar(Locale.getDefault)

  /**
   * Extract the locale
   */
  def unapply(in: String): Option[Locale] =
    if (currentLocale.set_?) None // don't duplicate
  else locales.get(in) // if it's a valid locale, it matches

  /**
   * Calculate the Locale
   */
  def calcLocale(in: Box[HTTPRequest]): Locale =
    if (currentLocale.set_?) currentLocale.get
  else oldLocalizeFunc(in)

  /**
   * Initialize the locale
   */
  def init() {
    // hook into Lift
    LiftRules.localeCalculator = calcLocale

    // rewrite requests with a locale at the head
    // of the path
    LiftRules.statelessRewrite.append {
      case RewriteRequest(ParsePath(UrlLocalizer(locale) :: rest,
                                    _, _, _), _, _) => {
        currentLocale.set(locale)
        RewriteResponse(rest)
      }
    }
  }
}
```

Chapter 19

Embedding JavaScript in an HTML page

19.1 Problem

What am I doing wrong? I'm trying to output a javascript object into the page (so my front end guy can do some stuff with the data without parsing it out of elements by id) but it's replacing all the double quotes with `&quot;` (only in view source - if I inspect it then firebug converts them to double quotes again)

I've copied the example from EXPLORING LIFT, but it still does the same:

```
& ".data_as_object *" #> {
  JsCrVar("myObject", JsObj(("persons", JsArray(
     JsObj(("name", "Thor"), ("race", "Asgard")),
     JsObj(("name", "Todd"), ("race", "Wraith")),
     JsObj(("name", "Rodney"), ("race", "Human"))
  ))))
```

Becomes:

```
<div class="data_as_object" style="display: none;">var myObject =
{&quot;persons&quot;: [{&quot;name&quot;: &quot;Thor&quot;,
&quot;race&quot;: &quot;Asgard&quot;}, {&quot;name&quot;:
&quot;Todd&quot;, &quot;race&quot;: &quot;Wraith&quot;},
{&quot;name&quot;: &quot;Rodney&quot;, &quot;race&quot;:
&quot;Human&quot;}]
};</div>
```

I've noticed that if what I'm outputting is a number rather than a string then it's fine.

19.2 Solution

Try:

```
& ".data_as_object *" #> {
  Script(JsCrVar("myObject", JsObj(("persons", JsArray(

    JsObj(("name", "Thor"), ("race", "Asgard")),
    JsObj(("name", "Todd"), ("race", "Wraith")),
    JsObj(("name", "Rodney"), ("race", "Human"))
  )))))
```

JsExp are also Nodes, so they render out, but they render out escaped. Putting Script() around them turns them into:

```
<script>
// <![CDATA[
....
]]>
</script>
```

Part III

Questions and Answers

Chapter 20

Scaling

Lift is a web framework built on the Scala programming language. Lift takes advantage of many of Scala's features that allow developers to very concisely code secure, scalable, highly interactive web applications. Lift provides a full set of layered abstractions on top of HTTP and HTML from "close to the metal" REST abstractions up to transportation agnostic server push (Comet) support. Scala compiles to JVM byte-code and is compatible with Java libraries and the Java object model. Lift applications are typically deployed as WAR files in J/EE web containers... Lift apps run in Tomcat, Jetty, Glassfish, etc. just like any other J/EE web application. Lift apps can generally be monitored and managed just like any Java web app. Web Applications, Sessions, and State. All web applications are stateful in one way or another. Even a "static" web site is made up of the files that are served... the application's state is defined in those files. The site content may be served out of a database, but the content served does not depend on identity of the user or anything about the HTTP request except the contents of the HTTP request. These contents can include the URI, parameters, and headers. The complete value of the response can be calculated from the request without referencing any resources except the content resources. For the purpose of this discussion, I will refer to these as session-less requests. News sites like the UK Guardian, MSNBC, and others are prototypical examples of this kind of site. Sessions. Some applications are customized on a user-by-user basis. These applications include the likes of Foursquare and others where many HTTP requests make up a "session" in which the results of previous HTTP requests change the behavior of future HTTP requests. Put in concrete terms, a user can log into a site and for some duration, the responses are specific to that user. There are many mechanisms for managing sessions, but the most common and secure method is creating a cryptographically unique token (a session id), and putting that token in the Set-Cookie response header such that the browser will present that Cookie in subsequent HTTP requests for a certain period of time. The server-side state is referenced by the Cookie and the state is made available to the web application during the scope of servicing the request and any mutations the web app makes to session state during the request are kept on the server and are available to the application in subsequent requests. Another available technique for managing state is to serialize application state in the Cookie and deliver it to the browser such that the server is not responsible for managing state across requests. As we've recently discovered, this is a tremendously insecure way to manage application state. Further, for any moderately complex application, the amount of data the needs to be transferred as part of each request and response is huge. Migratory Sessions. Many web application managers allow for server-managed sessions to migrate across a cluster of web