Building J2EE applications with Eclipse Web Tools

Servers, Dynamic Web Projects, and Servlets

WTP uses the term *dynamic Web project* to describe projects that let you develop Web applications that make use of a middle tier *Web application server* . At present WTP includes support for J2EE Web application servers in the JST subproject. We will:

1 add Apache Tomcat to your workspace,
2 create a dynamic Web project that uses Tomcat, and
3 develop a servlet that dynamically generates HTML using server-side XSLT.

Servers

The main feature that distinguishes Web applications from ordinary Web sites is that Web applications generate dynamic content. Rather than seeing unchanging content on Web pages, users sees content that changes in response to their requests. Web application servers are Web servers that have been extended with additional capabilities for hosting Web applications. Although Web servers have almost always supported the generation of dynamic content through technologies such as server-side includes and Common Gateway Interface (CGI) scripts, Web application servers go above and beyond ordinary Web servers by providing additional services for hosting and managing applications. Web applications become first-class objects that can be configured, deployed, started, and stopped. One of the most important application services provided by Web application servers is session management which layers the notion of sessions on top of the inherently sessionless HTTP.

Shortly after Netscape imbedded a Java virtual machine in its Web browser to support applets, they proposed Java servlets as a superior alternative to CGI. Servlets had the advantage of using threads instead of the more costly proceseses used by CGI. Servlet are what really started the use of server-side Java, which has become the sweet spot for Java development. Sun then standardized the API for Java servlets and added them to J2EE. In J2EE, the presentation functions are hosted in a *Web container* or, as it is sometimes called, a *servlet engine* . Sun provided an initial implementation of servlets in the Java Servlet Development Kit which they subsequently contributed to the Apache Jakarta project. This contribution resulted in the Tomcat servlet engine. Although there are many other servlet engines available now, Tomcat remains very popular and you'll be using it here.

WTP extends Eclipse with *server runtime environments* which are similar in spirit to the familiar *Java runtime environments* supported by JDT. Just as with JST you can select a Java main class and run it as a Java application using an installed Java runtime environment, with WTP you can select Web resources, such as HTML, JSP, and servlets, and run them on an installed server runtime enviroment. The WTP concept of server is not restricted to Web resources though. For example, database servers could be treated this way too. It would make perfect sense to select a Java stored procedure class and run it on a database server.

Using a server with WTP is a three-step process:

1 Obtain and install the server runtime enviroment.
2 Add the server runtime enviroment to your workspace.
3 Create a server configuration, and add dynamic Web projects to it.

First, you must obtain and install the server runtime environment. Like Eclipse, WTP does not include any runtimes. You must obtain the server runtime from elsewhere and install it on your machine. WTP does include an extension point that server providers can use to simplify the process of installing the runtime.

Second, you add the server runtime enviroment to your workspace. To add the server runtime environment, you need a *server adapter* for it, which is a special plug-in that lets you control a server using the server tools provided by WTP. WTP comes with a respectable list of server adapters and you can obtain others from commercial vendors and other Open Source projects. WTP includes an extension point where other server adapter providers can advertise the availability of server adapters and have them added to your Eclipse installation. Configuring the adapter involves telling WTP where to find the server runtime installation, and setting other parameters, for example, what JVM to use.

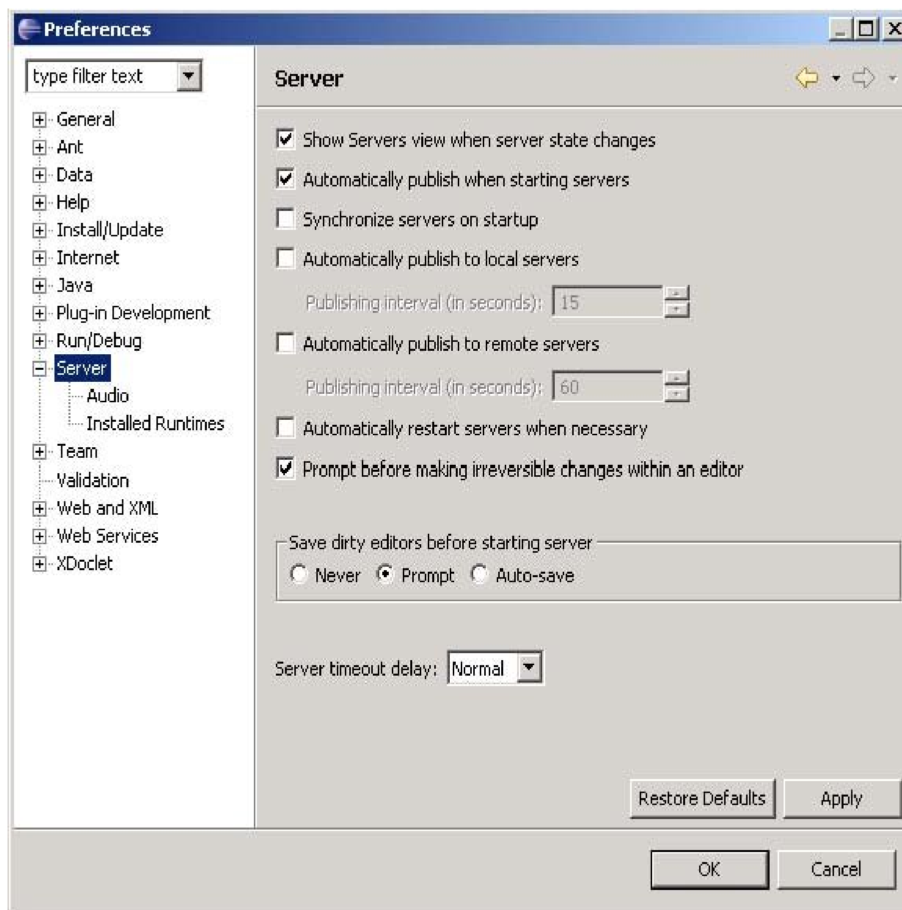
Although at present you need a specific server adapter for each type of server, the situation may change in the future. The

task of server control is in the process of being standardized using the Java Management Extension (JMX). JSR 77 defines J2EE Management APIs [JSR77] and JSR 88 defines J2EE Deployment APIs [JSR88] . As these aspects of server control become more widely supported it should be possible to create a common server adapter that works with servers from many providers.

Finally, you create a *server configuration* and add dynamic Web projects to it. A server configuration is a list of dynamic Web projects, and other configuration parameters, such as port numbers. When you select a Web resource to run, it gets deployed to a server that includes its project, the server gets started, and a Web browser is launched on a URL for the selected resource.

Do the following to add Tomcat to your workspace:

1 Open the Preferences dialog and select the Server page (see Figure 7.51, "Server Prefererences"). The main server preferences page lets you control how WTP reacts to various events that effect servers. For example, you can have WTP automatically publish changed resources to servers. Leave these settings as is and explore them later at your leisure.
2 Select the Audio preferences page (see Figure 7.52, "Server Audio Preferences"). This page lets you associate sounds with various server events. For example, you can associate a sound to play after the server has completed its startup sequence. This is handy for servers that take a long time to start. Play with these settings later.
3 Select the Installed Runtimes preferences page (see Figure 7.53, "Installed Server Runtimes"). This is where you add server runtime enviroments to your workspace.
4 Click the Add button. The New Server Runtime wizard opens (see Figure 7.54, "New Server Runtime").



Preferences

type filter text

- General
- Ant
- Data
- Help
- Install/Update
- Internet
- Java
- Plug-in Development
- Run/Debug
- Server
  - Audio
  - Installed Runtimes
- Team
- Validation
- Web and XML
- Web Services
- XDoclet

Audio

Enable sounds

Volume: 0

Sounds:

Sound	Audio File
Publishing	
Publish Finished	(none)
Server	
Server Started	(none)
Server Stopped	(none)

Play | Browse... | Reset

Restore Defaults | Apply

OK | Cancel

Preferences

type filter text

- General
- Ant
- Data
- Help
- Install/Update
- Internet
- Java
- Plug-in Development
- Run/Debug
- Server
  - Audio
  - Installed Runtimes
- Team
- Validation
- Web and XML
- Web Services
- XDoclet

Installed Server Runtime Environments

Add, remove, or edit installed server runtime definitions.
The checked runtime will be used by default when creating new projects.

Installed server runtimes:

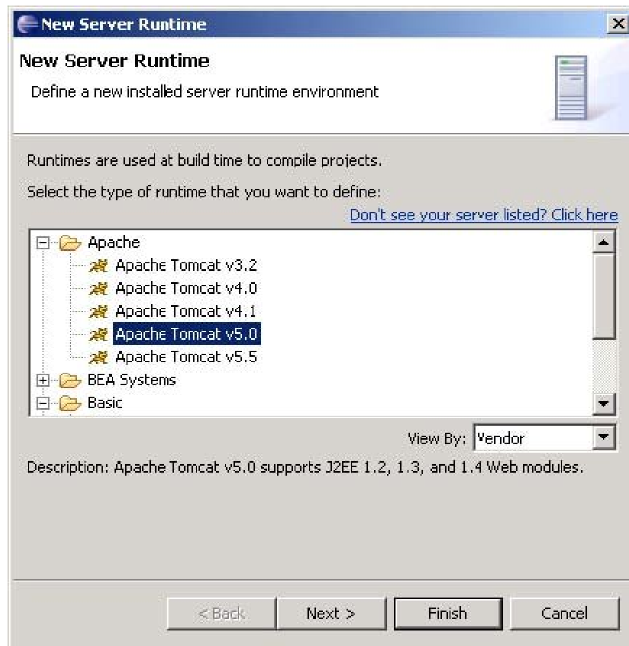
Name	Type

Add... | Edit... | Remove | Search...

