Starting with Lift

Derek Chen-Becker, Marius Danciu, David Pollak and Tyler Weir

March 5, 2009

Chapter 1

Welcome to Lift!

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

1.1 Why Lift?

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

One of the key strengths of Lift is the clean separation of presentation content and logic, based on the bedrock concept of the Model-View-Controller pattern[1]. One of the original Java web application technologies that's still in use today is JSP, or Java Server Pages[2]. JSP allows you to mix HTML and Java code directly within the page. While this may have seemed like a good idea at the start, it has proven to be painful in practice. Putting code in your presentation layer makes it more difficult to debug and understand what is going on within a page, and makes it more difficult for the people writing the HTML portion because the contents aren't valid HTML. While many modern programming and HTML editors have been modified to accomodate this mess, proper syntax highlighting and validation don't make up for the fact that you still have to switch back and forth between one or more files to follow the page flow. Lift takes the approach that there should be no

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

code in the presentation layer, but that the presentation layer has to be flexible enough to accomodate any conceivable uses. To that end, Lift uses a powerful templating system, a la Wicket[3], to bind user-generated data into the presentation layer. Lift's templating is built on the XML processing capabilities of the Scala Language[4], and allows things such as nested templates, simple injection of user-generated content, and advanced data binding capabilities. For those coming from JSP, Lift's advanced template and XML processing allows you to essentially write custom tag libraries at a fraction of the cost in time and effort.

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

Lift strives to encompass advanced features in a very concise and straightforward manner. Lift's powerful support for AJAX and COMET allow you to use Web 2.0 features with very little effort. Lift leverages Scala's Actor library to provide a message-driven framework for COMET updates. In most cases, adding COMET support to a page just involves extending a trait[7] to define the rendering method of your page and adding an extra function call to your links to dispatch the update message; Lift handles all of the backend and page-side coding to effect the COMET polling. AJAX support includes special handlers for doing AJAX form submission via JSON, and almost any link function can easily be turned into an AJAX version with a few keystrokes. In order to preform all of this client-side goodness, Lift has a class hierarchy for encapsulating Javascript calls via direct Javascript, jQuery and YUI. The nice part is that you, too, can utilize these support classes so that you can generate the code and don't have to put Javascript logic into your templates.

1.2 For more information

Lift has a very active community of users and developers. Since its inception in early 2007 the community has grown to hundreds of members from all over the world. The project's leader, David

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

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

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

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

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

Pollak[8]is constantly attending to the mailing list, answering questions and taking feature requests. There is a core group of developers who work on the project, but submissions are taken from anyone who makes a good case and can turn in good code. While we strive to cover everything you'll need to know in this book, here are several additional resources available for information on Lift:

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

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

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

1.3 Your first Lift application

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

Java 1.5 JDK Lift runs on Scala, which runs on top of the JVM; the first thing you'll need to install is a modern version of the Java SE JVM, available at http://java.sun.com/. Please install JDK 1.5 or newer. If you're using a Mac, the JDK is already installed.

Maven 2 Maven is a project management tool. Maven has extensive capabilities for building, dependency management, testing and reporting. If you haven't used Maven before you can think of it as an incredibly powerful version of `make` for now. You can download the latest version of Maven from http://maven.apache.org/. Brief installation instructions (enough to get us started) are on the download page, at http://maven.apache.org/download.html. You must have Maven 2.0.9 or greater installed. Mac OS X has Maven 2.0.6 installed. Please install 2.0.9 and make sure that the version that you installed is the first version accessed in the PATH.

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

Eclipse http://www.eclipse.org/ The Scala plugin developer recommends using the *Eclipse Classic* version of the IDE

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

NetBeans http://www.netbeans.org Requires using the NetBeans 6.5

IntelliJ IDEA http://www.jetbrains.com/idea/index.html Requires version 8 beta

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

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

```
cd work
```

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

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

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

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

Maven should produce more output, ending with

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

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

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

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

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

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

```
<prefix:element>
```

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

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

    <title>demo.helloworld:helloworld:1.0-SNAPSHOT</title>
    <script id="jquery" src="/classpath/jquery.js" type="text/javascript">
    </script>
  </head>
  <body>
    <lift:bind name="content" />
    <lift:Menu.builder />
    <lift:msgs/>
  </body>
</html>
```

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

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

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

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

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

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

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

```
package demo.helloworld.snippet

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

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

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

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

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

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

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

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

Chapter 2

Basic Lift

2.1 What we're going to cover

We're going to build a basic to-do application in Lift. The application will allow you to create an account, log in, and then enter and modify to-do items.

The goal is to show you:

- How to create a basic Lift web application
- How to create a model that maps to a relational database
- How to create web pages that correspond to back end logic
- How to bind the elements on the web page to dynamically created content

2.2 Creating a new project

The first thing to-do is create a new project. We do this with a build tool called Maven. Maven manages the compile and test phases of development as well as managing any dependencies by downloading resources from Internet.