application servers. In some environments such as Ruby on Rails, this is a hard requirement because only one request at a time can be served per process, thus for any moderate traffic site, there must be multiple processes serving pages. There are many strategies for migrating state across processes: storing state on disk, in memcached, in a database (relational or NoSQL), or having some proprietary cluster communications protocol. In any of these scenarios sessions can migrate across the grid of processes serving requests for a given web application. Web applications that support migratory state are often referred to as "stateless" because the session state does not reside in the same process as the web application. Session Affinity. Some applications require that all requests related to a particular session are routed to the same process and that process keeps session-related content in local memory. In a cluster, there are multiple mechanisms for achieving session affinity... the two most popular being HAProxy and Nginx. Availability, Scalability, Security, Performance, and User Experience. There are many vectors on which to measure the overall-quality of a web application. Let's take a quick peek at each one. Availability. Availability of an application is the amount of time it gives a meaningful response to a request. Highly available applications generally span multiple pieces of hardware and often multiple data centers. Highly available applications are also typically available during upgrades of part of the system that makes up the application. Highly available applications have very few single points of failure and those single points of failure are usually deployed on very reliable hardware. Scalability. A scalable application can, within certain bounds, respond with similar performance to increased load by adding hardware to process more load. No system is infinitely or linearly scalable. However, many systems have grossly disproportionate load demands such that, for example, you can add a lot of web application front-ends to a Rails application before there's enough load on the back-end RDBMS such that scaling is impaired.

Security. The Internet is a dangerous place and no request that is received from the Internet can be trusted. Applications, frameworks, systems and everything else must be designed to be secure and resist attacks. The most common attacks on web application are listed in the OWASP Top Ten. Performance. Web application performance can be measured on two vectors: response time to a request and system resources required to service the request. These two vectors are inter-dependent. User Experience. The user experience of a web app is an important measure of its quality. User experience can be measured on many different vectors including perceived responsiveness, visual design, interactivity, lack of "hicups", etc. Ultimately, because we're building applications for users, the user experience is very important. Lift's trade-offs. Given the number and complexity related to the quality of a web application, there are a lot of trade-offs, implicit and explicit, to building a framework that allows developers and business people to deliver a great user experience. Let's talk for a minute about what Lift is and what it isn't. Lift is a web framework. It provides a set of abstractions over HTTP and HTML such that developers can write excellent web applications. Lift is persistence agnostic. You can use Lift with relational databases, file systems, NoSQL data stores, mule carts, etc. As long as you can materialize an object into the JVM where Lift is running, Lift can make use of that object. Lift sits on top of the JVM. Lift applications execute in the Java Virtual Machine. The JVM is a very high performance computing system. There are raging debates as to the relative performance of JVM code and native machine code. No matter which benchmarks you look at, the JVM is a very fast performer. Lift apps take advantage of the JVM's performance characteristics. Moderately complex Lift apps that access the database can serve 1,000+ requests per second on quad-core Intel hardware. Even very complex Lift apps that make many back-end calls per request can serve hundreds of requests per second on EC2 large instances. Lift as proxy. Many web applications, typically REST applications, provide a very thin layer on top of a backing data store. The web application serves a few basic functions to broker between the HTTP request and the backing store. These functions include: request and parameter validation, authentication, parameter unpacking, back-end service request, and translation of response data to wire format (typically XML or JSON). Lift can service these kinds of requests within the scope of a session or without any session at all, depending on application design. For more information on Lift's REST features, see Lift RestHelper. When running these kinds of services, Lift apps can be treated without regard for session affinity. Lift as HTML generator. Lift has a powerful and secure templating mechanism. All Lift templates are expressed as valid XML and during the rendering process, Lift keeps the page in XML format. Pages rendered via Lift's templating mechanism are generally resistant to cross site scripting attacks and other attacks that insert malicious content in rendered pages. Lift's templating mechanism is designer friendly yet supports complex and powerful substitution rules. Further, the rendered