A Gentle Introduction to Lift

Derek Chen-Becker

Boulder JUG, August 11th, 2009

Many Thanks!

- Boulder JUG
- Kris Nuttycombe
- Fred Jean
- Tom Flaherty
- All of you!

About Me...

- Worked with Java for over 12 years
- Worked with various web frameworks
  - Straight Servlet/JSP (yikes!)
  - Struts
  - Tapestry
- Got involved with Scala/Lift in mid-2007
- Co-author of *The Definitive Guide to Lift*

A Little Background on Lift

- Started by David Pollak late 2006
- Attempts to take the best ideas from existing frameworks
- Heavily influenced by Scala's design
- Rapidly growing community and codebase
- Recently released 1.0, 1.1 slated for end of year (waiting on Scala 2.8)
- In use at places like SAP, and soon others

Shall We Dance?

- Let's look at a real-world example
- Real-time chat app (a lá Google Talk Web)
  - Combines AJAX and Comet
  - Traditionally not trivial

Define a Data Object

- Used to encapsulate “messages” to be passed around.
- Keep things simple and make it just hold Strings for now:

```
case class Messages(msgs : List[String])
```

Create a Chat Server

- We'll use an actor with some special traits here

```
object ChatServer extends Actor with ListenerManager {
  // Global list of messages
  private var msgs: List[String] = Nil

  // The message that we'll send to all subscribers
  protected def createUpdate = Messages(msgs)

  // Process messages that we receive
  override def highPriority = {
    case s: String if s.length > 0 =>
      msgs ::= s
      updateListeners()
  }
  this.start
}
```

Create the Page Component

- Again, we'll use an actor, but a page-specific one (class vs object)

```
class Chat extends CometActor with CometListenee {
  private var msgs: List[String] = Nil

  def render =
  <div>
    <ul>{msgs.reverse.map(m => <li>{m}</li>)}</ul>
    {ajaxText("", s => {ChatServer ! s; Noop})}
  </div>

  protected def registerWith = ChatServer

  override def highPriority = {
    case Messages(m) => msgs = m ; reRender(false)
  }
}
```

Set up Boot (Config)

Some minor tasks to tie things together:

```
class Boot {
  def boot {
    // where to search for snippets,
    LiftRules.addToPackages("com.test")

    LiftRules.early.append(makeUtf8)
  }

  /**
   * Force the request to be UTF-8
   */
  private def makeUtf8(req: HttpServletRequest) {
    req.setCharacterEncoding("UTF-8")
  }
}
```

Create the Master Page

We'll use Lift's excellent templating support:

`/templates-hidden/default.html`

```
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:lift="http://liftweb.net/">
  <head>
    <title>Real-Time Chat</title>
    <lift:CSS.blueprint />
    <lift:CSS.fancyType />
    <script id="jq" src="/classpath/jquery.js" type="text/javascript"/>
    <script id="json" src="/classpath/json.js" type="text/javascript"/>
  </head>
  <body>
    <lift:bind name="content" />
  </body>
</html>
```

Create the Chat Page

Using the template we just created:

`/index.html`

```
<lift:surround with="default" at="content">
  <h2>Chat away!</h2>
  <lift:comet type="Chat" />
</lift:surround>
```

That's It!

```
mvn -Djetty.port=9090 clean jetty:run
```

What's So Special Here?

Scala is what makes Lift possible:

- Pattern matching + case classes
- Actors
- Function as objects
- Traits and composition
- Immutability
- XML literal support

Pattern Matching

- Used for things like rewrites, custom dispatch
- Leverages case classes and the apply/unapply methods
- Wildcards, placeholders and guards allow for very expressive constructs:

```
LiftRules.rewrite.append {
  case RewriteRequest(ParsePath(List(“user”, username)), PostRequest, _)
       if userExists(username) ⇒
         RewriteResponse(List(“viewUser”), Map(“username” -> username))
  case RewriteRequest(ParsePath(List(“user”, _)), _, _) ⇒
         RewriteResponse(List(“noSuchUser”))
}
```

Function Passing

- Encapsulates logic concisely
- Allows lazy evaluation (indirectly, by-name params)
- Wildcard shorthand keeps simple things simple

```
// form handling
var name = “”; SHtml.text(name, name = _)

// loan pattern
DB.exec(DefaultConnectionIdentifier, “select * from widgets”) { rs ⇒
  while (rs.next) { ... }
}

// tweak the request
LiftRules.early { _.setCharacterEncoding(“UTF-8”) }

// guilt-free logging
Log.debug(“Sending ” + reallyExpensiveFunctionCall())
```