From a command prompt, type:

Listing 2.1: Create a new project

```
mvn archetype:generate -U \
 -DarchetypeGroupId=net.liftweb \
 -DarchetypeArtifactId=lift-archetype-basic \
 -DarchetypeVersion=1.0 \
 -DremoteRepositories=http://scala-tools.org/repo-releases \
 -DgroupId=com.liftworkshop \
 -DartifactId=todo \
 -Dversion=0.1-SNAPSHOT
```

Now change directories into the newly created todo directory with: `cd todo`

Type: `mvn jetty:run` and in your browser, go to http://localhost:8080 You will see that your application is running. Switch back to your terminal window and type `ctrl-c` to stop the Jetty web server.

2.3 Adding a model

Now that we've got a basic application running, we're going to add some custom code. Create the file `src/main/scala/com/liftworkshop/model/ToDo.scala` Put the following code into the file:

Listing 2.2: The ToDo model

```
package com.liftworkshop.model

import net.liftweb._
import mapper._
import http._
import SHtml._
import util._

class ToDo extends LongKeyedMapper[ToDo] with IdPK {
  def getSingleton = ToDo

  object done extends MappedBoolean(this)
  object owner extends MappedLongForeignKey(this, User)
  object priority extends MappedInt(this) {
    override def defaultValue = 5
  }
  object desc extends MappedPoliteString(this, 128)
}

object ToDo extends ToDo with LongKeyedMetaMapper[ToDo]
```

Let's explore the file.

The `package` statement tells the Scala compiler what package, or code hierarchy, these classes are associated with.

The `import` statements tell the compiler that we're going to be using resources from other packages and `object` singletons.

`class ToDo extends LongKeyedMapper[ToDo]` creates a class which is a subclass of the `LongKeyedMapper` class. This class maps objects, models, to a relational database. The class requires that the table has a primary key. The `[ToDo]` part declares that the Mapper is mapping a ToDo. You may be thinking that should be inferred by the compiler, but sometimes the compiler needs some extra hints.

`with IdPK` mixes a trait into `ToDo`. A trait is like a Java interface, that also defines methods. Many traits can be mixed into a single class. Traits can define rules about what kind of classes they can be mixed into. Traits are similar to Ruby mixins.

In this case, the `IdPK` trait adds an `id` field to the `ToDo` and makes the `id` field the primary key of `ToDo`.

Within the `ToDo` class, `def getSingleton = ToDo` defines the singleton or meta object that is associated with the model class. Huh? Each instance of `ToDo` class represents a row in the database. There may be many instances of the `ToDo` class. There is a single object that represents the database table itself. It has factory methods that will vend new instances of `ToDo`s or allow queries that will return a set of `ToDo`s. The `ToDo` object singleton is like Smalltalk and Ruby class objects. It is not like Java static methods as the singleton is an instance rather than a collection of static methods on a class. `getSingleton` associates the `ToDo` instance with the `ToDo` singleton.

`object done extends MappedBoolean(this)` defines a Boolean `done` field on the `ToDo` model. `object done` declares that there is a field on the `ToDo` model. The field is represented as a singleton, a single instance of a class, per `ToDo` model instance. The `done` field `extends MappedBoolean(this)` extends the `MappedBoolean` class and is associated with this instance of the `ToDo` model. This is the pattern for defining fields that map between model objects and relations databases.

`object owner extends MappedLongForeignKey(this, User)` defines an `owner` field on the `ToDo` model. The `owner` field is a foreign key reference to the `User` model/table.

We declare the priority field as a `MappedInt`. We subclass `MappedInt` and change the default value of the priority field to 5. The ability to subclass fields and change behavior such as default values, validation rules, etc. is a very powerful construct. It keeps the business logic right where the field is defined. It's also using Scala rather than some annotations. Because it's written in Scala, developers need not learn a separate annotations "language" in order to describe properties of a field. Additionally, the tools used to develop the application don't need to know about or validate a separate set of annotations.

`object desc extends MappedPoliteString(this, 128)` defines the `desc` field as a `String` on the `ToDo` model. The `desc` field is the description of the to-do item. A `MappedPoliteString` maps a `String` in Scala to a `VARCHAR` column in the database. It's "polite" because it will automatically truncate the string to the maximum column size, in this case 128 characters, before storing it in the database.