page can be evaluated and transformed during the final rendering phase to ensure that all script tags are at the bottom of the page, all CSS tags are at the top, etc. Lift's templating mechanism can be used to serve sessionless requests or serve requests within the context of a session. Further, pages can be marked as not requiring a session, yet will make session state available if the request was made in the context of a container session. Lift page rendering can even be done in parallel such that if there are long off-process components on the page (e.g., advertising servers), those components can be Sessionless Lift, forms and Ajax Lift applications can process HTML forms and process Ajax requests even if there's no session associated with the request. Such forms and Ajax requests have to have stable field names and stable URLs, but this is the same requirement as most web frameworks including Struts, Rails, and Django impose on their applications. In such a mode, Lift apps have the similar characteristics to web apps written on tops of Struts, Play, JSF and other popular Java web frameworks. Lift as Secure, Interactive App Platform Lift features require session affinity: GUID to function mapping, type-safe SessionVars and Comet. Applications that take advantage of these features need to have requests associated with the JVM that stores the session. I'll discuss the reason for this limitation, the down-side to the limitation, the downside to migratory session, and the benefits of these features. Application servers that support migratory sessions (sessions that are available to application servers running in multiple address spaces/processes) require a mechanism for transferring the state information between processes. This is typically (with the exception of Terracotta) done by serializing the stored data. Serialization is the process of converting rich data structures into a stream of bytes. Some of Scala's constructs are hard or impossible to serialize. For example, local variables that are mutated within a closure are promoted from stack variables to heap variables. When those variables are serialized at different times, the application winds up with two references even though the references are logically the same. Lift makes use of many of these constructs (I'll explain why next) and Lift's use of these constructs makes session serialization and migration impossible. It also means that Lift's type-safe SessionVars are not guaranteed to be serialized. One of the key Lift constructs is to map a cryptographically unique identifier in the browser to a function on the server. Lift uses Scala functions which close over scope, including all of the variables referenced by the function. This means that it's not necessary to expose primary keys to the client when editing a record in the database because the primary key of the record or the record itself is known to the function on the server. This guards against OWASP Vulnerability A4, Insecure Object References as well as Replay Attacks. From the developer's standpoint, writing Lift applications is like writing a VisualBasic application... the developer associates the user action with a function. Lift supplies the plumbing to bridge between the two. Lift's GUID to function mapping extends to Lift's Ajax support. Associating a button, checkbox, or other HTML element with an Ajax call is literally a single line: SHtml.ajaxButton(<b>PressMe</b>, () => Alert("You pressed a button at "+Helpers.currentTimeFormatted) Lift's Ajax support is simple, maintainable, and secure. There's no need to build and maintain routing. Lift has the most advanced server-push/Comet support of any web framework or any other system currently available. Lift's comet support relies on session affinity. Lift's comet support associates an Actor with a section of screen real estate. A single browser window may have many pieces of screen real estate associated with many of Lift's CometActors. When state changes in the Actor, the state change is pushed to the browser. Lift takes care of multiplexing a single HTTP connection to handle all the comet items on a given page, the versioning of the change deltas (if the HTTP connection is dropped while 3 changes become available, all 3 of those changes are pushed when the next HTTP request is made.) Further, Lift's comet support will work the same way once web sockets are available to the client and server... there will be no application code changes necessary for web sockets support. Lift's comet support requires that the connect is made from the browser back to the same JVM in which the CometActors are resident... the same JVM where the session is located.

The downside to Lift's session affinity requirement mainly falls on the operations team. They must use a session aware load balancer or other mechanism to route incoming requests to the server that the session is associated with. This is easily accomplished with HAProxy and Nginx. Further, if the server running a given session goes down, the information associated with that session is lost (note that any information distributed off-session [into a database, into a cluster of Akka actors, etc.] is preserved.) But, Lift has extended session facilities that support re-creation of session information in the event of session lost. Lift also has heart-beat functionality so that sessions are kept alive as long as a browser page is open to the application, so user inactivity will not result in session timeouts.

