

# The busy Java developer's guide to Scala: Scala and servlets

## Putting Scala to work with servlets

Skill Level: Introductory

[Ted Neward \(ted@tedneward.com\)](mailto:ted@tedneward.com)  
Principal  
ThoughtWorks; Neward & Associates

22 Dec 2008

In order for a language to be recognized as "real-world" and "ready for prime-time," the language has to be able to reach out to real-world environments and applications. In this article in the *[The busy Java developer's guide to Scala](#)* series, Ted Neward begins a tour of Scala in the real world by examining how Scala can interact with the core Servlet API and perhaps even improve it a little.

Scala is clearly an interesting language, well suited for showing off nifty new ideas in language theory and innovation, but at the end of the day, for it to be of any "real" use, it has to be able to meet practicing developers halfway and have some applicability in the "real world."

Now that we've looked at some of the core features of the language, can recognize some of Scala's linguistic flexibility, and have witnessed Scala in action creating DSLs, it's time to start reaching out to the environments that real applications use and show how Scala fits. We'll begin this new phase of the series by starting with the heart of most Java™ applications: the Servlet API.

## A review of servlets

Recall, if you will, from your Servlet 101 classes and tutorials, that the heart of the servlet environment is essentially a client-server exchange over a socket (typically port 80) using the HTTP protocol. The client can be any "User-Agent" (as defined by

the HTTP specification) and the server is a servlet container that finds, loads, and executes methods on a class I write that ultimately must implement the `javax.servlet.Servlet` interface.

### About this series

Ted Neward dives into the Scala programming language and takes you along with him. In this new developerWorks [series](#), you'll learn what all the recent hype is about and see some of Scala's linguistic capabilities in action. Scala code and Java code will be shown side by side wherever comparison is relevant, but (as you'll discover) many things in Scala have no direct correlation to anything you've found in Java -- and therein lies much of Scala's charm! After all, if Java could do it, why bother learning Scala?

Typically, the practicing Java developer doesn't write classes that implement the interface directly. Because the initial servlet specification expected to provide a generic API for other protocols beyond HTTP, the servlet namespace was split into two parts:

- A "generic" package (`javax.servlet`)
- An HTTP-specific one (`javax.servlet.http`)

As a result, some basic functionality was implemented in the generic package in an abstract base class called `javax.servlet.GenericServlet`; then further HTTP-specific functionality was implemented in the derived class `javax.servlet.http.HttpServlet`, which typically served as the base class for the actual "meat" of the servlet. `HttpServlet` provided a complete implementation of `Servlet`, delegating GET requests to an expected-to-be-overridden `doGet` method, POST requests to an expected-to-be-overridden `doPut` method, and so on.

## Hello, Scala. Hello, Servlets.

Naturally, the first servlet anyone writes is the ubiquitous "Hello, World" servlet; Scala's first servlet example should be no different. Recall from the introductory servlet tutorials of many years ago that the basic Java "Hello, World" servlet simply prints out the HTML response in Listing 1:

### Listing 1. The expected HTML response

```
<HTML>
  <HEAD><TITLE>Hello, Scala!</TITLE></HEAD>
  <BODY>Hello, Scala! This is a servlet.</BODY>
</HTML>
```

Writing a simple servlet to do this is almost absurdly easy in Scala and looks almost identical to its Java equivalent, as shown in Listing 2:

### Listing 2. Hello, Scala servlet!

```
import javax.servlet.http.{HttpServletRequest,
    HttpServletRequest => HSReq, HttpServletResponse => HSResp}

class HelloScalaServlet extends HttpServlet
{
  override def doGet(req : HSReq, resp : HSResp) =
    resp.getWriter().print("<HTML>" +
      "<HEAD><TITLE>Hello, Scala!</TITLE></HEAD>" +
      "<BODY>Hello, Scala! This is a servlet.</BODY>" +
      "</HTML>")
}
```

Notice that I use some well-placed import aliases to shorten up the type names of the request and response types; other than that, this looks almost identical to the Java servlet version. When compiling this remember to include a reference to the `servlet-api.jar` (which usually ships with the servlet container; in the Tomcat 6.0 release, it hides in the `lib` subdirectory) or the servlet API types won't be found.

