

Build Comet applications using Scala, Lift, and jQuery

Creating the e-commerce Auction Net site

Skill Level: Intermediate

[Michael Galpin \(mike.sr@gmail.com\)](mailto:mike.sr@gmail.com)

Software architect
eBay

24 Mar 2009

Web applications have gotten more and more advanced, and users are always expecting more out of them. One of the most advanced features is Comet, also known as reverse Asynchronous JavaScript and XML (Ajax) or server-side push. Comet allows for browser-based instant messaging, real-time stock quotes, and so on. Advanced Ajax libraries, such as jQuery, make it easy to write Comet applications on the client side, but getting them to scale on the server is still a challenge. That is where the Scala programming language and the Lift Web application framework can step in and deliver a scalable back end for your Comet application. In this tutorial, build a real-time Web auction using these technologies.

Section 1. Before you start

This tutorial is for developers interested in writing Comet-style Web applications. A basic knowledge of Web applications and Ajax interactions is useful. Lift is written in the Scala programming language, which runs on top of the Java™ Virtual Machine. Prior knowledge of Scala is not necessary, but experience with Java is certainly useful. You will see some sophisticated Scala in this article, so familiarity with a functional programming language like Haskell, OCaml, or Lisp helps as well. You will also use the jQuery JavaScript library. None of the JavaScript is that advanced, so familiarity with basic JavaScript is sufficient.

About this tutorial

Alex Russell first coined the term Comet in a blog entry back in 2006. He defined Comet as event-driven, where the server has an open line of communication to push data to the client (see [Resources](#) to read the blog entry).

In this tutorial, you will develop a Comet-style Web application called Auction Net using Scala, Lift, and jQuery. You start by going over the design of the application that you will be building in this tutorial. After you know what you want to build, you will break it down into various parts and learn how you can leverage the features of Lift to implement these features.

Prerequisites

To develop with Lift and run the sample code, you will need the following tools:

- [Java Development Kit \(JDK\)](#). JDK 1.5.0_16 was used to develop the application.
- [Apache Maven](#). Maven 2.0.9 was used to develop this application.
- Get the latest source code for Lift from the [Google Code site](#).

Section 2. Auction Net

Before I start talking about the technical details of the implementations, let's first take a look at the functional design of the example application. You will build a simple e-commerce site that you will call Auction Net.

Functional design

Auction Net will be an auction Web application, as the name suggests. It will allow people to sell (list) items and for other users to buy (bid) on those same items. There are a lot of potential complexities in an auction Web application, but I will simplify things greatly so you can focus on how to use Scala, Lift, and jQuery to easily create a Comet-style application. After you see how easy it is to create a Comet application using Lift, it can be tempting to start using Comet everywhere. It is similar to when Ajax applications first started appearing. You often saw some sites overuse Ajax and create a less functional application.

This site needs to let people sign up for the application and become registered users, which is common functionality, and something that would not benefit from Comet. A typical registration/login process is sufficient. After the users have registered, they need to be able to sell an item, so you need a way to create new items and to list existing items. Again, this is something that would not greatly benefit from Comet-style interactivity. Finally, you want users to also bid on items. This is where Comet becomes a nice feature. You want a user to see the current high bid for an item, and have the bid update automatically when another user bids on it.

You should now have a basic idea of how you want the site to work. You can list out several domain objects: user, item, and bid. For each object you can also list the operations you want to be able to perform on them. These operations can be used as the basis for several pages, some of which can be very interactive using Comet. Now let's look at how you implement all of this by using Lift.

Section 3. Implementation

Lift is a complete Web application stack. It provides a full Model-View-Controller (MVC) implementation, though its approach is a little different than most run-of-the-mill MVC frameworks. It makes heavy use of Maven to build the project structure and satisfy dependencies. That is why you don't even have to download or install Scala to use Lift—it will do that for you. This is also why you don't need a database or Web server to use Lift; it uses Maven to include a database (Apache Derby) and a Web server (Jetty). In fact, Jetty is particularly good for Comet-style applications; a fact leveraged by Lift, leaving nothing else to download.

Creating a Lift application

As mentioned, Lift uses Maven for pretty much everything, including creating an application. It uses Maven archetypes to create the project structure. The command you will use is `mvn archetype:generate` along with the following parameters:

Table 1. Sample event data fields

Parameter	Value	Description
<code>archetypeGroupId</code>	<code>net.liftweb</code>	This is the namespace for the archetype you want to use.
<code>archetypeArtifactId</code>	<code>lift-archetype-basic</code>	This is ID for the archetype. In this case it specifies a "basic" application, see below.
<code>archetypeVersion</code>	<code>0.10-SNAPSHOT</code>	This is the version of the

		archetype, which corresponds to the version of Lift. See below.
remoteRepositories	http://scala-tools.org/repo-releases	This is the URL to the Maven repository that contains the archetype.
groupId	org.developerworks	The namespace for your application. You can change this value. All code will be in subpackages below this package.
artifactId	auctionNet	This is the name of your application.

There are a few things that deserve some extra explanation. First, you are using the *basic* archetype. There is also a *blank* archetype. If you use the blank archetype, Maven will generate the minimal Lift application. This will be the application structure and the minimal boot strapping code. Instead, you will use basic, which sets up Lift's ORM technology, called Mapper, as well its Comet framework. It will also create a user model and all of the code needed for the standard user management pages: registration, login, and forgot your password. Many applications need this, including this one, so you will use it. That's one less wheel to re-invent!