Fields have a type beyond `String`, `Int`, `Boolean`. Lift comes with a number of helper field types including `MappedEmail`, `MappedUniqueId`, `MappedTimeZone`, `MappedPostalCode`, etc. The fields can have their own presentation logic for display and form generation. They can have validation and conversion rules. For example, `MappedEmail` validates the entry against a regular expression for email addresses as well as converting all input to lower case. The conversion to lower case also applies to queries against the field. This means that the logic of "all email addresses are in lower case" is encapsulated in a single location in your program, rather than at each location that does a query against the database. Mapped fields also have read and write access control rules. Thus, you could create a `MappedTaxpayerId` that would obscure itself on display except if it's being presented to the owner. Encapsulation of the business and presentation logic of the fields like this makes for more secure applications that also have lower defects. Keeping the semantic meaning (it's an email address, not just a `String`), with the fields for as long as possible helps everyone on the project better understand the code.

`object ToDo extends ToDo with LongKeyedMetaMapper[ToDo]` creates a singleton object which provides the meta functionality to the `ToDo` class. The `ToDo` object is a singleton. One instance will be created the first time the object is referenced in your program. It extends the `ToDo` class that we've defined with `LongKeyedMetaMapper[ToDo]`. The `LongKeyedMetaMapper` trait extends the class with the methods to create instances of `ToDo`, perform queries, and do general housekeeping. For example, the `ToDo` object has a `findAll()` method that returns a `List[ToDo]`. The `findAll()` method is supplied by `LongKeyedMetaMapper`.

2.4 Boot and Schemifier

Next, let's update the `Boot` file so that Lift's Schemifier will manage the `ToDo` table. Open `src/main/scala/bootstrap/liftweb/Boot.scala` and change the line:

Listing 2.3: Original Schemifier line

```
Schemifier.schemify(true, Log.infoF _, User)
```

To:

Listing 2.4: Updated Schemifier line

```
Schemifier.schemify(true, Log.infoF _, User, ToDo)
```

Scala is a statically typed language. That means at compile time, the types of all parameters and variables is known. This allows the compiler to perform many checks that one would otherwise have to write tests for. It also allows new team members to better understand code because the type of parameters to method calls must be declared. Further, a powerful typing system such as Scala's allows you to reason about the program's behavior.

Because the Lift model classes define the columns they expect to be in the database and the types and rules associated with those columns, Lift provides a mechanism to update the database to reflect the schema that the model expects. That mechanism is Schemifier which takes a collection of models, inspects the database and makes sure the right tables, right columns, right indexes, and right constraints are in the database. Schemifier has lots of different modes and hooks. It can inspect the database and recommend changes or make the changes for you. It can invoke code on table and column creation, allowing for migration of table data.

The above code adds the `ToDo` model to the list of models that Schemifier looks for in the database.

`Boot.scala` is the first code that's executed when the Lift servlet gets loaded into the servlet container. For those non-J2EE sorts, it means that `Boot` is run early and run only once.

`Boot` is where you wire your application together. You define rules that route URLs to code. You define how the database connections are generated, which, by default, looks to JNDI. You define how database transactions related to the HTTP request/response life-cycle, site navigation and access control rules, how to determine locale if browser headers are not good enough, etc. For the ToDo application, we'll only change Schemifier as the rest of the settings are correct.

2.5 Snippets: Bridging View and Logic

Web applications present information to the user primarily using HTML or XHTML. HTML designers are good at creating HTML and CSS, but writing code that involves looping, branching, fields, methods or logic is not what HTML designers do. Lift bridges between the XHTML markup and the logic that creates the dynamic content with XML `<lift:xxx/>` tags. These tags refer to Lift snippets... snippets of code that handle snippets of display logic. Some Lift tag behavior is predefined. Tags such as `<lift:surround/>` have a default behavior. However, you can create your own `<lift:xxx/>` tags. We're going to create the snippet code for `<lift:Util.in/>` and `<lift:Util.out/>` tags. The `<lift:Util.in/>` will only display its sub-children if a User is logged in. The `<lift:Util.out/>` tag will only display its content if no User is logged in. Thus:

```
<lift:Util.out>
  Please Log In <b>Dude</b>
</lift:Util.out>
```

Will only display "Please Log In **Dude**" if nobody is logged in.

Create the file `src/main/scala/com/liftworkshop/snippet/Util.scala`. Populate the file with:

Listing 2.5: Util snippet

```
package com.liftworkshop.snippet

import scala.xml.{NodeSeq}
import com.liftworkshop._
import model._

class Util {
  def in(html: NodeSeq) =
    if (User.loggedIn_?) html else NodeSeq.Empty

  def out(html: NodeSeq) =
    if (!User.loggedIn_?) html else NodeSeq.Empty
}
```

`package com.liftworkshop.snippet` defines the package for the code. Lift looks for classes in the "snippet" package of your project. If you look back in `Boot`, you'll see `LiftRules.addToPackages("com.liftworkshop")` line. This line tells Lift that your project package starts with "com.liftworkshop". Lift uses this base package name to look up snippets in the "snippet" sub-package, view code in the "view" sub-package, and comet code in the "comet" sub-package.

The `import` lines tell Scala which other packages we're using.