This isn't quite ready for use yet; according to the servlet spec, it must be deployed into a Web application directory (or in a `.war` file) with a `web.xml` deployment descriptor that describes what URL this servlet should be married up against. For such a simple example, it's easiest to use a pretty simple URL to go along with it, shown in Listing 3:

### Listing 3. web.xml, the deployment descriptor

```
<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
  "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
  <servlet>
    <servlet-name>helloWorld</servlet-name>
    <servlet-class>HelloScalaServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>helloWorld</servlet-name>
    <url-pattern>/sayHello</url-pattern>
  </servlet-mapping>
</web-app>
```

From this point forward, I'm going to assume that readers will adapt/modify the deployment descriptor as necessary because it really has nothing to do with Scala.

Not surprisingly, well-formed HTML looks a great deal like well-formed XML; this is one place where Scala's XML literal support can make writing this servlet much, much easier. (See the article "Scala and XML" in [Resources](#).) Instead of embedding the message directly in a String passed to the `HttpServletResponse`, Scala can

achieve a (simplistic, perhaps overly so) separation of logic and presentation by making use of this support to put the message itself into an XML instance and passing that back:

#### Listing 4. Hello, Scala servlet!

```
import javax.servlet.http.{HttpServletRequest,
    HttpServletResponse => HSReq, HttpServletResponse => HSResp}

class HelloScalaServlet extends HttpServlet
{
  def message =
    <HTML>
      <HEAD><TITLE>Hello, Scala!</TITLE></HEAD>
      <BODY>Hello, Scala! This is a servlet.</BODY>
    </HTML>

  override def doGet(req : HSReq, resp : HSResp) =
    resp.getWriter().print(message)
}
```

Because of Scala's inline expression evaluation facilities where XML literals are concerned, this means that making the servlet a bit more interesting becomes much easier. For example, adding the current date to the message becomes as easy as adding a Calendar expression into the XML itself, something along the lines of `{ Text( java.util.Calendar.getInstance().getTime().toString() ) }`. Or if that feels a little too verbose, something like Listing 5:

#### Listing 5. Hello, timed Scala servlet!

```
import javax.servlet.http.{HttpServletRequest,
    HttpServletResponse => HSReq, HttpServletResponse => HSResp}

class HelloScalaServlet extends HttpServlet
{
  def message =
    <HTML>
      <HEAD><TITLE>Hello, Scala!</TITLE></HEAD>
      <BODY>Hello, Scala! It's now { currentDate }</BODY>
    </HTML>
  def currentDate = java.util.Calendar.getInstance().getTime()

  override def doGet(req : HSReq, resp : HSResp) =
    resp.getWriter().print(message)
}
```

In essence, the Scala compiler is stitching together the XML object message into a single `scala.xml.Node`, then turning that into a `String` when passed into the `print` method of the response's `Writer`.

Don't downplay this — in a very real way, this is a powerful separation of presentation from logic that takes place entirely inside of one class. The XML message will be compile-time checked to make sure that it is syntactically correct and well-formed, something we don't get with standard servlets (or JSP, for that

matter). Thanks to Scala's type-inferencing nature, the actual type information around `message` and `currentTime` can be left off, making this almost read like a dynamic language a la Groovy/Grails. For a first outing, it's not a bad result.

Of course, read-only servlets are pretty boring.

## Hello, Scala. These are parameters.

Most servlets don't just hand back simple messages like static content or the current date and time; they take `POST`d form content, examine the contents, and respond accordingly. For example, perhaps the Web application wishes to know the identity of the individual using it and asks for a first name and last name:

### Listing 6. Challenge!

```
<HTML>
  <HEAD><TITLE>Who are you?</TITLE></HEAD>
  <BODY>
    Who are you? Please answer:
    <FORM action="/scalaExamples/sayMyName" method="POST">
    Your first name: <INPUT type="text" name="firstName" />
    Your last name: <INPUT type="text" name="lastName" />
    <INPUT type="submit" />
    </FORM>
  </BODY>
</HTML>
```

OK, so it's not going to win any user-interface design contests, but it serves the purpose: it's an HTML form and it will be looking to send its data to a new Scala servlet (bound against the `sayMyName` relative URL). The data, as per the servlet specification, will be stored in a name-value-pair collection, obtained via the `HttpServletRequest.getParameter()` API call where we pass in the name of the `FORM` element as a parameter to the API call.

