Web Application Developer's Guide

VERSION 8

Borland®
JBuilder®

Borland Software Corporation
100 Enterprise Way, Scotts Valley, CA 95066-3249
www.borland.com

Refer to the file `deploy.html` located in the `redist` directory of your JBuilder product for a complete list of files that you can distribute in accordance with the JBuilder License Statement and Limited Warranty.

Borland Software Corporation may have patents and/or pending patent applications covering subject matter in this document. Please refer to the product CD or the About dialog box for the list of applicable patents. The furnishing of this document does not give you any license to these patents.

For third-party conditions and disclaimers, see the Release Notes on your JBuilder product CD.

Printed in the U.S.A.

JBE0080WW21002webapps 3E3R1002
0203040506-9 8 7 6 5 4 3 2 1
PDF

Contents

Tables

Figures

Tutorials

Chapter

1

Introduction

Web Development is a feature of JBuilder Enterprise. Applet development is a feature of all editions of JBuilder

The *Web Application Developer's Guide* presents some of the technologies available for developing web-based multi-tier applications. A web application is a collection of HTML/XML documents, web components (applets, servlets and JavaServer Pages), and other resources in either a directory structure or archived format known as a web archive (WAR) file. A web application is located on a central server and provides service to a variety of clients.

This book details how these technologies are surfaced in JBuilder and how you work with them in the IDE and the editor. It also explains how these technologies fit together in a web application. Choose one of the following topics for more information:

- Chapter 2, "Overview of the web application development process"

  Introduces the technologies discussed in this book, including servlets, JavaServer Pages (JSPs), InternetBeans Express, Struts, and applets.

- Chapter 3, "Working with WebApps and WAR files"

  Explains how to create a web application and archive it into a WAR file in JBuilder. This chapter also discusses general WebApp concepts and structure.

- Chapter 4, "Working with servlets"

  Introduces servlets and the servlet API.

- Chapter 5, "Creating servlets in JBuilder"

  Explains the Servlet wizard options, how to run servlets, how to internationalize them, and how to create data-aware servlets.

- Chapter 6, "Developing JavaServer Pages"

  Introduces JSPs and the JSP API. Explains how to use the JSP wizard to create a JSP.

- Chapter 7, "Using InternetBeans Express"

  Explains the InternetBeans library and how to use the components with servlets and JSPs.

- Chapter 8, "Using the Struts framework in JBuilder"

  Explains the Struts framework and how to create a Struts-enabled web application.

- Chapter 9, "Configuring your web server"

  Explains how to configure your web server for running in JBuilder.

- Chapter 10, "Working with web applications in JBuilder"

  Explains how to compile, run, and debug servlets and JSPs.

- Chapter 11, "Deploying your web application"

  Explains web application deployment issues.

- Chapter 12, "Editing the web.xml file"

  Explains how to use the WebApp DD Editor to edit the `web.xml` file.

- Chapter 13, "Editing the struts-config.xml file"

  Explains how to use the Struts Config Editor to edit `struts-config.xml`.

- Chapter 14, "Working with applets"

  Explains how to create applets in JBuilder. Discusses the main issues involved in applet development and deployment and presents solutions.

- Chapter 15, "Launching your web application with Java Web Start"

  Explains how to use Web Start to launch non-web applications from a web browser.

The following web application tutorials are available:

- Chapter 16, "Tutorial: Creating a simple servlet"

  Takes you through the steps of writing a simple servlet that accepts user input and counts the number of visitors to a site.

- Chapter 17, "Tutorial: Creating a servlet that updates a guestbook"

  Takes you through the steps of writing a servlet that connects to a JDataStore database, accepts user input, and saves data back to the database.

- Chapter 18, "Tutorial: Creating a JSP using the JSP wizard"

  Takes you through the steps of writing a JSP that accepts and displays user input and counts how many times a web page has been visited.

- Chapter 19, "Tutorial: Creating a servlet with InternetBeans Express"

  Takes you through the steps of writing a servlet that uses InternetBeans components to query a database table and display its contents, accept user input, and save it back to the database.

- Chapter 20, "Tutorial: Creating a JSP with InternetBeans Express"

  Takes you through the steps of writing a JSP that uses InternetBeans components to query a database table and display its contents, accept user input, and save it back to the database.

- Chapter 21, "Tutorial: Running the CheckBoxControl sample application with Java Web Start"

  Walks you through the steps of launching a Swing-based sample application with Web Start.

This document contains many links to external web sites. These web addresses and links were valid as of this printing. Borland does not maintain these web sites and can not be responsible for their content or longevity.

If you have questions specific to developing web application applications in JBuilder, you can post them to the Servlet-JSP newsgroup, borland.public.jbuilder.servlet-jsp, by browsing to `http://www.borland.com/newsgroups/`.

Documentation conventions

The Borland documentation for JBuilder uses the typefaces and symbols described in the following table to indicate special text.

Table 1.1 Typeface and symbol conventions

Typeface	Meaning
`Monospaced type`	Monospaced type represents the following: • text as it appears onscreen • anything you must type, such as "Type `Hello World` in the Title field of the Application wizard." • file names • path names • directory and folder names • commands, such as `SET PATH` • Java code • Java data types, such as `boolean`, `int`, and `long`. • Java identifiers, such as names of variables, classes, package names, interfaces, components, properties, methods, and events • argument names • field names • Java keywords, such as `void` and `static`
Bold	Bold is used for java tools, bmj (Borland Make for Java), bcj (Borland Compiler for Java), and compiler options. For example: **javac**, **bmj**, **-classpath**.
Italics	Italicized words are used for new terms being defined, for book titles, and occasionally for emphasis.
Keycaps	This typeface indicates a key on your keyboard, such as "Press *Esc* to exit a menu."
[ ]	Square brackets in text or syntax listings enclose optional items. Do not type the brackets.
< >	Angle brackets are used to indicate variables in directory paths, command options, and code samples. For example, `<filename>` may be used to indicate where you need to supply a file name (including file extension), and <username> typically indicates that you must provide your user name. When replacing variables in directory paths, command options, and code samples, replace the entire variable, including the angle brackets (< >). For example, you would replace `<filename>` with the name of a file, such as `employee.jds`, and omit the angle brackets. **Note:** Angle brackets are used in HTML, XML, JSP, and other tag-based files to demarcate document elements, such as <font color=red> and <ejb-jar>. The following convention describes how variable strings are specified within code samples that are already using angle brackets for delimiters.

Table 1.1 Typeface and symbol conventions (continued)

Typeface	Meaning
Italics, serif	This formatting is used to indicate variable strings within code samples that are already using angle brackets as delimiters. For example, `<url="jdbc:borland:`*jbuilder*`\\samples\\guestbook.jds">`
...	In code examples, an ellipsis (...) indicates code that has been omitted from the example to save space and improve clarity. On a button, an ellipsis indicates that the button links to a selection dialog box.

JBuilder is available on multiple platforms. See the following table for a description of platform conventions used in the documentation.

Table 1.2 Platform conventions

Item	Meaning
Paths	Directory paths in the documentation are indicated with a forward slash (/). For Windows platforms, use a backslash (\).
Home directory	The location of the standard home directory varies by platform and is indicated with a variable, `<home>`. • For UNIX and Linux, the home directory can vary. For example, it could be `/user/<username>` or `/home/<username>` • For Windows NT, the home directory is `C:\Winnt\Profiles\<username>` • For Windows 2000 and XP, the home directory is `C:\Documents and Settings\<username>`
Screen shots	Screen shots reflect the Metal Look & Feel on various platforms.