Compared to the operational cost of a session aware load balancer, there are many costs associated with migratory sessions. First, there must be a persistence mechanism for those sessions. Memcached is an unreliable mechanism as memcached instances have no more stability than the JVM which hosts the application and being a cache, some sessions may get expired. Putting session data in backing store such as MySQL or Cassandra increases the latency of requests. Further, the costs of serializing state, transmitting the state across the network, storing it, retrieving it, transmitting it across the network, and deserializing it all costs a lot of cycles and bandwidth. When your Lift application scales beyond a single server, beyond 100 requests per second, the costs of migrating state on every request becomes a significant operational issue.

Session serialization can cause session information loss in the case of multiple requests being executed in multiple processes. It's common to have multiple tabs/windows open to the same application. If session data is serialized as a blob and two different requests from the same server are being executed at the same time, the last request to write session data into the store will overwrite the prior session data. This is a concurrency problem and can lead to hard to debug issues in production because reproducing this kind of problem is non-trivial and this kind of problem is not expected by developers.

The third issue with migratory sessions and session serialization is that the inability to store complex information in the session (e.g., a function that closes over scope) means that the developer has to write imperative code to serialize session state to implement complex user interactions like multi-screen wizards (which is a 400 line implementation in Lift). These complex, hand written serializations are error prone, can introduce security problems and are non-trivial to maintain.

The operational costs of supporting session affinity are not materially different from the operational costs of providing backing store for migratory sessions. On the other hand, there are many significant downsides to migratory sessions. Let's explore the advantages of Lift's design.

Lift's use of GUIDs associated with functions on the server: Increase the security of the application by guarding against cross site request forgeries, replay attacks, and insecure object references

Decrease application development and maintenance time and costs Increase application interactivity, thus a much better user experience Increase in application richness because of simpler Ajax, multi-page Wizards, and Comet Improved application performance because fewer cycles are spent serializing and transmitting session information No difference in scalability... just add more servers to the front end to scale the front end of your application The positive attributes of Lift's design decisions are evident at Foursquare which handles thousands of requests per second all served by Lift. There are very few sites that have more traffic than Foursquare. They have scaled their web front end successfully and securely with Lift. Other high volume sites including Novell are successfully scaling with Lift. If you are scaling your site, there are also commercial Lift Cloud manager tools that can help manage clusters of Lift's session requirements. Conclusion Lift provides a lot of choices for developing and deploying complex web applications. Lift can operate in a web container like any other Java web framework. If you choose to use certain Lift features and you are deploying across multiple servers, you need to have a session aware load balancer. Even when using Lift's session-affinity dependent features, Lift applications have higher performance, identical availability, identical scalability, better security, and better user experience than applications written with web frameworks such as Ruby on Rails, Struts, and GWT.

Chapter 21

How Lift does function/GUID mapping

Chapter 22

How Lift does Comet

I can speak to Lift's Comet Architecture which was selected by Novell to power their Pulse product after they evaluated a number of different technologies.

Lift's Comet implementation uses a single HTTP connection to poll for changes to an arbitrary number of components on the page. Each component has a version number. The long poll includes the version number and the component GUID. On the server side, a listener is attached to all of the GUIDs listed in the long poll requests. If any of the components has a higher version number (or the version number increases during the period of the long poll), the deltas (a set of JavaScript describing the change from each version) is sent to the client. The deltas are applied and the version number on the client is set to the highest version number for the change set.

Lift integrates long polling with session management so that if a second request comes into the same URL during a long poll, the long poll is terminated to avoid connection starvation (most browsers have a maximum of 2 HTTP connections per named server). Lift also supports DNS wild-carded servers for long poll requests such that each tab in the browser can do long polling against a different DNS wildcarded server. This avoids the connection starvation issues.

Lift dynamically detects the container the Servlet is running in and on Jetty 6 & 7 and (soon) Glassfish, Lift will use the platform's "continuations" implementation to avoid using a thread during the long poll.

Lift's JavaScript can sit on top of jQuery and YUI (and could sit on top of Prototype/Scriptaculous as well.) The actual polling code includes back-off on connection failures and other "graceful" ways of dealing with transient connection failures.

I've looked at Atmosphere and CometD (both JVM-oriented Comet technologies). Neither had (at the time I evaluated them) support for multiple components per page or connection starvation avoidance.