OK | Cancel

Note that you can also add server runtimes by means of the Search button The server tools will then search your hard disk for installed server runtimes and add them automatically. If you do that be patient since it takes a few minutes.

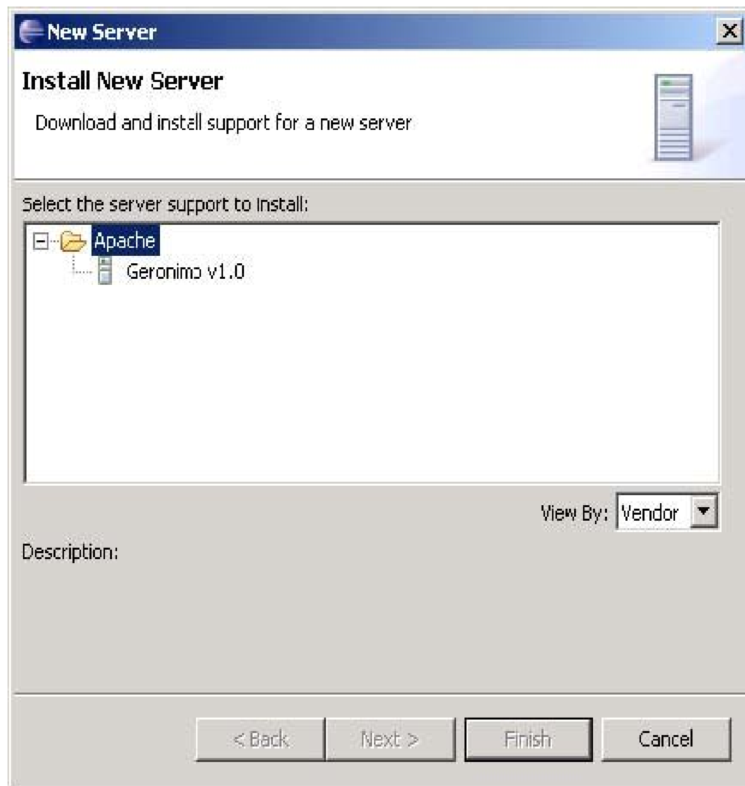
Figure 7.54. New Server Runtime



5. The New Server Runtime dialog lists all of the server adapters that are currently installed. WTP includes server adapters for many popular servers. WTP also provides an extension point where server adapter providers can list additional ones. Any provider is welcome to contribute an extension to WTP to advertise their adapters. To see the list of other available adapters, click the link labelled:

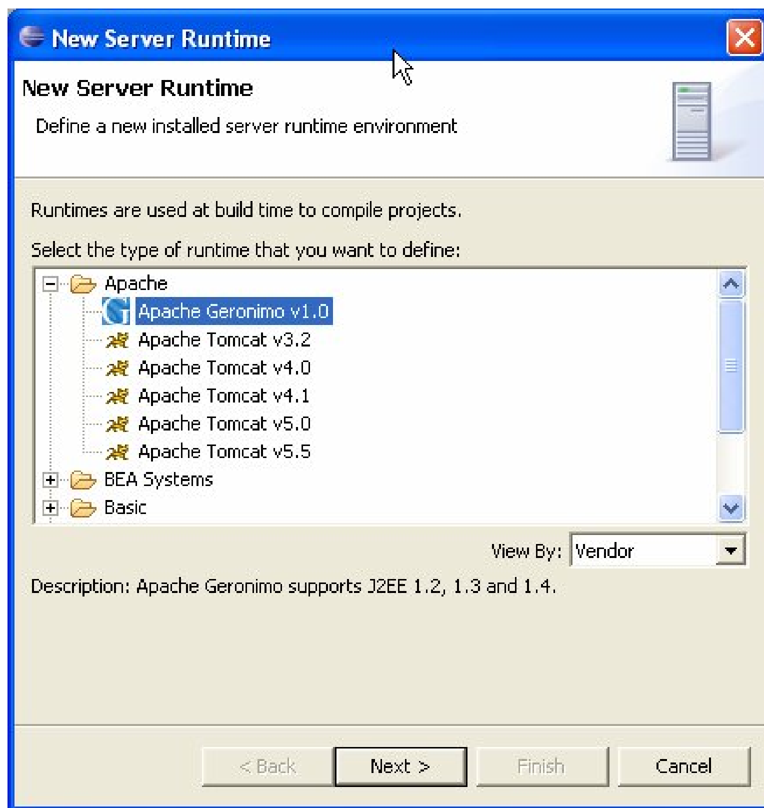
Don't see your server listed? Click here.

The Install New Server wizard opens (see Figure 7.55, “Install New Server”).



6. The downloadable server adapters advertised here simply Eclipse Features hosted on remote Update Manager sites. At the time of writing, only the Apache Geronimo project has contributed an extension to advertise their WTP server adapter, however, we expect other projects to follow suit once this capability is more well-known. Hosting the server adapter at the site where the server is developed lets it evolve independently of the WTP release schedule. This capability is also attractive for commercial vendors who may not want to contribute their adapters to WTP.

Select Finish to install the Geronimo server adapter that works with the WAS CE App server we will be using in our exercise. Allow the workbench to restart to finish the install.



We are going to use WAS CE for our server, download the server instance here:

http://www-128.ibm.com/developerworks/downloads/ws/wasce/

WTP provides an extension point for server adapters to simply the process of downloading and installing server runtimes. A server runtime provider can package their runtime as an Eclipse Feature and the server adapter can advertise the location of the Update Manager site. At present, the Apache Geronimo server adapter takes advantage of this capability.

The Geronimo Server wizard requires you to specify the location of the Geronimo installation directory. Enter the location or select it using the Browse button. You also need to specify a JRE. Be sure to specify a full JDK instead of a JRE since later you will be developing JSPs. JSP development requires a Java compiler which is not included in JREs. You can also specify a descriptive name for the server rutime envirnoment. Accept the default for now. Click the Finish button. The Installed Runtimes preference page now lists Geronimo (see Figure 7.57, “Installed Runtimes -Geronimo”). Click the checkbox to make Geronimo the default server runtime environment.

Figure 7.57. Installed Runtimes -Geronimo

You have now added Geronimo to your workspace and are ready to use it to run a Dynamic Web project.

