# Java & Struts2 & Spring & Hibernate & Eclipse Tutorial
# Building a web app from scratch

Update! Code has been moved to:
http://github.com/chrishulbert/JavaTutorial

Apr 2010
Document version 4
Project files: http://splinter.com.au/blog/?p=194
This document: http://www.scribd.com/doc/25244173/Java-Struts-Hibernate-Tutorial
Chris Hulbert (chris.hulbert@gmail.com)
Feel free to get in touch with me with any questions! I really hope you get something out of this.

## Table of contents

# Introduction

Hi, this tutorial was written from the perspective of an Asp.Net / C# developer trying to understand the basics of creating a typical web application using Java. In the Asp.Net world, everything is different as you are provided pretty much all you need, whereas with Java you get to choose and plug a whole bunch of components together. In this case, I've chosen Struts2 and Hibernate (and Spring and Sitemesh and Jtds and Log4j), and Eclipse as the IDE.

I'll start with the assumption that you're using a Windows PC, and that's about it. My database stuff does revolve around SQL server, but Hibernate is flexible so with a little configuration you can apply this to any other database as you want. There is an appendix for making this work with MySql too.

Also, please be flexible when you see small areas in the screenshots that don't match your screen. This is due to me taking the screenshots out of order whilst making this tutorial, eg you may see a folder in the project explorer that I haven't told you to make yet. Just ignore these if you notice them.

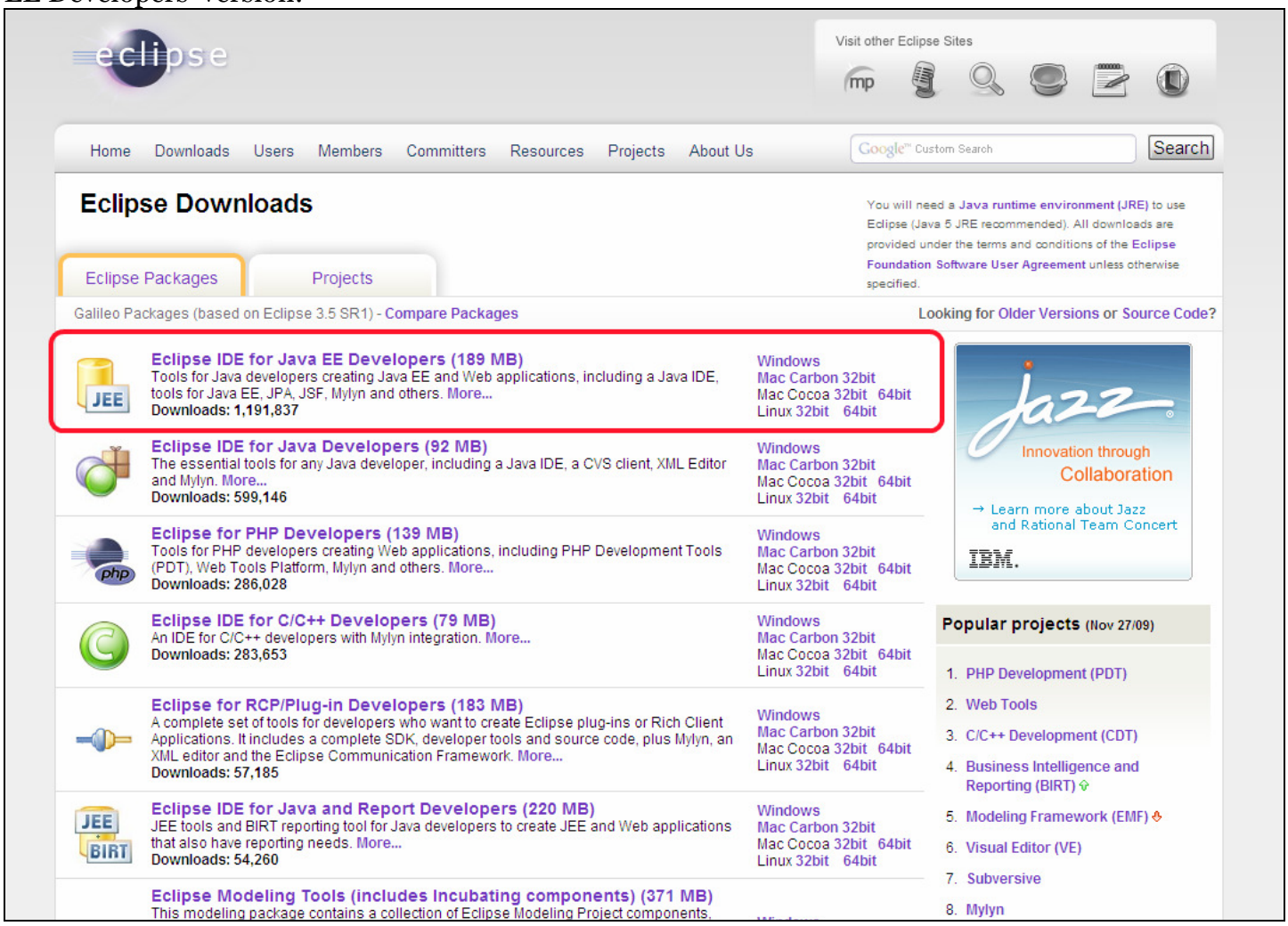
So lets get started!

# Installing Java

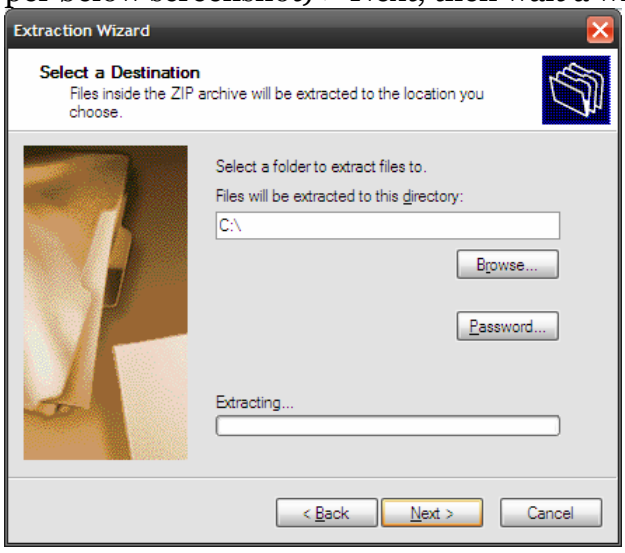
First up you'll need to install the latest version of Java. The simplest way to do this is to download it from http://java.com/
I downloaded and installed the version 'JavaSetup6u17-rv.exe'

# Installing Eclipse

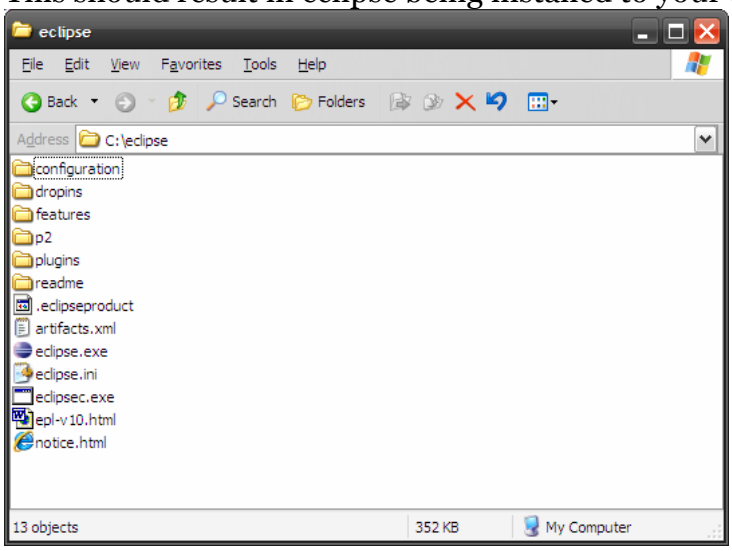
Now lets install Eclipse. Go to their website: http://eclipse.org/downloads/
There are a whole bunch of options to choose from. You want to download the 'Eclipse IDE for Java EE Developers' version:



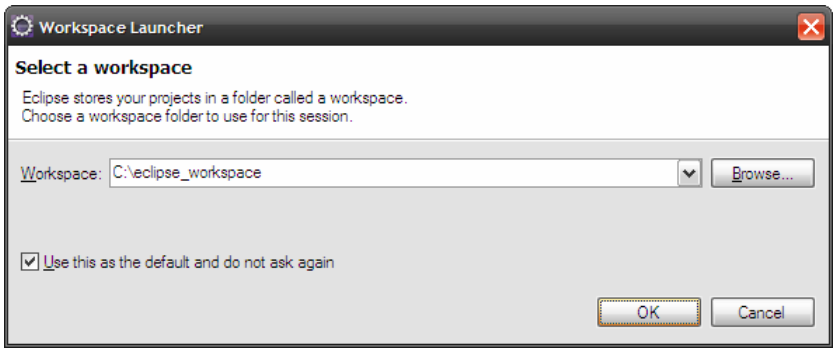

The version I downloaded was 'eclipse-jee-galileo-SR1-win32.zip', however a newer version may be available by the time you read this. Right click the file > Extract All > Next > "C:\' for the directory (as per below screenshot) > Next, then wait a while as it does its thing.



This should result in eclipse being installed to your c:\eclipse folder, like so:



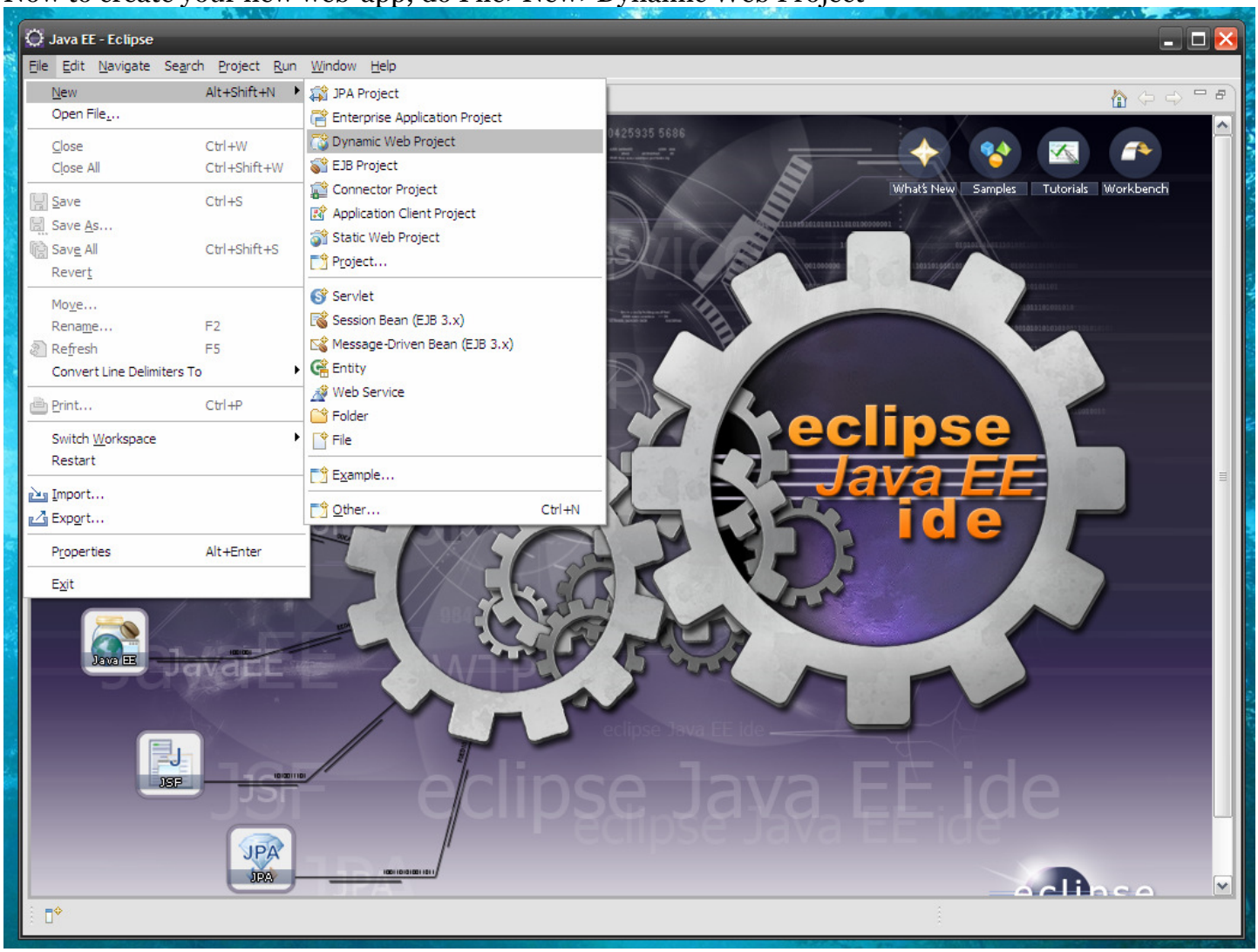

Then run eclipse.exe to start it up. It'll ask for a folder to use as the workspace the first time you run it. I created a folder C:\eclipse_workspace and used that. Also, select the 'Use this as the default' checkbox:



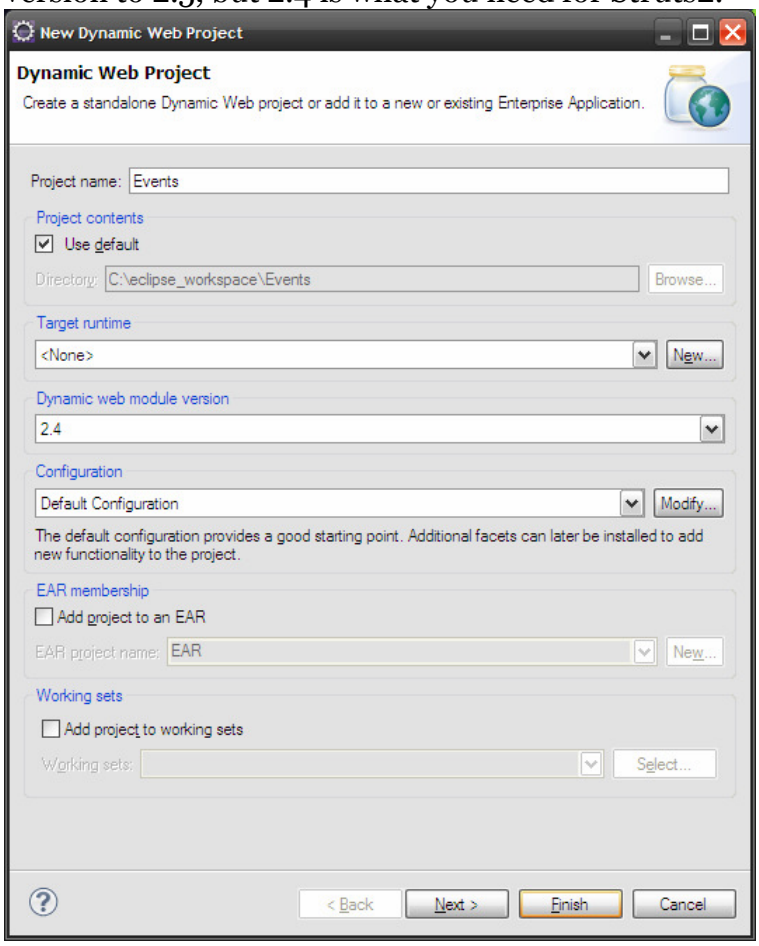
Great, you should now be in Eclipse:

# Creating the project

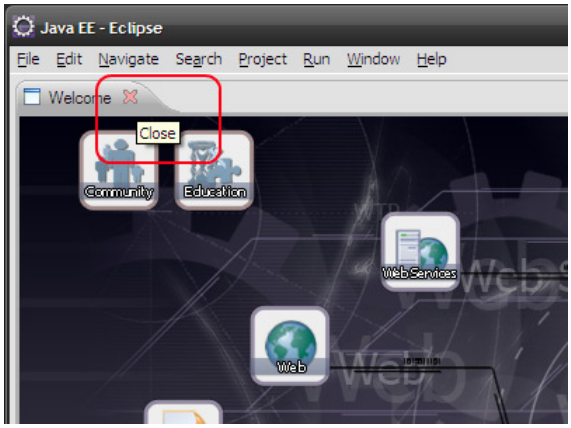
Now to create your new web-app, do File>New>Dynamic Web Project



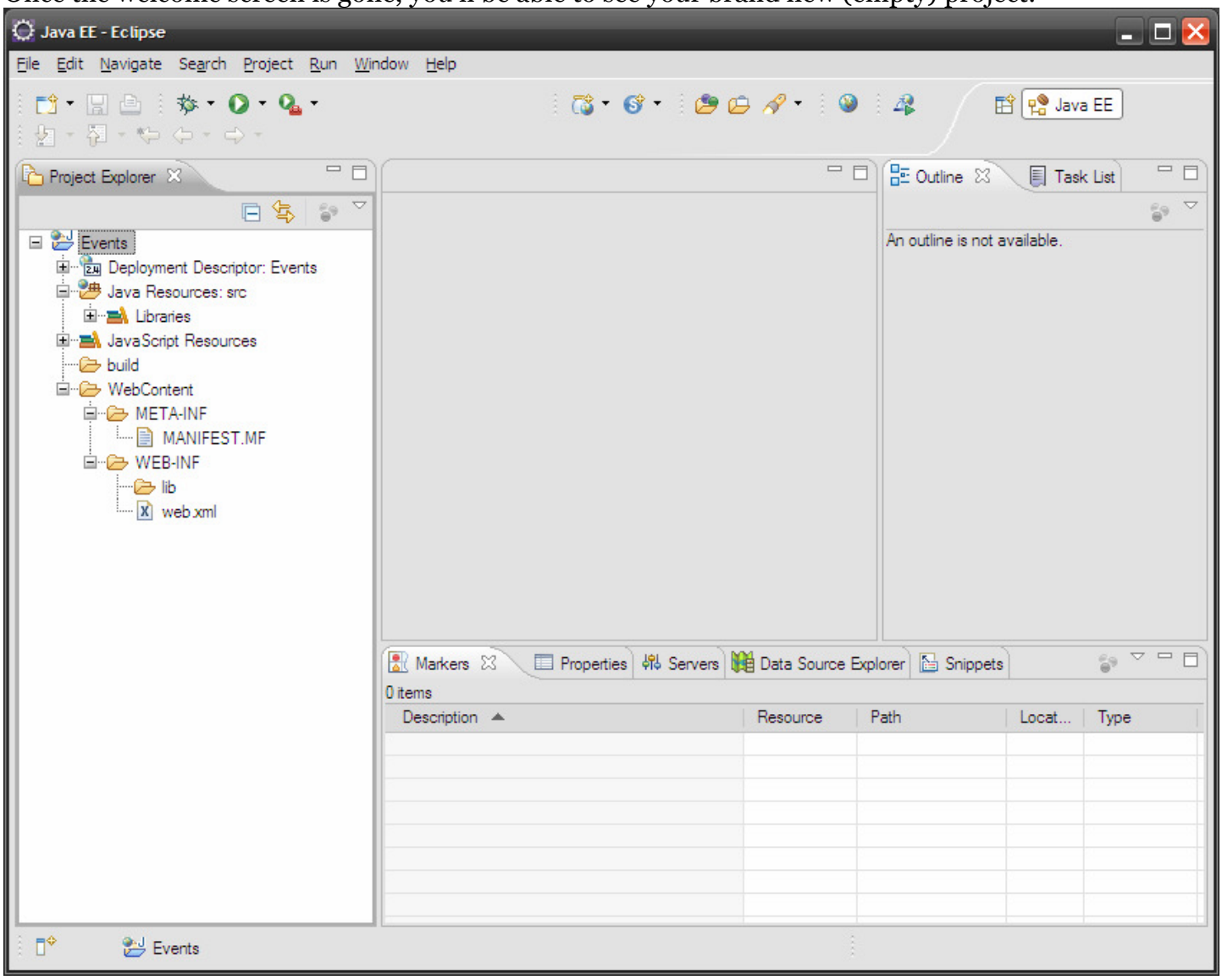
The following dialog will appear as below. Set the project name to 'Events' and leave the rest as-is, and click Finish. I originally had issues with my project because I set the dynamic web module version to 2.5, but 2.4 is what you need for Struts2.



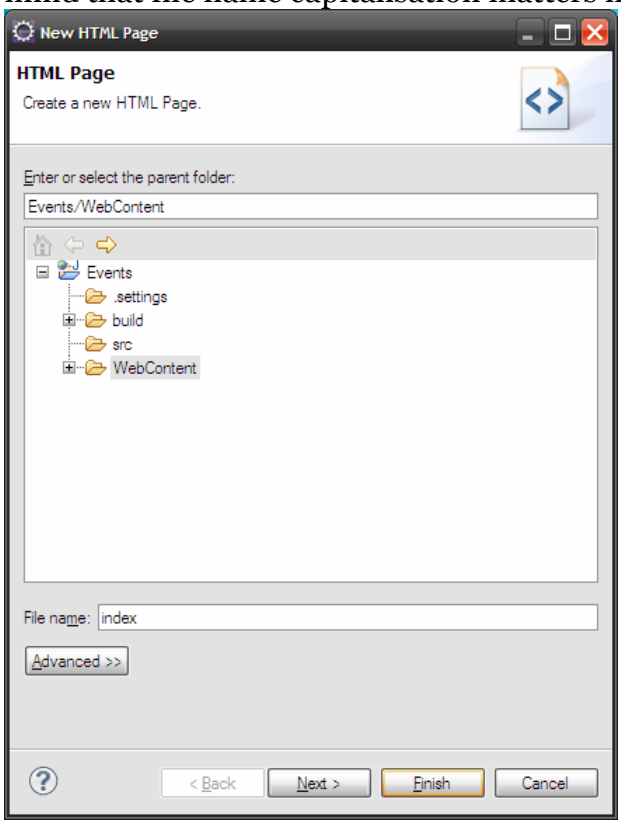

You'll then want to close the 'Welcome' screen:



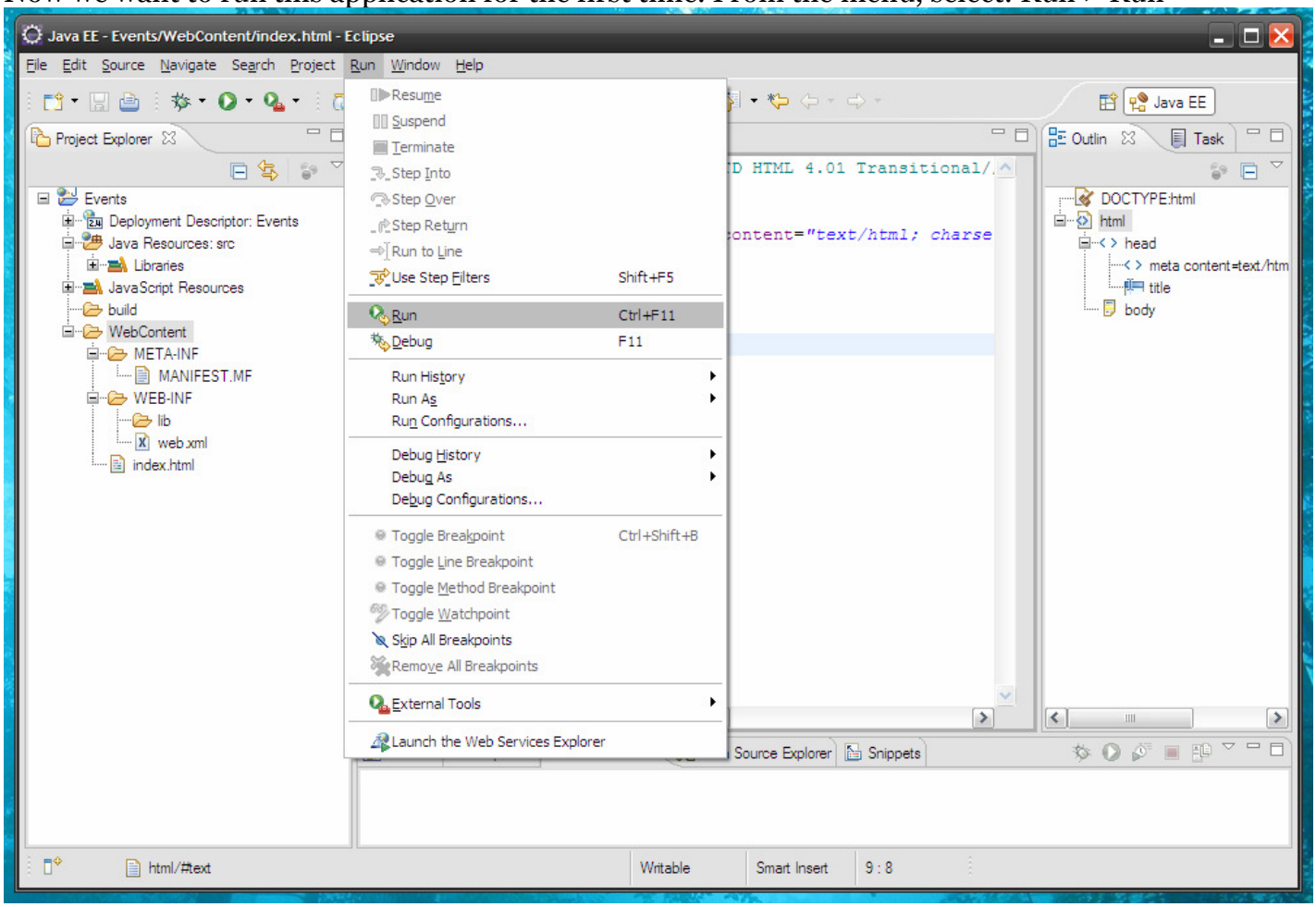
Once the welcome screen is gone, you'll be able to see your brand new (empty) project:



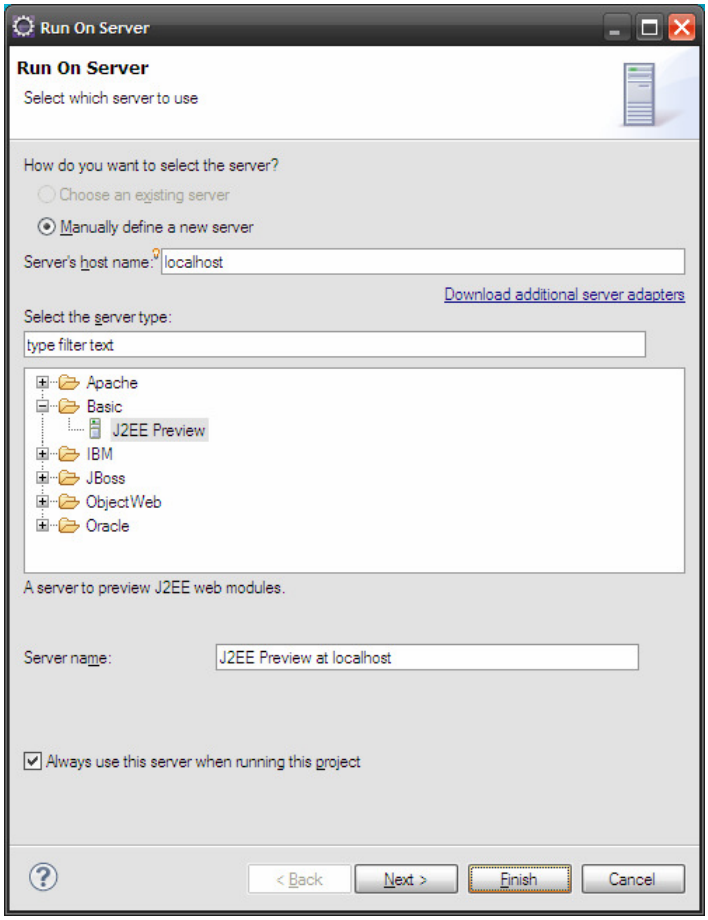
Lets start off by simply creating a static index page. In your Project Explorer window, right click on WebContent > New > Html Page. For the file name, simply type ‘index’ and click Finish. Keep in mind that file name capitalisation matters in the Java world, so be careful through this tutorial.




Now we want to run this application for the first time. From the menu, select: Run > Run

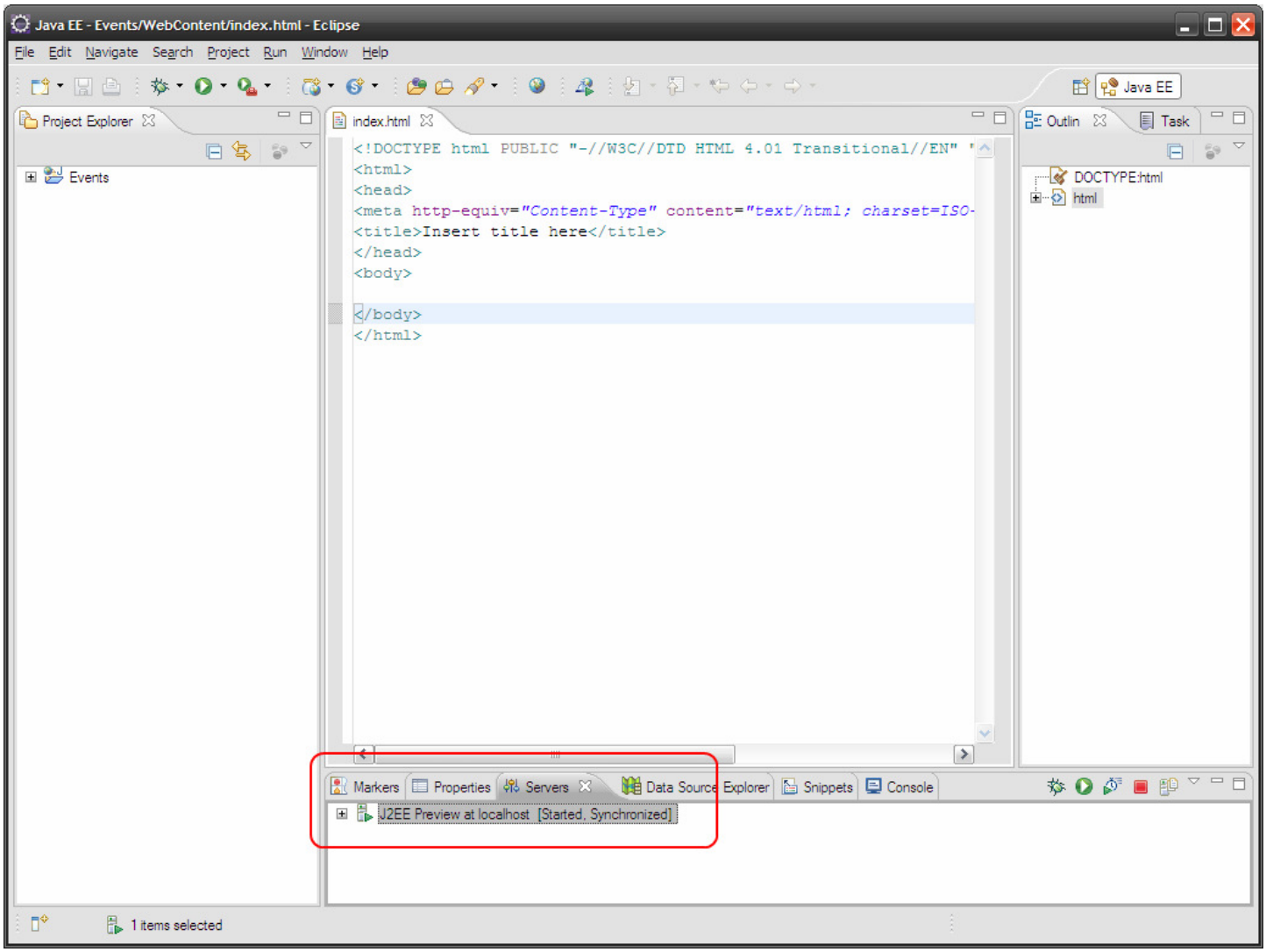


A servlet container (or simply 'server') is a java server such as Tomcat that runs your application for you (analogous to IIS in the .Net world). For development purposes, we need one of these created in Eclipse so we can see our application as it takes shape.
It will ask for details to create a servlet container. Choose Basic > J2EE Preview, as well as 'Always use this server', then click Finish. Later on in this document we'll switch to Tomcat but this will do for now.

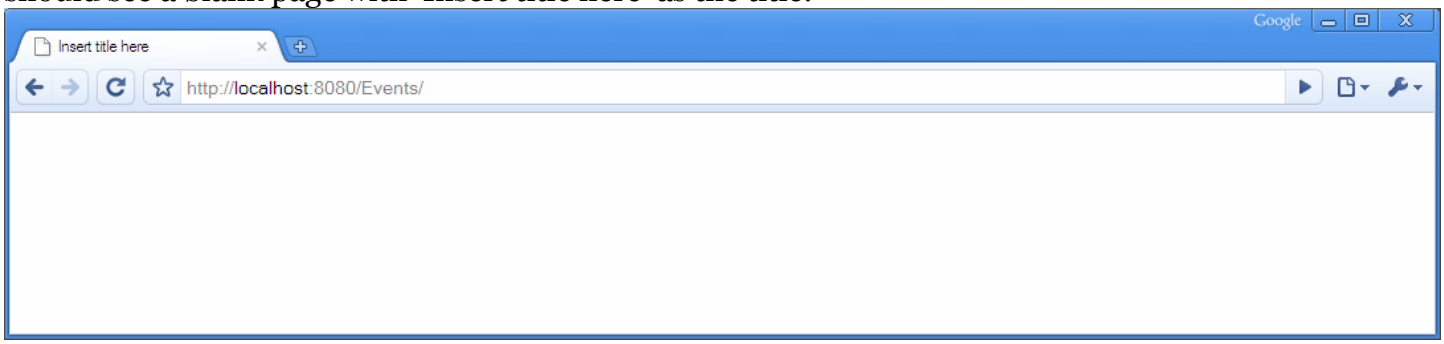



Depending on your system's Java configuration, you may see some more settings above for 'server runtime environment', you can leave these settings as the defaults.

It should play around for a while, and eventually settle down after getting started. Eclipse should now show the server as started in the servers tab at the bottom of the screen:

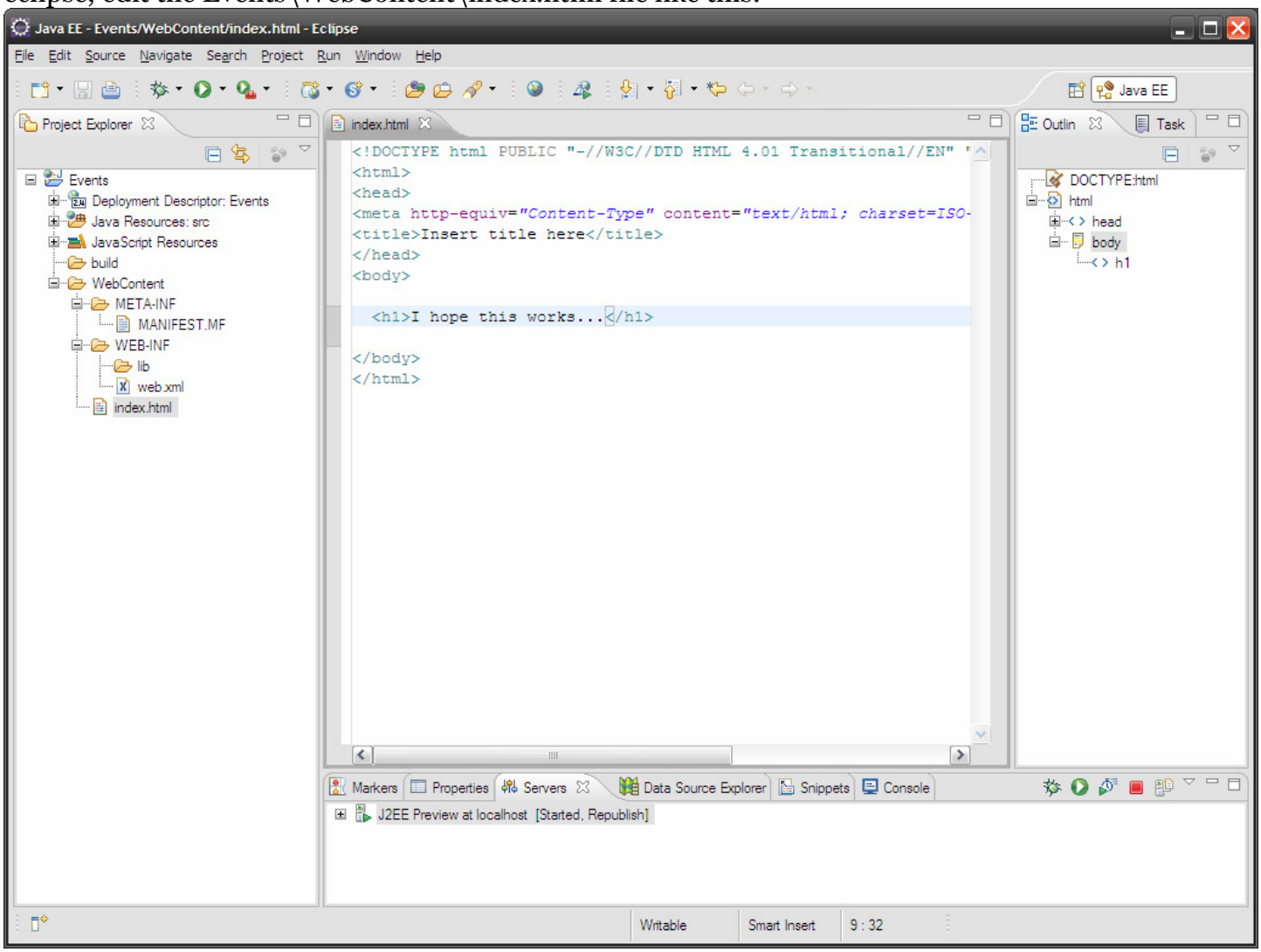



When it's done that, you can fire up your browser and go to http://localhost:8080/Events. You should see a blank page with 'Insert title here' as the title:

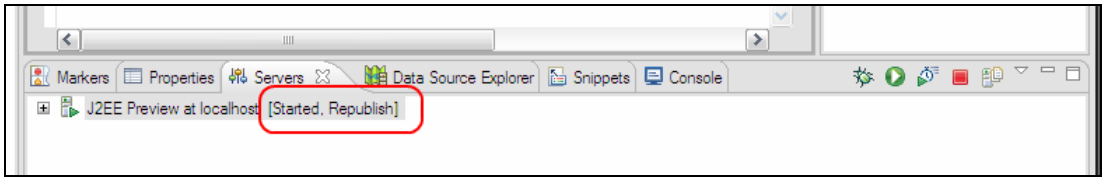



If you've gotten this far, great, your development environment is all set up and it's working.

