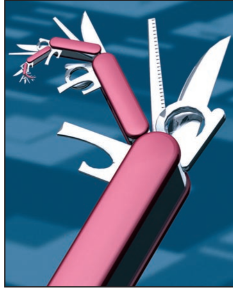


Scala and Lift — Functional Recipes for the Web

Debasish Ghosh • *Anshinsoft*
Steve Vinoski • *Verivue*

Given the Web's position as the ubiquitous global network for exchanging information and data, we face new challenges daily on how best to develop software for it. Imperative languages such as Java, C#, PHP, and Perl currently dominate server-side programming for the Web, but these languages often lack the appropriate levels of abstraction required for handling typical Web interactions. In today's age of programming language renaissance, the search is on for a language and Web framework that would let us model a Web application as a composition of referentially transparent functional abstractions, alongside safe handling of mutable state.

A common problem found in today's Web programming models is the use of string values: interactions between Web applications (including browsers) and Web servers occur primarily via textual representations such as JavaScript Object Notation (JSON) and via name-value string pairs. As a result, developers sometimes have to write a lot of code dedicated to validating unstructured strings at multiple layers of interaction. A framework that can abstract the request-response cycle via typed representations could help alleviate this problem to a great extent.

Another problem is the impedance mismatch between programming languages and Web media representations. XHTML, for example, is a monolithic structure, and most of today's Web frameworks make programmers write error-prone boilerplates to map form elements to code variables. Better framework support is also required to make such mapping more abstract and secure, and to ensure a complete separation of the presentation template from actual business logic.

Still another issue is the architectural mismatch between Web applications and what mainstream languages can practically support. For example, many developers use event-driven architecture to model Web applications. However, today's mainstream languages support event-driven programming via event listeners and callbacks, which result in an inversion of control of the main execution model.[1] Lightweight, event-based abstractions can provide a scalable way to model interactions on the Web.

In this column, we explore the applicability of the programming language Scala (www.scala-lang.org) together with Lift (http://liftweb.net) as a development framework for the Web. Scala offers strong functional capabilities, and Lift exploits Scala's benefits, described later, to implement a typed, extensible, event-based Web programming model.

What is Scala?

Scala is a hybrid object-oriented (OO) and functional language developed by Martin Odersky — one of the original authors of the Java compiler and currently a professor at Ecole Polytechnique Fédérale de Lausanne. Scala runs on the Java virtual machine (JVM), compiles to `.class` files, and is completely interoperable with the Java programming environment. Scala's syntax is lightweight, expressive, and concise due to semicolon inference, type inference, and the ability to define new control abstractions via closures.

Various features make Scala a viable choice for a future mainstream language on the JVM:

- Extensibility — supports composition of data structures via mixins and self-type annotations.
- Statically checked duck typing via structural typing — like dynamic languages but in a type-safe way.
- Higher-order functions — first-class functions that other functions can pass as argu-

 1089-7801/09/$25.00 © 2009 IEEE

ments and accept as return types. Scala has lexical closures, the bedrock of functional programming support. Scala's closure support leads to easy development of control abstractions and domain-specific languages.
- Immutable data structures — included as part of the standard library, which encourage developers to design referentially transparent abstractions.
- Advanced generator constructs — for example, for-comprehension that makes code more expressive and succinct.
- Pattern matching over abstract data types — patterns in Scala are represented internally as partial functions, which developers can compose using various combinators to construct extensible abstractions.
- Event-driven programming via the actor model.

One way to visualize the Web programming model is as a sequence of interactions based on events. Rich, immutable functional abstractions can act as a nice dual to asynchronous message-based concurrency in modeling this paradigm, and Scala offers both. Next, we discuss Scala actors, which offer a scalable, event-based message-passing concurrency model that the Lift framework uses extensively.

```
trait Actor extends OutputChannel[Any] {
  // actor behavior
  def act():Unit

  // asynchronous send
  def !(msg:Any) {
  send(msg,Actor.self)
  }

  // synchronous send
  def !?(msg:Any):Any=…

  // send and get back a future for reply value
  def !!(msg:Any):Future[Any]=…
}
```

Figure 1. Actors. Actors in Scala interact with each other through messages, which can be sent asynchronously, synchronously, or asynchronously with futures, providing easy access to resulting return values.

Scala Actors — Message-Passing Concurrency Model

Actors represent a model of computation based on asynchronous message-passing concurrency that doesn't restrict the message-arrival ordering. An actor has its own control thread, encapsulates its state, and, unlike shared-state concurrency models, offers a shared-nothing architecture to its clients. Specifically, an actor localizes all states within itself, and the only way to change its state is via the exchange of immutable messages. With an actor model, developers can't write code to share state across actors, which implies they needn't work with mutex locks or other resource synchronization primitives required to manage shared-state consistency. When the basic architecture is shared-nothing, each actor appears to act in its own process space. The actor implementation's success and scalability depend a lot on the language's ability to implement lightweight processes on top of the underlying native threading model. Every actor has its own mailbox for storing messages, implemented as asynchronous, race-free, nonblocking queues.

