fleetzone

Introduction to the Lift web framework

Jeppe Nejsum Madsen

First Cph Scala/Lift meetup, Feb 23rd 2011

About me

- Jeppe Nejsum Madsen
- CTO, Co-founder of FleetZone
- Worked with Scala/Lift since April 2009
- Lift Committer
- Building SaaS platform for fleet performance management based on Scala/Lift
- Contact:
  - jnm@fleetzone.dk
  - @nejsum


Lift – 10000 feet view

What is Lift?

1. Collection of useful Scala libraries, e.g.
   - Parsing/Constructing JSON
   - Common utilities (Logging etc)
2. Framework for writing scalable, secure & interactive web applications in a functional way using Scala
   - Excellent security out of the box
   - Easy to create Ajax & Comet applications that work
   - Builtin ActiveRecord style ORM with support for both SQL & NoSQL dbs

Boot – Configure your app

- Run once during startup
- Define your sitemap, transaction strategy etc.
- Most things in Lift can be configured using the LiftRules object:

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

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

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

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

SiteMap – Define your app structure

- One of the most powerful features in Lift!
- Defines your URL structure
  - If a page is not defined in the SiteMap Lift will not serve it (normally ☺)
  - Central place to provide access control for entire app
  - One or more navigational structures (e.g. menus) can be easily generated from the SiteMap
- SiteMap contains collection of Loc[T], each Loc defining a single URL (e.g. /about) or URL pattern e.g. (/orders/1234)
- Simple menus can be constructed using a DSL

SiteMap – Simple Menu

- Single page

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

- Submenus

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

- Parameterized for URLs like /param/somestuff

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

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

Templates – Define your layout

- Templates are (X)HTML files that define the static structure of your page
- Templates contain Snippets that render the dynamic parts by invoking Scala code
- Snippets can be specified in two different ways:
  1. Using the “traditional” XHTML style:
     - <lift:mySnippet>snippet body</lift:mySnippet>
  2. The recently added “designer friendly” style where snippets are added to the class attribute on any HTML element:
     - snippet body
- I prefer the designer friendly style since this makes it possible to write templates that validate against HTML5
- Together with CSS Selector Transforms, designer friendly markup provides a very powerful templating mechanism

Example template

```
<div id="main" class="lift:surround?with=default&at=content">
  <div>Hello World.  Welcome to your Lift application.</div>
  <div>Check out a page with <a href="/param/foo">query parameters</a>.</div>

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

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

Snippets – Render dynamic content

- All dynamic content in Lift is rendered by Snippets
- Snippets are functions: NodeSeq => NodeSeq
- Transforms the snippet body NodeSeq (which may be empty or ignored) into a resulting NodeSeq that is rendered on the page.
- In the simple case, the snippet name translates directly to a class with the same name in the snippet package.
- In more advanced scenarios, snippets can be defined per URL, overridden per session etc.

Built-in snippets

Lift comes with a lot of built-in snippets. Some useful ones are:

- Surround – Surround body with another template. Useful for defining common layout in a single file.
- Embed – Embed another template
- Menu – Render menu items from sitemap
- Msgs – Show warning & error messages
- Loc – Lookup localized strings
- LazyLoad - Load slow part of page lazily

A simple snippet

- Given a simple snippet invocation:

```
3 |    The current time is <span class="lift:HelloWorld">now</span>.
```

- And HelloWorld.scala

```
class HelloWorld {
  def render(body: NodeSeq): NodeSeq = Text(new java.util.Date().toString)
}
```

- This will render
  - The current time is Tue Feb 22 11:51:04 CET 2011
- Note that this Snippet ignore the body parameter (which in this case will contain Text("now") instance)

CSS Selector transforms

- Snippet binding on steroids (new since Lift 2.2-M1)
- Principle: Inside a snippet, use CSS selectors to replace various elements in the snippet body
- Template:

```
<div class="lift:HelloWorld.test">
  My name is <span id="name">the name</span>. Children:
  <ol>
    <li id="kids">kid 1</li>
    <li class="clearable">kid 2</li>
    <li class="clearable">kid 3</li>
  </ol>
</div>
```

- Snippet

