Introduction 2
Environment 2
DOWNLOAD J2EE 3
Eclipse 3
  Download and install 3
  Create a simple HelloWorld program 3
JBoss 6
  Download and install 6
  Configuring Eclipse to work with JBoss 7
  'HelloWorld' as a stateless session EJB. 8
    Introduction 8
    Make a new project for our server code. 8
    Make the sessionbean 8
    Create the remote interface 9
    Create the home interface 9
    Package the bean into a JAR to be deployed on JBoss 9
    Deploying the bean 10
    Changing our client to use the bean 11
    Running and debugging 12
Accessing a database 13
  Introduction 13
  MySql 13
  JDBC Driver for MySql : 16
  Download and install ANT 16
  Middlegen : generate EJB 17
    Adapting config/database/mysql.xml 17
    Adapt Build.xml 17
    Generate EJB's 19
  Back to eclipse : update the server application to read the database 20
    Add the beans to the project 20
    Modify the session bean to get its data from the entity beans 21
    Modify, run and debug the client. 24
JSP 26
  Required setup 26
    Download and install Apache Tomcat 26
    Download and install 'Sysdeo Eclipse Tomcat Launcher plugin' 27
    Download and install 'Solar Eclipse' 28
  A Simple JSP example : WhatTimeIsIt 29
Building and debugging a lot-of-tiers J2EE web application 33
  View-Controller (the model will come later) 33
    A first view 33
    Some More Views (we would not need a controller otherwise) 34
    Then comes the controller 36
    Add the Model 42
Store and get preferences and settings in XML files 42
What's next 42

Introduction

It all started with the article 'Working with Java – Enterprise Java' on Apples website. It showed that enterprise Java on OS X was possible, that everything worked. But it left me somewhat unhappy. As I am very GUI minded – I would not be a Mac user if I wasn't – I wanted an IDE that could help me with my OS X enterprise java stuff, and nowhere in that article was any hint to such an IDE. Secondly, but very related, I want to debug my server code. And last but not least, I do not want to spend my time writing repetitive code or code that can easily be generated. Just think about EJB-JAR.XML, a good part of jboss.xml and even the ejb's and their interfaces–all these getters and setters - they can easily be generated from the lay-outs of your database..
Fortunately, I got involved in a big J2EE project, exchanging messages between unrelated services via XML and https. We choose J2EE as the platform to implement our part of it, and JBoss as our application server.
The environment at work is Windows plus Oracle. For this article, I replaced the Oracle database by MySql, and Windows by OS X. All the rest stayed the same. If you prefer another database, you can do so. This article describes the necessary steps to make JBoss work with MySql. Doing this for another database is quite similar. If you want to use Windows or *nix, you can as well: all the software used trough this article is available on all these platforms.
The IDE used is Eclipse, a free product written by IBM and backed by others (Borland being one of them). Although it lacks some of the flashy features offered by commercial IDE's (just think of GUI-editors, wizards…), it is very good at what it does, and its performance is acceptable. It is even usable on a Pismo 500 MHz PowerBook, although you'll better have the fastest dual G4 at hand if you are using it on a daily basis. Eclipse has an extensible plug-in architecture. The integration with JBoss is handled by such a plug-in to name one. No doubt GUI-editing, EJB-generation and others will be added as plug-ins later. (If you all register WhereDidAllMyMoneyGo?, I could do this myself).
As stated earlier, MySql is the database system I use. To generate 'stupid code', I am using a command line tool called Middlegen. JBoss is my application server, Tomcat my webserver. All these tools are available for free under a GNU licence.
Later, I might add a chapter on version control (using CVS), a chapter on XML (using Castor), but currently, I'll limit myself to session and entity beans, and to JSP and servlets.
One last thing: if you do not know what an EJB is, what the difference is between a session and an entity bean, and what a local and a remote interface is, I suggest you read the relevant chapters of Suns J2ee tutorial first.
This document is pretty much written as a step-by-step instruction – making sense only when you repeat everything that is described on your own machine while reading.
Oh yes, one more thing: I did not pay any attention to formatting and layout. Sorry for the ugly headers – they are what Microsoft gave me.

Environment

All the stuff described in this article was written and tested on a dual GHz Quicksilver G4 running OS X 10.2.1. 10.2 is a prerequisite for Eclipse – threads on Apples Java developer mailing list learn that people using 10.1 have difficulties running Eclipse. A second prerequisite is a fast Internet connection. There is over 100 megabytes to download if you want to replay this article.
I unzipped or unstuffed all downloads with Stuffit Expander. You MUST use Stuffit Expander version 7.0 or later, as earlier versions make a mess of long filenames.
Everything I downloaded just worked as advertised for me after unstuffing. I do not have any unsupported tools, cracked or doubtful software or haxy installed, so my system is quite out-of-the-box.
As I am not a command line freak, I did not change the configuration of my terminal application. If you use another shell, I guess you know enough of Unix to adapt the few Unix commands presented in this document so they work with your shell.
And yes, I have the latest version of Apples developer tools installed. (Don't know if it is required though.)

DOWNLOAD J2EE

The first download – although not really required - is Suns J2ee sdk. This SDK contains a lot of Java classes and some java tools that you might need to build and run J2EE applications. It can be downloaded from http://java.sun.com/j2ee/sdk_1.3/. (Id did not download the 1.4 version, as OS X java is still at 1.3.1). Just download and unzip it. I unzipped it to Aa folder named /Applications/java/j2sdkee1.3.1.
This download is not really necessary for the examples of this article. All I did with this download was to include the file 'j2ee.jar' in my projects. J2ee.jar contains the interfaces implemented by enterprise beans. But these interfaces are – if you search a bit – included in jar-files that come with JBoss as well. But as I had downloaded the sdk earlier on alraedy, I just used it without looking further. But pf course, including the full J2ee.jar file in a J2ee application is overkill.
When you finished installing the j2ee-sdk, it might be a good time to read trough Suns excellent J2EE tutorial, available from java.sun.com as well.

Eclipse

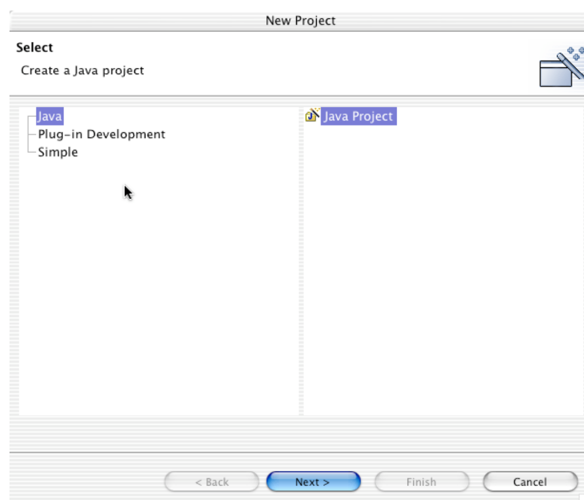
Download and install

First of all, Eclipse requires OSX 10.2 or later. If you are running on 10.1x, you should repackage eclipse (doing things with the info.plist file amongst other things. There is a thread on this on Apples Java developer mailing list).
Eclipse can be downloaded from http://www.eclipse.org/downloads/index.php.
The latest OSX builds are currently in the 'Stable build' category, next to the M1 build name. Future builds will appear in the 'Latest Release' category. The current stable build weights about 57 Mb. Download it and unzip it with a version of Stuffit Expander later than 7.0. (You might have to add the .zip extension to the file in order for the system to identify it as a zip archive.). If you have an older version, you should use the command line tools that come standard with OSX to unzip.
The result of the download process is a folder called 'Eclipse' that contains, well, Eclipse. Just move this folder to the 'Applications' folder.
You launch Eclipse by double-clicking it. Note that the help does not seem to work in the current stable build.
Then close Eclipse and restart it. My version complains that it is not able to restore the current workspace (which is the welcome screen), but once you dismissed that dialog, it should work Ok.

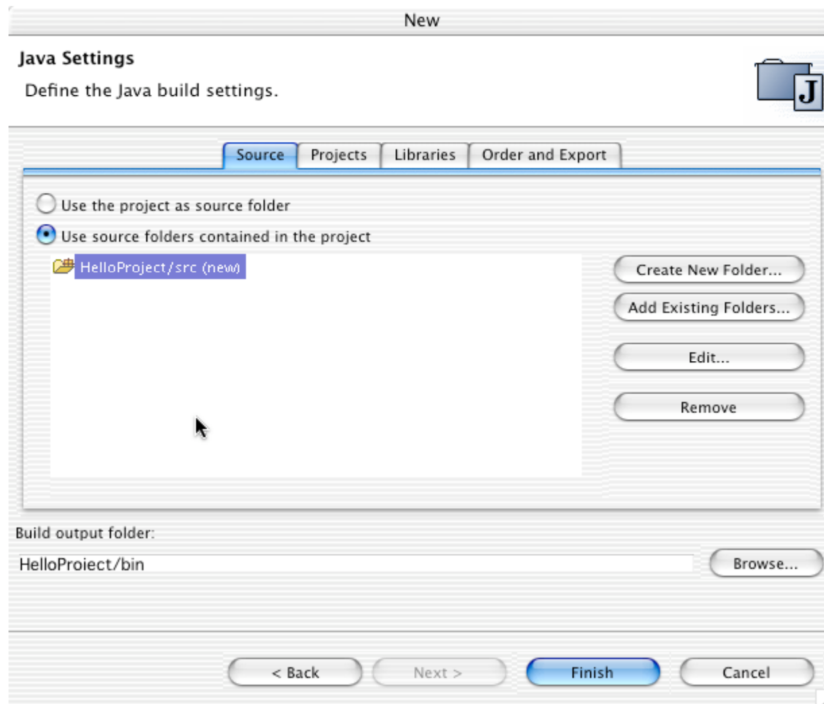
Create a simple HelloWorld program.

So lets create the omnipresent 'Hello World' program.

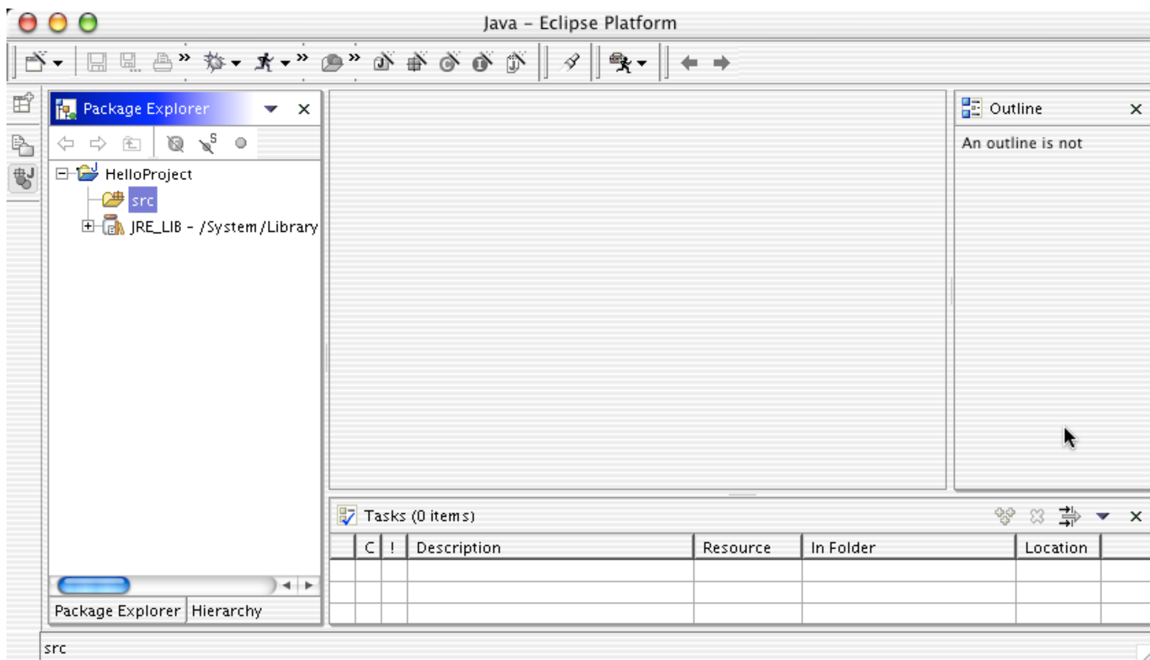
- Choose File->New->Project
- Select Java and Java Project, then click 'Next >>'



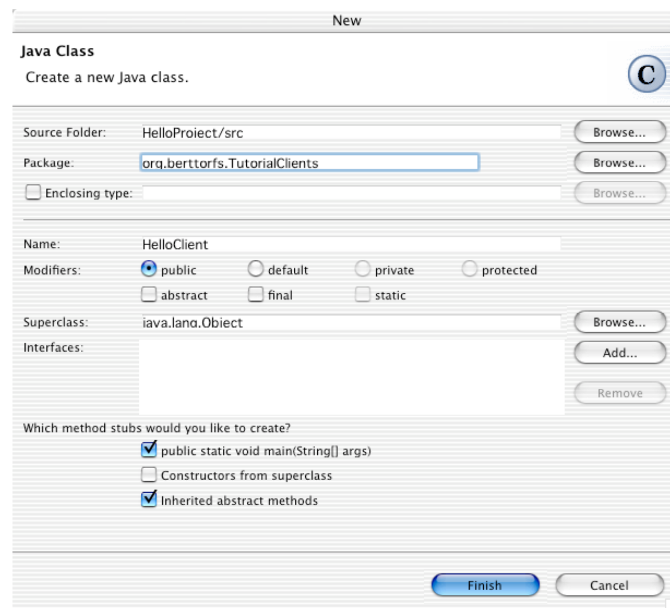
- Give your project a name (I named it HelloProject) and a location (I kept the default, which is inside the 'Eclipse' folder – bad choice me thinks) and press 'Next>>' again.
- On the next screen, check the 'Use source folders contained in the project' button, push the 'Create New Folder…' button and name the source folder 'src'. Answer 'Yes' when Eclipse suggests you to place builds in a build folder as well. Next Press 'Finish'.



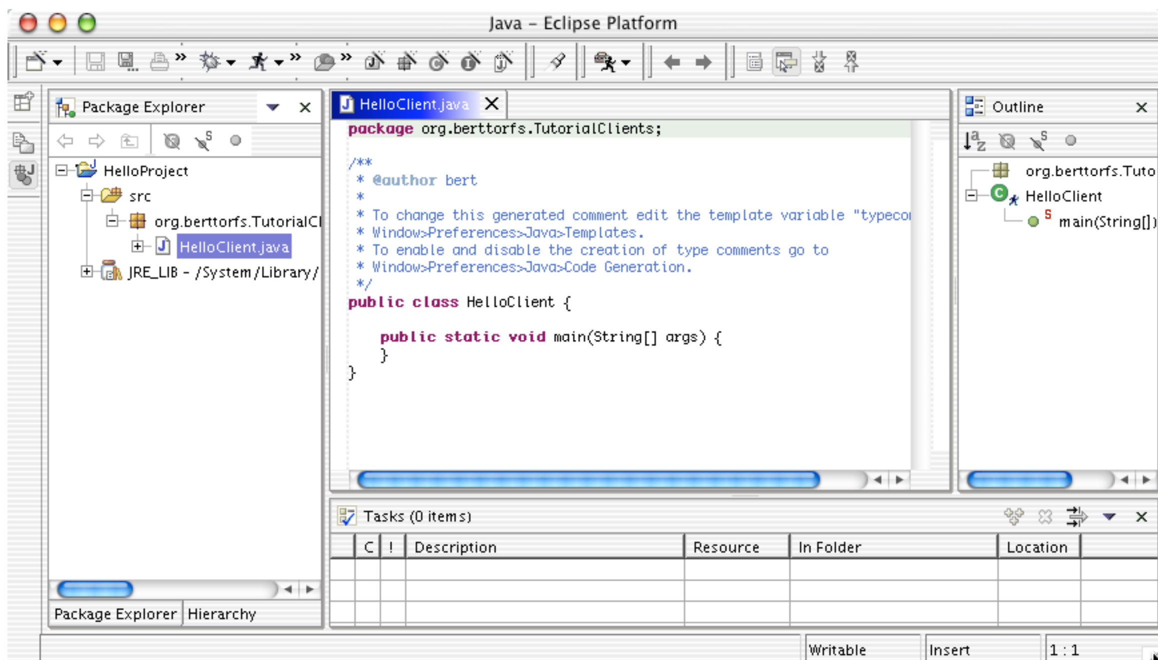
- The workspace should now look like this:



- Next thing to do is to create our program. Control-click the HelloProject folder and select 'New Class' from the contextual menu and fill in the dialog. Note that you cannot use the <Tab> key to move between form fields. <Tab> will just insert a tab into the current field. The dialog should be like this when filled:



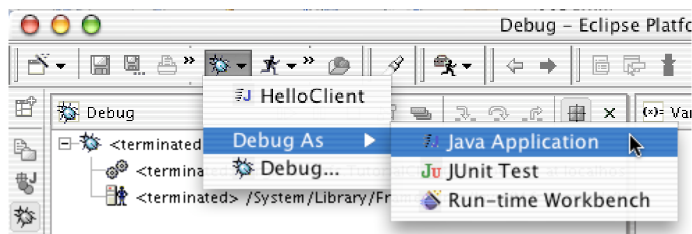
I checked ‘public static void main…’ so my class would have a main method. The package name was chosen to avoid name collisions with all the other HelloClients out there in cyberspace. When you next press ‘Finish’, you should get this:



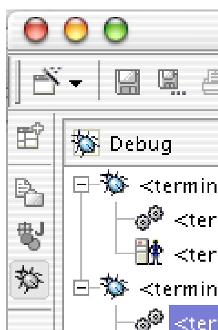
- As I do want my curly brackets to appear on a new line, I opened the ‘Window->Preferences’ dialog, scrolled to the ‘Java->Code Formatter’ category and checked ‘Insert a new line before an opening brace’, ‘insert new lines in control statements’ and ‘Clear all blank lines’. To apply these formatting rules to existing sources, control-click the editor pane and select ‘Format’ from the contextual menu.
- Next we complete the main method. It should be

```
public class HelloClient
{
   public static void main(String[] args)
   {
      System.out.println("Hello world");
   }
}
```

- Then click on the little arrow next to the ‘Bug’ button, and select ‘Debug as->Java Application’ from the menu.



