Contents of Appendixes

Appendix A: A Brief Tour of Maven

In this appendix, we'll discuss the Maven build tool and some of the basics of configuration and usage. Lift uses Maven for build management, so becoming acquainted with Maven is important to getting the most out of Lift. If you're already familiar with Maven, you can safely skip this appendix.

Introducing Maven

Maven is a project management tool, not simply a build tool. The Maven web site (`http://maven.apache.org/`) describes the goals of Maven as

- Making the build process easy
- Providing a uniform build system
- Providing quality project information
- Providing guidelines for best practices
- Allowing transparent migration to new features

As a project management tool, Maven goes beyond just controlling compilation of your code. By default, Maven comes equipped not only to perform development-centric tasks but to generate documentation from your code and for your project web site.

Everything in Maven is controlled via the `pom.xml` file, which contains both information and configuration details on the project. We'll be covering some of the basic aspects of the Project Object Model (POM) through the rest of this appendix. A complete POM reference is available at `http://maven.apache.org/pom.html`.

Understanding Life Cycles, Phases, and Goals

Maven is designed around the concept of project life cycles. While you can define your own life cycles, three types are built in: `default`, `clean`, and `site`. The `default` life cycle builds and deploys your project. The `clean` life cycle cleans (deletes) compiled objects or anything else that needs to be removed or reset to get the project to a pristine prebuild state. Finally, the `site` life cycle generates the project documentation.

Within each life cycle, a number of phases define various points in the development process. The most interesting life cycle (from the perspective of writing code) is `default`. At `http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html`, you

can find a full listing of life cycles and their phases. The most commonly used phases in the default life cycle follow:

- `compile`: This phase compiles the main source code of the project.
- `test`: This phase tests the main code using a suitable unit testing framework. These tests should not require that the code is packaged or deployed. This phase implicitly calls the `testCompile` goal to compile the test case source code.
- `package`: At this phase, Maven packages the compiled code into its distributable format, such as a JAR. The POM controls how a project is packaged through the `<packaging/>` element
- `install`: In this phase, you install the package into the local repository (see the "Working with Repositories" section), for use as a dependency in other projects locally.
- `deploy`: This phase is used in an integration or release environment. Copies the final package to the remote repository for sharing with other developers and projects.

Maven is typically run from the command line by executing command `mvn <phase>`, where `<phase>` is one of the phases listed previously. Most major IDEs have plug-ins for Maven as well.

Since phases are defined in order, all phases up to the one you specify will be run. For example, if you want to package your code, simply run `mvn package`, and the `compile` and `test` phases will automatically be run.

You can also execute specific goals for the various plug-ins that Maven uses. Execution of a specific goal is done with the command `mvn <plugin>:<goal>`.

For instance, the `compile` phase actually calls the `compiler:compile` goal by default. A common usage of executing a goal for Lift is the `jetty:run` goal, which compiles all of your code and then runs an instance of the Jetty web server (`http://www.mortbay.org/jetty/`) so that you can exercise your application. The Jetty plug-in is not directly bound to any life cycle or phase, so we have to execute the goal directly.

One final note is that you can specify multiple phases or goals in one command line, and Maven will execute them in order. This is useful if, for instance, you want to do a clean build of your project. Simply run `mvn clean jetty:run`, and the `clean` life cycle will run, followed by the `jetty:run` goal (and all of the prerequisites for `jetty:run`, such as `compile`).

Working with Repositories

Repositories are one of the key features of Maven. A repository is a location that contains plug-ins and packages for your project to use. There are two types of repository: local and remote. Your local repository is, as the name suggests, local to your machine and represents a cache of

artifacts downloaded from remote repositories as well as packages that you've installed from your own projects. The default locations of your local repository will be as follows:

- *Unix*: `~/.m2/repository`
- *Windows*: `C:\Documents and Settings\<user>\.m2\repository`

You can override the local repository location by setting the `M2_REPO` environment variable or by editing the `<home>/.m2/settings.xml` file. More details on customizing your Maven installation are available at `http://maven.apache.org/settings.html`.

Remote repositories are repositories that are reachable via protocols like HTTP and FTP and are generally where you will find the dependencies needed for your projects. Repositories are defined in the POM.

Listing A-1 shows the definition of the `scala-tools.org` release repository where Lift and many other Scala-related packages are found. The `scala-tools.org` site also has a snapshots repository where nightly builds of the `scala-tools` projects are kept. Maven has an internal default set of repositories, so you don't usually need to define too many extras.

Listing A-1. Defining a Repository

```
<repositories>
  <repository>
    <id>scala-tools.org</id>
    <name>Scala Tools Maven2 Repository</name>
    <url>http://scala-tools.org/repo-releases</url>
  </repository>
</repositories>
```

Also note that, sometimes, you may not have Internet access or the remote repositories will be offline for some reason. In this case, make sure to specify the `-o` (offline) flag so that Maven skips checking the remote repositories.

Adding Plug-ins

Plug-ins add functionality to the Maven build system. Lift is written in Scala, so the first plug-in that we need to add is the Maven Scala plug-in, which provides the ability to compile Scala code in your project. Listing A-2 shows how we configure the plug-in in the `pom.xml` file for a Lift application. You can see the Scala plug-in adds a `compile` and `testCompile` goal for the `build` phase, which makes Maven execute this plug-in when those goals are called (explicitly or implicitly). In addition, the configuration element allows you to set properties of the plug-in executions; in this case, we're explicitly specifying the version of Scala that should be used for compilation.

Listing A-2. Configuring the Maven Scala Plug-in

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

Dealing with Dependencies

Dependency management is one of the more useful features of Maven. Listing A-3 shows a declaration of the Jetty dependency for the default Lift application. The details of the specification are straightforward:

- The `groupId` and `artifactId` specify the artifact. A given group may have many artifacts under it; for instance, Lift uses `net.liftweb` for its `groupId` and the core artifacts are `lift-core` and `life-util`.
- The version is specified either directly or with a range, as we've used in this example. A range is defined as `<left>min,max<right>` where `left` and `right` indicate an inclusive or exclusive range: square brackets, `[]`, are inclusive, and parentheses, `()`, are exclusive. Omitting a version in a range leaves that portion of the range unbounded. Here, we configure the `pom.xml` file so that Jetty 6.1.6 or higher is used.

