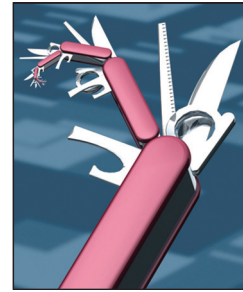


Scala Web Frameworks: Looking Beyond Lift



Dean Wampler • *Think Big Analytics*

Scala is a hybrid object-oriented and functional programming language for the Java Virtual Machine (JVM) that's growing in popularity. Two previous Functional Web columns presented the Lift framework, the best-known Web framework written in Scala.^{1,2} In terms of its prominence and full feature set, Lift is the Scala analog of the Ruby world's Ruby on Rails.

But other frameworks exist in the Scala world, just as alternatives exist to Rails in the Ruby world. One size doesn't fit all needs. A full list of Scala frameworks is available at <http://doi.ieeecomputersociety.org/10.1109/MIC.2011.104>. Some are full-stack frameworks for building multi-tier applications. Others are "point" tools for specific parts of an application, like template libraries for generating webpages (analogous to Java Server Pages). Still others focus on building particular kinds of networked servers, like REST response servers that are "headless."

Space considerations prevent us from discussing all these tools. It's hard to choose just a few representative examples, but here I focus on three: Play, a full-stack, commercially supported application framework; Scalatra, inspired by the lightweight, popular Sinatra framework; and Finagle, a highly scalable, headless server library.

Play

Play (www.playframework.org) is a Java-based Web framework with a very capable module architecture that makes it straightforward to write plug-in modules. Scala support is implemented as a module. It permits the use of Scala throughout the stack, including webpage templates and the database query layer.

A professional Web application developer accustomed to the polish and ease of use provided by Rails will feel at home with Play. Its creator,

Zenexity, has worked hard to create a developer-friendly experience.

Installing Play is easy. You download the zip file, expand it in a location of your choosing, and add the base directory to your environment's PATH variable, so the `play` command is on your path.

To install the Scala module, run this command:

```
play install scala
```

Now you can create a Scala Web application in a directory of your choosing:

```
play new SampleScalaApp --with scala
play run
```

The new application `SampleScalaApp` is now in a directory of the same name. Play's built-in Web server starts via the `run` command. By default, it listens for requests on port 9000. If you go to <http://localhost:9000> in your browser, you'll see the page shown in Figure 1, which provides instructions for what to do next.

The directory structure Play creates for an application will be familiar to Rails programmers. Because Play (and Rails) are designed to grow gracefully as applications become large, Play puts code for different application responsibilities in separate files so file sizes remain manageable.

The `SampleScalaApp/app` directory has a `view` subdirectory for views, which hold the webpage templates, a `models` subdirectory for domain classes, and a `controllers` subdirectory for the responders to user actions. However, because Scala code doesn't require the directory structure to match the package structure, you can put the files for your `controllers` and `models` in the `app` directory, if you prefer.