- You will be asked to save your source (saving and compiling are linked in Eclipse). Answer 'yes'. Eclipse should now switch from 'Resource' perspective to debug perspective, launch your program, and display its results in the console (the pane at the bottom of the debug perspective).
- If you would like to debug your program, just double-click in the margin next to the statement onto which you want the debugger to break.
- To switch back from debug perspective to resource perspective, click the little 'resource perspective' button on the left of the main window (the button showing a clipboard and a folder). A perspective is just a set of open panes that are suitable for a specific task. Please consult the Eclipse documentation if you want to learn more about perspectives.



JBoss

Now we are going to split our application in two parts: the interface and the business logic. The business logic can say 'Hello World'; the interface can display whatever the business side says. Big advantage of this approach is that we can put different 'Interfaces' onto our business logic: terminal clients, web clients, GUI-clients etc…, all linked to the same unique business code that gets the real work done. Or we can share the same business logic between different applications.
To do this, we need an application server to host the business part. We choose JBoss – a freeware J2EE application server – for this purpose. JBoss itself is an EJB container. It comes bundled with Jakarta Tomcat or Jetty to serve web pages, JSP pages and Servlets. You could run both JBoss and Tomcat (or Jetty) on different machines if you wanted. In a production/internet environment, it is wise to do so: Tomcat or Jetty will then run outside the firewall, just handling Internet traffic. JBoss will do the real business work from a server inside your firewall. As that server must have access to resources like databases etc, it is saver to keep in inside your firewall.
But for our example application, we will use the standard installation with JBoss and Jetty tied together.

Download and install

You can download JBoss from http://www.jboss.org. (another 28 Mb). I downloaded JBoss-3.0.3.zip. Unzip it with version 7.0 or later of Stuffit Expander. if you have version 6.x, expanding will create a useless JBoss installation, as versions prior to 7.0 truncate long filenames. (you can always use the command line tools if you do not have version 7.x).
When expanding is finished, just move your JBoss 3.0.3 folder to the applications folder.
If you want to, you can test JBoss. Just open a terminal and type

```
Prompt>cd /applications
Prompt>cd jboss-3.0.3
Prompt>cd bin
Prompt>./run.sh
```

You will now get a lot of messages, ending with (if everything went fine)

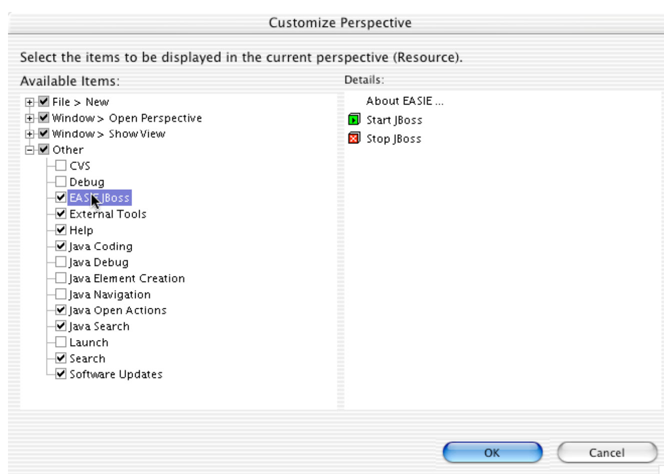
```
11:41:41,654 INFO  [Server] JBoss (MX MicroKernel) [3.0.3 Date:200209301503] Started in
0m:12s:188ms
```

Now just press <Ctrl><C> to terminate JBoss again, as we will use Eclipse to run and control JBoss.

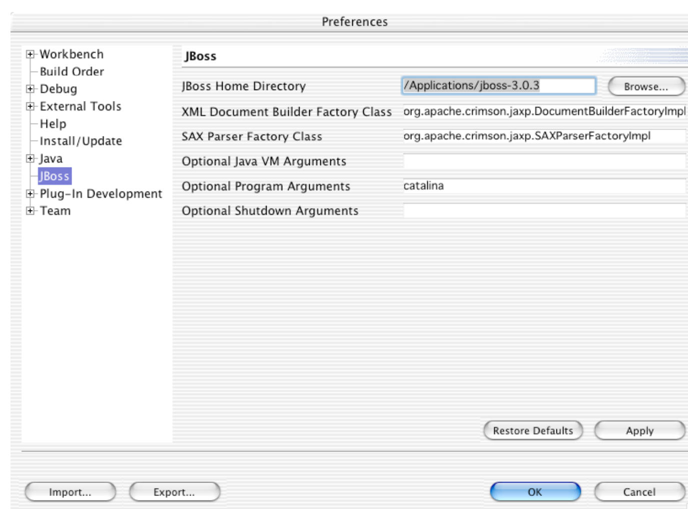
Configuring Eclipse to work with JBoss

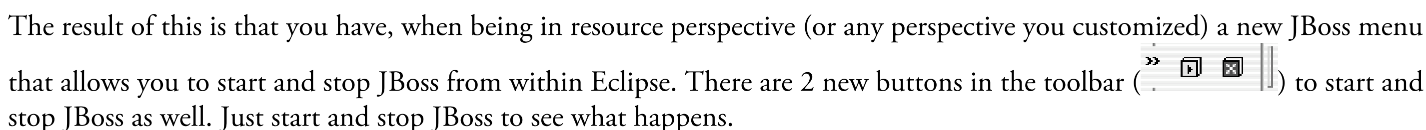
To control JBoss from within Eclipse, you need a plugin. A good place to find Eclipse plugins is http://eclipse-plugins.2y.net. The one we need for JBoss is called EASY JBoss Plugin, and can be downloaded from http://www.genuitec.com/products.htm#easie. I downloaded the plug-in for Eclipse Builds >= 20020409. After downloading and unzipping, you have a folder called 'PlugIns' with two subfolders. You should copy these two folders (not the surrounding 'Plugins' folder) into Eclipse:

- Close Eclipse
- <Ctrl>-Click on the Eclipse icon. Choose 'Show Package Contents' from the contextual menu.
- Navigate to the folder contents->Resources->Java->plugins.
- Drop the 2 folders (from the plugins folder you just downloaded) into the plugins folder contained in the Eclipse package and restart Eclipse.
- Open, in Eclipse, the 'Window->Customize perspective' dialog. Open the 'Others' category and check 'EasyJBoss'. (I did it with the resource perspective open, but you can do it with other perspectives as well).



- Now open the preferences dialog (window->preferences), you'll notice a new item in the on the left of the dialog named 'JBoss'. Select it and set the JBoss Home directory to where it is (in my case, it is /Applications/jboss-3.0.3). Just leave the rest as it is. Press 'Apply' and 'Ok'. (Not sure about the apply, but it does not hurt)



The result of this is that you have, when being in resource perspective (or any perspective you customized) a new JBoss menu that allows you to start and stop JBoss from within Eclipse. There are 2 new buttons in the toolbar () to start and stop JBoss as well. Just start and stop JBoss to see what happens.

'HelloWorld' as a stateless session EJB.

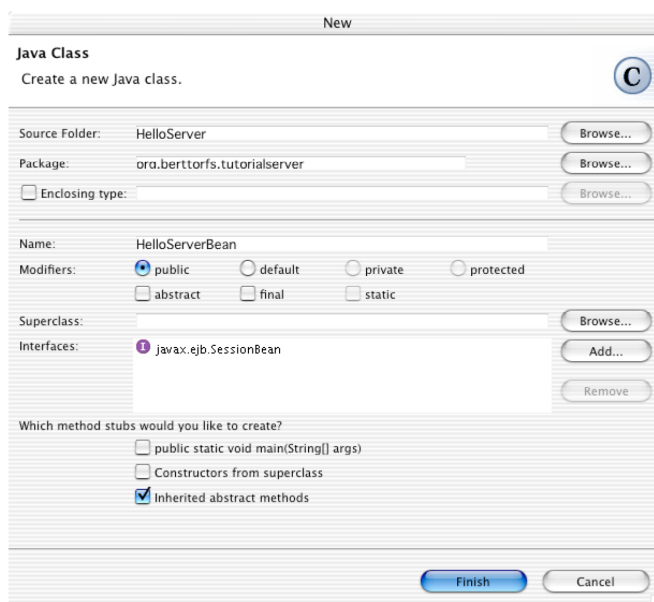
Introduction

Make a new project for our server code.

- Choose File->New->Project
- Name the new project 'HelloServer'. Press 'Next>'.
- Check the 'Use source folders contained in the project' button, push the 'Create New Folder…' button and name the source folder 'src'. Answer 'Yes' when Eclipse suggests you to place builds in a build folder as well. Press 'Finish' to create the project.
- <Ctrl>-click on the new project, choose properties from the contextual menu
- Choose 'Java Build Path' from the list on the left, and open the tab named 'Libraries'.
- Push the button 'add external jar'. Navigate to the J2EE Sdk and add the library 'J2EE.JAR from the Libs folder.

Make the sessionbean

- Now right-click the project folder and choose 'New->Other' from the contextual menu, and Java – Class in the dialog that opens. Press 'Next'.
- Complete the 'New Class' dialog. Name the new class 'HelloServerBean'. Make sure it implements the 'SessionBean' interface. Clear the checkbox to create the main method but leave the checkbox to create the inherited abstract methods.



- Press 'Finish'.
- A new class that implements all required sessionbean methods will be generated and opened in the editor. It just lacks an ejbCreate method, so it cannot be used as is. So we add a parameterless ejbCreate method:

```
public void ejbCreate()
{
}
```

- And we add a method named sayHello:

```
public String   sayHello() throws RemoteException
{
   return "Hello World. This is your sessionbean speaking!";
}
```

- Now press <Cmd>-S (or apple-S) to save your file. This will automatically compile your file. Any errors – you should not have any errors - will be listed in the Tasks pane of Eclipse. Just click on the error to jump to the line it occurs on.

Create the remote interface

- <Ctrl>-Click the HelloServer project. Choose 'New->Interface' from the contextual menu.
- Name the interface 'HelloServer' and select 'EJBObject' in 'Extended interfaces' field.
- Press 'Finish'.
- The remote interface should contain all the methods we want to call from our clients. So we add the sayHello() method listed above to the interface:

```
   public String     sayHello() throws RemoteException;
```

- Note that, as soon as you finished typing, the word 'RemoteException' will be underlined. Eclipse does compile as you type, and all things it cannot compile right away, will be underlined. In our case, this is because the package containing 'RemoteException' is not yet imported. To do this, press <Cmd+space> while the cursor is placed just after the word 'RemoteException'. Eclipse will add the necessary imports for you. Nice!

Create the home interface

The last thing to create is the home interface.

- Right-click the 'HelloServer' project and choose 'New->Interface'.
- Choose org.berttorfs.tutorialserver as the package. Name the interface 'HelloServerHome'. A home interface extends the 'EJBHome' interface. Select 'EJBHome' in the 'Extended Interfaces' field. You can leave the source folder empty, as we defined the folder named 'src' as such on the project level. Press 'Finish'.
- We have to add the create methods we need. In our case, we need only an empty create method that returns an object that implements the 'HelloServer' interface. Do not forget to press <Cmd+space> after 'RemoteException' and 'CreateException' to generate the necessary imports.

```
HelloServer create() throws  RemoteException, CreateException;
```

Package the bean into a JAR to be deployed on JBoss

So far for coding Java. We should now package our bean in a format that can be understood and deployed by JBoss. This is done into a JAR file that contains our compiled classes plus some deployment descriptors. We need two of them

- EJB-JAR.xml: this file is needed for every EJB application, regardless of the server onto which it is deployed. It lists all the EJBs to deploy and, for every EJB, its classes and interfaces.
- A second file we need is JBoss.xml. This files defines JNDI names for every EJB we want to locate remotely.

To add these files to our project, we proceed as follows:

- Right click the project node and select 'New->Folder' from the contextual menu.
- Name the new folder 'META-INF' and place it in the source folder (the one we called src).
- Right-click the META-INF folder and select 'New->File' from the contextual menu.
- Name the new file 'ejb-jar.xml' and press 'Finish'. Your favourite xml-editor should now open with an empty file. Type in the following :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN"
"http://java.sun.com/dtd/ejb-jar_2_0.dtd">
<ejb-jar>
  <enterprise-beans>
    <!-- Session Beans -->
    <session>
      <description>helloWorld Session bean example.</description>
      <display-name>HelloWorldServerDisplayName</display-name>
      <ejb-name>HelloWorldServer</ejb-name> <!--Matches with Jboss.xml -->
      <home>org.berttorfs.tutorialserver.HelloServerHome</home>
      <remote>org.berttorfs.tutorialserver.HelloServer</remote>
      <ejb-class>org.berttorfs.tutorialserver.HelloServerBean</ejb-class>
```

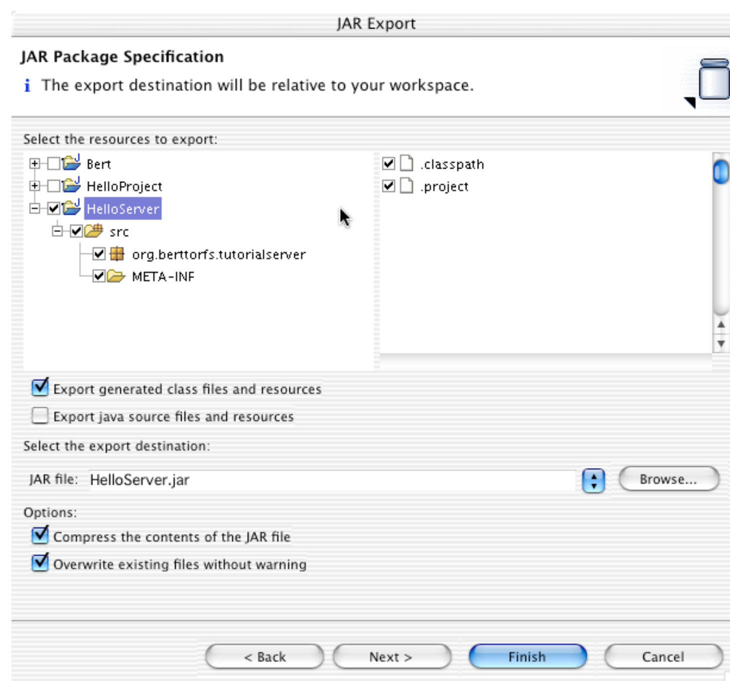
```
        <session-type>Stateless</session-type>
        <transaction-type>Container</transaction-type>
      </session>
   </enterprise-beans>
</ejb-jar>
```

- Next, we create the JBoss.xml file the same way. This file will link the JNDI-name 'ejb/HelloServer' to the EJB named 'HelloWoldServer' . Here is what it looks like:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE jboss PUBLIC "-//JBoss//DTD JBOSS 3.0//EN"
"http://www.jboss.org/j2ee/dtd/jboss_3_0.dtd">
<jboss>
   <enterprise-beans>
      <session>
         <ejb-name>HelloWorldServer</ejb-name><!--Matches ejb-jar.xml -->
         <jndi-name>ejb/HelloServer'</jndi-name>
      </session>
   </enterprise-beans>
</jboss>
```

And now we are ready to create our JAR file.