The scope of the dependency is optional and controls exactly where the dependency is used. If you want to learn more about scope, it's discussed in detail at `http://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html`. In this case, we specify a test scope, which means that the package will only be available to test phases.

Listing A-3. Adding a Dependency

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

As an example, let's say that you'd like to add a new library, and you want Maven to make sure you've got the most up-to-date version. We're going to add Configgy (`http://www.lag.net/configgy/`) as a dependency. Configgy is a library that handles simple logging and configuration so that you don't need to add a whole lot of support code. In particular, it excels in the scope of small applications, which is what we're doing here. For larger applications, you probably want to go ahead and use more heavyweight logging and configuration facilities like those provided by log4j, SLF4J, Java EE, and others.

First, we need to tell Maven where we can get Configgy, so in the `<repositories>` section add the code in Listing A-4.

Listing A-4. Adding the Configgy Repository

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

Then, in the `<dependencies>` section, add the code in Listing A-5.

Listing A-5. Adding the Configgy Dependency

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

That's it; you're done. The next time you run Maven for your project, it will pull down the Configgy JARs into your local repository. Maven will periodically check for new versions of dependencies when you build, but you can always force a check with the `-U` (update) flag.

Finding Additional Maven Resources

Obviously, we've only scratched the surface on what you can with Maven and how to configure it. If you're looking for more information, we've found the following set of references useful in learning and using Maven:

- `http://maven.apache.org`: The Maven home page
- `http://maven.apache.org/what-is-maven.html`: A brief description of Maven's goals
- `http://maven.apache.org/guides/introduction/introduction-to-the-pom.html`: An introduction to the `pom.xml` file
- `http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html`: An overview of the life cycles
- `http://suereth.blogspot.com/2008/10/maven-for-beginners.html`: A brief Maven usage tutorial
- `http://scala-blogs.org/2008/01/maven-for-scala.html`: A brief tutorial on using Maven geared toward Scala
- `http://mvnrepository.com/`: A web site that lets you search for Maven dependencies by name, which is invaluable when you're trying to add libraries to your project

Laying Out a Project

One of the things that allow Maven to work so well is that there is a standardized layout for projects. We're not going to cover of the standard locations for all of the parts of your Maven project, but we do want to highlight a few locations that are important to Lift applications specifically:

- `<application_root>/src/main/scala`: This directory is where you place your Scala source, such as snippets, model objects, and any libraries you write. The subfolder structure follows the traditional Java packaging style.
- `<application_root>/src/main/resources`: This directory is where you would place any resources that you want to go into the WAR file. Typically, it is used if you want to add entries to the `META-INF` directory in the WAR, since normal web resources should be placed under the `webapp/WEB-INF` directory.
- `<application_root>/src/main/webapp`: All of the web and static content for your application, including images, XHTML templates, JavaScript, and CSS, is placed under this directory. This is also the location for your `WEB-INF` directory (and the configuration files it contains). This directory is essentially what is packaged into the WAR in addition to the output from your Scala sources.

- `<application_root>/src/main/webapp/templates-hidden`: This is a special location for templates. As we discussed in Chapter 3, templates placed in this directory cannot be viewed directly by clients but are available to other templates.
- `<application_root>/src/test/scala`: This directory is where you can put all of your test code. As with `src/main/scala`, the subfolder structure follows the traditional Java packaging style

Appendix B: Message Handling

When we talk about message handling in Lift, we're talking about how you provide feedback to the users of your application. While there are already a lot of mechanisms for displaying data to the user via snippets, views, and so on, properly binding and setting up HTML-level elements can get complicated, especially when you're dealing with callback functions or error handling. Lift provides an alternate mechanism for displaying messages to users that is easy to use and allows flexibility in display on the client side.

Sending Messages

Messages for non-Comet requests are handled via the `S` object (yes, even AJAX is handled automatically); specifically, the error, notice and warning methods allow you to send a `String` or a `NodeSeq` back to the user for display, with or without an association with a particular element ID. The `error` method also provides an overload that takes a `List[FieldError]`, the type returned from Mapper field validation (see Chapter 6). The messages that you send are held by a `RequestVar` (see Chapter 3) in the `S` object, so you can send messages from anywhere in your stateful request/response life cycle without breaking the flow of your code. Listing B-1 shows how you could use messages in form processing to send feedback on missing fields.

Listing B-1. Using Messages in Form Processing

```
object data extends RequestVar[String]("")

def addNote (xhtml : NodeSeq) : NodeSeq = {
  def doAdd () = {
  //validate
  if (data.is == "") {
    S.error("noteField", "You need to provide a note")
  } else {
    Note.create.note(data).save
    S.notice("Note added")
    redirectTo("/viewNotes")
  }
  }
  bind("form", xhtml,
       "note" -> SHtml.text(data.is, data(_),
                            "id" -> "noteField"),
       "add" -> SHtml.submit("Add", doAdd))
}
```

In this particular case, we use two different messages. One is an error to be displayed when the form is reshown; this error is associated with the `noteField` element. The second message is a simple notice to let the user know that the data was successfully saved.

For Comet, the only difference in sending messages is that the `error`, `notice`, and `warning` methods are defined in the `CometActor` class, so you just use those directly, and Lift handles the rest.

Displaying Messages

The display of messages is handled by two built-in snippets, `<lift:Msgs/>` and `<lift:Msg/>`. The `Msgs1` snippet displays all messages not associated with a particular element `id`. The messages are displayed as an unordered list, but Lift allows customization of the messages via XML that you embed within the snippet. For each of the three message types, you can specify a `<lift:TYPE_msg>` and `<lift:TYPE_class>` element that controls the message label and CSS class, respectively. The default label is simply the title-case type (`Error`, `Notice`, and `Warning`). For example, Listing B-2 shows how we could change the error and notice messages.

Listing B-2. Custom Message Labels

```
<lift:msgs>
  <lift:error_msg>
    Danger, Will Robinson!
  </lift:error_msg>
  <lift:error_class>redtext</lift:error_class>
  <lift:notice_msg>FYI: </lift:notice_msg>
</lift:msgs>
```

The `Msg` snippet (`net.liftweb.builtin.snippet.Msgs`) is used to display all messages associated with a particular `Id` by specifying the `id` attribute on the `<lift:Msg/>` element. With `Msg`, you don't get a message label, so there's no override mechanism for it. You do, however, have the ability to change the message class on a per-type basis by setting the `noticeClass`, `errorClass`, or `warningClass` attributes on the `<lfit:Msg/>` element. Listing B-3 shows usage of `Msg` corresponding to our snippet in Listing B-1.