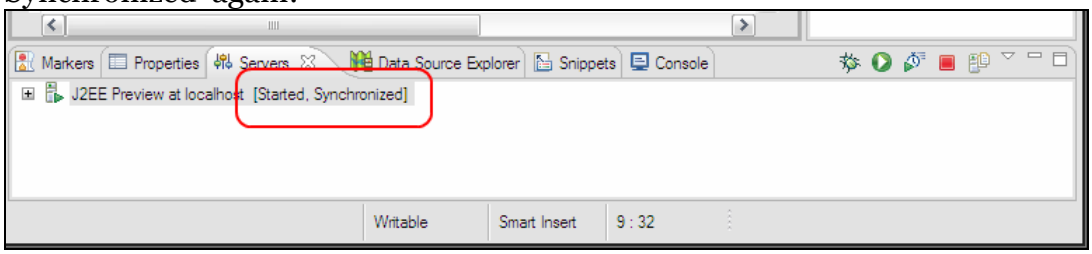
Now let's go back into Eclipse and put in the traditional 'hello world' (or whatever you feel like). In eclipse, edit the Events\WebContent\index.html file like this:



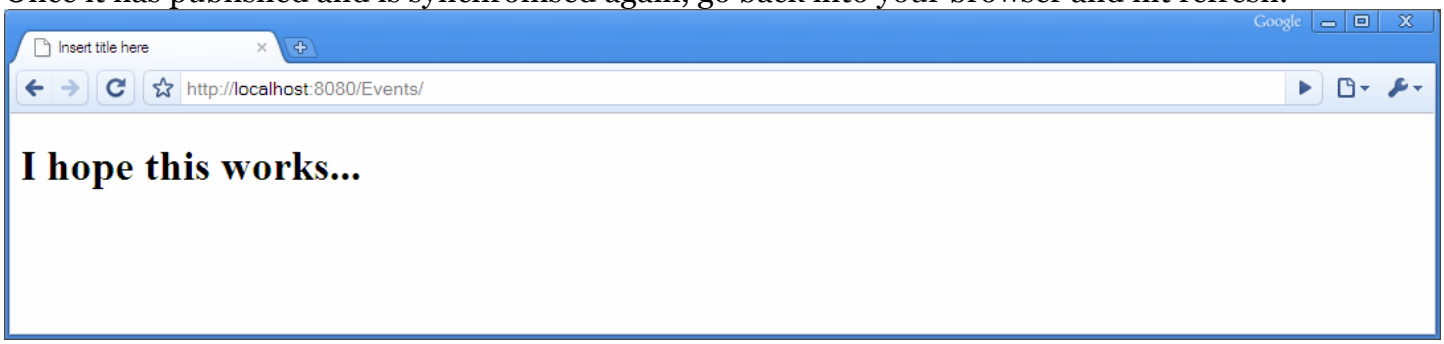

After you edit and save the file, you'll notice in the 'servers' view that it will say 'Republish' for a while:




This is because when you make edits to your application, it needs to publish those edits to the servlet container before you can see them in your browser. After a while it should settle back to 'Started, Synchronized' again:



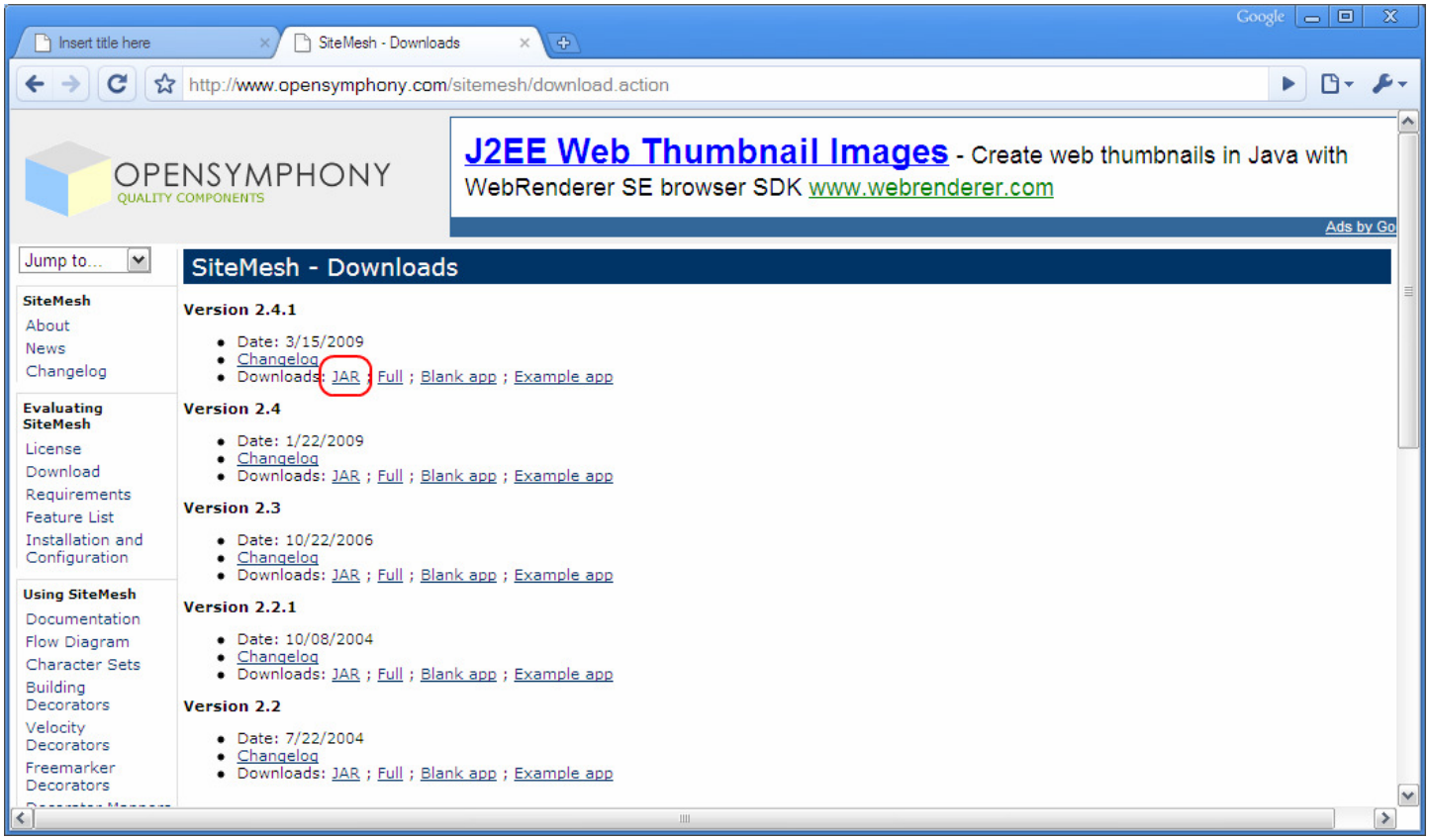
Once it has published and is synchronised again, go back into your browser and hit refresh:



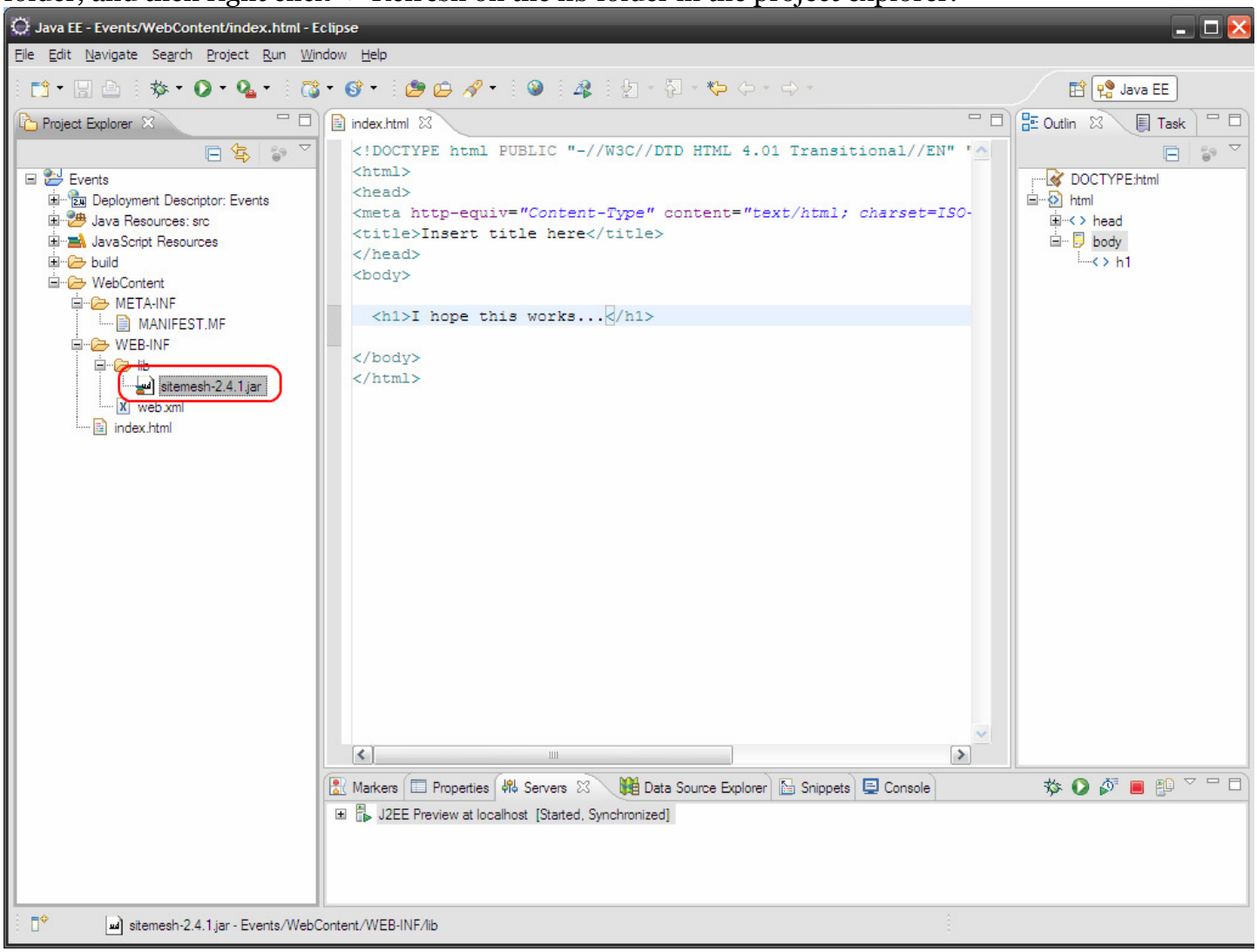

And as you can see above, it worked for me. Hopefully you'll be as lucky.

# Using Sitemesh for a master page

Most applications will have some common HTML that you want shared across all your pages. This will usually consist of your menu bar, css, logos etc. Sitemesh is a good way to accomplish this, as Struts does not have a built in way to do this. This common html is known as a Master page in Asp.net, or a template elsewhere.
Go to http://www.opensymphony.com/sitemesh/download.action and download the latest JAR version:



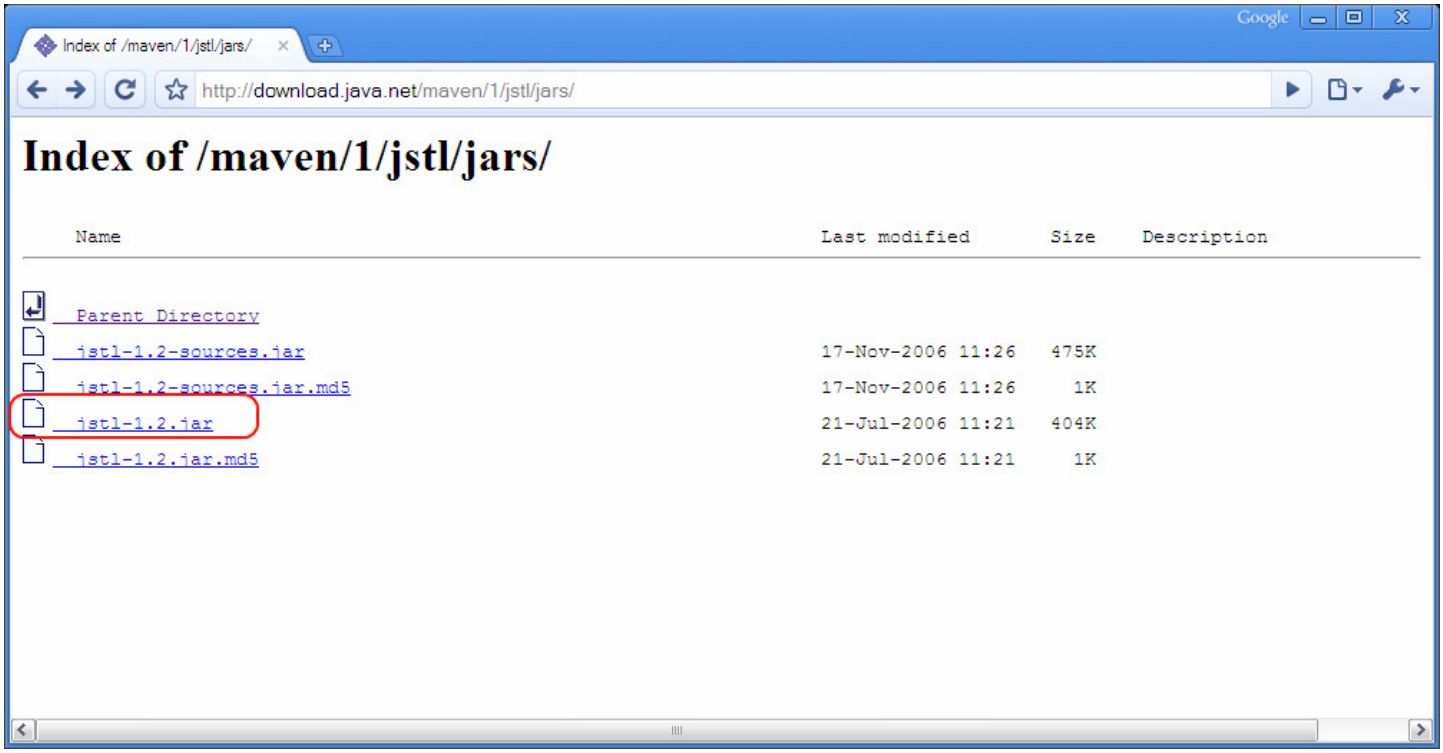
Then drag-and-drop the downloaded 'sitemesh-*.jar' file from explorer into your eclipse window, dropping it onto the Events\WebContent\WEB-INF\lib folder in the project explorer.
If you ever have problems dragging files onto the lib folder in Eclipse, you may have to unzip them first (I've had issues dragging and dropping from inside a zip file straight into Eclipse).
Alternatively, you can copy it into your 'C:\eclipse_workspace\Events\WebContent\WEB-INF\lib' folder, and then right click -> Refresh on the lib folder in the project explorer:




Please remember these steps to including Jar files in your application, as it'll be something we'll be doing quite often. If, later on, you ever see an error in your console like a 'class not found exception', firstly check that you've put all the Jars in the lib folder and that you've refreshed it in the eclipse project explorer

Next we need to include JSTL in the project, as it will be used by our sitemesh master template. JSTL gives JSP files extra abilities that we will use later so that the template will highlight the correct menu tab and show the correct submenu. Download jstl-1.2.jar from here:

http://download.java.net/maven/1/jstl/jars/jstl-1.2.jar

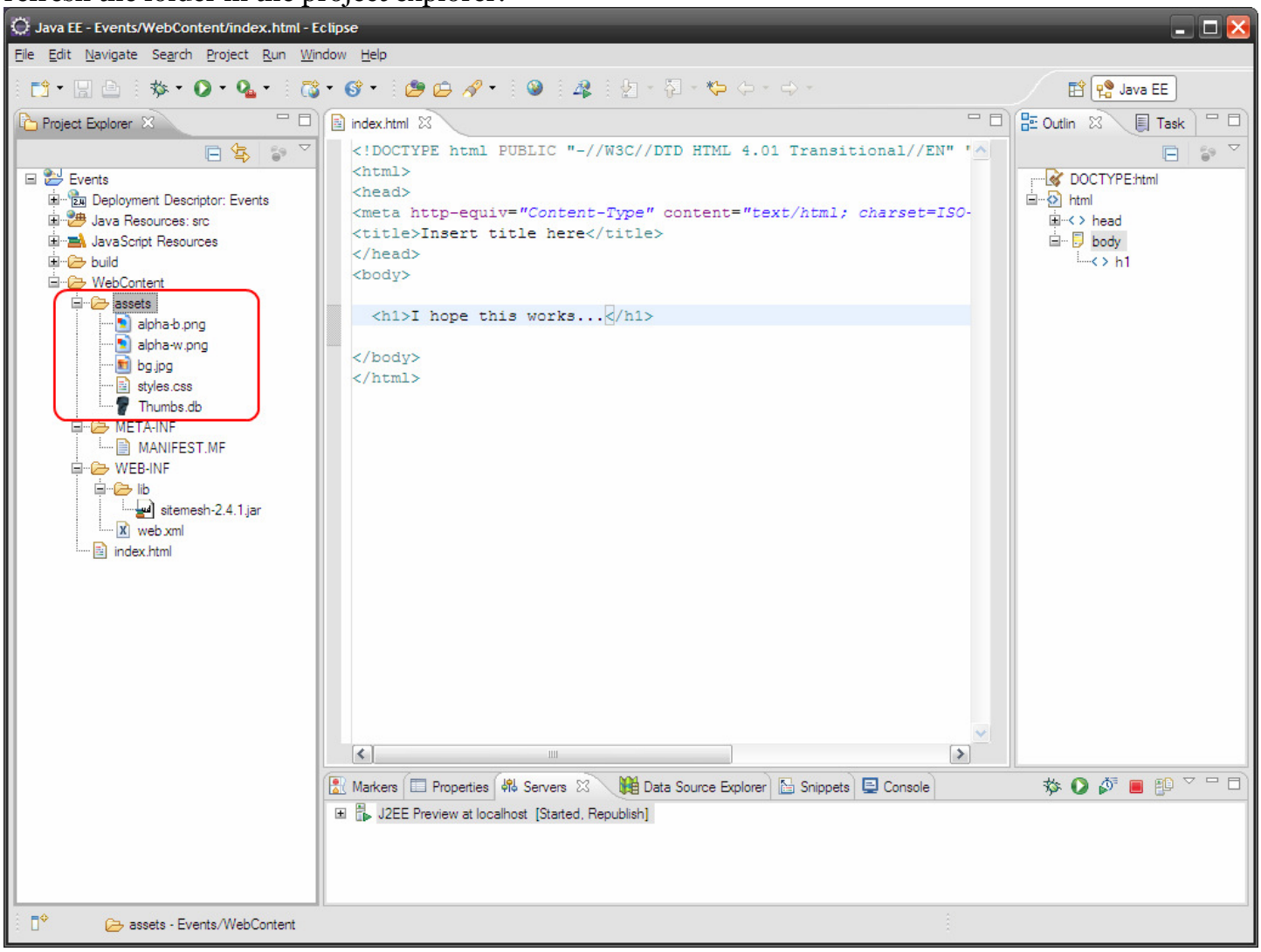



Once you've downloaded the Jar, copy it to your project's WebContent\WEB-INF\lib folder as you did for with the Sitemesh Jar.

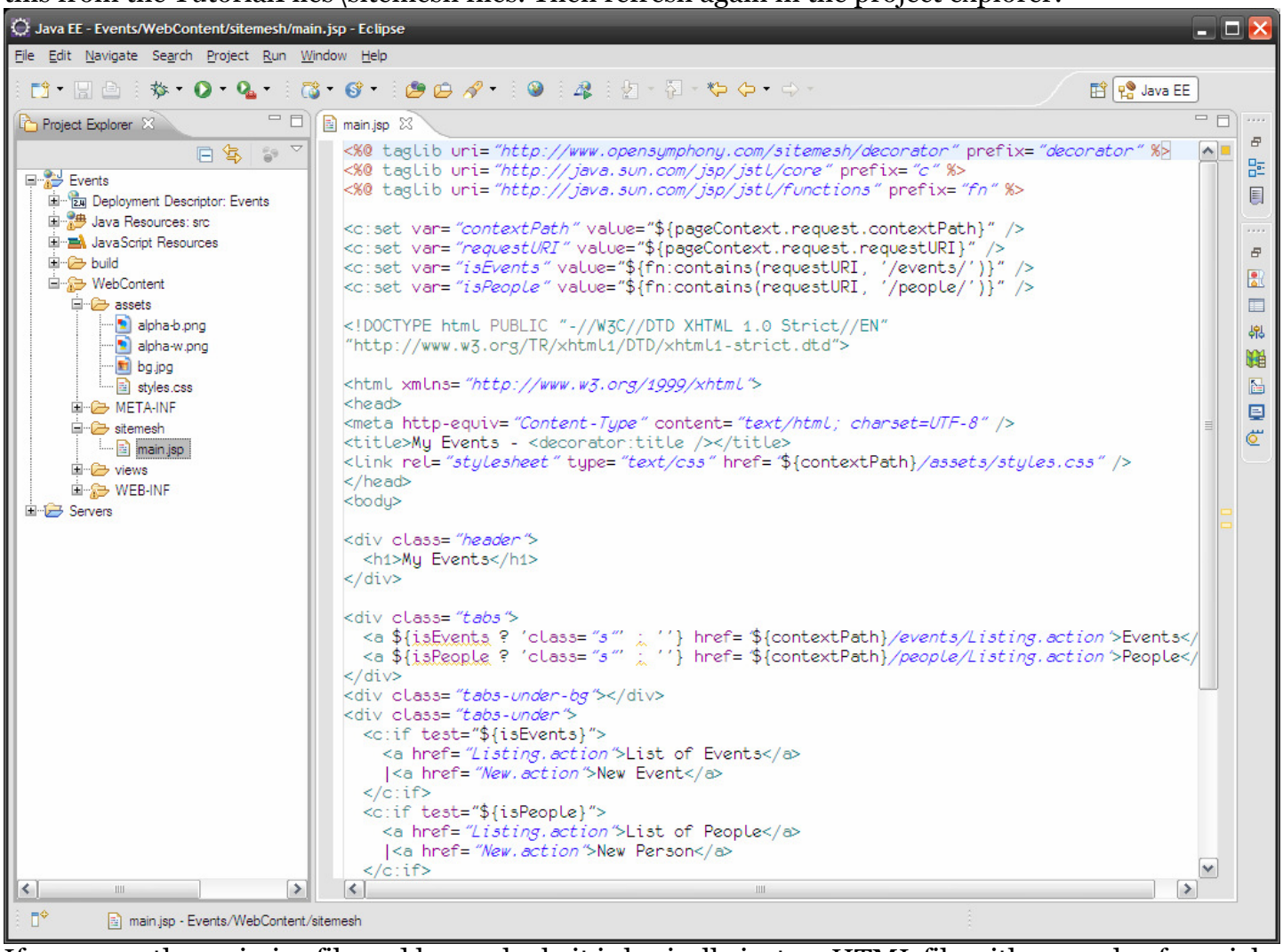
You can read further about JSTL here if you wish:

http://www.ibm.com/developerworks/library/j-jstl0211.html
http://www.mularien.com/blog/2008/04/24/how-to-reference-and-use-jstl-in-your-web-application/

Now we need to copy in the assets that are used by the template for this site. Create an 'assets' folder under WebContent, and copy in all the files from the zip file that came with this tutorial (the link to get the zip file is on my blog, for the address see the first page of this document). An easy way to do this is to drag the 'assets' folder from windows explorer and drop it on 'WebContent' in Eclipse. Then refresh the folder in the project explorer:

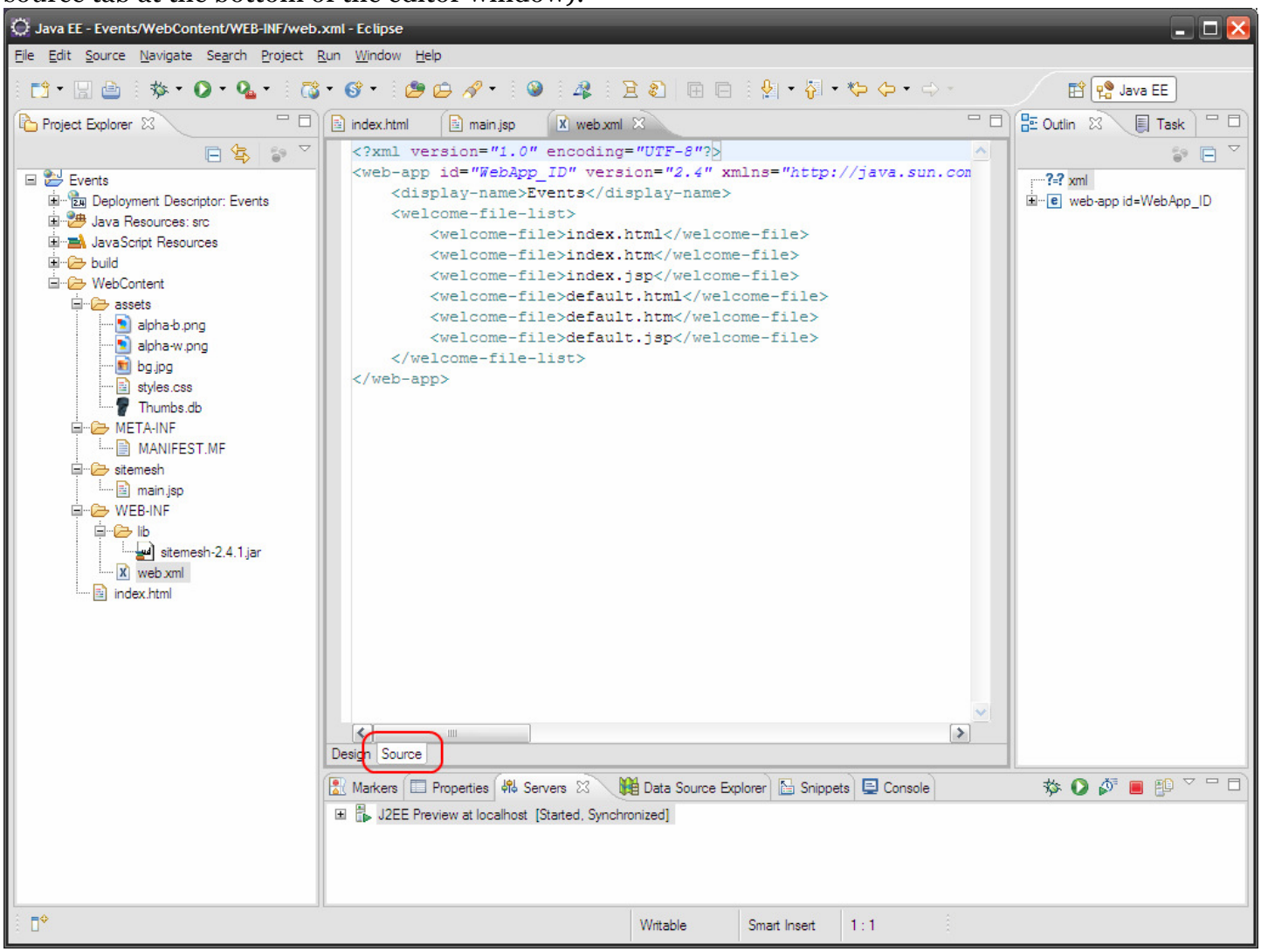


Next up, we need to create the WebContent\sitemesh folder where the template will go. You can copy this from the TutorialFiles\sitemesh files. Then refresh again in the project explorer:




If you open the main.jsp file and have a look, it is basically just an HTML file with a couple of special JSP tags. The important one is the *<decorator:body />* line. This is a placeholder that gets replaced with the contents of the <body>...</body> section of your pages.
Also, there are the *<c:if...* and other bits of JSTL code strewn through it to highlight and show the appropriate parts of the menu, depending on which page the user is at.

Now we have to configure Sitemesh. Open the WEB-INF\web.xml file in the source view (click the source tab at the bottom of the editor window):



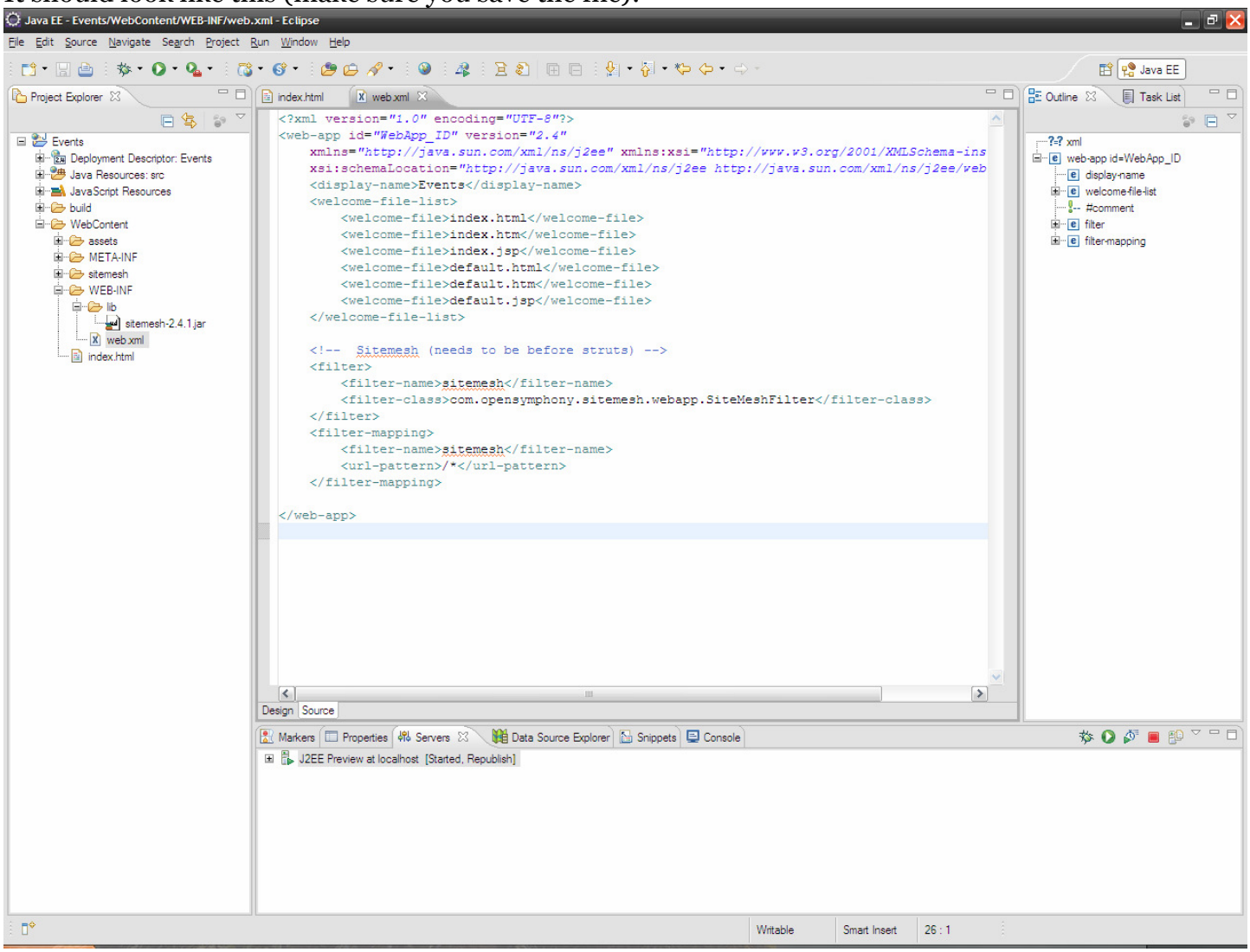
Add the following lines of code just before the close of the web-app section, just before the *</web-app>* line:

```
<!--  Sitemesh (needs to be before struts) -->
<filter>
	<filter-name>sitemesh</filter-name>
	<filter-class>com.opensymphony.sitemesh.webapp.SiteMeshFilter</filter-class>
</filter>
<filter-mapping>
	<filter-name>sitemesh</filter-name>
	<url-pattern>/*</url-pattern>
</filter-mapping>
```

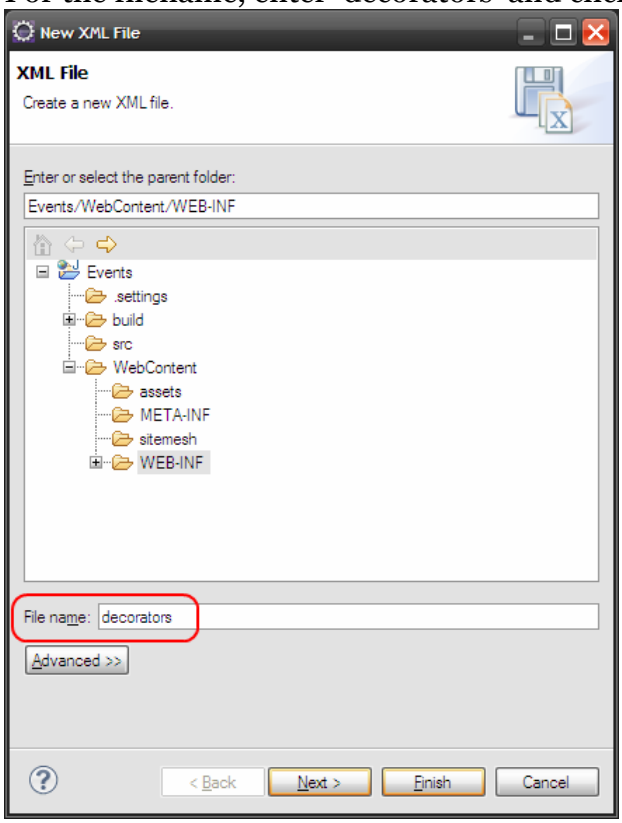
This will instruct the servlet container to pass every HTTP request through Sitemesh, so it can do it's thing.

At this point, might I suggest copying and pasting code such as above into your application from the TutorialFiles folder rather than just typing it or copying from this PDF, to save yourself the hassle of inevitable spelling errors and lost indentation.