It would be relatively easy to do the direct translation from the Java code like in the servlet in Listing 7:

### Listing 7. Response! (v1)

```
class NamedHelloWorldServlet1 extends HttpServlet
{
  def message(firstName : String, lastName : String) =
    <HTML>
      <HEAD><TITLE>Hello, {firstName} {lastName}!</TITLE></HEAD>
      <BODY>Hello, {firstName} {lastName}! It is now {currentTime}.</BODY>
    </HTML>
  def currentTime =
    java.util.Calendar.getInstance().getTime()

  override def doPost(req : HSReq, resp : HSResp) =
    {
```

```

    val firstName = req.getParameter("firstName")
    val lastName = req.getParameter("lastName")

    resp.getWriter().print(message(firstName, lastName))
  }
}

```

But this starts to take away from some of the advantages of the message separation I cited before because now the definition of the message has to take the parameters `firstName` and `lastName` explicitly; this could get unwieldy if the number of elements used inside the response gets larger than three or four. Plus, the `doPost` method is going to have to itself extract each and every one of those parameters before handing them off to message for display — that kind of coding gets tedious and is potentially prone to error.

One way to approach this is to factor the extraction of the parameters and the invocation of the `doPost` method itself into a base class, such as in version 2 shown in Listing 8:

### Listing 8. Response! (v2)

```

abstract class BaseServlet extends HttpServlet
{
  import scala.collection.mutable.{Map => MMap}

  def message : scala.xml.Node;

  protected var param : Map[String, String] = Map.empty
  protected var header : Map[String, String] = Map.empty

  override def doPost(req : HSReq, resp : HSResp) =
  {
    // Extract parameters
    //
    val m = MMap[String, String]()
    val e = req.getParameterNames()
    while (e.hasMoreElements())
    {
      val name = e.nextElement().asInstanceOf[String]
      m += (name -> req.getParameter(name))
    }
    param = Map.empty ++ m

    // Repeat for headers (not shown)
    //

    resp.getWriter().print(message)
  }
}
class NamedHelloWorldServlet extends BaseServlet
{
  override def message =
  <HTML>
  <HEAD><TITLE>Hello, {param("firstName")} {param("lastName")}!</TITLE></HEAD>
  <BODY>Hello, {param("firstName")} {param("lastName")}! It is now {currentTime}.
  </BODY>
  </HTML>

  def currentTime = java.util.Calendar.getInstance().getTime()
}

```

This version keeps the actual display servlet relatively simple (compared to its predecessor) and has the added advantage that the `param` and `header` maps are immutable. (Note that we could have defined `param` to be a method that referenced the request object, but that request object would have had to have been defined as a field, which would have introduced a concurrency concern in a big way because the servlet container intrinsically considers each of the `do` methods to be re-entrant.)

Of course, error handling is generally an important part of the processing of a Web application FORM and the fact that Scala, as a functional language, holds that everything is an expression means that we can write `message` to be either the results page (assuming that we like the input) or an error page (if we don't). Thus a validation function that checks the non-empty status of `firstName` and `lastName` could look something like Listing 9:

### Listing 9. Response! (v3)

```
class NamedHelloWorldServlet extends BaseServlet
{
  override def message =
    if (validate(param))
      <HTML>
        <HEAD><TITLE>Hello, {param("firstName")} {param("lastName")}!
        </TITLE></HEAD>
        <BODY>Hello, {param("firstName")} {param("lastName")}!
        It is now {currentTime}.</BODY>
      </HTML>
    else
      <HTML>
        <HEAD><TITLE>Error!</TITLE></HEAD>
        <BODY>How can we be friends if you don't tell me your name?!?</BODY>
      </HTML>

  def validate(p : Map[String, String]) : Boolean =
    {
      p foreach {
        case ("firstName", "") => return false
        case ("lastName", "") => return false
        //case ("lastName", v) => if (v.contains("e")) return false
        case (_, _) => ()
      }
      true
    }

  def currentTime = java.util.Calendar.getInstance().getTime()
}
```

Notice how pattern matching (with its ability to bind against either raw values such as the previous case or to bind to a local variable such as a case where we want to exclude anybody with an "e" in their name like in the previous comment) makes it easy to write fairly straightforward validation rules.

