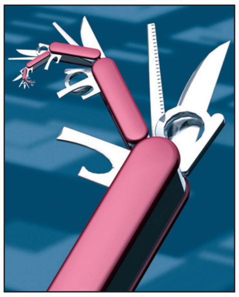

A Chat Application in Lift

David Pollak • *Lift Web Framework*
Steve Vinoski • *Verivue*

Last year, Debasish Ghosh and Steve Vinoski gave an overview of the Scala language, highlighting some of the features of Scala using the Lift Web framework in their article, "Scala and Lift — Functional Recipes for the Web."[1] We pick up where they left off in this column by taking a deeper dive into Lift, a Web framework in the vein of Seaside (www.seaside.st) and WebObjects (http://developer.apple.com/tools/webobjects).

In contrast to frameworks oriented around the model-view-controller (MVC) pattern, Lift abstracts the HTTP request–response cycle rather than wrapping HTTP concepts in APIs. This means you put HTML element definition and action in the same place:

```
var name = ""
SHtml.text(name, s => name = s)
```

This example creates an `<input type="text" value=""/>` tag and associates it with the function that sets the variable name to the value the user enters, whether submitted via Ajax or via a normal HTTP `GET` or `POST`. The advantages of Lift's approach are numerous, including increased security through randomly assigned HTML element names, enhanced maintainability, and unification between Ajax and normal HTTP.

Here, we show how to build a multiuser, real-time chat application in Lift and discuss Scala's language features that make Lift possible. The application provides a single chat server that takes chat messages and redistributes the messages out to all listeners. But before we present the code for the chat application, let's first briefly discuss Scala's support for actors and messages.

Actors and Messages

Similar to Erlang, Scala supports the actor model, an approach to concurrency in which each actor is an independent entity capable of sending and receiving messages to and from other actors and creating new actors. Whereas actors are baked into Erlang, they're a library in Scala. In fact, Lift has its own actor library that has different, more Web-friendly performance characteristics than the Scala actor library. An actor gives a few guarantees that provide a simple concurrency model: asynchronous message sending, processing at most one message at a time, and processing messages in order from the actor's mailbox.

Sending a message to an actor is asynchronous: the message-send method returns almost immediately. A message send places the message in the target actor's mailbox. An actor defines a set of messages that it can handle at the current time. When the application causes a message to be placed into the actor's mailbox, the actor is scheduled to review its messages. If the actor can handle at least one message in its mailbox, it's scheduled to execute that message and potentially other messages in its mailbox. Note that with actors, you don't need to lock private variables because you're guaranteed to execute code that can access those variables on only one thread at a time. Because message sending is asynchronous, it takes a lot of work to deadlock an actor.

The Chat Application

To follow along with our chat application instructions, you'll need to install on your computer version 2.2.1 of the Maven project life-cycle management tool (see http://maven.apache.org) and Java 1.6. The first thing we'll do is create a Lift project's shell:

```
mvn archetype:generate \
  -DarchetypeGroupId=\
net.liftweb \
  -DarchetypeArtifactId=\
lift-archetype-basic \
  -DarchetypeVersion=2.0-M2
```

When Maven prompts you, set the `groupId` to `com.liftcode`, the `artifactId` to `chat`, and use the default for everything else. Once this command completes, change to the chat directory and enter the following command:

```
mvn jetty:run
```

After this completes (which could take a while, depending on how much Maven needs to download), point your browser to http://localhost:8080, and you'll find you've got a running application.

Now, let's add a chat component. First, create a file named `Chat.scala` under the chat directory in the `src/main/scala/com/liftcode/comet` subdirectory. Add the following to the top of the file:

```
package com.liftcode.comet
import net.liftweb._
import http._
import actor._
import scala.xml.NodeSeq
```

Scala supports traits, which define a contract with the implementing class, such as Java's interfaces, as well as optionally providing implementation like Ruby's mix-ins. We can use that to advantage in the code for our chat application:

```
object ChatServer extends
LiftActor with ListenerManager
{
  private var msgs =
    List("Welcome")

  // message handler
  override def highPriority =
  {
    case s: String =>
      msgs ::= s
      updateListeners()
  }

  def createUpdate = msgs
}
```

The first line defines a singleton named `ChatServer`. By using the mix-in approach, we ensure that the `ChatServer` is both a `LiftActor` and a `ListenerManager`. The `ListenerManager` trait we mix into `ChatServer` provides the add/remove listener functionality and sends update messages when the actor's state changes, thus enabling `ChatServer` to keep track of listeners.

The code example's second line defines a private list variable named `msgs`, seeded with a "Welcome" message. If we receive a `String` as a message, we add it to our list of messages in the `highPriority` method and update our listeners. The `createUpdate` method provides the messages we want to send. This completes our `ChatServer` definition.

Next, we define the `CometActor`, which represents the chat component that lives in the user's browser. Lift supports "server-push" applications, also known as long-polling or Comet applications. A Comet component in Lift is an actor that responds to messages and pushes changes out to any browser pages that might be viewing the component. Before we explain how that works, let's look at the component:

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

  // what component do we
  // register with?
  def registerWith =
    ChatServer

  // define how to handle
  // messages from the
  // chat server
  override def highPriority =
  {
    case m: List[String] =>
      // set local state
      msgs = m
      // redraw ourselves
      reRender(false)
  }

  def render =
    bind("chat",
      "line" -> lines _,
      "input" -> SHtml.text(
         "",
         s => ChatServer ! s))

  private def lines(
    xml: NodeSeq):
      NodeSeq =
        msgs.reverse.
          flatMap(
            m => bind(
              "chat",
              xml,
              "msg" -> m))
}
```

The class definition, local state variable, listener registration, and message-handling code mirror the code in the chat server. This class also belongs in the `src/main/scala/com/liftcode/comet/Chat.scala` file we defined earlier.

Before we discuss how the rendering code works, let's look at the view code. Here's the HTML template that invokes the chat component. Place this code in the file `src/main/webapp/index.html` (replace any contents that might already be present):