Developer support and resources

Borland provides a variety of support options and information resources to help developers get the most out of their Borland products. These options include a range of Borland Technical Support programs, as well as free services on the Internet, where you can search our extensive information base and connect with other users of Borland products.

Contacting Borland Technical Support

Borland offers several support programs for customers and prospective customers. You can choose from several categories of support, ranging from free support on installation of the Borland product to fee-based consultant-level support and extensive assistance.

For more information about Borland's developer support services, see our web site at http://www.borland.com/devsupport/, call Borland Assist at (800) 523-7070, or contact our Sales Department at (831) 431-1064.

When contacting support, be prepared to provide complete information about your environment, the version of the product you are using, and a detailed description of the problem.

For support on third-party tools or documentation, contact the vendor of the tool.

Online resources

You can get information from any of these online sources:

World Wide Web	http://www.borland.com/
FTP	ftp://ftp.borland.com/ Technical documents available by anonymous ftp.
Listserv	To subscribe to electronic newsletters, use the online form at: http://info.borland.com/contact/listserv.html or, for Borland's international listserver, http://info.borland.com/contact/intlist.html

World Wide Web

Check www.borland.com/jbuilder regularly. This is where the Java Products Development Team posts white papers, competitive analyses, answers to frequently asked questions, sample applications, updated software, updated documentation, and information about new and existing products.

You may want to check these URLs in particular:

- http://www.borland.com/jbuilder/ (updated software and other files)
- http://www.borland.com/techpubs/jbuilder/ (updated documentation and other files)
- http://community.borland.com/ (contains our web-based news magazine for developers)

Borland newsgroups

You can register JBuilder and participate in many threaded discussion groups devoted to JBuilder. The Borland newsgroups provide a means for the global community of Borland customers to exchange tips and techniques about Borland products and related tools and technologies.

You can find user-supported newsgroups for JBuilder and other Borland products at `http://www.borland.com/newsgroups/`.

Usenet newsgroups

The following Usenet groups are devoted to Java and related programming issues:

- `news:comp.lang.java.advocacy`
- `news:comp.lang.java.announce`
- `news:comp.lang.java.beans`
- `news:comp.lang.java.databases`
- `news:comp.lang.java.gui`
- `news:comp.lang.java.help`
- `news:comp.lang.java.machine`
- `news:comp.lang.java.programmer`
- `news:comp.lang.java.security`
- `news:comp.lang.java.softwaretools`

Note These newsgroups are maintained by users and are not official Borland sites.

Reporting bugs

If you find what you think may be a bug in the software, please report it in the Support Programs page at `http://www.borland.com/devsupport/namerica/`. Click the "Reporting Defects" link to bring up the Entry Form.

When you report a bug, please include all the steps needed to reproduce the bug, including any special environmental settings you used and other programs you were using with JBuilder. Please be specific about the expected behavior versus what actually happened.

If you have comments (compliments, suggestions, or issues) for the JBuilder documentation team, you may email `jpgpubs@borland.com`. This is for documentation issues only. Please note that you must address support issues to developer support.

JBuilder is made by developers for developers. We really value your input.

Chapter

2

Overview of the web application development process

Web Development is a feature of JBuilder Enterprise. Applet development is a feature of all editions of JBuilder

This section introduces web application technologies, presents some of the differences between them, and discusses how to decide which technologies to use. It begins with a basic summary of the technologies presented in this book:

Table 2.1 Web application technologies

Technology	Description
Servlets	A server-side Java application which can process requests from clients.
JavaServer Pages (JSP)	An extension of servlet technology. JavaServer Pages use custom tag libraries and essentially offer a simplified way to develop servlets. They appear to be different during development, but when first run, they are compiled into servlets by the web server.
InternetBeans Express	A set of components and a tag library provided by Borland, used for easy presentation and manipulation of data from a database. This technology is used in conjunction with servlet or JSP technology and simplifies development of data-aware servlets or JSPs.
Struts	An open source framework provided by the Jakarta Project that is used for building web applications. Struts provides a flexible control layer based on standard technologies like Servlets, JSPs, JavaBeans, ResourceBundles, and XML.
JavaServer Pages Standard Tag Library (JSTL)	A tag library provided by Sun that is part of the Java Web Services Development Pack 1.0 (WSDP). It provides a set of tags that allow developers to do common tasks in a standard way. The JSTL consists of four areas, each with its own TLD (tag library descriptor) and namespace.

Table 2.1 Web application technologies (continued)

Technology	Description
Cocoon	A servlet-based, Java publishing framework for XML that is integrated into JBuilder. Cocoon allows separation of content, style, and logic and uses XSL transformation to merge them. It can also use logic sheets, Extensible Server Pages (XSP), to deliver dynamic content embedded with program logic written in Java. For more information on the Cocoon framework, see "Presenting XML with Cocoon" in the *XML Developer's Guide*.
Applets	A specialized kind of Java application that can be downloaded by a client browser and run on the client's machine.

The summary gives you some idea about the nature of each of these technologies, but how do you know which ones to use? What are the advantages and disadvantages of each of these technologies? We'll answer these questions and more in the following discussion.

Servlets

Servlets are Java programs that integrate with a web server to provide server-side processing of requests from a client browser. They require a web server that supports JavaServer technology, such as the Tomcat web server that ships with JBuilder. (Tomcat can also be integrated with web servers that don't support JavaServer technology, thus allowing them to do so. One example of this is IIS.) Java servlets can be used to replace Common Gateway Interface (CGI) programs, or used in the same situations where you might have previously used CGI. There are some advantages to using servlets over CGI:

- Reduced memory overhead
- Platform independence
- Protocol independence

You use a client program written in any language to send requests to the servlet. The client can be as simple as an HTML page. You could also use an applet for the client, or a program written in a language other than Java. On the server side, the servlet processes the request, and generates dynamic output which is sent back to the client. Servlets usually don't have a UI, but you can optionally provide one on the client side.

Servlets have some advantages over applets:

- You don't need to worry about which JDK the client browser is running. Java doesn't even need to be enabled on the client browser. All the Java is executed on the server side. This gives the server administrator more control.
- After the servlet is started, requests from client browsers simply invoke the `service()` method of the running servlet. This means that clients don't experience a performance hit while waiting for the servlet to load. This is much faster than downloading an applet.

Deployment of a servlet to the web server can be tricky, but it's certainly not impossible. JBuilder provides some tools which make deployment easier. These are discussed in Chapter 11, "Deploying your web application."

For more information on Java servlets, see Chapter 4, "Working with servlets," and Chapter 5, "Creating servlets in JBuilder."

JavaServer Pages (JSP)

JavaServer Pages (JSP) are an extension of servlet technology. They are essentially a simplified way of writing servlets, with more emphasis on the presentation aspect of the application.

The main difference between servlets and JSPs is that with servlets, the application logic is in a Java file and is totally separate from the HTML in the presentation layer. With JSPs, Java and HTML are combined into one file that has a `.jsp` extension. JSPs are evolving toward containing no Java code at all and using tag libraries only. For example, JSTL has tags for conditions and loops, which would replace Java code.

