Scalatra and Scalate using sbt in one Pomodoro

Gin-Ting Chen <chengt@chengin.com>

2010-07-30

Table of Contents

1. Concepts and Prereqs

Scalatra is a tiny, Sinatra-like web framework for Scala

Scalate is Scala based template engine that we're going to separate the view

A pomodoro is a time management technique only mentioned for fun and as a time reference, not something that you need to use.

I also assume that you have read about and setup sbt

2. Project creation

We're going to follow the sbt Quick Setup instructions which is as easy as creating a project folder (we're going to use HelloWorld for the folder), running the sbt command and following the prompts. When it asks you what version of scala to use enter 2.8.0. The input and output would look something like below

```
- ~/Projects/HelloWorld: sbt
Project does not exist, create new project? (y/N/s) y
Name: Hello World
Organization: chengin
Version [1.0]:
Scala version [2.7.7]: 2.8.0
sbt version [0.7.4]:
Getting Scala 2.7.7 ...
# I removed some content here
[info] Building project Hello World 1.0 against Scala 2.8.0
[info]    using sbt.DefaultProject with sbt 0.7.4 and Scala 2.7.7
>
```

Notice that it drops you to a prompt. That's the sbt shell prompt so exit out of that (by typing exit or just ctrl-c out). You will also notice that even though you selected to use scala 2.8.0 it still downloads and used 2.7.7. That's what sbt uses internally and should not effect anything.

If you want to ensure that everything is ok you can run "update" while in that sbt prompt and you should see a [success] message.

3. Adding the dependencies and making a web project

I'm going to assume that you followed the prereqs and read the quick intro to sbt so I won't go into it's mechanisms too much. If you get lost at any point just read the sbt quick setup.

What I need to do now is to tell sbt that HelloWorld is a web project (which allows me to run jetty through the sbt command shell) and also to add my dependencies for scalatra and scalate.

So first I'm going to create the project descriptor by first creating a build folder under ${PROJECT_HOME}/project and adding a file which I'll, for obvious reasons, call HelloWorld.scala.

```
- ~/Projects/HelloWorld: mkdir project/build
- ~/Projects/HelloWorld: vim project/build/HelloWorld.scala
```

The contents of that file will be

```
import sbt._

class HelloWorld(info: ProjectInfo) extends DefaultWebProject(info)
{
    val jettyVersion = "6.1.22"
    val servletVersion = "2.5"
    val slf4jVersion = "1.6.0"
    val scalatraVersion = "2.0.0-SNAPSHOT"
    val scalateVersion = "1.2"
    val scalaTestVersion = "1.2-for-scala-2.8.0.final-SNAPSHOT"

    val jetty6 = "org.mortbay.jetty" % "jetty" % jettyVersion % "test"
    val servletApi = "javax.servlet" % "servlet-api" % servletVersion % "provided"

    // scalaTest
    val scalaTest = "org.scalatest" % "scalatest" % scalaTestVersion % "test"

    // scalatra
    val scalatra = "org.scalatra" %% "scalatra" % scalatraVersion

    // scalate
    val scalate = "org.fusesource.scalate" % "scalate-core" % scalateVersion
    val scalatraScalate = "org.scalatra" %% "scalatra-scalate" % scalatraVersion

    val sfl4japi = "org.slf4j" % "slf4j-api" % slf4jVersion % "runtime"
    val sfl4jnop = "org.slf4j" % "slf4j-nop" % slf4jVersion % "runtime"

    // repositories
    val scalaToolsSnapshots = "Scala Tools Repository" at "http://nexus.scala-tools.org/content/repositor
    val sonatypeNexusSnapshots = "Sonatype Nexus Snapshots" at "https://oss.sonatype.org/content/reposito
    val sonatypeNexusReleases = "Sonatype Nexus Releases" at "https://oss.sonatype.org/content/repositori
    val fuseSourceSnapshots = "FuseSource Snapshot Repository" at "http://repo.fusesource.com/nexus/conte
}
```

Again, verify that everything is correct by running "sbt update". At this point we can also test start jetty (even though nothing is going to be visible by running jetty-run. If you do this you should see a [success] msg and be able to see a default landing page when you navigate to http://localhost:8080. Stop jetty by typing jetty-stop

Even though I commented the build file relatively well, I'll explain some piece of it as some of this has interesting facts.

The first section of my build file just contains the versions so that I can quickly upgrade/downgrade should newer versions of my dependencies come out.

The 2nd section is solely so that I can run jetty using sbt.

The 3rd section I'm not using right now because this example is a horrible example of good TDD practices :P But it's there should you decide (and you should) to add tests.

The scalatra section is interesting. This did not work prior to scalate releasing version 1.2. I used to have to git clone scalatra and build it manually and drop my jars for scalatra and scalatra-scalate into a lib folder in my ${PROJECT_HOME} directory. But since version 1.2 of scalate was released, I have not had to do that. But of course, since you can see that I still rely on SNAPSHOT code, all that could change at anytime. :P

Scalate requires sfl4j so that is why I added the sfl4j dependencies into the scalate section but I included them only as runtime dependencies.