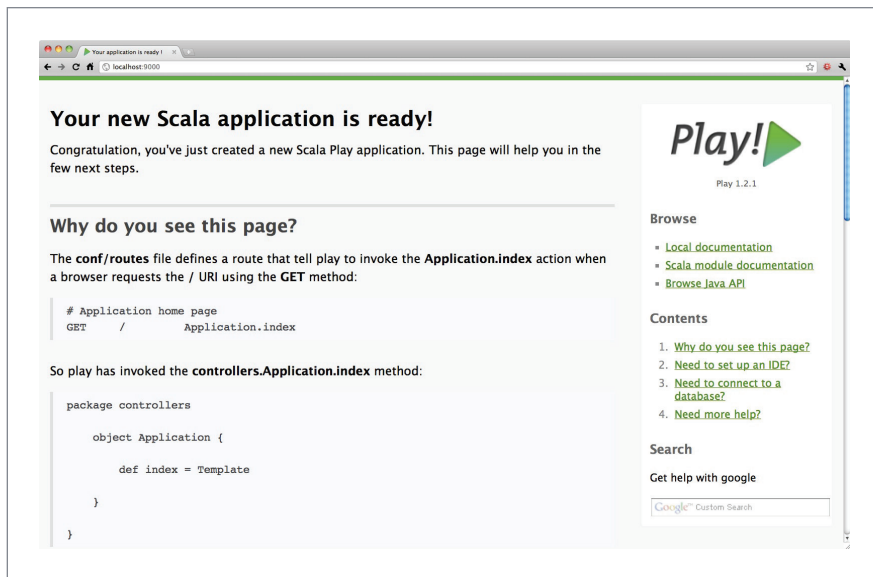


Figure 1. Your initial Play application webpage.

GET	/	Application.index
GET	/contacts	Application.list
POST	/contacts	Application.create
POST	/contacts/{id}	Application.save
GET	/contacts/{id}	Application.form
GET	/contacts/new	Application.form
POST	/contacts/{id}/delete	Application.delete
<pre># Map static resources in /app/public to the /public URL GET / staticDir:public</pre>		

Figure 2. Routing table for ZenContact in the `ZenContact/conf/routes` file. The table covers all the life-cycle steps required to view and manage a list of contacts.

The simple examples that come with the Scala module do just that.

Configuration of various properties, such as the database persistence settings, occurs in `SampleScalaApp/conf/application.conf`. Routing URL requests to the controllers that handle them is defined in `SampleScalaApp/conf/routes`.

Let's look at the ZenContact sample application that comes with the Scala module to see examples of what these various directories and files might contain. Figure 2 shows the routing table for ZenContact. It covers all the life-cycle steps required to view and manage a list of contacts.

First, the expression `{id}` defines a variable `id` that will be given whatever

value appears in this position in an incoming URL path. The `id` will be passed to the controller for use as a database lookup key, for example.

Using the routes from Figure 2, the URL `http://localhost:9000/contacts` will get routed to the `list` method in the `Application` singleton object, which is defined in `ZenContact/app/controllers.scala`, which looks like this (simplified slightly for brevity):

```
package controllers
/* imports ... */

object Application extends
  Controller {
  def index = {
```

```
    Template('now -> new Date)
  }

  def list = {
    new Template(
      "contacts" -> Contact.find(
        "order by name,
        firstname ASC").
        list()
    )
  }
  ...
}
```

The sidebar, “An Aside on Scala Syntax” offers a brief explanation of some Scala features used in this and subsequent examples.

The `list` method instantiates a new HTML page `Template` to format the response. The latter is passed key-value pairs, in which the keys are names of variables that will be referenced in the HTML template – in this case, a `contacts` variable. A `find` method on a singleton named `Contact`, which corresponds to a domain model object of the same name, is called to query the database for all the contacts, ordered by name. The query result is converted to a Scala list. (At the Java byte-code level, `Contact.find` will look exactly like a static `find` method defined in a traditional Java class named `Contact`.)

Here is the `Contact` domain model class defined in `ZenContact/app/models.scala` (again simplified for brevity):

```
package models
/* imports ... */

case class Contact(
  id: Pk[Long],
  @Required firstname:
    String,
  @Required name: String,
  @Required birthdate: Date,
  @Email email: Option[String]
)

object Contact extends
  Magic[Contact]
```

An Aside on Scala Syntax

For readers unfamiliar with Scala syntax, here are a few pointers:

- Compared to Java, Scala import statements use the “_” character instead of “*” as a wildcard.
- Semicolons are inferred.
- The `object` keyword declares a singleton object. The runtime will only instantiate one instance. Scala uses objects to hold methods and fields that would be declared `static` in Java classes.
- When the `case` keyword is used, it adds extra features to a class, including a corresponding singleton object (called a *companion*) with the same name (used for factories, pattern matching, and so on).
- The whole class body is the primary constructor, so the constructor argument list is passed after the class name.
- A method definition begins with `def`. Types for return values are usually inferred, and parentheses are usually omitted if there are no arguments. The method body begins after the “=” sign.
- Scala supports the syntax `key -> value` to pass key-value pairs to maps and methods that want them.
- Pattern matching is like switch statements on steroids. In pattern-matching expressions, each potential match begins with the `case` keyword, followed by a match expression and the body to execute if the match succeeds. The match expression and body are separated by “=>”.
- You subclass with the `extends` keyword. Using the `with` keyword, you can implement pure interfaces or mix in additional behaviors. Both pure Java-like interfaces and mix-ins are defined using a feature called *traits*.