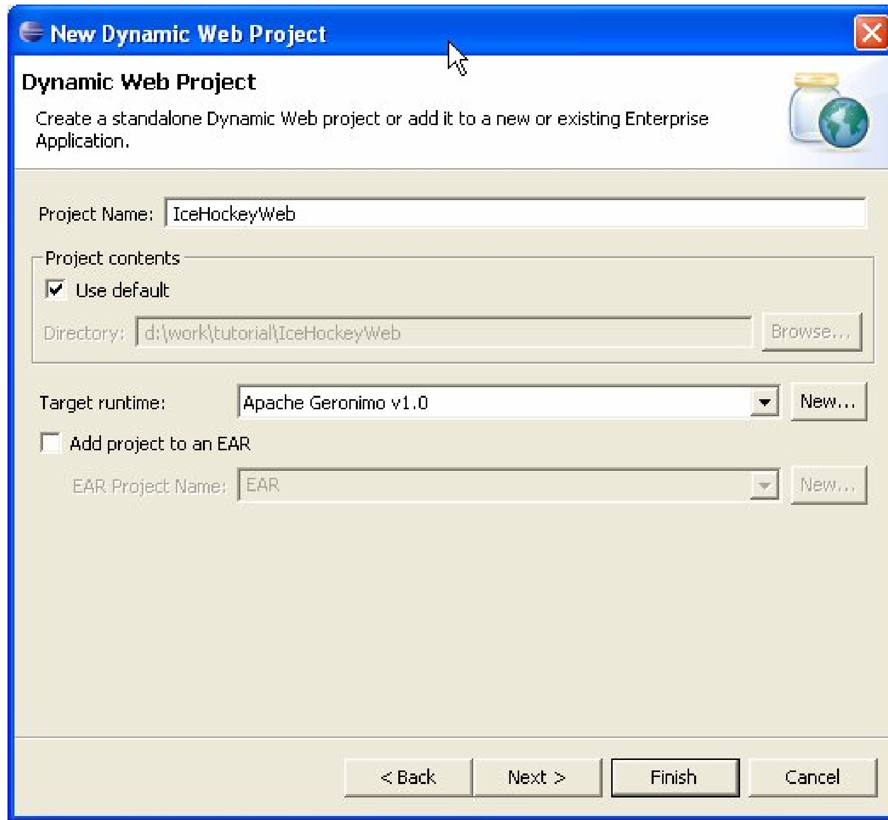
Dynamic Web Projects

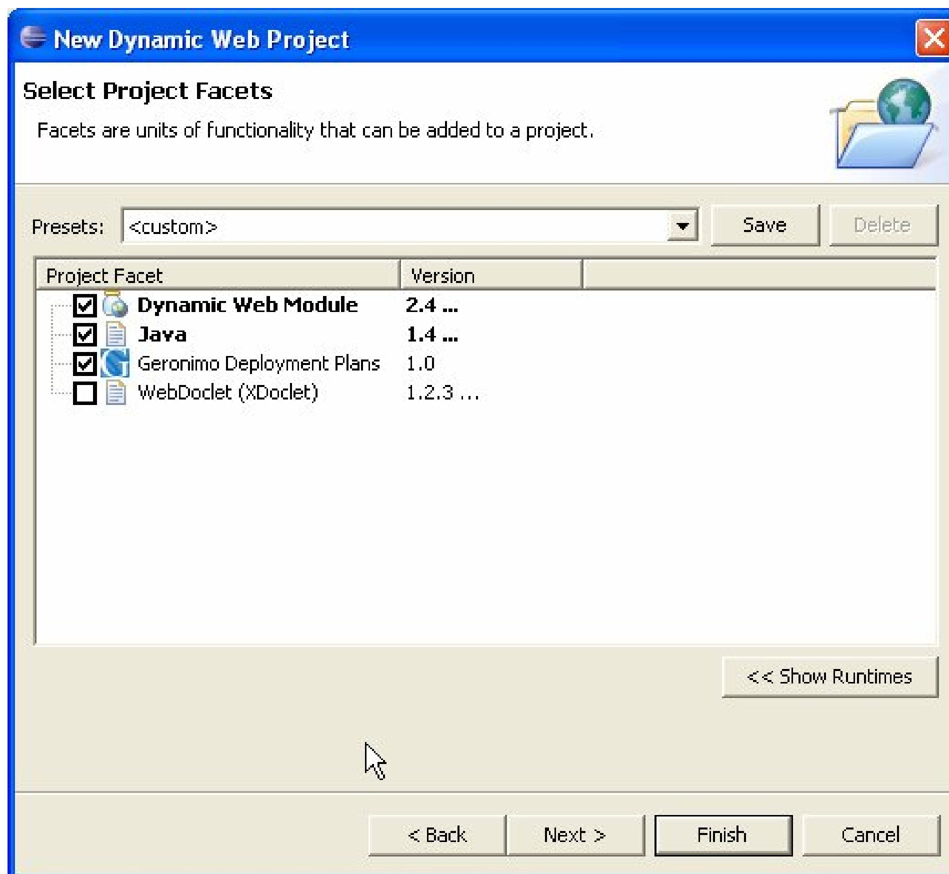
WTP provides Dynamic Web projects to host the development of J2EE modules. Since J2EE modules contain Java code, each module is developed in a separate Dynamic Web project so it can have its own Java class path as defined by JDT. Within a Dynamic Web project, you can freely arrange the Web resources in separate folders, just as JDT let's you arrange Java source in separate source folders. When you are ready to run your Web resources, WTP assembles them into the format specified by J2EE and deploys them to the server associated with the project. In fact, one Dynamic Web project can be associated with one or more server configurations. You can select one of the server configurations as the project default.

Here you create a Web module for League Planet ice hockey schedules. Do the following:

1 In the Project Explorer view, select the Dynamic Web Projects folder, right click, and select the New->Dynamic Web Project menu item. The New Dynamic Web Project wizard opens (see Figure 7.58, “New Dynamic Web Project”).
2 Enter IceHockeyWeb for the Project Name and select Geronimo as the Target runtime. Click the Next button. The Select Project Facets page is displayed (see Figure 7.59, “Select Project Facets”).
3 A project facet describes some runtime aspect of the Web module. For Geronimo 5.0, you can specify the J2EE version, the Java version, and, optionally, the XDoclet version. Each server defines a set of supported facets and their allowed values. WTP configures the Web module and sets up the class path for the project so that it matches the specified facets. Accept the defaults here and click the Next button. The Web Module page is displayed (see Figure 7.60, “Web Module”).
4 The Web Module page lets you specify its context root name and the directories for its Web and Java resources. The context root is the name that appears in the URL for the Web application. Specify icehockey as the context root and

accept the defaults for the directory names. Click Finish. WTP creates the project and populates it with configuration files such as the J2EE Web deployment descriptor, web.xml (see Figure 7.61, " Dynamic Web Project -IceHockeyWeb ").





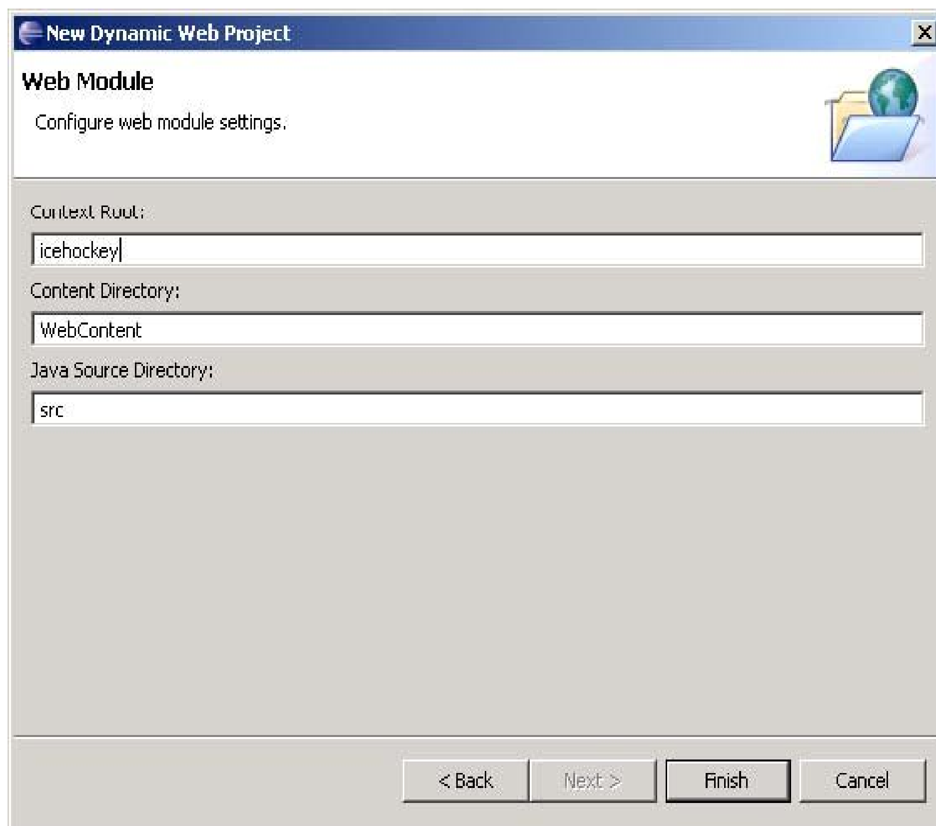
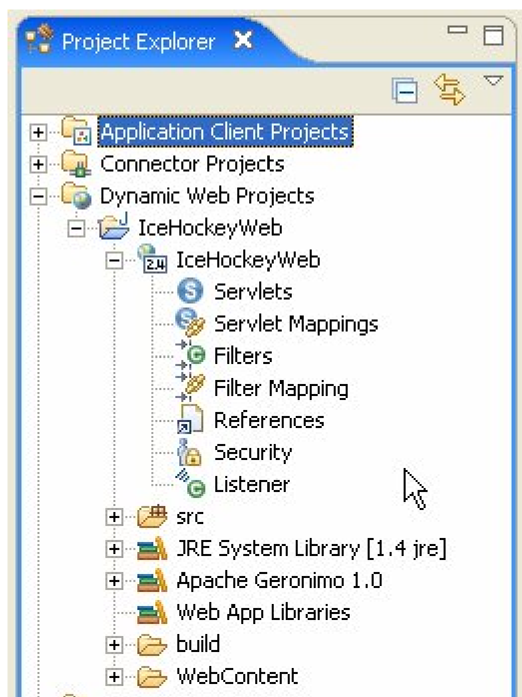


Figure 7.61. Dynamic Web Project -IceHockeyWeb



You have now created a Dynamic Web project named IceHockeyWeb and targetted it to Geronimo. Next you'll add a servlet to it.

Servlets

The first server-side component defined by J2EE was called a *servlet* as the counterpart to the J2SE client-side *applet* component. Since the introduction of the servlet, J2EE has expanded to include JSPs, EJBs, and Web services. In practice, you will develop these more specialized components rather than servlets. However, servlets still have their uses, and a knowledge of servlets will help you understand JSPs which are compiled into servlets.