It should look like this (make sure you save the file):



Next we need to make the sitemesh config file. Right click on WEB-INF -> New -> XML. For the filename, enter ‘decorators’ and click finish:

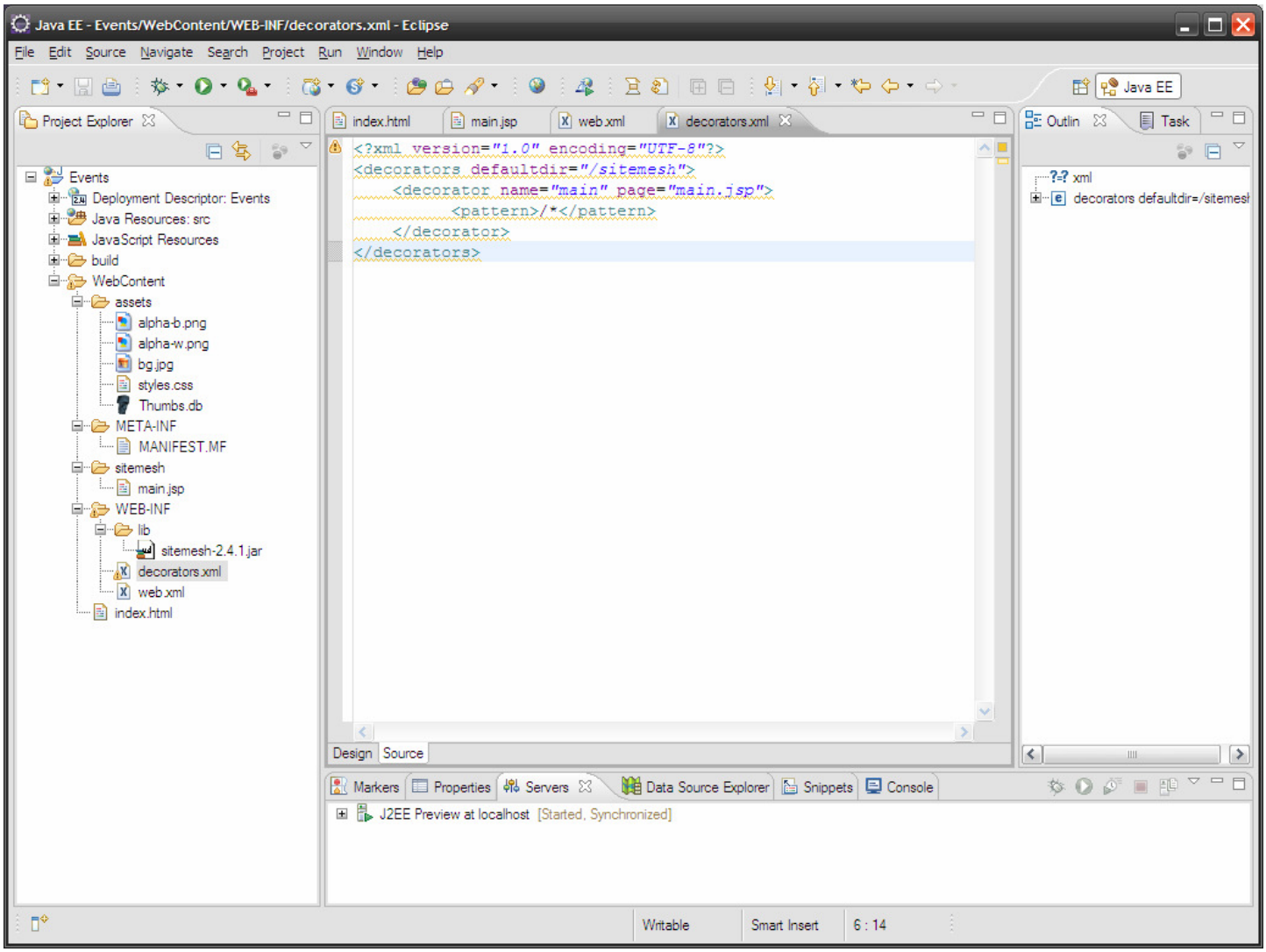



The contents of the file you just created should be:

### WebContent\WEB-INF\decorators.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<decorators defaultdir="/sitemesh">
    <decorator name="main" page="main.jsp">
        <pattern>/*</pattern>
    </decorator>
</decorators>
```

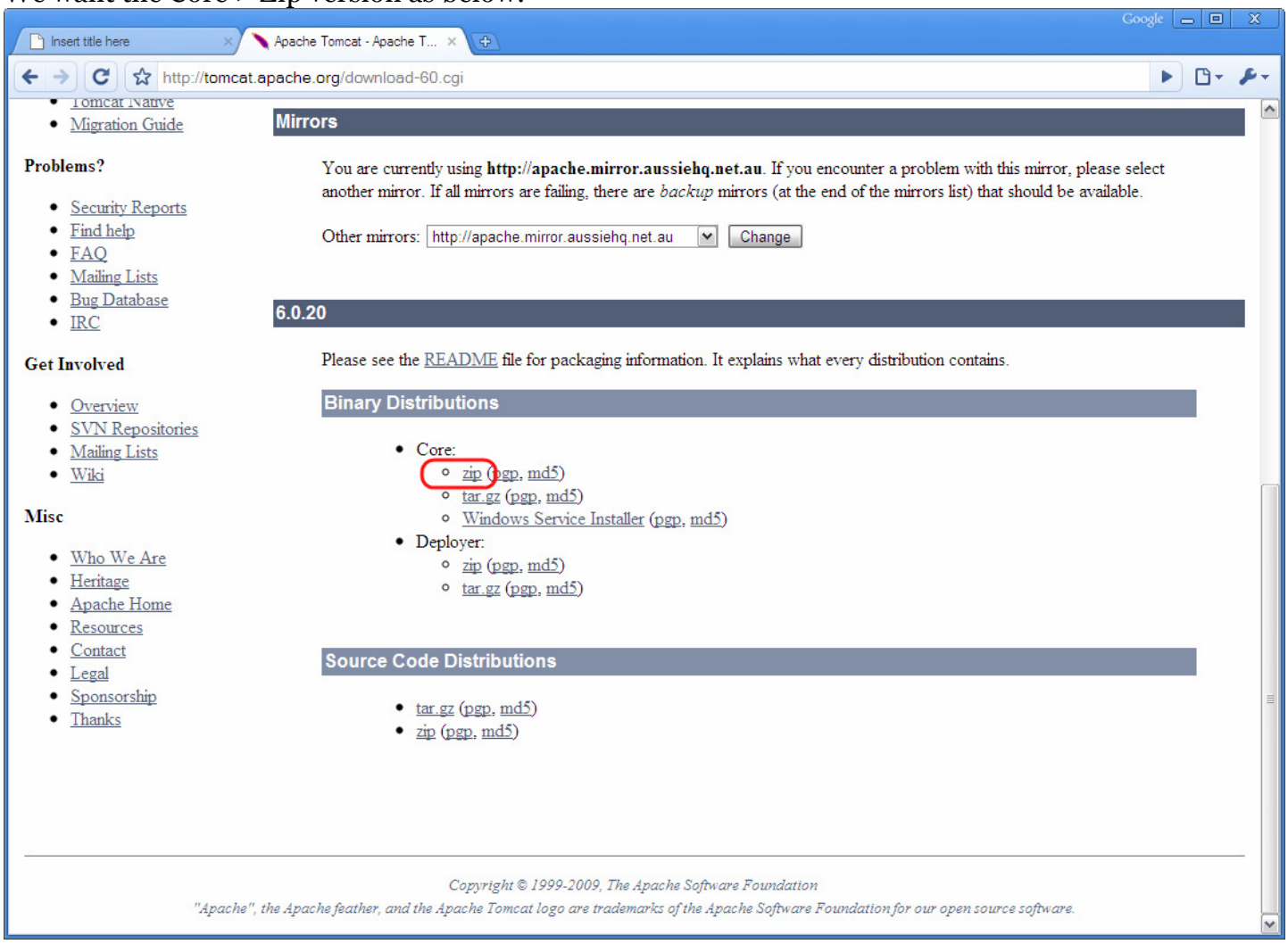
Like so:



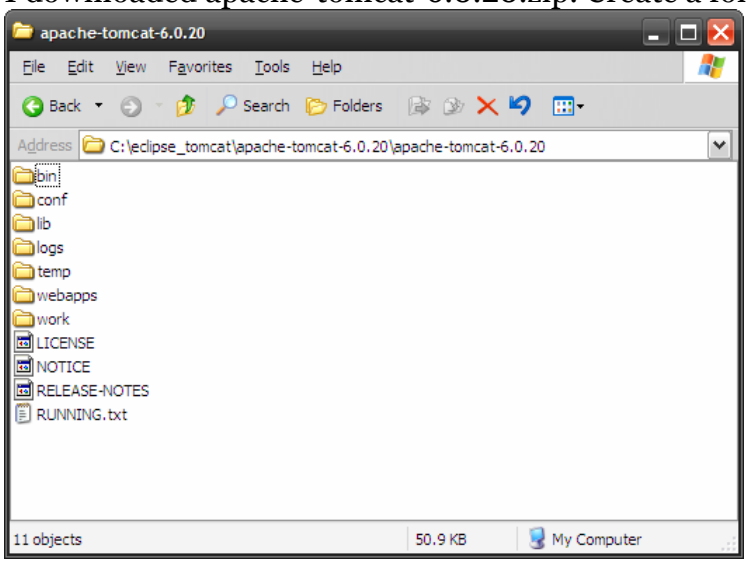

Basically, that will instruct sitemesh to apply it's master page (main.jsp) to every request. You could configure here to use different master pages for different sections of your site, if you wanted.
But before we can test our new sitemesh setup, we need to install Tomcat.

# Installing Tomcat for our development server

Now this is where the J2EE preview server stops being helpful. So go and download Tomcat 6: http://tomcat.apache.org/download-60.cgi
We want the Core > Zip version as below:

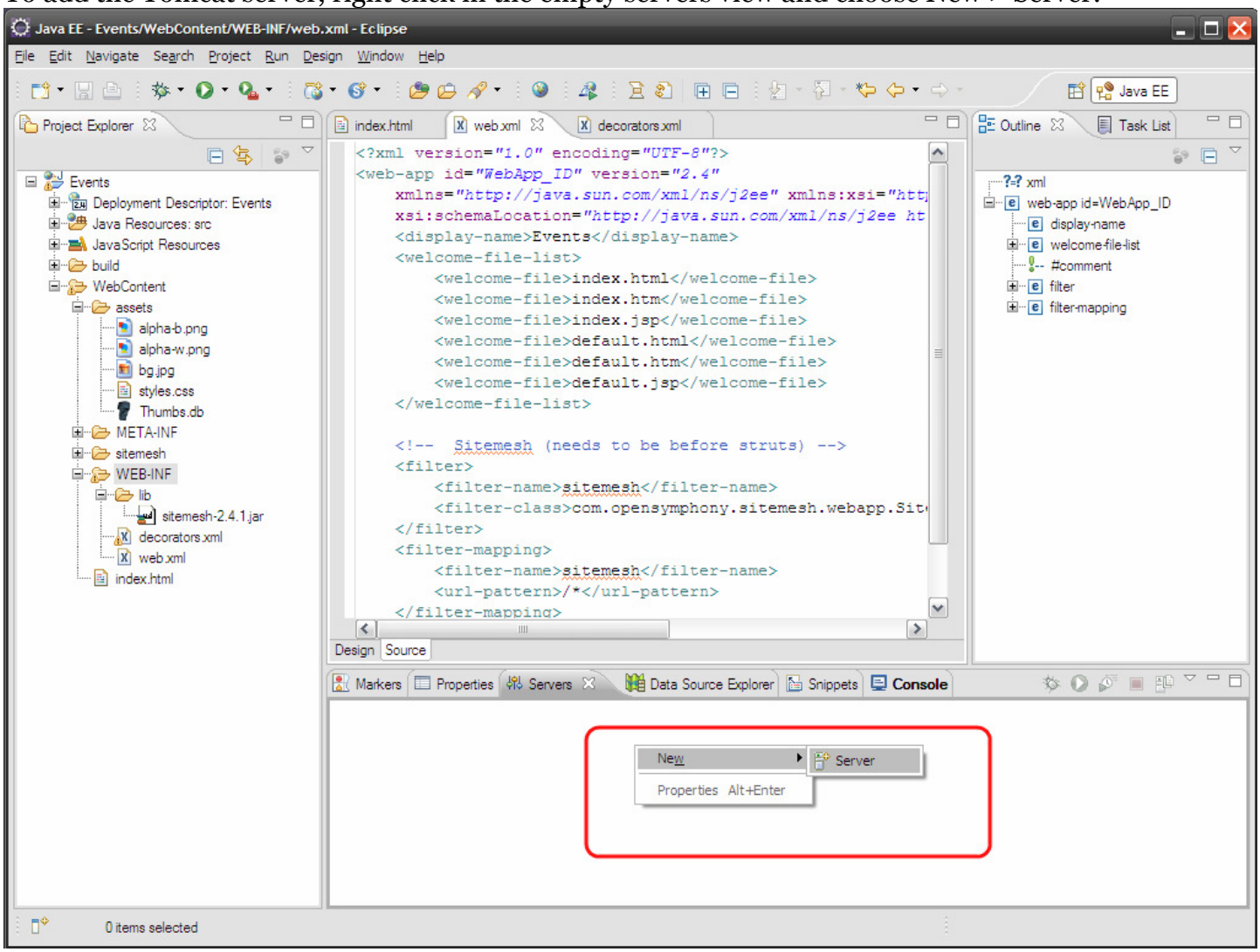



I downloaded apache-tomcat-6.0.20.zip. Create a folder C:\eclipse_tomcat and unzip it there like so:

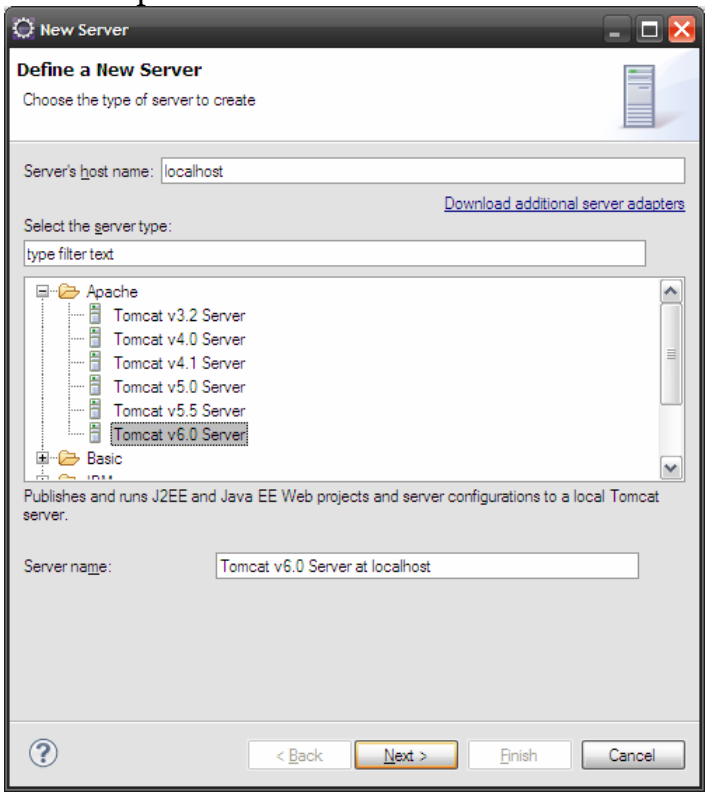



I assume you'll be downloading a newer version than mine, or maybe you're unzipping it to a different folder to me, so just remember the name of the folder that contains the bin and lib (etc) folders for later.

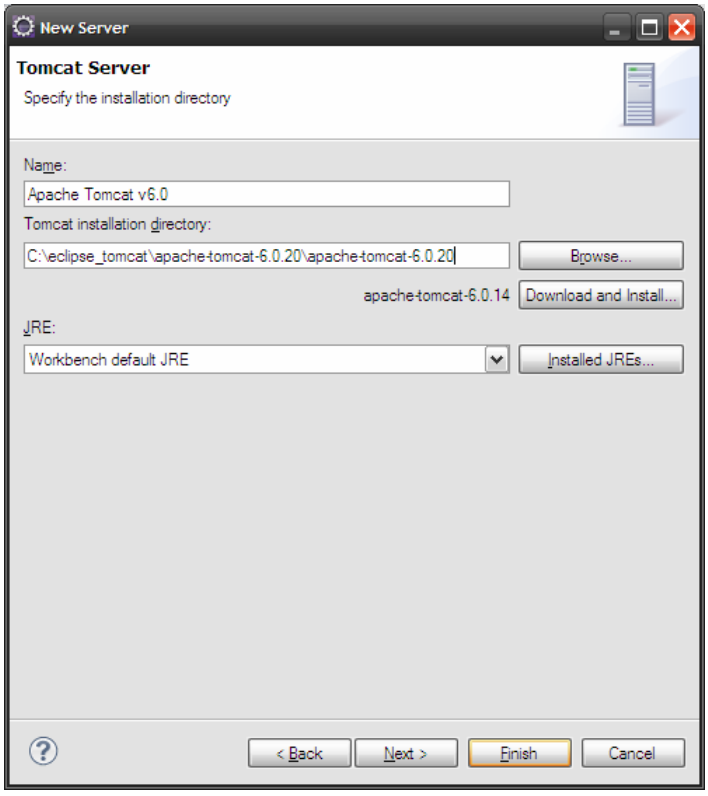
Now go into eclipse and stop and delete the J2EE Preview server from the servers view. Right click it > Stop, then Right click > Delete.
To add the Tomcat server, right click in the empty servers view and choose New > Server:



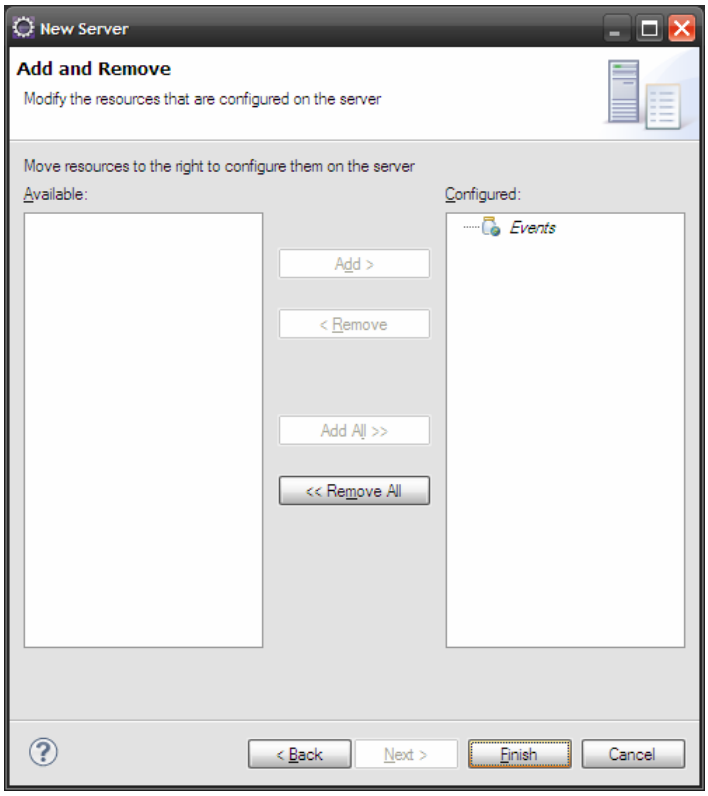
Choose Apache>Tomcat v6.0 Server and click Next:



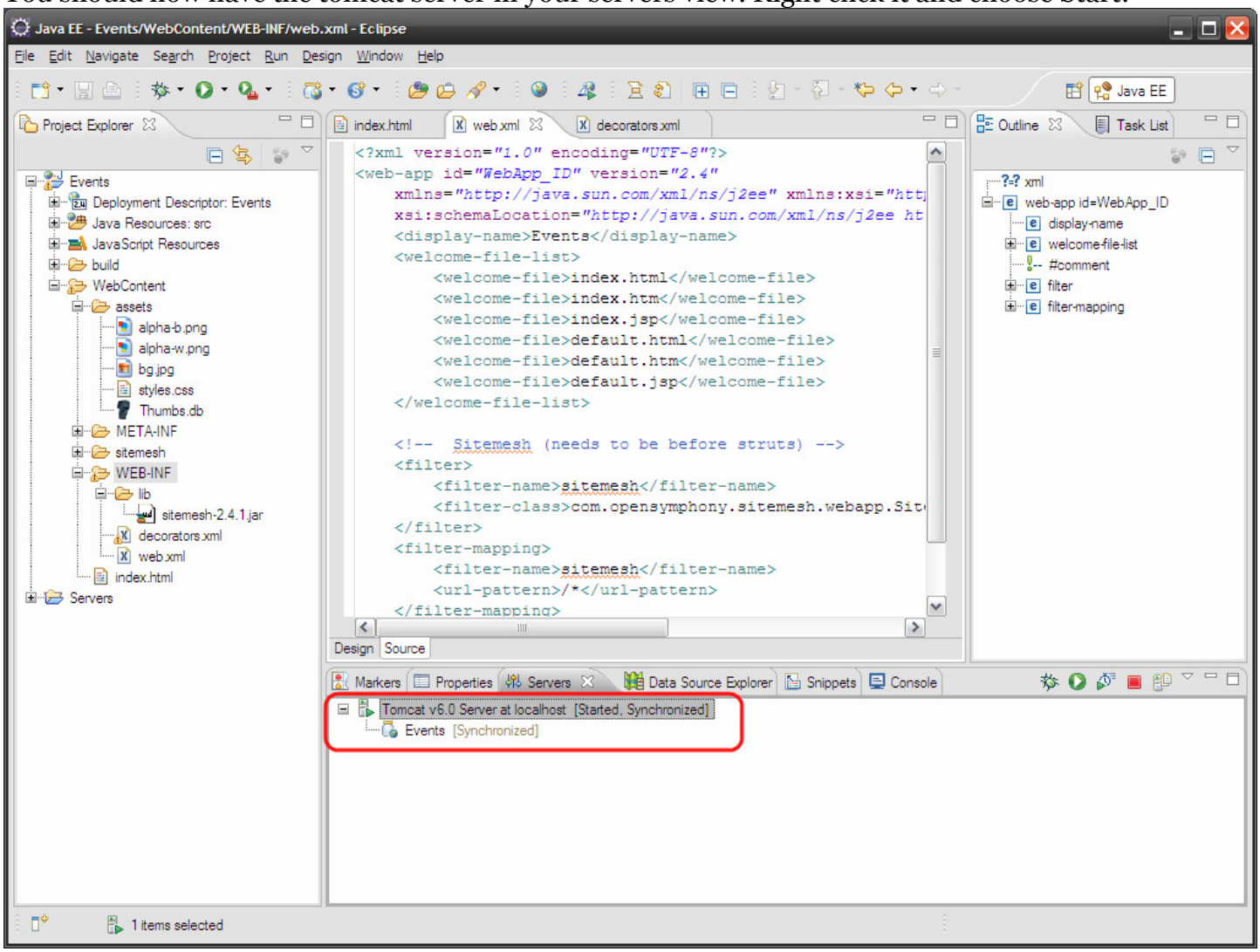

In the Tomcat installation directory, enter the folder where tomcat was unzipped from before. This folder has to contain tomcat's bin and lib (etc) subfolders for the Next and Finish buttons to be enabled. Then click Next:



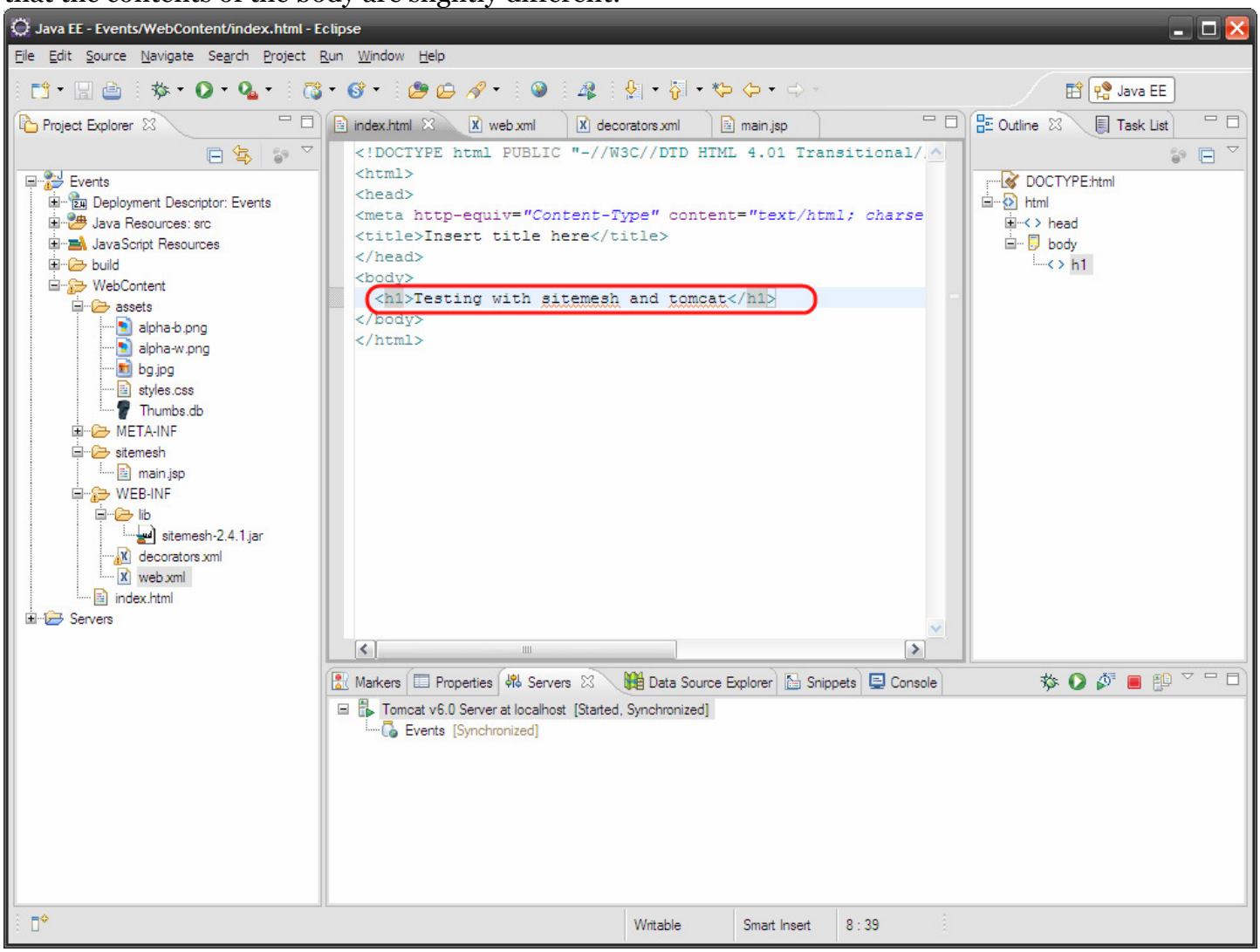
Click ‘Add All’ and then Finish:



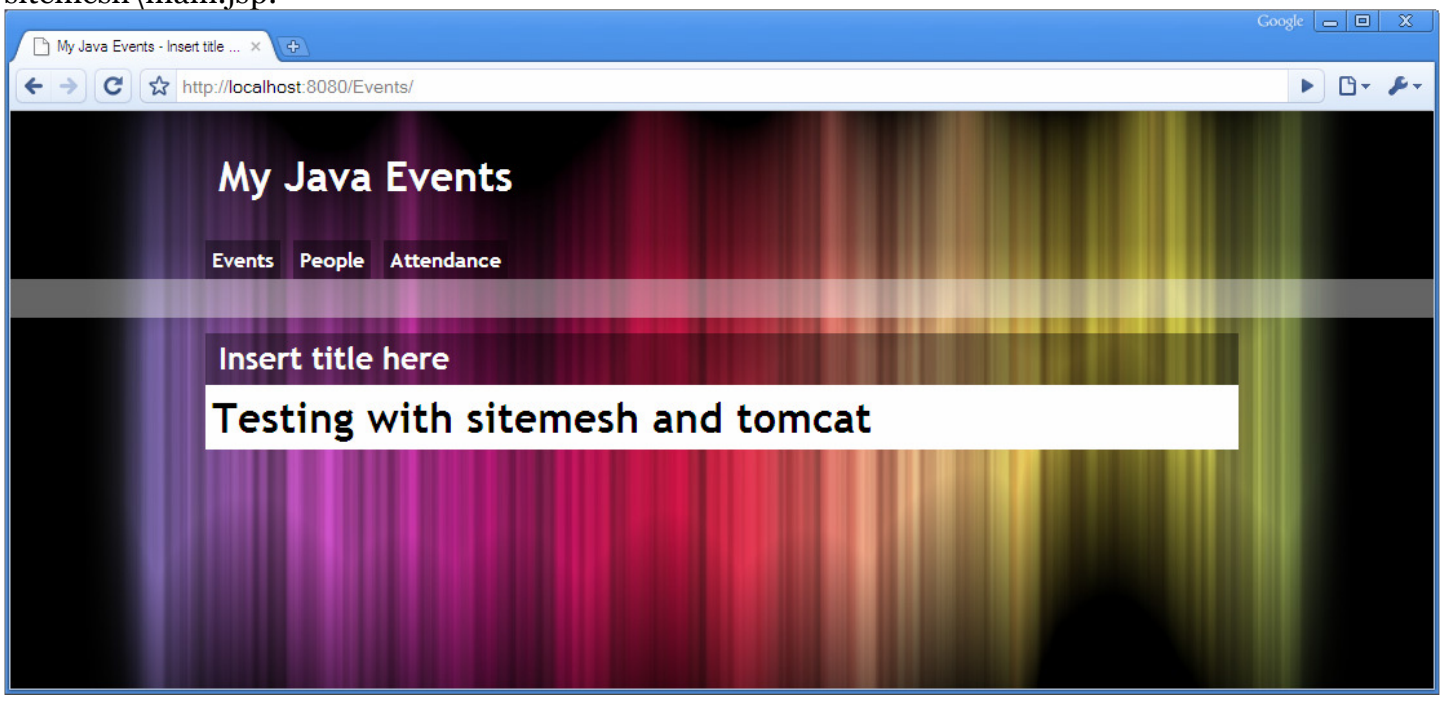

You should now have the tomcat server in your servers view. Right click it and choose Start:



I'm not sure why you have to do this (maybe it's a caching issue), but edit and save your index.html so that the contents of the body are slightly different:




Once the server goes from 'Republish' to 'Synchronized' status, go into your browser and hit refresh. You should see your message from the index.html as before, but wrapped with the template from sitemesh\main.jsp:



To see how sitemesh handles the page titles, go into your index.html and edit the title line like this:

```
<title>This is the index page</title>
```

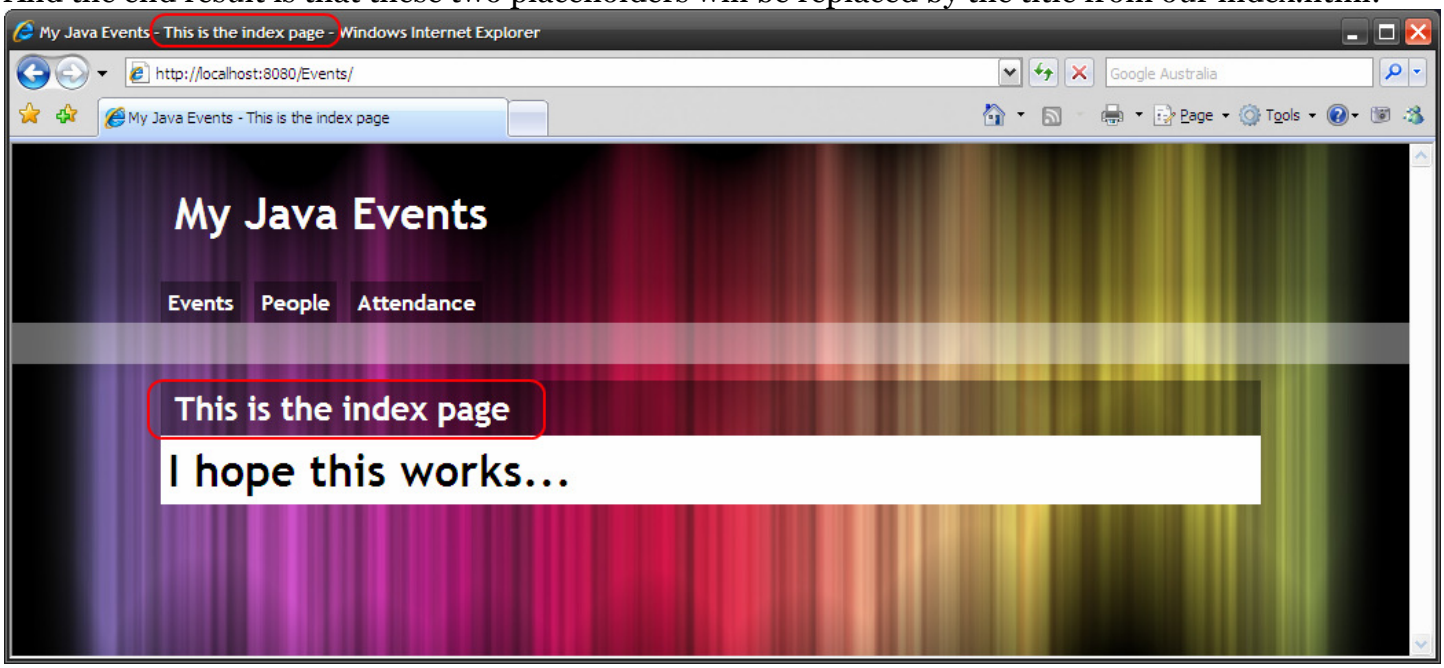
Then look into the sitemesh\main.jsp page. You can see the title placeholders by looking for 'decorator:title'. You can see it in the title line:

```
<title>My Java Events - <decorator:title default="Welcome!" /></title>
```

Also later on in the bodyhead div:

```
<div class="bodyhead">
    <h2><decorator:title default="Welcome!" /></h2>
</div>
```

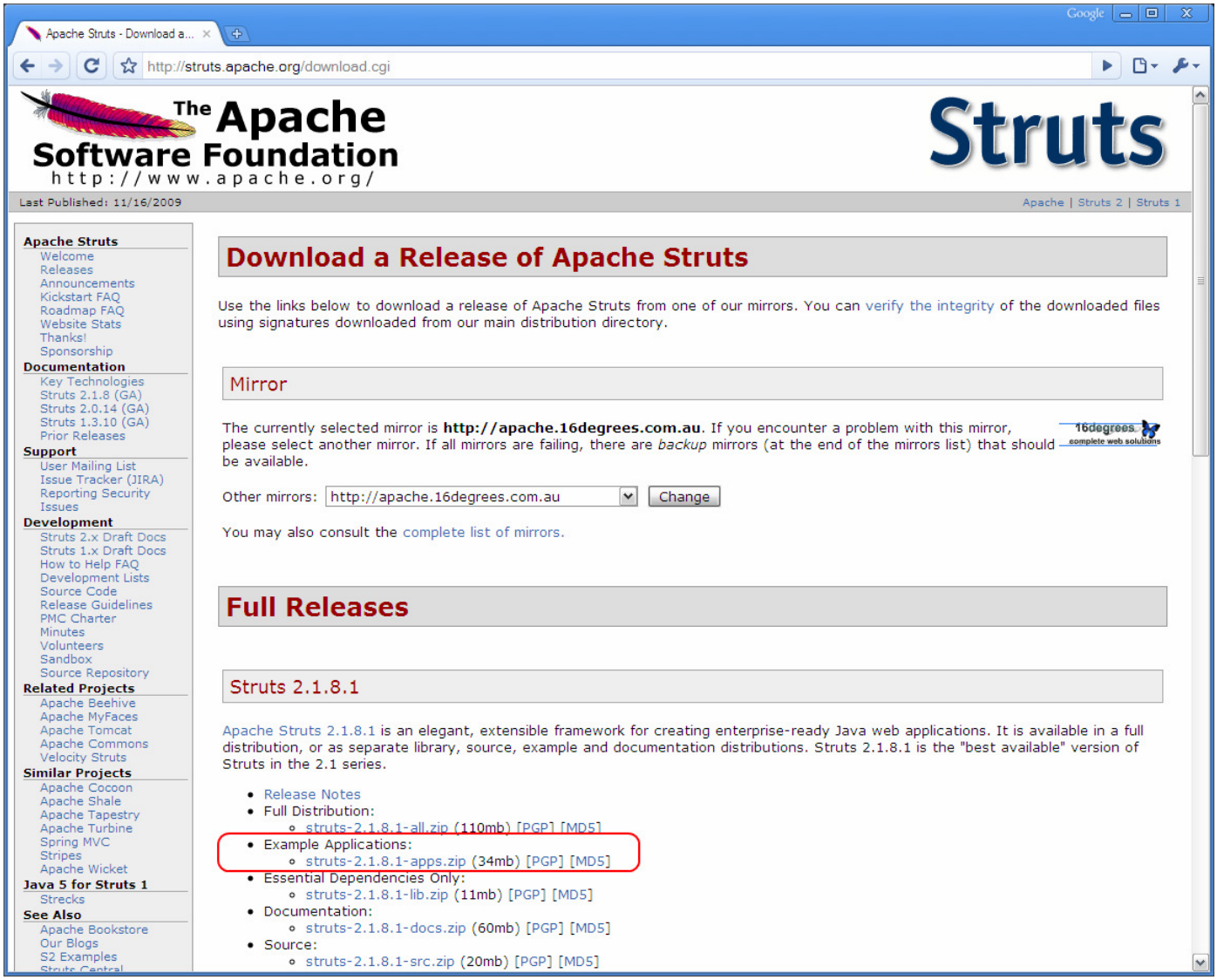
And the end result is that these two placeholders will be replaced by the title from our index.html:



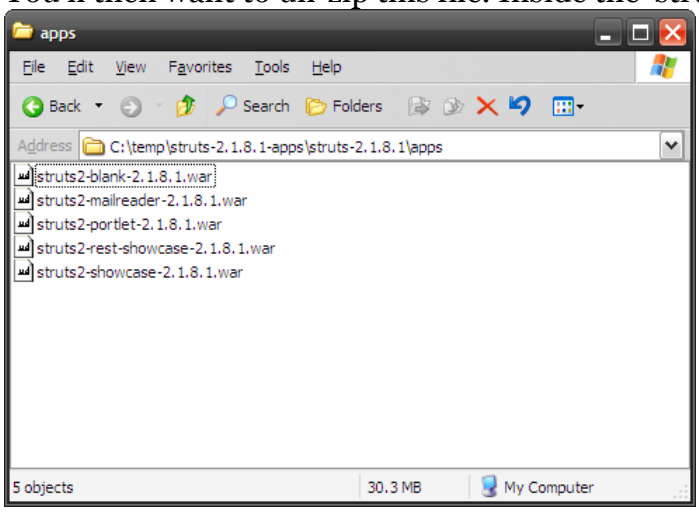

Next, we'll work on integrating Struts2 into the project.

# Struts 2

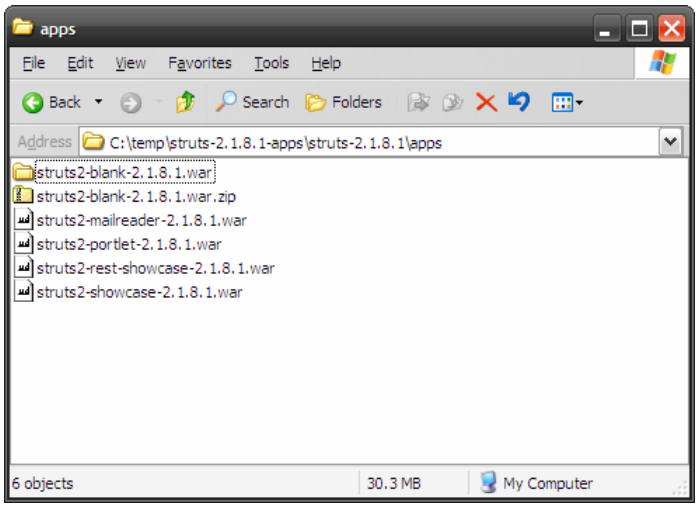
Struts 2 is a popular Java web framework, as used by many websites and web applications. Firstly, we need to download it. I downloaded it from here:
http://struts.apache.org/download.cgi
I grabbed the 'Example Applications' version (struts-2.1.8.1-apps.zip). I'll explain later why I avoided the other versions:



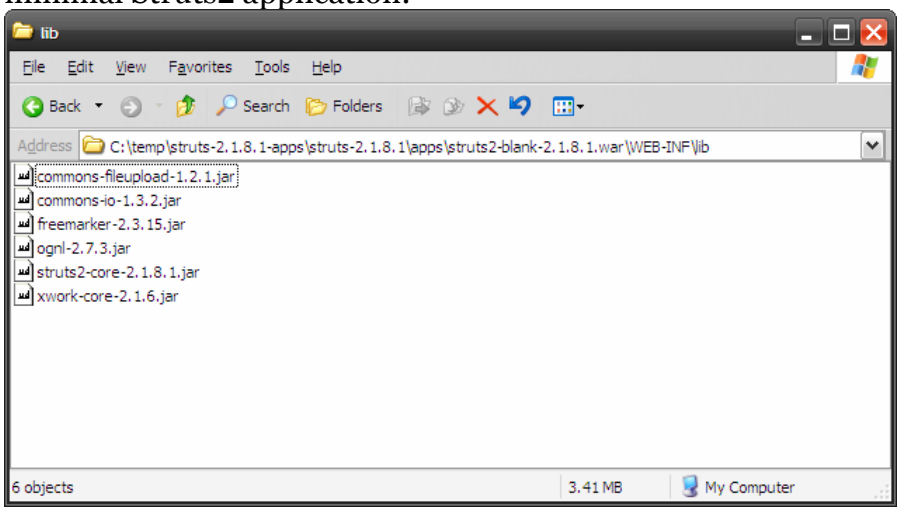

You'll then want to un-zip this file. Inside the 'struts-2.1.8.1\apps' folder are some sample WAR files:



Rename the ‘struts2-blank-2.1.8.1.war’ file to end with .zip and unzip it:

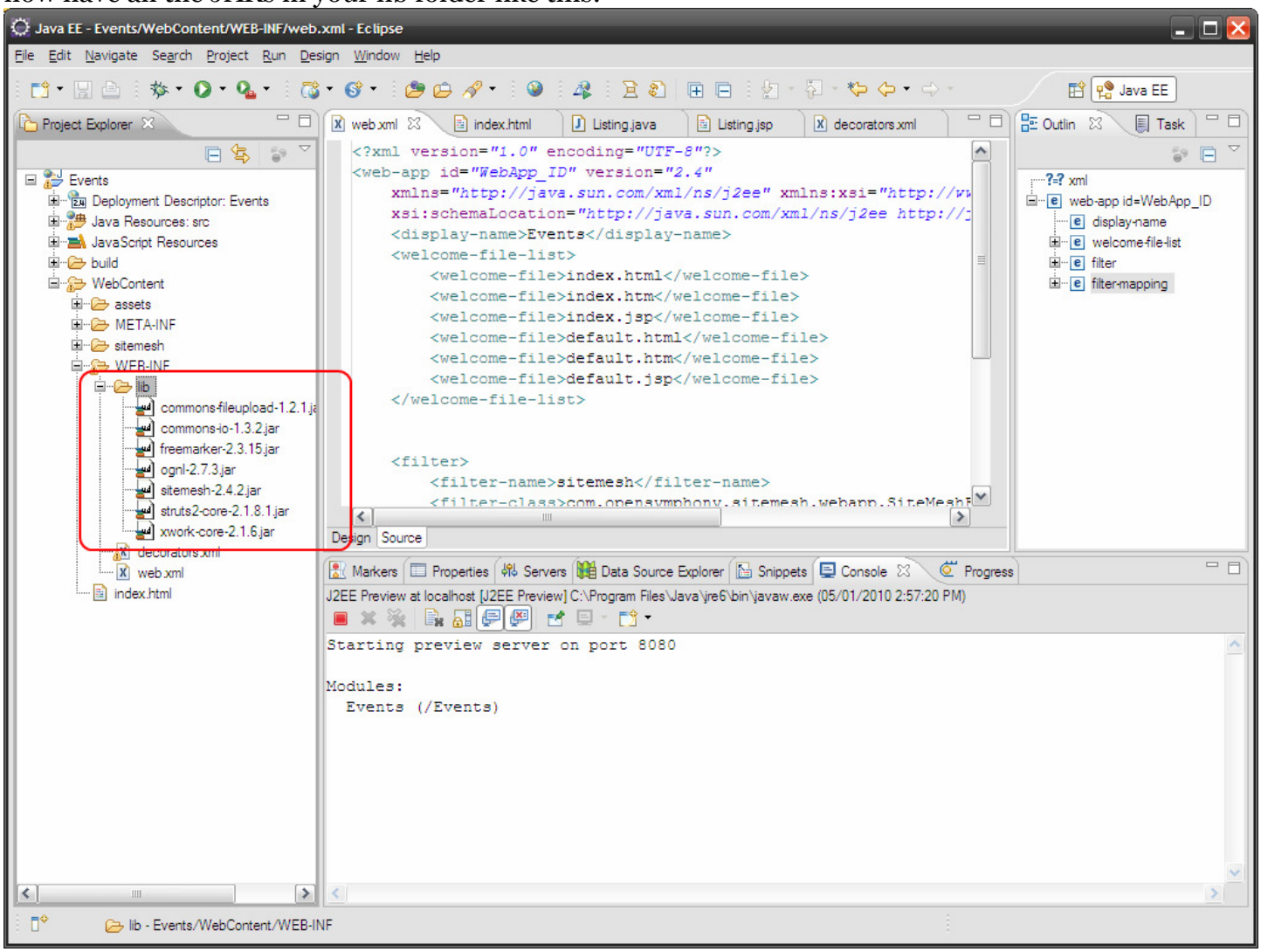



Now go into the WEB-INF\lib folder of that newly unzipped war file to find the necessary JARs for a minimal Struts2 application:



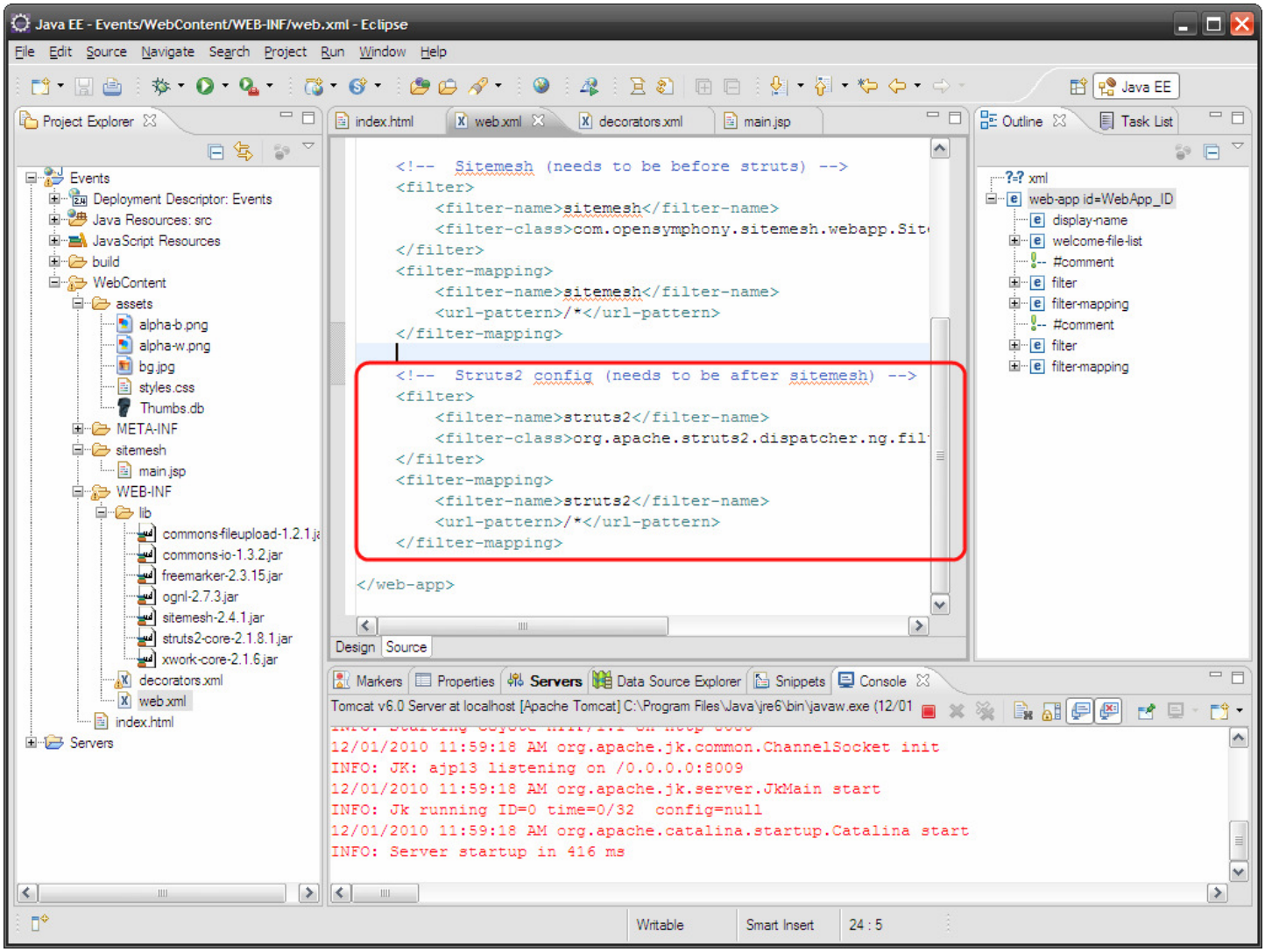

The reason I went to all the effort of grabbing the JARs from this sample application, rather than downloading the ‘struts-2.1.8.1-lib.zip’ version from the website, is that the lib version includes all sorts of extra bits and pieces that complicate matters and slow things down, which isn’t really helpful for a tutorial or indeed most of your applications.

Now copy all those jars into the WebContent\WEB-INF\lib folder of your project. Remember that you can drag and drop from an unzipped file in windows explorer straight into Eclipse. You should now have all the JARs in your lib folder like this:



Now we need to configure our WebContent\WEB-INF\web.xml to send all the web requests through Struts2. Add the following lines just after your sitemesh filter settings:

```
<!--  Struts2 config (needs to be after sitemesh) -->
<filter>
  <filter-name>struts2</filter-name>
  <filter-class>org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter</filter-class>
</filter>
<filter-mapping>
  <filter-name>struts2</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

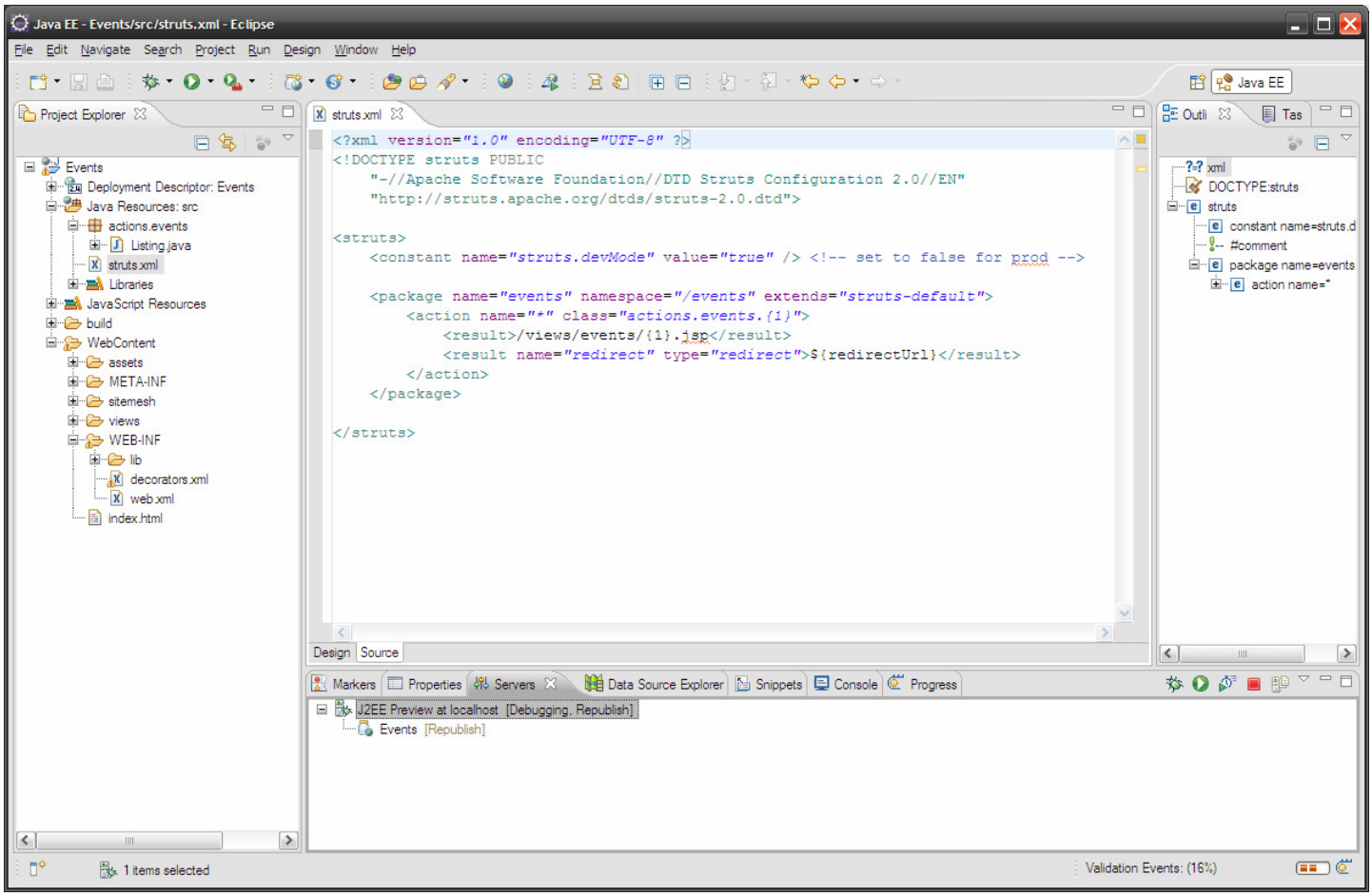



These lines instruct the servlet container to pass all web requests through struts (as we did before with sitemesh). This is how the struts libraries get their 'foot in the door' with the web request lifecycle, so to speak.

Next we'll create the struts configuration file. From the menu, choose File > New > Other > XML > XML and click Next. Choose the Events\src folder. File name is 'struts' and click finish. The contents of the file are to be:

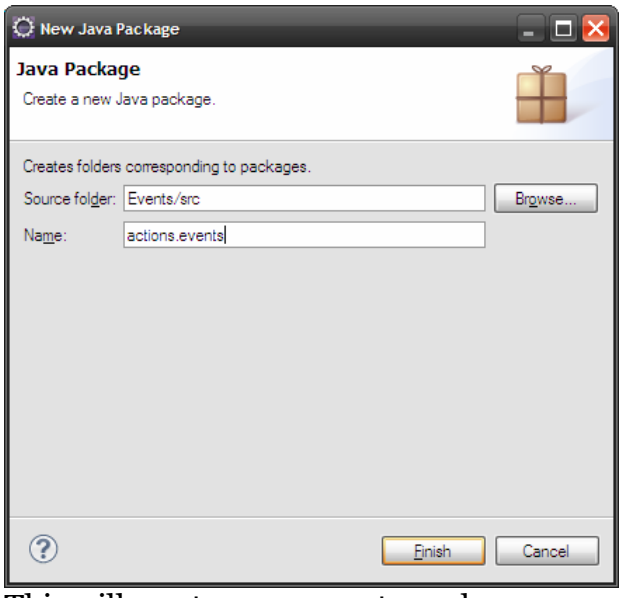
### src\struts.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
    "http://struts.apache.org/dtds/struts-2.0.dtd">

<struts>
  <constant name="struts.devMode" value="true" /> <!-- set to false for prod -->

  <package name="events" namespace="/events" extends="struts-default">
    <action name="*" class="actions.events.{1}">
      <result>/views/events/{1}.jsp</result>
      <result name="redirect" type="redirect">${redirectUrl}</result>
    </action>
  </package>

</struts>
```

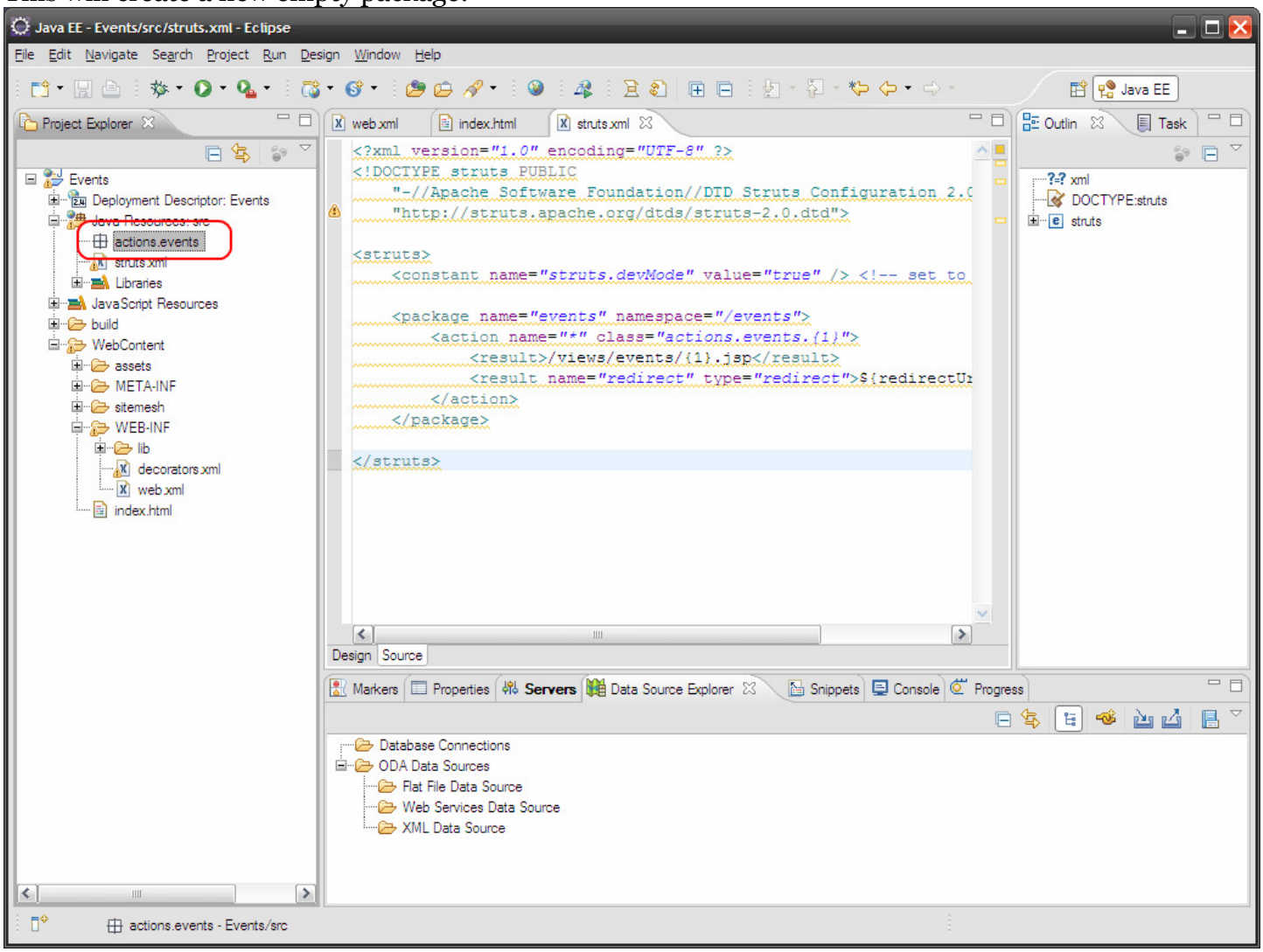



This is a basic struts configuration. Basically it says to run in development mode, and sets us up for one action package, where it's views are stored, and gives it the ability to send redirect responses.

Now we'll create the action code for an events listing. First we have to make a package called events. Right click on 'Java Resources: src' (hereafter referred to as simply 'src') > New > Package. Call it 'actions.events':

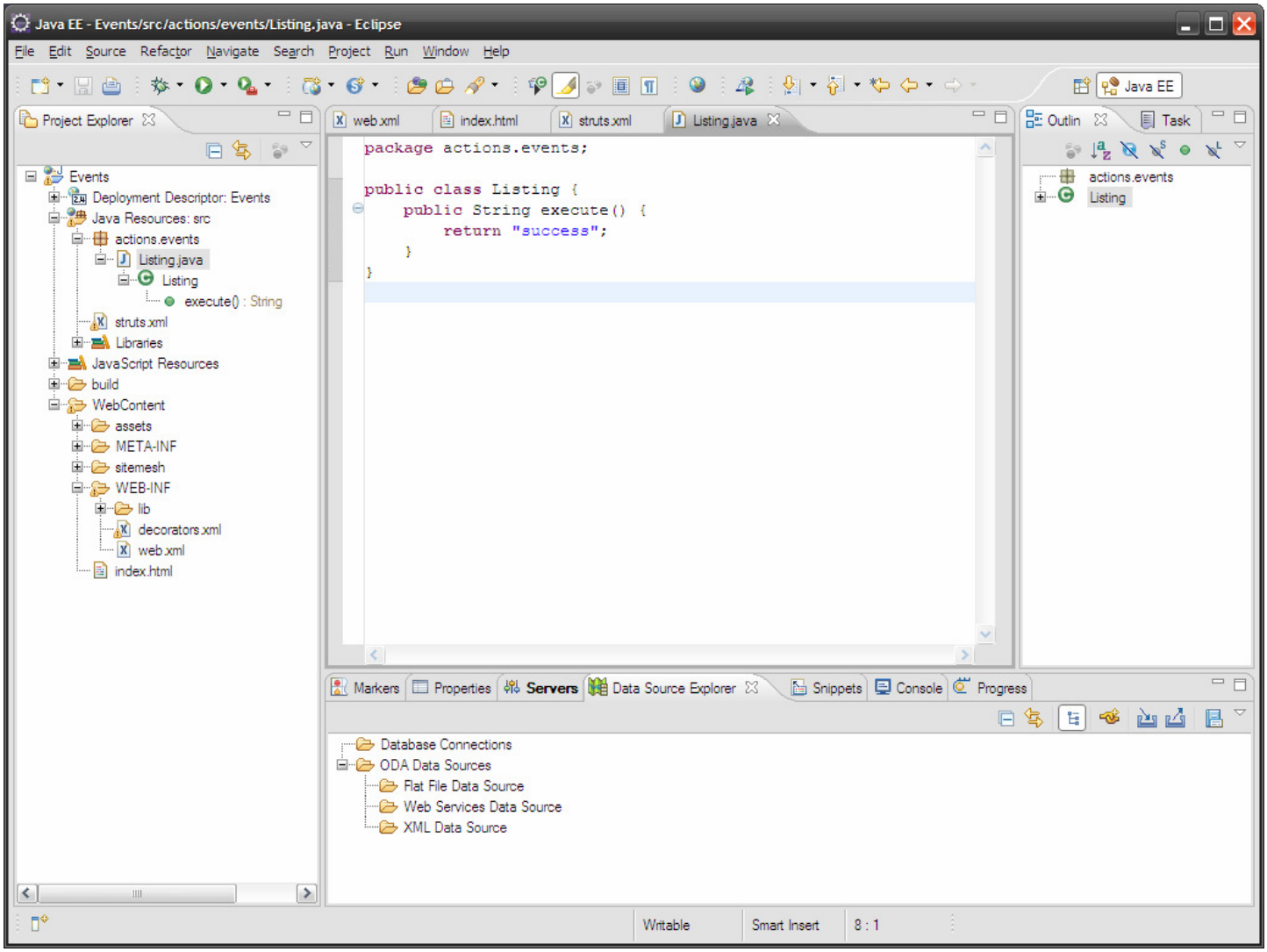



This will create a new empty package:



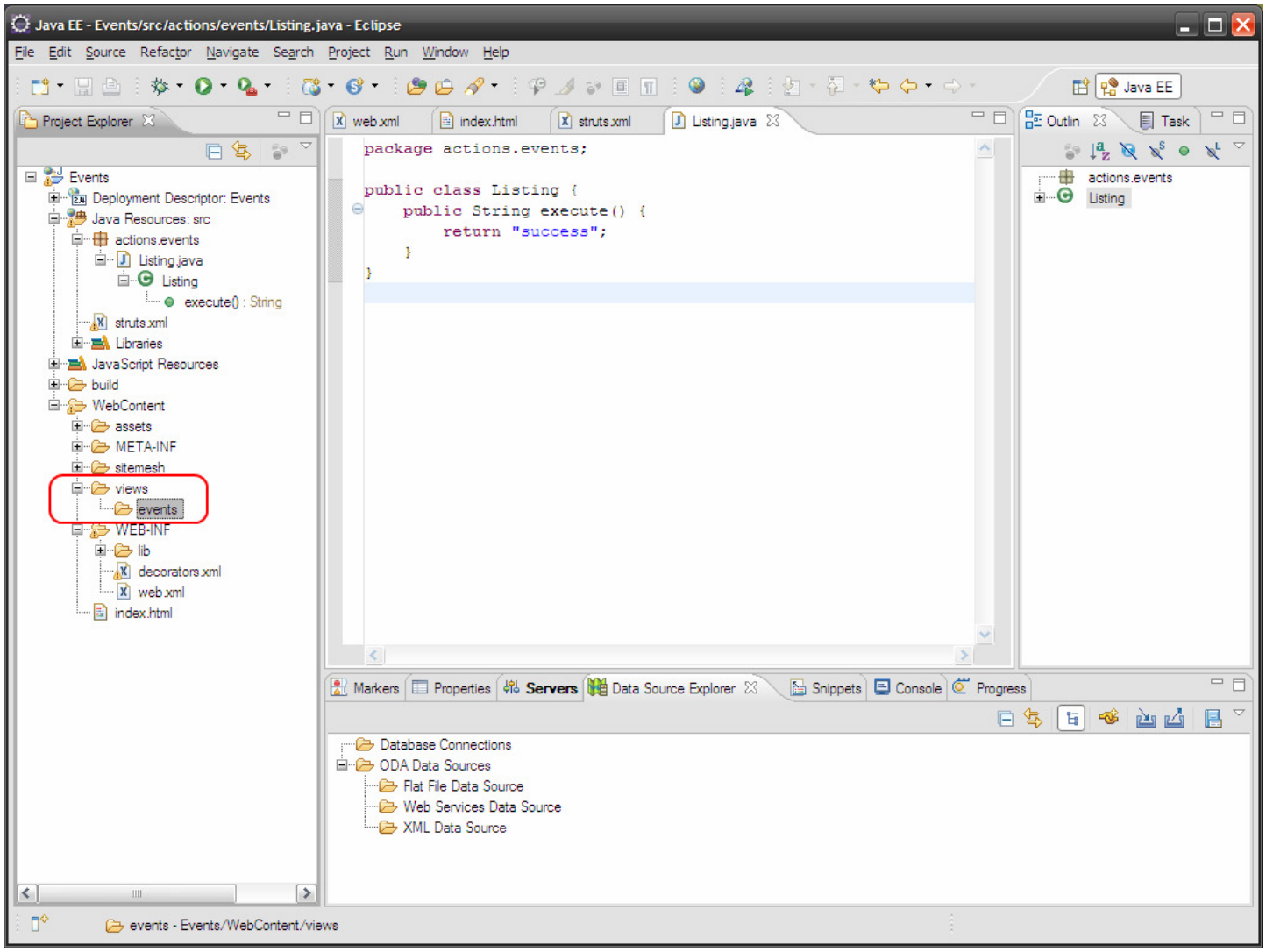
Then, on your new package in the project explorer, right click > New > Class. Enter the name 'Listing' and click finish. This file should have the following contents:

```
package actions.events;
public class Listing {
  public String execute() {
    return "success";
  }
}
```

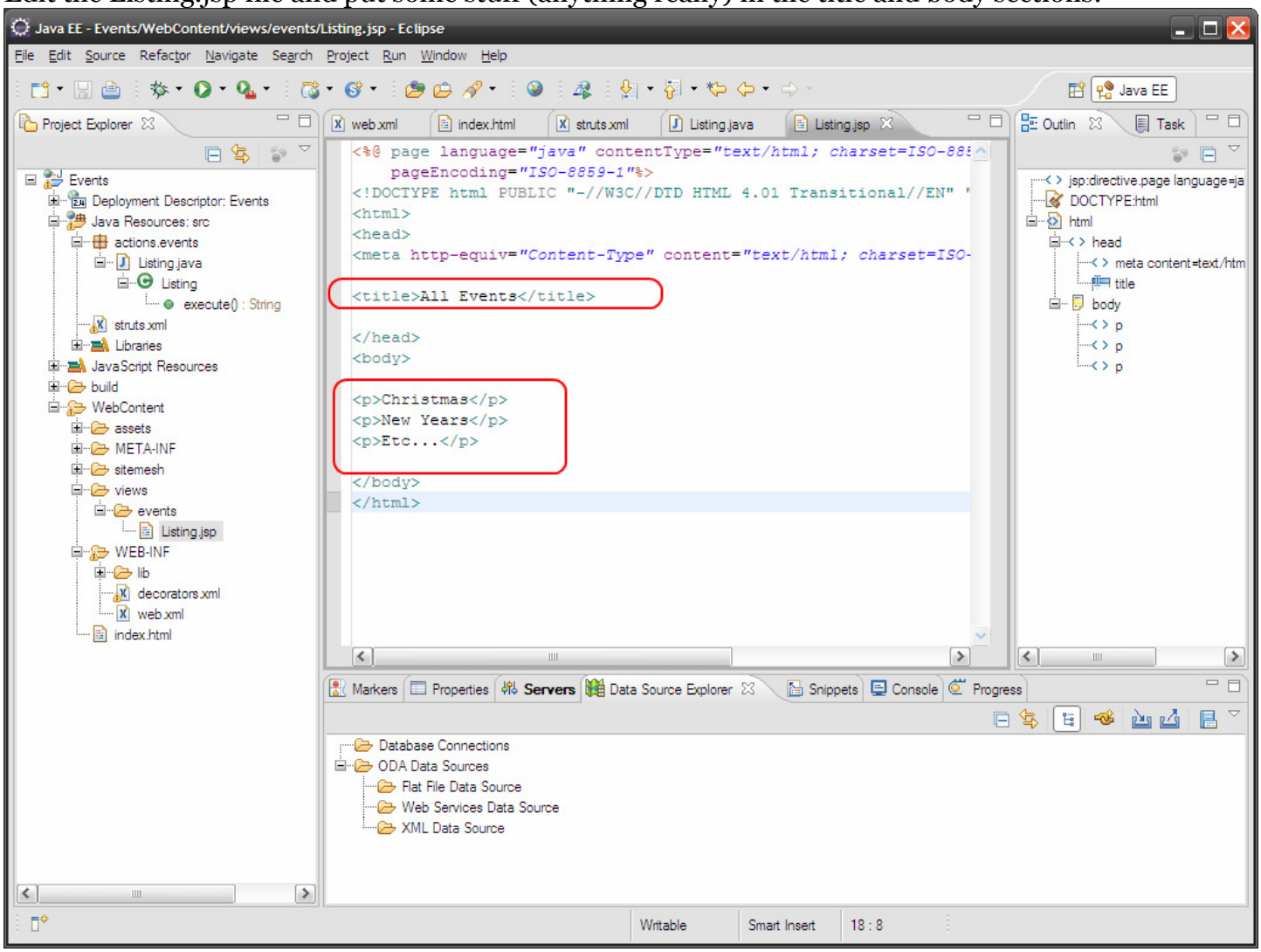



This class acts as the 'Controller' in the MVC pattern. Basically the 'execute' function is what will get called each time someone browses to http://myserver/myapp/events/Listing.action, and it is responsible for loading all the data necessary to display the page, and updating any data if the user has posted changes. If you need a refresher on what MVC and a Controller is, have a read: http://en.wikipedia.org/wiki/Model–view–controller

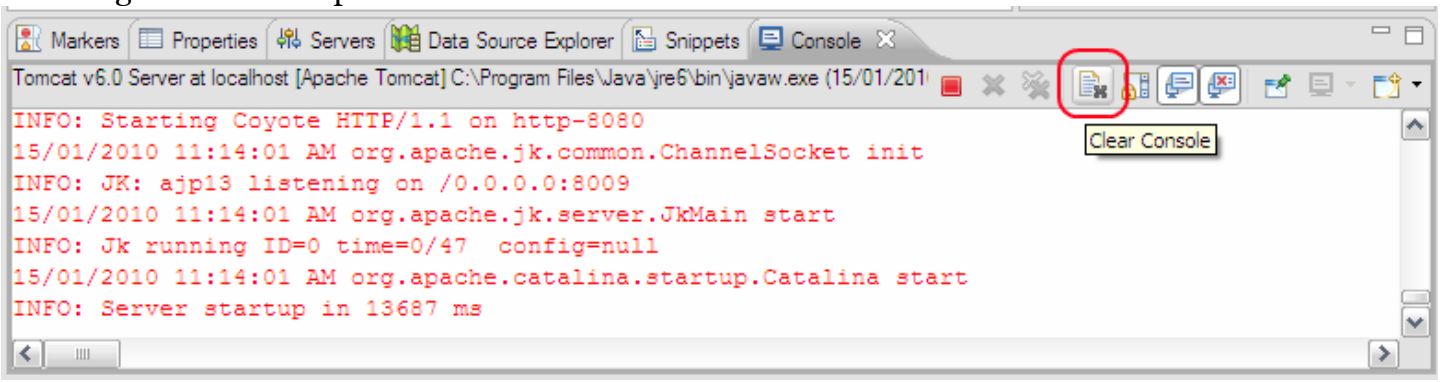
Next we need to create the corresponding 'View' for this events listing action. Create the folders WebContent\views and WebContent\views\events:



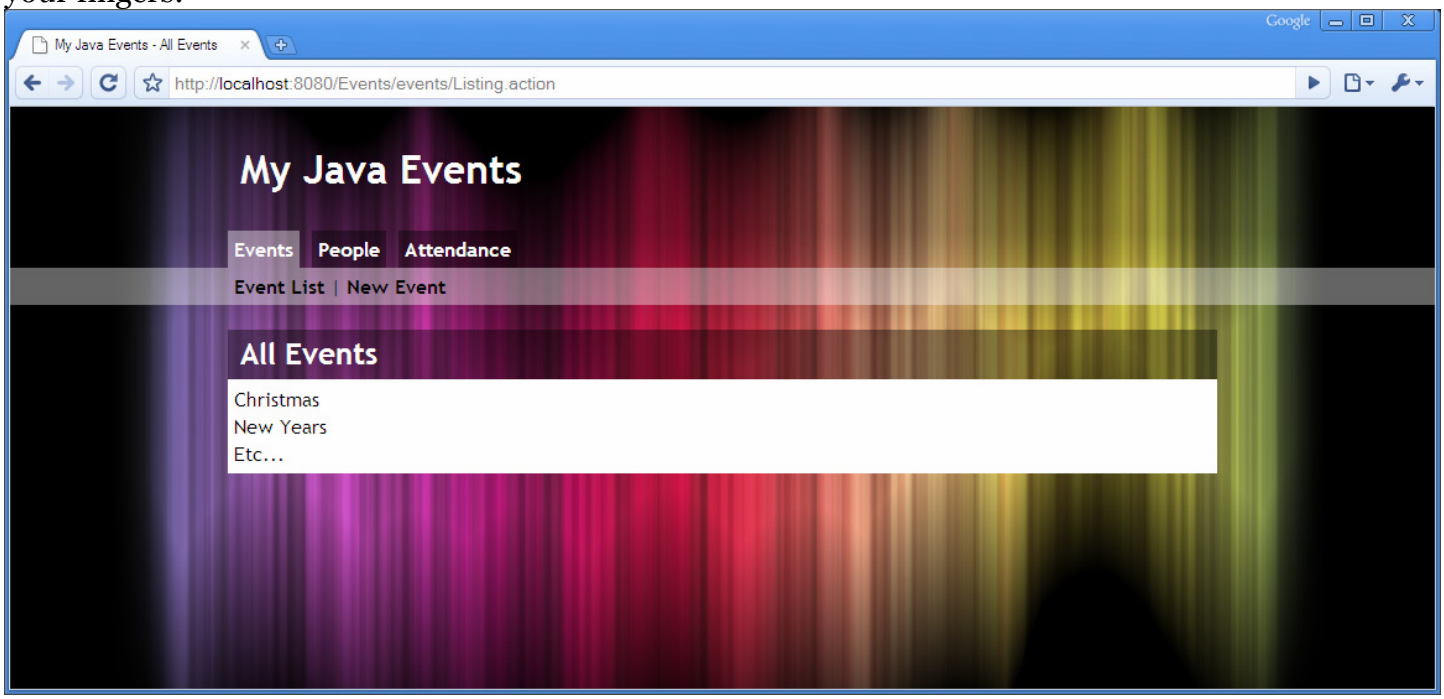
Right click on the events folder > New > JSP. Enter 'Listing' for the file name and click finish.
Edit the Listing.jsp file and put some stuff (anything really) in the title and body sections:




Lets give it a try. If you look in the Servers view, you'll notice that the server has a status of 'Restart'. This means it'll need to be restarted to reflect your changes. Right click it and choose Restart, and wait as it does it's thing.
You may want to have a look in the Console view to see if there were any errors reported. If there are any errors, it's a good idea to clear the console (as below) then restart the server. Otherwise you may be seeing errors from a previous run.

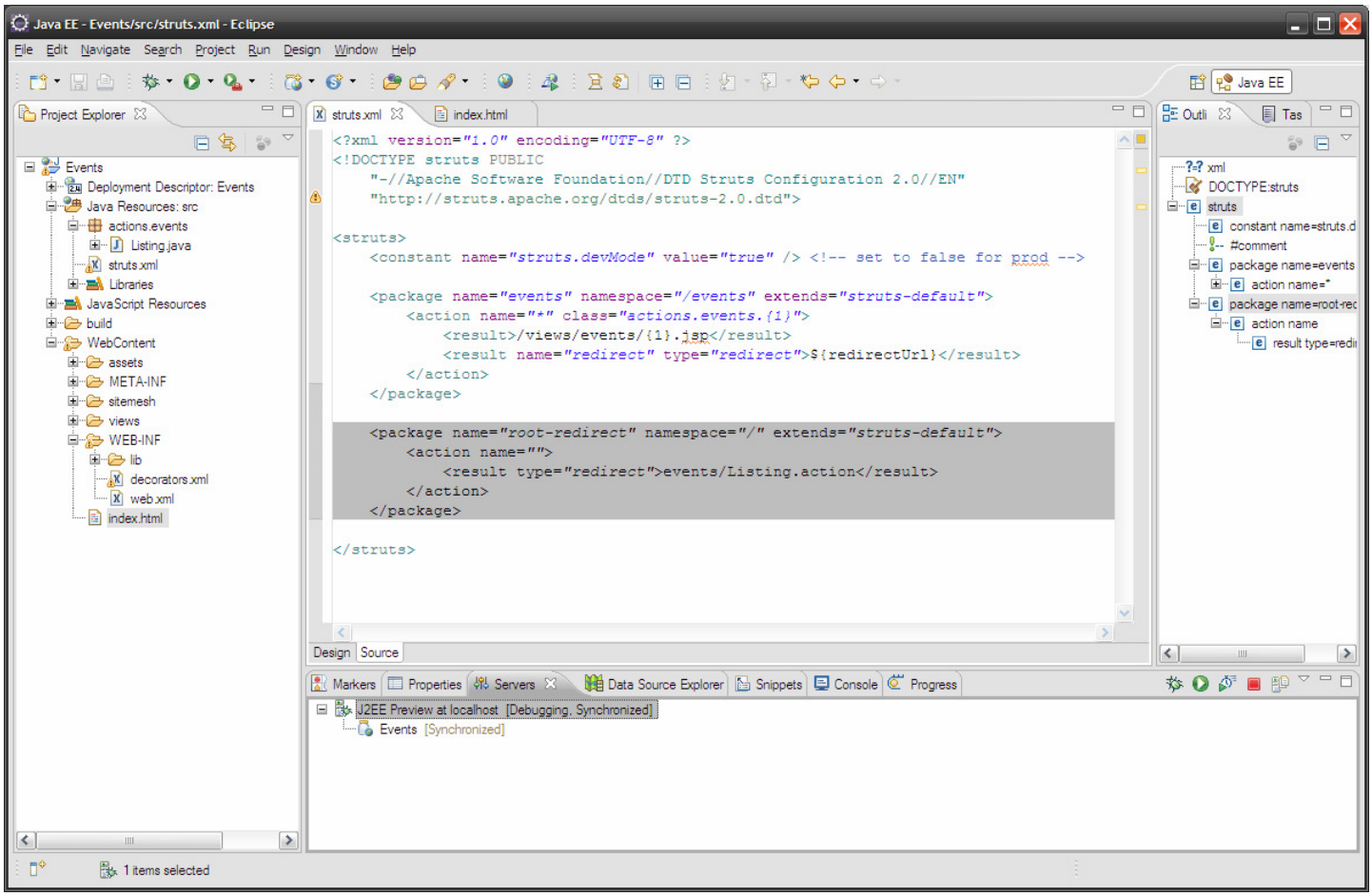


Once it has started OK, browse to: http://localhost:8080/Events/events/Listing.action and cross your fingers:



Next I wish for the root URL to automatically redirect the user to this struts action. To do so, edit the src\struts.xml file and add the following package underneath the other package:

```
<package name="root-redirect" namespace="/" extends="struts-default">
  <action name="">
    <result type="redirect">events/Listing.action</result>
  </action>
</package>
```




Then delete the WebContent\index.html file, because we don't need it now.
Restart the server and browse to http://localhost:8080/Events and it should automatically redirect you to the events listing. This will make things easier for your users, however you may want to change this later to show a specific action instead of redirecting, if neat URL's are important.
And thus you should have Struts2 working.

# Logging

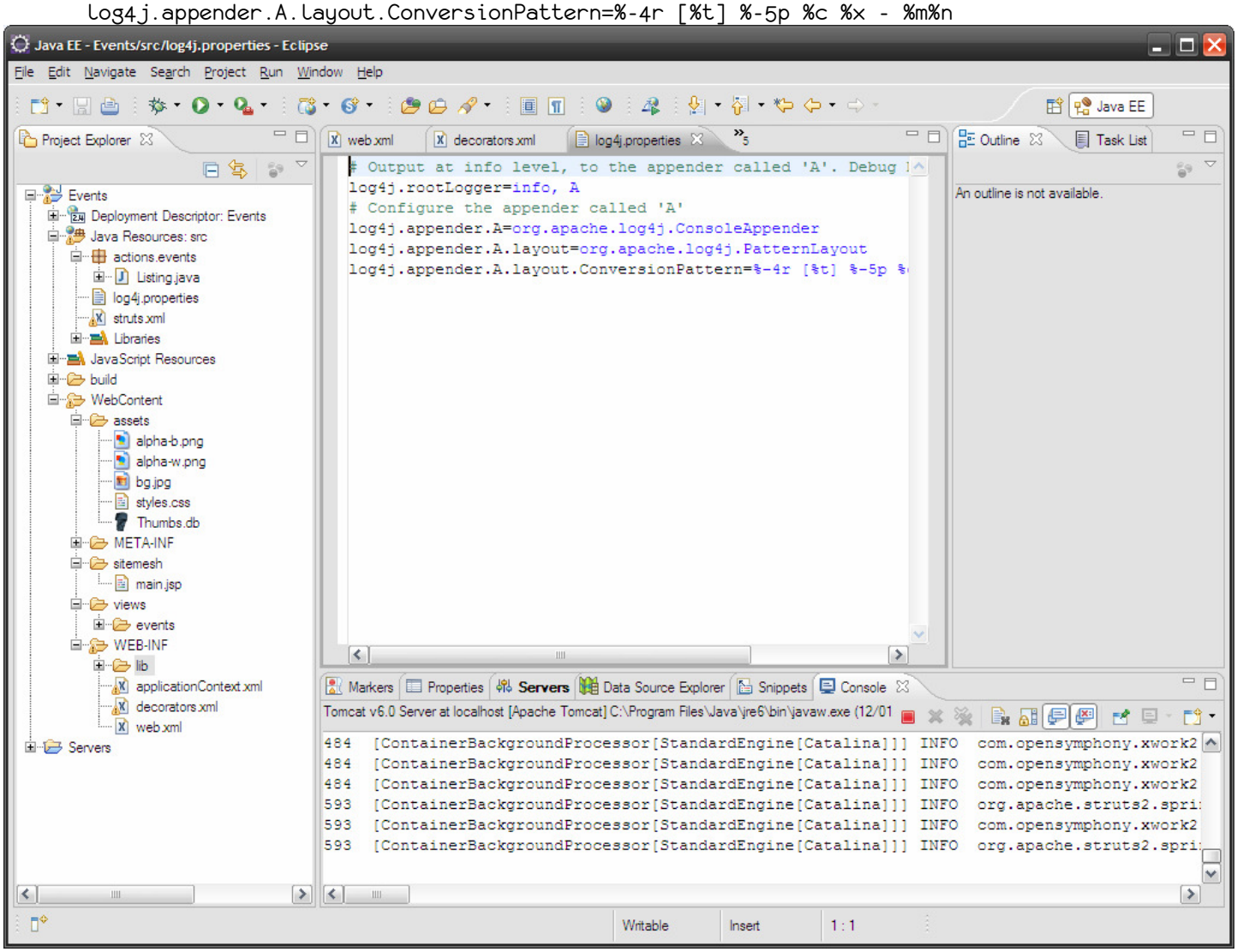
Now we'll configure some logging libraries that are required later on by spring and hibernate. These logging frameworks are Slf4J, Log4j, and Apache Commons Logging.

## *Log4j*

Firstly we'll install log4j. Log4j is the most popular java logging library, which we'll be using.
Download version 1.2 (1.3 is dead and 2.0 is beta) from here:
http://logging.apache.org/log4j/1.2/download.html
I downloaded the apache-log4j-1.2.15.zip. From that zip file, copy log4j-1.2.15.jar into the lib folder of your application.
Then create 'log4j.properties' in your src folder (Ctrl+N > General > File > Next > Parent folder: Events/src > File name: log4j.properties > Finish):

**src\log4j.properties**

```
# Output at info level, to the appender called 'A'
log4j.rootLogger=info, A
# Configure the appender called 'A'
log4j.appender.A=org.apache.log4j.ConsoleAppender
log4j.appender.A.layout=org.apache.log4j.PatternLayout
log4j.appender.A.layout.ConversionPattern=%-4r [%t] %-5p %c %x - %m%n
```




This is a very basic log4j configuration. It sends all logging to the console, only showing messages of 'info' severity or higher.

## SLF4J

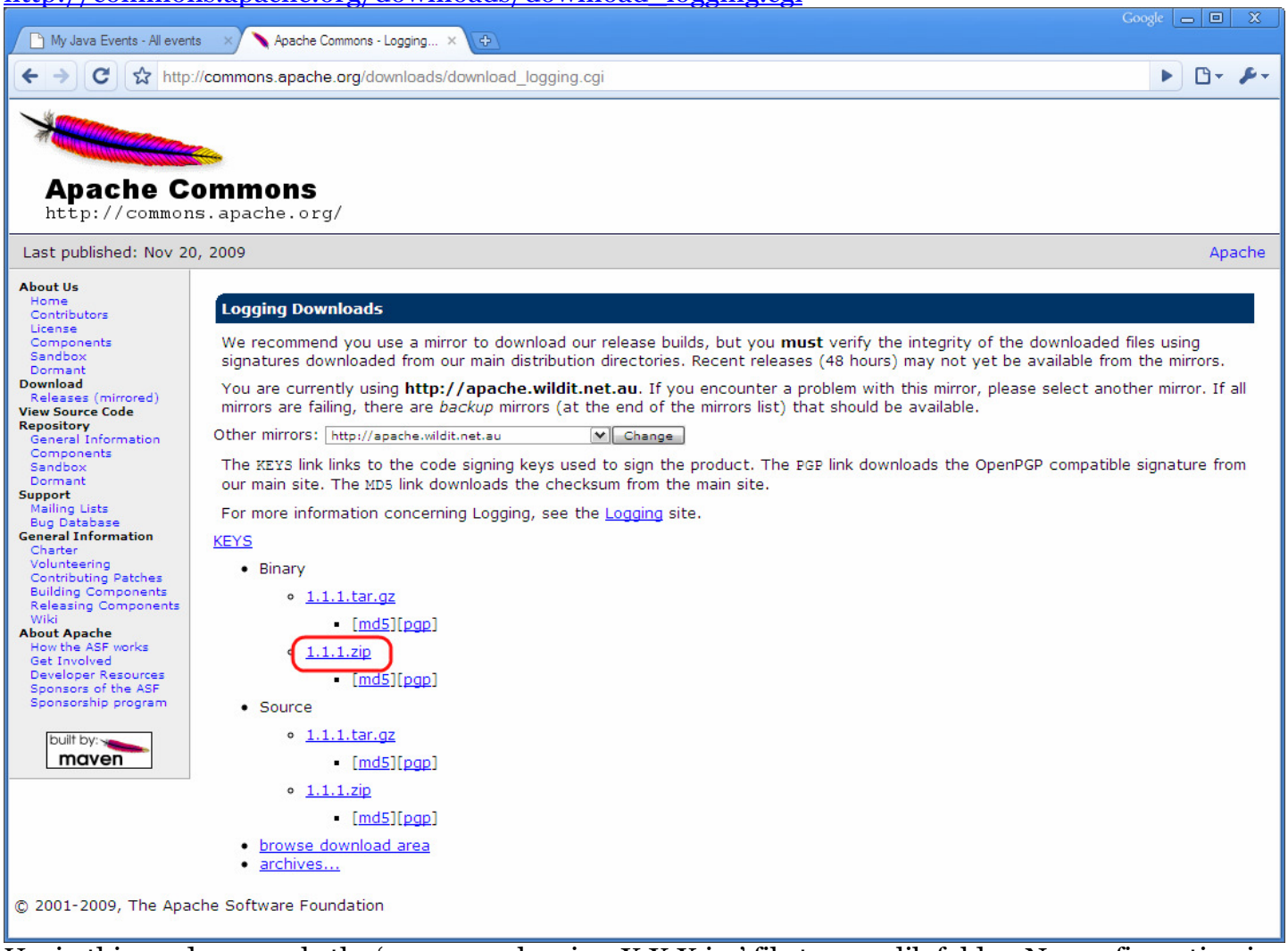
Next, we'll setup SLF4J. SLF4J is the 'Simple Logging Façade for Java', which is used by Hibernate to output to whichever logger library you're using (log4j in our case). This is used because the Hibernate team did not want to limit you to only using Log4j, and SLF4J can be configured to output to pretty much any logger you decide to use. Having said that, pretty much everyone just uses Log4j.
Download the latest zip version of SLF4J from here:
http://www.slf4j.org/download.html


I downloaded slf4j-1.5.10.zip. Unzip it and then copy only the following two files to your lib folder:
slf4j-api-1.5.10.jar (this is the core of slf4j)
slf4j-log4j12-1.5.10.jar (this makes slf4j output to log4j)

There is no configuration file for SLF4J. Just the presence of the slf4j-log4j12-*.jar file is enough for it to know it has to output to Log4j.

## *Apache Commons Logging*

And finally we'll setup Apache Commons Logging. This is just like SLF4J in that it is used by Spring to output to whichever logger library you're using (log4j in our case).
Download the latest binary zip version from here:
http://commons.apache.org/downloads/download_logging.cgi




Unzip this, and copy only the 'commons-logging-X.X.X.jar' file to your lib folder. No configuration is needed, it'll automatically detect that we're using log4j and output to it.

## Creating the database

Next we'll get started with accessing the database using Hibernate. But firstly we need to create a database with some sample data. I'm using SQL server here, but you can use any database engine. You'll need to create a new database, and create a new user with read/write access to that database. Once you've created your database, the script to create the tables is included with this tutorial in 'Create Database.sql.txt'. Basically it makes 3 tables and puts some sample data in them:

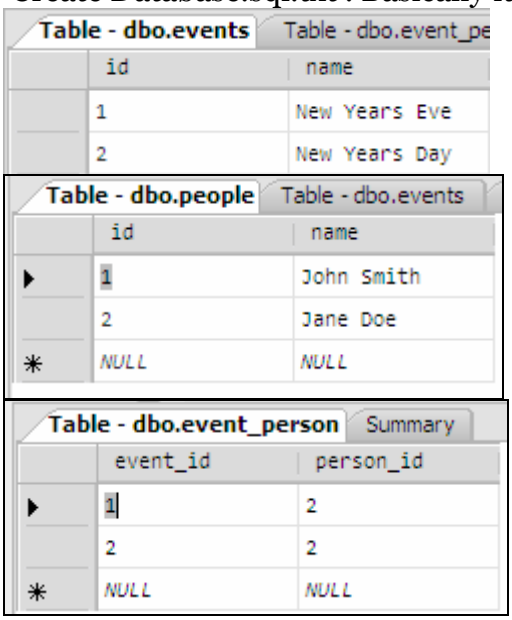


Table - dbo.events | Table - dbo.event_pe

| id | name |
|---|---|
| 1 | New Years Eve |
| 2 | New Years Day |

Table - dbo.people | Table - dbo.events

| id | name |
|---|---|
| 1 | John Smith |
| 2 | Jane Doe |
| NULL | NULL |

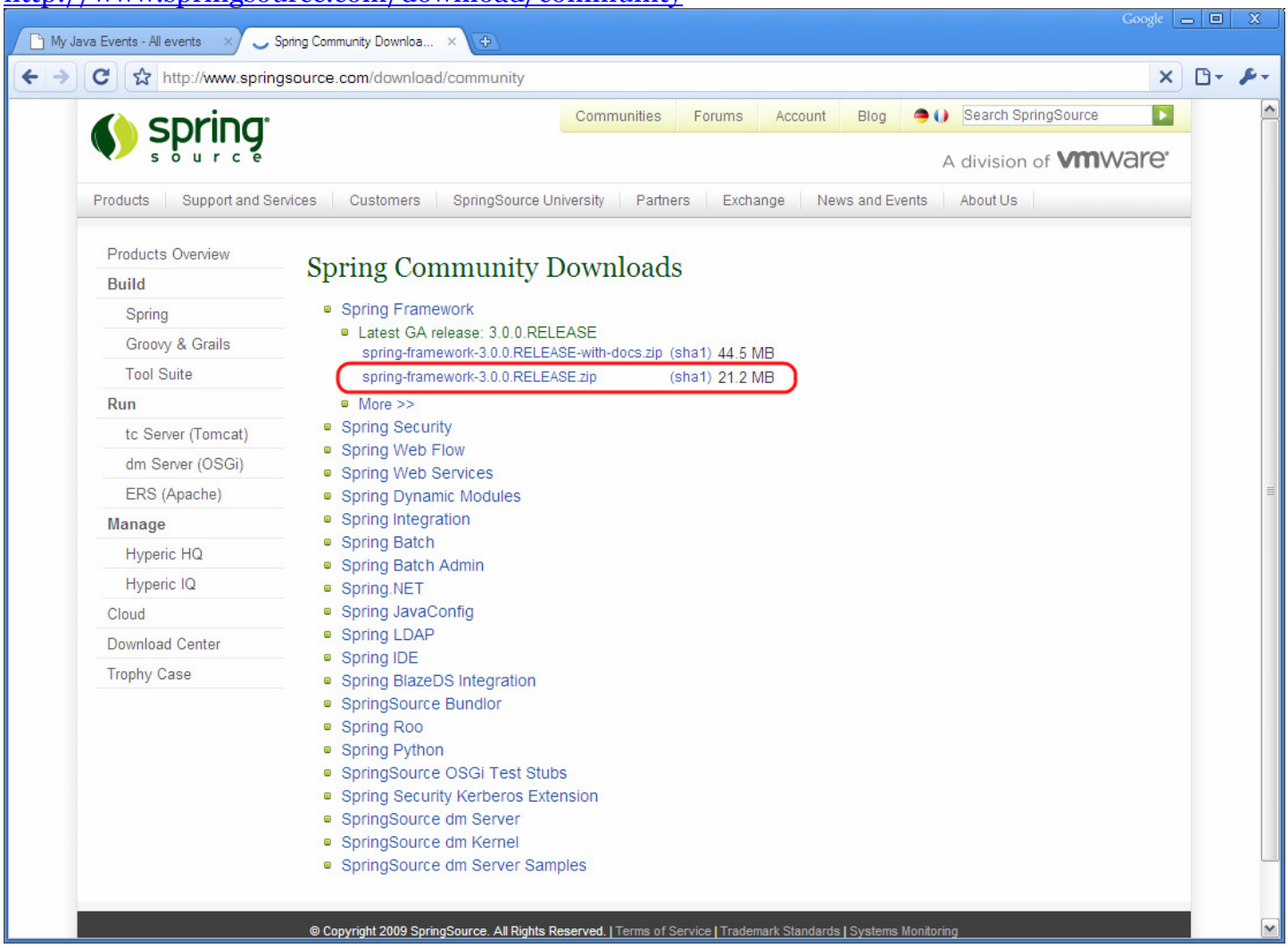
Table - dbo.event_person | Summary

| event_id | person_id |
|---|---|
| 1 | 2 |
| 2 | 2 |
| NULL | NULL |

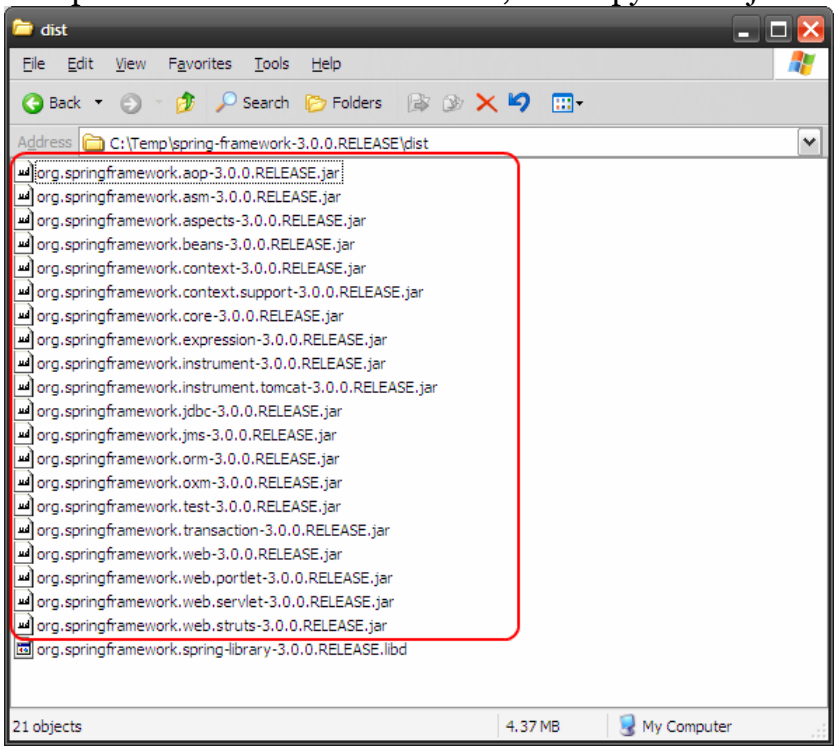
If you're using a different database, the script may work with some modifications, but I'll have to leave it up to you.

# Spring

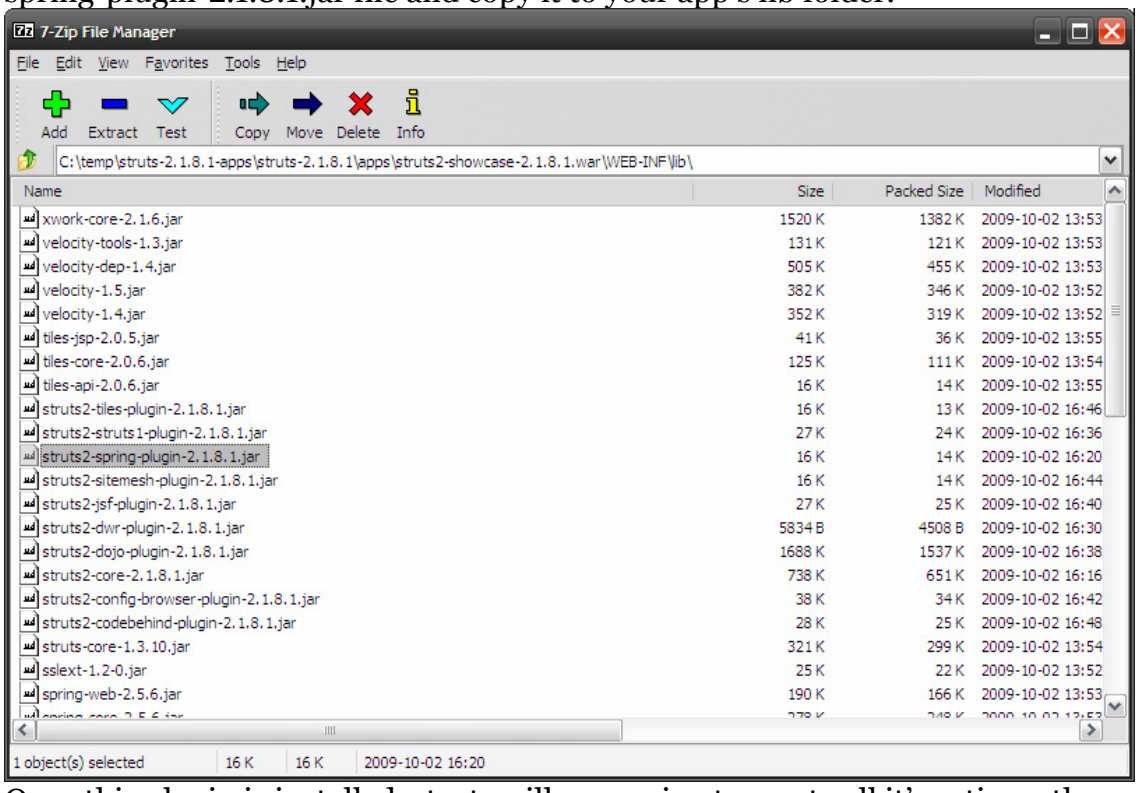
Spring is a popular dependency injection framework for java. You may want to read up on the basics here:
http://en.wikipedia.org/wiki/Dependency_injection
Basically we're going to use it as a 'glue' between Struts and Hibernate. It'll handle the lifecycle of our Hibernate session factory (there should be one of these for the entire application), our Hibernate sessions (one of these per HTTP request), and passing these to our Struts actions.
Go to the spring site and download the latest GA release:
http://www.springsource.com/download/community



Unzip it and look in the 'dist' folder, and copy all the jar's into your application's lib folder:




Next we need the struts2 spring plugin. Go into the struts-2.X.X.X-apps.zip file that you downloaded earlier, and unzip the struts2-showcase-2.X.X.x.war file. Inside it, grab the WEB-INF\lib\struts2-spring-plugin-2.1.8.1.jar file and copy it to your app's lib folder.

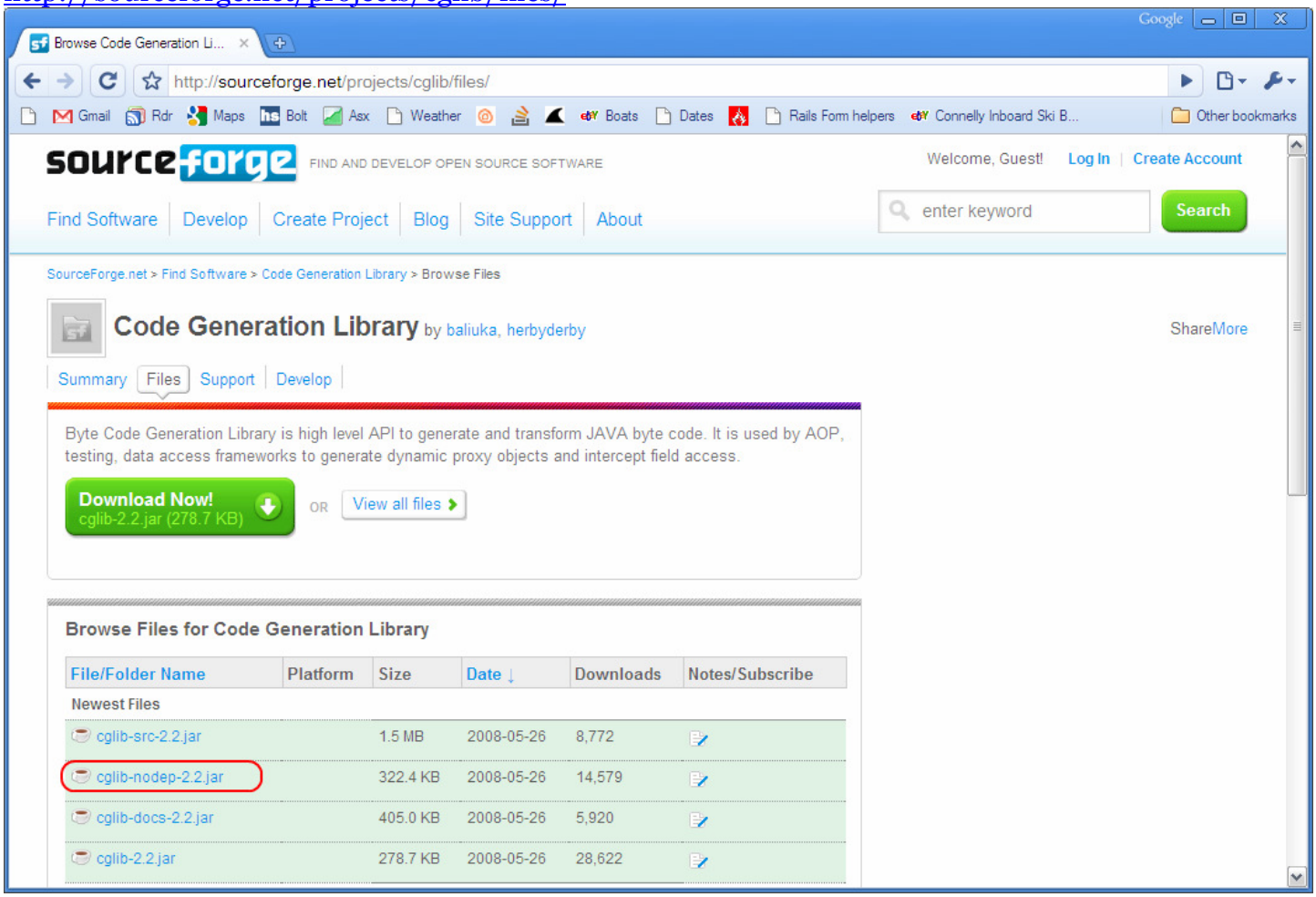



Once this plugin is installed, struts will use spring to create all it's actions, thus giving spring the opportunity to 'inject' the 'dependencies' into those actions - the dependencies in our case are the Hibernate sessions.

A couple of small dependencies are required for Spring's transaction manager and annotations. If 'transaction manager' and 'annotations' didn't make much sense, you'll see later when we configure hibernate and create our business services layer. For now, you need to download the following:

AOP Alliance
http://sourceforge.net/projects/aopalliance/files/aopalliance/1.0/aopalliance.zip/download
Once you've downloaded aopalliance.zip, unzip it and copy aopalliance.jar to your lib folder.

CGLIB2:
http://sourceforge.net/projects/cglib/files/

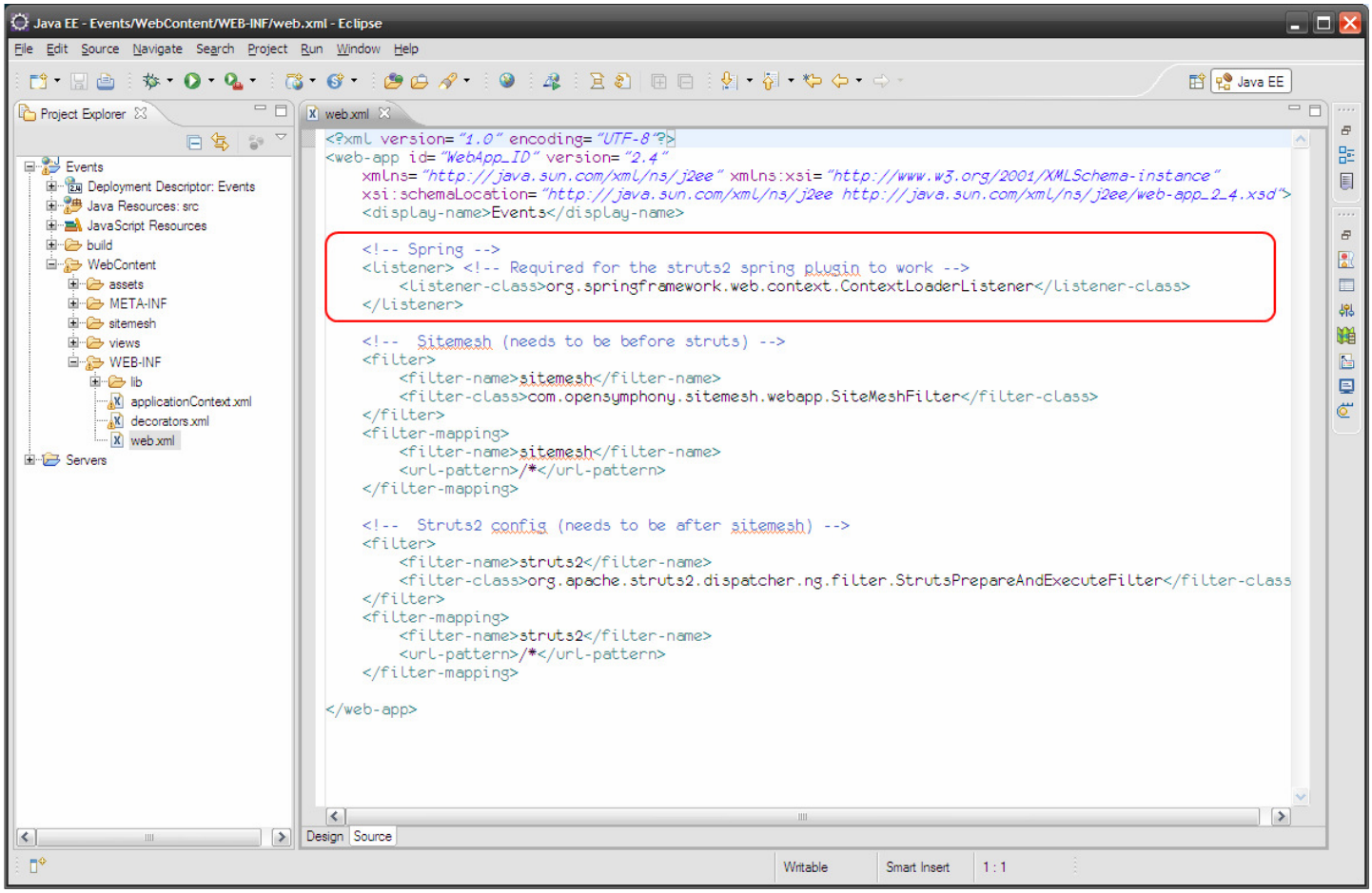



Download the latest cglib-nodep-*.jar version (NODEP is important)
Here's the version I downloaded:
http://sourceforge.net/projects/cglib/files/cglib2/2.2/cglib-nodep-2.2.jar/download
Once you've downloaded it, copy the jar to your project's lib folder.

Now we need to configure spring. Add the following into your web.xml above the sitemesh filter:

```
<!-- Spring -->
<listener> <!-- Required for the struts2 spring plugin to work -->
  <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
```

The above lines plug spring into the application and http-request lifecycle, so it can create (and destroy) everything at the right time.

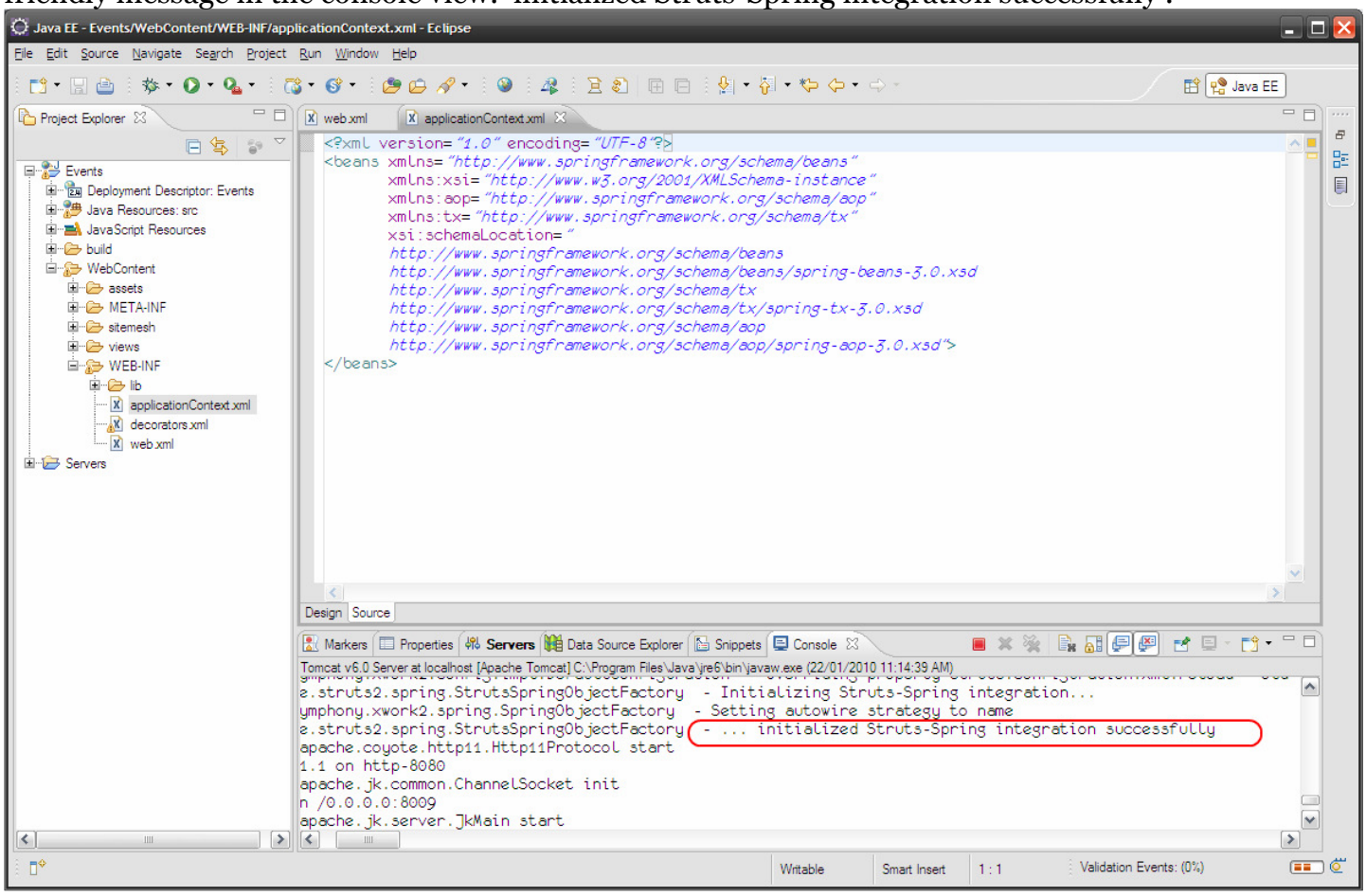


Next we need to create the spring config file. Create a file 'applicationContext.xml' in the WebContent\WEB-INF folder. For now, the config file is to be bare because we haven't configured Hibernate yet:

**WebContent\WEB-INF\applicationContext.xml**

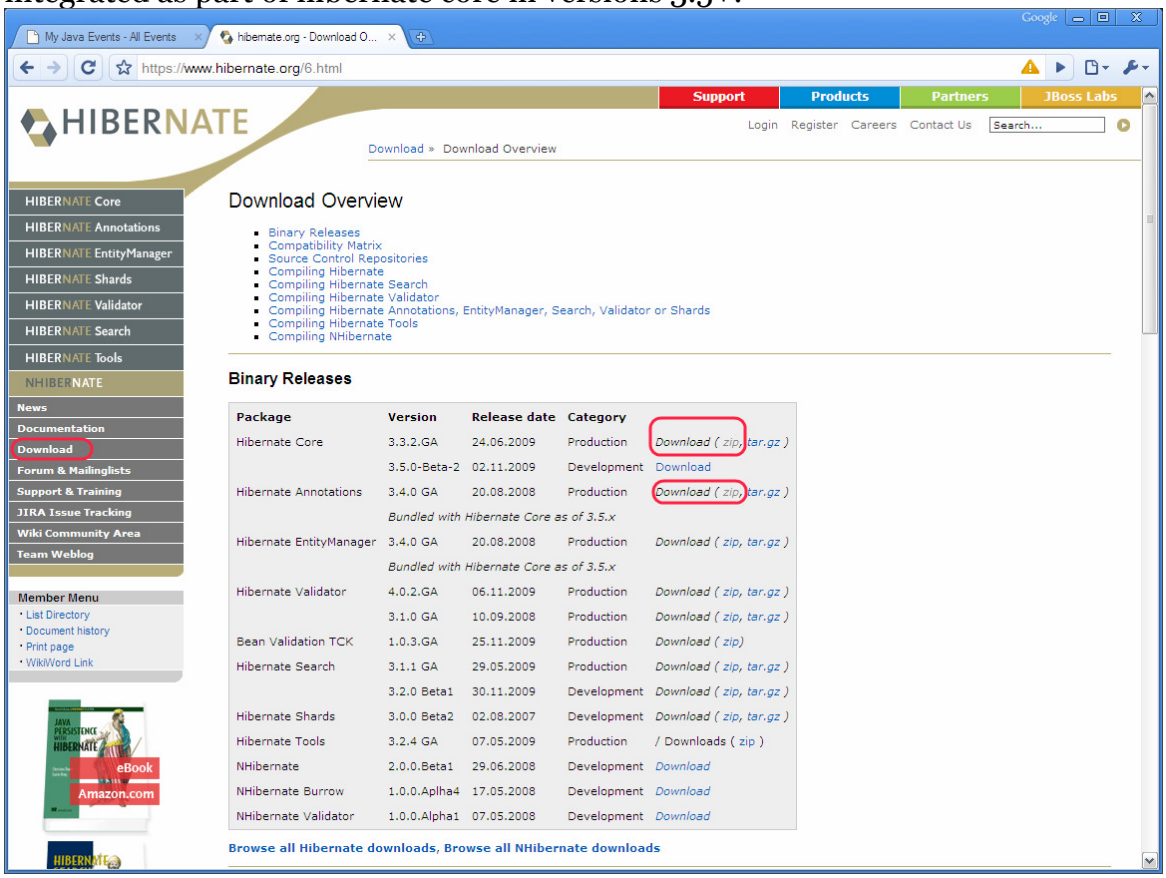
```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
       http://www.springframework.org/schema/tx
       http://www.springframework.org/schema/tx/spring-tx-3.0.xsd
       http://www.springframework.org/schema/aop
       http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">
</beans>
```

Save everything, clear the console and restart the server. When it's restarted, you'll hopefully see this friendly message in the console view: 'initialized Struts-Spring integration successfully':



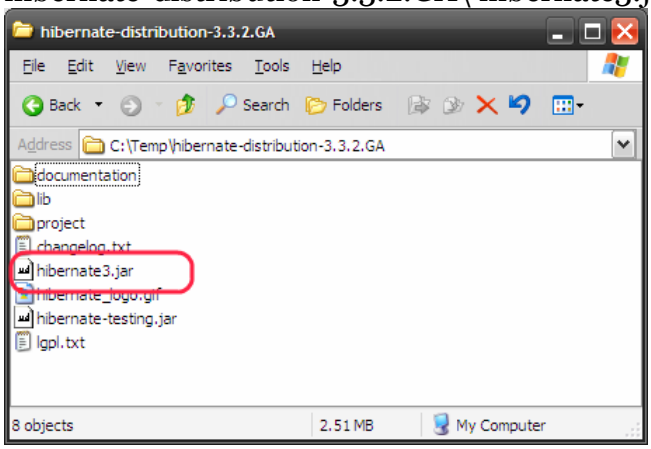
# Hibernate

This is where we start working on connecting to the database from Java. I'm using Hibernate for this, because that seems to be the most common way to do this these days. Firstly you need to download it from the hibernate website at www.hibernate.org.
Click on the 'download' link. We'll need both Hibernate Core and Hibernate Annotations. If by the time you read this 3.5 has been released then you won't need annotations as it is going to be integrated as part of hibernate core in versions 3.5+.

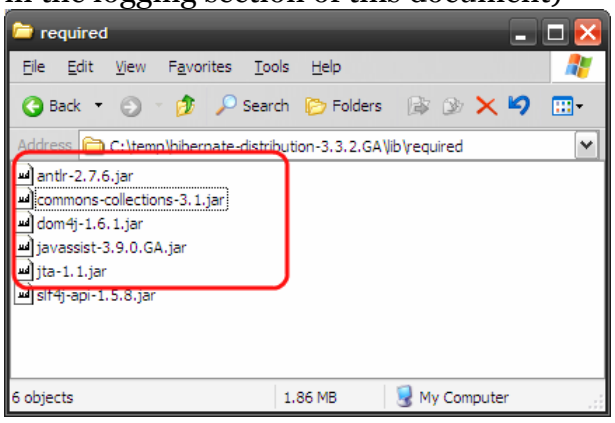



## *Jars*

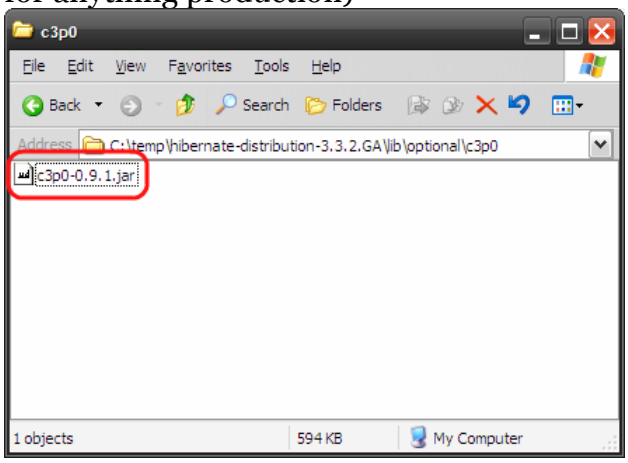
Unzip these files and copy the following jars into your project's WebContent\WEB-INF\lib folder: hibernate-distribution-3.3.2.GA\ hibernate3.jar



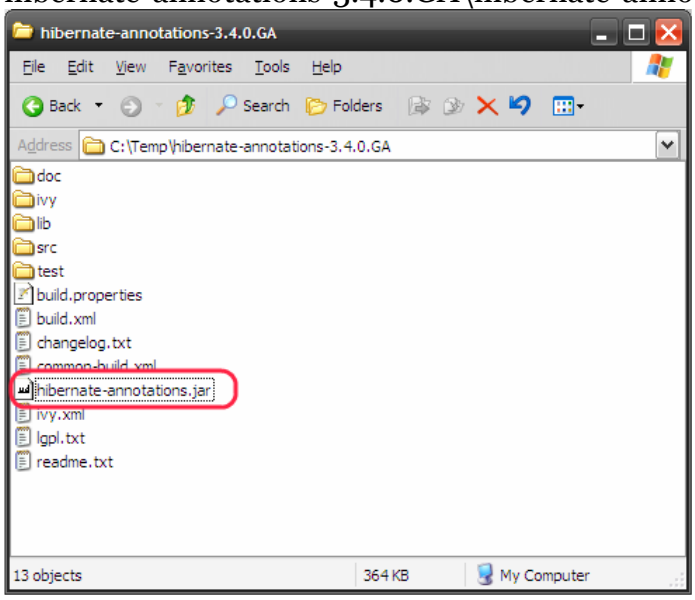
hibernate-distribution-3.3.2.GA\lib\required*.jar (except slf4j*.jar because we already grabbed that in the logging section of this document)



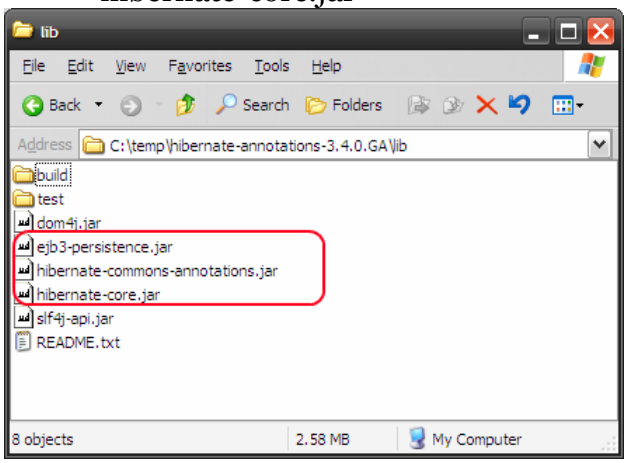

hibernate-distribution-3.3.2.GA\lib\optional\c3p0*.jar (its technically optional, but this is essential for anything production)




hibernate-annotations-3.4.0.GA\hibernate-annotations.jar



hibernate-annotations-3.4.0.GA\lib\
ejb3-persistence.jar
hibernate-commons-annotations.jar
hibernate-core.jar

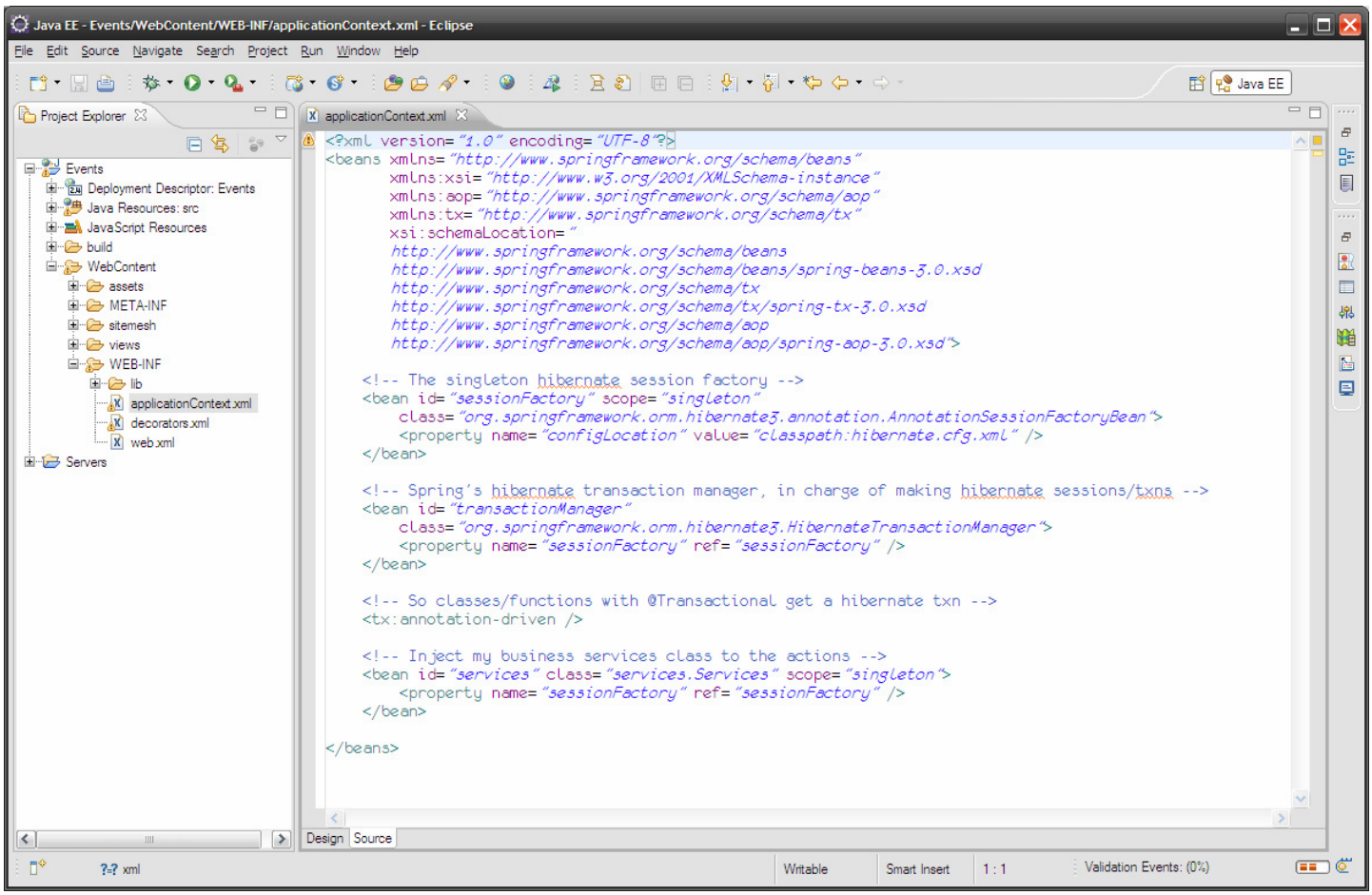



We have to install the Jtds jdbc driver because we're talking to sql server. Browse to:
http://sourceforge.net/projects/jtds/files/
Download the latest jtds-*-dist.zip (jtds-1.2.5-dist.zip in my case), extract jtds-*.jar and copy it to your lib folder.

## Configuration

We now need to configure spring to initialise our hibernate objects. Edit the WebContent/WEB-INF/applicationContext.xml file and add the following before the </beans> line:

```
<!-- The singleton hibernate session factory -->
<bean id="sessionFactory" scope="singleton"
class="org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean">
  <property name="configLocation" value="classpath:hibernate.cfg.xml" />
</bean>

<!-- Spring's hibernate transaction manager -->
<bean id="transactionManager"
  class="org.springframework.orm.hibernate3.HibernateTransactionManager">
  <property name="sessionFactory" ref="sessionFactory" />
</bean>

<!-- So classes/functions with @Transactional get a hibernate txn -->
<tx:annotation-driven />

<!-- Inject my business services class to the actions -->
<bean id="services" class="services.Services" scope="singleton">
  <property name="sessionFactory" ref="sessionFactory" />
</bean>
```

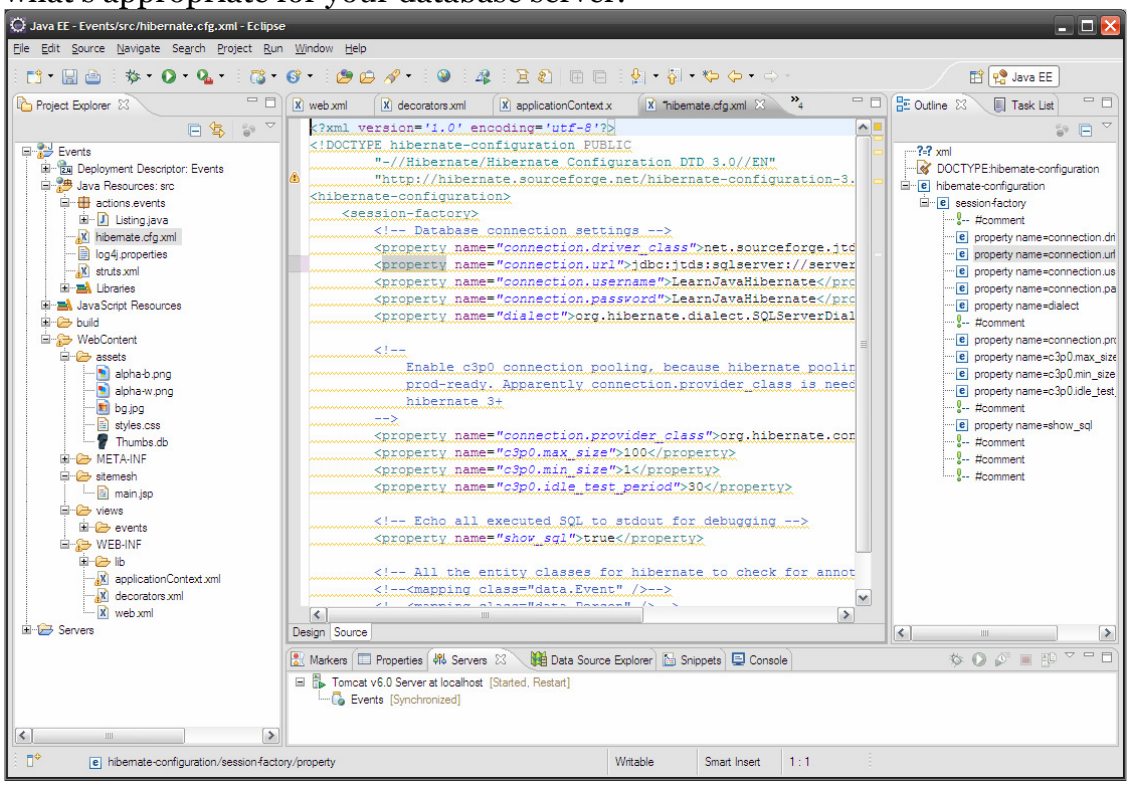


Now to configure hibernate. Create and edit the following hibernate configuration file:

**\src\hibernate.cfg.xml**

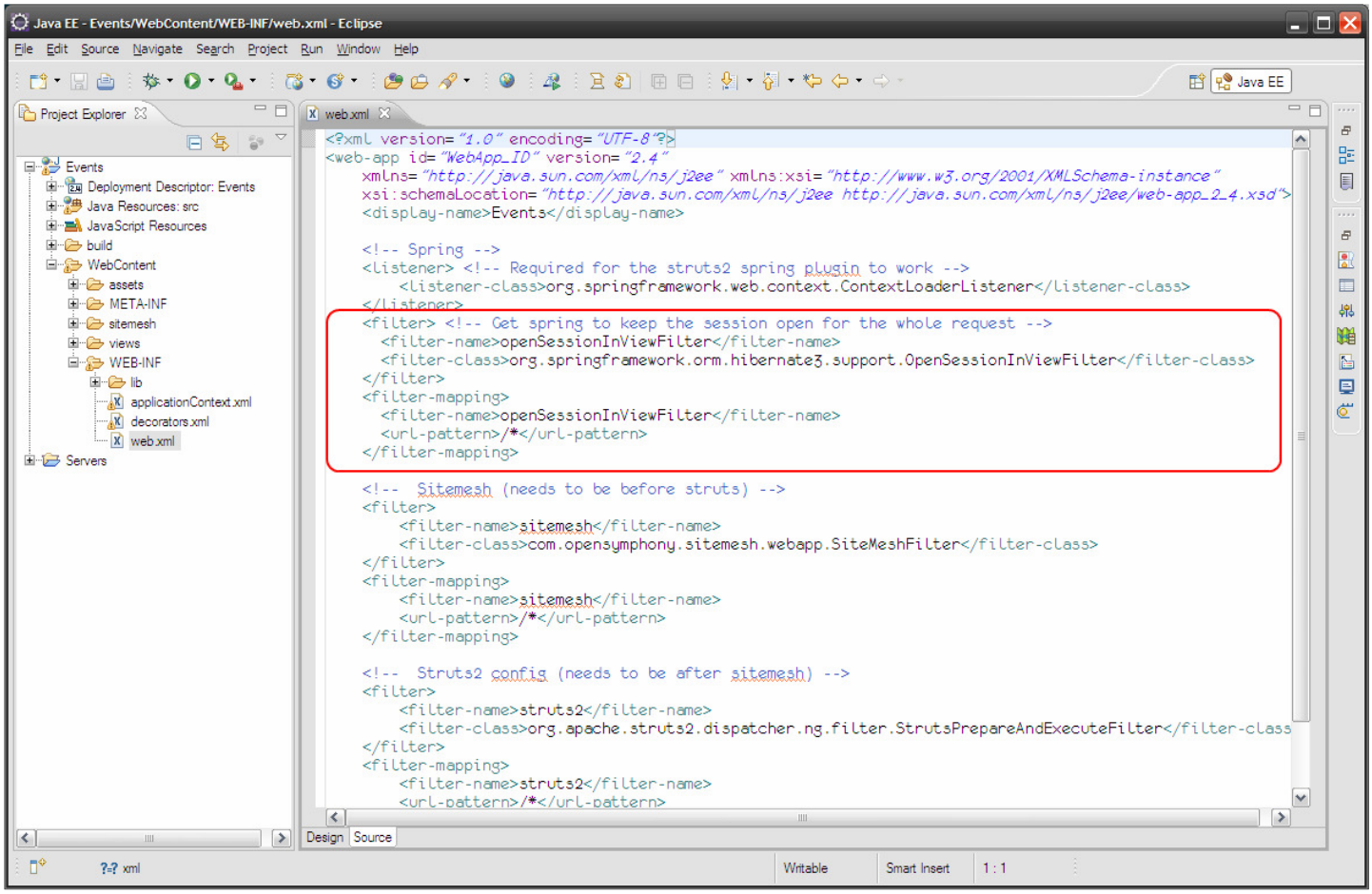
```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <!-- Database connection settings -->
    <property name="connection.driver_class">net.sourceforge.jtds.jdbc.Driver</property>
    <property name="connection.url">jdbc:jtds:sqlserver://myserver/mydatabase</property>
    <property name="connection.username">myusername</property>
    <property name="connection.password">mypassword</property>
    <property name="dialect">org.hibernate.dialect.SQLServerDialect</property>

    <!--
      Enable c3p0 connection pooling, because hibernate pooling is not
      prod-ready. Apparently connection.provider_class is needed in
      hibernate 3+
    -->
    <property name="connection.provider_class">org.hibernate.connection.C3P0ConnectionProvider</property>
    <property name="c3p0.max_size">100</property>
    <property name="c3p0.min_size">1</property>
    <property name="c3p0.idle_test_period">30</property>

    <!-- Echo all executed SQL to stdout for debugging -->
    <property name="show_sql">true</property>

    <!-- All the entity classes for hibernate to check for annotations here -->
    <mapping class="data.Event" />
    <mapping class="data.Person" />
  </session-factory>
</hibernate-configuration>
```

Make sure you change the bits in bold above (myserver,mydatabase,myusername,mypassword) to reflect your database settings. And if you're using a different database server eg mysql or oracle, you'll have to change the driver class and dialect. You'll need to check the hibernate documentation to see what's appropriate for your database server.



Next up we have to configure spring so that it'll keep the hibernate session open for the entirety of the web request lifecycle. To do this, we need to add a new filter to the web.xml file. Add the following lines to WebContent\WEB-INF\web.xml file just under the spring listener:

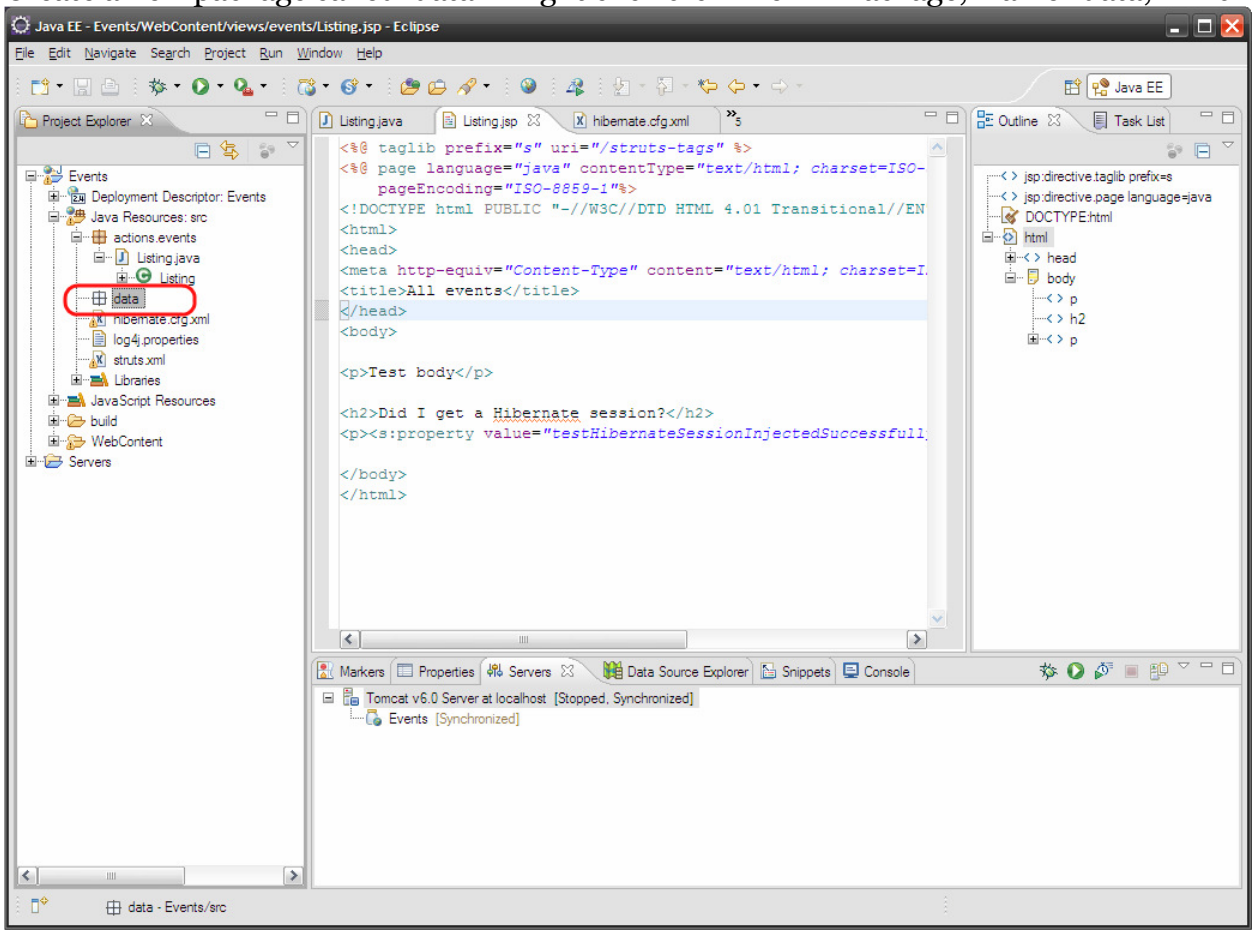
```
<filter> <!-- Get spring to keep the session open for the whole request -->
  <filter-name>openSessionInViewFilter</filter-name>
  <filter-class>org.springframework.orm.hibernate3.support.OpenSessionInViewFilter</filter-class>
</filter>
<filter-mapping>
  <filter-name>openSessionInViewFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```




The above filter ensures that the hibernate session is open when we need it in our views, for example when iterating through lazily-loaded lists returned by hibernate. Without it, these views would fail with an obscure OGNL error that can easily confuse.

## *Data objects layer*

We need to create the data object classes. Instances of these classes represent rows in the database. Create a new package called 'data' – right click src > New > Package; Name=data; Finish



Make two classes in this 'data' package (right click data > New > Class). Hibernate will use instances of these classes to represent rows in the people and events tables in the database.

**src\data\Person.java**

```
package data;

import java.util.Set;
import javax.persistence.*;

@Entity
@Table(name="people")
public class Person {
  @Id @GeneratedValue
  Long id;
  public Long getId() {return id;}
  public void setId(Long id) {this.id = id;}

  String name;
  public String getName() {return name;}
  public void setName(String name) {this.name = name;}

  @ManyToMany
  @JoinTable(
      name="event_person",
      joinColumns=@JoinColumn(name="person_id"),
      inverseJoinColumns=@JoinColumn(name="event_id")
  )
  Set<Event> events;
  public void setEvents(Set<Event> events) {this.events = events;}
  public Set<Event> getEvents() {return events;}
}
```

**src\data\Event.java**

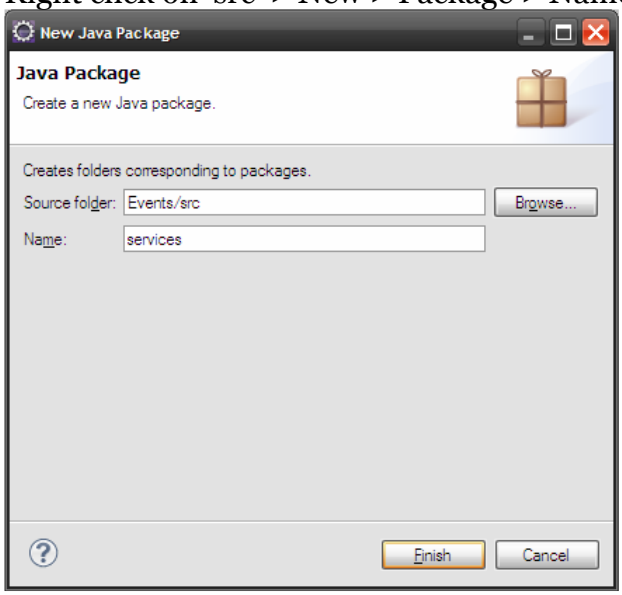
```
package data;

import java.util.Set;
import javax.persistence.*;

@Entity
@Table(name="events")
public class Event {
  @Id @GeneratedValue
  Long id;
  public Long getId() {return id;}
  private void setId(Long id) {this.id = id;}

  String name;
  public String getName() {return name;}
  public void setName(String value) {name = value;}

  @ManyToMany
  @JoinTable(
      name="event_person",
      joinColumns=@JoinColumn(name="event_id"),
      inverseJoinColumns=@JoinColumn(name="person_id")
  )
  Set<Person> people;
  public void setPeople(Set<Person> people) {this.people = people;}
  public Set<Person> getPeople() {return people;}
}
```

The lines above that start with @ are the 'annotations' that are a new Java feature, and are used in this case to tell Hibernate the database table name and the many to many relationship details. See: http://docs.jboss.org/hibernate/stable/annotations/reference/en/html_single/

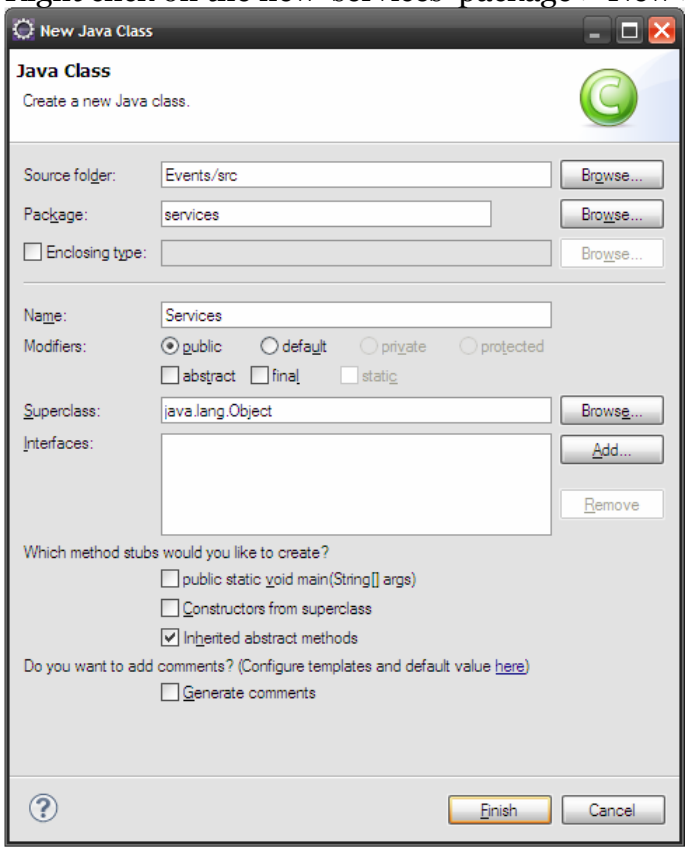
## *Business Services layer*

Now we need to create the business services layer. In our example, we'll just create a single class for this. This class is responsible for all database interactions, eg finding events, creating people, etc. This class is to be a 'singleton', ie there will only ever be one instance of it shared between all threads. Hence it has no instance variables, and has to use sessionFactory.getCurrentSession() whenever it wants to find the appropriate Hibernate session.

Right click on 'src' > New > Package > Name: services > Finish




Right click on the new 'services' package > New > Class > Name: Services > Finish



Edit the new Services class so it's just like this:

**\src\services\Services.java**

```
package services;

import org.springframework.transaction.annotation.Transactional;
import org.hibernate.SessionFactory;
import org.hibernate.Session;
import data.*;
import java.util.List;

// This class is the business services tier in the application.
// @Transactional is needed so that a Hibernate transaction is set up,
// otherwise updates won't have an affect
@Transactional
public class Services {
	// So Spring can inject the session factory
	SessionFactory sessionFactory;
	public void setSessionFactory(SessionFactory value) {
		sessionFactory = value;
	}
	// Shortcut for sessionFactory.getCurrentSession()
	public Session sess() {
		return sessionFactory.getCurrentSession();
	}
	public Event getEventById(long id) {
		return (Event) sess().load(Event.class, id);
	}
	public Person getPersonById(long id) {
		return (Person) sess().load(Person.class, id);
	}
	public void deleteEventById(long id) {
		sess().delete(getEventById(id));
	}
	public void deletePersonById(long id) {
		sess().delete(getPersonById(id));
	}
	public void createEvent(String name) {
		Event theEvent = new Event();
		theEvent.setName(name);
		sess().save(theEvent);
	}
	public void createPerson(String name) {
		Person p = new Person();
		p.setName(name);
		sess().save(p);
	}
	@SuppressWarnings("unchecked")
	public List<Event> getEvents() {
		return sess().createQuery("from Event").list();
	}
	@SuppressWarnings("unchecked")
	public List<Person> getPeople() {
		return sess().createQuery("from Person").list();
	}
	public void removePersonFromEvent(int personId, int eventId) {
		getEventById(eventId).getPeople().remove(getPersonById(personId));
	}
	public void addPersonToEvent(int personId, int eventId) {
		getEventById(eventId).getPeople().add(getPersonById(personId));
	}
}
```

## *Common Action code*

Now that we've got a business services layer, we want it to be accessible to all our actions. The way I'm going to do this is to make a common base class for all the actions. Create a new package called 'actions.base'. Within that package, create a class called 'BaseAction'. The code for this class is here:

### src\actions.base\BaseAction.java

```
package actions.base;

import services.Services;
import com.opensymphony.xwork2.ActionSupport;

public class BaseAction extends ActionSupport {
	// So that spring can inject the business singleton
	protected Services services;
	public void setServices(Services value) {
		services=value;
	}

	// For redirect results
	protected String redirectUrl;
	public String getRedirectUrl() {
		return redirectUrl;
	}
	public String redirect(String to) {
		redirectUrl = to;
		return "redirect";
	}
}
```

This class has a setter for the services class, so that Spring can inject the services singleton we created before. It also has convenience methods for when actions want to do a redirect.

## *Events Listing*

Now lets make our struts action show an events listing from the database. Make the following changes to your Listing action:

### src\actions.events\Listing.java

```
package actions.events;

import actions.base.BaseAction;
import data.*;
import java.util.List;

public class Listing extends BaseAction {

	public String execute() {
		events = services.getEvents();
		return "success";
	}

	List<Event> events;
	public List<Event> getEvents() { return events; }
}
```

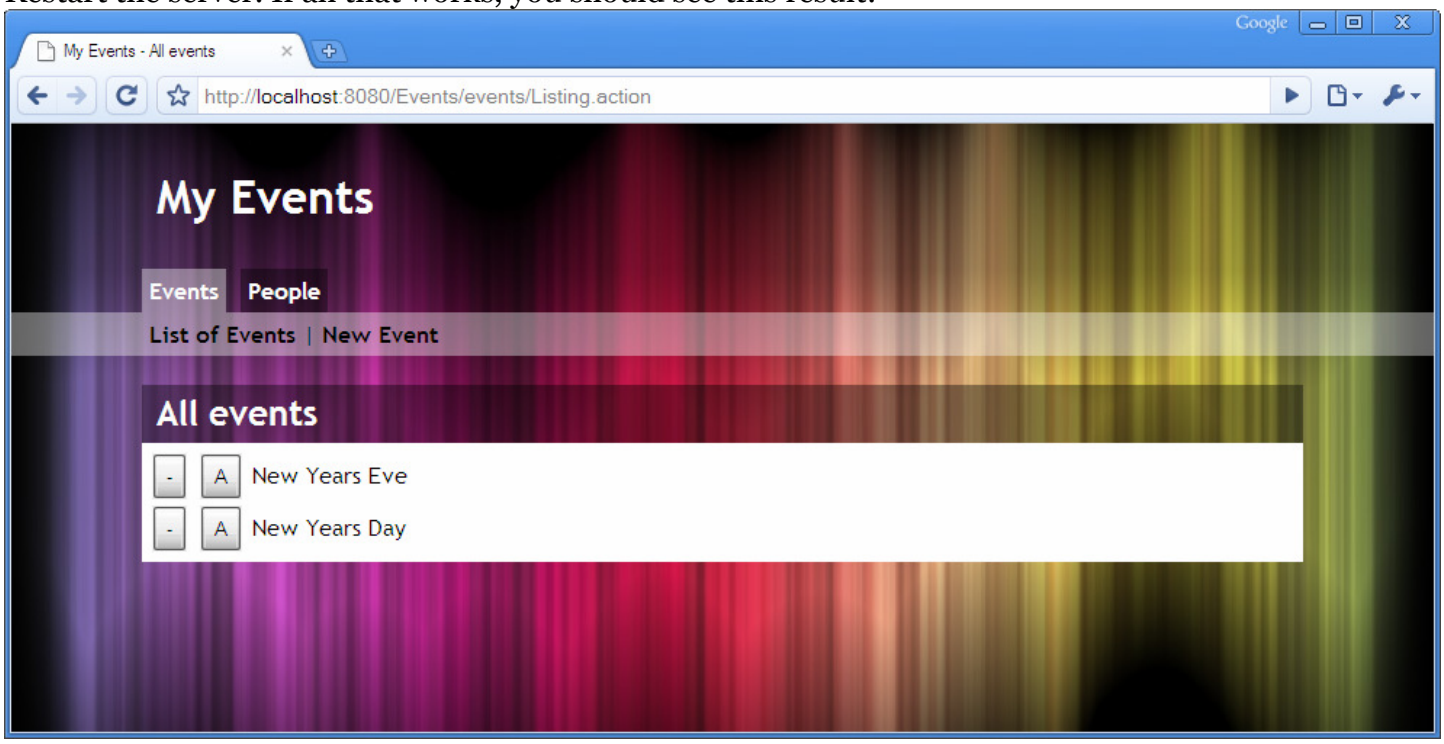
And modify the view as below:

### WebContent\views\events\Listing.jsp

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>All events</title>
</head>
<body>

<s:iterator value="events">
  <form action='Delete.action'>
    <input type='hidden' name=id value='<s:property value="id" />' />
    <input type='submit' value='-' title='Delete' />
  </form>
  <form action='Attendance.action'>
    <input type='hidden' name=id value='<s:property value="id" />' />
    <input type='submit' value='A' title='Attendance' />
  </form>
  <s:property value="name" />
  <br>
</s:iterator>

</body>
</html>
```

The interesting bit above is the s:iterator which loops through the event listing. Since it's value is "events", it calls the getEvents() function from the action, and loops through the elements of the returned array. For each iteration of the loop, the <s:property value="name" /> outputs the result of getName() from that event, and <s:property value="id" /> does the same with getId(). See: http://struts.apache.org/2.1.8.1/docs/tag-reference.html

Restart the server. If all that works, you should see this result:

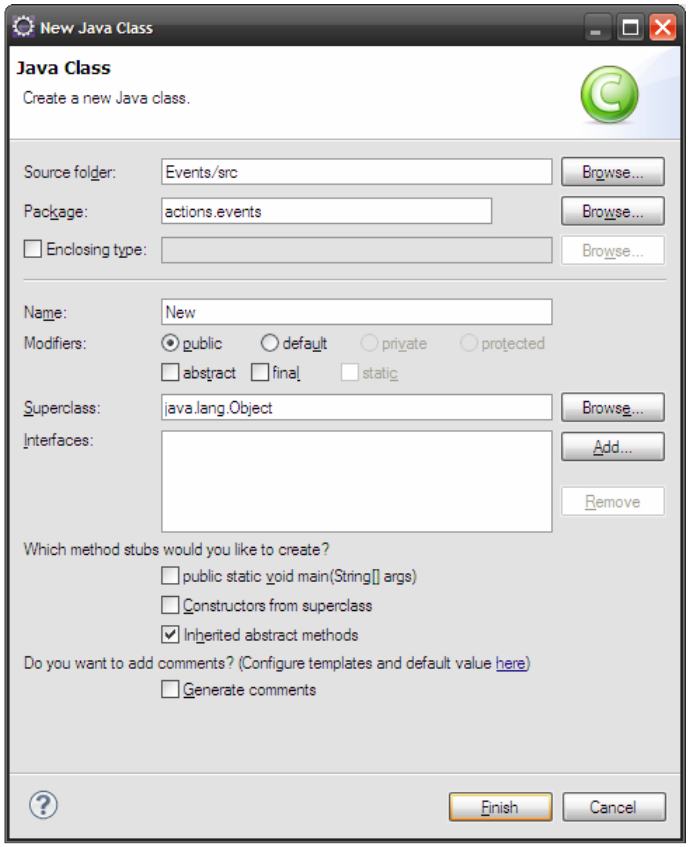



If you can see that, great! You've got all the various components (Struts, Hibernate, Spring, etc) all working together successfully. You might want to save your project now, and use it as a base for new applications instead of starting from scratch next time. From here on, we're just fleshing out the application with data entry abilities, but the fundamentals are done.

# Creating Data Entry forms

## *New Event*

We want to make a web form where the user can create a new event.
To create the action class, right click on the 'actions.events' package (src\actions.events) > New > Class. Enter 'New' as the class name and click 'Finish'. The code follows:

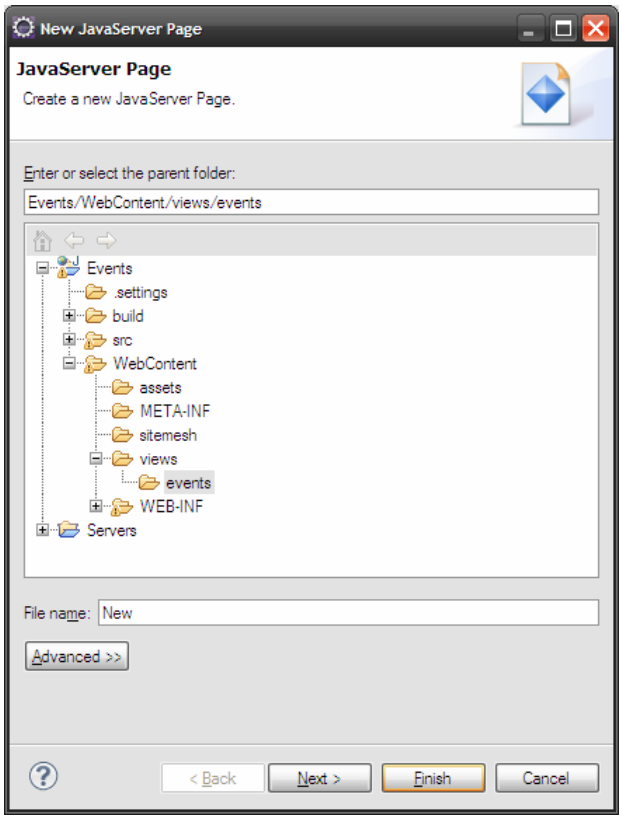



### src\actions.events\New.java