You can handle integration with Play’s Java-based object-relational mapping (ORM) layer using annotations (such as the `@Required` annotation on some of `Contact`’s fields) and having the “companion” singleton `Contact` extend a `Magic` class that provides the `find` method, for example.

So, what are the benefits of using Scala? All the code you would write in Java becomes more concise in Scala, and you gain the additional benefit of Scala’s rich collections library. A great illustration of this is the new Anorm API in Play’s Scala module (<http://scala.playframework.org>). It isn’t a traditional ORM, but a wrapper for the lower-level Java Database Connectivity (JDBC) API. Anorm embraces a view I discuss elsewhere,³ that there are benefits to working directly with the collections that your database driver provides, as long as those collections offer useful methods for working with them. In contrast, the benefits of converting back and forth between those collections and domain objects don’t always outweigh the disadvantages of extra runtime complexity and overhead.

Anorm wraps JDBC with Scala collections semantics and more convenient handling of the checked

exceptions used in JDBC. Anorm also embraces the view that SQL itself is the best domain-specific language for talking to your database, so you should embrace it and not try to hide from it. Anorm makes it easy to convert back and forth between Scala collections and data from queries or data that’s used for updates. You can parse results with *pattern matching* and a built-in *parser combinator* library.

Here’s an example query adapted from the Anorm documentation:

```
val countries =
  SQL("Select name,population
      from Country")().collect {
    case Row("France", pop:Int)
      => ("France", pop)
    case Row(name:String, pop:Int)
      if(pop > 1000000) =>
      (name, pop)
  }
```

`Country` is a database table, and the block passed to `collect` uses pattern matching to select the rows we care about. In this case, we select France and all other countries where the population is greater than 1 million (note that Scala `case` matching is *eager*; that is, the first match “wins”). Each `case` “body” returns the tuple `(name, population)`. The `collect`

method will ignore any rows that don’t match one of the cases, effectively implementing a filter.

Play provides a rich, well-designed framework for building multi-tier Web applications that will feel familiar to the Ruby on Rails developer moving to Scala. The Scala module adds powerful APIs that exploit Scala’s functional programming features.

Scalatra

One popular alternative to Rails in the Ruby world is a lightweight framework called Sinatra. It’s ideal for quickly building lightweight Web applications with minimal code, where massive scalability and interoperability with extensive third-party services are less important. Compared to Rails, Sinatra is easier to use for websites without database persistence requirements, for example. Scalatra (<https://github.com/scalatra/scalatra>) started as a port of Sinatra to Scala, but has since added new capabilities of its own.

Recall that in Play, you normally define routing, controllers, models, and views in separate files. This *separation of concerns* makes sense for larger applications. In Scalatra, you can define everything in one file, which is very convenient for small,

```
/* package declaration and imports ... */

// UrlSupport and ScalateSupport are "traits";
// mixins of additional behaviors.
class TemplateExample extends ScalatraServlet
  with UrlSupport with ScalateSupport {

  // Scala supports embedded XML literals, which we
  // use to create this page template. They are mapped
  // to a Seq (sequence) of Node objects.
  object Template {

    // ""multi-line string"".
    def style() =
      """
      pre { border: 1px solid black; padding: 10px; }
      body { font-family: Helvetica, sans-serif; }
      h1 { color: #8b2323 }
      """

    // The expression { title } will be replaced
    // with the value for the title method argument,
    // using the Scalate template engine.
    def page(title:String, content:Seq[Node]) = {
      <html>
        <head>
          <title>{ title }</title>
          <style>{ Template.style }</style>
        </head>
        <body>
          <h1>{ title }</h1>
          { content }
          <hr/>
          <a href={url("/")}>hello world</a>
          <a href={url("/date/2009/12/26")}>date
            example</a>
          <a href={url("/form")}>form example</a>
        </body>
      </html>
    }
  }
}
```