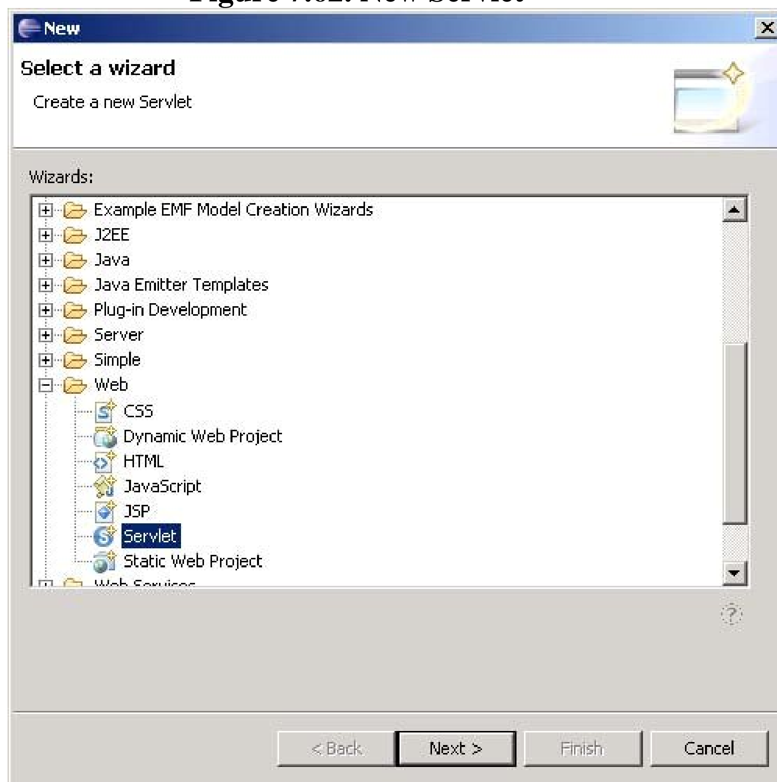
In the previous iterations, you developed schedule.xml, an XML version of the hockey schedule, and schedule.xsl, an XSLT stylesheet for transforming it to HTML. You included a processing instruction in schedule.xml so that Web browsers could apply schedule.xsl to it and display the result as HTML. Although client-side XSLT is appealing, it has a few drawbacks.

First, not all Web browsers support XSLT, and those that do support it have some minor differences. If you want to reach the maximum number of browsers and ensure the highest possible fidelity on them, then you can't rely on client-side XSLT support. However, the situation is sure to improve over time as users upgrade to modern Web browsers and the minor bugs are corrected.

In this iteration, you will develop a servlet that applies XSLT on the server using the *Transformation API for XML (TrAX)* which is part of *JSR 63: The Java API for XML Processing (JAXP)* [JSR63] . Do the following:

1 The new project has no Web resources. Copy `schedule.xml` , `schedule.xsl` , and `schedule.css` from the `icehockey` folder into the `WebContent` folder of `IceHockyWeb`. This completes project setup.
2 In the Project Explorer, select the IceHockeyWeb project, right click, and select the New->Servlet menu item. The New Servlet wizard opens (see Figure 7.62, “New Servlet”).

Figure 7.62. New Servlet



1 With Web->Servlet selected, click the Next button. The Create Servlet wizard opens (see Figure 7.63, “Create Servlet”).
2 Ensure the IceHockeyWeb is selected as the Project and \IceHockeyWeb\src is selected as the Folder. Enter org.leagueplanet as the Java package and ScheduleServlet as the Class name. The Superclass should be set to javax.servlet.http.HttpServlet. Click the Next button. The next page of the wizard is displayed (see Figure 7.64, “Create

Servlet -URL Mappings”).

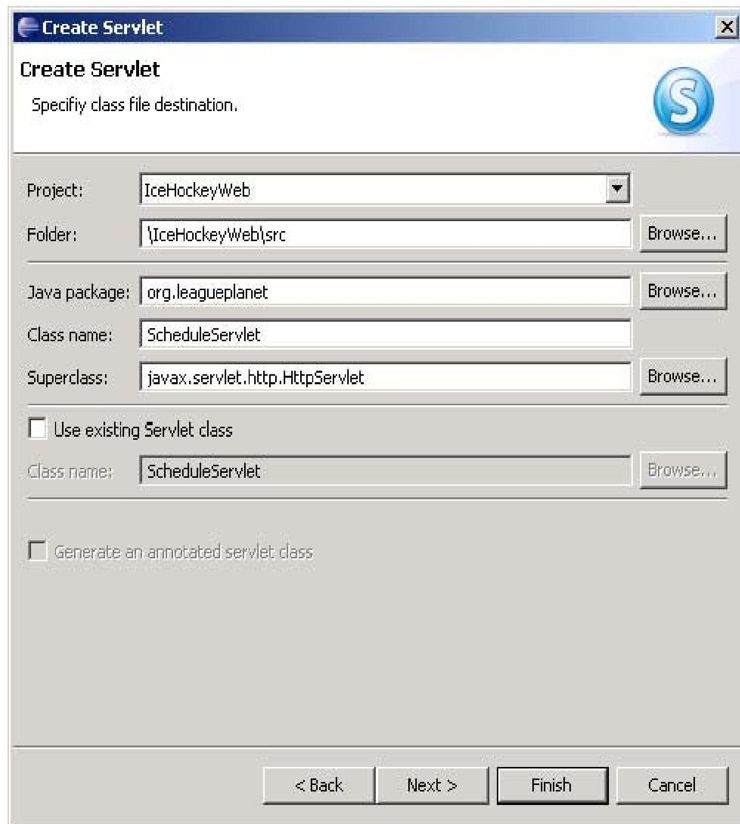
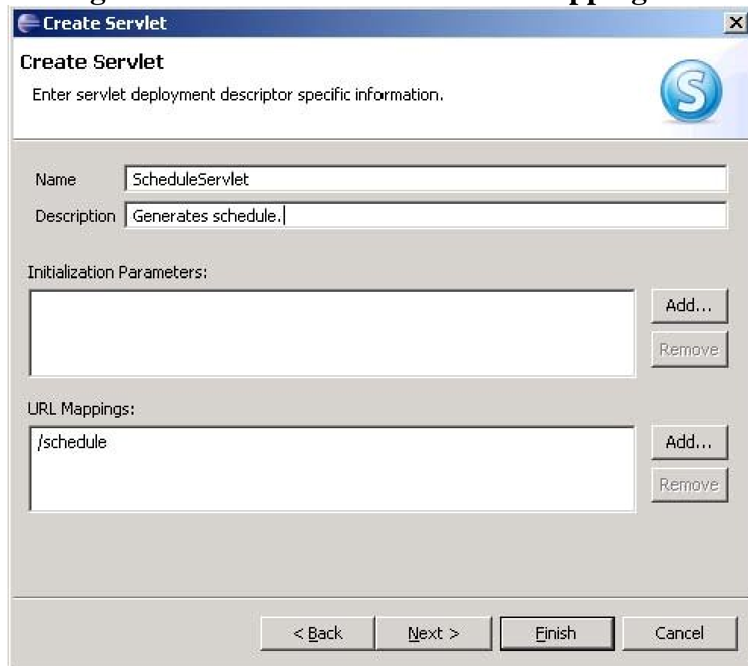


Figure 7.64. Create Servlet -URL Mappings



1 This page lets you specify information that goes in web.xml, the Web module deployment descriptor. Accept the default Name and enter a brief Description. You'll modify the URL mappings next. A URL mapping defines how the server runtime maps URLs to servlets. The default mapping uses the prefix /ScheduleServlet. However, this is a bad choice since it exposes the implementation technology. You may want to change the implementation technology later, but not break any existing URLs. A better choice is to use a prefix that doesn't expose the implementation technology. Select the /ScheduleServlet URL mapping and click the Remove button. Then click the Add button and enter the mapping /schedule.

Click the Next button to continue. The final wizard page appears (see Figure 7.65, “Create Servlet -Method Stubs”).
2 The final page lets you specify details about the servlet class. The superclass contains methods that handle some of the most comment HTTP methods, such as GET and POST. The wizard lets you select the methods that you want to handle in your servlet and will create stubs for these. You will only handle the GET method so just leave the doGet method checked. Click the Finish button. The wizard adds the new servlet information to web.xml, generates the Java source file for the servlet, and open it in the Java source editor (see Figure 7.66, “ScheduleServlet Created”).