When the web server processes the JSP file, a servlet is actually generated, but you as a developer are not going to see this generated servlet. In fact, when you are compiling and running your JSP within the JBuilder IDE, you may see exceptions or errors which are actually occurring in the generated servlet. This can be a bit confusing, because it can be somewhat difficult to tell which part of your JSP is causing a problem when the error message refers to a line of code which is actually part of the generated code. (Note that JBuilder's debugger allows you to view the original source or the generated source.)

JSPs also allow the use of tag libraries. A *tag library* is a collection of related custom tags. A *custom tag* invokes a *custom action*, also known as a reusable JSP module. JBuilder's JSP wizard supports the automatic integration of the following tag libraries into your JSP:

- InternetBeans Express 1.1: A tag library provided by Borland, used for easy presentation and manipulation of data from a database.

- JSTL 1.0: A tag library provided by Sun that includes a set of tags to allow developers to do common tasks in a standard way.
- Struts 1.0: An open source tag library provided by the Jakarta Project that is used for building web applications.

Note You can also define your own tag libraries and use them in JBuilder. Additionally, third parties may provide more advanced tag library support or other frameworks through OpenTools.

JSPs have some advantages and some disadvantages compared to servlets. These are some of the advantages:

- Less code to write.
- Easy to incorporate existing JavaBeans.
- Deployment is easier. More of the deployment issues are automatically taken care of for you, because JSPs map to a web server in the same way that HTML files do.
- You can use tags only, and don't need to embed HTML code in your JSP. This means that page authors don't have to know Java at all. (Of course, you can still embed HTML code in your JSP. With careful planning, the HTML code can be cleanly separated from the Java code, making the JSP more readable.)

These are some of the disadvantages of JSPs:

- Invisible generated servlet code can be confusing, as previously mentioned.
- Since the HTML and Java are not in separate files, a Java developer and a web designer working together on an application must be careful not to overwrite each other's changes.
- The combined HTML and Java in one file can be hard to read, and this problem intensifies if you don't adhere to careful and elegant coding practices.
- The JBuilder designer does not support designing `.jsp` files.

JSPs are very similar to ASPs (Active Server Pages) on the Microsoft platform. The main differences between JSPs and ASPs are that the objects being manipulated by the JSP are JavaBeans, which are platform independent. Objects being manipulated by the ASP are COM objects, which ties ASPs completely to the Microsoft platform.

For more information on JSP technology, see Chapter 6, "Developing JavaServer Pages."

InternetBeans Express

JBuilder's InternetBeans Express technology integrates with servlet and JSP technology to add value to your application and simplify servlet and JSP development tasks. InternetBeans Express is a set of components and a JSP tag library for generating and responding to the presentation layer of a web application. It takes static template pages, inserts dynamic content from a live data model, and presents them to the client; then it writes any changes that are posted from the client back into the data model. This makes it easier to create data-aware servlets and JSPs.

InternetBeans Express contains built-in support for DataExpress DataSets and DataModules. It can also be used with generic data models and EJBs.

For more information on InternetBeans Express, see Chapter 7, "Using InternetBeans Express."

Struts

The Struts open source framework is based on the Model 2, or Model-View Controller, approach to software design. In this framework, the model contains the data, the view is the presentation of the data, and the controller controls the interaction between the model and the view.

- The view is typically a JSP page.
- The model can be any data access technology, from JDBC to an EJB.
- The controller is a Struts servlet called `ActionServlet`.

This framework, which is a collection of classes, servlets, JSP tags, a tag library, and utility classes, cleanly divides the HTML from the Java code and the visual presentation from the business logic.

The major advantage of using Struts is the division between Java code and HTML tags. With Struts, your web application becomes easier to understand. A web designer doesn't have to search through Java code to make changes to the presentation, and a developer doesn't have to recompile code when redesigning the flow of the application.

Other than its complexity, the Struts framework provides few disadvantages for the Java developer. JBuilder provides a robust set of tools that simplify the complexity and help keep classes and `xml` files in sync.

For more information about Struts in JBuilder, see Chapter 8, "Using the Struts framework in JBuilder." For more information about Struts, see "The Jakarta Project: Struts" at `http://jakarta.apache.org/struts/index.html`.

JavaServer Pages Standard Tag Library (JSTL)

The JSTL is a tag library from Sun that is part of the Java Web Services Development Pack 1.0 (WSDP). It provides a set of tags that allow developers to do common tasks in a standard way. For example, instead of using different iteration tags from numerous vendors, JSTL defines a standard tag that works the same everywhere. This standardization lets you learn a single tag and use it on multiple JSP containers.

The JSTL is divided into four areas, each with its own TLD (tag library descriptor) and namespace. These four areas are Core, XML processing, internationalization, and database access.

For more information on JSTL, see the JavaServer Pages Standard Tag Library at `http://java.sun.com/products/jsp/jstl/`.

Applets

There was much ado about applets when the Java language first became available. Web technology was less developed then, and applets promised solutions to some of the problems faced by developers at that time. In fact, applets became so popular that to this day, developing an applet is often one of the first assignments given in beginning Java courses. As a result, a common mistake among Java developers is to rely too much on applets. Applets certainly have their place, but they are not a magic solution to your web development problems.

Some of the disadvantages of applets are:

- Deployment and testing can be difficult.
- You are relying on the client machine having Java enabled in its browser.
- Different browsers support different versions of the JDK, and usually not the latest version.
- The applet can be slow to start the first time, because it needs to be downloaded by the client.

Some of these problems do have solutions, which are discussed in more detail in Chapter 14, "Working with applets." When considering using an applet, it is best to think about whether some other Java technology can better serve your purposes.

There are some advantages to using applets. These include:

- Applets can provide more complex user interfaces (UI) than servlets or JSPs.
- Since applets are downloaded and run on the client side, the web server doesn't need to support Java. This can be important if you are writing a web application for a site where you don't have control over the web server (such as a site hosted by an outside ISP).
- Applets can locally validate data entered by the user, instead of validating it on the server side. You could also accomplish this with JavaScript in conjunction with a servlet or JSP.
- After the initial download of the applet, the number of requests from the browser to the server can be reduced, since a lot of processing can be accomplished on the client machine.

For more information about applets and solving applet issues, see Chapter 14, "Working with applets."

Deciding which technologies to use in your web application

Now that you've seen an overview of the various web technologies, how do you decide which to use in your web application? The following tips may help you make this decision:

- Do you need a very complex UI? If your UI is more complex than just data entry components (such as text fields, radio buttons, combo boxes or list boxes, submit buttons, etc.) and images, you may want to use an applet.
- If you want to do a lot of server-side processing, you should use a servlet or JSP.
- If you want to avoid making your users download a lot of code and speed up application startup, you should use a servlet or a JSP.
- If you want control over the version of the JDK for the application (without downloads), or you are concerned about users who have Java disabled in their browsers, you should use a servlet or a JSP.
- If you are looking for a replacement for CGI, with less memory overhead, you should use a servlet or JSP.
- If you need something similar to an ASP, but you prefer it to be platform independent, you should use a JSP.

- If you need a complex UI, but you also want some of the features of servlets or JSPs, consider combining an applet and a servlet. You can have an applet on the browser (client) side talk to a servlet on the server side.
- If you want to use a servlet or JSP, and you want to make it data-aware, you should use InternetBeans Express.
- If you're thinking of developing a tag library of standard routines, such as control structures or date and number formatting, look at JSTL first to see if it already has the routines you need.
- If you want to separate the HTML from the Java code, use a Struts web application.