Figure 3. Scalatra example, part 1. This segment defines a template singleton object, which is the template for building the HTML pages. It exploits Scala's ability to embed XML literals into code. The embedded HTML snippets are processed with the Scalate template engine.

simple applications. As the application size grows, you can separate responsibilities into different files.

Let's look at a simple one-file example of a Scalatra application, broken into several sections, which

I adapted from the examples that come with the distribution. (Actually, a `web.xml` file is also required to configure the Web server.) The first section, which is shown in Figure 3, defines an HTML template

that will be rendered with the Scalate template engine (<http://scalate.fusesource.org>). The second section, shown in Figure 4, defines how the application should respond to various requests.

Setting up a Scalatra project and running it in development mode isn't as straightforward as it is for Play. Some familiarity with Maven or the Scala build tool, `sbt` (<https://github.com/harrah/xsbt/wiki>) helps. The Scalatra `README.markdown` file that comes with the distribution describes the details.

Once you have the project set up and running with the example code in Figures 3 and 4, you will get the page Figure 5 shows when you go to `http://localhost:8080` (the default port). The "hello world" link at the bottom takes you to the same page.

Clicking the "date example" link produces Figure 6, which demonstrates the parsing and handling of URL path values.

Note how the route definition automatically decomposes the URL path `/date/2009/12/26` into year, month, and day values.

Finally, clicking the "form example" link yields Figure 7. (I entered the word "Hello!" into the text field before taking the screen shot.) Clicking the "Submit" button produces Figure 8.

The value in the form text field, `Hello!`, was passed as a parameter with the POST and used by the application to prepare the response shown to the user.

Although Scalatra requires very little code to create applications, it actually scales better than you might expect because it uses Jetty (<http://jetty.codehaus.org/jetty/>) as the underlying Web server.

Scalatra is a great tool for quickly building lightweight Web applications, especially if you're already familiar with Scala and Java tools, like `sbt` and Jetty. As with Play and its Scala module, Scalatra lets you use the power of Scala collections

and other functional features to minimize the code you write and maximize your ability to transform data as needed.

Finagle

Finally, let's consider Finagle (<https://twitter.github.com/finagle>), which was developed at Twitter for building very fast, RPC-style servers using Netty, a client-server socket API based on Java's New IO (NIO) library. Finagle is designed to meet Twitter's needs for extreme scalability.

Finagle is a good example of a very focused server development tool that doesn't attempt to provide a full Web stack. Instead, it focuses on serving a specific need – the development of fast, lightweight client-server networking applications, in which the ability to scale is paramount.

For clients, Finagle offers connection pooling, load balancing, failure detection, failover, retry, and other features important for distributed, reliable, and scalable client access to services. For servers, Finagle offers “backpressure” (a defense against denial-of-service attacks or other rogue clients), service registration, and support for protocols like HTTP, Comet, Thrift, and Memcached/Kestrel.

For the purposes of this column on the functional Web, Finagle demonstrates the elegance and power of compositional semantics that are common in functional languages such as Scala. Finagle uses an elegant composition mechanism for handling the parallel paths of normal and exceptional processing that any Web application must handle.