- Right-Click the HelloServer project and choose 'Export' from the contextual menu.
- Select 'JAR file' from the dialog and press 'Next>'.
- In the next dialog, name the export destination 'HelloServer.jar'. Make sure 'META-INF' is selected in the 'resources to export' list. Press 'Next>'.



- In the last dialog, check 'save the description of this jar in the workspace' and name the file '/HelloServer/Builddescription.jardesc'. Uncheck 'Export class files with compile errors'. Press 'Next>' and 'Finish'.
- Just double-click the newly generated file (builddescription.jardesc). It is standard XML that tells Eclipse how to make a Jar from our project.

Deploying the bean

If everything went well, we have a file called 'HelloServer.jar' in our workspace folder in the eclipse folder. This file has to be deployed by JBoss. One of the stronger features of JBoss is that it can deploy and undeploy files while it is running. You just have to drop the jar files in the JBoss deploy folder to get them deployed. Removing Jar files from that folder will undeploy them. To demonstrate this, we'll start JBoss and deploy our HelloServer bean.

- Select 'JBoss->Start JBoss' from the menu. (See the section about installing and running JBoss earlier in this document if the JBoss menu does not show).
- Now go to the finder, and locate the freshly created 'HelloServer.jar' file. Make sure the Eclipse console remains visible.
- Drop that file into the JBoss deploy directory. (/Applications/JBoss-3.0.3/server/default/deploy/. Keep an eye on the console in Eclipse.
- If everything went well, these lines will show up on the console (it might take some seconds before JBoss notices that something did change).

```
11:26:00,713 INFO  [MainDeployer] Starting deployment of package:
file:/Applications/jboss-3.0.3/server/default/deploy/HelloServer.jar
11:26:01,710 INFO  [EjbModule] Creating
11:26:01,777 INFO  [EjbModule] Deploying HelloWorldServer
11:26:01,823 INFO  [EjbModule] Created
11:26:01,826 INFO  [EjbModule] Starting
11:26:01,914 INFO  [EjbModule] Started
11:26:01,915 INFO  [MainDeployer] Deployed package: file:/Applications/jboss-
3.0.3/server/default/deploy/HelloServer.jar
```

Changing our client to use the bean

With our application server running and our bean deployed, all that is left to do for us is to update our client to get the string to display from the server. There for, we need to do a few things:
- Add all libraries (jars) that are needed to communicate with JBoss and to communicate with EJB's to our project.
- Adapt the client program to:
  - Make contact with the application server.
  - Get an instance of an object that implements the home interface of our bean from the Server (JBoss).
  - Ask that object to create an object implementing the remote interface of our bean.
  - And ask that object how we should say hello.
  - Display whatever that object said.

So first, we add the J2EE libraries to our project
- Right-click the 'HelloProject' project. Select 'Properties' from the contextual menu.
- Select 'Build Path' on the left, and open the tab named 'libraries'.
- Add your j2ee.jar to the library list (Via 'Add External Jars'). My j2ee.jar file is in /Applications/java/j2sdkee1.3.1/lib. The j2ee sdk can be downloaded from Suns Java website.

Next, we must add the JBoss client library and Log4j (used by JBoss, en not yet rolled into J2EE 1.3) to our project as well, as we should assume that the average client machine does not have JBoss installed:
- In the libraries path of the properties dialog, add the files '/Applications/jboss-3.0.3/client/jbossall-client.jar' and '/Applications/jboss-3.0.3/client/log4j.jar'

And finally, our client must know the interfaces it has to use.
- Open the tab named 'projects' in the properties dialog, section 'Build Path'.
- Check the 'HelloServer' project.

Note that it would have been better if we created 2 different project for our HelloServer project: one containing interfaces and one containing the implementation. That way, we could only link the interfaces into our client project. Now we linked both the interface and the implementation. Not that it hurts, but it is just more than is actually needed, and it just does not make any sense to split an application in multiple tiers if the client tier contains the server tier.

When all the libraries are added to our project, we can change our HelloClient.java program to connect to the server, to get the home interface, create the remote interface and sent the 'sayHello' message to our application server. Here is my implementation of the new client program:

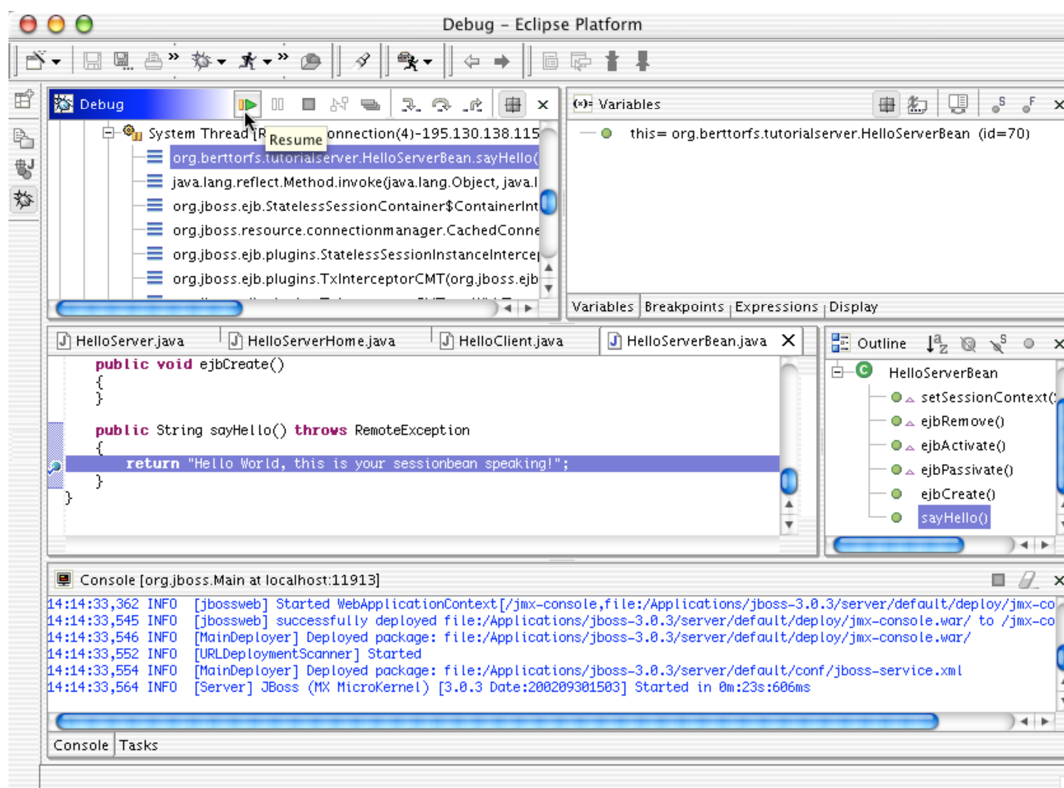
```
package org.berttorfs.TutorialClients;
import java.util.Properties;
import javax.naming.InitialContext;

import org.berttorfs.tutorialserver.HelloServer;
import org.berttorfs.tutorialserver.HelloServerHome;
```

```
/**
 * @author  bert
 *
 * To change this generated comment edit the template variable "typecomment":
 * Window>Preferences>Java>Templates.
 * To enable and disable the creation of type comments go to
 * Window>Preferences>Java>Code Generation.
 */
public class HelloClient
{
   public static void main(String[] args)
   {
      Properties props = new Properties();
      props.put("java.naming.factory.initial", "org.jnp.interfaces.NamingContextFactory");
      props.put("java.naming.factory.url.pkgs", "org.jboss.naming:org.jnp.interfaces");
      props.put("java.naming.provider.url", "localhost:1099");
      try
      {
         InitialContext ctx = new InitialContext(props);
         HelloServerHome theHelloHome = (HelloServerHome) ctx.lookup("ejb/HelloServer");
         HelloServer theHelloServer = (HelloServer) theHelloHome.create();
         System.out.println(theHelloServer.sayHello());
      }
      catch (Exception e)
      {
         e.printStackTrace();
      }
   }
}
```

Running and debugging.

Let us debug our client.

- Lets set a breakpoint in our HelloServerBean. To set a breakpoint, we double-click in margin of the editor window next to the line that says 'Return "Hello World, this is …"'. A blue dot should appear.
- Select the 'HelloClient' project again.
- Switch to the debug perspective by clicking the icon with the bug in the vertical toolbar on the left of the Eclipse window (selecting 'window->open perspective->debug from the menu will do the trick as well).
- With the debug perspective open, click on the small arrow pointing down next to the bug symbol in the horizontal toolbar on top of the Eclipse window.
- Select 'debug As…->Java Application' from the popup menu.
- The program will start running, some windows will be refreshed etc…, until de program arrives at our breakpoint, inside our Bean that is running inside JBoss. Exciting, no?
- Push the 'Resume' button at this point. Our HelloWorld message will display in the console, and our application will finish.



Accessing a database

Introduction

The j2ee environment contains entity beans to connect to and communicate with databases. An entity bean represents one table in a database. Reading from and writing to databases is done via the entity beans. The first implementation of entity beans forced one to communicate with these beans via a remote interfaces – as if they were running on a different machine. In most cases however, session beans and entity beans are all running on the same machine. Since version 2 of the EJB specifications, entity beans can be accessed via a local interface, which is much faster if your beans are all deployed on one server.

Entity beans are just an abstraction layer above your database system, nothing more, nothing less. They deal with input-output, transaction management etc…, not with business logic. Business logic should be implemented elsewhere. Maybe I add a section on business logic and how it could be implemented later to this article.

Entity beans are not that hard to do, but there are an awful lot of things to do to create them. Fortunately, most of this stuff is stupid repetitive typing work that can be automatically generated from our database scheme. And that is what we will do in this chapter. We will first download and install MySql and create a sample MySql database with some sample data. Next, we will download and install a JDBC driver for MySql. Then comes ANT, a free build tool required to pilot Middlegen. We will use Middlegen (yep, another download) to generate EJBs from the database structure and Xdoclet (no, not another download, as it comes bundled with Middlegen) to generate the necessary interfaces from tags set in the EJBs by Middlegen. Nice thing about Middlegen and Xdoclet is that they also generate sample JBoss.xml and ejb-jar.xml files, so we just have to copy and paste the additional XML in our current files to deploy the entity beans.

Once our entity beans are done, we will change our session bean (the HelloServer bean) to read from the database and send the result of the read back to the client. The client will the list whatever the server read from the database.

So lets start by setting up our database:

MySql

- Download the MacOsX build from http://www.MySql.org. (http://www.mysql.com/Downloads/MySQL-3.23/mysql-3.23.53-apple-darwin6.1-powerpc.tar.gz)

Normally, Stuffit expander will launch when the download finishes to untar the downloaded file. If you have expander 7.0 or later, this is Ok. Older Stuffit versions will create erroneous output, so be sure to have the latest free version. After the download, you should have a folder named mysql-3.23.53-apple-darwin6.1-powerpc.

- Move this folder to the applications directory, as that is where applications reside in OS X. (I do not like locations that are not visible from within the finder, like /usr/local/).
- Open a terminal and navigate to the MySql folder.

```
Prompt>Cd /applications
Prompt>Cd my<tab><enter>
```

- The <tab> stands for the tab-key. As the applications folder contains only one folder starting with 'my', typing <tab> will make the terminal finish the rest of the command by substituting 'my' with mysql-3.23.53-apple-darwin6.1-powerpc, saving you a lot of typing work.

If you never used MySql before, you must initialize some MySql system tables. This is done by executing a script that resides in the scripts folder inside the MySql folder. Just type

```
Prompt>./scripts/mysql_install_db
```

MySql will answer:

```
Preparing db table
Preparing host table
Preparing user table
Preparing func table
Preparing tables_priv table
Preparing columns_priv table
Installing all prepared tables
021031 18:47:42  ./bin/mysqld: Shutdown Complete
To start mysqld at boot time you have to copy support-files/mysql.server
to the right place for your system
PLEASE REMEMBER TO SET A PASSWORD FOR THE MySQL root USER !
This is done with:
./bin/mysqladmin -u root  password 'new-password'
./bin/mysqladmin -u root -h D5760ED6.kabel.telenet.be  password 'new-password'
See the manual for more instructions.
You can start the MySQL daemon with:
cd . ; ./bin/safe_mysqld &
You can test the MySQL daemon with the benchmarks in the 'sql-bench' directory:
cd sql-bench ; run-all-tests
Please report any problems with the ./bin/mysqlbug script!
The latest information about MySQL is available on the web at
http://www.mysql.com
Support MySQL by buying support/licenses at https://order.mysql.com
```

Now you can start MySql with the command

```
Prompt>./bin/safe_mysqld --user=mysql&
```

The & at the end of the command launches the command in batch mode so you can continue typing in commands in your terminal.
The previous command creates the mysql database which will hold all database privileges, the test database which you can use to test MySQL, and also privilege entries for the user that run mysql_install_db and a root user (without any passwords). It also starts the mysqld server.
Next we are going to do some tests:

```
Prompt>Cd bin
Prompt>./mysqladmin -u root drop test.
```

```
Prompt>./mysqladmin version
```

If everything was installed correctly, you'll get something like

```
./mysqladmin  Ver 8.23 Distrib 3.23.53, for apple-darwin6.1 on powerpc
Copyright (C) 2000 MySQL AB & MySQL Finland AB & TCX DataKonsult AB
This software comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to modify and redistribute it under the GPL license

Server version          3.23.53
Protocol version        10
Connection              Localhost via UNIX socket
UNIX socket             /tmp/mysql.sock
Uptime:                 5 min 2 sec

Threads: 1  Questions: 1  Slow queries: 0  Opens: 5  Flush tables: 1  Open tables: 0
Queries per second avg: 0.003
```

To terminate MySql, you type

```
Prompt>./mysqladmin -u root shutdown
```

Whenever you need MySql, you can start it from within the MySql folder with the command

```
Prompt>./bin/safe_mysqld
```

It is a good idea to set a root password for the database (not to be confused with your systems root password. In the command below, we assign a password 'verysecret' to the root account. When prompted for the current password, just press enter, as the root account has no password yet.

```
Prompt>Cd bin
Prompt>./mysqladmin -u root -p password verysecret
Enter password:<enter>
```

So far for installing and configuring MySql. We will now start an interactive MySql session and create a database with some tables. To keep everything simple, we will do that using the root account. So, from within the bin directory, we type

```
Prompt>./mysql -u root -p
Prompt>Enter password:
```

When prompted for a password, type whatever password you assigned (we used 'verysecret'). Whatever you type will not be shown, so somebody looking over your shoulder cannot steal your password. The system will respond with:

```
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 4 to server version: 3.23.53
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
```

To create a database, we type (the prompt now says MySql)

```
Mysql>Create database partdb;
Mysql>use partdb;
```

Then type the following

```
Mysql>Create table partgroup(
partgroupcode char(4) not null,
name varchar(50) not null,
```

```
primary key (partgroupcode)
);
Mysql>Create table part (
partID int not null,
partgroupcode char(4) not null,
name varchar(50) not null,
unitprice float not null,
picturename varchar(50),
primary key (partID)
);
```

Next, we insert some initial data:

```
Mysql>insert into partgroup(partgroupcode, name) values ('HARD', 'Hardware');
Mysql>insert into partgroup(partgroupcode, name) values ('SOFT', 'Software');
Mysql>Insert into part (partID, partgroupcode, name, unitprice, picturename) values (1,
'HARD', 'Apple I-Mac 15inc 700 Mhz', 1200, 'imac.jpg');
Mysql>Insert into part (partID, partgroupcode, name, unitprice, picturename) values (2,
'HARD', 'PowerMac G4 Dual 1.25 Ghz', 3500, 'pmac.jpg');
Mysql>Insert into part (partID, partgroupcode, name, unitprice, picturename) values (3,
'SOFT, 'Final Cut Pro', 1000, 'fcpro.jpg');
Mysql>Insert into part (partID, partgroupcode, name, unitprice, picturename) values (4,
'SOFT', 'DVD Studio Pro', 1000, 'idvd.jpg');
```

You exit your MySql session with

```
Mysql>exit;
```

This does not stop MySql. The database is still running, waiting for incoming connections. To shutdown, you should type, from within the MySql bin directory,

```
Prompt>./mysqladmin -u root -p shutdown
Enter password:
```

JDBC Driver for MySql :

This can be downloaded from www.mysql.com/download/api-jdbc.html. The file I downloaded was called 'download.php'. It is a zip file with the wrong extension. You can unzip it by dragging it over the 'Stuffit Expander' icon, or by changing its file extension to '.zip'. When you open the resulting folder (in my case, it was called download.php Folder), you will see a folder named mysql-connector-java.2.14. Open it, and copy the file named mysql-connector-java-2,0,15-bin.jar to /library/java/extension on your system disk. That way, it will be accessible to any java application on your system.