```
package actions.events;

import actions.base.BaseAction;

public class New extends BaseAction {

	public String execute() {
		if (name!=null && name.length()>0)
		{
			services.createEvent(name);
			return redirect("Listing.action");
		}
		return "success";
	}

    String name;
    public String getName() {return name;}
    public void setName(String value) {name = value;}
}
```

The execute method, if passed a name, will use the services class to create a new event with that name and redirect the user to the events listing.

Now lets create the matching view. Right click on Events\WebContent\views\events > New > JSP > File name: 'New' > Finish.



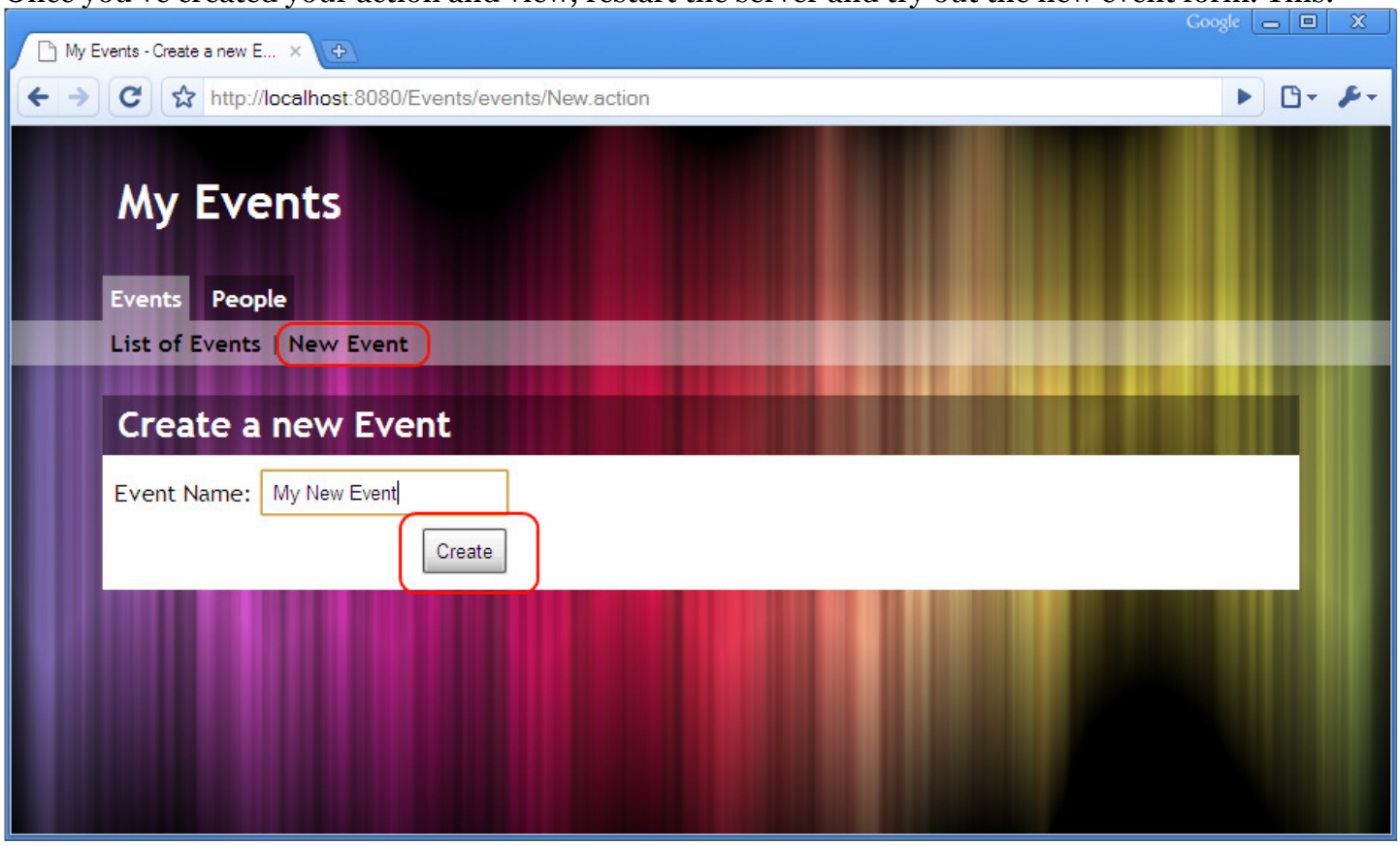

The view code is to be:

### WebContent\views\events\New.jsp

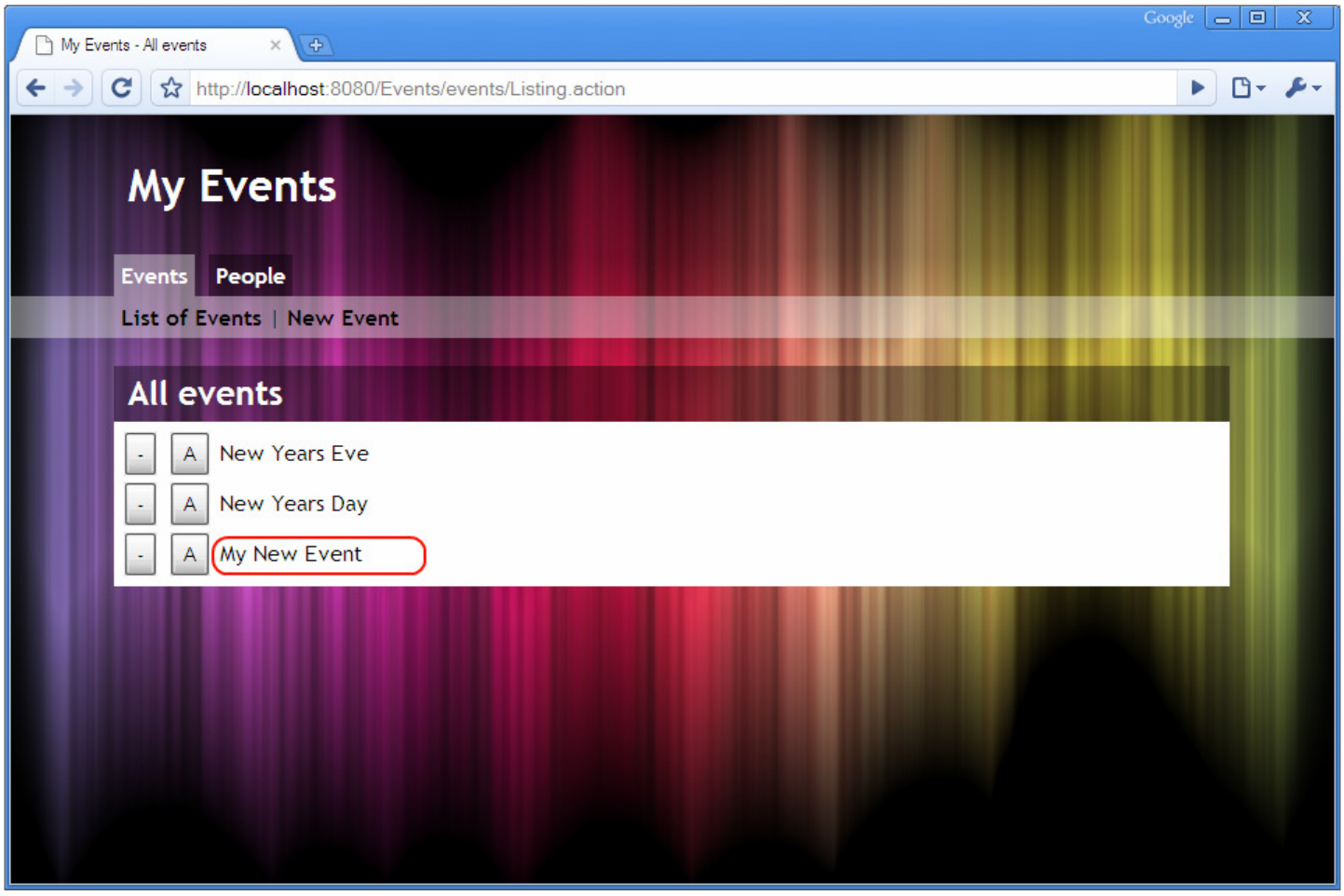
```
<%@ taglib prefix="s" uri="/struts-tags" %>
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Create a new Event</title>
</head>
<body>

<s:form>
  <s:textfield label="Event Name" name="name"/>
  <s:submit value="Create" />
</s:form>

</body>
</html>
```

The s:form tag creates an html form which will submit back to the same action. The s:textfield creates a textbox with the label 'event name', automatically pre-filling in the value if it can be grabbed from the action's getName() getter. Keep in mind that when the user submits the form, Struts will automatically use the setName() setter to set the name variable, and so if not for the redirect, the name they entered would be displayed again in the textbox. Again, these tags are all documented on the struts site:
http://struts.apache.org/2.1.8.1/docs/tag-reference.html

Once you've created your action and view, restart the server and try out the new event form. This:




Should result in this:



## *Delete event form*

Next up, we want a 'delete event' form so that when you click on the '-' button on the events list, it allows you to delete a form.
Create a new class called 'Delete' in the actions.events package. Here's the code:

### src\actions.events\Delete.java

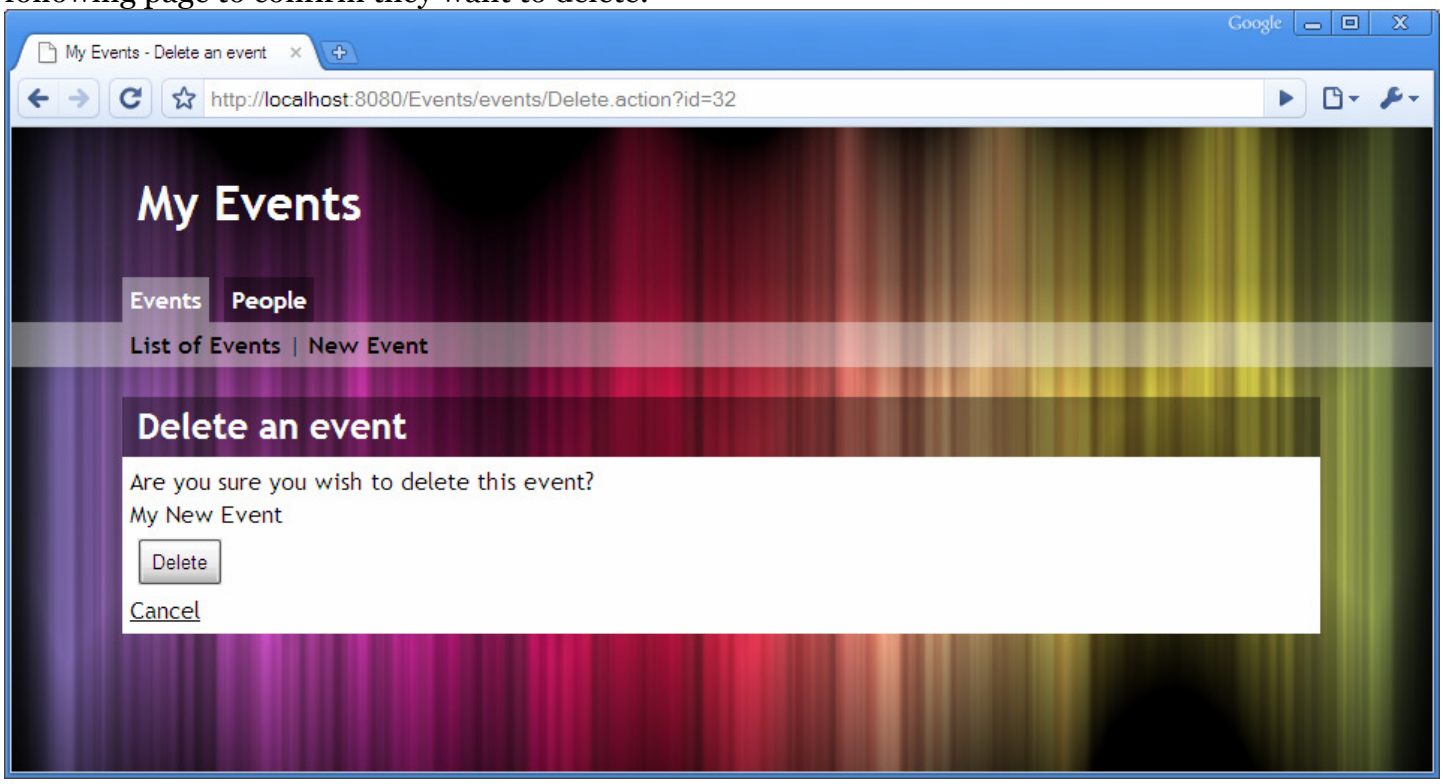
```
package actions.events;

import data.Event;
import actions.base.BaseAction;

public class Delete extends BaseAction {

  public String execute() {
    if (isPostBack) {
      services.deleteEventById(id);
      return redirect("Listing.action");
    }
    return "success";
  }

  int id;
  public void setId(int value) {id = value;}
  public int getId() {return id; }

  boolean isPostBack;
  public void setIsPostBack(boolean value) {isPostBack = value;}

  Event getEvent() {return services.getEventById(id); }
  public String getEventName() {return getEvent().getName();}
}
```

Create a corresponding view:

### WebContent\views\events\Delete.jsp

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Delete an event</title>
</head>
<body>

<s:form>
  Are you sure you wish to delete this event?<br />
  <s:property value="eventName" />
  <s:hidden name="id" />
  <input type="hidden" value="true" name="isPostBack" />
  <s:submit value="Delete" />
</s:form>

<s:a action="Listing">Cancel</s:a>

</body>
</html>
```

The idea is that when the user first clicks on the delete button in the event list, it takes them to the following page to confirm they want to delete:




Note that the id for the event is in the URL, as the query string '?id=32' above. Struts processes this just the same as it would process a submitted form variable, and sends it to the action via the setId() setter. When the view is processed, the <s:property value="eventName" /> calls the getEventName() on the action. This function in turn calls getEvent(), which passes the id to the services class to get the applicable event row. Once this is done, the event name is used to replace the s:property placeholder, so the user can see the event they are going to delete.
The <s:hidden name="id" /> is used so that when the user presses the delete button, the id is passed back as part of the form submission.
The hidden 'isPostBack' element is used so that the action can tell the difference between a form posting, and the original page display. Its value is sent to the setIsPostBack() setter function, which sets the isPostBack Boolean, which the execute() method uses to determine that this is a post back and they have confirmed the deletion.

## *Event Attendance*

Now let's create a screen so that users can update who is attending each event, when they click on the 'A' button next to each event in the events list screen. Create a new action class in the actions.events package called 'Attendance', here is the code:

### src\actions.events\Attendance.java

```
package actions.events;

import data.*;
import actions.base.BaseAction;
import java.util.Set;
import java.util.List;
import java.util.ArrayList;

public class Attendance extends BaseAction {

  public String execute() {
    return "success";
  }

  int id;
  public void setId(int value) {id = value;}
  public int getId() {return id;  }

  Event getEvent() {return services.getEventById(id);  }

  public String getEventName() { return getEvent().getName(); }

  public Set<Person> getAttendees() { return getEvent().getPeople(); }

  // Return a list of people not attending
  public List<Person> getNonAttendees() {
    List<Person> nonAttendees = new ArrayList<Person>();
    for(Person person : services.getPeople()) {
      if (!getAttendees().contains(person))
       nonAttendees.add(person);
    }
    return nonAttendees;
    }
}
```

Above, there are getters and setters for the id, which is used for the event id. The getEvent function uses the business services to get the event object from the database, using the id that was saved by the id setter. The getEventName and getAttendees merely return fields from this event object. The getNonAttendees makes a list every person from the database who isn't an attendee, so that the view can present them as options to add their attendance to the event.

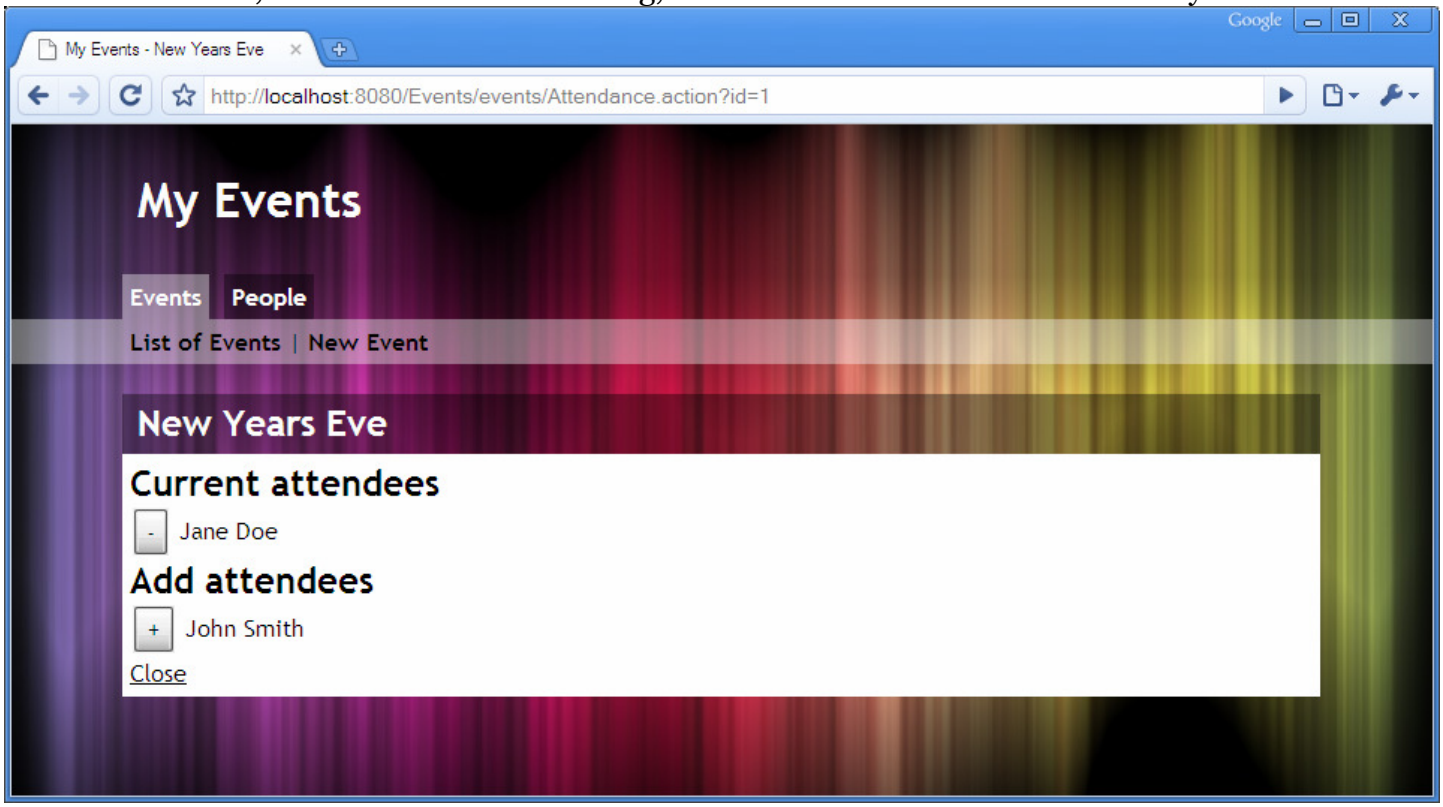
Next create a view to match:

**WebContent\views\events\Attendance.jsp**

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title><s:property value="eventName" /></title>
</head>
<body>

<h2>Current attendees</h2>

<s:iterator value="attendees">
  <form method='post' action='AttendanceRemove.action'>
    <input type='hidden' name='personId' value='<s:property value="id" />' />
    <input type='hidden' name='eventId'  value='<s:property value="[1].id" />' />
    <input type='submit' value='-' title='Remove' />
  </form>

  <s:property value="name" />
  <br>
</s:iterator>

<s:if test="attendees.isEmpty()">
  <em>There are no attendees</em><br/>
</s:if>

<h2>Add attendees</h2>

<s:iterator value="nonAttendees">
  <form method='post' action='AttendanceAdd.action'>
    <input type='hidden' name='personId' value='<s:property value="id" />' />
    <input type='hidden' name='eventId'  value='<s:property value="[1].id" />' />
    <input type='submit' value='+' title='Add' />
  </form>

  <s:property value="name" />
  <br>
</s:iterator>

<s:if test="nonAttendees.isEmpty()">
  <em>There is no one left to add</em><br/>
</s:if>

<s:a action="Listing">Close</s:a>

</body>
</html>
```

This view loops through all the attendees and non-attendees, presenting them as lists to the user, with buttons next to each person to either add them or remove them as an attendee. These buttons have two hidden elements for the event id and person id. If you notice the s:property for the event id's has a strange [1], it is used to distinguish between the id of the s:iterator value (attendees / nonAttendees) and the id of the action. Basically, the [1] means to go up one level on the OGNL stack and grab the id value from there, thereby skipping the id from the iterator.
There are also the s:if sections of this view that are used to display a friendly message to the user if the lists were empty.
More details on OGNL can be read here:
http://struts.apache.org/2.0.14/docs/ognl-basics.html

Restart the server, browse to the events listing, and click on the 'A' button next to any of the events:




Currently if you click on the the '+' and '-' buttons, it won't work. We need to create the actions that'll get called when the user clicks on those buttons. Create the following classes in src\actions.events:

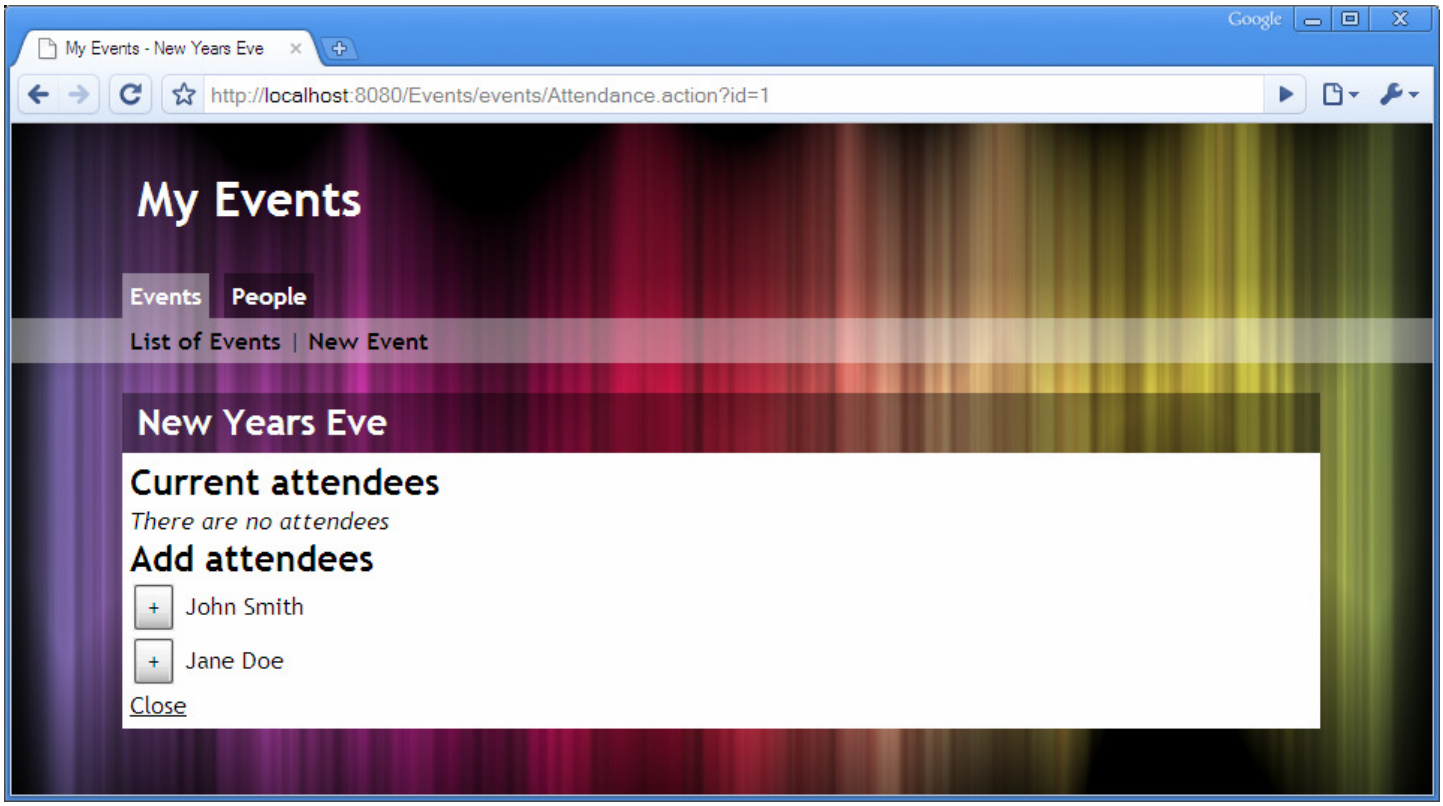
**src\actions.events\AttendanceAdd.java**

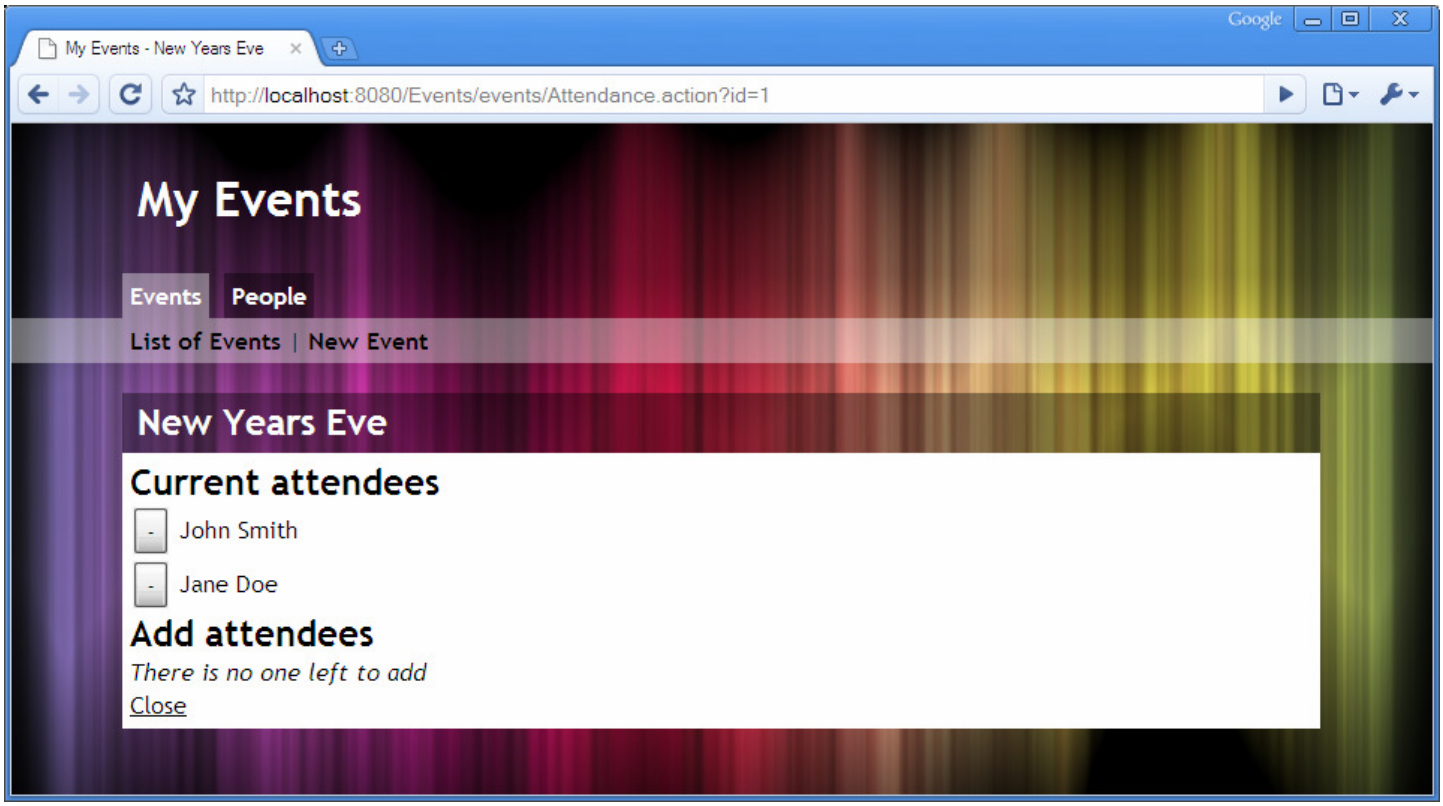
```
package actions.events;

import actions.base.BaseAction;

public class AttendanceAdd extends BaseAction {
  public String execute() {
    services.addPersonToEvent(personId, eventId);
    return redirect(String.format("Attendance.action?id=%d", eventId));
  }

  int eventId;
  public void setEventId(int value) {eventId = value;}
  int personId;
  public void setPersonId(int value) {personId = value;}
}
```

**src\actions.events\AttendanceRemove.java**

```
package actions.events;

import actions.base.BaseAction;

public class AttendanceRemove extends BaseAction {
  public String execute() {
    services.removePersonFromEvent(personId, eventId);
    return redirect(String.format("Attendance.action?id=%d", eventId));
  }

  int eventId;
  public void setEventId(int value) {eventId = value;}
  int personId;
  public void setPersonId(int value) {personId = value;}
}
```

Restart the server and try clicking on the add and remove buttons, to check you can modify the event attendance:

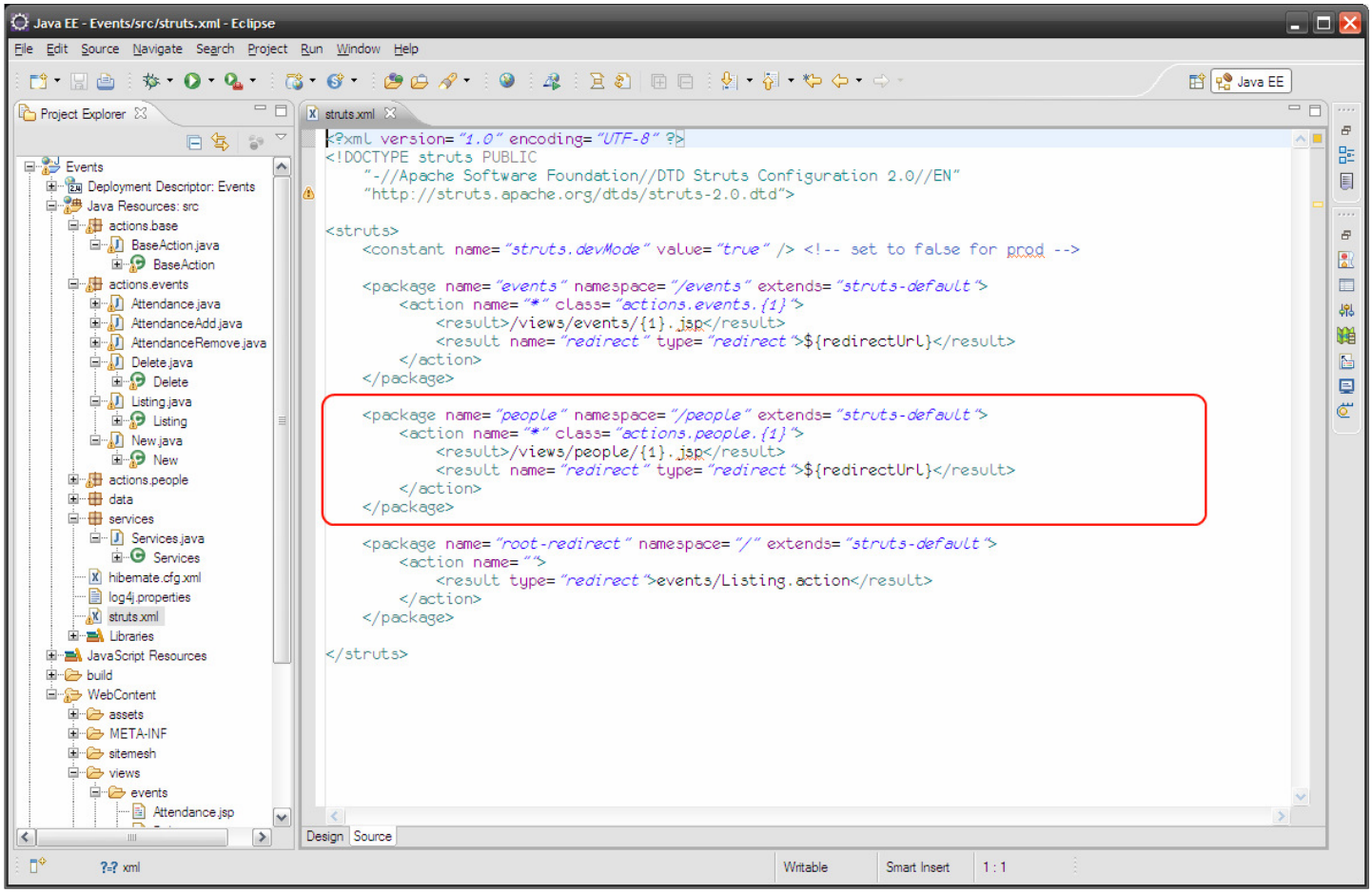


## *People Listing*

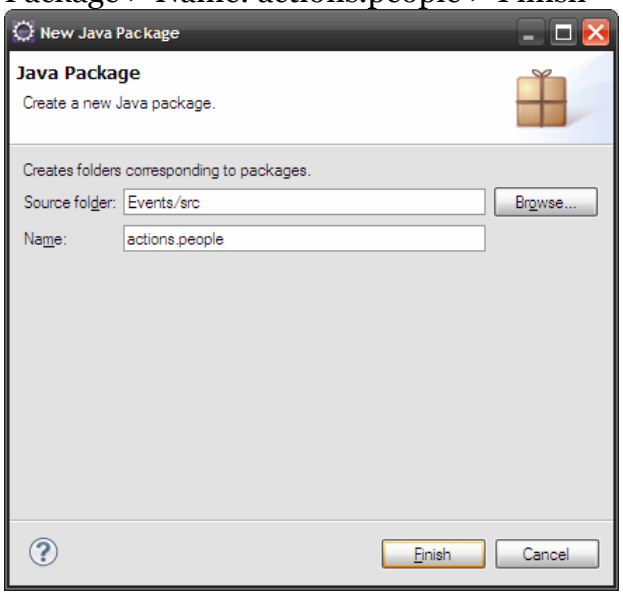
Lets make the people listing now. This listing is going to be in a new package, so the URL's will look like http://somewhere/something/people/Listing.action instead of http://somewhere/something/events/Listing.action. To do this, we'll need to set up a new package within the struts configuration.
Open up your src\struts.xml and add the following after your events package

```
<package name="people" namespace="/people" extends="struts-default">
  <action name="*" class="actions.people.{1}">
    <result>/views/people/{1}.jsp</result>
    <result name="redirect" type="redirect">${redirectUrl}</result>
  </action>
</package>
```




This configuration is needed by struts whenever you add a new package to your application.

Next we need to create a new source package for the action classes to go into. Right click src > New > Package > Name: actions.people > Finish




Create the new action class now. Right click the actions.people package > New > Class > Name: 'Listing' > Finish. The code for this class shall be:

**src\actions.people\Listing.java**

```
package actions.people;

import actions.base.BaseAction;
import data.*;
import java.util.List;

public class Listing extends BaseAction {

  public String execute() {
    people = services.getPeople();
    return "success";
  }

  List<Person> people;
  public List<Person> getPeople() { return people; }
}
```

As you can see, all this action does is use the business services to get a list of all people, and make that list available to the view via the getPeople() getter function.

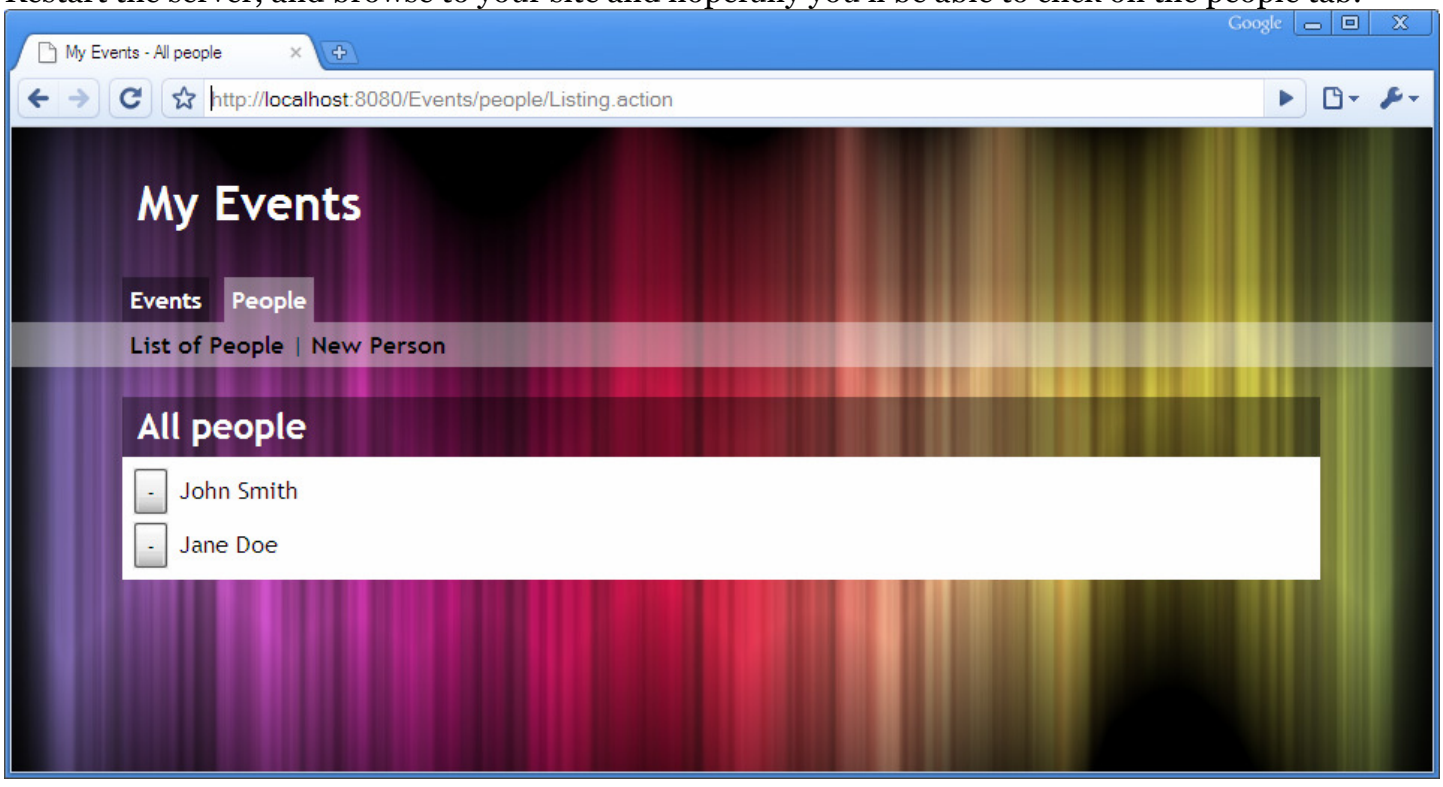
Now to make the view. Create a new folder under WebContent\views called 'people'. Right click the new 'people' folder > New > JSP > File name: 'Listing' > Finish. The code follows:

**WebContent\views\people\Listing.jsp**

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>All people</title>
</head>
<body>

<s:iterator value="people">
  <form action='Delete.action'>
    <input type='hidden' name=id value='<s:property value="id" />' />
    <input type='submit' value='-' title='Delete' />
  </form>
  <s:property value="name" />
  <br>
</s:iterator>

</body>
</html>
```

The view is nothing new, all it does is iterate over the list of people from the action's getPeople() function, displays all their names with the s:property, and display a delete button next to each, with the id for each person embedded in each button's form.