Next, notice the version of the archetype that you will use. This corresponds to the version of Lift that you are using. In this case, I used a snapshot of 0.10. The latest official release at the time of writing was 0.9. However, there were some nice features available in 0.10, so I opted for the "bleeding edge." The downside is that some of the code you see here is available for download, but it may need to be tweaked slightly to work for the official 0.10 release. Also note that I used the "repo-snapshots" repository because I used a snapshot. There is a "repo-releases" repository for the official releases. Now that you understand the command that you are going to issue to Maven, let's run the command and see what it does. The full command and the output is shown in Listing 1.

Listing 1. Creating a Lift project command with Maven

```
$ mvn archetype:generate -DarchetypeGroupId=net.liftweb
-DarchetypeArtifactId=lift-archetype-basic -DarchetypeVersion=0.10-SNAPSHOT
-DremoteRepositories=http://scala-tools.org/repo-snapshots
-DgroupId=org.developerworks.lift -DartifactId=auctionNet
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'archetype'.
[INFO] -----
[INFO] Building Maven Default Project
[INFO]   task-segment: [archetype:generate] (aggregator-style)
[INFO] -----
[INFO] Preparing archetype:generate
[INFO] No goals needed for project - skipping
[INFO] Setting property: classpath.resource.loader.class =>
'org.codehaus.plexus.velocity.ContextClassLoaderResourceLoader'.
[INFO] Setting property: velocimacro.messages.on => 'false'.
```

```

[INFO] Setting property: resource.loader => 'classpath'.
[INFO] Setting property: resource.manager.logwhenfound => 'false'.
[INFO] [archetype:generate]
[INFO] Generating project in Interactive mode
[INFO] Archetype repository missing. Using the one from
[net.liftweb:lift-archetype-basic:RELEASE -> http://scala-tools.org/repo-releases]
found in catalog internal
[INFO] snapshot net.liftweb:lift-archetype-basic:0.10-SNAPSHOT:
checking for updates from lift-archetype-basic-repo
[INFO] snapshot net.liftweb:lift-archetype-basic:0.10-SNAPSHOT:
checking for updates from scala-tools.org
[INFO] snapshot net.liftweb:lift-archetype-basic:0.10-SNAPSHOT:
checking for updates from scala-tools.org.snapshots
Downloading: http://scala-tools.org/repo-snapshots/net/liftweb/lift-archetype-basic/
0.10-SNAPSHOT/lift-archetype-basic-0.10-SNAPSHOT.jar
15K downloaded
Define value for version: 1.0-SNAPSHOT: :
Confirm properties configuration:
groupId: org.developerworks.lift
artifactId: auctionNet
version: 1.0-SNAPSHOT
package: org.developerworks
Y: : y
[INFO] -----
[INFO] Using following parameters for creating OldArchetype: lift-archetype-basic:
0.10-SNAPSHOT
[INFO] -----
[INFO] Parameter: groupId, Value: org.developerworks.lift
[INFO] Parameter: packageName, Value: org.developerworks.lift
[INFO] Parameter: basedir, Value: /Users/michael/code/lift/auction2
[INFO] Parameter: package, Value: org.developerworks.lift
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] Parameter: artifactId, Value: auctionNet

```

This command does everything you need, downloading all the libraries you will need as well as setting up your project structure. You can immediately start your application by running `mvn jetty:run` as seen in Listing 2.

Listing 2. Starting the application

```

$ cd auctionNet/
$ mvn jetty:run
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'jetty'.
[INFO] -----
[INFO] Building auctionNet
[INFO]   task-segment: [jetty:run]
[INFO] -----
[INFO] Preparing jetty:run
[INFO] [resources:resources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [yuicompressor:compress {execution: default}]
[INFO] nb warnings: 0, nb errors: 0
[INFO] Context path = /
[INFO] Tmp directory = determined at runtime
[INFO] Web defaults = org/mortbay/jetty/webapp/webdefault.xml
[INFO] Web overrides = none
[INFO] Webapp directory = /Users/michael/code/lift/auctionNet/src/main/webapp
[INFO] Starting jetty 6.1.10 ...
2008-12-06 18:11:43.621::INFO: jetty-6.1.10
2008-12-06 18:11:44.844::INFO: No Transaction manager found - if your webapp
requires one, please configure one.
INFO - CREATE TABLE users (id BIGINT NOT NULL GENERATED ALWAYS AS IDENTITY , firstname

```

```

VARCHAR(32) , lastname VARCHAR(32) , email VARCHAR(48) , locale VARCHAR(16) , timezone
VARCHAR(32) , password_pw VARCHAR(48) , password_slt VARCHAR(20) , textarea
VARCHAR(2048) , superuser SMALLINT , validated SMALLINT , uniqueid VARCHAR(32))
INFO - ALTER TABLE users ADD CONSTRAINT users_PK PRIMARY KEY(id)
INFO - CREATE INDEX users_email ON users ( email )
INFO - CREATE INDEX users_uniqueid ON users ( uniqueid )
2008-12-06 18:11:47.199::INFO: Started SelectChannelConnector@0.0.0.0:8080
[INFO] Started Jetty Server
[INFO] Starting scanner at interval of 5 seconds.

```

You will notice some libraries being loaded, and all of the code generated in the previous goal will be compiled and packaged into a WAR file. That info has been deleted from Listing 2. What is important to notice is that Lift will create your database for you. Remember I mentioned that because you used the basic archetype you got a user management system for free? The database table for that is created in Listing 2. Now you have your users already, with no extra work needed. You just need to create domain models for items and bids.