Traits and Composition

- Allow very fine-grained aggregation of common functionality
- Used extensively in Lift. Heaviest usage is in Mapper

```
class MyEntity extends LongKeyedMapper[MyEntity] with IdPK { ... }

Class MySnippet extends DispatchSnippet {
  def dispatch : DispatchIt = {
    case “view” ⇒ viewStuff _
    case “review” ⇒ reviewStuff _
  }
  ...
}
```

XML Support

- Very useful for transformation of data

```
val links : List[(String,String)] =
  List((“Link A”, “http://...”),
       (“Link B”, “http://...”),
       (“Link C”, “https://...”))

val myLinkList =
  <ul>{ links.flatMap({case (name,url) ⇒
          <li><a href={url}>{name}</a></li>}) }</ul>
```

- Ripe for abuse! Bind and chooseTemplate are better for large/complex XML tasks

View First, part 1

- Define the template for our view
- Make a space for the table rows

```
<lift:surround at=”content”>
<lift:UserOps.linkView>
<h1><user:name />'s Favorite Links:</h1>

<div>
<table>
  <thead><tr><th>Name</th><th>Link</th></tr></thead>
  <tbody>
  <user:links>
    <tr><td><link:name /></td><td><link:link /></td></tr>
  </user:links>
  </tbody>
</table>
</div>
</lift:UserOps.linkView>
</lift:surround>
```

View First, part 2

- Create a snippet method to back the view

```
class UserOps {
  def linkView (xhtml : NodeSeq) : NodeSeq = {
   val user = ... // wave wand here

   bind("user", xhtml,
        "name" → Text(user.name),
        "links" → user.links.flatMap({ case (name,url) ⇒
          bind("link", chooseTemplate("user", "links", xhtml),
               "name" → Text(name),
               "link" → <a href={url}>{url}</a>)}))
  }
}
```

- Arbitrary nesting of bind calls
- Promotes separation of markup and code

View First, part 3

- Arbitrary composition of XML templates and snippets

```
<lift:surround at=”content”>
<lift:Homepage>
<h1>Welcome, <page:username/>!</h1>
<page:content />
</lift:Homepage>
</lift:surround>
```

```
class Homepage {
  def render (xhtml : NodeSeq) : NodeSeq = {
  ...
  bind(“page”, xhtml,
           ...
           “content” → currentContent ++ <lift:embed what=”welcomefooter” />)
  }
}
```

Comet and AJAX Made Easy

- Rich set of existing traits and generator methods reduce work involved in making dynamic pages work
- JavaScript abstractions help with client-side code

```
class Airplane {
  def render (xhtml : NodeSeq) : NodeSeq = {
    var wheelsDown = false
    var landing = false

    import JsCmds.{Alert,Noop}

    def safetyCheck =
      if (landing && (!wheelsDown)) Alert("You're going to crash!") else Noop

    bind("airplane", xhtml,
     "wheels" ->
       SHtml.ajaxCheckbox(wheelsDown, d => { wheelsDown = d; safetyCheck }),
     "landing" ->
       SHtml.ajaxCheckbox(landing, l => { landing = l; safetyCheck }))
  }
}
```

SiteMap

- Not just a menu
  - Access control
  - Rewriting
  - Custom template and snippet selection
  - Type-safe parameters
- Not required, but very useful
- Widgets available for advanced menu rendering (superfish)

Leverage Existing Libs

- For example, ScalaJPA makes JPA simple(r)

```
@Entity
class SimpleUser {
  @Id var id : Long = _
  @Basic var name : String = _
  @Basic var age : Long = _
}

object EM extends LocalEMF(“my-persistence-unit”) with ThreadLocalEM

val users =
  EM.createQuery(“select user from SimplerUser user”).findAll.map(_.name)
```

- Lift has modules that use OpenAuth, AMQP, textile and more...

Conclusion

- Scala drives more concise code: more time spent writing logic and less time writing boilerplate
- Lift utilizes a lot of advanced Scala features to make hard things appear simple
- Lift is designed with simplicity, security and stability as primary goals
- Leverage existing Java frameworks and libraries (JEE, JPA, etc)