Restart the server, and browse to your site and hopefully you'll be able to click on the people tab:



## *New Person*

We want to create a form so that we can add new people to the database. Create the following action in actions.people:

### src\actions.people\New.java

```
package actions.people;

import actions.base.BaseAction;

public class New extends BaseAction {

  public String execute() {
    if (name!=null && name.length()>0)
    {
      services.createPerson(name);
      return redirect("Listing.action");
    }
    return "success";
  }

    String name;
    public String getName() {return name;}
    public void setName(String value) {name = value;}
}
```

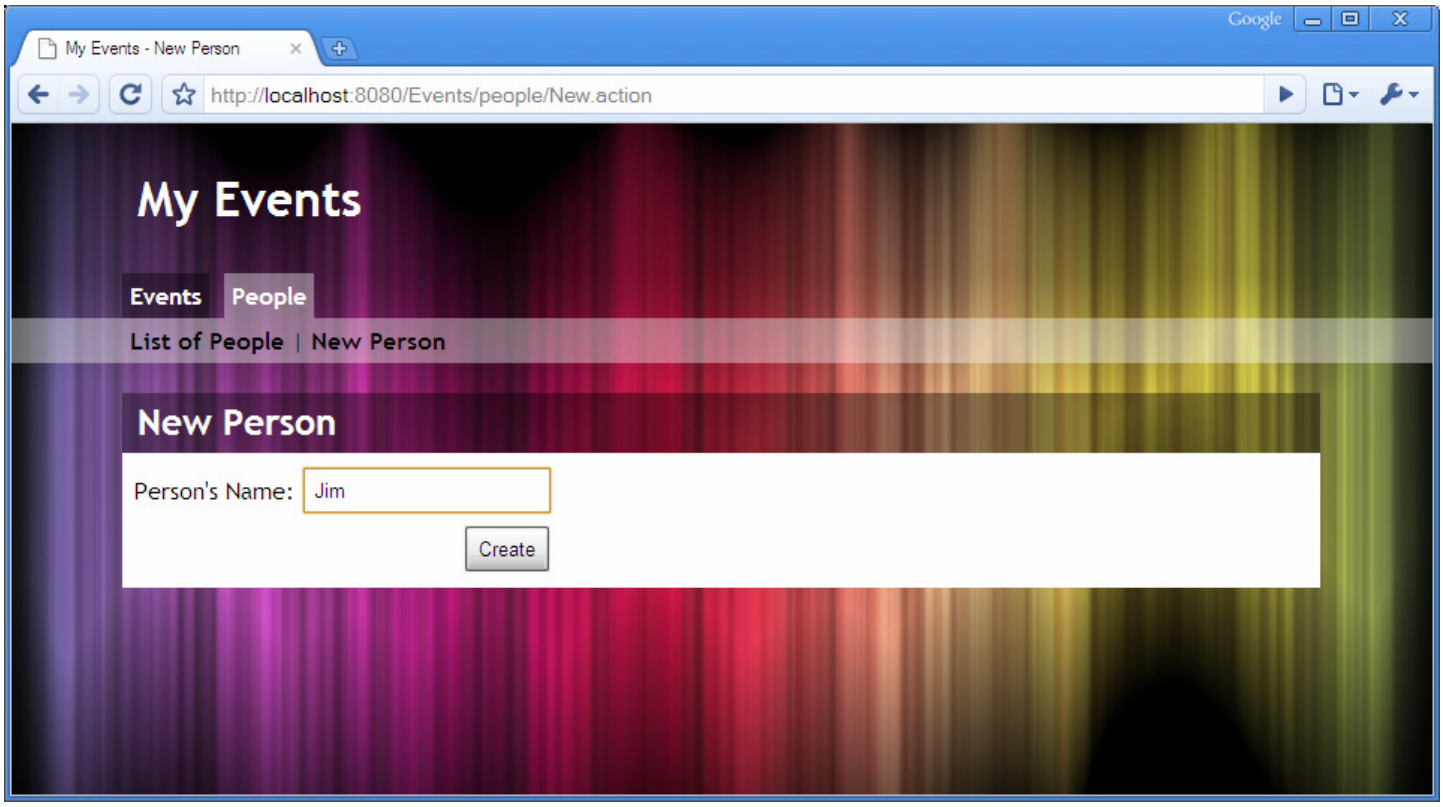
This action checks to see if the name has been set, as it will be when the user does a postback, and if so it uses the business services to create the new person and redirects the browser to the people listing page afterwards. The view code will be as follows:

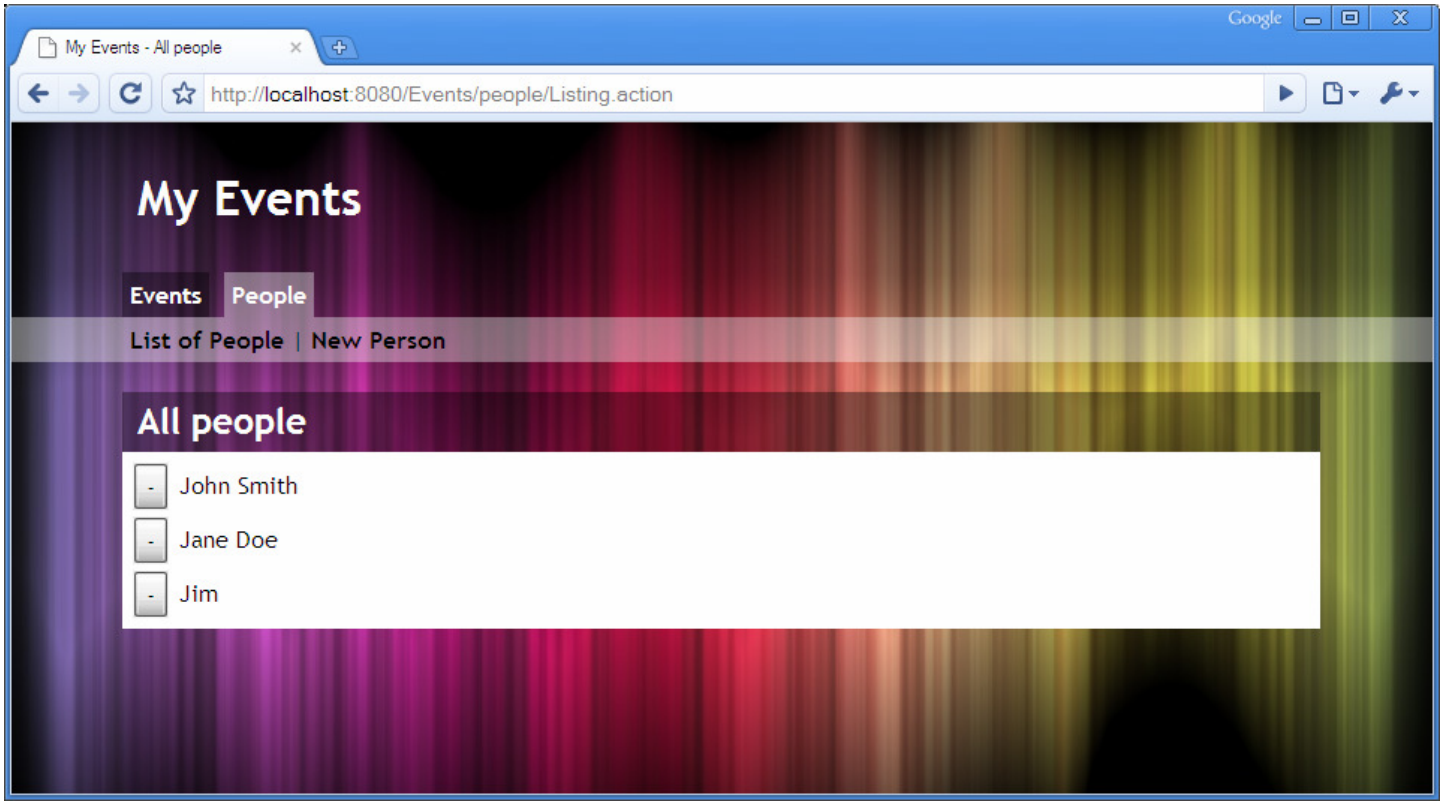
### WebContent\views\people\New.jsp

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>New Person</title>
</head>
<body>

<s:form>
  <s:textfield label="Person's Name" name="name"/>
  <s:submit value="Create" />
</s:form>

</body>
</html>
```

Remember that the s:textfield links through it's name="name" property to the getName and setName properties in the action, so that when the user does a postback the action can see what was inputted.

Restart the server and try adding a few names:



## *Delete a person*

Go to the people listing in your browser, and you'll see the delete buttons next to each person. We now need to wire them up to an action. Create the following action class:

### src\actions.people\Delete.java

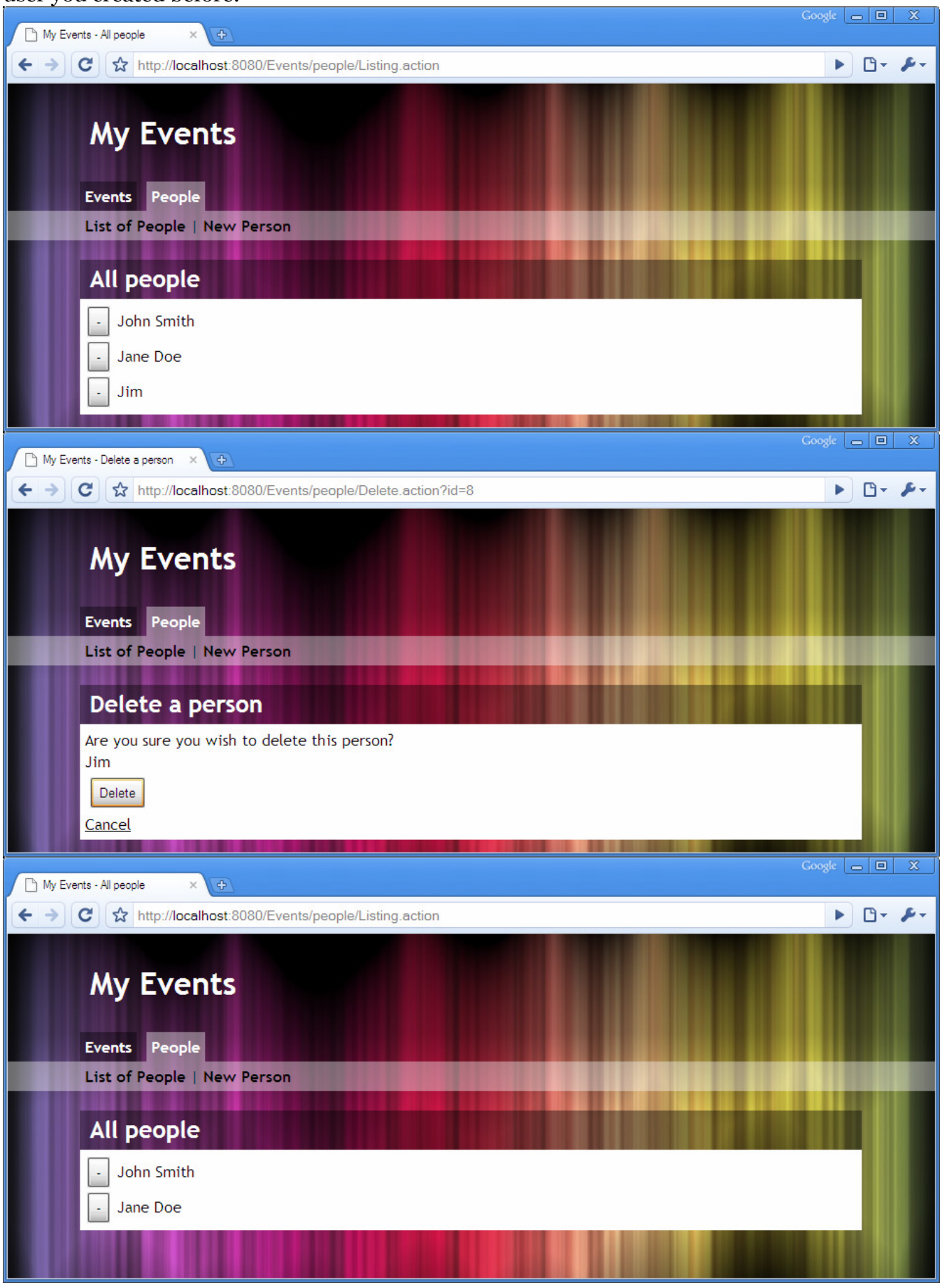
```
package actions.people;

import actions.base.BaseAction;

public class Delete extends BaseAction {

  public String execute() {
    if (isPostBack)
    {
      services.deletePersonById(id);
      return redirect("Listing.action");
    }
    return "success";
  }

  int id;
  public void setId(int value) {id = value;}
  public int getId() {return id; }

  boolean isPostBack;
  public void setIsPostBack(boolean value) {isPostBack = value;}

  public String getName() { return services.getPersonById(id).getName(); }
}
```

Upon the first load of the page, only the person's id will be given to the action, so it displays the page which asks the user to confirm the deletion. This deletion confirmation page has a hidden 'isPostBack' variable which has the value 'true', so that when the action is called for the post back, it sees this variable is set to true and knows to do the deletion. The view code follows:

### WebContent\views\people\Delete.jsp

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Delete a person</title>
</head>
<body>

<s:form>
  Are you sure you wish to delete this person?<br />
  <s:property value="name" />
  <s:hidden name="id" />
  <input type="hidden" value="true" name="isPostBack" />
  <s:submit value="Delete" />
</s:form>

<s:a action="Listing">Cancel</s:a>

</body>
</html>
```

Now to try deleting a user. Reset the server and browse to the people list, and try to delete the new user you created before:



## Where to from here?

Congratulations if it all worked out well for you. If you're having issues, feel free to email me, or you may have more luck asking questions at stackoverflow.com.

Here's a bunch of links to useful tutorials that have helped me and would be a good place to start learning more:

http://www.struts2.net/tutorial1.htm

http://struts.apache.org/2.1.8.1/docs/tutorials.html

http://www.struts2.org/developing-struts2-applications-with-eclipse-ide-struts-2-1-x/

http://stackoverflow.com/ (this is a great forum where you'll find help when you get stuck)

http://www.vaannila.com/struts-2/struts-2-example/struts-2-ui-tags-example-1.html

http://struts.apache.org/2.1.8.1/docs/tag-reference.html (very useful when making jsp's)

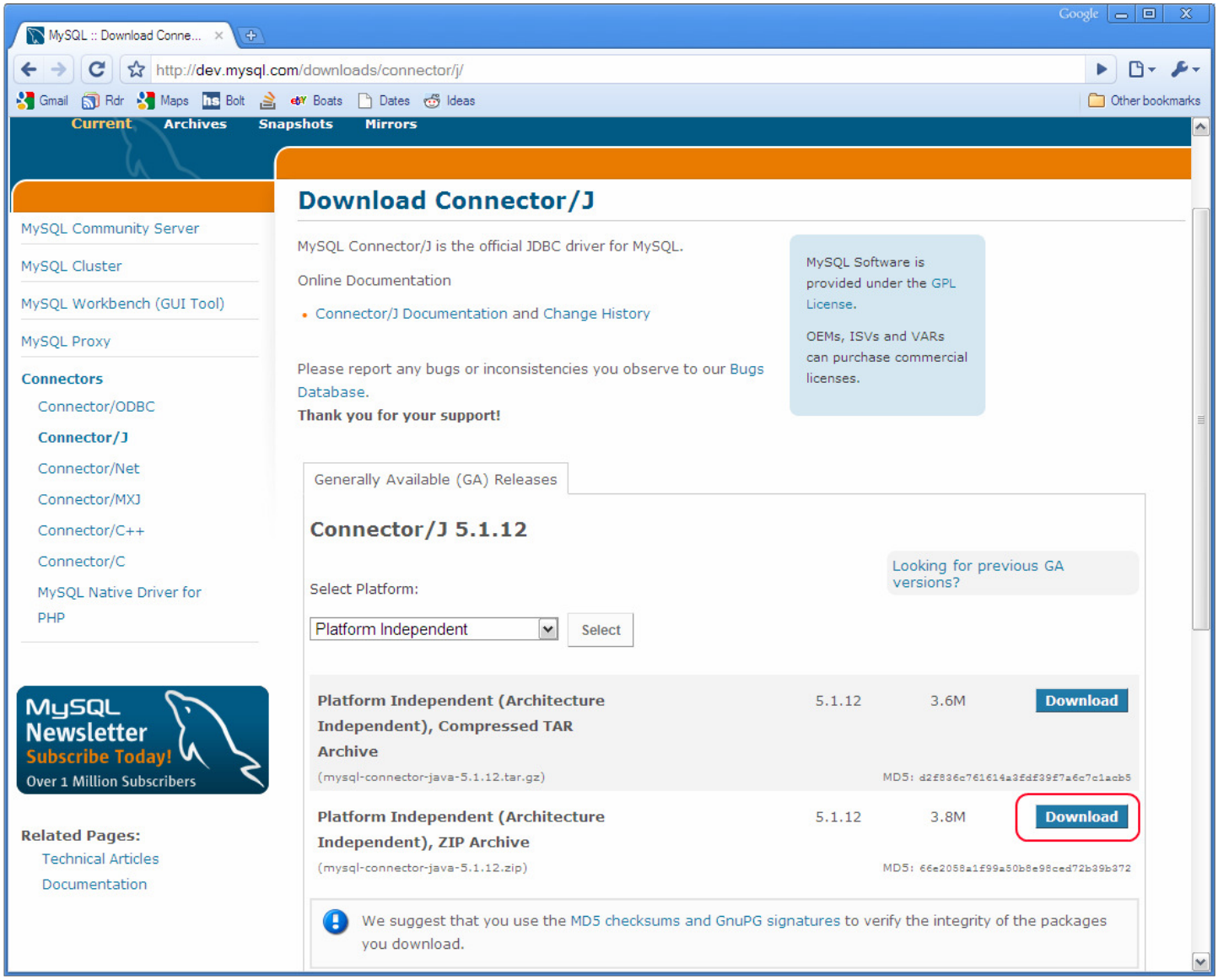
http://struts.apache.org/2.0.14/docs/ognl-basics.html (to understand what you can do with 'value' in your jsp struts tags eg *<s:property value="[1].id" />* for when you're iterating through people, but you want the action's id, not the persons id)

http://levelup.lishman.com/hibernate/getting-started/index.php (hibernate tutorial)

# Appendix: Using MySql

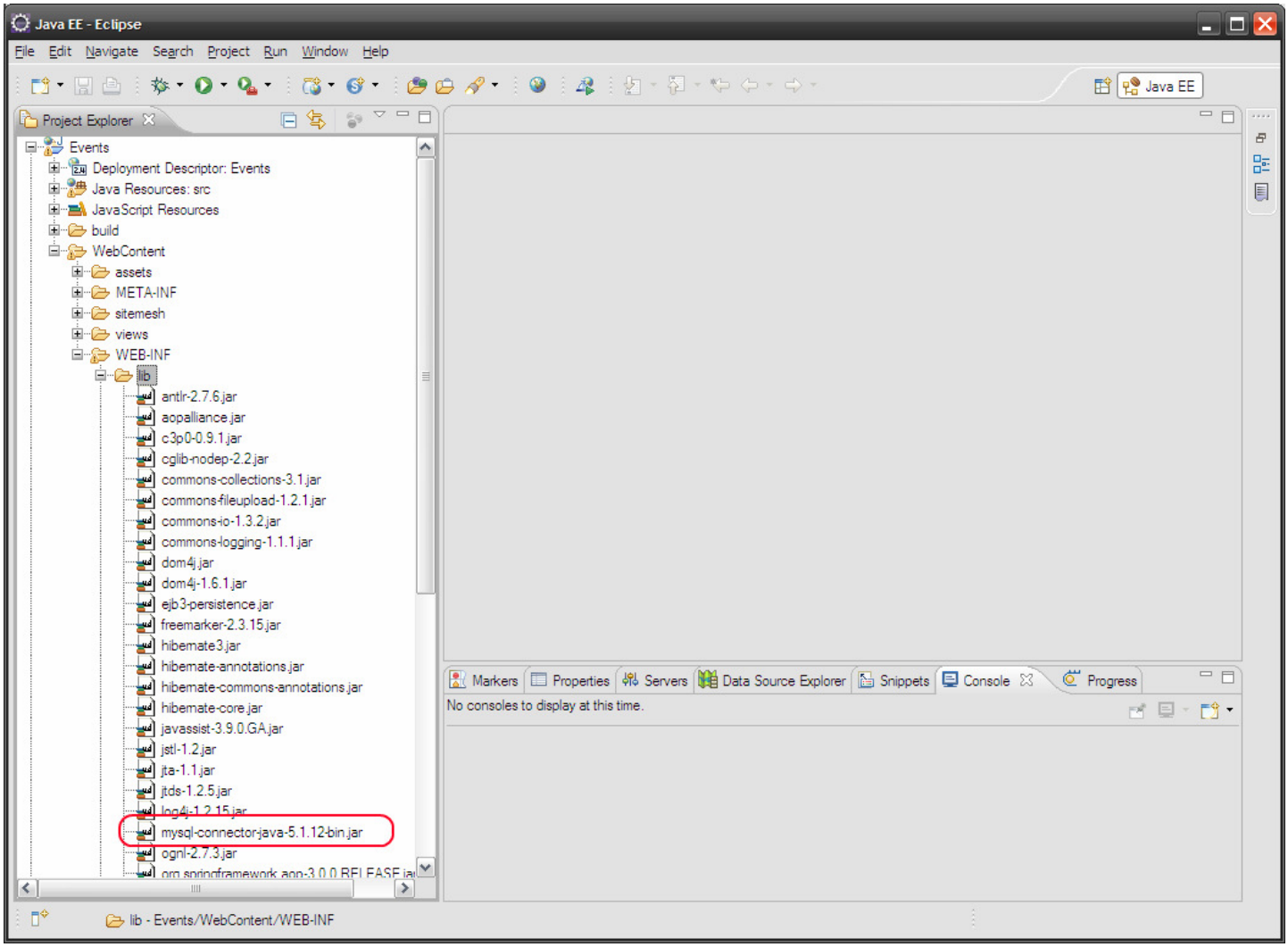
If, like a lot of people who have gotten in touch with me, you'd like to make all this work with MySql instead, here's how:

Firstly, go to http://dev.mysql.com/downloads/connector/j/ and download the zip version:

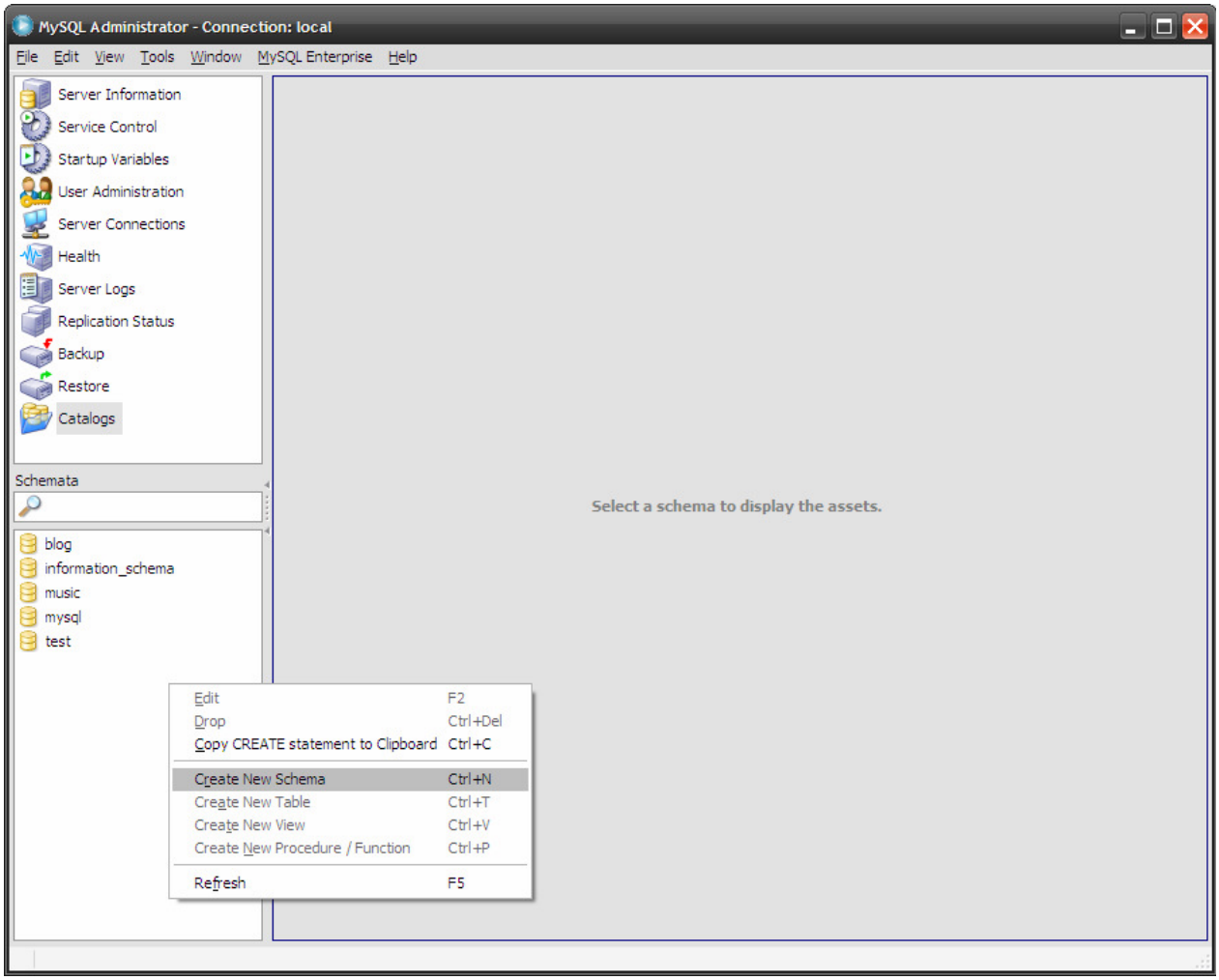



I downloaded 'mysql-connector-java-5.1.12.zip' which was the latest at the time of writing.

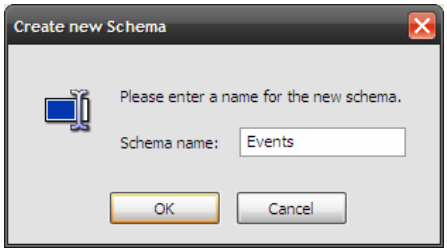
Unzip the downloaded file and find the 'mysql-connector-java-X-bin.jar' file. This is the JDBC driver that is used to connect Java to MySql. Copy this file into your project's WebContent\WEB-INF\lib folder:



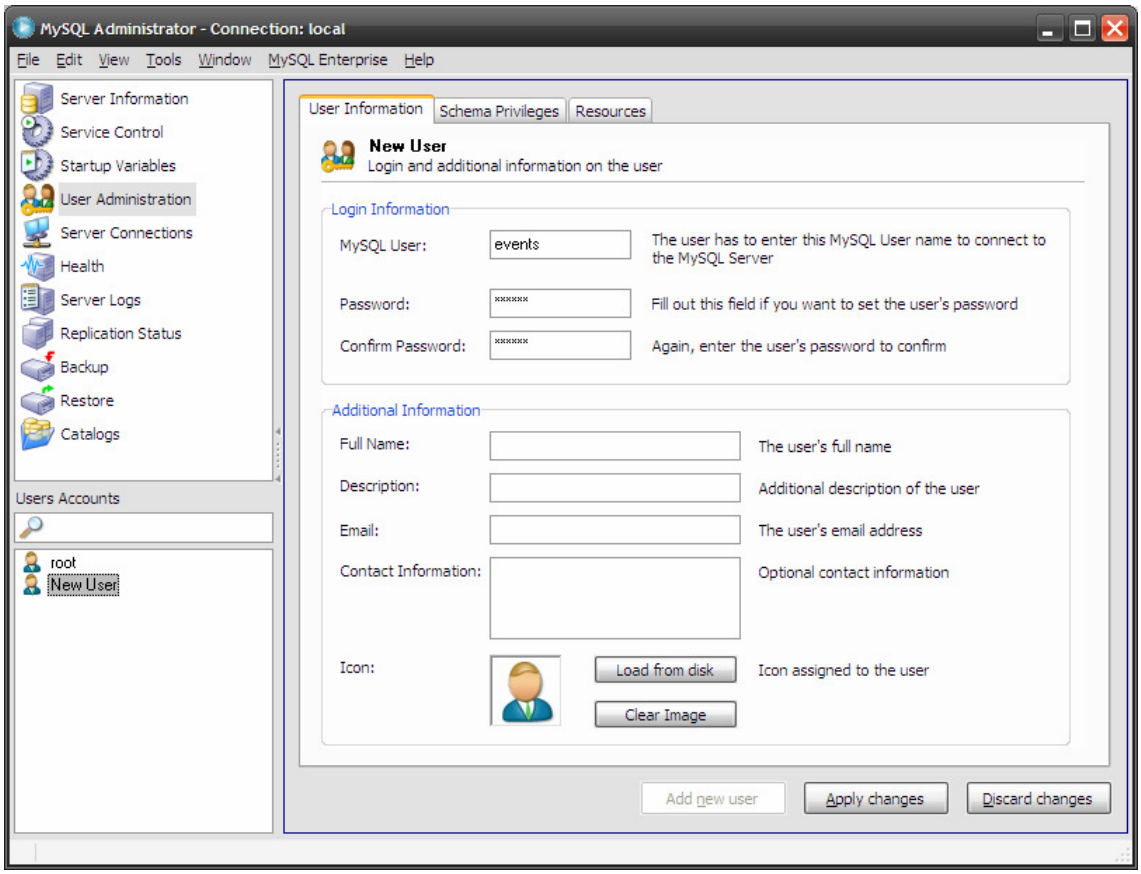
Now lets setup your database. Go into MySql Administrator and create a new database/catalog/schema whatever you call it. To do this, go to the 'catalogs' view and right click below > Create new schema:



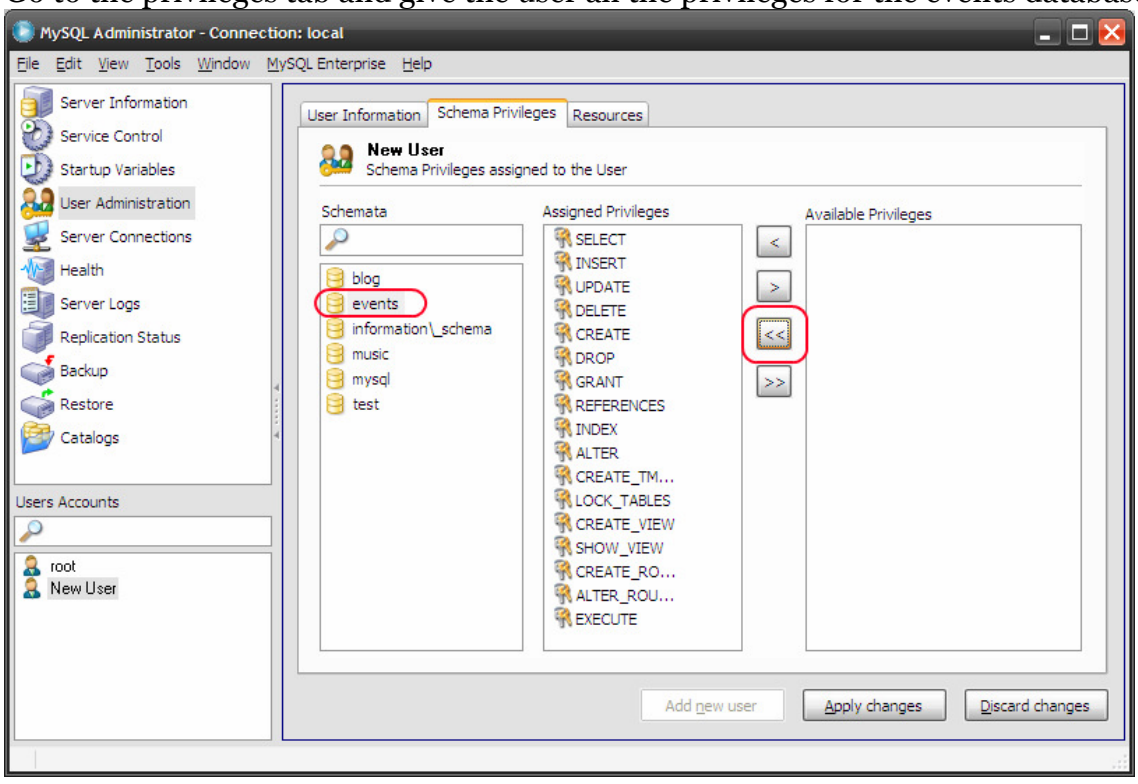

Call it Events:



Create a new user called ‘events’, with the password ‘events’:




Go to the privileges tab and give the user all the privileges for the events database:

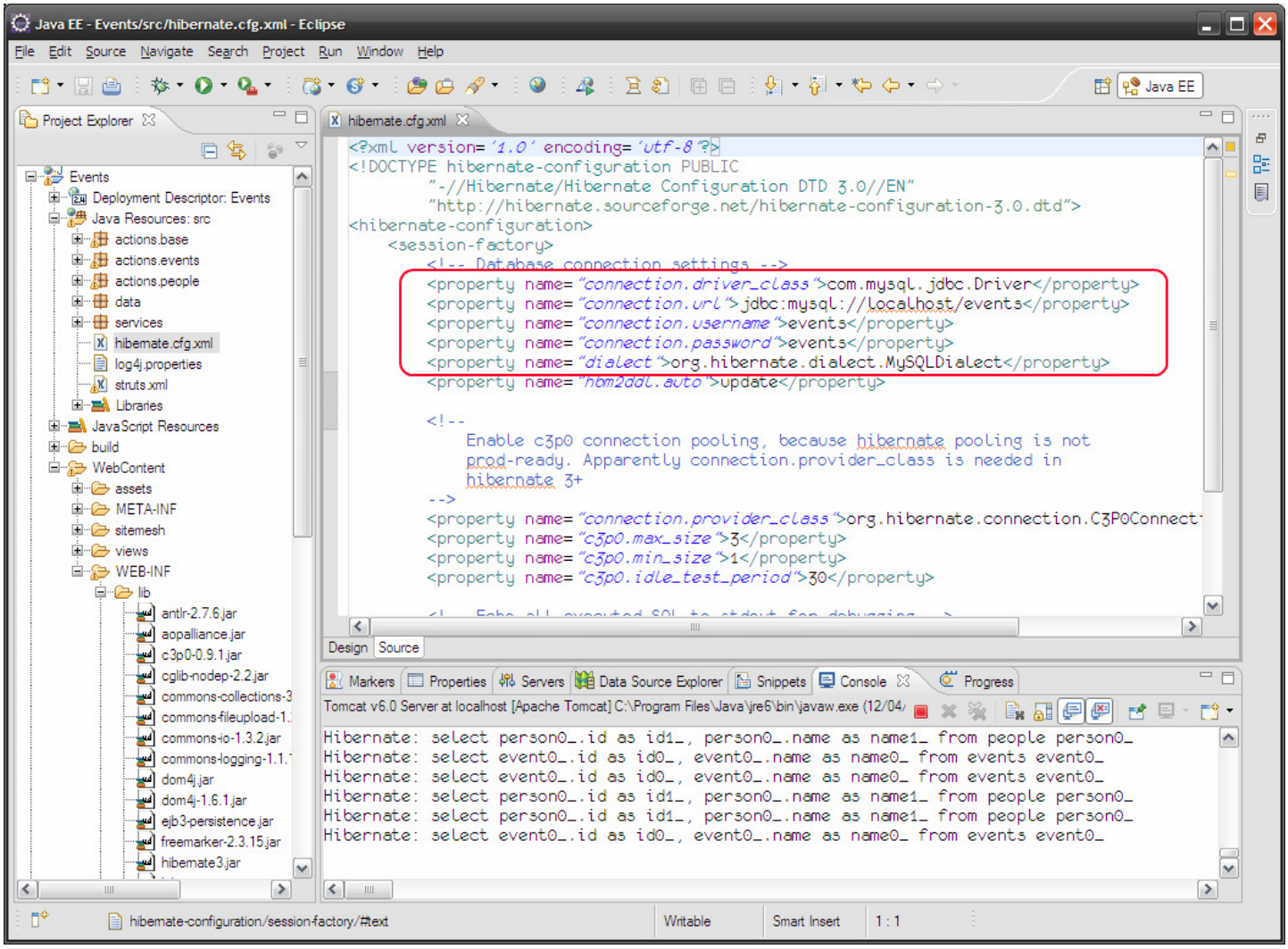



Then click apply changes to create the events user. You won’t need to do any more work directly with mysql, or create the table structure, because hibernate will do that for us.

Next we need to modify our hibernate configuration to suit. Open the src\hibernate.cfg.xml file in Eclipse, and make the following changes:

- The value of connection.driver_class is to be 'com.mysql.jdbc.Driver'.
- Value of connection.url is 'jdbc:mysql://localhost/events', assuming your database name is 'events' and your local machine is the mysql server.
- Set the connection.username and connection.password appropriately.
- Set the dialect to 'org.hibernate.dialect.MySQLDialect'.

It should look like this:




Save everything and restart the server, and it should work.

For more details, eg if you need to customise your connection url for an obscure MySql install, see: http://dev.mysql.com/doc/refman/4.1/en/connector-j-reference-configuration-properties.html