Download and install ANT

Middelgen, the software we will use to generate EJBs, requires ANT to pilot it. ANT is a - apologies for this over-simplification - kind of make tool. You have to provide it with an XML file (named Build.xml) that lists the different steps to execute, which files to read, which files to create etc… in order to do something. In case of Middlegen, the Build.xml file will tell Ant that it has to launch Middlegen, connect to a certain database, create EJBs for a given set of tables, place the output files in a specific location, launch Xdoclet using the output files created by Middlegen, create JBoss.xml and ejb-jar.xml and so on. Fortunately, Middlegen comes with and example build.xml file that only has to be modified slightly to work with our database.
I downloaded and unzipped http://jakarta.apache.org/builds/jakarta-ant/release/v1.5.1/bin/jakarta-ant-1.5.1-bin.zip.
Unzipping Ant (As always, using Expander 7.0 or later) will give you a folder called jakarte-ant-1.5.1, which you can safely move to your application folder.
In order for Ant to function, you must create an environment variable named ANT_HOME. In order to do so, I added the following lines to a file called '.tcshrc'.

```
setenv J2EE_HOME /applications/java/j2sdkee1.3.1
setenv ANT_HOME /applications/jakarta-ant-1.5.1
setenv PATH $J2EE_HOME/bin:$ANT_HOME/bin:$PATH
```

The file named '.tcshrc' is in your home folder (if it is not, you must create it). You'll need an editor that can read hidden files (like Metrowerks Codewarrior or BBedit or plain old VI - from the command line) in order to do so. If you use another shell, you probably know enough of Unix to know what file you must modify to customize your shell.

Middlegen : generate EJB

Another download is Middlegen. Download and unzip the binaries from http://boss.bekk.no/boss/middlegen/. This will give you a folder named Middlegen-2.0-b1. As always, I moved it to my applications folder.
In order to generate our EJBs, we first must adapt the sample build.xml file and a file that describes how to connect to our database (config/database/mysql.xml). When these modifications are done, we can generate our beans.

Adapting config/database/mysql.xml

The Middlegen folder contains a subfolder named samples, which contains a folder named config which contains a folder named database. Inside that folder, there is the mysql.xml file we need. Just save a copy of the original file to be sure. Then open mysql.xml and change it like this:

```
<!-- ================================================================== -->
<!-- ant properties/targets for mysql                                   -->
<!-- note: this is not a proper xml file (there is no root element)     -->
<!--       it is intended to be imported from a *real* xml file         -->
<!-- ================================================================== -->

<property name="database.script.file"           value="${src.dir}/sql/${name}-
mysql.sql"/>
<property name="database.driver.file"           value="/Library/Java/Extensions/mysql-
connector-java-2.0.14-bin.jar"/>
<property name="database.driver.classpath"      value="${database.driver.file}"/>
<property name="database.driver"                value="com.mysql.jdbc.Driver"/>
<property name="database.url"                   value="jdbc:mysql://localhost/partdb"/>
<property name="database.userid"                value="root"/>
<property name="database.password"              value="verysecret"/>
<property name="database.schema"                value=""/>
<property name="database.catalog"               value=""/>

   <property name="jboss.datasource.mapping"       value="mySQL"/>
```

What the values of database driver and database url mean and how the are assembled is described in the documentation that came with the ODBC driver we use.

Adapt Build.xml

The /Samples folder contains a sample build.xml file. We have to modify it slightly:
First of all, we change

```
<!DOCTYPE project [
    <!ENTITY database SYSTEM "file:./config/database/hsqldb.xml">
    <!ENTITY ejb      SYSTEM "file:./config/ejb/jboss.xml">
]>
```

to

```
<!DOCTYPE project [
    <!ENTITY database SYSTEM "file:./config/database/mysql.xml">
    <!ENTITY ejb      SYSTEM "file:./config/ejb/jboss.xml">
]>
```

This instructs Middlegen to use the info contained in mysql.xml – the file we just edited - to connect to our database. The next line, we change to

```
<project name="EJBTutorial" default="all" basedir=".">

   <property file="${basedir}/build.properties"/>
   <property name="name"                          value="org.berttorfs"/>
```

This just names our project. Next, we scroll down until we see

```
<!-- =================================================================== -->
<!-- Run Middlegen                                                       -->
<!-- =================================================================== -->
```

in that area, we alter the code so it looks like this. I put the things I modified in bold:

```
<!--
         We can specify what tables we want EJBs generated for.
         If none are specified, EJBs will be generated for all tables.
         Comment out the <table> elements if you want to generate for all tables.
         Also note that table names are CASE SENSITIVE for certain databases,
         so on e.g. Oracle you should specify table names in upper case.
         -->
         <!--table generate="true" name="persons" pktable="persons_pk"/>
         <table generate="true" name="flights" pktable="flights_pk"/>
         <table name="reservations"/-->

         <table generate="true" name="partgroup" pktable="partgroup_pk"/>
         <table generate="true" name="part" pktable="part_pk"/>

         <!--
         If you want m:n relations, they must be specified like this.
         Note that tables declare in multiple locations must all have
         the same value of the generate attribute.
         -->
         <!--many2many>
            <tablea generate="true" name="persons"/>
            <jointable name="reservations" generate="false"/>
            <tableb generate="true" name="flights"/>
         </many2many-->

         <cmp20
            destination="${build.gen-src.dir}"
            package="${name}.ejb"
            interfacepackage="${name}.interfaces"
            jndiprefix="${unique.name}"
            pkclass="true"
            dataobject="false"
            viewtype="local"
            mergedir="${basedir}/src/middlegen"
            readonly="false"
            fkcmp="true"
            guid="false"
         >
>
            <!-- Let the EJBs use The Sequence Block PK generation pattern◊
<!--        <sequenceblock
               blocksize="5"
               retrycount="2"
               table="SEQ_BLOCK"
```

```
              />
-->
```

next, we set the logical name of the database our EJBs have to connect to. This logical name will appear in the generated jbosscmp-jdbc.xml (see further), and will be linked with a physical database server via configuration files of JBoss later on:

```
<!-- ================================================================== -->
<!-- Run Ejbdoclet on Middlegen-generated sources                       -->
<!-- ================================================================== -->
<target
   name="ejbdoclet"
>
   <mkdir dir="${build.ejb-meta.dir}"/>

   <taskdef
      name="ejbdoclet"
      classname="xdoclet.modules.ejb.EjbDocletTask"
      classpathref="lib.class.path"
   />
.
.
.
<jboss
         version="3.0"
         xmlencoding="ISO-8859-1"
         destdir="${build.ejb-meta.dir}/META-INF"
         validatexml="true"
         datasource="java:/MySqlDS"
         datasourcemapping="${jboss.datasource.mapping}"
      />
   </ejbdoclet>
</target>
```

A last thing to modify is

```
<!-- ================================================================== -->
<!-- Run Ejbdoclet on Middlegen-generated sources                       -->
<!-- ================================================================== -->
<target
name="ejbdoclet"
>
```

(In the snippet above, I just removed the sentence 'depends="postprocess-ejb-jar"). This modification unlinks the generation of the entity beans from the generation of the interfaces and the xml. Middlegen will now stop after it generated the EJBs. We must launch it again with the target name 'ejbdoclet' in order to generate the interfaces and the xml files. I had to do this because, for one reason or another, the ANT job refuses to continue after the Middlegen step.

Generate EJB's

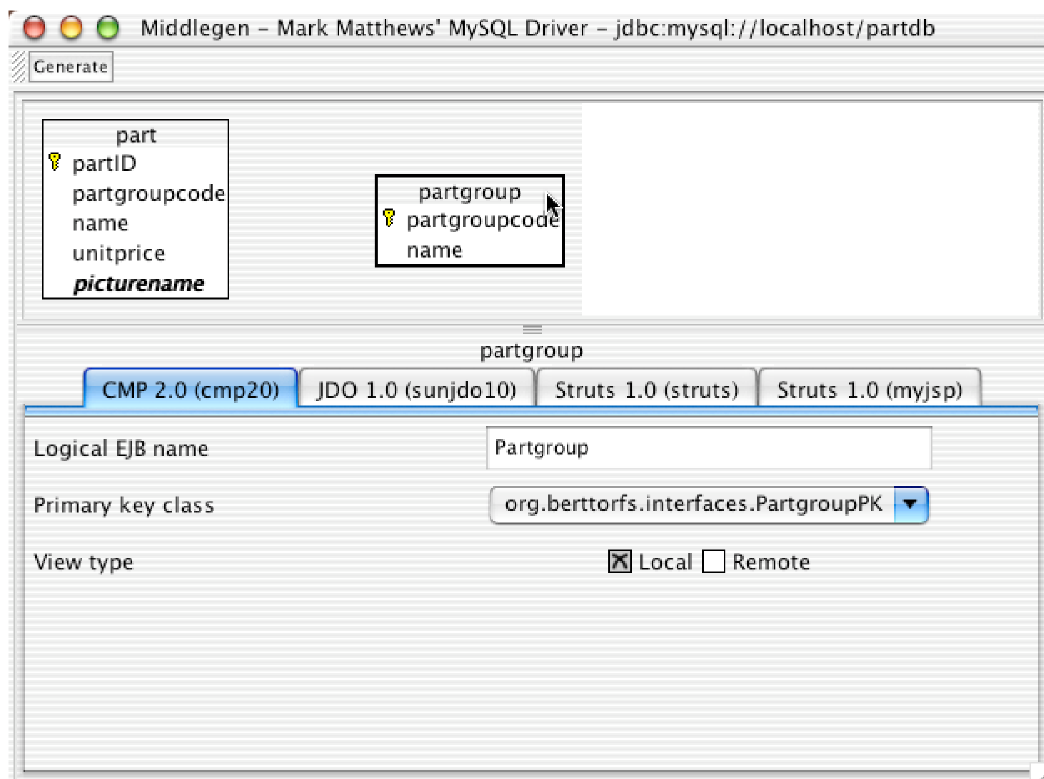
This is the easier step. Just navigate in the terminal to the place where your modified 'build.xml' resides and enter

```
Prompt> cd /applications/middlegen-2.0-b1/samples
```

Next run Ant to generate the EJBs

```
Prompt>ant middlegen
```

You will get a lot of text back from ant, and the following window will open (you might have to resize things a little, play with the split bar and shuffle the tables around to make it look like the screen dump below):



Click on each of the tables and verify if the checkbox 'remote' is not checked. As we will only access our database via its local interface, we do not want to generate remote interfaces. There are no relationships drawn between the tables neither. MySql currently does not support referential integrity on its native tables (the ones we created – it does offer some other table types that support referential integrity though. Just take a look at the manual if you want to know how to do it.).
When finished checking, just press 'Generate' and quit the GUI application.
Then run Ant again. This time, we will generate the necessary interfaces and some deployment descriptors.

```
Prompt>ant ejbdoclet
```

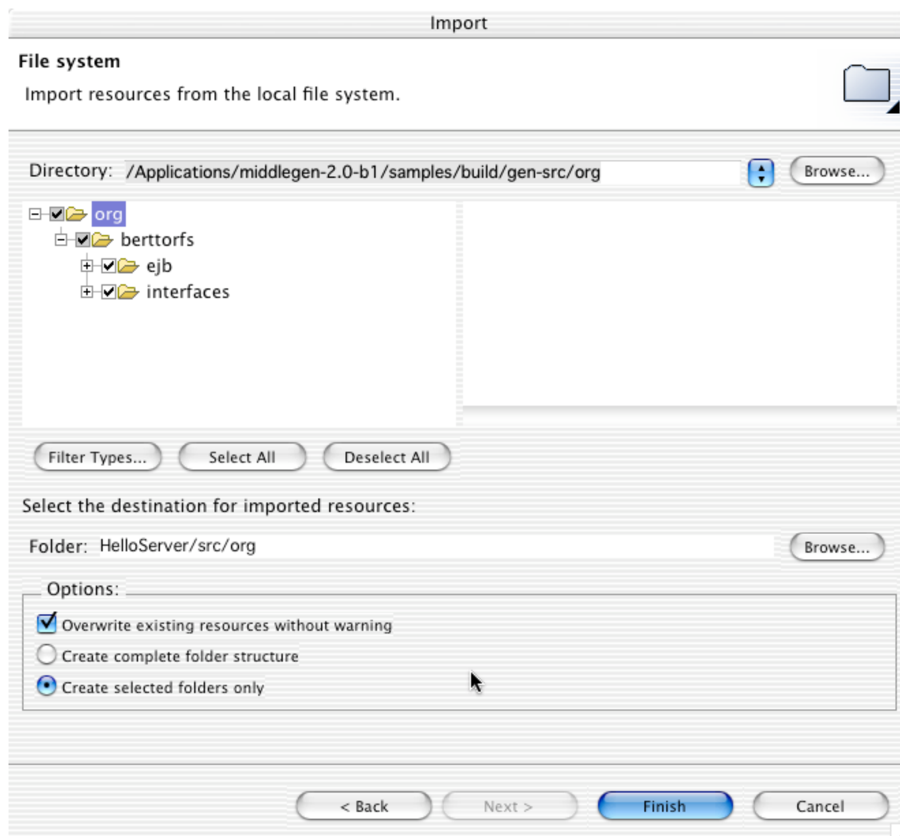
This will generate all interfaces and some utility classes. All generated classes are in the samples/build folder inside the Middlegen folder.

Back to eclipse : update the server application to read the database

The code for our entity beans is now generated and ready for use. We will now update our server application. Instead of saying 'Hello', it will now return a list of products (or parts) in our catalog. Later, we will modify our client to print that list.

Add the beans to the project

- Open Eclipse and select the project named HelloServer.
- Control-click and select 'import' from the contextual menu.
- Select 'File System' from the 'import source' list, and press 'Next>'.
- Enter (or browse) to '/applications/Middlegen-2.0-b1/samplesbuild/gen-src/org in the directory field and check the checkboxes next to 'ejb' and 'interfaces'.
- As the destination for imported sources, we enter (or browse) HelloServer/src/org. We leave everything else as is and press 'finish'.



Now our generated beans are part of our project, and we can start using them.

Modify the session bean to get its data from the entity beans

Up to now, all our session bean could do, was to return the string "Hello World, this is your sessionbean speaking!" via its method sayHello. We will now add another method called 'getPartCatalog' which returns an array of partCatalogEntries. These entries will be read from the database. We will

- Create a class 'partCatalogEntry' to hold our catalog entries. This class – or an array of it - will be passed between our sessionbean and a client somewhere on the network.
- We will add a method to our sessionbean that returns an array of partCatlogEntry objects
- We will write code for that method; It should get its data from the part and the partgroup table.
- We will modify our deployment descriptors.
- Next, we will deploy our new sessionbean to JBoss.
- And finally, we will adapt our client to call our new getPartCatalog method.

So lets go:

Create a class to hold a product catalog entry

Our session bean returns partCatalogEntries. We define these as a class:

- Right click the HelloServer project and select new->Java Class from the contextual menu (or new->Other, and select java class from the possibilities shown next).
- We will place the new class in the package 'org.berttorfs.interfaces, name it PartCatalogEntry, it inherits from java.lang.Object, and it implements java.io.Serializable – it has to, as it will be passed over a network to a remote client.
- Modify the code. Declare some variables:

```
public class PartCatalogEntry implements  Serializable
{
   private    Integer      id;
   private    String name;
   private    String category;
   private    double              unitPrice;
```

```
    private   String pictureName;
}
```

- Now select the 5 variables, right click and select 'source->generate getter and setter' from the contextual menu. Select all the proposed setters and getters in the window that shows next, and bingo, ready is your class! Save your file.