Note that servlets and JSPs are similar enough that deciding between them is largely a matter of personal preference. Also, keep in mind that many web applications will use some combination of two or more of these technologies.

The basic web application development process

Whichever web technologies you choose, you are still going to have to follow the same basic steps to develop your web application and get it working on the web server. These are the steps:

Step	Description
Design your application	Decide how you are going to structure your application and what technologies you will use. Decide what the application will accomplish, and how it will look.
Configure your web server in the JBuilder IDE	You can optionally set up your web server to work in the JBuilder IDE, so you can compile, run, and debug your application with the same web server you intend to use for deployment. You can also use Tomcat, the web server that ships with JBuilder, for compiling, running, and debugging.
Develop your application	Create a WebApp, then write the code for the application. Whether your application is composed of applets, servlets, JavaServer Pages or Struts classes, using JBuilder's many tools simplifies development tasks.
Compile your application	Compile the application in the JBuilder IDE.
Web run your application	Run the application in the JBuilder IDE. This gives you a working preview of the application, without the need to deploy it first. You should do some local testing of the application at this stage.

Step	Description
Deploy your application	Edit the web.xml and struts-config.xml files and deploy your application to the web server. (If you use JBuilder tools, you might not have to edit these files.) The specifics of your approach to this step depends largely on which web server you are using.
Test your application	Test your application running on the web server. This helps you find any problems with deployment, or with the application itself. You should test from a client browser on a different machine than the web server. You may also want to try different browsers, since the application may appear slightly different in each one.

Web applications vs. distributed applications

You might be considering using a distributed application for your program rather than a web application. Both handle client/server programming. However, there are some differences between the two technologies.

Web applications require a browser on the client side and a web server on the server side. For example, applets are downloaded to multiple client platforms where they are run in a Java Virtual Machine (JVM) provided by the browser running on the client machine. Servlets and JSPs run inside a Java-enabled web server that supports the JSP/Servlet specifications.

Web applications can be made available to anyone who has access to the Internet, or you can put them behind a firewall and use them only within your company's intranet.

A web application can be part of a larger distributed application, which, in turn, can be part of an enterprise, or J2EE, application. For a J2EE application example and supporting documentation, see the *Java 2 Platform, Enterprise Edition Blueprints* at `http://java.sun.com/j2ee/blueprints/`.

In general, distributed applications manage and retrieve data from legacy systems. The legacy system may exist on numerous computers running different operating systems. A distributed application uses an application server, such as the Borland Enterprise Server, for application management. Distributed applications do not have to be Java-based; in fact, a distributed application can contain many different programs, regardless of what language they are written in or where they reside.

Distributed applications are usually confined to a network within a company. You could make parts of your distributed application available to customers over the Internet, but then you would be combining a distributed application with a web application.

Technologies used in a distributed application include the Common Object Request Broker Architecture (CORBA) and Remote Method Invocation (RMI):

- CORBA's primary advantage is that clients and servers can be written in any programming language. This is possible because objects are defined with the Interface Definition Language (IDL) and communication between objects, clients, and servers are handled through Object Request Brokers (ORBs). For more information on CORBA, see "Exploring CORBA-based distributed applications" in the *Enterprise JavaBeans Developer's Guide*.
- Remote Method Invocation (RMI) enables you to create distributed Java-to-Java applications, in which the methods of remote Java objects can be invoked from other Java virtual machines, possibly on different hosts. For more information about RMI, see the RMI sample in the `<jbuilder>/samples` folder.

Chapter

3

Working with WebApps and WAR files

Web Development is a feature of JBuilder Enterprise

JBuilder provides some important features for managing the structure of your web application. There are two key concepts you need to understand to make effective use of these features: WebApps and web archive (WAR) files.

The WebApp

A WebApp describes the structure for a web application. It is essentially a directory tree containing web content used in your application. It maps to a `ServletContext`. A deployment descriptor file called `web.xml` is always associated with each WebApp. This deployment descriptor contains the information you need to provide to your web server when you deploy your application.

Using a WebApp is required if you have servlets or JSPs in your project. Although you probably wouldn't use a WebApp if your web application contains only an applet, you must use one in a web application which contains an applet and a servlet or JSP. That way, you can store the whole WebApp in a single WAR file. You might have several WebApps in one web site. JBuilder supports this notion by allowing you to have multiple WebApps in one project.

It's important to think about the structure of your web applications during the planning stages. How many WebApps does it have? What are their names? Will you store these WebApps in WAR files? Do your WebApps require any framework support? By planning the structure from the beginning, you make deployment easier later on. JBuilder does support the notion of a default WebApp, rooted in the `<projectdirectory>/defaultroot` directory. If you don't specify WebApps for your web applications, they go into the default WebApp.

For more information on how JBuilder works with WebApps, see "Creating a WebApp with the Web Application wizard" on page 3-3, and "The WebApp and its properties" on page 3-5.

Web archive (WAR) files

A WAR file is an archive file for a web application. It's similar to a JAR file. By storing your entire application and the resources it needs within the WAR file, deployment becomes easier. You copy just the WAR file to your web server, instead of making sure many small files get copied to the right place. JBuilder can automatically generate a WAR file when you build your project, when you build your WebApp, or when you build both. You can also choose never to create a WAR file, which might be useful during the initial stages of development, before you are ready to test deployment.

For more information on how JBuilder works with WAR files, see "The WAR file" on page 3-13.

Tools for working with WebApps and WAR files

Here is a list of the tools that JBuilder provides for working with WebApps and WAR files:

Table 3.1 JBuilder WebApp and WAR file tools

Tool	Description
Web Application wizard	A simple wizard for creating a WebApp. It allows you to specify the WebApp name, the root directory for the web application's documents, when to build a WAR file, which JSP/Servlet frameworks the WebApp should support, and a default launch URI.
WebApp node	A node in the project pane of the JBuilder IDE representing the WebApp. This node has a properties dialog box for configuring the WebApp. Contained within the WebApp node are other nodes for the deployment descriptors, the root directory, and an optional WAR file.

Table 3.1 JBuilder WebApp and WAR file tools (continued)

Tool	Description
WAR file node	A node in the project pane of the JBuilder IDE representing the WAR file. It has a properties dialog box and a viewer for the contents of the WAR file.
WebApp DD Editor	A user interface and editor for the `web.xml` deployment descriptor file that is required for each WebApp. You can also edit server-specific deployment descriptors, such as WebLogic's `weblogic.xml`, in JBuilder. Deployment descriptors are discussed in detail in "Editing deployment descriptors" on page 11-4.
Struts Config Editor	A user interface and editor for the `struts-config.xml` deployment descriptor file that is required to support the Struts framework.

Creating a WebApp with the Web Application wizard

The Web Application wizard creates a new WebApp. Before you can use the Web Application wizard, you must have a server selected for your project. To create a WebApp:

1. Select File | New. Click the Web tab of the object gallery.
2. Select Web Application. Click OK.
3. Enter a Name for your WebApp. The Directory field is automatically filled in as you type.
4. Enter the Directory location that will be your WebApp's document root. This field is automatically filled in as you type the name. The directory name creates a subdirectory of the project directory. You can also click the ellipsis button to browse and create a new directory, or choose an existing directory. Choosing the project root or the `src` directory isn't recommended.
5. Specify when to build a WAR file by selecting one of the following settings from the Build WAR drop-down list.
   - When Building Project Or WebApp
   - When Building WebApp Only
   - When Building Project Only
   - Never

   Whenever a WAR file is generated, it will have the same name as the WebApp and be placed in the directory that contains the document root directory. If you select Never here, don't worry. You can always change your mind later on. This drop-down list corresponds to a setting in the WebApp properties dialog box.