`class Util` defines the name of the class. By default, Lift looks for a class named "Util" when it encounters the `<lift:Util.xxx/>` tag. Lift will convert underscore separated names to CamelCase names when it looks up classes. Lift then looks for a public method named what comes after the period in the tag name. In Scala, unless you define a method's protection level, it's public. Lift also only looks for methods defined in the class, not on super-classes. This is to avoid any accidental access to methods from view tags. You can also define custom mapping rules for snippets. Also, Lift will look for a method called `render` on the class if there's no period in the tag name. Thus, `<lift:Util/>` will map to `Util.render`.

Some examples:

View code	Class	Method
`<lift:foo/>`	`Foo`	`render`
`<lift:FooBar/>`	`FooBar`	`render`
`<lift:foo_bar/>`	`FooBar`	`render`
`<lift:foo_bar.baz/>`	`FooBar`	`baz`

`def in(html: NodeSeq) =` defines a method that takes a parameter named `html` with has `NodeSeq` as the type. We do not need to declare the return type as the compiler will infer it. For any methods that are moderately complex, it's good citizenship to explicitly declare the return type so someone can look at the code and understand what it's doing.

The body of the method `if (User.loggedIn_?) html else NodeSeq.Empty` returns `html`, the child tags of the XHTML in the view. `User.loggedIn_?` is a method on the `User` object. It returns true if a user is logged into the application for the current browser session. In Scala, every statement has a return type. Thus, if/else is like the Java ternary operator. If a user is logged in, we return the child XHTML, otherwise we return an empty sequence of nodes, XHTML. Scala does not require an explicit `return` statement as Scala returns the last expression evaluated in the method.

The out method works similarly, but returns the child nodes if no user is logged in.

2.6 Lift's "View First" design

A philosophical digression into a major architectural decision.

My first design goal with Lift was to make sure that no programming logic and no programming symbols make it into the static display templates.

ERB and JSP and ASP all have the fatal flaw of allowing code in the view. This is bad for a bunch of reasons. First, it makes editing the templates difficult with HTML layout tools unless those tools are familiar with the syntax being used. Second, there are "foreign" symbols in the layout, which tends to be less than optimal for the HTML designers. (On the Rails side of things, every Rails team I've seen has a Ruby coder that also does the design. While this is the norm in the Rails community, it is the exception when team sizes are more than 2 or 3.) Third, every single Rails, JSP, and ASP project I've ever seen (and I've been seeing them for a very long time) has some non-trivial amount of business logic creep into the display. Fourth, Scala has a very nice type system and when the type system is used correctly, the compiler finds a lot of program errors, but when the code is buried in templates, one has a much more difficult time using the powerful Scala compiler tools.

So, the static templates in Lift are strictly for display only. They can be manipulated with standard design tools (e.g., Dreamweaver). They can never contain program logic.

Rails' "controller first" dispatch mechanism makes the assumption that there is only one piece of "logic" on the page and the rest is decoration. My experience doing web work is just the opposite. There are typically 3 or more of pieces of logic on a page (dynamic menu bars, search boxes, shopping cart, real-time chat, etc.) and having to choose which piece of logic make the "controller" is less than optimal.

So, the quintessential use of lift's templates are as follows:

```
<html>
 ...
<lift:Show.myForm form="POST">
  <tr>
    <td>Name</td>
    <td><f:name><input type="text"/></f:name></td>
  </tr>
  <tr>
    <td>Birthyear</td>
    <td><f:year>
         <select><option>2007</option></select>
    </f:year></td>
  </tr>
  <tr>
    <td> </td>
    <td><input type="submit" value="Add"/></td>
  </tr>
</lift:Show.myForm>
</html>
```

So we've got a Lift snippet invocation with the valid HTML form and some additional tags. So far (with the proper name-space declarations) this page is valid XHTML. This page can be viewed in a browser or opened and edited in Dreamweaver.

In Lift, the snippet is the equivalent of a Rails controller: it is the instantiation of a class and invocation of a method on the class. Because you can have multiple snippets on a page, you can

call out multiple logic streams on a given page and there's no need to choose the primary logic stream.

The `form="post"` attribute is a shortcut. It automatically wraps the enclosed XHTML in a `<form method='post' target={current page... it's a post-back}>...</form>` tag.

The `<f:xxx/>` tags are bind points for the business logic. They allow your snippet to easily replace the tag and its children with what is supposed to be displayed.

So, your Lift code will look like:

```
class Show {
  def myForm(xhtml: NodeSeq) = {
    var name = ""
    def handleYear(year: String) {
      ... the form's been submitted... do something
    }
    bind("f", xhtml, "name" -> text(name, name = _),
                 "year" -> select((1900 to 2007).
                        toList.map(_.toString).
                        reverse.map(v => (v, v)),
                        Empty, handleYear _))
  }
}
```

Note that no display code has crept into the snippet. You've simply bound the XHTML created by `text()` and `select()` to the `<f:name/>` and `<f:year/>` tags in the incoming XHTML.