```
<lift:surround with="default"
      at="content">
  <lift:comet type="Chat">
    <ul>
      <chat:line>
        <li><chat:msg/></li>
      </chat:line>
    </ul>

    <lift:form>
      <chat:input/>
      <input type="submit"
          value="chat"/>
    </lift:form>
  </lift:comet>
</lift:surround>
```

The first element, `lift:surround`, surrounds this page with the default template, and the `lift:comet` element on the second line invokes the Comet chat component. The `lift:comet` tag's children are passed to the component so that the component can "bind" the business logic to the view. Let's look at the binding:

```
def render =
  bind("chat",
    "line" -> lines _,
    "input" ->
      SHtml.text("",
        s => ChatServer ! s))
```

We bind the `chat:line` tag to the function `lines`, and we bind the `chat:input` tag to an input text element, which, when submitted, sends the `String` to the `ChatServer` as an asynchronous message via the `!` operator. In our view, we make the form an Ajax form using the `<lift:form/>` tag.

Finally, let's look at how the message lines are rendered:

```
private def lines(
  xml: NodeSeq):
    NodeSeq =
      msgs.reverse.
        flatMap(
          m => bind(
            "chat",
            xml,
            "msg" -> m))
```

We take the list of messages (newest to oldest) and reverse the list, so we display oldest to newest. For each list element, we bind the message to the view. So, we're replacing

```
<li><chat:msg/></li>
```

with

```
<li>Hello</li>
```

for each message.

At this point, you can compile and run your application again via the same `mvn jetty:run` command. Point multiple browser applications to http://localhost:8080, and you'll see instantly updated chat messages in all your browsers.

Most importantly, note that you haven't implemented any HTTP plumbing. Your controller code is simply the following function:

```
s => ChatServer ! s
```

You can have many such controllers on a Web page. In fact, you can have multiple Comet components on a single page.

Discussion

Lift's Comet implementation uses a single HTTP connection to poll for changes to an arbitrary number of components on the page. Each component has a version number. The long poll includes the version number and the component's globally unique identifier (GUID). On the server side, a listener is attached to all the GUIDs listed in the long-poll requests. If any component has a higher version number, or the version number increases during the long-poll period, the listener sends the deltas — a JavaScript set describing the change from each version — to the client. The browser applies the deltas and sets the client's version number to the change set's highest version number. Lift integrates long polling with session management so that, if a second request comes into the same URL during a long poll, the long poll is terminated to avoid connection starvation (some browsers have a default maximum of two HTTP connections per named server). Lift also supports Domain Name System (DNS) wild-carded servers for long-poll requests such that each tab in the browser can do long polling against a different DNS server. This avoids connection starvation issues. Lift dynamically detects the container in which the servlet is running. On Jetty versions 6 and 7 and Glassfish version 3.0, Lift uses the platform's continuations implementation to avoid using a thread during the long poll. Lift's JavaScript can sit on top of the jQuery and YUI JavaScript frameworks. The actual polling code includes back-off on connection failures and other graceful ways of dealing with transient connection failures.

Scala Features

Let's take a tour of the Scala language features that we used for our example.

- *Singletons.* With built-in singletons, we have a unified object model in which everything is an object, yet we can pass a singleton as a parameter to a method or return a singleton, such as with the `registerWith` method.
- *Pattern matching.* We matched incoming messages against patterns and took appropriate action. Additionally, your code can execute pattern matching inline or convert it to a function and return it from a method. We did the latter when we returned the pattern-matching function from the `highPriority` methods.
- *Type inferencing.* Our code is very concise because we didn't have to declare many types. We declared types when the compiler couldn't figure them out. Places where you generally must declare types in Scala are places you should document them, or places for which you'd ordinarily write a test in a dynamic language such as Ruby.
- *Traits.* We're able to compose complex behavior into our classes by mixing traits together. Traits

have the benefits of multiple inheritance without the diamond problem. This highlights the fact that Scala is an advanced OO language as well as an advanced functional language.

- *Immutable data types.* We passed the string list as a message to the listeners from the chat server. Just as `String` is immutable in Java, `List` is immutable in Scala. That means it's safe to share references and access the same references from multiple threads without synchronization. Scala has many immutable types including immutable collections and XML in its standard library.
- *Function passing.* In Scala, functions are objects. We were able to pass the function

```
s => ChatServer ! s
```

to the `SHtml.text` method. The method associates the function with the GUID assigned to the HTML element.

Together, these features supply significant utility for frameworks such as Lift. Our Lift example reveals the productivity it affords developers, even for an application as involved as a long-polling chat server.

Reference

1. D. Ghosh and S. Vinoski, "Scala and Lift — Functional Recipes for the Web," *IEEE Internet Computing*, vol. 13, no. 3, 2009, pp. 88–92.

David Pollak has been writing commercial software since 1977. He wrote the first real-time spreadsheet and the world's highest-performance spreadsheet engine. Since 1996, Pollak has been using and devising Web development tools. He's also developed numerous commercial projects in Ruby on Rails. In 2007, Pollak founded the Lift Web Framework. He's also the author of *Beginning Scala* (Apress, 2009).

Steve Vinoski is a member of the technical staff at Verivue in Westford, Mass. He's a senior member of IEEE and a member of the ACM. You can read Vinoski's blog at http://steve.vinoski.net/blog and reach him at vinoski@ieee.org.