6 Select the JSP/Servlet Frameworks you want your WebApp to support. The following frameworks are included with JBuilder.

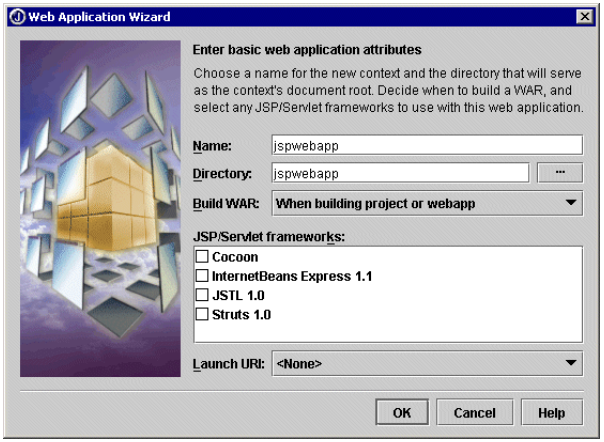
- Cocoon — a servlet-based Java publishing framework for XML. See "Presenting XML documents" in the *XML Developer's Guide*.
- InternetBeans Express — a component library that makes it easier to create data-aware servlets and JSPs. See Chapter 7, "Using InternetBeans Express."
- JSTL (JavaServer Pages Standard Tag Library) — JSTL provides a standard way to accomplish common coding tasks using simple tags. More information is available at `http://java.sun.com`.
- Struts — an open source framework known as the Model 2, or Model-View Controller, approach to software design. See Chapter 8, "Using the Struts framework in JBuilder."

You can also make user-defined JSP tag libraries available in this wizard by adding them in the Configure Libraries dialog box. For more information, see "Using the Configure Libraries dialog box to manage user-defined frameworks" on page 6-6.

7 Specify a default Launch URI, if desired. This field might be filled in for you, if the selected framework supports a default launch URI. For example, if you choose the Cocoon framework, the Launch URI is set to `index.xml`. If a Launch URI is specified, the Web Run, Web Debug, and Web Optimize commands are available on the context menu for the WebApp node in the project pane.

8 Click OK to close the wizard and create the WebApp.

Figure 3.1 Web Application wizard



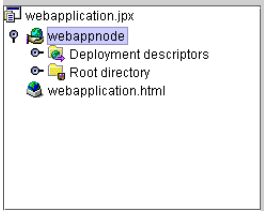
You can also use the Web Application wizard to import an existing web application. Fill in the Name field and use the Directory field to point to the location of the directory containing your existing web application. If

the web application is valid and the deployment descriptor(s) are correct for the currently configured web server, it can be run from within the JBuilder IDE immediately.

The WebApp and its properties

A Java-enabled web server locates a web application by its `ServletContext`, which maps to the WebApp. A WebApp is represented in the JBuilder IDE by a WebApp node. This is a node in the tree of the project pane which has the name of the WebApp.

Figure 3.2 Project pane showing a WebApp node



The WebApp node has two or three child nodes; a Root Directory for the application, a Deployment Descriptors node representing the `WEB-INF` directory for the WebApp, and an optional WAR file node.

You should place web content files (such as HTML, SHTML, and JSP files) in the WebApp's root directory or one of its subdirectories. Web content files are files which can be accessed directly by the client browser. Java resources used by the WebApp (such as servlets or JavaBeans) should have their source files in the source directory for the project. These files are not directly accessible by the client browser, but are called by something else, such as a JSP file. JBuilder's Servlet wizard, JSP wizard, and Web Start Launcher wizard make it easy to create web applications that follow these rules. Make sure to specify an existing WebApp when using these wizards.

Root directory

The root directory defines the base location for the web application. Every part of the web application will be located relative to the root directory. Web content files, such as HTML, SHTML, JSP, or image files, should be placed in this directory. Web content files are files which can be accessed directly by the client browser.

The files in the WebApp's root directory (and any subdirectories of the root directory) are automatically displayed in the Root Directory node of the project pane. Only files of the types you specify on the WebApp page of the WebApp Properties dialog box are displayed. The default file types are the ones you typically work with in a web application. This allows you to work with HTML files or image files using your favorite tools for working with those file types. Save these files in the WebApp's root directory, or one of its subdirectories. Then just click the project pane's Refresh button to see the current file set in JBuilder.

Deployment descriptors

Each WebApp must have a `WEB-INF` directory. This directory contains information needed by the web server when the application is deployed. This information is in the form of deployment descriptor files. These files have `.xml` extensions. They are shown in the Deployment Descriptors node in the project pane. The `WEB-INF` directory is actually a subdirectory of your WebApp's root directory. It isn't shown under the Root Directory node of the project pane because it is a protected directory that cannot be served by the web server. You can change this with the Directories page of the WebApp Properties dialog box.

The WebApp's Deployment Descriptors node always contains a deployment descriptor file called `web.xml`. This is required by all Java-enabled web servers and is created by JBuilder when you create the WebApp. If you are using the Struts framework in your WebApp, the `struts-config.xml` file will also be located in the Deployment Descriptors node.

Your web server may also require additional deployment descriptors which are unique to that particular web server. These can be edited in JBuilder and are created if they are listed as required by the currently configured web server plug-in. Check your web server's documentation to find out which deployment descriptors it requires.

JBuilder provides a deployment descriptor editor for editing the `web.xml` file. You can also edit server-specific deployment descriptors in JBuilder. Some mappings in the `web.xml` file are required for the WebApp to work as desired within the JBuilder IDE. These mappings will be written for you when you use JBuilder's wizards to create the pieces of your web application. Other mappings may be required when you deploy your application. For more information on deployment descriptors and the WebApp DD Editor, see "Editing deployment descriptors" on page 11-4."

WebApp properties

The WebApp node in the project pane has various properties you can edit.

To edit the WebApp node's properties, right-click the node and select Properties from the menu. The WebApp Properties dialog box has five pages:

- WebApp
- Directories
- Classes
- Dependencies
- Manifest

The WebApp page

The WebApp page of the WebApp Properties dialog box indicates the name of the WebApp, the directory location of the WebApp, and the directory location of the WAR file. There is a drop-down list indicating when to generate (or update) the WAR file, and a checkbox indicating whether or not to compress the archive. The setting for creating the archive corresponds to the "Build WAR" drop-down list in the Web Application wizard. You may wish to turn WAR generation off during development, and only enable it before you build the project for the final time prior to deployment.

The bottom of the WebApp page is divided into two tabbed pages: JSP/Servlet Frameworks and File Types Included.