Also, you've bound two functions (the anonymous function `name = _` and `handleYear`) to the XHTML form elements. When the form is posted, these functions (which are bound to local variables) will be statefully invoked.

If you are displaying a table rather than a form, then the same binding logic still works. For example:

```
<table>
  <lift:Show.users>
    <tr>
      <td><f:first_name>David</f:first_name></td>
      <td><f:last_name>Pollak</f:last_name></td>
    </tr>
  </lift:Show.users>
</table>
```

The snippet looks like:

```
class Show {
  def users(xhtml: NodeSeq) =
    Users.findAll.flatMap(user => bind("f",
          xhtml, "first_name" -> user.firstName,
                "last_name" -> user.nameName))
}
```

If you take the time to clearly define the bind points, then you can have no display code at all in your snippets.

Can display logic slip into a snippet? Yes.

Has display logic ever crept into a method called from an ERB template? Yes, and very often it's a source of a potential Cross Site Scripting vulnerability.

Has business logic ever crept into an ERB template? Yes.

In Lift, display can creep into a snippet, but business logic cannot creep into a the static display template. Yes, your designers will still have to police putting display logic in the snippet code, but the coders will not have to police business logic in the templates.

Back to our regularly scheduled code writing.

2.7 Updating the ToDo view

We're going to add view code to our application.

Open `src/main/webapp/index.html` and replace the file with:

Listing 2.6: index.html

```
<lift:surround with="default" at="content">
  <lift:Util.out>Please
    <lift:menu.item name="Login">Log In</lift:menu.item>
    </lift:Util.out>
</lift:surround>
```

`<lift:surround with="default" at="content">` surrounds into child content with a template called `default` and places the content of this file at a bind-point called `content`. Your page can bind to multiple child items. Thus your page can define main content, left side content, footer content, etc. Additionally, templates can be hierarchical and nested. By convention, templates are located in the `src/main/webapp/templates-hidden` directory. Lift will not directly serve request for directories that have `-hidden` in their name.

We're using our `<lift:Util.out/>` tag to display content only if there's no user logged in.

`<lift:menu.item name="Login">Log In</lift:menu.item>` tag accesses some Lift built in snippets to create an `<a href="..."/>` to the page in site navigation called "Login". Thus, your application doesn't need to know the URL of the Login page in order to access it and if site navigation changes, the URL will get updated. Links will not be generated if page access control rules prohibit the user from accessing the page.

2.8 Seeing if the snippet works

Enter `mvn jetty:run` on your terminal. Note in the terminal output that Schemifier has created the `todo` table:

```
INFO - CREATE TABLE todo (priority INTEGER , ...
INFO - ALTER TABLE todo ADD CONSTRAINT todo_PK PRIMARY KEY(id)
INFO - CREATE INDEX todo_owner ON todo ( owner )
```

Please navigate to http://localhost:8080 You should see your application running and a message asking you to log in. Using the navigation on the left side of your browser, please click the "Sign Up" link. Once you've successfully completed the signup process, you'll be returned to the home page. There should no longer be a message prompting you to log in.

So far, we've created a model to hold our to-do items, written a snippet to bridge between the view and business logic, and we've seen how to update our view to access the snippet.

Next, we'll create the view and snippet code that allows us to create a to-do item.

2.9 Creating a to-do item

Create a new snippet in `src/main/scala/com/liftworkshop/snippet/TD.scala` with the following code:

Listing 2.7: TD.scala

```
package com.liftworkshop.snippet

import com.liftworkshop._
import model._

import net.liftweb._
import http._
import SHtml._
import S._

import js._
import JsCmds._

import mapper._

import util._
import Helpers._

import scala.xml.{NodeSeq, Text}

class TD {
  def add(form: NodeSeq) = {
    val todo = ToDo.create.owner(User.currentUser)

    def checkAndSave(): Unit =
    todo.validate match {
      case Nil => todo.save ; S.notice("Added "+todo.desc)
      case xs => S.error(xs) ; S.mapSnippet("TD.add", doBind)
    }

    def doBind(form: NodeSeq) =
    bind("todo", form,
        "priority" -> todo.priority.toForm,
        "desc" -> todo.desc.toForm,
        "submit" -> submit("New", checkAndSave))

    doBind(form)
  }
}
```

There's a single top level method: `def add(form: NodeSeq)`.

`val todo = ToDo.create.owner(User.currentUser)` creates an instance of our `ToDo` model and sets the item's owner to the currently logged in user. This instance will be visible for the lifespan of the creation task, not the lifespan of the HTTP request. This statefullness makes

it easier to write very complex applications in Lift because you don't have to repopulate object values each time you service a request.