There's obviously more that could be done here. One of the classic problems that plagues Java Web applications is the SQL injection attack, carried via unescaped

SQL command characters passed in via a `FORM` and concatenated against raw strings containing SQL structure before being executed against the database. Verifying that the `FORM` input is correct could be done using the regular expression support in the `scala.regex` package or even some of the parser combinator ideas discussed in the last three articles in this series. In fact, the whole validation process could be lifted into the base class using a default `validate` implementation that simply returns `true` by default. (Just because Scala is a functional language, let's not overlook good object design approaches.)

## Conclusion

While it's not nearly as full featured as some of the other Java Web frameworks out there, this tiny little Scala servlet framework I've created here serves two basic purposes:

- To demonstrate that Scala's features can be leveraged in some interesting ways to make programming for the JVM easier.
- To serve as a gentle introduction to the idea of using Scala for Web applications, which naturally leads to a reference to the "lift" framework, referenced in the [Resources](#) section.

That's it for this installment; until next time, enjoy!

## Downloads

Description	Name	Size	Download method
Sample Scala code for this article	j-scala12238.zip	179KB	<a href="#">HTTP</a>

[Information about download methods](#)

# Resources

## Learn

- "[The busy Java developer's guide to Scala: Functional programming for the object oriented](#)" (Ted Neward, developerWorks, January 2008): The first article in this series gives you an overview of Scala and explains its functional approach to concurrency among other things. Others in the series:
  - "[Class action](#)" (February 2008) details Scala's class syntax and semantics.
  - "[Don't get thrown for a loop!](#)" (March 2008) dives deep inside Scala's control structures.
  - "[Of traits and behaviors](#)" (April 2008) leverages Scala's version of Java interfaces.
  - "[Implementing inheritance](#)" (May 2008) is polymorphism done the Scala way.
  - "[Collection types](#)" (June 2008) is all "tuples, arrays, and lists," oh my!
  - "[Packages and access modifiers](#)" (July 2008) covers Scala's package and access modifier facilities and the `apply` mechanism.
  - "[Building a calculator, Part 1](#)" (August 2008) delivers the first part of the lesson from this article.
  - "[Building a calculator, Part 2](#)" (October 2008) delivers the second part of the lesson from this article, using parser combinators.
  - "[Building a calculator, Part 3](#)" (November 2008) delivers the third part of the lesson and shows you how to wire everything together into a seamless whole (and suggests some extensions that could be made to the language and interpreter).
- "[Scala and XML](#)" (developerWorks, April 2008) shows you how Scala lets you navigate and process parsed XML and details the first-class support for XML that is built into the language.
- This [wiki on the Scala lift framework](#), a framework for writing Web applications, demonstrates the framework's famous underpinnings (Seaside, Rails, Django, Wicket, etc.). Here's a [Getting Started tutorial](#).
- "[Functional programming in the Java language](#)" (Abhijit Belapurkar, developerWorks, July 2004): Explains the benefits and uses of functional programming from a Java developer's perspective.
- "[Scala by Example](#)" (Martin Odersky, December 2007): A short, code-driven introduction to Scala (in PDF).

- [Programming in Scala](#) (Martin Odersky, Lex Spoon, and Bill Venner; Artima, December 2007): The first book-length introduction to Scala.
- [Bjarne Stroustrup](#): Designed and implemented C++, which he has described as "a better C."
- [Java Puzzlers: Traps, Pitfalls, and Corner Cases](#) (Addison-Wesley Professional, July 2005) reveals oddities of the Java programming language through entertaining and thought-provoking programming puzzles.
- The [developerWorks Java technology zone](#): Hundreds of articles about every aspect of Java programming.

### Get products and technologies

- [Download Scala](#): Start learning it with this series.
- [SUnit](#): Part of the standard Scala distribution, in the *scala.testing* package.

### Discuss

- [developerWorks blogs](#): Get involved in the [developerWorks community](#).

## About the author

Ted Neward

Ted Neward is a consultant with ThoughtWorks, a worldwide consultancy, and the principal of Neward & Associates, where he consults, mentors, teaches, and presents on the Java, .NET, XML Services, and other platforms. He resides near Seattle, Washington.

## Trademarks

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.