Listing B-3. Per-ID Messages

```
<lift:Stuff.addNote form="POST">
  <form:note />
    <lift:msg id="noteField" errorClass="redtext" />
  <form:add />
</lift:Stuff.addNote>
```

In Listing B-3, we are associating an id to the `<lift:msg>` tag. What this means is that only errors, warnings, or notices associated with this particular `id` will be rendered. To associate a message with an `id`, you only need to call the `S.error`, `S.warning`, or `S.notice` functions and pass the `id` referred by `<lift:msg>`.

Appendix C: Lift Helpers

Lift provides a fairly useful collection of helper artifacts. The helpers are essentially utility functions that minimize the need for boilerplate code. This appendix is intended to introduce some of the more common utility classes and objects to you so that you're familiar with them. If you would like more details, you can look at the API documentation for the `net.liftweb.util` package.

Using Box

`Box` (or Scala's `Option` class on steroids) is a utility class that mimics Scala's `Option` type (also heavily used inside Lift). To understand some of the underlying concepts and assumptions, let's take a quick look at `Option` class first. The `Option` class allows a type-safe way of dealing with a situation where you may or may not have a result. `Option` has two values: `Some(value)`, where `value` is actually the value, and `None`, which is used to represent nothing. A typical example for `Option` is outlined using Scala's `Map` type. Listing C-1 shows a definition of a `Map`, a successful attempt to get the value of key `a`, and an attempt to get the value of key `i`. Notice that when we retrieved the existing key-value pair for `a`, the value returned was `Some(A)`, and when we asked for the value of key `i`, we received `None`.

Listing C-1. An Option and Map Example

```
scala> val cap = Map("a" -> "A", "b" -> "B")
cap: scala.collection.immutable.Map[java.lang.String,java.lang.String] =
  Map(a -> A, b -> B)

scala> cap.get("a")
res1: Option[java.lang.String] = Some(A)

scala> cap.get("i")
res2: Option[java.lang.String] = None
```

Getting the value out of an `Option` is usually handled via Scala's matching mechanism or via the `getOrElse` function, as shown in Listing C-2.

Listing C-2. Fetching a Value from an Option

```
def prettyPrint(foo: Option[String]): String = foo match {
  case Some(x) => x
  case None => "Nothing found."
}
// or
def prettyPrint(foo: Option[String]): String =
  foo.getOrElse("Nothing found.")

// Usage:
scala> prettyPrint(cap.get("a"))
res7: String = A

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

`Box`, in Lift, covers the same base functionality as `Option` but expands the semantics for missing values. If we have an `Option` that is `None` at some point, we can't really tell why that `Option` is `None`, although in many situations, knowing why would be quite helpful. With `Box`, on the other hand, you have either have a `Full` instance (corresponding to `Some` with `Option`) or an instance that subclasses `EmptyBox` (corresponding to `None`). `EmptyBox` can either be an `Empty` instance or a `Failure` instance incorporating the cause for the failure. So you can think of `Box` as a container with three states: full, empty, or empty for a particular reason. The `Failure` case class takes three arguments: a `String` message to describe the failure, a `Box[Throwable]` for an optional exception related to the failure, and a `Box[Failure]` for chaining based on earlier `Failures`.

As an example of how we can use `Box` instances in real code, consider the case where we have to do a bunch of null checks, perform an operation, and then perform more null checks, other operations, and so on. The following pseudocode is an example of this sort of structure:

```
val x = getSomeValue();
if (x != null) {
  val y = getSomeOtherValue();
  if (y != null) {
    compute(x, y);
  }
}
```

This is tedious and error-prone in practice. Now, let's see if we can do better by combining Lift's `Box` with Scala's for comprehensions, as shown in Listing C-3.

Listing C-3. A Nested Operations Example Using Box

```
def getSomeValue(): Box[Int] = ...
def getSomeOtherValue(): Box[Int] = ...
def compute(x: Int, y: Int) = x * y

val res =
  for (x <- getSomeValue();
       y <- getSomeOtherValue() if x > 10)
    yield compute(x, y)
println(res)
```

In the Listing C-3, we have two values, `x` and `y`, and we want to do some computation with these values. But we must ensure that computation is done on the correct data. For instance, the computation cannot be done if `getSomeValue()` returns no value. In this context, the two functions return a `Box[Int]`. The interesting part is that if either or both of the two functions return an `Empty Box` instead of `Full` (`Empty` impersonating the nonexistence of the value), the `res` value will also be `Empty`. However, if both functions return a `Full` (like in Listing C-3), the computation is called. For example, if the two functions return `Full(12)` and `Full(2)`, `res` will be a `Full(24)`.

But we have something else interesting here: the `if x > 10` statement (this is called a "guard" in Scala). If the call to `getSomeValue` returns a value less than or equal to 10, the `y` variable won't be initialized, and the `res` value will be `Empty`. This is just a taste of some of the power of using `Box` for comprehensions; for more details on for comprehensions, see *The Scala Language Specification*, section 6.19, or one of the many Scala books available.

Lift's `Box` extends `Option` with a few ideas, mainly the fact that you can add a message about why a `Box` is `Empty`. `Empty` corresponds to `Option`'s `None` and `Full` to `Option`'s `Some`. So you can pattern match against a `Box` as shown in Listing C-4.

Listing C-4. A Pattern-Matching Box Example

```
a match {
  Full(author) => Text("I found the author " +
                  author.niceName)
  Empty => Text("No author by that name.")
           Failure(message, _, _) => Text("Nothing found
                due to " + message)
           // message may be something like "Database
           // disconnected."
}
def confirmDelete {
  (for (val id <- param("id");     // get the ID
        val user <- User.find(id)) // find the user
  yield {
    user.delete_!
    notice("User deleted")
    redirectTo("/simple/index.html")
  }) getOrElse {error("User not found");
  redirectTo("/simple/index.html")}
}
```

In conjunction with Listing C-4, we can use other `Box` functions, such as the `openOr` function shown in Listing C-5.

Listing C-5. An openOr Example

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

We won't be detailing all of the `Box` functions here, but a few words on the most common functions might be beneficial:

- `openOr`: Return the value contained by this `Box` if it's `Full` or the value of the given parameter if the `Box` is `Empty`.