`def checkAndSave(): Unit` defines a method. Scala allows you to create methods that live inside of other methods. These inner methods have access to the variables inside the method scope. `checkAndSave` has a return type of `Unit` which is like Java's `void`. Because `checkAndSave` is a recursive method, its return type must be explicitly defined. `todo.validate` checks the validation rules for each field in `todo`. It returns a `List[FieldError]` (pronounced "List of FieldError"). We `match` the list against a pattern. If the list is `Nil` (it has no elements), save the `todo` instance and display a notice telling the user that the item was saved. If we have validation failures, display those as errors. `S.mapSnippet("TD.add", doBind)` tells Lift that during the current HTTP request servicing, when the `TD.add` snippet is requested, call the `doBind` method rather than looking up the snippet in the class/method manner. This means that the current state (the current `todo` instance) will be preserved. Yep, it's kinda a weird way to think. Please continue reading and it may make more sense.

`def doBind(form: NodeSeq)` defines a method that takes XHTML and binds it to business logic. `"priority" -> todo.priority.toForm` binds the `<todo:priority/>` tag to a form element, in this case a `<select/>`, that the priority field generates. `"desc" -> todo.desc.toForm` does the same for the `desc` field. `"submit" -> submit("New", checkAndSave)` creates a submit button with the value "New". When the form is processed, the `checkAndSave` method will be called.

In Lift, you don't have to explicitly name HTML form elements. You don't have to explicitly look for parameters of a POST or a GET. Lift takes care of this work for you. You specify the type of the form element, the default value of that element and the chunk of code, the function, to invoke when the form is submitted. Lift takes care of managing the form field names and the state for you. This has a bunch of benefits. First, you don't have to coordinate form field names across your team. Second, only the form fields presented to users can be submitted back which avoids form tampering. Third, form field names are randomly generated making it very difficult for a hacker to engage in replay attacks. If you want, you can dive down to the HTTP bare metal and get the parameters. If you want, you can name your own HTML form elements. But Lift lets you abstract away the HTTP request/response cycle.

In this code, `todo.priority` and `todo.desc` have taken care of generating their own form elements. When the form is submitted, those fields will be updated in the instance of `todo` created at the top of the method. The instance of `todo` used in `checkAndSave` is the same instance that have just its fields updated. If validation fails, we map the `TD.add` snippet to an instance of `doBind` (the `doBind` method is turned into a function instance which is an object that is bound to the current variable scope) that refers to the same instance of `todo`. If your head is spinning, don't worry.

2.10 Updating the priority and desc fields

Please open the `ToDo.scala` file in the model package.

Update the `priority` object so that it looks like:

Listing 2.8: Update ToDo.scala

```
object priority extends MappedInt(this) {
  override def defaultValue = 5
```

```
   override def validations = validPriority _ :: super.validations

   def validPriority(in: Int): List[FieldError] =
   if (in > 0 && in <= 10) Nil
   else List(FieldError(this, <b>Priority must be 1-10</b>))

   override def _toForm = Full(select(ToDo.priorityList,
                               Full(is.toString),
                               f => set(f.toInt)))
 }
```

`override def validations = validPriority _ :: super.validations` adds the `validatePriority` function to the list of validation functions for this field. The validation functions are consulted for each field during validation. They return either an empty list, `Nil`, indicating no validation problems, or they return a list of errors.

`def validPriority(in: Int): List[FieldError]` defines the method that validates the priority. `if (in > 0 && in <= 10) Nil` if the priority if valid, return no errors. Otherwise, return a list of one error: `else List(FieldError(this, <b>Priority must be 1-10</b>))`.

`override def _toForm` override the default form generation for the `priority` field. `select(ToDo.priorityList,` create an HTML `<select/>` with a list of values. Set the default value to the current priority field value (is), converted to a String: `Full(is.toString)`. When the form is submitted, call this function: `f => set(f.toInt)`. The function takes a single parameter, `f`. The compiler infers the type of `f` to be `String` by looking at the signature of the `select()` method. `f.toInt` converts the `String` to an `Integer` and the `set()` method is called on the `priority` field. We can safely call `toInt` because our function will only be called with one of the values originally supplied to the `select()` function. Lift will not call your function with another value, so you can be sure that if someone tampers with the form parameters, your application will not be called with a bogus value.

Let's update the `desc` field to add validators:

Listing 2.9: Update ToDo.scala

```
object desc extends MappedPoliteString(this, 128) {
   override def validations =
     valMinLen(3, "Description must be 3 characters") _ ::
       super.validations
 }
```

In this case, we're using the `valMinLen()` validator method built into Lift's `MappedString`, the superclass of `MappedPoliteString`. By default, `MappedString` generates an `<input type="text"/>` form element, so we don't have to explicitly define a `_toForm` method.