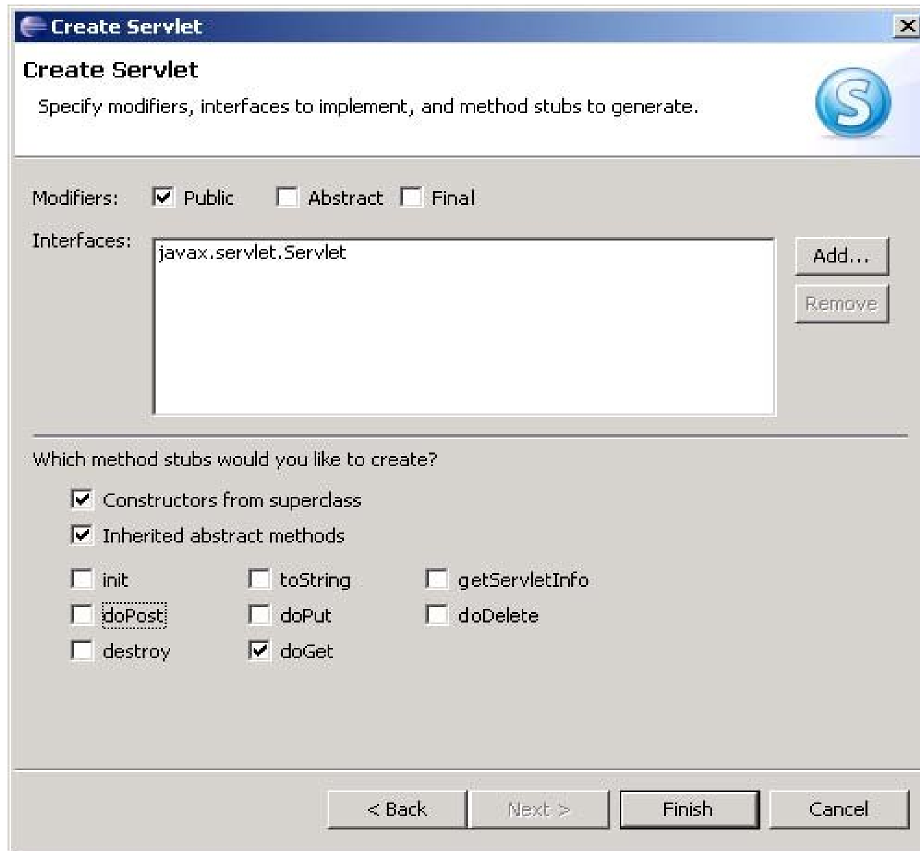
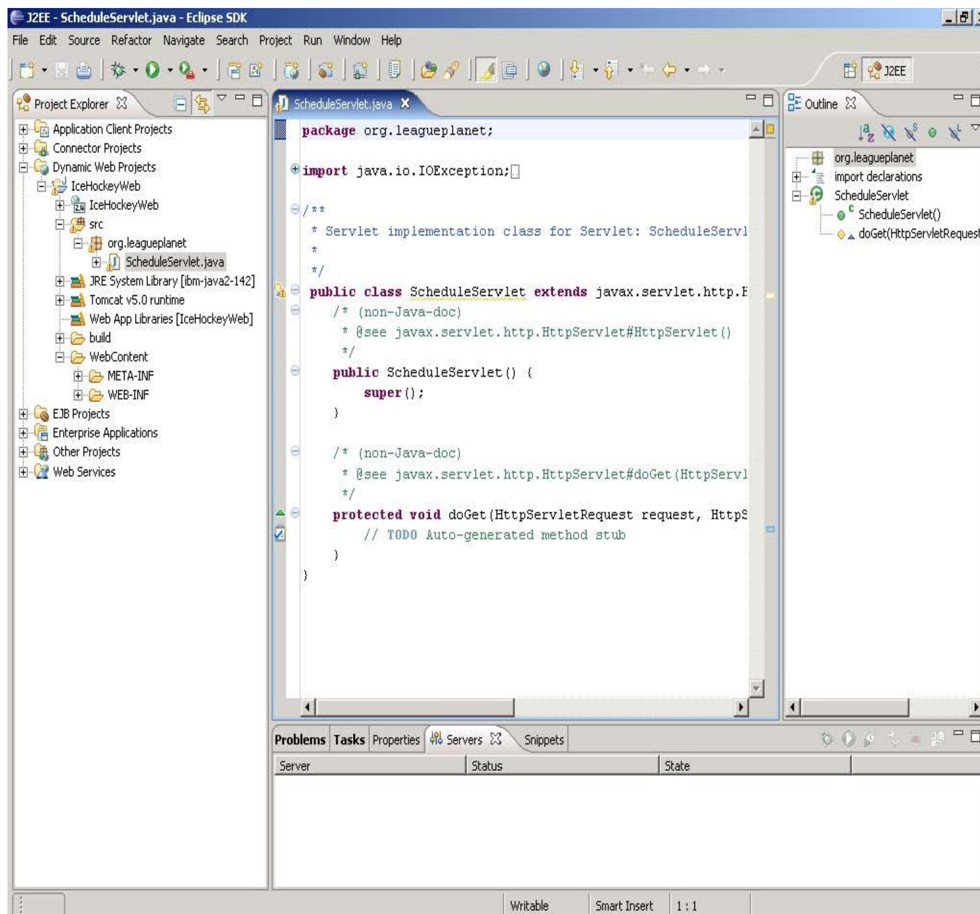


Figure 7.66. ScheduleServlet Created



7. Edit ScheduleServlet.java so that it matches Example 7.12, “Listing of ScheduleServlet.java” .

```
package org.leagueplanet;


import java.io.IOException;import java.io.InputStream;import
java.io.PrintWriter;

import javax.servlet.ServletContext;import
javax.servlet.ServletException;import
javax.servlet.http.HttpServletRequest;import
javax.servlet.http.HttpServletResponse;import
javax.xml.transform.Result;import javax.xml.transform.Source;import
javax.xml.transform.Templates;import javax.xml.transform.Transformer;import
javax.xml.transform.TransformerException;import
javax.xml.transform.TransformerFactory;import
javax.xml.transform.stream.StreamResult;import
javax.xml.transform.stream.StreamSource;

/**

 * Servlet implementation class for Servlet: ScheduleServlet
 *
 */

public class ScheduleServlet extends javax.servlet.http.HttpServlet implements
    javax.servlet.Servlet {
```

```
private static final long serialVersionUID = 1L;

protected void doGet(HttpServletRequest request,HttpServletResponse response)
    throws ServletException, IOException {

  try {ServletContext context = getServletContext();InputStream xsl =
    context.getResourceAsStream("schedule.xsl");Source xslSource = new
    StreamSource(xsl);

    TransformerFactory factory =
    TransformerFactory.newInstance();Templates templates =
    factory.newTemplates(xslSource);Transformer transformer =
    templates.newTransformer();

    InputStream xml = context.getResourceAsStream("schedule.xml");

    Source xmlSource = new StreamSource(xml);


    PrintWriter out = response.getWriter();

    Result htmlResult = new StreamResult(out);


    transformer.transform(xmlSource, htmlResult);


  response.flushBuffer(
  );out.flush();} catch
  (TransformerException
  e) {throw new
  ServletException(e);}
  }}
```

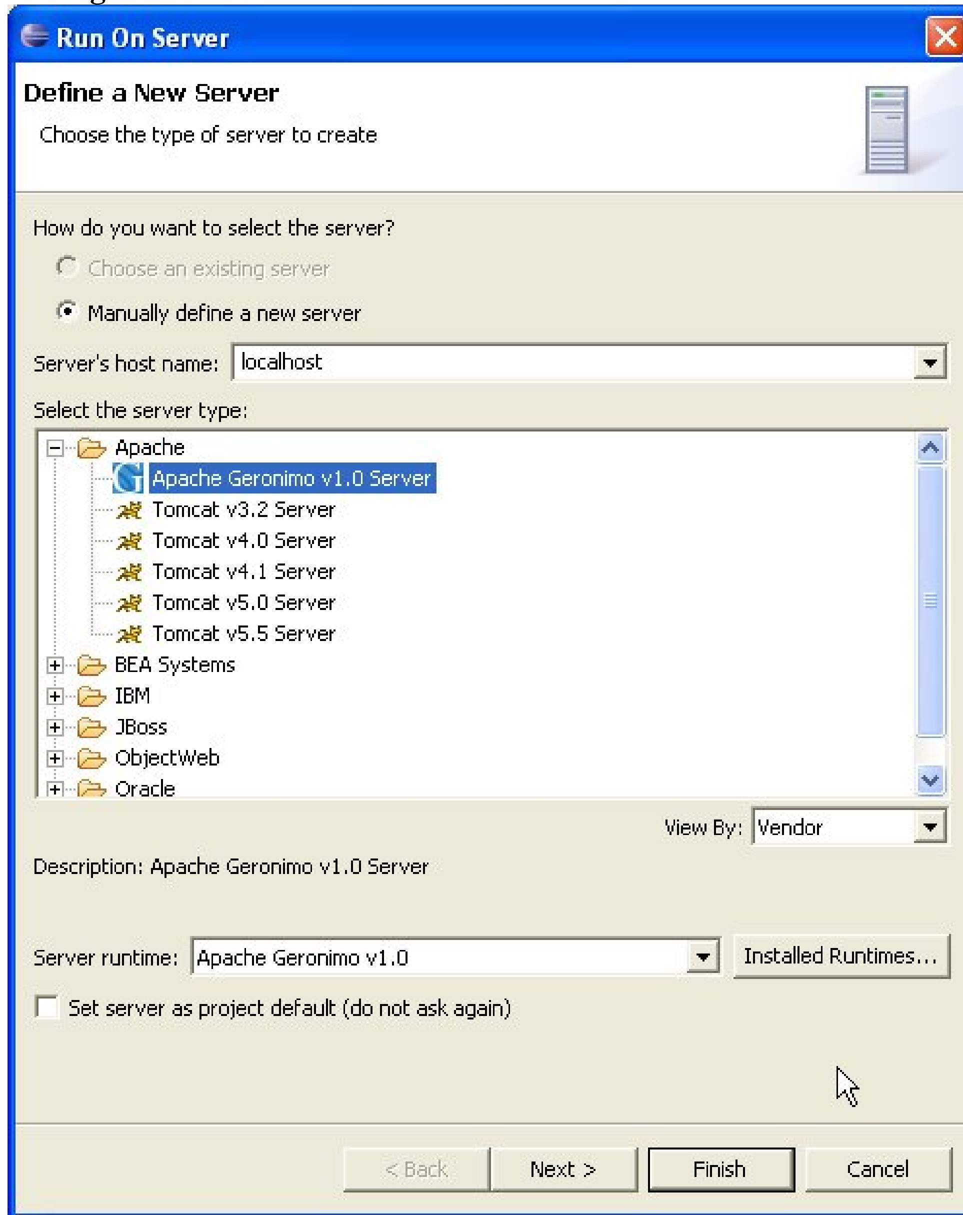
The servlet uses TrAX to apply schedule.xsl to schedule.xml. TrAX uses the AbstractFactory creational pattern as described in Chapter 3 of *Design Patterns* [Gamma1995] by Erich Gamma et al.This pattern lets you create a transformer without specifiying the concrete implementation class.There are several Java XSLT implementations, such as Xalan and Saxon, so using the Abstract-Factory pattern lets your code be independent of the particular implementation that is configured inyour JDK. Using TrAX therefore makes your code more portable.