Domain modeling with Mapper

Lift provides its own object relational modeling (ORM) technology, known as Mapper. As you saw in the previous example, you are already using it for user management. The generated `User` code can be found in `<groupId>.model.User`, where `groupId` is what you used earlier when generating the application using Maven. In this case, that is `org.developerworks` so the user model is in `org.developerworks.model.User`. All Scala code can be found in `/auctionNet/src/main/scala`. The user model is shown in Listing 3.

Listing 3. Default Lift User model

```

/**
 * The singleton that has methods for accessing the database
 */

object User extends User with MetaMegaProtoUser[User] {
  override def dbTableName = "users" // define the DB table name
  override def screenWrap = Full(<lift:surround with="default"
at="content"><lift:bind /></lift:surround>)
  // define the order fields will appear in forms and output
  override def fieldOrder = id :: firstName :: lastName :: email ::
    locale :: timezone ::
    password :: textArea :: Nil
  // comment this line out to require email validations
  override def skipEmailValidation = true
}

/**
 * An O-R mapped "User" class that includes first name, last name, password
 * and we add a "Personal Essay" to it
 */

class User extends MegaProtoUser[User] {
  def getSingleton = User // what's the "meta" server
  // define an additional field for a personal essay

```

```

object textArea extends MappedTextarea(this, 2048) {
  override def textareaRows = 10
  override def textareaCols = 50
  override def displayName = "Personal Essay"
}
}

```

This code serves as a good way to understand Lift's Mapper API. Start at the bottom of Listing 3 with the `User` class. This is extending an existing class called `MegaProtoUser`. You can take a look at this class in the Lift source code, but as the comments in the code indicate, it provides a first name, last name, and password. This code shows how you can customize the user. In this case, it adds a "Personal Essay" that is mapped to a database column called `textArea`.

One of the unusual things about Mapper is that the fields (database columns) in the mapped class are not typical Scala fields, like a `var` or (if the field is immutable) a `val`. Instead they are Scala objects, that is, singletons. You can think of them as singletons that are nested inside an enclosing class, so you can have multiple `Users` (as it is a class), but each `User` can have exactly one `textArea` object. The advantages of using an object can be seen in the `User` class. Your object extends the Lift class `net.liftweb.mapper.MappedTextarea`. By subclassing an existing class, you can override behavior to customize the field. In the `User` class, you do this to change how this field will be represented as an HTML `TextArea` element. That is right, all of the Mapper field types (`MappedString`, `MappedLong`, and so on) all have an HTML representation built in. For example, the `MappedTextarea` class is shown in Listing 4.

Listing 4. Lift's MappedTextarea class

```

class MappedTextarea[T<:Mapper[T]](owner : T, maxlen: Int) extends
  MappedString[T](owner, maxlen) {

  /**
   * Create an input field for the item
   */
  override def _toForm: Can[NodeSeq] = {
    val funcName = S.mapFunc({s: List[String] => this.setFromAny(s)})
    Full(<textarea name={funcName}
      rows={textareaRows.toString}
      cols={textareaCols.toString} id={fieldId}>{is.toString}</textarea>)
  }
  override def toString = {
    val v = is
    if (v == null || v.length < 100) super.toString
    else v.substring(0,40)+" ... "+v.substring(v.length - 40)
  }
  def textareaRows = 8
  def textareaCols = 20
}

```

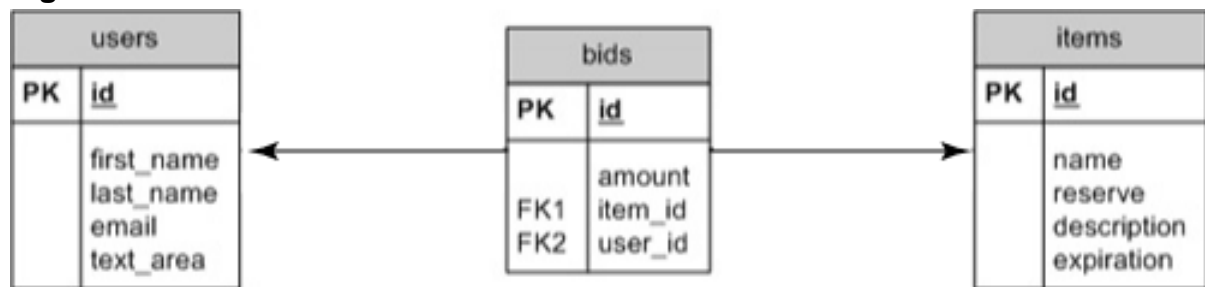
As you can see, a `MappedTextarea` extends `MappedString`, and thus, will be treated like a string when being mapped to a database column. It has a `_toForm`

method that uses Scala's XML syntax and Lift's helper functions to create a typical HTML TextArea. In your `User` class you did an override of the number of rows and columns as well as the display name. When subclassing a mapped type, you have a lot of power. It is a great place to add custom validation logic. Let's say you didn't want to allow any HTML characters in the essay, then you could plug in a regular expression to check for this. The `MappedString` class has methods for executing regular expressions, and for other common things like checking minimum and maximum lengths, and even checking the uniqueness of a string against the database. Of course, you can do more sophisticated logic because you are subclassing the type and can add any code you like.

Hopefully, the `User` class makes sense. If you go back to the `User` class in Listing 3, notice the method called `getSingleton`. This method is returning the `User` object defined above the `User` class. This object represents the metadata about the `User` class and how it is mapped to the database. The common things to do here are to define the name of the database table and the fields to be shown for listing `Users` or for generating a `User` form. Also, the `User` object gives you a place to attach methods that are more class-level, like factory and finder methods. Scala does not have static variables and methods, so these cannot be associated directly to the `User` class.