Finally, let's add the `priorityList` helper method to our `ToDo` object:

Listing 2.10: Update ToDo.scala

```
object ToDo extends ToDo with LongKeyedMetaMapper[ToDo] {
  lazy val priorityList = (1 to 10).
      map(v => (v.toString, v.toString))
}
```

A `lazy val` is calculated once, the first time it's accessed. `(1 to 10)` creates a Range of

numbers from 1 to 10, inclusive. `map` passes each number to a function which converts the `Integer` to a `Pair[String, String]` (pronounced Pair of Strings). This value has been calculated once and can be used by any `ToDo.priority` for form generation.

2.11 Hooking up the view to the form

Open `src/main/webapp/index.html`

Inside the `<lift:surround>....</lift:surround>` tag, insert these tags:

Listing 2.11: Insert into index.html

```
<lift:Util.in>
  <lift:TD.add form="post">
    <table>
      <tr>
        <td>Description:</td>
        <td><todo:desc>To Do</todo:desc></td>
      </tr>

      <tr>
        <td>
          Priority
        </td>
        <td>
          <todo:priority>
            <select><option>1</option></select>
          </todo:priority>
       </td>
      </tr>
      <tr>
        <td> </td>
        <td>
          <todo:submit>
            <button>New</button>
          </todo:submit>
        </td>
      </tr>
    </table>
  </lift:TD.add>
</lift:Util.in>
```

Inside the `<lift:Util.in/>` snippet tag, we have the `<lift:TD.add form='post'/>` snippet tag. This tag invokes the `TD.add` snippet which binds HTML form elements to the `<todo:priority/>`, `<todo:desc/>`, and `<todo:submit/>` elements. Because we've specified `form='post'` attribute, Lift creates a post-back form and places it around the snippet. Lift processes snippets outside-in. Thus, the `<lift:TD.add/>` snippet tag will only be processed if it is returned from the `<lift:Util.in/>` tag.

2.12 Trying it out again

At your terminal, stop the running todo app if it's still running. Type `mvn jetty:run` and point your browser to `http://localhost:8080` and then log in using the username/password you created when you tested the app last time around. Once you log in, you should see the form that lets you add a to-do item. It's nice to have a "write only" application. Next, we'll let you view and edit to-do items.

2.13 Display and Editting to-do items

Now that we've created a mechanism to add to-do items, it's time to display those items and allow Ajax-style editting of the items.

First, let's define the view. Insert the following code inside the `<lift:Util.in/>` tag:

Listing 2.12: index.html inside <lift:Util.in/>

```
<lift:TD.list all_id="all_todos">
  <div id="all_todos">
    <div>Exclude done <todo:exclude/></div>
    <ul>
      <todo:list>
        <li>
          <todo:check><input type="checkbox"/></todo:check>
          <todo:priority>
            <select><option>1</option></select>
          </todo:priority>
          <todo:desc>To Do</todo:desc>
        </li>
      </todo:list>
    </ul>
  </div>
</lift:TD.list>
```

Some things of note in the above code. Our snippet tag, `<lift:TD.list all_id="all_todos">`, has an `all_id` attribute. This will pass information, the name of the `id` attribute, the the snippet. We can see the `id` attribute on `<div id="all_todos">`. This defines the section of the HTML that we're going to change each time we service an Ajax request. Finally, `<todo:list>` has a single child `<li>` which also has bind points. Our snippet will iterate over the list of to-do items, binding each to the children of `<todo:list/>`.

At the top level (*not* inside the `class TD {...}` definition) insert this code:

Listing 2.13: TD.scala define object

```
object QueryNotDone extends SessionVar(false)
```

Lift has a type-safe mechanism for storing variables during the scope of a given session. In Java, you would do something like casting a value retrieved from the `HttpSession`. For example: `String myString = (String) httpSession.getAttribute("Something");` The problem with this code is that a developer may not remember that `Something` is support to be a `String` and put a `String[]` in it. Additionally, there's no well defined logic for creating a default value for the attribute.

Lift's `SessionVar` is type-safe. You cannot put anything other than it's defined type into it and when something comes out of it, it's the same type that was put in. A `String` is always a `String` and a `Boolean` is always a `Boolean`. Additionally, the `SessionVar` has a default value. If the `SessionVar` accessed, but it is not defined, it will calculate its default value and set itself to the default value.

The above code defines a `SessionVar` named `QueryNotDone` with a default value of `false`. We do not need to explicitly define that type as Boolean as Scala infers that from the default value.

If `QueryNotDone` is `true`, only the incomplete to-do items will be displayed.