The servlet uses the servlet context to get input streams for schedule.xml and schedule.xsl. Thistechnique is prefered to directly accessing the file system since these resources might not be available as loose files. For example, the servlet engine may be executing the Web application withoutunzipping its WAR file. The servlet wraps these input streams as TrAX source streams. The servletgets the output writer from the HTTP response and wraps it as a TrAX result stream.

The servlet creates a transformer from the schedule.xsl source stream and then applies it to theschedule.xml source stream, writing the HTML output to the response result stream.

8. Select ScheduleServlet.java, right click, and select the Run AS -> Run on Server menu item. The Run On Server wizard opens (see).

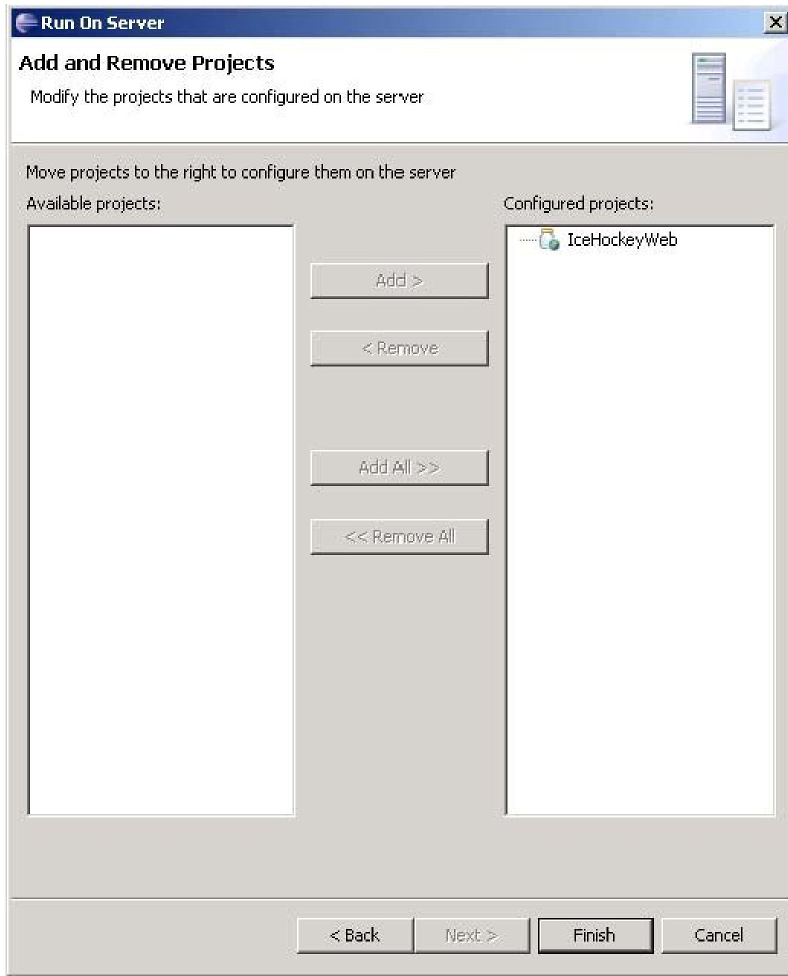
Figure 7.67. Define a New Server



1 You now must create a new server configuration. Although you already added Geronimo to your workspace, that just specifies where the runtime is installed. Now you have to create a configuration for it. Recall that a configuration is a list of Dynamic Web projects that will be deployed to the server and some other information such as port numbers. WTP uses the term Server to mean a server configuration. This page of the wizard lets you select the server runtime to use. Since you only have Geronimo(WAS CE) installed, leave that as the select. You can also set this Server to be the default associated with the project. Click Next to continue. The Add and Remove Projects page is displayed (see Figure 7.68, “Add and Remove Projects”).

10. You can select the Dynamic Web projects to include in the Server. You only have one project available, IceHockyWeb, which has been automatically added for you since it contains the servlet you want to run. Click the Finish button. The wizard creates the Server, starts it, publishes the Ice-HockeyWeb project to it, and launches the Web browser using the URL mapping for the servlet (see Figure 7.69, “ Run On Server -ScheduleServlet.java ”). As the Server starts, startup messages are displayed in the Console view.

2 The wizard created a special new project named Servers to hold the Server you just created (see Figure 7.70, “Servers Project”). The new Server is named Apache `Geronimo v1.0 Server @ local-host-config` . The Server configuration files are normal project resources so you and view and edit them using the WTP editors. Doing so, however, requires a knowledge of server administration. Many of the Geronimo configuration files contain detailed comments to assist you. Consult the Geronimo documentation for more details.



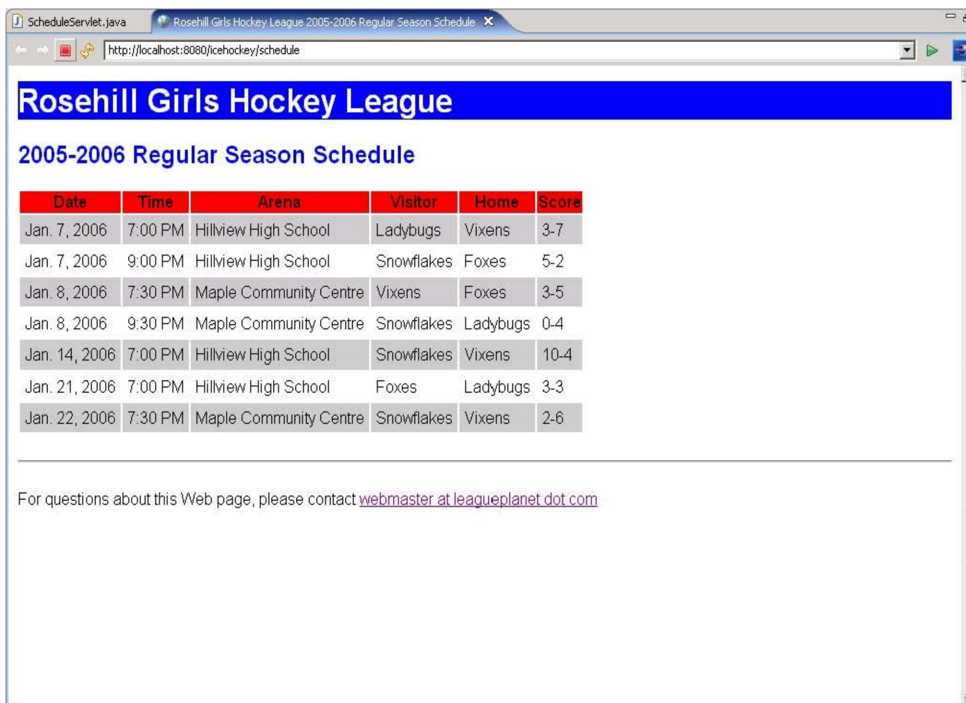
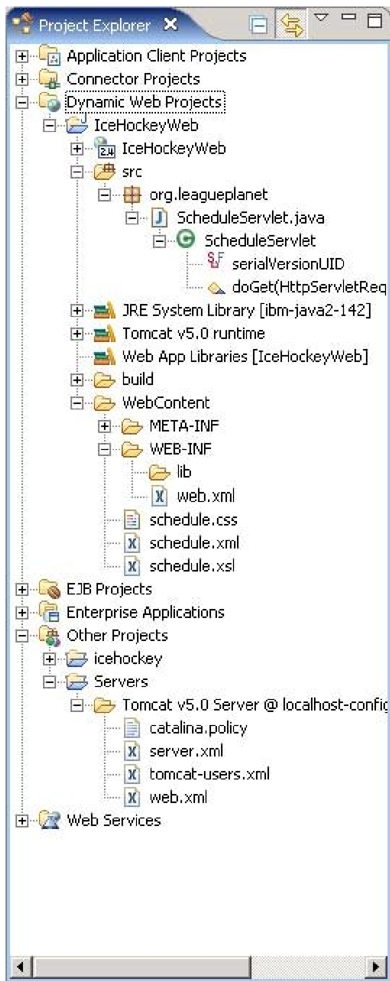
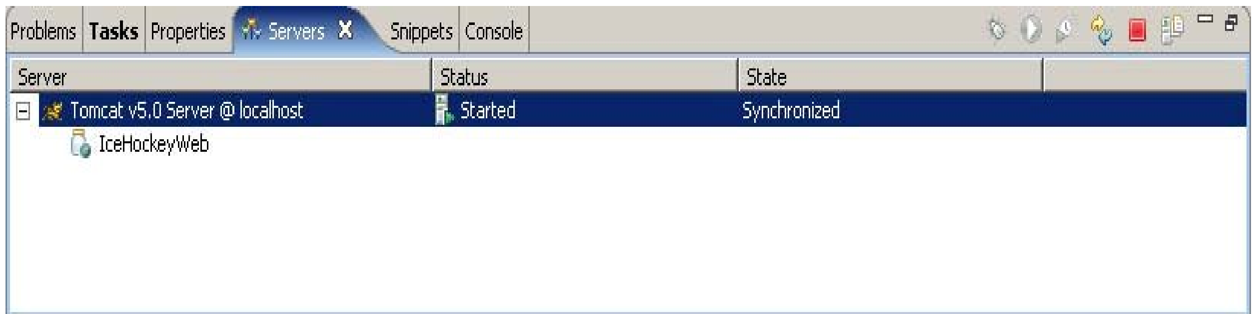