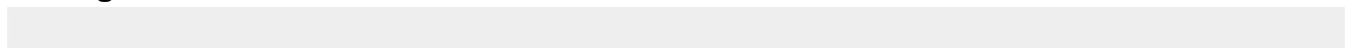
You might have noticed that the object and the class are both called `User`. This is a common paradigm in Scala, known as companion objects. For your custom models, you will break with this convention slightly to make some of the code more obvious. You will append a "MetaData" suffix with this in mind. Now that you have an understanding of Lift's Mapper API, let's create some custom models. First take a look at Figure 1, which shows the data model that you want to create.

Figure 1. Auction Net data model



This is a very simple model of an auction. An item has a name and description, as well as a reserve (minimum) price and expiration date. Any item can be bid on by any user, so you have a classic many-to-many relationship. In a relational database schema the bids act like a join table, but they are actually meaningful in their own right because each bid also has an amount. Now that you know what you want to model (in code) let's take a look at the `Item` model. It is shown in Listing 5.

Listing 5. The Item model



```

object ItemMetaData extends Item with KeyedMetaMapper[Long, Item]{
  override def dbName = "items"
  override def fieldOrder = List(name, description, reserve, expiration) }
class Item extends KeyedMapper[Long, Item] with CRUDify[Long, Item] {
  def getSingleton = ItemMetaData
  def primaryKeyField = id

  object id extends MappedLongIndex(this)
  object reserve extends MappedInt(this)
  object name extends MappedString(this, 100)
  object description extends MappedText(this)
  object expiration extends MappedDateTime(this)

  lazy val detailUrl = "/details.html?itemId=" + id.is
  def bids = BidMetaData.findAll(By(BidMetaData.item, id))

  def tr = {
    <tr>
      <td><a href={detailUrl}>{name}</a></td>
      <td>{highBid.amount}</td>
      <td>{expiration}</td>
    </tr>
  }
  def highBid:Bid = {
    val allBids = bids
    if (allBids.size > 0){
      return allBids.sort(_._amount.is > _._amount.is)(0)
    }
    BidMetaData.create
  }
}
}

```

Again, let's start with the `Item` class. Most of what you see is pretty standard. You define each of the fields as objects that extend a mapped data type (long, integer, string, date, and so on). You define another attribute, `detailUrl`, as a lazy val. This just means that it is an immutable value that is not calculated until it is invoked. This does not do much here, but can be a useful syntax for cases where the evaluation is complex and not always invoked. You have a `bids` method that queries all of the bids, and you'll look at the `Bid` class soon. You also have a method called `tr` that represents your item as a row in an HTML table.

Finally, you have a `highBid` method that gets the highest bid by sorting all of the bids retrieved from the database. This sort could have been done by the database, but it demonstrates how easy Scala makes it to do common things like sort a list. You passed a closure to the sort method, and you used Scala's closure shorthand syntax. The underscores (`_`) is "filled in" by the parameters being passed to the closure. For a list of bids, this will be two items for comparison. You want to sort the bids by the bid amount, hence `_.amount.is`. The last part (`.is`) calls the `is` method on the amount object. Remember, you have a complex object here, not just a flat field, so the `is` method retrieves the value of the field. Finally, after you sort, notice the `(0)` syntax. The sort yields a list of bids. You want the first element of that list, and `(0)` gives you that. Again, this is Scala shorthand. You are actually calling the `apply` method on the list with a value of `0`. The list is treated like a function, and this

is always syntactic sugar for the `apply` method.

I have explained most of the guts of the `Item` class, but not the actual declaration of the class. You first declare that `Item` extends the Lift trait `KeyedMapper`. This is a parameterized trait, where the parameters are the type of the primary key of the mapped class and the mapped class itself. Notice that you extend `KeyedMapper` with another trait called `CRUDify`. You are leveraging Scala's mix-in model to simulate multiple inheritance. You want the behavior of `KeyedMapper` and of `CRUDify`. Notice that `CRUDify` is also a parameterized trait. This is another common paradigm in Scala. Traits are parameterized to allow them to be type safe. If you look at mix-ins in other languages (Python and Ruby, for example), they are usually either trivial or they have a requirement (like the existence of certain fields or methods) but have no way of expressing this. You could mix them in with your class and not realize that you need to modify your class to use the mix-in until you start getting runtime errors. Parameterized traits in Scala avoid this problem.

Your `Item` model has both the `KeyedMapper` and `CRUDify` trait. The `KeyedMapper` trait maps it to a database table, but what about `CRUDify`? As the name suggests, this provides basic CRUD: Create, Read, Update, and Delete functions for the model. Lift will create all of the boilerplate code (including UI) for showing a list of `Items`, creating a new `Item`, editing an existing `Item`, or deleting an `Item`. In the functional design you said you needed to be able to list new items, and `CRUDify` gives you a cheap way to do this. No extra code to write, just an extra trait to use. Now that you have seen the `Item` model, take a look at the `Bid` model. It is shown in Listing 6.

Listing 6. The Bid model

```
object BidMetaData extends Bid with KeyedMetaMapper[Long, Bid]{
  override def dbName = "bids"
  override def fieldOrder = amount :: Nil
  override def dbIndexes = Index(item) :: Index(user) :: super.dbIndexes
}
class Bid extends KeyedMapper[Long, Bid]{
  def singleton = BidMetaData
  def primaryKeyField = id
  object id extends MappedLongIndex(this)
  object amount extends MappedLong(this)
  object item extends MappedLongForeignKey(this, ItemMetaData)
  object user extends MappedLongForeignKey(this, User)
}
```

This is a simpler class. Notice that you have an `Item` object and a `User` object. These objects extend Lift's parameterized class `MappedLongForeignKey`. Notice how you passed in the metadata objects (`ItemMetaData` from Listing 5 and the `User` object from Listing 3) to indicate what you were joining to. Also notice that in your metadata object, you specified database indexes on the two foreign key columns, anticipating that you will query bids based on either an item or a user. Now

you have your domain model defined, so you are ready write some code that uses it.