Let's define the query that will retrieve the ToDo rows from the database. Insert this method into the body of the TD class in TD.scala:

Listing 2.14: toShow method on the TD class

```
private def toShow =
  ToDo.findAll(By(ToDo.owner, User.currentUser),
               if (QueryNotDone) By(ToDo.done, false)
               else Ignore[ToDo],
               OrderBy(ToDo.done, Ascending),
               OrderBy(ToDo.priority, Descending),
               OrderBy(ToDo.desc, Ascending))
```

We're defining the `toShow` method with returns a `List[ToDo]`. This method calls the `findAll()` method on the `ToDo` object with a bunch of query parameters. `By(ToDo.owner, User.currentUser)` tells the query to only return the rows where `owner` equals the `User` who is currently logged in. Next, we test `QueryNotDone`. If it's `true`, we add an additional constraint: `done` equals `false`, otherwise we have a dummy query parameter: `Ignore`. Finally, we define how the to-do items should be ordered.

Next, we add a method to the TD class that will render the description field:

Listing 2.15: desc method in TD class

```
private def desc(td: ToDo, reDraw: () => JsCmd) =
  swappable(<span>{td.desc}</span>,
          <span>{ajaxText(td.desc,
                          v => {td.desc(v).save; reDraw()})}
          </span>)
```

We define a private method, `desc`. It takes a `ToDo` instance and a function, `reDraw`, that will redisplay the list of to-do items. It's a function that takes no parameters, but returns a `JsCmd`. A `JsCmd` is a JavaScript command that the browser can execute. This is the JavaScript that will be sent back to the browser in response to an Ajax request.

`swappable` creates HTML that displays the first parameter, a `<span/>` containing the `ToDo`'s `desc`. When user clicks on the swappable, it swaps to display the second parameter, a `<span/>` containing an `ajaxText` field. The user can edit the `ajaxText` field and when the field loses focus or the user presses `Enter`, the updated text is sent back to the server via an Ajax call and the `swappable` swaps itself back to displaying the first parameter.

When the server receives the Ajax call, it passes the parameter to the function: `v => {td.desc(v).save; reDraw()}`, which sets the `desc` field and saves the `ToDo` instance and then calls reDraw to update the browser. The user sees the updated list of to-do items.

Next, add this method to the TD class:

Listing 2.16: doList method in TD class

```
private def doList(reDraw: () => JsCmd)(html: NodeSeq): NodeSeq =
 toShow.
 flatMap(td =>
  bind("todo", html,
      "check" -> ajaxCheckbox(td.done,
              v => {td.done(v).save; reDraw()}),
      "priority" ->
      ajaxSelect(ToDo.priorityList, Full(td.priority.toString),
              v => {td.priority(v.toInt).save; reDraw()}),
      "desc" -> desc(td, reDraw)
  ))
```

We're defining the `doList` method which takes two parameters: `reDraw` and `html`. The parameter list is split into two lists to allow the method to be *curried*. This means we can create a function that has the first parameter supplied, but it does not have the second parameter supplied. We'll see more of this in a minute.

This method calls `toShow` to get the list of to-do items to show. Using the `flatMap` method, we iterate over each to-do item and `bind`ing the item to the incoming html. We bind `<todo:check/>` to an `ajaxCheckbox`. An `ajaxCheckbox` generates the HTML to display a checkbox. When the checkbox is toggled, an Ajax request is made on the server and Lift forward the new value of the checkbox to the function. The function, `v => {td.done(v).save; reDraw()}`, updates the to-do item's done field and saves the to-do item, then it calls `reDraw` to update the browser.

The same thing is done with the `<todo:priority/>` and `<todo:desc/>` tags.

Finally, we add the list method to the TD class:

Listing 2.17: list method in TD class

```
def list(html: NodeSeq) = {
 val id = S.attr("all_id").open_!

 def inner(): NodeSeq = {
  def reDraw() = SetHtml(id, inner())

  bind("todo", html,
      "exclude" ->
       ajaxCheckbox(QueryNotDone, v => {QueryNotDone(v); reDraw}),
      "list" -> doList(reDraw) _)
 }

 inner()
}
```

Because the list method is public, Lift will call it when Lift encounters the `<lift:TD.list/>` tag in the view code.

`val id = S.attr("all_id").open_!` retrieves the attribute `all_id` from the snippet tag and stores it in the `id` variable.

We define the `inner` method. This method is bound to the variables in `list`'s scope.

Inside `inner`, we define the `reDraw` method which creates an `SetHtml JsCmd` with the `NodeSeq` generated by calling the `inner` method. `SetHtml` will set the HTML children to a new value. In this case, it's the updated HTML based on redrawing all the to-do items.

The `bind()` call binds the view template to an ajaxCheckbox and the list of our to-do items.

Well... that's all the code we need to write. Time to play with our new application.

At the terminate, type: `mvn clean jetty:run`. The clean part of the command tells Maven to remove any previously compiled files. It's best practices to-do a `clean` every couple of compile cycles.

Point your browser at `http://localhost:8080` and you'll see your application running. Log in. You'll see any to-do items that you created in "write only" mode. You can mark to-do items done and the list instantly updates. You can click on the description of any item and it'll swap to an editable field. If you log out and log in as a different user, you'll have a fresh set of to-do items.