Consider the server example shown in Figure 9, which is adapted from an example in the distribution. It demonstrates an HTTP server that separates exception handling from normal control-flow processing and how they're composed together to build the service.

```
beforeAll {
  contentType = "text/html"
}

// Routing: HTTP GET request for URL
// http://server:port/ (i.e., empty path)
get("/") {
  Template.page("Scalatra: Hello World",
    <h2>Hello world!</h2>
    <p>Referer: { (request referer) map {
      Text(_) } getOrElse { <i>none</i> }}</p>
    <pre>Route: </pre>
  )
}

// Routing: HTTP GET request for a URL with
// the path "/date/YYYY/MM/DD", where Y, M,
// and D will be assigned to the year, month,
// and day parameters, respectively.
get("/date/:year/:month/:day") {
  Template.page("Scalatra: Date Example",
    <ul>
      <li>Year: {params("year")}</li>
      <li>Month: {params("month")}</li>
      <li>Day: {params("day")}</li>
    </ul>
    <pre>Route: /date/:year/:month/:day</pre>
  )
}

// Routing: HTTP GET request that will return
// a form with one text field.
get("/form") {
  Template.page("Scalatra: Form Post Example",
    <form action={url("/post")} method='POST'>
      Post something:
      <input name='submission' type='text' />
      <input type='submit' />
    </form>
    <pre>Route: /form</pre>
  )
}

// Routing: HTTP POST request, invoked when
// the form is submitted using POST.
post("/post") {
  Template.page("Scalatra: Form Post Result",
    <p>You posted: {params("submission")}</p>
    <pre>Route: /post</pre>
  )
}

protected def contextPath =
  request.getContextPath
}
```

Figure 4. Scalatra example, part 2. The second half of the file defines the content type used for the returned pages (“text/html”) and how the application should respond to various queries.