```
myBox openOr "The box is Empty"
```

- `map`: Apply a function on the values inside this `Box`, and return a `Box` with the result. An `Empty Box` stays `Empty`.

```
myBox map (value => value + " suffix")
```

- `dmap`: Apply a function on the value inside this `Box`, and return the function value. If the `Box` is `Empty`, return a default value. The function `dmap(default_value)(x)` is equivalent with `map(x) openOr default_value`.

```
myBox dmap("default")(value => value + " suffix")
```

- `!!`: If the argument is null, return `Empty`; otherwise, return `Full` and the argument's value. Note that this function pertains to `Box` objects.

```
Box !! (<a reference>)
```

- `?~`: Transform an `Empty` to a `Failure`, and pass a message. If the `Box` is a `Full`, it will just return itself.

```
myBox ?~ ("Error message")
```

- `isDefined`: Return `true` if this `Box` contains a value.

```
myBox isDefined
```

- `isEmpty`: Return `true` if this `Box` is `Empty`.

```
myBox isEmpty
```

- `asA[B]`: Return `Full[B]` if the content of this `Box` can be converted to type `B`; otherwise, return `Empty`.

```
myBox asA[Person]
```

- `isA[B]`: Return `Full[B]` if this `Box` contains an instance of class `B`; otherwise, `return Empty`.

```
myBox isA[Person]
```

Note that `Box` contains a set of implicit conversion functions from and to `Option` and `Iterable`.

Remember that `Box` is heavily used in Lift, as most of Lift's API operates with `Boxes`. The rationale is to avoid null references and to operate safely in context where values may be missing. Of course, programmatically, a variable of type `Box` can be set to `null`, but we strongly recommend against doing so. There are cases, however, where you may be using third-party Java libraries with APIs that return `null` values. To cope with such cases in Lift, you can use the `!!` function to `Box` that value. For instance, Listing C-6 shows how we can deal with a possible null value.

Listing C-6. A Null Example

```
var x = getSomeValueThatMayBeNull();
var boxified = Box !! x
```

In this case, the `boxified` variable will be `Empty` if `x` is `null` or `Full` if `x` is a valid value or reference.

Using ActorPing

The `ActorPing` object provides convenient functionality to schedule sending messages to Actors. Listing C-7 shows some examples of scheduling messages.

Listing C-7. An ActorPing Example

```
// Assume myActor an existing Actor
// And a case object MyMessage
// Send the MyMessage message after 15 seconds
ActorPing.schedule(myActor, MyMessage, 15 seconds)
// Send the MyMessage message every 15 seconds. The
// cycle is stopped if recipient actor exits or replied
// back with UnSchedule message
ActorPing.scheduleAtFixedRate(myActor, MyMessage, 0 seconds, 15 secods)
```

Using ClassHelpers

The `ClassHelpers` object provides convenient functions for loading classes using Java reflection, instantiating *dynamically* loaded classes, invoking methods via reflection, and more. Listing C-8 shows an example of locating a class and then invoking a method on it.

Listing C-8. A ClassHelper Example

```
import _root_.net.liftweb.util.Helpers._
// lookup the class Bar in the three packages specified
// in the list
findClass("Bar", "com.foo" :: "com.bar" :: "com.baz" :: Nil)
invokeMethod(myClass, myInstance, "doSomething")
```

Performing Monadic Conversions

The `MonadicCondition` trait, combined with the `MonadicConversions` object, provides a convenient way of chaining together Boolean expressions along with an associated message. For instance, Listing C-9 shows how we might attempt to define a combined Boolean condition.

Listing C-9. A Simple Boolean Example

```
val isTooYoung = false;
val isTooBig = false;
val isTooLazy = true;
val isAGoodFit = isTooYoung && isTooBig && isTooLazy
```

As you can see, we have no way of telling if the `isAGoodFit` `val` was false because of `isTooYoung`, `isTooBig`, or `isTooLazy` unless we test them again. The `MonadicConversions` object contains implicit conversions to and from a `MonadicCondition` trait, which can be used to chain the conditions as shown in Listing C-10.

Listing C-10. A Monadic Example

```
import net.liftweb.util._
import net.liftweb.util.MonadicConversions._
val isAGoodFit =
  (isTooYoung ~ "too young") &&
  (isTooBad ~ "too bad") &&
  (isToLazy ~ "too lazy")
println(isAGoodFit match {
  case False(msgs) => msgs mkString("Test failed
                    because it is '", "' and '", "'.")
  case _ => "success"
})
```

Now, if `isAGoodFit` (a subclass of `MonadicCondition`) is a `False` instance, we can tell why it failed as we have the messages now.

Using ControlHelpers

The `ControlHelpers` object provides convenient functionality for `try`/`catch` situations via the `tryo` method. Basically, the `tryo` method wraps code that would return an arbitrary type `T` and returns a `Box[T]` indicating whether or not the code threw any exceptions. Listing C-11 shows some examples of how you can use `tryo`.

Listing C-11. A ControlHelpers Example

```
val thumbnail =
  tryo {
    java.awt.Toolkit.getDefaultTookit.getImage(...)
  } match {
    case Full(image) => image
    case Failure(message, _, _) => {
      Log.error("Could not load thumbnail: " + message)
      defaultThumbnail
    }
  }
// or, you can use a callback function instead:
val thumbnail =
  tryo(ex => Log.error("Error loading thumbnail: " + ex) {
    java.awt.Toolkit.getDefaultTookit.getImage(...)
  } openOr defaultThumbnail
//  You can also ignore certain exceptions:
val thumbnail =
  tryo(classOf[java.io.FileNotFoundException]) {
    java.awt.Toolkit.getDefaultTookit.getImage(...)
  } match {
    ...
  }
```

Fixing CSS URLs

When using CSS, it's common to want to embed URLs for images, as shown in Listing C-12. The issue with this is that an absolute URL won't match up with the application URL unless you're deploying in the root context path (`/`).

Listing C-12. An Example CSS Fragment