Add and implement 'getPartCatalog' in our sessionbean

Now lets change our session bean. We add a new method named 'getPartCatalog' to the beans implementation file HelloServerBean.java. Here is the code:

```
    public PartCatalogEntry[]    getPartCatalog(String     inPartGroup) throws RemoteException
    {
        Collection     thePartGroups;
        Collection     theParts;
        String  thisPartgroupCode;
        ArrayList      thePartCatalogEntries = new ArrayList();
        //PartGroupUtil was generated by middlegen. It contains all the code to get the local home
        // interface of our partGroup entity bean using JNDI on JBoss. Nice
        PartgroupLocalHome    thePartGroupHome = PartgroupUtil.getLocalHome();
        thePartGroups = thePartGroupHome.findAll();
        IteratortheIterator = thePartGroups.iterator();
        while (theIterator.hasNext())
        {
            PartgroupLocal thisPartGroup = (PartgroupLocal) theIterator.next();
            thisPartgroupCode = thisPartGroup.getPartgroupcode();
            //Now, use the partgroupcode to get all the parts belonging to this group
            PartLocalHome thePartHome = (PartLocalHome)PartUtil.getLocalHome();
            theParts = thePartHome.findByPartgroupcode(thisPartgroupCode);
            Iterator     thePartIterator = theParts.iterator();
            while (thePartIterator.hasNext())
            {
                PartLocal thisPart = (PartLocal)thePartIterator.next();
                thePartCatalogEntries.add(new PartCatalogEntry(  thisPart.getPartId(),
                                                                 thisPart.getName(),
                                                                 thisPartGroup.getName(),
                                                                 thisPart.getUnitprice(),
                                                                 thisPart.getPicturename()));
            }
        }

        return (PartCatalogEntry[]) thePartCatalogEntries.toArray(
                                      new PartCatalogEntry[thePartCatalogEntries.size()]);
    }
```

Of course, this will not compile. There are uncaught exceptions. To generate them automatically, we select the block of code where uncaught exceptions are thrown (this block start just below the comment on PartGroupUtil and continues until the return statement – the return statement not included) and right click. Select 'Source->surround with try-catch block' from the contextual menu. And Eclipse will do its magic again, adding the try, the braces and

```
        catch (NamingException e)
        {
        }
        catch (FinderException e)
        {
        }
```

No need to memorize all the exceptions thrown by the methods you call. Eclipse does it for you.

After having finished the bean, we must add our new method to the HelloServer interface as well (that is the remote interface).
Just add the following declaration:

```
public PartCatalogEntry[]   getPartCatalog(String    inPartGroup) throws RemoteException;
```

Change the deployment descriptors

To deploy entity beans, we need to add a lot of XML to our deployment descriptors. The description of an entity bean does not only list its interfaces and implementation, but also the different finder methods, all the fields that it contains, the relationships between the beans etc…. Fortunately, we do not have to write this file ourselves: it was generated by Middlegen. The same goes for Jboss.xml – which contains the JNDI-names of our beans – and jbosscmp-jdbc.xml – which maps bean names to table names, and which tells JBoss which database connection it has to get the data for our beans.

ejb-jar.xml

Open the file ejb-jar.xml that was generated by Middlegen (/applications/Middlegen-2.0-b1/samples/build/ejb-meta/META-INF/ejb-jar.xml and copy the complete section starting with <!-- Entity Beans --> and ending with the last </entity> (we have 2 entity beans, so the <entity> section appears twice.). Paste it in the ejb-jar.xml file of our project.

jboss.xml

Copy the tho <entity> sections into jboss.xml of our project. (after the </session> tag and before the </enterprise-beans> tag.

Jbosscmp-jdbc.xml

Create a new file named jbosscmp-jdbc.xml into the META-INF folder of our project. When it opens in the editor, just copy the contents of the generated file with the same name in it. Notice the first few lines:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE jbosscmp-jdbc PUBLIC "-//JBoss//DTD JBOSSCMP-JDBC 3.0//EN"
"http://www.jboss.org/j2ee/dtd/jbosscmp-jdbc_3_0.dtd">

<jbosscmp-jdbc>
   <defaults>
     <datasource>java:/MySqlDS</datasource>
     <datasource-mapping>mySQL</datasource-mapping>
   </defaults>
  <enterprise-beans>
      <entity>
         <ejb-name>Partgroup</ejb-name>
         <table-name>partgroup</table-name>
…
```

They tell that the jdbc-datastore to use for all entity beans that follow is called java:/MySqlDS. This datastore has to be defined somewhere to JBoss, so it can connect to the right database at runtime. That is what we will do in the next step.

Configure JBoss to connect to MySql

The file jbosscmp-jdbc tells JBoss we use a datastore named 'java:/MySqlDS for our entity beans. Where that datastore is physically stored, is described in an XML file that is deployed just like any other java application. An example of such a file can be found in /applications/jboss3.0.3/docs/examples/jca/mysql-service.xml. Copy this file to the desktop (or anywhere else) and open it with a text editor. Go to line 46 in that file (the line that reads

```
<attribute name="JndiName">MySqlDS</attribute>
```

and change the lines following this to

```
<attribute name="JndiName">MySqlDS</attribute>
```

```
  <attribute name="ManagedConnectionFactoryProperties">
   <properties>
    <config-property name="ConnectionURL"
type="java.lang.String">jdbc:mysql://localhost/partdb</config-property>
    <config-property name="DriverClass"
type="java.lang.String">com.mysql.jdbc.Driver</config-property>
    <!--set these only if you want only default logins, not through JAAS -->
    <config-property name="UserName" type="java.lang.String">root</config-property>
    <config-property name="Password" type="java.lang.String">verysecret</config-
property>
   </properties>
  </attribute>
```

Note that userId and password are hard coded in our deployment descriptors, which may not be the safest way to do so. But if our server is locked away in a computer room, only accessible by trusted people, it might not be as bad as it seems. When finished editing, copy the file to /applications/jboss3.0.3/server/default/deploy/.
A last file to modify is
In the file /applications/jboss-3.0.3/server/default/conf/standardjbosscmp-jdbc.xml
This file describes the default behaviour of our database connection. Open it and add change the defaults section as follows: (things I changed are bold).

```
<defaults>
      <datasource>java:/MySqlDS</datasource>
      <datasource-mapping>mySQL</datasource-mapping>

      <create-table>false</create-table>
      <remove-table>false</remove-table>
      <read-only>false</read-only>
      <time-out>300</time-out>
      <pk-constraint>true</pk-constraint>
      <fk-constraint>false</fk-constraint>
      <row-locking>true</row-locking>
      <preferred-relation-mapping>foreign-key</preferred-relation-mapping>
      <read-ahead>
      <strategy>on-load</strategy>
      <page-size>1000</page-size>
      <eager-load-group>*</eager-load-group>
      </read-ahead>
      <list-cache-max>1000</list-cache-max>
</defaults>
```

As we want our database to keep living without any application, we set the create-table and remove-table attributes to false. I set the row level locking to true as well.

Deploy the session bean

Go back to Eclips, right click the file named builddescription.jardesc and choose 'create jar' from the contextual menu. Next locate the newly created jar-file (in my case, it was in /applcications/eclipse/workspace/ and move it to the JBoss deploy folder. (/applcications/jboss3.0.3/server/default/deploy/).
If you want to receive some feedback from JBoss when you deploy your application, you can start JBoss from within Eclipse before deploying the application. JBoss will then report via the console on the progress and (hopefully) the success of the deployment.

Modify, run and debug the client.

Almost finished. Last thing to do is to adapt our client so it lists our part catalog in stead of the hello message. So navigate to the file 'HelloClient' inside the HelloProject project.
Currently, our HelloClient says hello on the line

```
System.out.println(theHelloServer.sayHello());
```

In stead of the method ‘sayHello’, we will now call update our source to call our servers method

```
public PartCatalogEntry[] getPartCatalog(String      inPartGroup) throws RemoteException
```

Our server does not use the argument named ‘inPartGroup’, so we could pass anything we want.
The main method of our updated client looks like this :

```
public static void main(String[] args)
{
   PartCatalogEntry[] thePartCatalog = null;
   Properties props = new Properties();
   props.put("java.naming.factory.initial", "org.jnp.interfaces.NamingContextFactory");
   props.put("java.naming.factory.url.pkgs", "org.jboss.naming:org.jnp.interfaces");
   props.put("java.naming.provider.url", "localhost:1099");
   try
   {
      InitialContext ctx = new InitialContext(props);
      HelloServerHome theHelloHome = (HelloServerHome) ctx.lookup("ejb/HelloServer");
      HelloServer theHelloServer = (HelloServer) theHelloHome.create();
      thePartCatalog = theHelloServer.getPartCatalog("dummy");
   }
   catch (Exception e)
   {
      e.printStackTrace();
   }
   for (int i = 0; i < thePartCatalog.length; i++)
   {
      System.out.println(thePartCatalog[i].getCategory() + "        " +
                         thePartCatalog[i].getName() + "          " +
                         thePartCatalog[i].getUnitPrice());
   }
}
```

Now run in, debug it (yes, you can debug your server code) and do anything with it you want.

JSP

Lets move from the server to the client now. Eclipse, together with some freely available plug-ins, allows us to write, debug and deploy web sites using J2EE technologies (servlets and Jsp, it is). To deploy J2EE web applications, we need a web server that is able to read, unzip, compile and execute servlets. To run jsp pages, we need a web server that can transform jsp pages into servlets, compile and execute them.
Although JBoss comes with Jetty installed (Jetty is such a web server), we will use a standalone Tomcat installation for our webserver. The reason for this is that I do not know of any Eclipse plug-in that lets you debug jsp pages when running Jetty within Jboss. I do know about a plug-in that lets you debug jsp pages when running with a standalone Tomcat. So that is what we will to.
Later, when we know how to make and debug simple JSP pages, we will go back to Jboss and make our JSP client communicate with our sessionbean.
Here is an outline of what we will be doing next:

- Downloads and installs:
  - Apache Tomcat
  - Sysdeo Tomcat plugin
  - Solar Eclipse
- Make and deploy a simple JSP page.
- Make and deploy a 2 page JSP application with a little controller servlet.
- Share data between these 2 pages using java beans

Required setup

Download and install Apache Tomcat

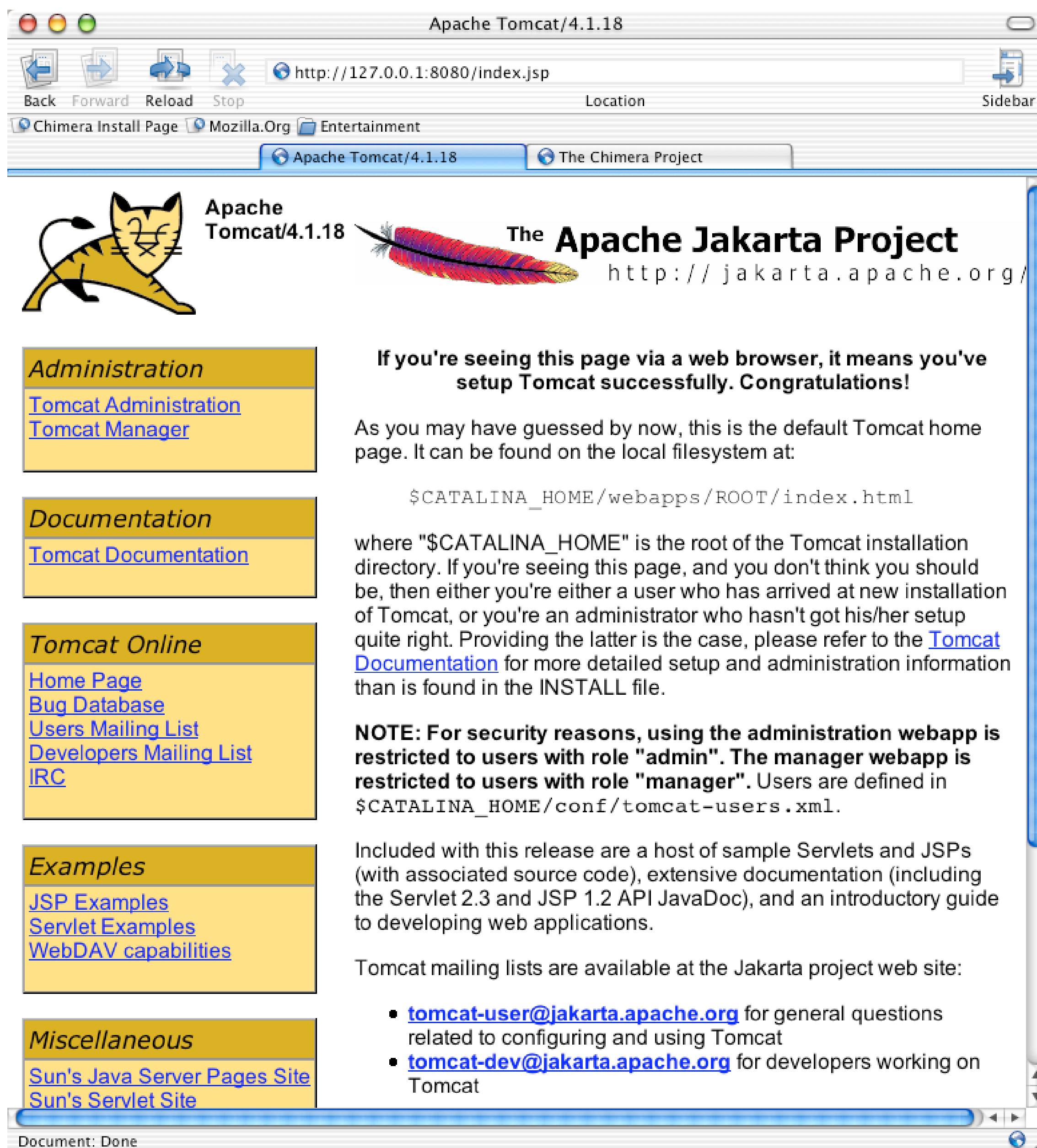
http://jakarta.apache.org/
I downloaded the file tomcat-4.1.18.zip and unzipped it into my Applications folder. And that is basically all there is to do. Be sure that the environment variable 'JAVA_HOME' is declared. I added the declaration of JAVA_HOME to my .tcshr file :

```
setenv JAVA_HOME /Library/Java/Home
```

The file named '.tcshrc' is in your home folder (if it is not, you must create it). You'll need an editor that can read hidden files (like Metrowerks Codewarrior or BBedit or plain old VI - from the command line) in order to do so. If you use another shell, you probably know enough of Unix to know what file you must modify to create environment variables.
To start Tomcat, open a terminal, navigate to applications/jakarta-tomcat-4.1.18/bin, and enter the following:

```
Prompt> cd applications/jakarta-tomcat-4.1.18/bin
Prompt> ./startup.sh
Using CATALINA_BASE:   /applications/jakarta-tomcat-4.1.18
Using CATALINA_HOME:   /applications/jakarta-tomcat-4.1.18
Using CATALINA_TMPDIR: /applications/jakarta-tomcat-4.1.18/temp
Using JAVA_HOME:       /Library/Java/Home
```

Next, you can test your installation by opening a web browser and navigate to http://localhost:8080/. (or http://127.0.0.1:8080/ if your browser says that dns failed to locate localhost, as my chimera did). If everything went well, you should see this:



If Tomcat fails to start, complaining that port 8080 is already occupied, you probably have Jboss still running. Jboss comes with Jetty, which is another Web server that is configured to use port 8080. So be sure that you shut down Jboss before starting Tomcat.
Once you have Tomcat working, shut it down, As we will be control it from within JBoss.