Chapter 23

Advanced Concepts

23.1 Snippet Resolution

Lift snippets transform markup to dynamic content. The are functions that transform `NodeSeq => NodeSeq`.

Snippets can be invoked from templates via tags:

```
<lift:surround with="default" at="content">
  <p>
   You have reached this page, but you can only get here if you've logged in
   first.
  </p>
</lift:surround>
```

or via class attributes.

```
<p class="lift:surround?with=default;at=content">
  You have reached this page, but you can only get here if you've logged in
  first.
</p>
```

In both cases, the surround (See Section 9.11) snippet will be invoked with attribute `with` set to `default` and `at` set to `content`. The parameter passed to the surround `NodeSeq => NodeSeq` function is:

```
<p>
  You have reached this page, but you can only get here if you've logged in
  first.
</p>
```

Lift will resolve from the snippet name to a function in the following steps.

23.1.1 `LiftSession.liftTagProcessing`

Lift consults a `List[PartialFunction[(String, Elem, MetaData, NodeSeq, String), NodeSeq]]` located in LiftSession.liftTagProcessing for the rules to use to evaluate the snippet name, attributes, etc. into the resulting `NodeSeq`. `LiftSession.liftTagProcessing` is the result of `LiftRules.liftTagProcessing` or else the default Lift tag processor. If you need special snippet resolution mechanisms, you can place them in `LiftRules.liftTagProcessing`. By default, the snippets get processed by `LiftSession.processSnippet`.

23.1.2 `LiftRules.liftTagProcessing`

`LiftRules.liftTagProcessing` looks for the `form` attribute and sets the `isForm` variable. Next, Lift determines if the contents of the snippet should be evaluated eagerly by looking for one of `eager_eval`, `l:eager_eval`, or `lift:eager_eval` attributes.

If the snippet is an eager evaluation, the child tags will be evaluated for any snippets.

Either the originally passed children or the eagerly evaluated children will be referred to as children in the next section.

23.1.3 Snippet name resolution

Lift looks for the named snippet in the following locations in order:

- `S.locateMappedSnippet` - the complete snippet name without any camel or snake application is used to look up a `NodeSeq => NodeSeq` in within the scope of the current extended request[1]. Snippets may be registered using `S.mapSnippet`.
- `SiteMap Loc` snippet - the current `SiteMap Loc` (`S.location`) will be queried to see if it has a `NodeSeq => NodeSeq` that matches the current snippet name (`loc.snippet(snippetName)`).
- `LiftRules.snippets` - next, the snippet name is split at the '.' character to determine the snippet name and snippet method name. The `snippets RulesSeq` is tested for a match between the `List[String]` that results from splitting the name at the period and `NodeSeq => NodeSeq`.
- If the above mechanisms do not result in a NodeSeq => NodeSeq, Lift looks for a `Class` that matches the name.
  - `S.snippetForClass` - is checked to see if a `Class` has been associated with the snippet name. If none is found...

[1]For the purposes of this discussion, the extended request is the scope of a `RequestVar`. This is the scope of a full page render plus any subsequent Ajax operations that originate from that page. This means that a snippet may be registered using `S.mapSnippet` during page rendering and the same snippet function with the same scope binding will be used by any Ajax commands.

- `LiftRules.snippetDispatch` is checked to see if theres an instance of `DispatchSnippet` that matches to snippet name. Lift's built-in snippets are registered with `LiftRules.snippetDispatch`. If there's no match...
- Lift tries reflection to find a matching class name (note that Lift will try camel case and snake case for class names, so the `foo_bar` snippet will match the class `foo_bar` as well as `FooBar`). Lift looks for classes in the `snippet` subpackage of all the packages added via `LiftRules.addToPackages`. So if you call `LiftRules.addToPackages("foo.bar")` in `Boot.scala`, then Lift will search for the classes `foo.bar.snippet.foo_bar` and `foo.bar.snippet.FooBar`.