```
.boxStyle {
  background-image: url('/img/bkg.png')
}
```

Lift provides an easy way to remedy this using the `LiftRules.fixCSS` method to instruct Lift to automatically fix any URLs in the specified CSS files, as shown in Listing C-13. The first argument is a `List` of path components (without the `.css` suffix) to the CSS file to fix, and the second argument is the prefix to prepend to any embedded URLs within the CSS file. If you specify an `Empty` prefix, Lift will apply the context path returned by `S.contextPath` function.

Listing C-13. A fixCSS Example

```
class Boot {
  def boot(){
    ...
    LiftRules.fixCSS("styles" :: "theme" :: Nil, Empty)
    ...
  }
}
```

With this setup, Lift will apply the context path to any URLs within the `/styles/theme.css` file.

Internally, when you call `fixCSS`, a dispatch function is automatically created and prepended to `LiftRules.dispatch`. This function is needed to intercept the browser request to this `.css` resource. Also internally, we configure Lift to serve the CSS file and not to pass it on to the container as usual.

Using BindHelpers

Binders are extensively discussed in other chapters, so we won't reiterate them here. What we do want to cover is the `chooseTemplate` method, which lets you extract fragments of a given XML input, as shown in Listing C-14.

Listing C-14. Choosing a Template

```
//Assume the following markup.
<lift:CountGame.run form="post">
  <choose:guess>
    Guess a number between 1 and 100.<br/>
    Last guess: <count:last/><br />
    Guess: <count:input/><br/>
    <input type="submit" value="Guess"/>
  </choose:guess>
  <choose:win>
    You Win!!<br />
    You guessed <count:number/> after <count:count/> guesses.<br/>
  </choose:win>
</lift:CountGame.run>

// And the Scala code

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

class CountGame {
  def run(xhtml: NodeSeq): NodeSeq = {
```

```
      ...
      if (userWon) {
        ...
        bind("count", chooseTemplate("choose", "win", xhtml), ...)
      } else {
        ...
        bind("count", chooseTemplate("choose", "guess", xhtml), ...)
    }
  }
```

As you can see, we can take a single snippet input and programmatically decide which part of it to use for display with the `chooseTemplate` method.

Using the NamedPF Class

The `net.liftweb.util.NamedPF` class provides extremely useful functions for invoking partial functions that are chained into lists of functions. An example of how we can chain partial functions using `NamedPF` is shown in Listing C-15.

Listing C-15. A NamedPF Example

```
var f1: PartialFunction[Int,Int] = {
  case 10 => 11
  case 12 => 14
}

var f2: PartialFunction[Int,Int] = {
  case 20 => 11
  case 22 => 14
}

NamedPF(10, f1 :: f2 :: Nil)
```

Remember that many `LiftRules` variables are `RuleSeq`-s variables, meaning that most of the time, we're talking about lists of partial functions. Hence, internally, Lift uses `NamedPF` for invoking such functions that are ultimately provided by the user. If you would like to see some real-world examples, take a look at the source code for the `LiftRules` object.

Using Additional Lift Functionality

Besides what we've just covered, Lift has a few more utility functions and objects that you can use for specific application needs. This functionality is somewhat low-level, so we're only going to touch briefly on the functions here.

- The `HttpHelpers` object provides helper functions for HTTP query parameter manipulation, URL encoding and decoding, and so on. The most commonly used functions in `HttpHelpers` are `urlEncode` and `urlDecode`, which perform URL encoding and decoding on strings, as defined by RFC1738 (see `http://en.wikipedia.org/wiki/Percent-encoding` for more details). The other functions in the `HttpHelpers` trait mostly pertain to parsing or generating HTML markup and URLs, so they are primarily intended for internal Lift use. You can look at the API documentation for more details.
- Lift provides its own JSON parser (`net.liftweb.util.JSONParser`) in case you ever need one. At a first glance, it may be a bit redundant with Scala's JSON parser, but in fact, Scala's parser has its own problems with large JSON objects, and thus Lift uses its own JSON parser implemented using combinator parsers.
- The `net.liftweb.util.LD` (Levenshtein distance) object provides utility functions for calculating the edit distance between words. For more details on edit distance and its uses, see `http://en.wikipedia.org/wiki/Levenshtein_distance.`
- The `net.liftweb.util.ListHelpers` object, along with the related `net.liftweb.util.SuperList` trait, provide utility functions for manipulating lists that are not provided by Scala libraries. Among other things, `SuperList` adds some `Box`-like functionality to `List` via the `headOr` and `or` methods, a `replace` method that can be used to immutably replace a specific element in a `List` by index, and functions for rotating and permuting a `List`'s elements.
- The `net.liftweb.util.SecurityHelpers` object provides various functions used for random number generation, encryption, and decryption using the Blowfish algorithm, hash calculations (MD5, SHA, and SHA-256 are supported), and base64 encoding and decoding.
- The `net.liftweb.util.TimeHelpers` object provides utility functions that handle date and time operations. This includes formatting and parsing, as well as a set of implicit conversion functions that allow you to use time units for values. For example, you can use the value "10 seconds" in your code, and `TimeHelpers` will return the value in milliseconds.

Appendix D: Internationalization

The ability to display pages to users of multiple languages is a common feature of many web frameworks. Lift builds on the underlying Java internationalization foundations (primarily `java.util.Locale` and `java.util.ResourceBundle`) to provide a simple yet flexible means for using locales and translated strings in your applications. Locales are used to control not only the language of the text that's presented to the user but also number and date formatting, among other things. If you want more details on the underlying foundation of Java internationalization, we suggest that you visit the Internationalization home page located at `http://java.sun.com/javase/technologies/core/basic/intl/`. In this appendix, we'll be covering the specifics of how Lift utilizes Java internationalization features to simplify making your web application multilingual.

Using Resource Bundles

Resource bundles are sets of property files that contain keyed strings for your application to use in messages.

Note Lift's built-in support generally only deals with `PropertyResourceBundles`, though, technically, resource bundles can have other formats.

In addition to the key/value pair contents of the files, the file name itself is significant. When a `ResourceBundle` is specified by name, the base name is used as the default, and additional files with names of the form