Actors

I used Lift's `CRUDify` trait to handle item management and Lift's out-of-the-box support for user management, so all you need to build is a bidding system. You could do this with normal controller/CRUD code, but you want to make this a Comet system, so you'll use Scala's concurrency stack, Actors. An Actor is like a light-weight thread, but with no shared memory, so there is never any need for synchronization, locking, and so on. Actors communicate with messages. Scala's combination of case classes (essentially, typed data structures) and pattern matching makes it easy to listen and respond to messages. You will first create an Auctioneer actor. It will listen for messages to bid on an item, and will dispatch messages stating that there is a new bid on an item. Let's start by looking at the types of messages that you will use. The messages are shown in Listing 7.

Listing 7. Auction messages

```
case class AddListener(listener:Actor, itemId:Long)
case class RemoveListener(listener:Actor, itemId:Long)
case class BidOnItem(itemId:Long, amount:Long, user:User)
case class GetHighBid(item:Item)
case class TheCurrentHighBid(amount:Long, user:User)
case class Success(success:Boolean)
```

The first two messages are just for adding and removing listeners, based on an Item ID. You will send an `AddListener` message to the Auctioneer to say that you are interested in a particular Item. When you want to bid, you will send a `BidOnItem` message. When a new bid is made, you want the Auctioneer to send out a new `TheCurrentHighBid` message. Finally, the `Success` message is used to indicate that an `AddListener` request was a success. Now you will be able to do pattern matching against these strongly typed objects. Let's take a look at the Auctioneer in Listing 8.

Listing 8. The Auctioneer actor

```
object Auctioneer extends Actor{
  val listeners = new HashMap[Long, ListBuffer[Actor]]
  def notifyListeners(itemId:Long) = {
    if (listeners.contains(itemId)){
      listeners(itemId).foreach((actor) => {
        val item = ItemMetaData.findByKey(itemId).open_!
        actor ! highBid(item)
      })
    }
  }
  def act = {
    loop {
      react {
```

```

    case AddListener(listener:Actor, itemId:Long) =>
      if (!listeners.contains(itemId)){
        listeners(itemId) = new ListBuffer[Actor]
      }
      listeners(itemId) += listener
      reply(Success(true))
    case RemoveListener(listener:Actor, itemId:Long) =>
      listeners(itemId) -= listener
    case GetHighBid(item:Item) =>
      reply(highBid(item))
    case BidOnItem(itemId:Long, amount:Long, user:User) =>
      val item =
        ItemMetaData.findAll(By(ItemMetaData.id, itemId)).firstOption.get
      val bid = BidMetaData.create
      bid.amount(amount).item(item).user(user).save
      notifyListeners(itemId)
  }
}

def highBid(item:Item):TheCurrentHighBid = {
  val highBid = item.highBid
  val user = highBid.user.obj.open_!
  val amt = highBid.amount.is
  TheCurrentHighBid(amt, user)
}
start
}

```

The Auctioneer keeps track of who is interested in each item by keeping a map. The key to the map is the ID of an Item, and the value is a list of interested Actors. The main part of any Actor is its `act` method. This is an abstract method in the Actor trait that you must implement. The `loop -> react` construct is typical of Actors. The function `loop` is defined in the `scala.actors.Actor` object; it takes a closure and repeatedly executes it. There is also a `loopWhile` that takes a predicate and loops as long the predicate is true. This is an easy way to provide a hook for shutting down an Actor. The `react` method is defined in the `scala.actors.Actor` trait. It receives a message and executes the closure that is passed to it. Inside that closure is where you use Scala's pattern matching. You match against the type of message that could come in. In particular, when a `BidOnItem` message is received, you save the new bid to the database, and then notify listeners.

The `notifyListeners` method uses the map of Item ID to get all of the Actors that are interested in a particular Item. It then sends a new `TheCurrentBid` message to each interested Actor. That is what the `actor ! highBid(item)` code does. (This could actually be written as `actor.!(highBid(item))`.) In other words, there is a method called `!` on the Actor class. The `actor ! highBid(item)` syntax is nicer looking, and is consistent with how other languages (like Erlang) implement actors. These languages have specific syntactical support for actors, but Scala does not. Actors are essentially a Domain Specific Language (DSL) built on top of Scala using its powerful syntax.

If you go back to your Auctioneer, the last thing of note is the last line of code.

That is the `start` method being called, and does just what it says, it starts the Actor, causing its `act` method to be invoked asynchronously. Your `Auctioneer` will run forever, sending and receiving messages from other Actors. If you are used to concurrent programming in Java, this is a lot different, but in many ways much simpler. Now the obvious question is who is going to be sending and receiving messages to the `Auctioneer`? After all, it would be pretty uninteresting to have a single Actor in an application. To answer this question, you will use Lift's `CometActors`.

CometActors

Lift's `CometActor` trait is an extension of Scala's `Actor` trait. It makes it easy to use an Actor as part of a Comet application. The `CometActor` can react to messages from other Actors to send UI updates to the user. The Maven archetype creates a comet package, and any `CometActors` you put in there will automatically be available to your application. For Auction Net, you will create a `CometActor` to make bids and to receive updates on new high bids. I call this Actor `AuctionActor`, and it is shown in Listing 9.