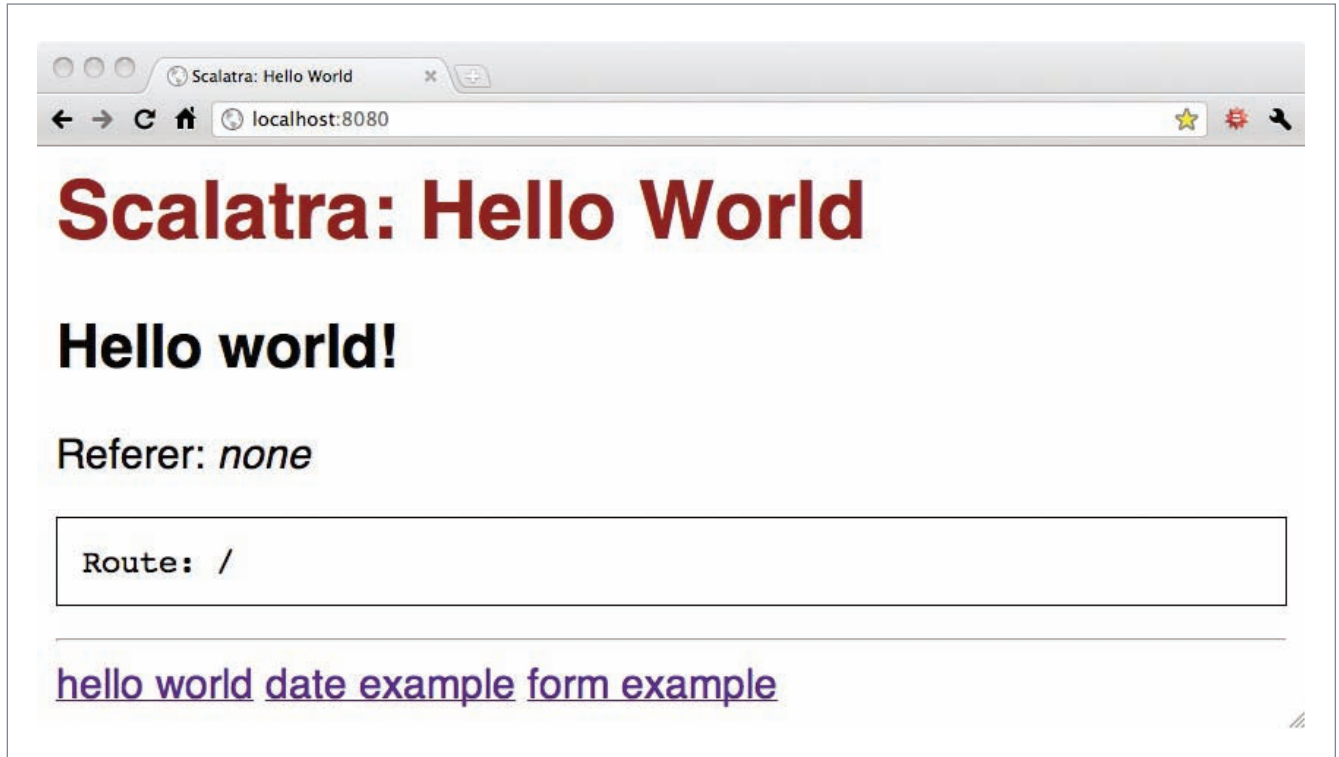


Figure 5. “Hello World” (and home) page for the Scalatra example.



Figure 6. Date example. This page demonstrates the parsing and handling of URL path values.

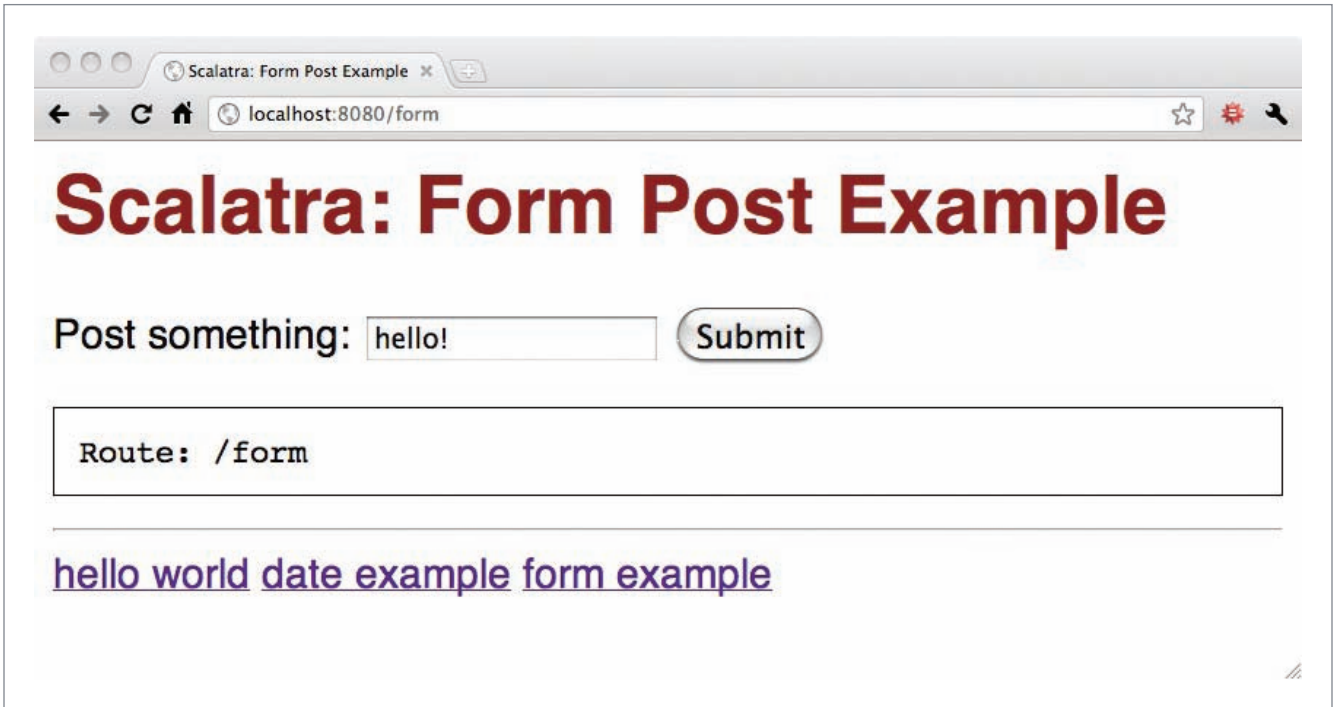


Figure 7. Form post example. This shows a conventional form for the user to fill in.