```
<base name>_<ISO language code>
```

can be used to specify translations of the default strings in a given language. As an example, consider Listing D-1, which specifies a default resource bundle for an application that reports the status of a door (open or closed).

Listing D-1. Default Door Bundle

```
openStatus=The door is open
closedStatus=The door is closed
```

Suppose this file is called `DoorMessages.properties`. We can provide an additional translation for Spanish by creating a file called `DoorMessages_es.properties`, shown in Listing D-2.

Listing D-2. Spanish Door Bundle

```
openStatus=La puerta está abierta
closedStatus=La puerta está cerrada
```

When you want to retrieve a message, Lift will check the current `Locale` and see if there's a specialized `ResourceBundle` available for it. If so, it uses the messages in that file; otherwise, it uses the default bundle (we'll cover message retrieval in more detail in the next two sections).

Lift supports using multiple resource bundle files so that you can break up your messages into functional groups. You specify this by setting the `LiftRules.resourceNames` property to a list of the base names (without a language or `.properties` extension):

```
LiftRules.resourceNames = "DoorMessages" ::
  "DoorknobMessages" :: Nil
```

The order that you define the resource bundle names is the order that they'll be searched for keys. The message properties files should be located in your `WEB-INF/classes` folder so that they are accessible from Lift's class loader (because the properties files are retrieved with `ClassLoader.getResourceAsStream`); if you're using Maven, this will happen if you put your files in the `src/main/resources` directory.

Retrieving Localized Strings in Scala Code

Retrieving localized strings in your Scala code is primarily performed using the `S.?` method. When invoked with one argument the resource bundles are searched for a key matching the given argument. If a matching value is found, it's returned. If it can't be found, Lift calls `LiftRules.localizationLookupFailureNotice` on the `(key, current Locale)` pair and then simply returns the key. If you call `S.?` with more than one argument, the first argument is still the key to look up, but any remaining arguments are used as format parameters for `String.format` executed on the retrieved value. For example, Listing D-3 shows a sample bundle file and the associated Scala code for using message formatting.

Listing D-3. Formatted Bundles

```
// bundle
tempMsg=The current temperature is %0.1 degrees
// code
var currentTmp : Double = getTemp()
Text(S.?("tempMsg", currentTemp))
```

Lift also provides the `S.??` method, which is similar to `S.?` but uses the `ResourceBundle` for internal Lift strings. Lift's resource bundles are located in the `i18n` folder with the name `lift-core.properties`. The resource bundle name is given by `LiftRules.liftCoreResourceName` variable. Generally, you won't use the `S.??` method.

Adding Localized Strings to Templates

You can add localized strings directly in your templates through the `<lift:loc />` tag. You can provide a `locid` attribute on the tag, which is used as the lookup key. If you don't provide the attribute, the contents of the tag will be used as the key. In either case, if the key can't be found in any resource bundles, the contents of the tag will be used.

Listing D-4 shows some examples of how you could use `<lift:loc />`. In both examples, assume that we're using the resource bundle shown in Listing D-2. The fall-through behavior lets us put a default text (English) directly in the template, although for consistency, you should usually provide an explicit bundle for all languages.

Listing D-4. Using the <lift:loc /> Tag

```
<!-- using explicit key -->
<lift:loc locid="openStatus">The door is open</lift:loc>
<!-- should be the same result -->
<lift:loc>openStatus</lift:loc>
```

Calculating the Locale

The `Locale` for a given request is calculated by the function set in `LiftRules.localeCalculator`. The default behavior is to call `getLocale` on the `HttpServletRequest`, which allows the server to set it if your clients send locale preferences. If that call returns `null`, `Locale.getDefault` is used. You can provide your own function for calculating locales if you like.

Appendix E: Logging in Lift

Logging is a useful part of any application, Lift or otherwise. Logging can be used to audit user actions, give insight into runtime performance and operation, and even to troubleshoot and debug issues. Lift comes with a thin logging facade that sits on top of the log4j library. This facade provides simple access to the most common logging functions and aims to be easy to use, flexible, and most importantly, inconspicuous. If you do decide that Lift's logging facilities don't meet your needs, it's possible to use any Java logging framework you desire, but it's still useful to understand Lift's framework since Lift uses it internally for logging. In this appendix, we're going to cover how to configure and use logging from within your Lift application.

Configuring Logging

Out of the box, Lift sets up a log4j logging hierarchy using a cascading setup as defined in the `LogBoot._log4jSetup` method. First, it checks to see if log4j is already configured; this is commonly the case when a Lift application is deployed on a Java EE application server that uses log4j (JBoss, for example). If not, Lift attempts to locate a `log4j.props` or `log4j.xml` configuration file in the classpath, and if it finds either, it will use them for configuration. Failing that, Lift will fall back to configuring log4j using `theLogBoot.defaultProps` variable.

Usually, it's simplest to just provide a `log4j.props` or `log4j.xml` file in your resources to configure logging. If you prefer, you can provide your own `LogBoot.loggerSetup` function to use instead of `_log4jSetup` if you want to do something special, like use `configureAndWatch` to automatically load changes to your logging configuration.

If you would prefer, Lift's logging framework also supports the SLF4J logging framework. Enabling SLF4J is as simple as calling `Slf4jLogBoot.enable` in your `boot` method, as shown in Listing E-1. Note that you need to add both SLF4J and a backend as dependencies in your `pom.xml` file, and you should configure SLF4J before enabling it.

Listing E-1. Enabling SLF4J