Listing 9. The Auction Actor

```
class AuctionActor extends CometActor {
  var highBid : TheCurrentHighBid = null
  def defaultPrefix = "auction"
  val itemId = S.param("itemId").map(Long.parseLong(_)).openOr(0L)
  override def localSetup {
    Auctioneer !? AddListener(this, this.itemId) match {
      case Success(true) => println("Listener added")
      case _ => println("Other ls")
    }
  }
  override def localShutdown {
    Auctioneer ! RemoveListener(this, this.itemId)
  }
  override def lowPriority : PartialFunction[Any, Unit] = {
    case TheCurrentHighBid(a,u) => {
      highBid = TheCurrentHighBid(a,u)
      reRender(false)
    }
    case _ => println("Other lp")
  }
}
```

This listing shows the life cycle aspects of the `CometActor`. When it is invoked, the `localSetup` method will be called. This uses the `itemId` attribute to send a `AddListener` message to the `Auctioneer` Actor. Notice that I used `!?` for sending the message. This is a synchronous call. You `AuctionActor` is of no use to you until the `Auctioneer` knows that it is interested in a particular Item. When the reply comes back from `Auctioneer`, you use pattern matching to decide what to do with it, and simply log the information. When a `CometActor` goes out of use,

the `localShutdown` method is called. This simply sends a `RemoveListener` message to the `Auctioneer`. During the `Auction Actor`'s lifetime, it listens for messages using the `lowPriority` method (there is also a `highPriority` method, and so on) Again, the `Auction Actor` uses pattern matching when you receive a message. It looks for a `TheCurrentHighBid` message and reacts to it by keeping track of the high bid and calling `reRender`. This is a method defined on `CometActor`. I omitted the rendering code in Listing 9 so you could concentrate on the life cycle. Now let's look at the rendering code in Listing 10.

Listing 10. Auction Actor rendering

```
class AuctionActor extends CometActor {
  var highBid : TheCurrentHighBid = null
  def defaultPrefix = "auction"
  val itemId = S.param("itemId").map(Long.parseLong(_)).openOr(0L)
  def render = {
    def itemView: NodeSeq = {
      val item = if (itemId > 0)
        ItemMetaData.findByKey(itemId).openOr(ItemMetaData.create)
      else ItemMetaData.create
      val currBid = item.highBid
      val bidAmt = if (currBid.user.isEmpty) 0L else currBid.amount.is
      highBid = TheCurrentHighBid(bidAmt,
        currBid.user.obj.openOr(User.currentUser.open_!))
      val minNewBid = highBid.amount + 1L
      val button = <button type="button">{S.?("Bid Now!")}</button> %
        ("onclick" -> ajaxCall(JsRaw("$('#newBid').attr('value')"), bid _))
      (<div>
        <strong>{item.name}</strong>
        <br/>
        <div>
          Current Bid: ${highBid.amount} by {highBid.user.niceName}
        </div>
        <div>
          New Bid (min: ${minNewBid}) :
          <input type="text" id="newBid"/>
          {button}
        </div>
        {item.description}<br/>
      </div>)
    }
    bind("foo" -> <div>{itemView}</div>)
  }
  def bid(s:String): JsCmd = {
    val user = User.currentUser.open_!
    Auctioneer ! BidOnItem(itemId, Long.parseLong(s), user)
    Noop
  }
}
```

The `render` method is an abstract method on the `CometActor` trait that you must implement. In this method, you look up the item and then create a fragment of XHTML to send back to the user. The one really interesting piece of code in here is the button value, which creates a basic button that says "Bid Now!." The `S.?` function allows the string to be localized, in case you were wondering. The `%` method adds an attribute to the element. In this case, you create a JavaScript function to respond to the `onclick` event from the button. The `ajaxCall` function is defined in

Lift's `SHtml` helper object. The first parameter it requires is a `JsExp` (JavaScript expression) instance. You use the `JsRaw` object to wrap a raw string of JavaScript and create a `JsExp`.

The raw JavaScript you pass is pretty simple if you are familiar with jQuery. The jQuery library is included by default with Lift. The `$` function in jQuery takes a CSS-style selector and returns the elements from the DOM tree that satisfy the selector. In this case, you pass in `#newBid`, which in CSS specifies an element with an ID of `newBid`. The jQuery library adds an `attr` function to DOM elements. This provides an easy way to get the values of attributes on the element. In this case, you want the `value` attribute. If you look further down in the code, you will see that the `newBid` element is a text input field, so the value attribute will be whatever the user enters into the text input field.

The jQuery expression will be evaluated and its value will be passed to the Ajax call. The second parameter of the `ajaxCall` function is a closure that takes a string and returns an instance of Lift's `JsCmd` type. Here you once again use Scala's shorthand syntax for the closure passing the string to the `bid` function that you defined. The `bid` function gets the current user, parses the string into a long, and uses these two values to construct a `BidOnItem` message. It sends this message to the `Auctioneer`. It then returns Lift's `Noop` object. This is a `JsCmd` that indicates `noop`, that is, do nothing.

Now you might have also noticed that the `bid` method used the `itemId` as part of the `BidOnItem` message. You might be wondering if this value is still around when the user bids on an item. This is the beauty of closures; they retain the enclosing context from when they were created. Whenever the closure passed into `ajaxCall` is executed, it will "remember" all of the data that it had access to when it was created.

Now you have seen how the `Auction` Actor works, and you just need to create a page that uses it. This will be your item details page, and it is shown in Listing 11.