Figure 7.70. Servers Project



12. The new Server is also displayed in the Servers view where you can control it using pop-up menu items (see Figure 7.71, “Servers View”). The Servers view lets you start, stop, and restart servers, optionally in debug mode. You can also create new Servers and add and remove their projects.



Creating JSP's

JSP is the J2EE recommended way to dynamically generate Web pages. You will normally use JSP to generate HTML, however you can generate any textual content, XML for example. JSP is a template language. A JSP document consists of template text and JSP markup. The template text is send back to the client unchanged, but the JSP markup is executed on the server and the results are inserted into the output stream.

A JSP document has access to Java objects that live in various *scopes* , including application, session, request, and page. Application-scoped objects are accessible by all pages in the Web application. These are like global variables. Session-scoped objects are accessible by all pages within a single HTTP session. You'll explore session objects in the next two iterations. Request-scoped objects are accessible by all pages within a single request. Typically a servlet will set up request objects and forward the request to a JSP. Page-scoped objects are accessible only within a single JSP. These are like local variables.

When a Web browser requests a JSP, the server tranlates it into a Java servlet, compiles it, and then executes it. The fact that JSPs are compiled instead of interpretted makes them very efficient at runtime. The compilation is only done when the JSP is first requested, or if the JSP has been modified since the last request. You can also precompile JSPs into servlets to avoid the overhead of compilation in production.

JSP markup consists of directives, tags, and scriptlets. Directives control aspects of the page, such as if it is session aware. Tags are like HTML markup and are suitable for use by non-programmers. Scriptlets consist of arbritary Java source code fragments and are suitable for use by programmers. In general, scriptlets should be kept to a minimum so that the pages can be easily modified by non-programmers. The recommended design pattern for JSP is to use servlets which should handle the requests, perform detail computations, generate results to be displayed, and then forward the request to a JSP for presentation. Another reason to mimimize the amount of Java code in JSP scriplets is that it can't be easily reused elsewhere. You'll have to copy and paste useful scriptlets from one JSP to another. Copy and paste is a bad development practice since it increases code bulk and makes maintentance difficult. If you need to correct an error or make an enhancement, you'll have to locate every JSP that contains the scriptlet. If you find yourself copying and pasting scriplets, you should refactor the common code into Java source sources files so it can be reused across multiple JSPs.

A more complete discussion of JSP markup is beyond the scope of this book. See *JavaServer Pages* [Whitehead2001] by Paul Whitehead or *JSP: JavaServer Pages* [Burd2001] by Barry Burd for good treatments of this topic.

WTP includes a JSP creation wizard and a JSP structured source editor. JSP is actually a very complex source format since it combines HTML, JavaScript, and CSS in the template text with the JSP directives, tags, and scriptlets. The JSP editor provides many advanced features including syntax highlighting and content assist for JSP tags as well as full content assist for Java scriptlets.

You can set breakpoints in JSP source files and debug them just like you debug Java code. You can step from the JSP source code into any Java source code called by scriplets and tags. In fact, since JSPs are compiled into servlets, you are debugging Java code. However, the debugger shows you the JSP source code instead of the translated Java servlet code. The mapping from the Java bytecodes back to the original JSP source code has been standardized in *JSR 45: Debugging Support for Other Languages* [JSR45] .

In this iteration you'll develop JSPs that allow League Planet users to log in and out of the Web site. Users are not required to log in, but if they do then additional function is available to them. For example, fans can set up interest profiles and managers can update game schedules and scores. These functions require that users identify themselves to the League Planet Web application. The login state of each user is held in a session variable. We'll discuss the how J2EE manages sessions in the next iteration. To develop the login and logout JSPs, do the following:

1. In the Project Explorer, select the src folder of the IceHockeyWeb project, right click, and select the New->Class menu item to create a Java class named User in the org.leagueplanet package. This class will be used to hold the login state of a user. Edit User.java so that it matches Example 7.13, "Listing of User.java" . User is a simple JavaBean. It contains two properties: a boolean flag that indicates if the user if logged in, and a string that holds the user id. The class also has two methods: one to login and another to logout.

Example 7.13. Listing of User.java

```
package org.leagueplanet;

public class User {

  private boolean loggedIn = false;

  private String userId = "";

   public boolean isLoggedIn()
     {return loggedIn;}

   public void setLoggedIn(boolean loggedIn)
     {this.loggedIn = loggedIn;}

   public String getUserId()
     {return userId;}

    public void setUserId(String
      userId) {if (userId == null)
      {this.userId = "";} else
      {this.userId = userId;}}

  public void logIn(String userId) {
    setLoggedIn(true);
    setUserId(userId);

  }


  public void logOut() {
    setLoggedIn(false);
    setUserId("");

  }
}
```

2. Create a new servlet class named LoginServlet in the org.leagueplanet package using the steps you learned in the previous iteration. Map this servlet to the URL /login. Edit LoginServlet.java so that is matches Example 7.14, “Listing of LoginServlet.java” . This servlets handles GET and POST methods.

   For the GET method, the servlet simply forwards the request to either login.jsp or logut.jsp which you'll create next. The servlet determines the correct JSP by examining the User object in the session. The getUser method retrieves the session object from the request. The boolean true argument on the getSession method causes a new session object to be created if one doesn't already exist. The forward method selects login.jsp if the user is not logged, and logout.jsp if the user is logged in.

   For the POST message, the servlet looks for an action parameter. If the action if Logout, the servlet logs out the user. If the action is Login, the servlet looks for the userId and password parameters and validates them. The validation logic here is trivial. The userId must be at least two characters long and the password must be guest. In practice, the login request would come over a secure connection and the password would be checked against a database. If a validation error occurs, the error message is attached to the request so login.jsp can display it. The userId is also attached to the request so it can be redisplayed. This illustrates the technique ussing of request-scoped objects.

Example 7.14. Listing of LoginServlet.java

```
package org.leagueplanet;

import java.io.IOException;

import javax.servlet.RequestDispatcher;

import javax.servlet.ServletContext;import
javax.servlet.ServletException;import
javax.servlet.http.HttpServletRequest;import
javax.servlet.http.HttpServletResponse;import
javax.servlet.http.HttpSession;

/**

 * Servlet implementation class for Servlet: LoginServlet
 *
 */

public class LoginServlet extends javax.servlet.http.HttpServlet implements
    javax.servlet.Servlet {

  private static final long serialVersionUID = 1L;

  private User getUser(HttpServletRequest request) {

    // get the current session or create itHttpSession session =
    request.getSession(true);

    // get the user or create it and add it to the sessionUser user = (User)
    session.getAttribute("user");if (user == null) {
     user = new
     User();session.setAttribute("user
     ", user);}

    return user;
  }

  private void forward(HttpServletRequest request,HttpServletResponse response)
      throws ServletException, IOException {

    User user = getUser(request);String url = user.isLoggedIn() ? "/logout.jsp" :
    "/login.jsp";

    ServletContext context = getServletContext();RequestDispatcher
    dispatcher =
    context.getRequestDispatcher(url);dispatcher.forward(request,
    response);
  }

    protected void doGet(HttpServletRequest
        request,HttpServletResponse response) throws
        ServletException, IOException {forward(request,
        response);}

  protected void doPost(HttpServletRequest request,HttpServletResponse response)
      throws ServletException, IOException {
```

```
User user = getUser(request);

String userId =
request.getParameter("userId");
if (userId == null)userId =
"";request.setAttribute("userId
", userId);

String password =
request.getParameter("password");if
(password == null)password = "";

String action =
request.getParameter("action");if
(action == null)action = "Login";
      if (action.equals("Logout"))
        {user.logOut();} else {if
        (userId.length() < 2)
        {request.setAttribute("userIdM
        essage","User id must have at
        least 2 characters!");} else {if
        (!password.equals("guest"))
        {request.setAttribute("passwor
        dMessage","Wrong password! Try
        using: guest");} else
        {user.logIn(userId);}}}

  forward(request,
response);
}
}
```

1 Select the WebContent folder, right click, and select the New->JSP menu item. The New JSP wizard opens (see Figure 7.72, “New JSP”).