- Once the class is found, Lift will try to instantiate the class the following ways:
  * Lift will look at the current location (`S.location`) and if the parameter type of the `Loc` is not `Unit`, Lift get the current parameter and look for a constructor that matches the current parameter type or `Box` of current parameter type (and superclasses of both). If there's a match the constructor will be called with the parameters. For example, if the current page is a `Loc[Dog]` and `Dog` is a subclass of `Animal`, the following constructors will match:
    · `class MySnippet(dog: Dog)`
    · `class MySnippet(animal: Animal)`
    · `class MySnippet(dog: Box[Dog])`
    · `class MySnippet(animal: Box[Animal])`
    · `class MySnippet(dog: Dog, session: LiftSession)`
    · `class MySnippet(animal: Animal, session: LiftSession)`
    · `class MySnippet(dog: Box[Dog], session: LiftSession)`
    · `class MySnippet(animal: Box[Animal], session: LiftSession)`
  * If a typed constructor cannot be found, try the zero argument constructor;
  * If the zero argument constructor cannot be found, try to treat the `Class` as a Scala `object` singleton and get the instance that the singleton refers to.
- Once we've got an instance of the potential snippet handling class:
  * If it's a `StatefulSnippet`, register with `S.overrideSnippetForClass`;
  * Update the `LiftSession` `snippetMap` `RequestVar` so subsequent references to the snippet during the same extended request uses same instance (that way if any instance variables are set on the class instance, they are picked up by subsequent accesses to the same snippet);
  * Next, Lift attempts to invoke the snippet method. If no explicit method is given, the `render` method is used.
    · Stateful and Dispatch use `dispatch` method to find the `NodeSeq => NodeSeq`
    · Non-dispatch, do the following method lookup:
    · method that takes no parameters and returns `CssBindFunc`, `NodeSeq => NodeSeq`, invoke the method and apply the function to the children; or
    · try to invoke the named method with `Group(children)` (`NodeSeq` signature) or invoke it with no parameters. If the return value is `NodeSeq`, `Node`, or `Seq[Node]`, then it was successful.

23.1.4 Post-processing of results

- LiftRules.snippetDispatch (built in snippets registered here)

parallel snippets

23.2 The Merging Phase

Part IV

Misc

Chapter 24

Releases

24.1 Lift 2.2-RC1

December 8, 2010

The Lift team is pleased to announce Lift 2.2-RC1. In the month since the 2.2-M1 release, the team has closed 53 tickets and made significant improvements to Lift based on community feedback.

Lift is an elegant, expressive framework that allows any size team build and maintain secure, highly interactive, scalable web applications quickly and efficiently. Lift is built on Scala and compiles to JVM byte-code. Lift applications deploy as WAR files on popular application servers and web containers including Jetty, Glassfish and Tomcat. Lift applications can be monitored and managed with the same proven infrastructure used to manage and monitor any Java web application. Lift is open source licensed under an Apache 2.0 license.

Lift features include:

- Community... the Lift community is 2,400 members strong, super-active and always there to help with questions
- Best Comet (server-push) support that allows the creation of dynamic application such as Novell Vibe
- Super simple Ajax for creating highly interactive web applications without worrying about HTTP plumbing
- Secure by default... Lift apps are resistant to the OWASP top 10 vulnerabilities including XSS, XSRF, and parameter tampering
- Concise and Maintainable... Lift apps typically contain fewer lines of code than corresponding Rails apps, yet are type safe so that many errors are flagged by the compiler
- Scalable... Lift apps scale to millions of users across many servers, yet are highly efficient for single-box implementations
- Compatible... Lift apps can take advantage of any Java library as well as the growing collection of Scala libraries

Lift 2.2-RC1 improvements include:

- HTML5 Support: Lift supports parsing HTML5 input files and rendering HTML5 to the browser in addition to Lift's XHTML support
- Wiring: Spreadsheets meet web application yielding an automatic mechanism for updating dependent elements on a page, making it even easier to build dynamic apps with Lift
- Wizard and Screen Improvements: Build complex screens more easily with new helper methods for creating form elements and improved life-cycle callbacks
- CSS Selector Transforms Improvements: including appending attributes, multiple selectors applying to a single element, and element lifting

- Support for migratory sessions: `ContainerVars` provide type-safe, guaranteed serializable session variables that can migrate across application servers in a cluster
- Improved i18n: including per-page localization strings and localization strings and HTML stored in templates rather than Java resource files which makes editing much easier
- Security Improvements: including creation of new sessions on login
- MongoDB Improvements: performance improvements as well as new features
- Support for Scala 2.8.1 as well as 2.8.0 and 2.7.7
- ProtoUser support for Record: Lift's `ProtoUser` and `CRUDify` can be used on Record-based persistence classes as well as Mapper-based persistence classes
- Squeryl integration improvements: Lift is updated to use the latest version of Squeryl

Lift-powered sites include:

- Foursquare: the multi-million user location based service that services millions of check-ins a day on their Lift-powered system
- Novell Vibe: enterprise collaboration software platform based on Google Wave
- Innovation Games: The fun way to do serious business – seriously
- Xerox/XMPie: the leading provider of software for cross-media, variable data one-to-one marketing
- Exchango: The easy and convenient way to give and get free stuff.