Listing 11. The item details view

```
<lift:surround with="default" at="content">
  <lift:comet type="AuctionActor">
    <auction:foo>Loading...</auction:foo>
  </lift:comet>
</lift:surround>
```

As you can see, this is a very simple page. You use Lift's default page layout and just drop in a snippet for the `Auction` Actor. If you are curious about the `auction:foo` tag, look back at listing 10. There I defined the `defaultPrefix` for the `CometActor` as `auction` and used a `bind` command to bind the XHTML created in `render` to `foo`, hence `auction:foo`. This is loaded asynchronously to

the page, so you put in the "Loading..." text as a placeholder until the `CometActor` has been loaded.

To add the page to the site, you need to add it to the application's `SiteMap`. This is another Lift construct that is a white list of pages that can be accessed, and allows Lift to construct navigation menus, breadcrumbs, and so on. It is specified in Lift's `Boot` class, as shown in Listing 12.

Listing 12. Lift Bootstrap code

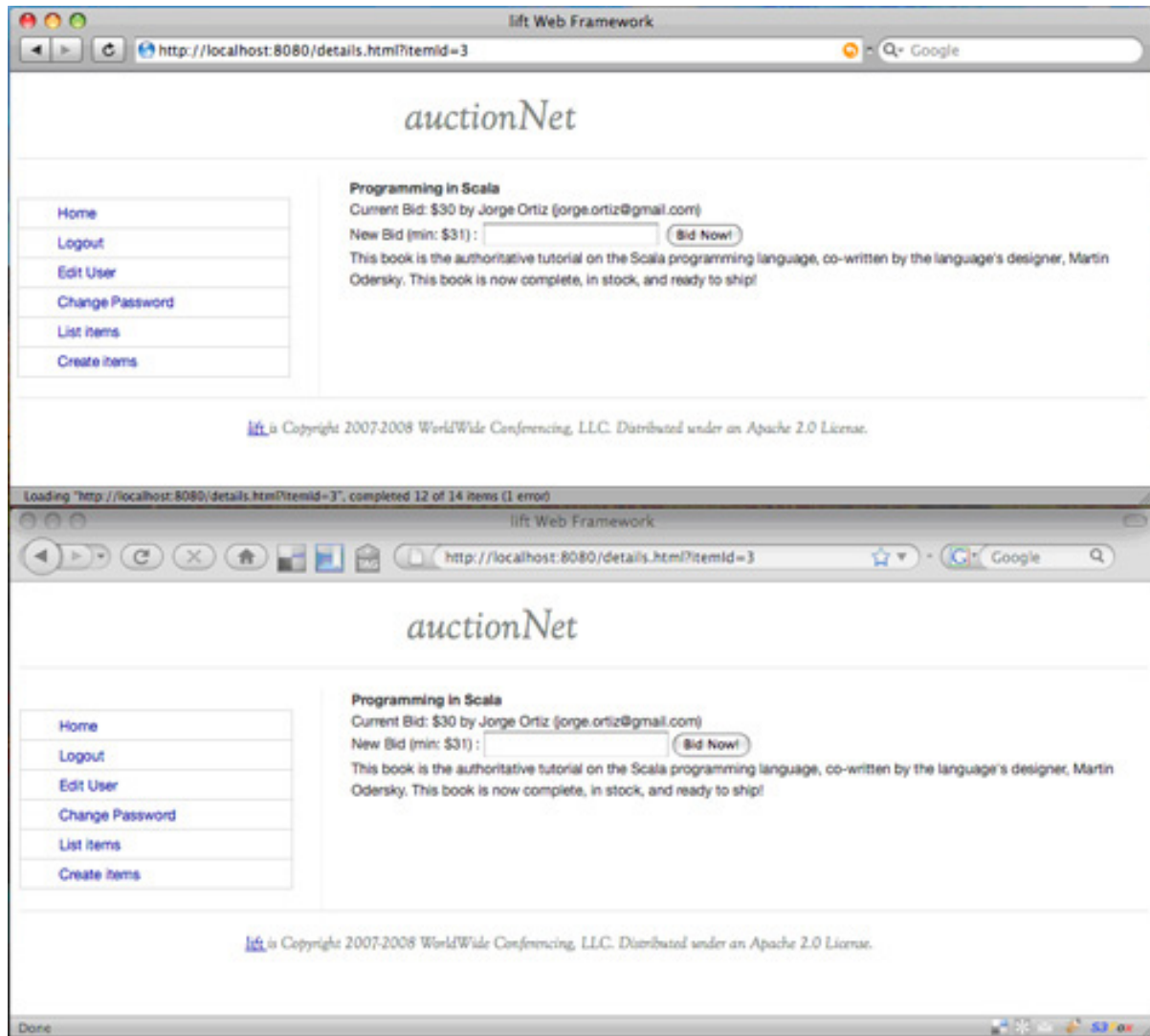
```
class Boot {
  def boot {
    if (!DB.jndiJdbcConnAvailable?)
      DB.defineConnectionManager(DefaultConnectionIdentifier, DBVendor)
    // where to search snippet
    LiftRules.addToPackages("org.developerworks")
    Schemifier.schemify(true, Log.infoF "_", User, ItemMetaData, BidMetaData)
    // Build SiteMap
    val entries:List[Menu] = Menu(Loc("Home", List("index"), "Home")) ::
      Menu(Loc("Item Details", List("details"), "Item Details", Hidden)) ::
      User.sitemap ++ ItemMetaData.menus
    LiftRules.setSiteMap(SiteMap(entries:_*))
  }
  /*
   * 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)
  LiftRules.appendEarly(makeUtf8)
  S.addAround(DB.buildLoanWrapper)
}
/**
 * Force the request to be UTF-8
 */
private def makeUtf8(req: HttpServletRequest) {
  req.setCharacterEncoding("UTF-8")
}
}
```

This file (`Boot.scala`) also contains code for changing your database setting, but none of that was changed for this application. Most of the above code was generated as part of the Maven archetype. There are two main things that were changed. First, the `Item` and `Bid` metadata objects were added to the call to the `Schemifier`. This is what will create your database for you. Next, in the entries that are used to build the `SiteMap`, you added a new location for the item details page and also added the CRUD pages for the `Item`. That is it for the code, so you are ready to run the applicaiton.