2 Enter the name login.jsp and click the Next button. The Select JSP Template page of the wizard is displayed (see Figure 7.73, “Select JSP Template”).

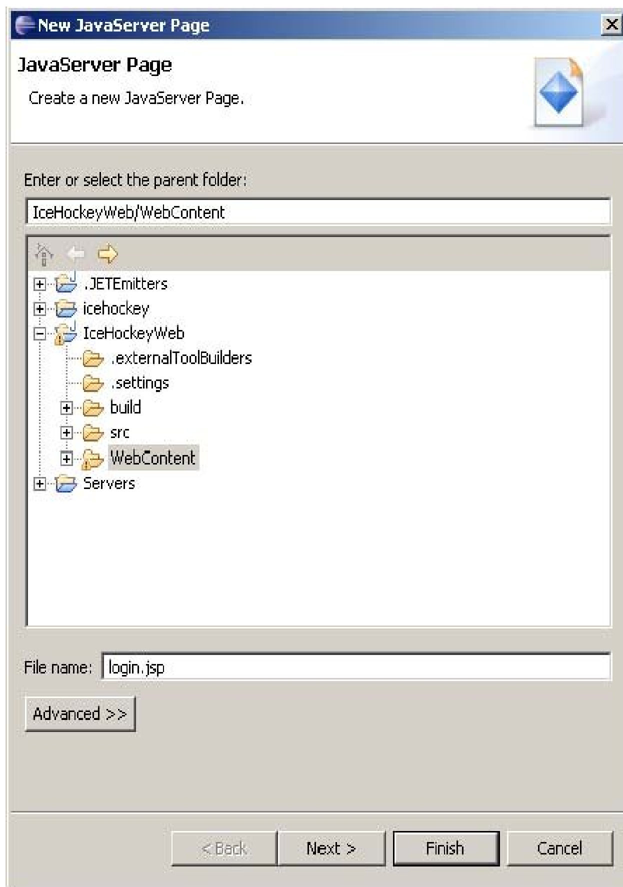
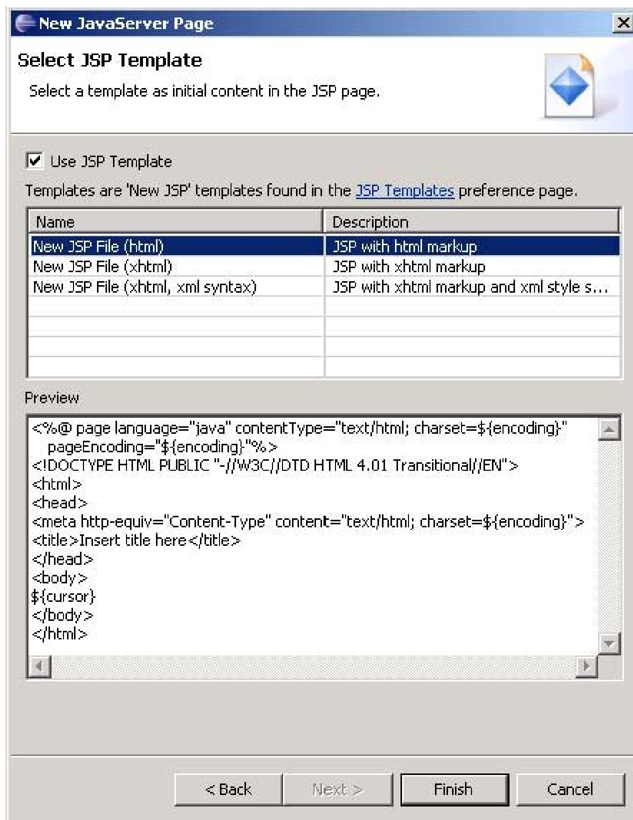


Figure 7.73. Select JSP Template



5. The wizard lets you select a template for the style of JSP you want. You can select templates that use the traditional JSP markup syntax and that generate HTML or XHTML pages, or the newer XML-compliant syntax for use with XHTML. Select the New JSP File (html) template and click the Finish button. The wizard creates login.jsp and opens it in the JSP source editor. Edit it so that it matches Example 7.15, "Listing of login.jsp" . Experiment with content assist as you edit.

   Note the first line of login.jsp which contains a page directive with the `session="true"` attribute. This enables HTTP session tracking. login.jsp also contains a scriptlet that retrieves the userId and error messages for the request object. The remainder of the login.jsp is HTML template text, except for the small scriplets that write the userId and error messages into the HTML form. This illustrates the technique of server-side validation.

Example 7.15. Listing of login.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"

  pageEncoding="ISO-8859-1" session="true"%><!DOCTYPE HTML PUBLIC "-//W3C//DTD
HTML 4.01 Transitional//EN"><html><head><meta http-equiv="Content-Type"
content="text/html; charset=ISO-8859-1"><title>League Planet Login</title><link
rel="stylesheet" href="schedule.css" type="text/css"><link rel="stylesheet"
href="validator.css" type="text/css"><%String userId = (String)
request.getAttribute("userId");
      if (userId == null)userId = "";

        String userIdMessage = (String)
            request.getAttribute("userIdMess
            age");if (userIdMessage ==
            null)userIdMessage = "";

      String passwordMessage = (String)
          request.getAttribute("passwordMes
          sage");if (passwordMessage ==
          null)passwordMessage = "";%>
</head>
<body>
<h1>League Planet Login</h1>


<form action="login" method="post">
<table>

  <tr><th align="right">User id:</th><td><input name="userId" type="text"
    value="<%= userId %>"></td><td><span class="validator"><%=
    userIdMessage %></span></td>
  </tr>

  <tr><th align="right">Password:</th><td><input name="password"
    type="password" value=""></td><td><span class="validator"><%=
    passwordMessage %></span></td>
  </tr>

  <tr><td colspan="2"> </td><td><input name="action" type="submit"
    value="Login">  <input
      name="reset" type="reset" value="Reset" /></td>
  </tr>


</table>
</form>
```

```
</body>
</html>
```

6. Create a second JSP named logout.jsp and edit it so that it matches Example 7.16, “Listing of logout.jsp” . logout.jsp also contains a page directive that enables HTTP session tracking. the user session object is retrieved in the HTML <head> element using the <jsp:useBean> tag. The userId is written into the page using the <jsp:getProperty> .

Example 7.16. Listing of logout.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"

  pageEncoding="ISO-8859-1" session="true"%><!DOCTYPE HTML PUBLIC "-//W3C//DTD
HTML 4.01 Transitional//EN"><html><head><meta http-equiv="Content-Type"
content="text/html; charset=ISO-8859-1"><jsp:useBean
class="org.leagueplanet.User" id="user" scope="session" /><title>League Planet
Logout</title><link rel="stylesheet" href="schedule.css"
type="text/css"></head><body><h1>League Planet Logout</h1>

<form action="login" method="post">
<table>

  <tr><th align="right">User id:</th><td><jsp:getProperty name="user"
    property="userId" /></td>
  </tr>

  <tr><td colspan="2"></td><td><input name="action" type="submit"
    value="Logout" /></td>
  </tr>

  </table>
  </form>


  </body>
  </html>
```

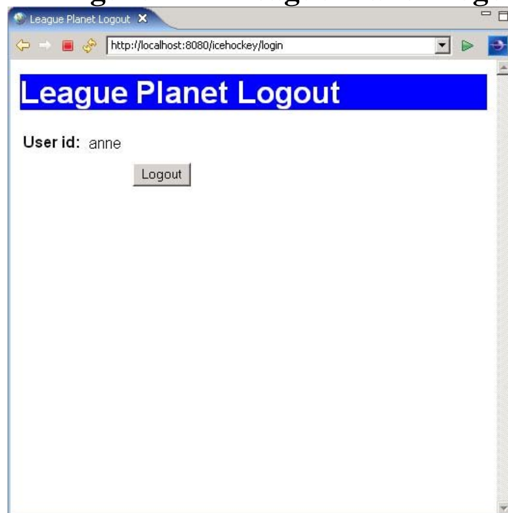
7. In the Project Explorer, select the LoginServlet in either the src folder or under the Servlets category of the IceHockey Web item, right click, and select the Run As->Run on Server menu item. The project is published, the server starts, and a Web browser is opened on the URL:

```
http://localhost:8080/icehockey/login
```

The LoginServlet recieves the GET request and forwards it to login.jsp. The Web browser displays the League Planet Login page (see Figure 7.74, “League Plant Login”).

8. Enter an invalid userId and password and click the Login button to test the server-side validation logic. Enter a valid userId, e.g. `anne` , and password, i.e. `guest` , and click the Login button. The Web browser displays the League Planet Logout page (see Figure 7.75, "League Planet Logout"). Note that logout.jsp correctly retrieved the userId from the session object and displayed it in the Web page.

Figure 7.75. League Planet Logout



9. Experiment with debugging by setting breakpoints in the servlet and JSP scriplets, and repeating the above testing. This time select the Debug As->Debug on Server menu item instead of Run As>Run on Server. All the familiar Java debugging views open.

At this point you should be comfortable with creating HTML, CSS, JavaScript, XML, DTD, JSP, and servlets in both static and dynamic Web projects. You should also be able to control servers.