```
class Boot {
  def boot {
    Slf4jLogBoot.enable()
    ...
  }
}
```

Logging Basics

Logging in Lift is performed via the `net.liftweb.util.Log` object. This object provides some basic log methods that we'll summarize in this section. Each log method comes in two forms: one with just an `Object` argument and one with `Object` and `Throwable` arguments. These correspond one-to-one with the log4j logging methods, although the parameters for Lift's logging method are passed by name. Passing parameters by name is done so that computation of the log message can be deferred. The deferred computation is useful to avoid processing messages for log statements below the current logging threshold, a topic we'll cover more in the next section.

These are Lift's basic logging methods:

- `trace`: This method logs a message at the `trace` level, which is generally intended for very detailed tracing of processing, even more detailed than the `debug` level.
- `debug`: This logs a message at the `debug` level, which is typically used to output internal variable values or other information that is useful in debugging and troubleshooting an application.
- `info`: This method logs a message at the `info` level, which is appropriate for general information about the application.
- `warn`: This one logs a message at the `warning` level, which should be used for reporting errors that can be handled cleanly, such as someone trying to submit a character string for a numeric field value.
- `error`: This method logs a message at the `error` level, which should be used for messages relating to errors that can't be handled cleanly, such as a failure to connect to a backing database.
- `fatal`: This one logs a message at the `fatal` level, which should be used for messages that relate to conditions under which the application cannot continue to function.
- `never`: This essentially throws away the passed message and is useful for some of Lift's functionality that requires a log output function (for example, `Schemifier.schemify` in Chapter 6).
- `assertLog`: This method allows you to test an assertion condition, and if true, log the assertion as well as a given message.

Listing E-2 shows an example of using a few different logging methods within a form-processing function that handles logins. We start out with some tracing of method entry (and corresponding exit at the end) and add an assertion to log the case where someone is logging in a second time. We add a debug statement that uses a function to generate the message, so the

string concatenation won't take place if debug logging is disabled. Finally, we log an error message if a problem occurred during authentication.

Listing E-2. Logging Examples

```
import _root_.net.liftweb.util.Log
class ExampleSnippet {
  def login (xhtml : NodeSeq) : NodeSeq = {
    ...
    def processLogin(name : String, password : String) = {
      Log.trace("Starting login process")
      try {
        ...
        Log.assertLog(User.currentUser.isDefined,
                      "Redundant authentication!")
        ...
        Log.debug(() => ("Retreiving auth data for " + name))
        ...
        if (!authenticated) {
          Log.error("Authentication failed for " + name)
       }
      } finally {
        Log.trace("Login process complete")
      }
    }
  }
}
```

Using Log Guards

We want to provide a brief discussion on the use of log guards and why they're usually not needed with Lift's log framework. A log guard is a simple test to see if a given log statement will actually be processed. The `Log` object provides a test method (returning a `boolean`) for each log level:

- `isDebugEnabled`
- `isErrorEnabled`
- `isInfoEnabled`
- `isTraceEnabled`
- `isWarnEnabled`

Fatal logging is always enabled, so there is no test for that level.

Log guards are fairly common in logging frameworks to avoid expensive computation of log messages that won't actually be used. Log guards are particularly relevant with debug logging, since debugging logs often cover a large section of code and usually aren't enabled in production.

The Log object can implicitly log guards for you because of the pass-by-name message parameters. As we showed in Listing E-2, simply converting your log message into a closure allows the Log object decide whether to execute the closure based on the current log level. You get the flexibility and simplicity of adding log statements anywhere you want without explicit log guards and without losing the performance benefit of the guards.

To explain log guards a bit more, let's assume for instance that the debug method were declared as `def debug(msg:AnyRef): Unit`. When debug is called, the parameter would first be evaluated and then passed to the method. Inside the method, a test is made to see if the debug level is enabled to know if we actually need to log that message or not. In this case, even if the debug level is turned off, we've still evaluated the parameters. Performing these parameter evaluations requires unnecessary computing if we don't need debug logs and, in an application that uses logging heavily, would likely lead to a significant impact on performance. So in this situation, we would have to test if the debug level is on even before calling the debug method using something like this:

```
if (Log.isDebugEnabled) {debug("Retreiving auth data")}
```

This is not very convenient and is prone to performance issues if you forget to wrap every single log statement. With Lift's logging methods, the pass-by-name parameters allow us to lazily evaluate each parameter only if we need to actually perform the logging.

Logging Mapper Queries

In addition to application-level logging, the Mapper framework provides a simple interface for logging queries. The `DB.addLogFunc` method takes a function `(String, Long) => Any` that can log the actual query string along with its execution time in milliseconds. Listing E-3 shows a trivial log function example.

Listing E-3. Mapper Logging

```
class Boot {
  def boot {
    DB.addLogFunc((query, time) =>
      Log.debug(() => (query + ":" + time + "ms")))
  }
```

Appendix F: E-Mail in Lift

Sending e-mail is a common enough task (for user registration, notifications, etc.) within a web application that we've decided to cover it here. Although e-mail isn't Lift's primary focus, Lift does provide some facilities to simplify e-mail transmission.

Setting Up Your Application to Send E-Mail

Configuration of the mailer can be handled in a few different ways. The simplest way is to define the system property `mail.smtp.host` to be the name of your mail server. If you would like programmatic control, the `net.liftweb.util.Mailer` object defines a `hostFunc` function `var, () => String`, which is used to compute the hostname of your SMTP server to be used for transmission. The default value is a function that simply looks up the `mail.smtp.host` system property and uses that `String`. If that property isn't defined, the mailer defaults to `localhost`. Setting the system property is the simplest way to change your SMTP relay, although you could also define your own function to return a custom hostname and assign it to `Mailer.hostFunc`.

Sending Messages

The mailer interface is simple but covers a wide variety of cases.

The `Mailer` object defines a number of case classes that correspond to the components of an RFC822 e-mail. The addressing and subject cases classes—`From`, `To`, `CC`, `BCC`, `ReplyTo`, and `Subject`—should all be self-explanatory. For the body of the e-mail, you have three main options:

- `PlainMailBodyType`: This option represents a plain-text e-mail body based on a given `String`.
- `XHTMLMailBodyType`: This one represents an XHTML e-mail body based on a given `NodeSeq`.
- `XHTMLPlusImages`: This is similar to `XHTMLMailBodyType`, but in addition to the `NodeSeq`, you can provide one or more `PlusImageHolder` instances that represent images to be attached to the e-mail (embedded images, so to speak).

The `Mailer.sendMail` function is used to generate and send e-mail. It takes three arguments: the `From` sender address, the `Subject` of the e-mail, and a `varargs` list of recipient addresses and body components. The mailer creates MIME/Multipart messages, so you can send more than one body (e.g., plain text and XHMTL) if you would like. Listing F-1 shows an example of sending e-mail to a group of recipients in both plain text and XHTML formats. The `Mailer` object defines some implicit conversions to `PlainMailBodyType` and `XHTMLMailBodyType`, which we use here. We also have to do a little `List` trickery to be able to squeeze multiple arguments

into the final vararg argument, since Scala doesn't support mixing regular values and coerced sequences in vararg arguments.

Listing F-1. Sending a Two-Part E-Mail

```
import _root_.net.liftweb.util.Mailer
import Mailer._
...
val myRecips : List[String] = ...
val plainContent : String = "..."
val xhtmlContent : NodeSeq = ...