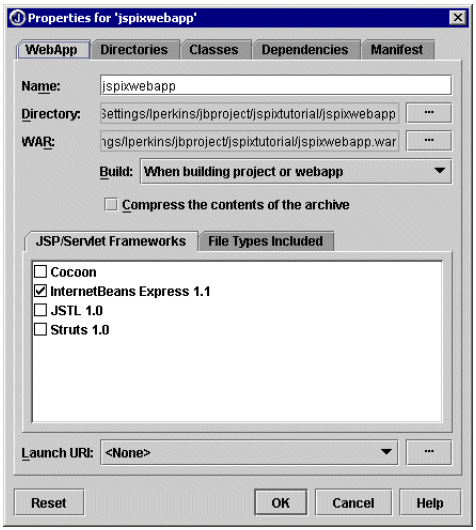
The JSP/Servlet Frameworks page allows you to select JSP/Servlet Frameworks to be used in the WebApp. The page automatically contains four frameworks that ship with JBuilder; Cocoon, InternetBeans Express, JSTL (JavaServer Pages Standard Tag Library), and Struts. If you use the Configure Libraries dialog box to add user-defined frameworks, these frameworks will also be available in this list.

The File Types Included page contains a list of file types to include for both file browsing and WAR generation. Only the file types which are checked will be included, based on file extension. The file extensions associated with each file type can be viewed or changed in the IDE Options dialog box, available from the Tools menu.

The Launch URI drop-down list is used to specify a URI to launch when running the WebApp. If a WebApp has a specified Launch URI, you can Web Run/Debug/Optimize directly from the context menu of the WebApp node in the project pane. Some frameworks, like Cocoon, have a preferred Launch URI, which is set automatically when the framework is chosen. This preferred Launch URI can be overridden. If the desired URI

is not shown in the drop-down list, click the ellipsis (...) button to browse to it.

Figure 3.3 WebApp page of WebApp Properties dialog box



See also

- "Creating a WebApp with the Web Application wizard" on page 3-3
- "Presenting XML documents" in the *XML Developer's Guide*
- Chapter 7, "Using InternetBeans Express"
- Chapter 8, "Using the Struts framework in JBuilder"
- "Using the Configure Libraries dialog box to manage user-defined frameworks" on page 6-6

The Directories page

The Directories page of the WebApp Properties dialog box allows you to specify custom subdirectories and specific files under the `WEB-INF` directory for inclusion in your project and your archives. It also allows you to exclude directories that should never be included, such as the `CVS` subdirectory that is present in every directory under CVS control. The Directories page provides the following possibilities:

- The ability to specify a list of directories to exclude. These exclusions are applied after any inclusions for `WEB-INF`.
- The option to treat `WEB-INF` as any other directory, so that it appears under the Root Directory folder in the project pane and displays the same file types as other directories in the WebApp.

- The option for specific subdirectories under `WEB-INF` to be included in your project and archives.

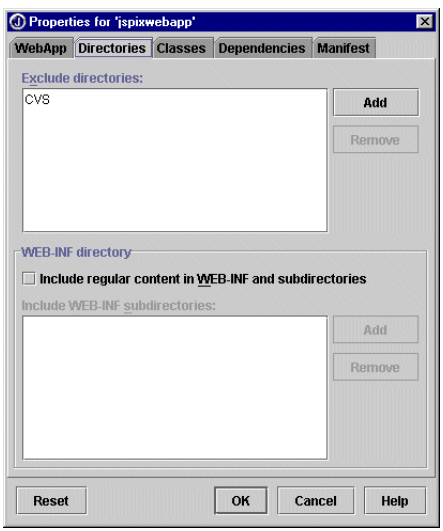
The reason subdirectories of `WEB-INF` must be explicitly added is that often it contains subdirectories that should never be included in a WAR file. Note that `WEB-INF/classes` and `WEB-INF/lib` are special directories which contain files that are generated by JBuilder and although they are never shown in the project pane, they are always included in WAR archives. You should not add files to `WEB-INF/classes` and `WEB-INF/lib`.

If `WEB-INF` is treated as a regular directory, then deployment descriptors will appear in both the Deployment Descriptors folder and under WEB-INF in the Root Directory folder in the project pane. Clicking on a deployment descriptor shown in either place will open the Deployment Descriptor Editor because the duplicate nodes in the project pane represent the same file.

Both lists on the Directories page match directory names using the same rules:

- Paths may contain one or more elements, separated by slashes.
- Each path element is considered separately.
- The `*` and `?` characters play their typical wildcard roles.
- When comparing the paths on disk, they are considered relative to the WebApp's root directory, or to the WebApp's `WEB-INF` directory.
- Paths that do not start with a slash are checked backwards, from the end. The most common case for this is a single path element with no leading slash, which means any path that ends with that element.
- Paths that start with a slash are checked forwards.

Figure 3.4 Directories page of WebApp Properties dialog box



The Classes page

The Classes page has options that control which Java classes and resources are copied into the `WEB-INF/classes` subdirectory that is relative to the project root on the disk (used during development) and the `WEB-INF/classes` subdirectory of the generated WAR file.

On the Classes page, you choose what parts of the project are included in the WAR file. You can also choose additional classes or files.

To specify the parts of the project you want to include,

1. Choose one of these options for Classes. If you choose Specified Only or Specified And Dependent, you must add the classes with the Add Classes button.
   - Specified Only
   - Specified And Dependent
   - All
2. Choose one of these options for Resources. If you choose Specified Only, you must add the resources with the Add Files button.
   - Specified Only
   - All

   For example, if you want to include all the project classes and resources in the archive, you would select All for both Classes and Resources. Selecting All for Resources adds all class resources in the project's source path. Note that class resources are distinct from web content files. To be considered a class resource, a file must be in the project's source path and its extension must be set to Copy on the Resource tab of the Build page in Project Properties.

   Web content files should not be in the project's source path. Web content should be placed in the WebApp's root directory or one of its subdirectories. It is copied to the corresponding location in the WAR file when the WAR file is built. File types which are considered web content are selected on the WebApp page of the WebApp Properties dialog box.

Caution If you select All for both Classes and Resources, every class file in your output path is included in the WAR file. This may mean that class files and resources will be included which are not necessary. Be aware that generating a WAR with these settings could take some time and become very large.

If you don't want to include any dependencies in your archive and only specified resources, you would choose Specified Only for both Classes and Resources and then add the classes and resources you want with the Add Classes and Add Files buttons. To add classes and files, classes must be on the project output path and files must be on the project source path.

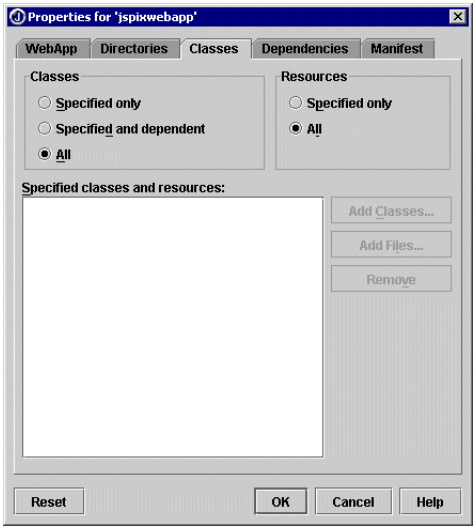
3 Choose the Add Classes button if you selected Specified Only or Specified And Dependent in the Classes category. Then select the classes to add to your archive. The classes must be on the project output path. If you choose Specified And Dependent, additional class dependencies are included in the archive.

   Remember that the classes are copied to `WEB-INF/classes` or its subdirectories. The classes you select should therefore be classes that are accessed on the server side, not classes that need to be served by the web server. For example, servlet classes should be selected, but not applet classes.