```
class HelloWorld {
  def test = "#name"    #> "Jeppe" &
             "#kids *" #> List("Sille", "Theo") &
             ClearClearable
}
```

My name is Jeppe. Children:

1. Sille
2. Theo

Snippet using URL param

- Remember the Loc[ParamInfo] for handling URLs like /param/stuff?
- Here's how to use the passed parameter:

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

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

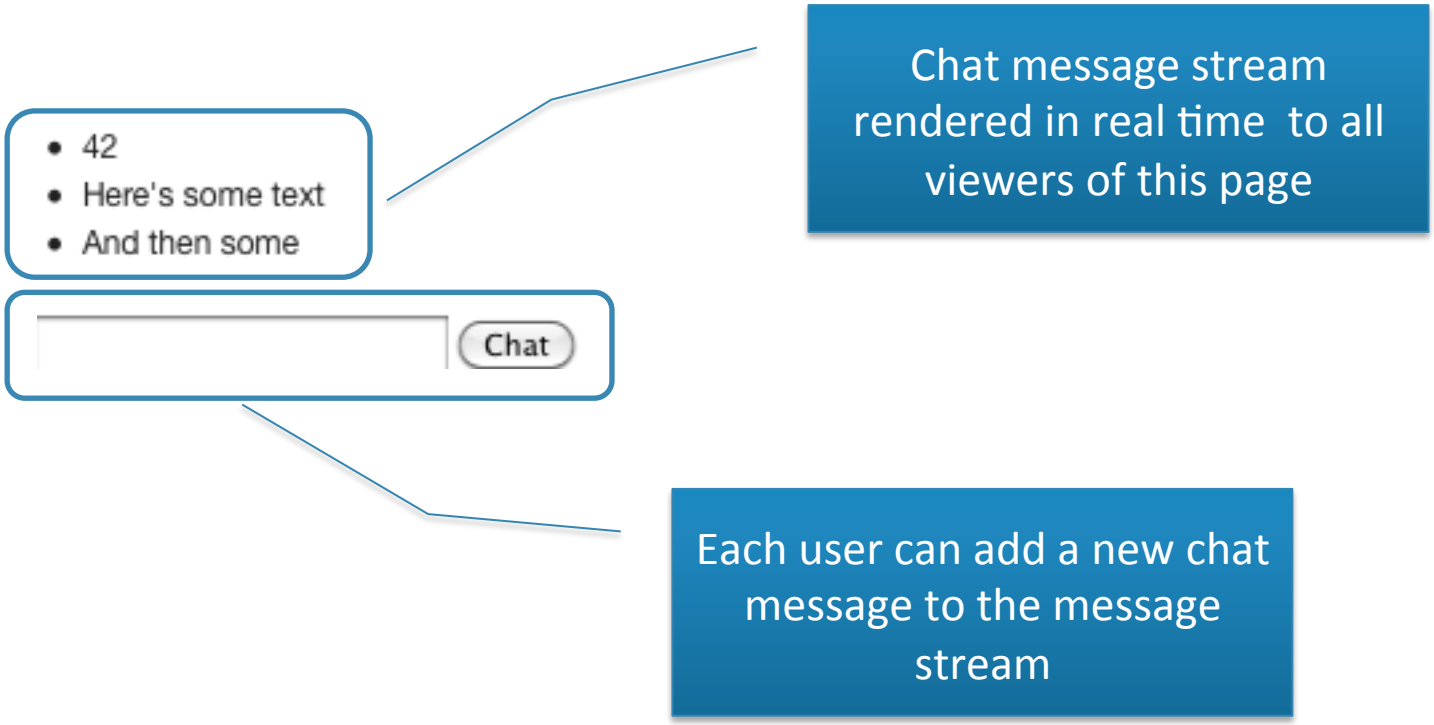
- Alternative snippet:

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

So, what is it good for?

The ubiquitous Chat App

From http://seventhings.liftweb.net/comet



Chat App 1 – The Server

```
/**
 * The chat server
 */
object ChatServer extends LiftActor with ListenerManager {
  private var msgs = Vector("Welcome") // the private data

  // what we send to listeners on update
  def createUpdate = msgs

  // handle incoming messages
  override def lowPriority = {
    case s: String => {
      msgs = (msgs :+ s.trim).filter(_.length > 0).takeRight(20)
      updateListeners()
    }
  }
}
```

Chat App 2 – User input

Template

```
<form class="lift:Form.ajax">
  <input class="lift:ChatIn" id="chat_in">
  <input type="submit" value="Chat">
</form>
```

Snippet

```
object ChatIn {
  // max count per session
  private object lineCnt extends SessionVar(0)

  def render =
    "*" #> SHtml.onSubmit(s => {
      if (s.length < 50 && lineCnt < 20) { // 20 lines per session
        ChatServer ! s // send the message
        lineCnt.set(lineCnt.is + 1)
      }
      SetValById("chat_in", "") // clear the input box
    })
}
```

Chat App 3 – Render chats

```
<ul class="lift:comet?type=Chat">
  <li>Line 1</li>
  <li class="clearable">Line 2</li>
  <li class="clearable">Line 3</li>
</ul>

class Chat extends CometActor with CometListener {
  private var msgs: Vector[String] = Vector() // private state

  // register this component
  def registerWith = ChatServer

  // listen for messages
  override def lowPriority = {
    case v: Vector[String] => msgs = v; reRender()
  }

  // render the component
  def render = ClearClearable & "li *" #> msgs
}
```

Mapper – Lift ORM

- Original Lift ORM, Record is newer and supports NoSQL (but not as well tested for SQL as Mapper)

```
class Account extends LongKeyedMapper[Account] with IdPK  {
  def getSingleton = Account // what's the "meta" server
  object name extends MappedPoliteString(this, 64)
}

object Account extends Account
 with LongKeyedMetaMapper[Account]
 with CRUDify[Account]{
}
```

- Now you can do things like:

```
val acc = Account.create.name("Savings").save
val allAccounts = Account.findAll
val savingsAcc = Account.find(By(Account.name, "Savings"))
savingsAcc.delete
```

Resources

- http://simply.liftweb.net
  - David Pollaks on-going book on Lift (and where most of the code samples are from)
- http://exploring.liftweb.net/
  - Another great Lift book with the PocketChange example app. Continuously updated
- The Lift mailing list
  - A great community

Thank you!

Think this sounds interesting?

Would you like to work with Scala & Lift?

We're hiring ☺

Workshop

- Start a new Lift App:
  - Install JDK & GIT
  - git clone git://github.com/lift/lift_22_sbt.git
  - cd lift_22_sbt/lift_basic/
  - sbt
  - > update
  - > ~jetty-run
  - Browse to http://localhost:8080
  - Go hack the Scala & HTML code!