Figure 8. Form post result example. When the user submits the form in Figure 7, the content of the field is used to prepare the response shown here.

Note the composition of error and normal response handling in the definition of `myService`. The underlying `SimpleFilter` and `Service`

types that are subclassed by `HandleExceptions` and `Respond`, respectively, support a composition protocol that's common in Scala libraries –

that is, the `andThen` method, which composes invocation of the two apply methods in the objects so that `HttpServer` handles exceptions first,

```

/* package declaration and imports ... */
object HttpServer {
  /* A simple Filter that catches exceptions and
   * converts them to appropriate HTTP responses. */
  class HandleExceptions
    extends SimpleFilter[HttpRequest, HttpResponse]{
    def apply(
      request: HttpRequest,
      service: Service[HttpRequest, HttpResponse]) = {
      // "handle" is invoked asynchronously.
      // If an exception occurred, it sets the
      // corresponding error status code.
      service(request) handle { case error =>
        val statusCode = error match {
          case _: IllegalArgumentException => FORBIDDEN
          case _ => INTERNAL_SERVER_ERROR
        }
        val errorResponse =
          new DefaultHttpResponse(HTTP_1_1, statusCode)
        errorResponse.setContent(
          copiedBuffer(error.getStackTraceString, UTF_8))
        errorResponse // return value
      }
    }
  }

  /* The service itself. Simply echoes back "hello!".
   * Note that no error handling is required here! */
  class Respond extends Service[HttpRequest, HttpResponse]{
    def apply(request: HttpRequest) = {
      val response = new DefaultHttpResponse(HTTP_1_1, OK)
      response.setContent(copiedBuffer("hello!", UTF_8))
      Future.value(response) // asynchronous
    }
  }

  def main(args: Array[String]) {
    val handleExceptions = new HandleExceptions
    val respond = new Respond

    // Compose the error Filter and Service together:
    val myService: Service[HttpRequest, HttpResponse] =
      handleExceptions andThen respond

    val server: Server = ServerBuilder()
      .codec(Http())
      .bindTo(new InetSocketAddress(8080))
      .name("httpserver")
      .build(myService)
  }
}

```

Figure 9. A Finagle example. This segment demonstrates the separation of exception handling from normal control flow and how these handlers are composed together.


then normal processing. In either case, the `Respond` object returns a response asynchronously (using a `Future`) to the client. Note this model's power in separating concerns and building services that compose from smaller pieces.

Web application development might be approaching 20 years old, but we're still learning new tricks as we apply the elegance, concision, and power of functional programming ideas. The example Web and service frameworks I discussed here – Play, Scalatra, and Finagle – demonstrate these capabilities, while leveraging the best established features in traditional object-oriented frameworks. □

References

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.
2. D. Pollak and S. Vinoski, "A Chat Application in Lift," *IEEE Internet Computing*, vol. 14, no. 3, 2010, pp. 88–91.
3. D. Wampler, *Functional Programming for Java Programmers*, O'Reilly Media, 2011.

Dean Wampler is a principal consultant at Think Big Analytics (<http://thinkbiganalytics.com>). He specializes in Scala and "big data" analytics using the Hadoop ecosystem of tools. Wampler has a PhD in physics from the University of Washington. He's the coauthor of *Programming Scala* (2009) and the author of *Functional Programming for Java Developers* (2011), both published by O'Reilly Media. He's a member of IEEE and the ACM. Contact him at dean@deanwampler.com and follow him on Twitter, [@deanwampler](https://twitter.com/deanwampler).

 Selected CS articles and columns are also available for free at <http://ComputingNow.computer.org>.