- Snapsort: Compare and decide on cameras
- No Fouls: Find pickup basketball games

Please join the Lift community and help use grow Lift. And a super-big thanks to the 30+ Lift committers who have grown the Lift community and code-base to what it is today... and what it will be in the future!

24.2 Lift 2.2

January 5, 2011

The Lift team is pleased to announce Lift 2.2. In the three months since the 2.1 release, the team has closed over 100 tickets and made significant improvements to Lift based on community feedback.

Lift is an elegant, expressive framework that allows any size team build and maintain secure, highly interactive, scalable web applications quickly and efficiently. Lift is built on Scala and compiles to JVM byte-code. Lift applications deploy as WAR files on popular application servers and web containers including Jetty, Glassfish and Tomcat. Lift applications can be monitored and managed with the same proven infrastructure used to manage and monitor any Java web application. Lift is open source licensed under an Apache 2.0 license.

Lift features include:

- Community... the Lift community is 2,400 members strong, super-active and always there to help with questions
- Best Comet (server-push) support that allows the creation of dynamic application such as Novell Vibe
- Super simple Ajax for creating highly interactive web applications without worrying about HTTP plumbing
- Secure by default... Lift apps are resistant to the OWASP top 10 vulnerabilities including XSS, XSRF, and parameter tampering
- Concise and Maintainable... Lift apps typically contain fewer lines of code than corresponding Rails apps, yet are type safe so that many errors are flagged by the compiler
- Scalable... Lift apps scale to millions of users across many servers, yet are highly efficient for single-box implementations
- Compatible... Lift apps can take advantage of any Java library as well as the growing collection of Scala libraries

Lift 2.2 improvements include:

- HTML5 Support: Lift supports parsing HTML5 input files and rendering HTML5 to the browser in addition to Lift's XHTML support
- Wiring: Spreadsheets meet web application yielding an automatic mechanism for updating dependent elements on a page, making it even easier to build dynamic apps with Lift
- Wizard and Screen Improvements: Build complex screens more easily with new helper methods for creating form elements and improved life-cycle callbacks
- CSS Selector Transforms Improvements: including appending attributes, multiple selectors applying to a single element, and element lifting

- Support for migratory sessions: `ContainerVars` provide type-safe, guaranteed serializable session variables that can migrate across application servers in a cluster
- Improved i18n: including per-page localization strings and localization strings and HTML stored in templates rather than Java resource files which makes editing much easier
- Security Improvements: including creation of new sessions on login
- MongoDB Improvements: performance improvements as well as new features
- Support for Scala 2.8.1 as well as 2.8.0 and 2.7.7
- ProtoUser support for Record: Lift's `ProtoUser` and `CRUDify` can be used on Record-based persistence classes as well as Mapper-based persistence classes
- Squeryl integration improvements: Lift is updated to use the latest version of Squeryl
- Designer-friendly templates
- Stateless renderingincluding the HTML pipeline
- Support for MVC-style development

Lift-powered sites include:

- Foursquare: the multi-million user location based service that services millions of check-ins a day on their Lift-powered system
- Novell Vibe: enterprise collaboration software platform based on Google Wave
- Innovation Games: The fun way to do serious business – seriously
- Xerox/XMPie: the leading provider of software for cross-media, variable data one-to-one marketing
- Exchango: The easy and convenient way to give and get free stuff.
- Snapsort: Compare and decide on cameras
- No Fouls: Find pickup basketball games

Please join the Lift community and help use grow Lift. And a super-big thanks to the 30+ Lift committers who have grown the Lift community and code-base to what it is today... and what it will be in the future!

Index