4. Creating a simple Scalatra servlet

Create a new file called ${PROJECT_HOME}/src/main/scala/main/com/chengin/web/HelloWorld.scala. I'm using the package com.chengin.web just for show but, if you change it, remember to replace that for whatever you name it to when it's referenced later. The contents of that file will look like

```
package com.chengin.web

import org.scalatra._

class HelloWorld extends ScalatraServlet {

    before {
        contentType = "text/html"
    }

    get("/") {
        "Hello World"
    }

    protected def contextPath = request.getContextPath
}
```

If you tried to skip ahead and ran sbt update and sbt jetty-run you will notice that you are still at the default landing page. That's because we have to first tell our web container (jetty in this case) about our servlet and also what URLs it can handle. The way that we do this is by creating a web.xml in src/main/webapp/WEB-INF and adding a brief descriptor for our new servlet. It will look something like

```
<!-- this will be under src/main/webapp/WEB-INF/web.xml -->
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_
         version="2.5">
    <servlet>
        <servlet-name>HelloWorld</servlet-name>
        <servlet-class>com.chengin.web.HelloWorld</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>HelloWorld</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>

    <servlet-mapping>
        <servlet-name>default</servlet-name>
        <url-pattern>/images/*</url-pattern>
        <url-pattern>/css/*</url-pattern>
        <url-pattern>/js/*</url-pattern>
    </servlet-mapping>
</web-app>
```

As you can see, we first create a servlet section to help locate our servlet. The servlet-class will map to the fully qualified classname of our new servlet and the servlet-name is just an arbitrary name that we give it. Next we create the servlet-mapping section that tells our web container what url patterns our servlet (make sure the name here matches the one that you just gave in your servlet section) will be able to handle. The next servlet section will work for sure in tomcat and jetty but may not work in other servlet containers and basically maps the url patterns that you see to the 'default' servlet. The reason this is necessary is that with the all encompassing /* pattern that we gave above, all requests to resources that we bundle with our webapp will incorrectly be handled by our servlet. Other ways that you can get around this issue is to use a Filter instead and programmatically map those resources out or by making a special pattern (for example /servlet or /*.do as common in other webframeworks) for our servlet. There are other ways around this as well but I'm running on a pomodoro here (actually my second :)).

Now if we run sbt update jetty-run we should see our Hello World msg when browsing to http://localhost:8080. Success! But we're not done yet.

5. Adding Scalate

Earlier we already added all our scalate dependencies so we don't have to worry about touching the build descriptor (remember that file? It was the one under ${PROJECT_HOME}/project/build) again. Also, I should mention that you didn't actually have to run update under sbt since we didn't update any of the dependencies after our initial run of update. But you knew that already since you read the sbt intro doc right? ;)

We're going to create an index.scaml file under ${PROJECT-HOME}/src/main/webapp/WEB-INF. Those of you that know anything about Java Servlets would probably be scratching their heads as this is not a publically accessible location. But the fact is that we don't need to make it publically accessible since we are not going to serve it directly and will instead have the scalate template engine render it for us (more on this in a bit). So go ahead and create the file and make a very simple

```
-@ val content:String
!!! 5
%html
  %head
    %title Hello World
  %body
    %h1= content
```

I'm using scaml syntax but won't get into that. It should be pretty self explanatory what will happen but you can also read up on the scaml syntax on scalate site if you need.

Next, I'm going to tell my servlet to use that as my index page instead of the hardcoded Hello World. I added some comments to explain what was going on.

```
package com.chengin.web

import org.scalatra._
import scalate.ScalateSupport // ScalaSupport is a trait that adds scala support

class HelloWorld extends ScalatraServlet with ScalateSupport { // adds the scala trait to your servlet
    before {
        contentType = "text/html"
    }

    get("/") {
        // templateEngine is defined for you by ScalateSupport and can be used to help render your index.
        templateEngine.layout("/WEB-INF/index.scaml", Map("content" -> "Hello World"))
    }

    protected def contextPath = request.getContextPath
}
```

I'm using the layout method of templateEngine rather than the renderContent documented on scalatra site because that does not allow for layouts (which is kind of like Tiles or SiteMesh). The map adds an attribute key called content with the value of Hello World that will be rendered in our scaml.

Run jetty-run in sbt again and browse to http://localhost:8080 again. It will take a while (compilation of all things scala is painfully slow :P) but you will see your Hello World msg in a h1 header tag. Now one last thing (again because it is currently documented incorrectly).

6. Adding a layout

Let's strip out our index.scaml to simply say

```
- attributes("title") = "Hello from layout"
-@ val content: String
%h1= content
```

Next let's create a new file called default.scaml and place it under ${PROJECT_HOME}/src/webapp/WEB-INF/scalate/layouts with the contents

```
-@ var title: String = "Default Title"
-@ val body: String
!!! 5
%html
  %head
    %title= title
  %body
    %p
      != body
```

Now run jetty again and browse to http://localhost:8080. Notice what happened? You now get the default layout as a wrapper for your content and by adding an attribute for title in your index.scaml, you have replaced the default title of 'Default Title' with your own more descriptive page comment.

That's it! We've now create a simple web app using a very light weight framework of scalatra, scalate and sbt. And, hopefully, all it took you was 1 pomodoro. :)