```
Prompt> cd applications/jakarta-tomcat-4.1.18/bin
Prompt> ./shutdown.sh
Using CATALINA_BASE:   /applications/jakarta-tomcat-4.1.18
Using CATALINA_HOME:   /applications/jakarta-tomcat-4.1.18
Using CATALINA_TMPDIR: /applications/jakarta-tomcat-4.1.18/temp
Using JAVA_HOME:       /Library/Java/Home
Prompt>
```

Download and install 'Sysdeo Eclipse Tomcat Launcher plugin'

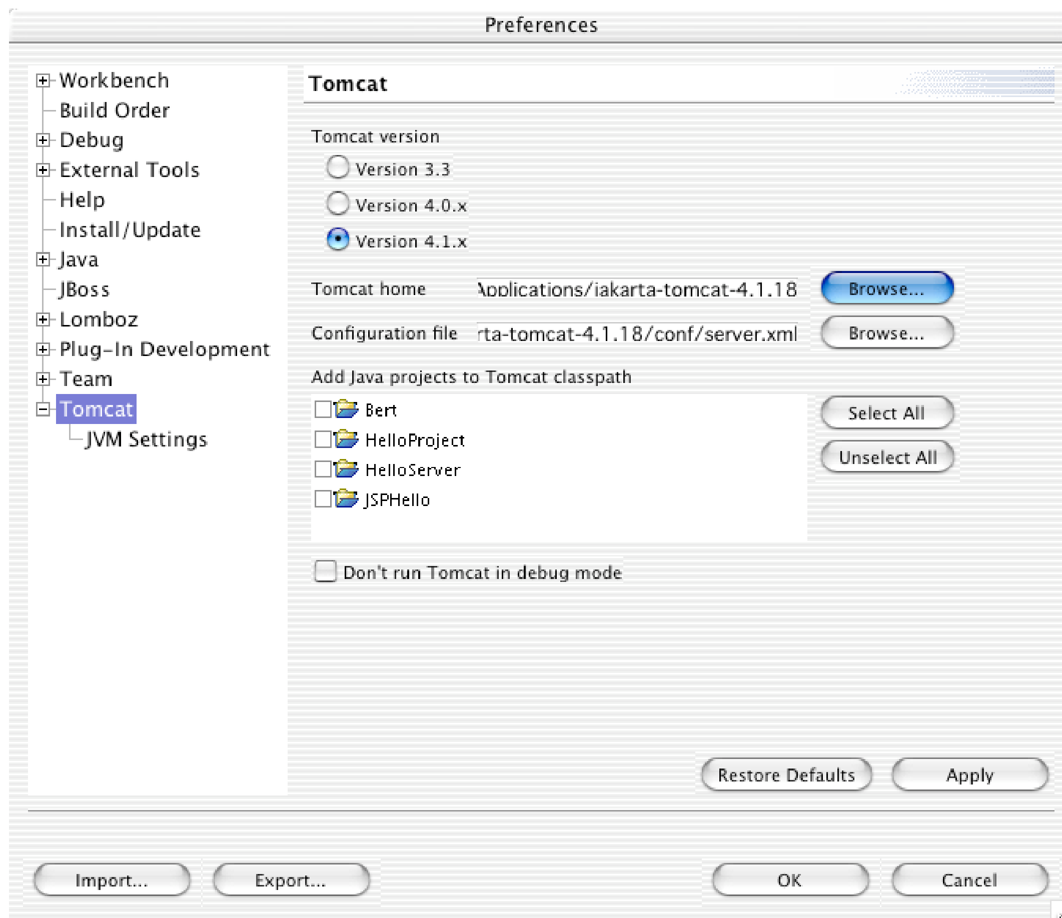
The 'Sysdeo Eclipse Tomcat Launcher plugin' (we 'll call it Sysdeo for the rest of this article) does to Tomcat what the EASY JBoss Plugin does to JBoss. It allows you to start and stop Tomcat from within JBoss, and to debug Servlets and Jsp pages. It also lends a hand when setting up a web application project in Eclipse. J2EE web applications do have a predefined structure. Sysdeo can set up this structure for you.

Download and install

You can download Sysdeo from http://www.sysdeo.com/eclipse/tomcatPlugin.html. I downloaded the file named tomcatPluginV201.zip. and unzipped it into the Eclipse plugin folder (Eclipse/Contents/Resources/Java/plugins). You open the Eclipse/Contents folder by <ctrl><Click>ing the Eclipse executable.

Setup

- Select (when being in resource perspective) 'Window->Customize Perspective' from the menu.
- Check 'Tomcat' in the category 'Others'.
- Repeat these steps for all the perspectives from where you want to start or stop Tomcat.
- Select the menu 'Window->Preferences' and tell Sysdeo where it can find Tomcat.



You should now have a new menu called 'Tomcat' from where you can start and stop Tomcat. (yes, you can try it now).

Download and install 'Solar Eclipse'

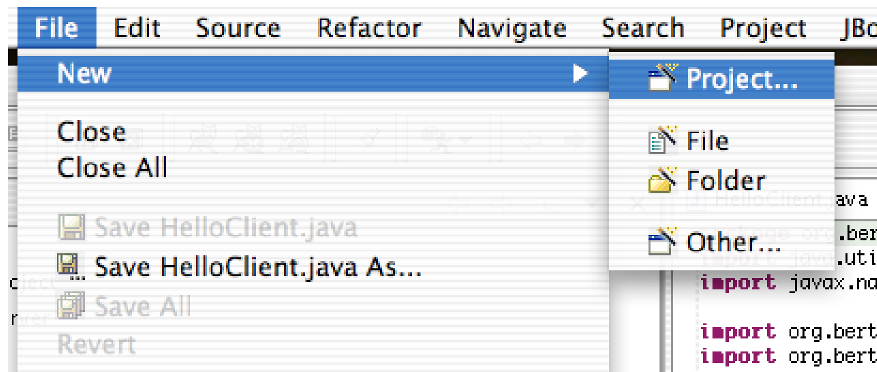
Solar eclipse is an Eclipse plug-in that does syntax highlighting for jsp and for XML files. You can download it from http://sourceforge.net/projects/solareclipse/. When using IE 5.2, the name of the download file gets truncated. (IE5.2 actually launches GraphicConvertor to handle the download on my Mac), so you must manually restore its full name (net.sf.solareclipse_0.3.1.bin.dist.zip) if you want to unzip it via a double-click.

To install Solar Eclipse, just unzip it and move the four plugin folders into the plugin folder of the Eclipse package. (<Ctrl-Click> on the eclipse icon and choose 'Show package contents' from the contextual menu to get into the Eclipse package). Alternatively you could download the Lomboz plugin. Jsp syntax colouring is just a very small part of its feature set.

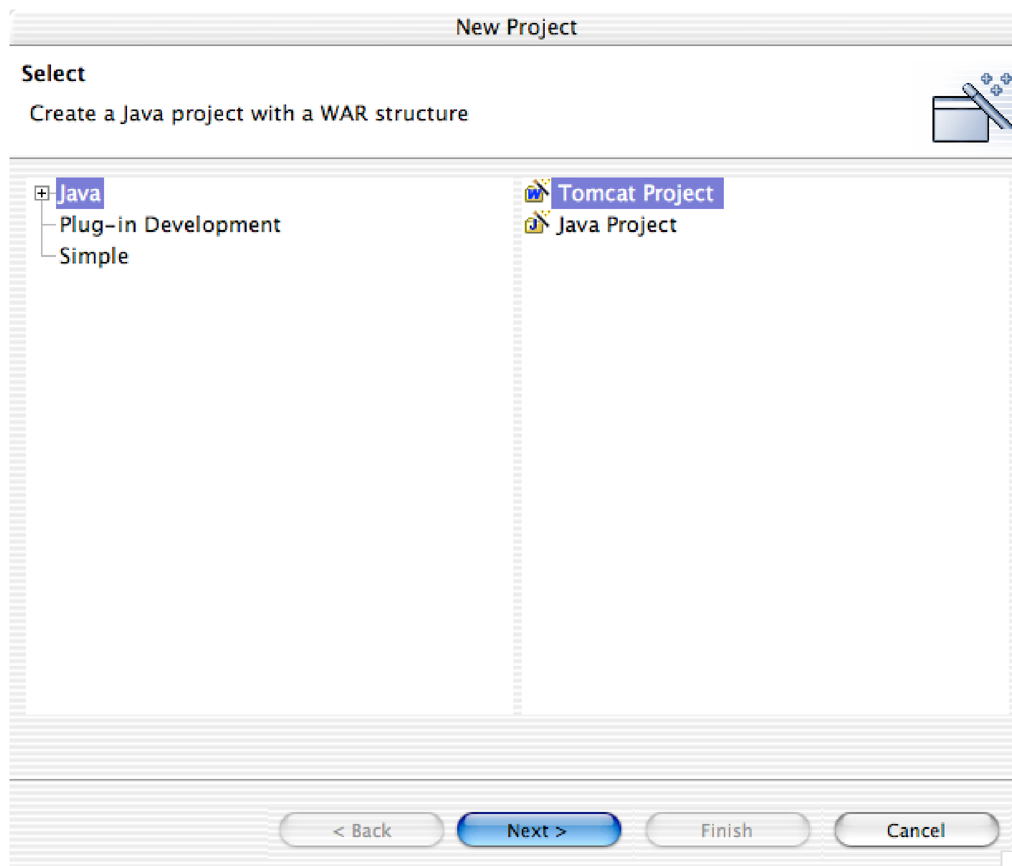
A Simple JSP example : WhatTimeIsIt.

The first JSP example is mainly copied from the Sysdeo documentation. You can read the original – in French – at the following address : http://www.eclipsetotale.com/articles/tomcat/tomcatPluginDocFR.html. Here we go :

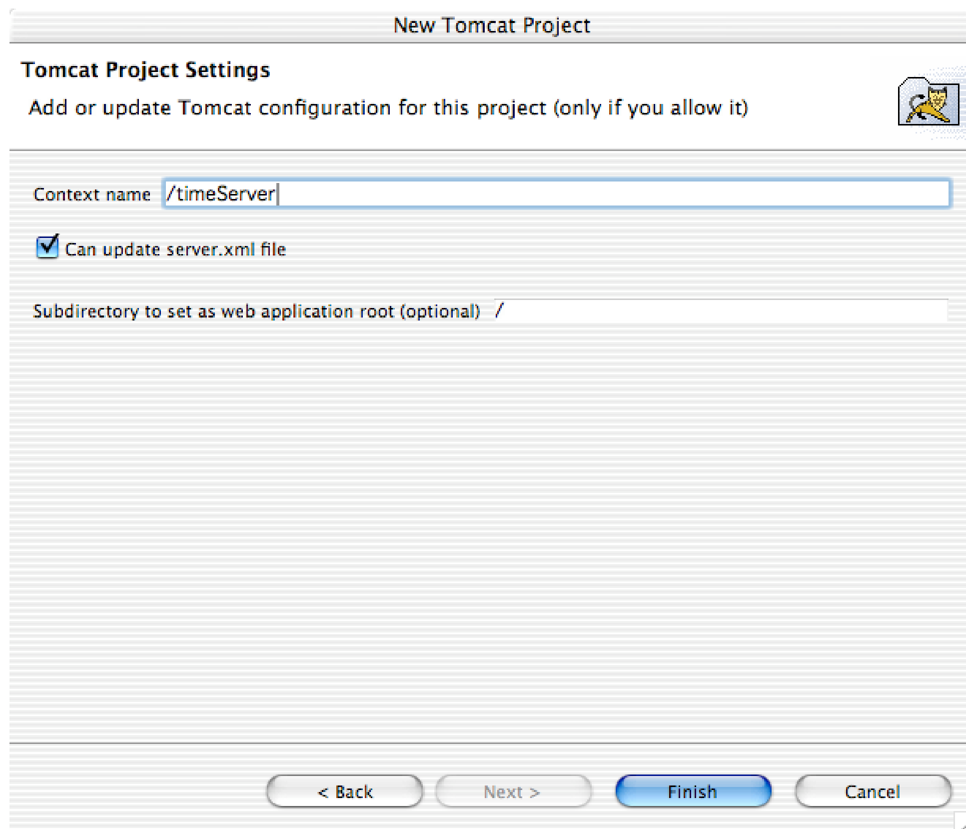
- Choose File->New Project



- Select 'Tomcat Project' from the dialog that opens next.



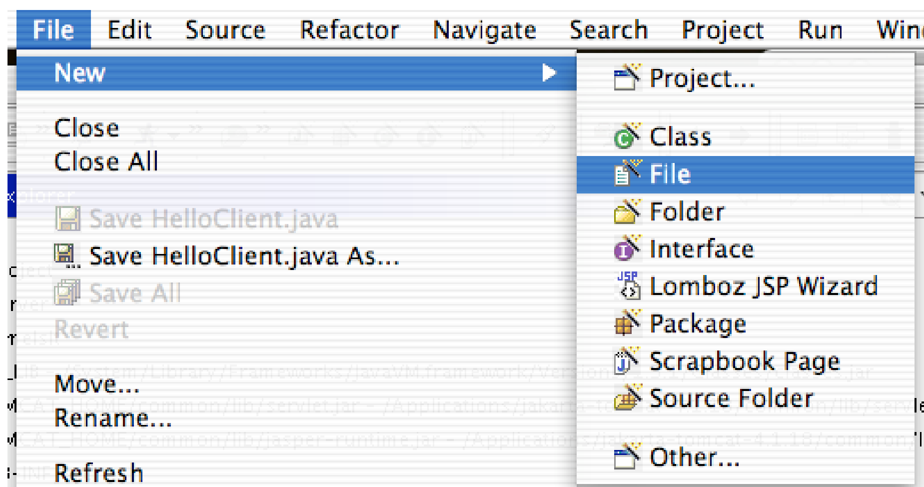
- Enter a name for your project (I called it WhatTimeIsIt) and press 'Next>'.
- The dialog that opens next opens only when you choose 'Tomcat Project' from the 'New' dialog. It is part of the Sysdeo plug-in. As the context name, I entered 'timeServer'. We will have to use that context name later to connect to the web application we are building. I also checked the 'Can update server.xml' checkbox. If you check this, the Sysdeo plug-in will make all the changes necessarily to deploy our project in Tomcat for us. If you are an experienced Tomcat administrator, you can leave this checkbox unchecked.



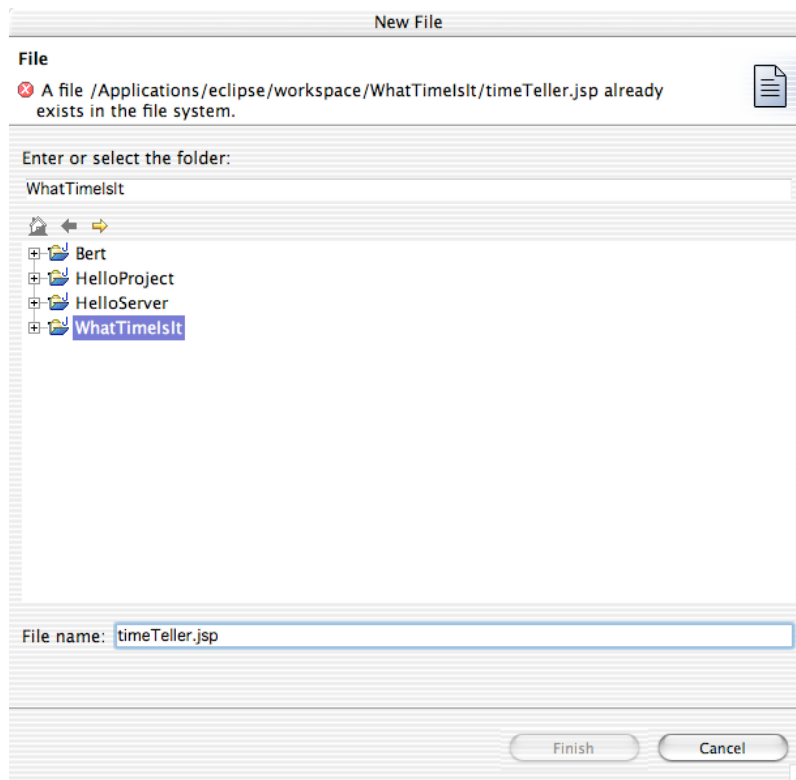
- Press 'Finish'.
- If you now look at the Server.xml configuration file of Tomcat, you will see that the line

```
<Context path="/timeServer" docBase="/Applications/eclipse/workspace/WhatTimeIsIt"
workDir="/Applications/eclipse/workspace/WhatTimeIsIt/work/org/apache/jsp" />
```

  was added just beforte the </Host></Engine></Service> closing tags. Sysdeo did this because you checked the 'Can update server.xml checkbox.
- Next, we create a simple JSP page. Choose <file->new->file> from the menu.



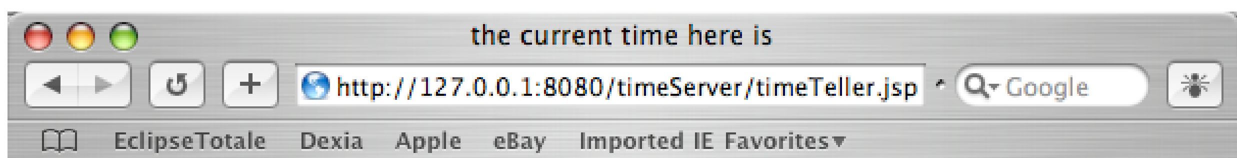
- Create the new file at the root of our new project. I named the new file 'timeTeller.jsp'.



- Press ‘Finish’. You will now the contents (empty) of timeTeller.jsp in the edit pane of Eclipse. Enter the following and save the file :

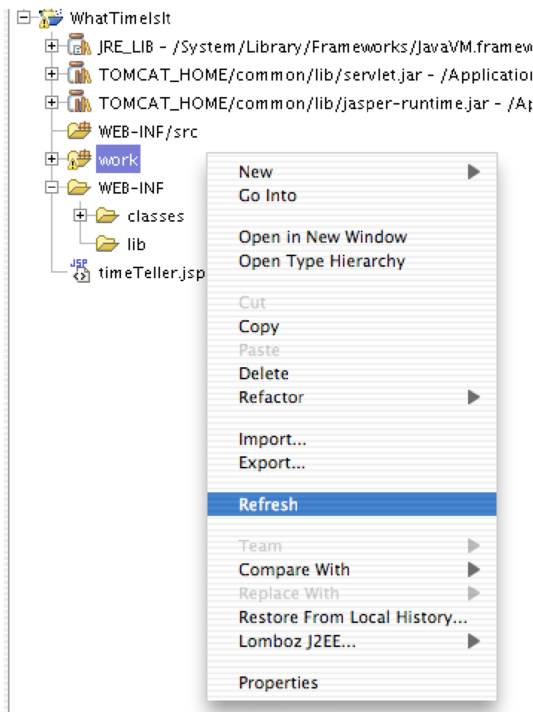
```
<%@ page info="Tells what time it is at the servers location" %>
<html>
<head><title>the current time here is</title></head>
<H1>
<%= (new java.util.Date()).toLocaleString() %>
</H1>
</html>
```

- Now start Tomcat
- And open the page http://127.0.0.1:8080/timeServer/timeTeller.jsp in your favorite browser. You should see the following:



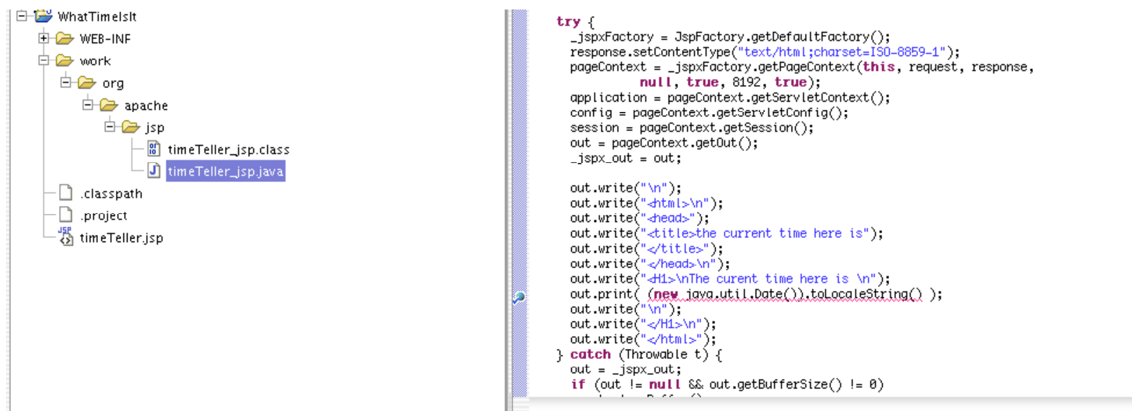
JSP pages are transformed into servlets before the are compiled and executed. So debugging JSP at the source level is not possible. If you use the Sysdeo pluging, you can debug the servlet code that was generated from the JSP page. To do so, you must follow the next steps :

- Select the work-directory of your project and select ‘refresh’ from the contextual menu



- Now navigate into the org.apache.jsp tree until you arrive at the file name ‘timeTeller_jsp.java. That file contains the servlet code that was generated from your jsp. You can put breakpoints in that file as you can do in any other java file. Just place a breakpoint before the line that says