4 Choose the Add Files button if you selected Specified Only in the Resources category. Then select the files to add to your archive. The files must be on the project source path. They will be copied to their corresponding location in `WEB-INF/classes`. For example, `/myproject/src/com/whatever/whatever.properties` is copied to `/WEB-INF/classes/com/whatever/whatever.properties`.

Note The Add Files dialog box can't look inside archive files. If a file or package you need is inside an archive file, extract it first to your source folder, then add it using the Add Files button.

Figure 3.5 Classes page of WebApp Properties dialog box



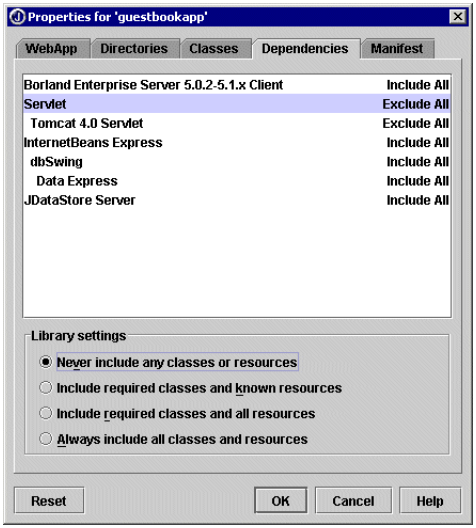
The Dependencies page

The Dependencies page allows you to specify what to do with library dependencies for your WebApp. You generally want to include required libraries. Unlike regular archives, library JAR files are stored as-is in the `WEB-INF/lib` directory (when the Library Setting on this page is Always Include All Classes And Resources — the recommended setting).

Libraries are set to Include All by default, except for the Servlet library, which shouldn't be included in a WAR file (it is provided by the web server). You should also exclude any other libraries provided by the server and any libraries which aren't used in your WebApp. Ensuring that these unnecessary libraries are excluded will make the WAR file smaller and make building your WebApp faster.

The Dependencies page of the WebApp Properties dialog box looks the same as the Dependencies page for any type of archive. See the online Help topic "Archive Builder wizard, Determine what to do with library dependencies" for more information.

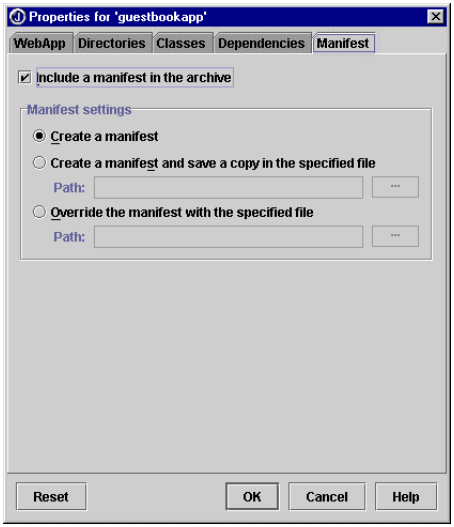
Figure 3.6 Dependencies page of WebApp Properties dialog box



The Manifest page

The Manifest page of the WebApp Properties dialog box is the same as the Manifest page for any type of archive. See the online Help topic "Archive Builder wizard, Set archive manifest options" for more information.

Figure 3.7 Manifest page of WebApp Properties dialog box

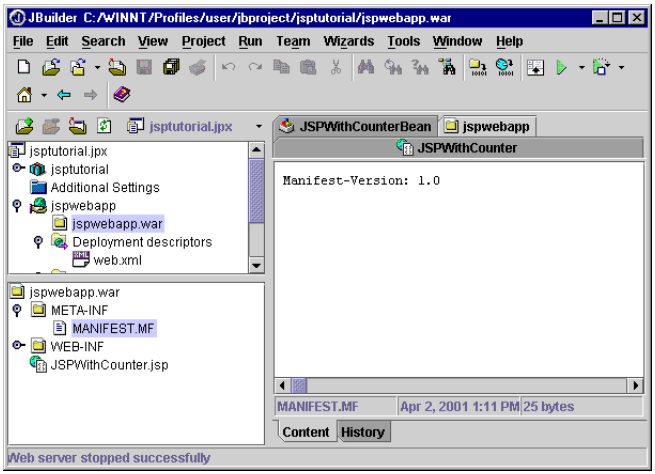


The WAR file

The WAR file is an archive for the web application. Putting all the files and resources needed for the WebApp into the WAR file makes deployment easier. Assuming everything is set up correctly and all the necessary resources are in the WAR file, all you should need to do for deployment is to copy the WAR file to the correct location on the web server.

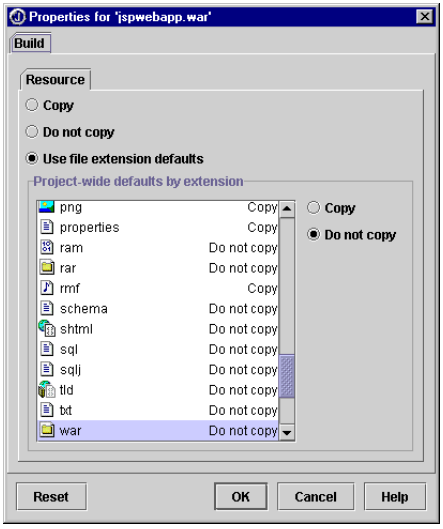
If a WAR file is present, a node representing the WAR file appears under the WebApp node in the project tree. Opening the WAR file node displays the contents of the WAR file in the structure pane, and also displays the WAR file viewers in the AppBrowser. The WAR file's viewers consist of the Content view and the History view.

Figure 3.8 WAR file node open in JBuilder IDE



The WAR file node has a Properties dialog box, accessible by right-clicking the node and selecting Properties. In the Properties dialog box, you can specify whether the WAR file is considered a Java resource. This is the same Resource page which is available in the Build Properties dialog box for all non-Java file types. For more information on Resource properties, see the "Resource" section of the online Help topic "Build page (Project Properties dialog box)." Note that the more important properties for the WAR file are found on the WebApp page of the WebApp Properties dialog box.

Figure 3.9 WAR file properties dialog



To have JBuilder create a WAR file for your web application, first you must have a WebApp node in your project. You can create this WebApp with the Web Application wizard, available in the object gallery, or you can use the default WebApp.

You can tell JBuilder to automatically create or update a WAR file whenever the project is built, whenever the WebApp is built, both, or neither. To change the WebApp's WAR file settings,

1. Right-click the WebApp node in the project pane and select Properties from the menu.
2. Click the WebApp tab of the WebApp Properties dialog box.
3. Select the desired Build option from the drop-down list. The following Build choices are available.
   - When Building Project Or WebApp
   - When Building WebApp Only
   - When Building Project Only
   - Never
4. Verify that the name and location of your desired WAR file is correct.

You have the option to compress the WAR file if you wish.

Now that you have a WAR file associated with your WebApp, keep in mind that the various pieces of your web application must be associated with that WebApp to be added to the WAR file. Many of the wizards have a WebApp drop-down list for you to select your WebApp when creating these pieces.

Applets in a WAR file

You can add an applet to the WAR file, but it requires some extra work. For the applet to work in the WebApp, the classes must be accessible. This means the HTML file and the classes for the applet must be located under the WebApp's root directory or one of its subdirectories. They cannot be under the WEB-INF directory or its subdirectories.