Erlang, which implements a lightweight process model on top of operating system primitives,[2] inspired Scala's actor model. Scala implements this model as a library on top of the JVM and offers message-based concurrency using pattern-matching techniques that are more dependable than shared-memory concurrency with locks. Being an object-oriented-functional hybrid language, Scala actors integrate the benefits of Erlang's lightweight processes and shared-nothing message passing with strongly typed messages.

Scala's Salient Features

Message sends are usually asynchronous. When any given part of the code sends a message to an actor, the language runtime stores that message in the actor's mailbox, typically implemented as a queue. As the contract in Figure 1 shows, however, Scala also supports synchronous message sends that await a reply from the receiver.

`receive` typically waits in a loop, picking up the first message in the mailbox that matches any of the patterns specified in the match clause. In case none of the patterns match, the actor suspends.

The messages over which pattern matching takes place are usually implemented as immutable case classes, Scala's variant of algebraic data types. Scala implements a message `receive` in an actor using partial functions that are defined only for the set of messages to which the actor is supposed to respond:

```
def receive[R]
   (f:PartialFunction
   [Any,R]):R = …
```

This creates possibilities for implementing interesting message-receive

patterns using functional combinators that let developers compose receive operations as sequences or alternatives. For example, in the following snippet, the message-receive pattern first tries to run the genericHandler. If it suspends, then it runs the specialHandler as an alternative:

```
trait GenericServer extends
Actor {
  //..
  def act=loop {receive
     {genericHandler orElse
     specialHandler}}
  //..
}
```

Similarly, we also have the andThen combinator, which sequences receive handlers one after the other. This loop combinator is defined in terms of andThen:

```
def loop(body: => Unit): Unit
   = body andThen loop(body)
```

Scala actors are lightweight. The shared-memory threads that the JVM offers are heavyweight and incur significant penalties from context-switching overheads. To ensure lower process payload per instance, Scala's creators designed its actors as lightweight event objects. The Scala runtime schedules and executes the actors on an underlying worker thread pool that gets automatically resized when all threads block on long-running operations. In fact, Scala unifies two models of actor implementation: a thread-based model and an event-based model. In the thread-based model (implemented by receive), every actor is associated with a JVM thread that implements full stack-frame suspension when the actor blocks. However, unlike Erlang processes, JVM threads are expensive — hence, thread-based actors can't scale. The alternative implementation — based on events and implemented as react — liberates the running thread when the actor blocks on a message. Scala implements a wait on react as a continuation closure that captures the rest of the actor's computation:

```
def react
   (f: PartialFunction
   [Message, unit]):Nothing=…
```

react never returns (its return type is Nothing). Hence, stack-frame suspension involved in react-based message processing doesn't exist. When the suspended actor receives a message that matches one of the patterns the actor specifies, the Scala runtime executes the continuation by scheduling the task to one of the worker threads from the underlying thread pool.[3]

Lift: Functional Web Framework

Lift is a Web framework founded as an open source project by David Pollak, a software consultant in the San Francisco area. It's not part of Scala itself but rather is built on top of Scala's functional features. Lift's design and implementation extensively apply the advantages of immutable data structures, higher-order functions, abstract data types, and pattern matching. Scala's strict type system supplements this functional richness and adds to the framework's security and correctness. It also lets users write type-safe code that's resistant to attacks such as Structured Query Language (SQL) injection.

One of the main areas in Lift that shines with higher-order functions is HTML form processing. To ensure a complete separation of presentation logic from the code, Lift doesn't allow direct mapping of HTML tags to form fields. Instead, Lift implements controllers as snippets, which use closures to bind form elements to proper location and data. Consider the following form template — a pure XML file containing only Lift-specific and custom tags, without any code whatsoever:

```
<lift:Ledger.add form=POST>
  <entry:description />
  <entry.amount /><br />
  <entry:submit />
</lift:Ledger.add>
```

Lift processes the form elements and tags in the Lift-rendering pipeline via a combination of snippets that process the tag's XML contents. Figure 2 shows a sample snippet for the previous template.

Note that the code performs the form-element mappings in the call to the bind method, using higher-order functions that Lift automatically invokes when the user submits the form. This ensures a complete separation of the presentation and logic in HTML form processing, where Lift does all the plumbing of managing the controller pipelines during form rendering and submission.

Pattern matching over abstract data types, a common idiom in functional languages, eschews the complexity of the Visitor[4] design pattern. Pattern matching can make code more expressive — one example of its usage in Lift is the way it handles URL rewriting. The application code defines a mapping from RewriteRequest to RewriteResponse using pattern matching that the developer appends to the default rule set in LiftRules.rewrite, thus ensuring it automatically gets plugged into Lift's processing chain. Figure 3 provides an example of user-defined rewrite rules using Scala pattern matching. This rule rewrites URIs of the form /item/<itemname>, so that the showItem template can handle them.

Lift uses all the functional programming benefits that Scala offers to implement a functional Web framework. As the previous examples

illustrate, Lift APIs are also based around closures and pattern matching, which encourage client application code to be functional as well.