```
out.print( (new java.util.Date()).toLocaleString() );
```



- Refresh the page in your browser…
- … and eclipse will switch to debug perspective and open the servlet code where you placed the breakpoint. Be sure to browse around a bit in the variables pane : it gives a fairly good idea on the internal structure and data of servlets.

Building and debugging a lot-of-tiers J2EE web application

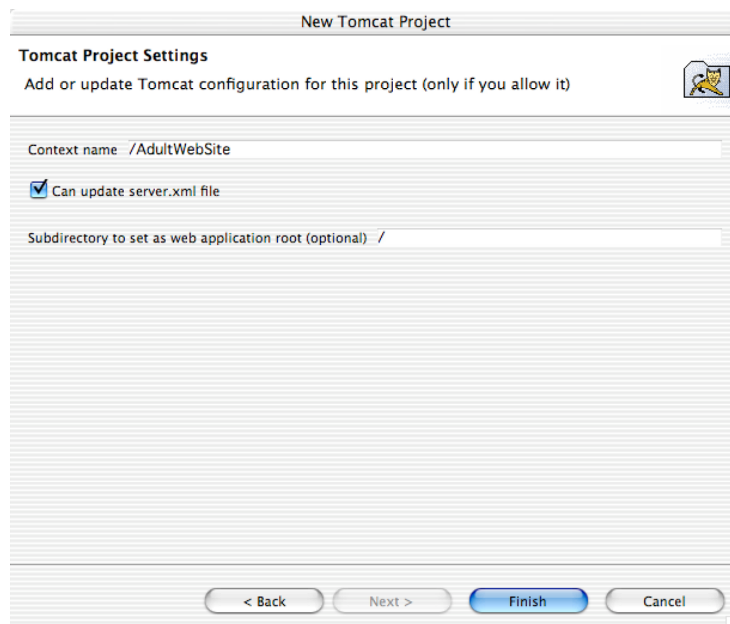
So we pretty much covered up the basic tools and techniques. Lets move on now and glue everything together.
Our first application will have a front end written in HTML/Jsp and use a controller servlet to navigate between the different pages. Later, we will add a model component that will implement some business intelligence.
Our example application is a simple one. A first page will ask for some personal information. Based on what we enter, you'll get a second page for adults, or one for children.

View-Controller (the model will come later)

In a first iteration of this example, we will create 4 objects and modify some XML.

- A file 'GetInfo.jsp' will ask some very personal details.
- A servlet 'AdultWebSiteController' will take a look at these details and open, after some investigation, a second page
- The page 'AdultPage.jsp' shows adult contents.
- The page 'ChildrenPage.jsp is intended for younger public.
- The Javabean 'PersonalInfo.java' will hold whatever the user entered.

So lets start by creating a new project. Create a new Tomcat project and name it AdultWebSite. To make our life easy, we let Eclipse update our Server.xml file



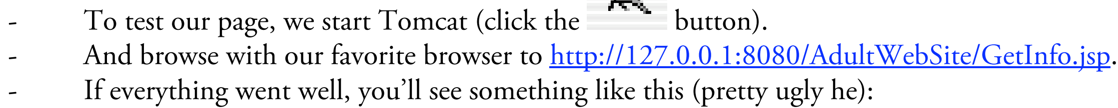
A first view

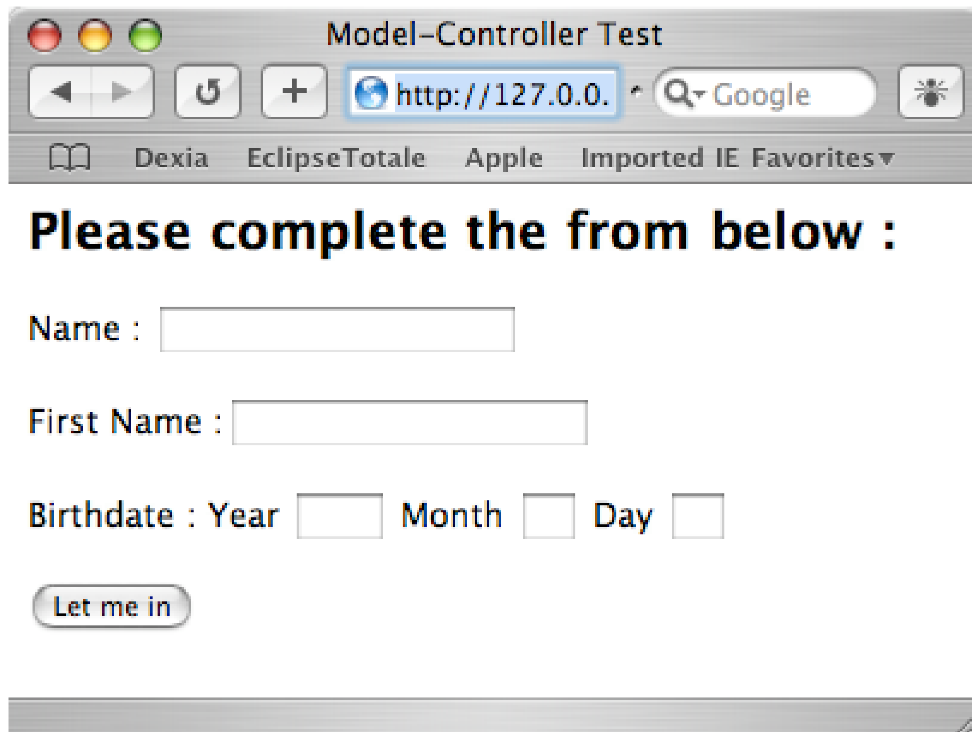
Once our project is created, we add the page 'GetInfo.jsp' at the root of our project.

- <Control-Click> the AdultWebSite project and choose 'New file' from the contextual menu.
- Choose the project root as the place to store the file, and enter 'GetInfo.jsp' as the name of the new file.
- The new file will be opened in the editor pane. Enter the following code :

```
<%@ page contentType="text/html;charset=windows-1252"%>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=windows-1252">
<title>Model-Controller Test
</title>
</head>
<body>
<h2>
Please complete the from below :
</h2>
<form action="/AdultWebSite/SiteController/WhichPage" name="sender"
```

```
method="post">
<p>
Name : <input type="text" name="Name" maxlength="50">
</p><p>
First Name :<input type="text" name="FirstName" maxlength="50">
</p><p>
Birthdate : Year <input type="text" name="year" size="4" maxlength="4">
Month <input type="text" name="month" size="2" maxlength="2">
Day <input type="text" name="day" size="2" maxlength="2">
</p>
<input type="submit" name="submit" value="Let me in">
</body>
</html>
```

- To test our page, we start Tomcat (click the  button).
- And browse with our favorite browser to http://127.0.0.1:8080/AdultWebSite/GetInfo.jsp.
- If everything went well, you'll see something like this (pretty ugly he):

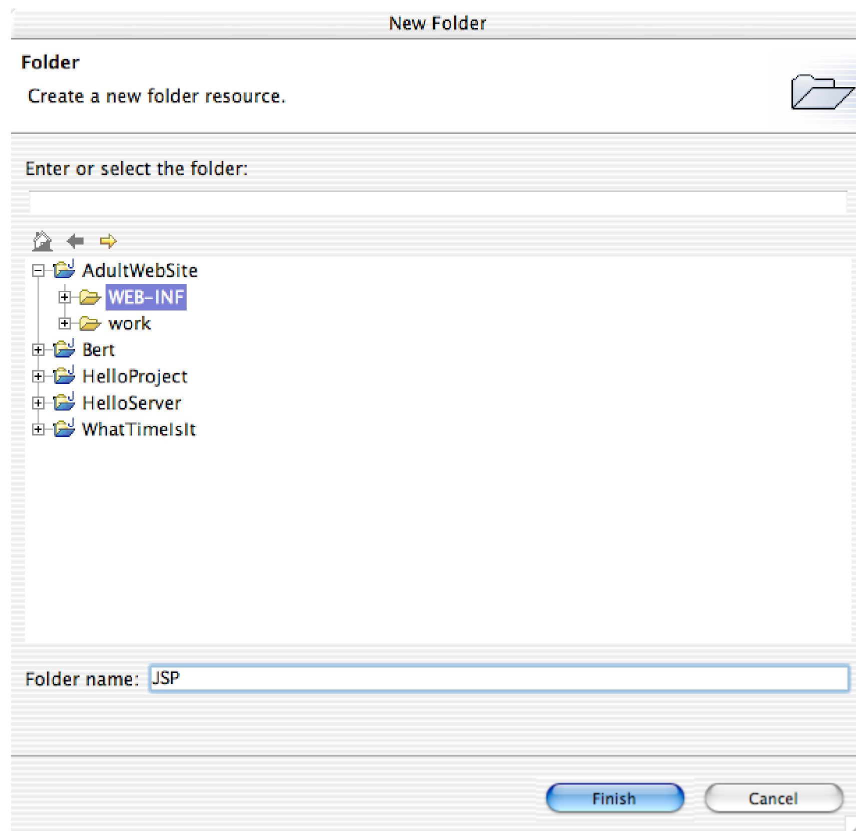


- Pressing the 'Let me in' button will bring you nowhere (to a Tomcat error page to be exact, which is close to nowhere).

Some More Views (we would not need a controller otherwise)

If you want to navigate between pages, you'll need at least two of them. Our example will have 4. We did the first one in the previous paragraph. The 3 other pages are listed below. We will create the first one together. For the others, only the source code is given

- <Cmd-click> on the WEB-INF folder and select <new->folder> from the contextual menu.
- Name the new folder JSP. JSP files placed inside the WEB-INF folders are not directly accessible from a web browser. So placing your files inside WEB-INF is a first-and-easy way to protect your site.



- Then <Ctrl-Click> the newly created JSP folder. Select <New->File> from the contextual menu and name the file Adults.jsp. Then put something like this in that file :

```
<%@ page contentType="text/html;charset=windows-1252"%>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=windows-1252">
<title>Model-Controller Test
</title>
</head>
<body>
<h2>
This page is only to be viewed by mentally healthy people over 18.
</h2>
<p>
</p>
</body>
</html>
```

- Save the file
- Create another file 'Children.jsp with the following contents

```
<%@ page contentType="text/html;charset=windows-1252"%>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=windows-1252">
<title>Model-Controller Test
</title>
</head>
<body>
<h2>
This is a page suited for children
</h2>
<p>
</p>
</body>
</html>
```

- And one last page named ‘InvalidInput.jsp’.

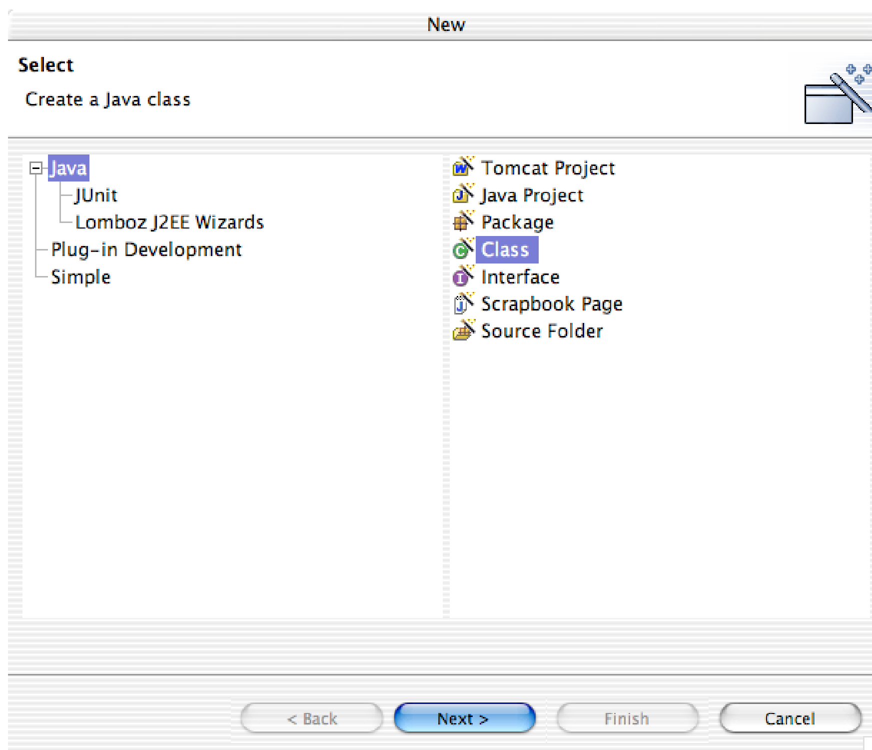
```
<%@ page contentType="text/html;charset=windows-1252"%>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=windows-1252">
<title>Model-Controller Test
</title>
</head>
<body>
<h2>
Whatever you entered was invalid or incomplete. I could not make out where to go.
</h2>
<p>
</p>
</body>
</html>
```

Then comes the controller

Make a first version of the controller servlet

Designers should make HTML pages. If not, the pages risk to look like the one above. The code to test if somebody is an adult should be written by a developer. Stuff written by designers and stuff written by developers should not be mixed – they do not match and are created with different tools. Mixing a complex HTML document with a complex Java program gives an overly complex jsp file that nobody understands. Please consult an article on Model-View-Controller if you want to learn more on separating interface and business logic.
So lets create a controller object. For faceless HTML communication, J2EE offers us Servlets. To add a servlet to our project, we right-click the AdultWebSite project, and select ‘New->Other…’.

- Select Java and Class. Press ‘Next’



- Give the class a name and a location as shown in the screenshot below. Make the class inherit from httpServlet.

New

Java Class

Create a new Java class.

Source Folder: AdultWebSite/WEB-INF/src Browse...

Package: org.BertTorfs.AdultWebSite Browse...

Enclosing type: Browse...

Name: AdultWebSiteController

Modifiers: public default private protected

abstract final static

Superclass: javax.servlet.http.HttpServlet Browse...

Interfaces: Add...

Remove

Which method stubs would you like to create?

public static void main(String[] args)

Constructors from superclass

Inherited abstract methods

< Back Next > Finish Cancel

- And press ‘Finish’.
- The following file will be created :

```
package org.BertTorfs.AdultWebSite;
import javax.servlet.http.HttpServlet;
/**
 * @author bert
 *
 * To change this generated comment edit the template variable "typecomment":
 * Window>Preferences>Java>Templates.
 * To enable and disable the creation of type comments go to
 * Window>Preferences>Java>Code Generation.
 */
public class AdultWebSiteController extends HttpServlet
{
}
```

- Just add the following in between the two brackets:

```
   protected void doPost(HttpServletRequest arg0,HttpServletResponse arg1)      throws
ServletException, IOException
   {
   }
```

- Then place the caret just afer ‘HttpServletRequest’ and press <Command-Space>. Do the same with the insertion point after HttpServletResponse, ServletException and IOException. This will make Eclipse add the necessary imports to the file.
- Copy the ‘doPost’ method and past it in again. Name the new method ‘doGet’.
- Paste again, and name the new method doAnything. Foreward both toGet and doPost to doAnything. Output some http in the ‘doAnything’ method. Here is what my code looks like:

```
   protected void doPost(HttpServletRequest arg0,HttpServletResponse arg1)
     throws ServletException, IOException
   {
        doAnything(arg0, arg1);
   }
```

```
protected void doGet(HttpServletRequest arg0,HttpServletResponse arg1)
   throws ServletException, IOException
{
     doAnything(arg0, arg1);
}

protected void doAnything(HttpServletRequest request,HttpServletResponse response)
   throws ServletException, IOException
{
   response.setContentType( "text/html" );
   PrintWriter out = response.getWriter();
   out.println( "<html>" );
   out.println( "<head>" );
   out.println( "<title>A Sample Servlet</title>" );
   out.println( "</head>" );
   out.println( "<body>" );
   out.println( "<h1>A Sample Servlet</h1>" );
   out.println( "</body>" );
   out.println( "</html>" );
   out.close();
}
```

- Eclipse will probably complain about PrintWriter. Just place the insertion point after “PrintWriter” and press <cmd-space> to add the necessary imports.

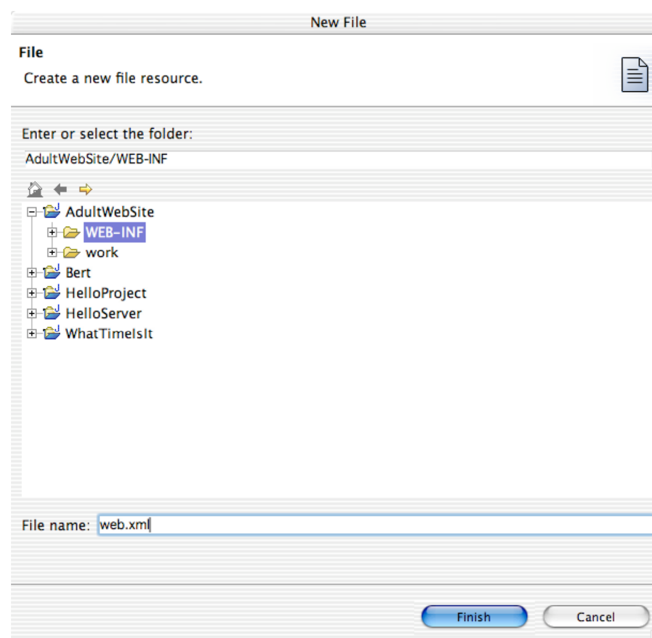
Route the action to the servlet

Out welcome page contains the following <form> tag.

```
<form action="/AdultWebSite/SiteController/WhichPage" name="sender"
method="post">
```

Whenever we push the submit button, the URL http://127.0.0.1//AdultWebSite/SiteController/WhichPage will be called. If we want this URL to point to our servlets ‘doPost’ method, we have to add a web.xml file to our project.
There is one already one Web.xml in the Tomcat folder (tomcat/conf/web.xml). That file contains already some default servlet declarations with their mappings. We have to add our own. Project (or application) specific web-xml files are stored in the WEB-INF folder of the application (or project).

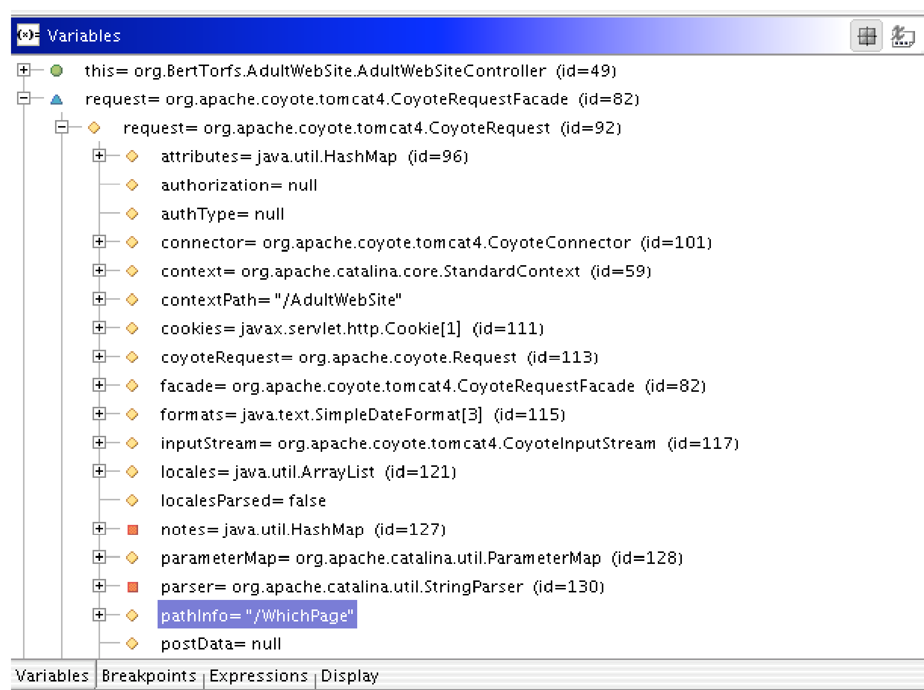
- <Ctrl-Click> (or right-click if you have a multi-button mouse) the WEB-INF folder of the project and choose <New->File> from the menu.
- Name the new file web.xml and click ‘Finish’.

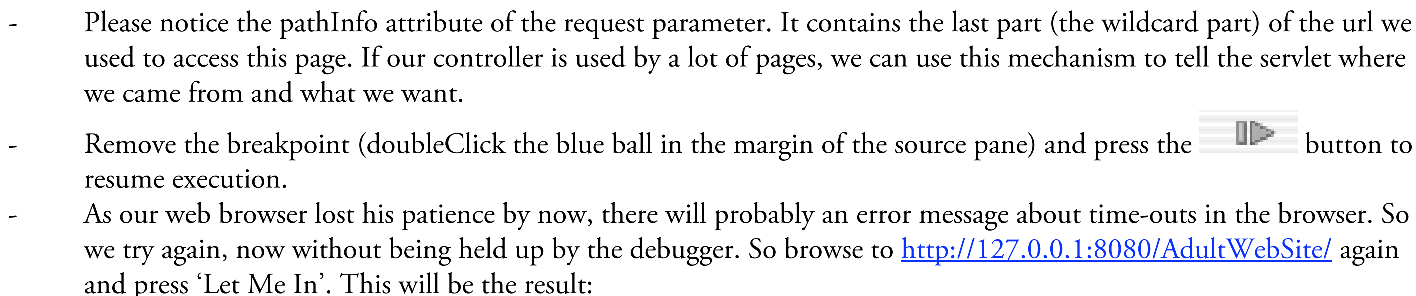


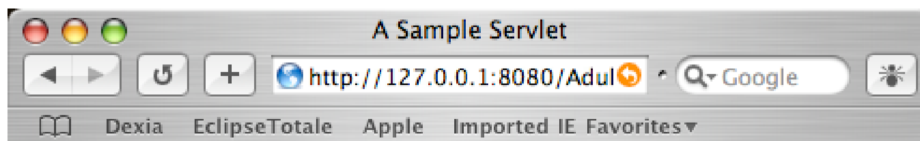
- Then edit the file and add the following XML (I copied the initial version from the web.xml file from the conf directory).

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
   <servlet>
    <servlet-name>Process</servlet-name>
    <servlet-class>org.BertTorfs.AdultWebSite.AdultWebSiteController</servlet-class>
    <init-param>
      <param-name>debug</param-name>
      <param-value>2</param-value>
    </init-param>
  </servlet>

  <servlet-mapping>
    <servlet-name>Process</servlet-name>
    <url-pattern>/SiteController/*</url-pattern>
  </servlet-mapping>
  <welcome-file-list>
   <welcome-file>GetInfo.jsp</welcome-file>
  </welcome-file-list>
</web-app>
```

- The first <servlet> tag and its matching <servlet-mapping> tag links the servlet named org.BertTorfs.AdultWebSite.AdultWebSiteController to the URL http://machine:Port/AdultWebSite/SiteController/*. The '*' can be anything, so we can have multiple URLs that all go to our controller servlet. You do not have to specify the context root (/AdultWebSite) in the servlet mapping.
- We also added an entry to the welcome-file list. That way, just typing AdultWebSite will bring us to our opening page.
- Now put a breakpoint inside the 'doAnything' method of the servlet, fire up Tomcat, and open your favorite browser on the page http://127.0.0.1:8080/AdultWebSite/ (I am behind a proxy, so localhost does not work for me. Your mileage may vary). You can omit the 'index.jsp' from the URL, as that was specified as the welcome-file.
- When the page opens, press the 'Let me in' button.
- Your servlet will be opened in the source pane with the line where you placed a breakpoint selected. As your debugging holds up the execution of the request, chances are that your browser will give an error, telling that your site could not be opened in a reasonable time.
- Open the variables pane of the debug perspective, and click on the small '+' next to 'Request'. You'll see something like this :



- Please notice the pathInfo attribute of the request parameter. It contains the last part (the wildcard part) of the url we used to access this page. If our controller is used by a lot of pages, we can use this mechanism to tell the servlet where we came from and what we want.
- Remove the breakpoint (doubleClick the blue ball in the margin of the source pane) and press the  button to resume execution.
- As our web browser lost his patience by now, there will probably an error message about time-outs in the browser. So we try again, now without being held up by the debugger. So browse to http://127.0.0.1:8080/AdultWebSite/ again and press 'Let Me In'. This will be the result:



- Just for fun, try to go to http://127.0.0.1:8080/AdultWebSite/WEB-INF/JSP/Adults.jsp. You'll get an error message from Tomcat. Pages placed inside the WEB-INF folder cannot be viewed or downloaded from a browser. Only you can serve them, no one can get them. People on the web can never look at the source code of your JSP's when they are placed inside WEB-INF. Basic security at no cost!

Make the servlet open the other pages

Now there is one more thing to do. In stead the 'A sample servlet' message, we want to see actual adult content if we are over 18. So lets remove whatever we wrote in the servlets doAnything method, and make it control the other jps's. Here is what you shoud do :

- Add another method – doForeward - to the servlet. Here is the code :

```
protected void doForward(String forward, HttpServletRequest req,
                         HttpServletResponse resp)
 {
    RequestDispatcher rd = getServletContext().getRequestDispatcher(forward);
    // Delegate the processing of this request
    try
    {
       rd.forward(req, resp);
    }
    catch (ServletException e)
    {
    }
    catch (IOException e)
    {
    }
  }
```

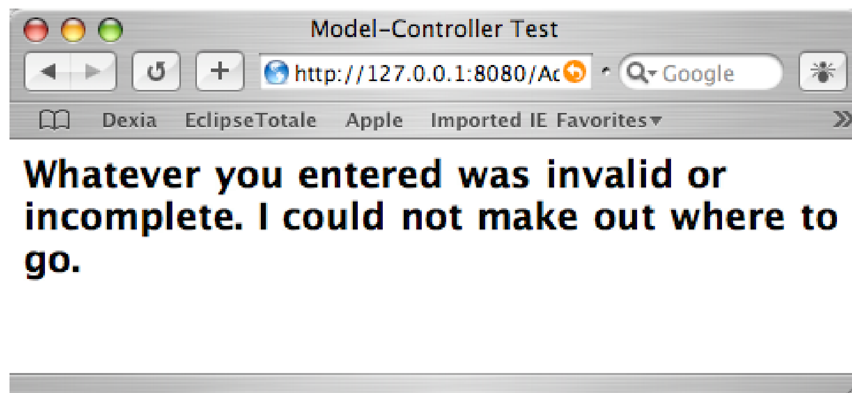
This method basically opens the JSP page whose name and path is passed as a string via the foreward parameter. Please ignore the non-existent exception handling – we should give the user a descend error message in stead of a stack trace.

- Next, we enter the following in the 'doAnything' method

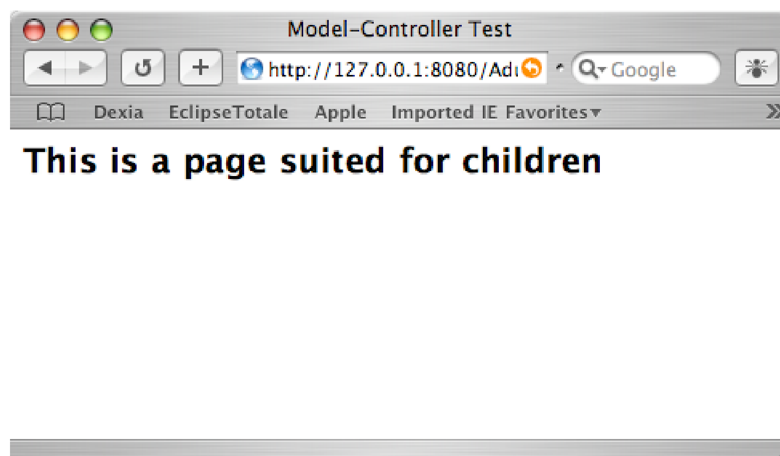
```
protected void doAnything(HttpServletRequest request,HttpServletResponse response)
throws ServletException, IOException
{
   String ls_year = (String) request.getParameter("year");
   String ls_action = (String) request.getPathInfo();
   String ls_NextPage;
```

```
    ls_NextPage = null;

    if (ls_action.compareToIgnoreCase("/WhichPage".toString()) == 0)
    {
       long ll_year;
       try
       {
          ll_year = Long.parseLong(ls_year);
          if (ll_year <= 1985)
          {
             ls_NextPage = new String("/WEB-INF/JSP/Adults.jsp");
          }
          else
          {
             ls_NextPage = new String("/WEB-INF/JSP/Children.jsp");
          }
        }
       catch (NumberFormatException e)
       {
          ls_NextPage = new String("/WEB-INF/JSP/InvalidInput.jsp");
       }
    }
    if (ls_NextPage == null)
       ls_NextPage = new String("/WEB-INF/JSP/InvalidInput.jsp");
    doForward(ls_NextPage, request, response);
 }
```

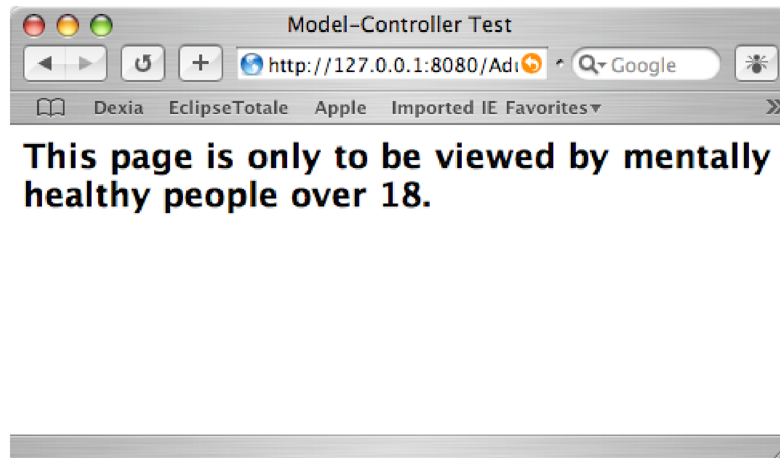
- And test the page. Start Tomcat, open the URL http://127.0.0.1:8080/AdultWebSite/, and press the 'Let me In' button without entering anything. You should see this :



- Go back, and enter 2002 as the year of birth. You should see this



- Go back, enter 1875 as the Birthdates year, and you'll get this:



Add the Model

(**to be done** – we will create a session bean with one method 'isAdult(birthday)' that will be used by the controller servlet)

Store and get preferences and settings in XML files

To be done. The legal age to be considered 'adult' will be read from an XML file. So will the names of the JSP pages our controller forwards to.

What's next

Next comes logging, Strut, writing an eclipse plugin that calls Middlegen straight from within a database browser running in Eclipse, communicating with clients running on Palm or Symbian devices over Bluetooth etc…. When time allows, I'll complement – or write new articles – on the topics described above. If all of you register a copy of WhereDidAllMyMoneyGo?, I could start writing tomorrow :). Otherwise, you will need some more patience.

Bert Torfs