Running the application

If you left the application running, you can just run `mvn install` to recompile the code. If it isn't running, just run `mvn jetty:run` again to start it back up. You should be able to access the page at `http://localhost:8080`. You can register, create some items, and start bidding. To view the Comet features in action, use two browsers, log in as a different user in each browser, and then view the same item. Each user can bid on the item, and both will be updated when the other bids. A screenshot of this is shown in Figure 2.

Figure 2. Bidding on Auction Net



If you are curious about what is going on when you bid, you can use something like Firefox's Firebug to watch the HTTP traffic. Listing 13 shows sample output.

Listing 13. Comet traffic

```
try { destroy_LC2EAICJRLWEKDM4EXIPE2(); } catch (e) {}
```

```
try{jQuery('#LC2EAICJRLWEKDM4EXIPE2').each(function(i) {
  this.innerHTML = '\u000a <div><div>\u000a
  <strong>Programming in Scala</strong>\u000a';});} catch (e) {}
try { /* JSON Func auction $$ F1229133085612927000_NGB */
  function F1229133085612927000_NGB(obj) {
    lift_ajaxHandler('F1229133085612927000_NGB='+
    encodeURIComponent(JSON.stringify(obj)),
    null,null);} } catch (e) {}
try { destroy_LC2EAICJRLWEKDM4EXIPE2 = function() {}; } catch (e) {}

lift_toWatch['LC2EAICJRLWEKDM4EXIPE2'] = '11';
```

As you can see from Listing 13, Lift sends back a JavaScript script to execute. This script uses jQuery once again to replace the contents of the original rendering with the new rendering. That is what the `jQuery('#...').each` expression does. It finds the element with that very ugly looking ID (randomly generated by Lift for security purposes; another nice feature of Lift) and passes in a closure to each element it found. This closure replaces the innerHTML with the HTML generated on the server. The full HTML in Listing 13 was truncated for the sake of brevity and readability. It then initiates another Comet session. After looking at this, you can be happy that this was automatically created by Lift, and you did not have to do it yourself!

Section 4. Summary

In this tutorial, you created a Comet-style Web application for auctions, using Lift to make this task very easy. You started from scratch and saw how to begin with Lift using Maven. You also looked at creating domain models using Lift's Mapper APIs and had a chance to look at one of the more powerful features of Scala, its Actors library. You used Actors to send and receive messages about your auction system, and you used Lift's `CometActors` to turn an Actor into a Comet endpoint, receiving Ajax calls and sending back JavaScript to the client. Finally, you saw how to use jQuery with Lift, and how Lift's Ajax and Comet stack uses jQuery as well.

Downloads

Description	Name	Size	Download method
Sample code for this article	auctionNet.zip	14KB	HTTP

[Information about download methods](#)

Resources

Learn

- "[Comet: Low Latency Data for the Browser](#)": Alex Russell, project lead for The Dojo Toolkit and president of the Dojo Foundation, coined the term Comet in this blog entry.
- If you are new to Scala, you will definitely want to check out Ted Neward's series of developerWorks articles: "[The busy Java developer's guide to Scala](#)" (developerWorks).
- Did you like how you could write HTML directly inside a Scala classes in Lift? Learn about the secret sauce in the developerWorks article "[Scala and XML](#)" (developerWorks, April 2008).
- You can easily deploy a Lift application to any application server. See how in "[Give Geronimo a Lift](#)" (developerWorks, July 2008).
- For all questions related to Lift, check out the [official wiki of Lift](#).
- See how you can use Python and the Prototype JavaScript library to create a Comet application that can be deployed to the Google App Engine in "[Creating mashups on the Google App Engine](#)" (developerWorks, August 2008).
- You can run Comet applications on Java Web servers other than Jetty. See an example in "[Developing Rich Internet Applications for WebSphere Application Server Community Edition](#)" (developerWorks, September 2008).
- If you want to stick with Java, then you might want to look into Direct Web Remoting for creating Comet applications. See how in "[Ajax for Java developers: Write scalable Comet applications with Jetty and Direct Web Remoting](#)" (developerWorks, July 2007).
- Get an introduction to jQuery in the article "[Simplify Ajax development with jQuery](#)" (developerWorks, April 2007).
- [developerWorks technical events and webcasts](#): Stay current with developerWorks technical events and webcasts.
- [Podcasts](#): Tune in and catch up with IBM technical experts.

Get products and technologies

- Java technology is a programming language and a platform. Download the [Java Development Kit \(JDK\)](#) from Sun Microsystems or from IBM.
- [Maven](#), an Apache project, is a software project-management and comprehension tool.
- You can get the latest source code for Lift from its [Google Code site](#).

About the author

Michael Galpin

Michael Galpin has been developing Java software professionally since 1998. He currently works for eBay. He holds a degree in mathematics from the California Institute of Technology.