How Lift Uses Actors

Lift uses actors and immutability as part of its implementation framework and encourages Web-application development based on events and messages. One of the main areas in Lift that uses actors as a basic architectural unit is its Comet support. Ajax and Comet extend the traditional Web model with asynchronous interactions via partial updates of the document object model. Comet uses long-polling HTTP requests to let the server push data to the client without any additional requests. Comet is event driven and asynchronous — a perfect fit for actor modeling. The `CometActor` uses the Scala actor `react` event loop to send messages that abstract JavaScript commands for pushing data to the client:

```
trait CometActor extends Actor
   with BindHelpers { ..
  def act={
    loop {
      react(composeFunction)
    }
  }
}
```

Figure 4 shows a sample implementation of a `CometActor` from the Lift examples. The user just has to define the function that he or she wants rendered as the Comet request — Lift takes care of all the heavy lifting underneath, including managing timeouts.

For this `Clock` example,

- `ActorPing.schedule()` schedules a tick message every 10 seconds; and
- the method `lowPriority` is a `PartialFunction` that goes into the actor event-loop and does a partial update of specific fragments on the client side, without rerendering the entire content.

```
class Ledger {
  def add (xhtml:Group):NodeSeq = {
    var desc=..
    var amount=0
    def processEntryAdd () { ... }

    bind(entry,xhtml,
      description -> SHtml.text(desc, desc=_),
      amount -> SHtml.text(amount,amount=_),
      submit -> SHtml.submit(Add,processEntryAdd))
  }
}
```

Figure 2. Ledger. Lift supports mapping of form elements through higher-order functions, invoked automatically in its processing pipeline.

```
LiftRules.rewrite.append {
  case RewriteRequest(ParsePath(item::itemname::
     Nil,_,_,_),_,_) =>
    RewriteResponse(showItem::Nil,
      Map(itemname -> itemname))
}
```

Figure 3. Rules. Scala pattern matching used in Lift for URL rewriting.

```
class Clock extends CometActor {
  override def defaultPrefix=Full("clk")
  ActorPing.schedule(this,Tick,10 seconds)

  private lazy val spanId=uniqueId+"_timespan"

  def render=bind("time" -> timeSpan)

  def timeSpan=(<span id={spanId}>{timeNow}</span>)

  override def lowPriority={
    case Tick =>
      partialUpdate(SetHtml(spanId, Text(timeNow.
         toString)))
      ActorPing.schedule(this, Tick, 10 seconds)
  }
}
```

Figure 4. CometActor. Actor support in Scala provides for easy implementation of Comet-enabled Web applications.

Besides using actors as part of its implementation, Lift also encourages event-driven, actor-based

modeling at the application level. Developers can model user interactions as asynchronous messages delivered to Scala actors, which can then trigger domain logic or writes in the database.

In this column, we've provided only a small taste of the power and utility of both Scala and Lift — we encourage you to explore the references for further information. A future column will cover the Lift Web framework in much more detail. ◨

References

1. P. Haller and M. Odersky, "Event-Based Programming without Inversion of Control," LNCS 4228, Springer, pp. 4–22.
2. J. Armstrong, *Programming Erlang: Software for a Concurrent World*, Pragmatic Bookshelf, 2007.
3. P. Haller and M. Odersky, "Scala Actors: Unifying Thread-Based and Event-Based Programming," *Theoretical Computer Science*, vol. 410, nos. 2–3, 2009, pp. 202–220.
4. E. Gamma et al., "Design Patterns: Abstraction and Reuse of Object-Oriented Design," *Proc. 7th European Conf. Object-Oriented Programming*, Addison-Wesley, 1993, pp. 406–431.

Debasish Ghosh is the chief technology evangelist at Anshinsoft. He's a senior member of the ACM. You can read Ghosh's blog at http://debasishg.blogspot.com and contact him at dghosh@acm.org.

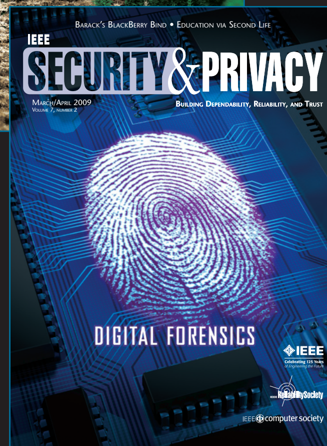
Steve Vinoski is a member of the technical staff at Verivue. He's a senior member of the IEEE and a member of the ACM. You can read Vinoski's blog at http://steve.vinoski.net/blog and contact him at vinoski@ieee.org.

Lower nonmember rate of $32 for *S&P* magazine!

IEEE Security & Privacy **is THE premier magazine for security professionals.**

Top security professionals in the field share information on which you can rely:

- Silver Bullet podcasts and interviews
- Intellectual Property Protection and Piracy
- Designing for Infrastructure Security
- Privacy Issues
- Legal Issues and Cybercrime
- Digital Rights Management
- The Security Profession



Visit our Web site at www.computer.org/security/

Subscribe now!

www.computer.org/services/nonmem/spbnr

WANT TO READ MORE?
computing now
http://computer.org/cn/elsewhere