Mailer.sendMail(From("no-reply@foo.com"), Subject("Just a test"),
                (plainContent :: xhtmlContent :: myRecips.map(To(_))) : _*)
```

When you call sendMail, you're actually sending a message to an actor in the background that will handle actual mail delivery; because of this, you shouldn't expect to see a synchronous relay of the message through your SMTP server.

Appendix G: JPA Code Listings

To conserve space and preserve flow in the main text, we've placed full code listings for the JPA chapter (Chapter 10) in this appendix. The full library demonstration is available under the main Lift git repository at `http://github.com/dpp/liftweb/tree/master`. To illustrate some points, we've included selected listings from the project.

Listing G-1 contains the `Author` entity.

Listing G-1. Author.scala

```
package com.foo.jpaweb.model
import javax.persistence._
/**
  An author is someone who writes books.
*/
@Entity
class Author {
  @Id
  @GeneratedValue(){val strategy = GenerationType.AUTO}
  var id : Long = _
  @Column{val unique = true, val nullable = false}
  var name : String = ""
  @OneToMany(){val mappedBy = "author",
               val targetEntity = classOf[Book]}
  var books : java.util.Set[Book] =
  new java.util.HashSet[Book]()
}
```

Listing G-2 contains the `orm.xml` mapping functionality.

Listing G-2. orm.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation=
    "http://java.sun.com/xml/ns/persistence/orm
     http://java.sun.com/xml/ns/persistence/orm_1_0.xsd"
   version="1.0">
  <package>com.foo.jpaweb.model</package>
  <entity class="Book">
  <named-query name="findBooksByAuthor">
    <query>
    <![CDATA[from Book b where b.author.id = :id order
             by b.title]]></query>
  </named-query>
  <named-query name="findBooksByDate">
    <query><![CDATA[from Book b where b.published
      between :startDate and :endDate]]></query>
  </named-query>
  <named-query name="findBooksByTitle">
    <query><![CDATA[from Book b where lower(b.title)
      like :title order by b.title]]></query>
  </named-query>
  <named-query name="findAllBooks">
    <query><![CDATA[from Book b order by
      b.title]]></query>
  </named-query>
  </entity>
  <entity class="Author">
  <named-query name="findAllAuthors">
    <query><![CDATA[from Author a order by
      a.name]]></query>
  </named-query>
  </entity>
</entity-mappings>
```

Listing G-3 contains the `Enumv` trait.

Listing G-3. The Enumv Trait

```
package com.foo.jpaweb.model
/* adds a valueOf function, assumes name is defined
add optional description */
trait Enumv  {
  this: Enumeration =>
  private var nameDescriptionMap =
```

```
    scala.collection.mutable.Map[String, String]()
  /* store a name and description for forms */
  def Value(name: String, desc: String) : Value = {
  nameDescriptionMap += (name -> desc)
  new Val(name)
  }
    /* get description if it exists else name */
  def getDescriptionOrName(ev: this.Value) =
  try {
    nameDescriptionMap(""+ev)
  } catch {
    case e: NoSuchElementException => ev.toString
  }
  /* get name description pair list for forms */
  def getNameDescriptionList =
    this.elements.toList.map(v => (v.toString,
      getDescriptionOrName(v) ) ).toList
  /* get the enum given a string */
  def valueOf(str: String) =
  this.elements.toList.filter(_.toString == str) match {
    case Nil => null
    case x => x.head
  }
}
```

Listing G-4 shows the `EnumerationType` class.

Listing G-4. The EnumvType Class

```
package com.foo.jpaweb.model
import java.io.Serializable
import java.sql.PreparedStatement
import java.sql.ResultSet
import java.sql.SQLException
import java.sql.Types
import org.hibernate.HibernateException
import org.hibernate.usertype.UserType
/**
 * Helper class to translate enum for hibernate
 */
abstract class EnumvType(val et: Enumeration with Enumv) extends UserType {
  val SQL_TYPES = Array({Types.VARCHAR})
  override def sqlTypes() = SQL_TYPES
  override def returnedClass = classOf[et.Value]
  override def equals(x: Object, y: Object): Boolean = {
  return x == y
  }
```

```
  override def hashCode(x: Object) = x.hashCode
  override def nullSafeGet(resultSet: ResultSet,
                           names: Array[String],
                           owner: Object): Object = {
  val value = resultSet.getString(names(0))
  if (resultSet.wasNull()) return null
  else {
    return et.valueOf(value)
  }
  }
  override def nullSafeSet(statement: PreparedStatement,
       value: Object, index: Int): Unit = {
  if (value == null) {
    statement.setNull(index, Types.VARCHAR)
  } else {
    val en = value.toString
    statement.setString(index, en)
  }
  }
  override def deepCopy(value: Object): Object = value
  override def isMutable() = false
  override def disassemble(value: Object) =
    value.asInstanceOf[Serializable]
  override def assemble(cached: Serializable, owner:
      Object): Serializable = cached
  override def replace(original: Object, target: Object,
      owner: Object) = original
}
```

And finally, the `web.xml` file in Listing G-5 shows the `LiftFilter` setup as well as `persistence-context-ref`.

Listing G-5. JPA web.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application
2.3//EN"
"http://java.sun.com/j2ee/dtds/web-app_2_3.dtd">
<web-app>
<filter>
  <filter-name>LiftFilter</filter-name>
  <display-name>Lift Filter</display-name>
  <description>The Filter that intercepts lift
    calls</description>
  <filter-class>
    net.liftweb.http.LiftFilter
```

```
  </filter-class>
  <persistence-context-ref>
  <description>
    Persistence context for the library app
  </description>
  <persistence-context-ref-name>
    persistence/jpaweb
  </persistence-context-ref-name>
  <persistence-unit-name>
    jpaweb
  </persistence-unit-name>
  </persistence-context-ref>
</filter>
      <filter-mapping>
  <filter-name>LiftFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
</web-app>
```