The Applet wizard doesn't support putting the applet files within a WebApp, so you will have to move the applet's HTML file and compiled class files to the WebApp's root directory (or one of its subdirectories). Depending on where you move the class files, you may need to edit the `<applet>` tag's `codebase` attribute so that it can still find them.

If you include loose (unarchived) applet class files in a WebApp, make sure to select Java Class File as one of the included file types on the WebApp page of the WebApp Properties dialog box. Otherwise, the class files won't be included in the WAR.

You can also put an applet's JAR file in the WAR file. The same rules apply. The JAR file needs to be accessible. This means it needs to go under the WebApp's root directory, or one of its subdirectories, but not under the `WEB-INF` directory.

Chapter

4

Working with servlets

Web Development is a feature of JBuilder Enterprise

Java servlets provide a protocol and platform-independent method for building web-based applications without the performance limitations of CGI programs. A servlet runs inside a web server and, unlike an applet, does not need a graphical user interface. A servlet interacts with the servlet engine running on the web server through requests and responses. A client program, which can be written in any programming language, accesses the web server and makes a request. The request is then processed by the web server's servlet engine, which passes it on to the servlet. The servlet then sends a response through the web server back to the client.

Today, servlets are a popular choice for building interactive web applications. A variety of third-party web servers with servlet engine extensions are available: Tomcat (the Servlet/JSP API reference implementation), Sun iPlanet and others. Web servers with servlet engines, also known as servlet containers, can also be integrated with web-enabled application servers. JBuilder Enterprise includes the Borland Enterprise Server and also provides support for WebLogic, WebSphere and Sun iPlanet.

Note JBuilder Enterprise includes Tomcat as the default web server.

One critical advantage to the servlet technology is speed. Unlike CGI programs, servlets are loaded into memory once and run from memory after the initial load. Servlets are spawned as a thread, and are, by nature, multi-threaded. And, since they are based on the Java language, they are platform independent.

JavaServer Page (JSP) technology is an extension of the servlet technology created to specifically support authoring of HTML and XML pages. JSP technology makes it easier to combine fixed or static template data with dynamic content. Even if you're comfortable writing servlets, there are several compelling reasons to investigate JSP technology as a complement to your existing work. For more information on writing JSPs, see Chapter 6, "Developing JavaServer Pages."

For more information on servlets, see the following web sites. These web addresses and links are valid as of this printing. Borland does not maintain these web sites and can not be responsible for their content or longevity.

- Java Servlet Technology at `http://java.sun.com/products/servlet/index.html`
- Java Servlet Technology: White Paper at `http://java.sun.com/products/servlet/whitepaper.html`
- Java Servlet Technical Resources page at `http://java.sun.com/products/servlet/technical.html`
- Java Servlet Third-Party Resources page at `http://java.sun.com/products/servlet/resources.html`
- The Servlet Trail of The Java tutorial at `http://java.sun.com/docs/books/tutorial/servlets/index.html`

You can also look at the following tutorials for information on creating servlets in JBuilder:

- Chapter 16, "Tutorial: Creating a simple servlet"
- Chapter 17, "Tutorial: Creating a servlet that updates a guestbook"
- Chapter 19, "Tutorial: Creating a servlet with InternetBeans Express"

Servlets and JSPs

Both JSP and servlet technology have merits. How do you decide which to use in a given situation?

- *Servlets* are a programmatic tool and are best suited for low-level application functions that don't require much presentation logic.
- *JSPs* are a presentation-centric, declarative means of binding dynamic content and logic. JSPs should be used to handle the HTML representation that is generated by a page. They are coded in HTML-like pages with structure and content familiar to web content providers. However, JSPs provide far more power than ordinary HTML pages. JSPs can handle application logic through the use of JavaBeans components, Enterprise JavaBeans (EJB) components, and custom tags.

JSPs themselves can also be used as modular, reusable presentation components that can be bound together using a templating mechanism.

JSPs are compiled into servlets, so theoretically you could write servlets to support your web-based applications. However, JSP technology was designed to simplify the process of creating pages by separating web presentation from web content. In many applications, the response sent to the client is a combination of template data and dynamically-generated data. In this situation, it is much easier to work with JSP pages than to do everything with servlets.

Servlets and web servers

In 1999, Sun Microsystems delivered the then-current latest versions of the Servlet and JSP APIs to the Apache Software Foundation. Apache, along with Sun and a variety of other companies, developed and released the official JSP/Servlet reference implementation, called Tomcat. It is the only reference implementation available. Tomcat is available free to any company or developer. For more information about the Apache Software Foundation and Tomcat, go to `http://jakarta.apache.org`. For more information about JBuilder and web servers, see Chapter 9, "Configuring your web server."

JBuilder Enterprise delivers Tomcat to use as your web server, so that you can successfully develop and test your servlets and JSPs within the JBuilder development environment. Many other web servers support Sun's servlet and JSP APIs. For a list of these products, see the "Servers and Engines" topic on Sun's Servlet Technology Industry Momentum page at `http://java.sun.com/products/servlet/industry.html`.

The servlet API

The servlet API is contained in the `javax.servlet` package. All servlets must directly or indirectly implement the `javax.servlet.Servlet` interface. This interface allows the servlet to run in a servlet engine (an extension to a web server). It also defines the servlet's lifecycle. The servlet API is embedded into many web servers, including Tomcat, the default web server provided with JBuilder. The following table lists the more

commonly used classes and interfaces in the `servlet` package and provides a brief description of each.

Table 4.1 Overview of Servlet API

Name	Class or Interface	Description
`GenericServlet`	Class	Defines a generic, protocol-independent servlet.
`RequestDispatcher`	Interface	Defines an object that receives requests from the client and sends them to any resource (such as a servlet, HTML file, or JSP file) on the server.
`Servlet`	Interface	Defines methods that all servlets must implement.
`ServletConfig`	Interface	A servlet configuration object used by a servlet container to pass information to a servlet during initialization.
`ServletContext`	Interface	Defines a set of methods that a servlet uses to communicate with its servlet container, for example, to get the MIME type of a file, dispatch requests, or write to a log file.
`ServletException`	Class	Defines a general exception a servlet can throw when it encounters difficulty.
`ServletInputStream`	Class	Provides an input stream for reading binary data from a client request, including an efficient `readLine` method for reading data one line at a time.
`ServletOutputStream`	Class	Provides an output stream for sending binary data to the client.
`ServletRequest`	Interface	Defines an object to provide client request information to a servlet.
`ServletResponse`	Interface	Defines an object to assist a servlet in sending a response to the client.
`SingleThreadModel`	Interface	Ensures that servlets handle only one request at a time.
`UnavailableException`	Class	Defines an exception that a servlet throws to indicate that it is permanently or temporarily unavailable.

The servlet.HTTP package

You use the `javax.servlet.http` package to create servlets that support HTTP protocol and HTML generation. The HTTP protocol uses a set of text-based request and response methods (HTTP methods), including:

- `GET`
- `POST`
- `PUT`
- `DELETE`
- `HEAD`
- `TRACE`
- `CONNECT`
- `OPTIONS`

The HttpServlet class implements these HTTP methods. To start, you simply extend HttpServlet and override either the doGet() or doPost() methods. For further control, you can also override the doPut() and doDelete() methods. If you create a servlet with JBuilder's Servlet wizard, you can select which methods you want to override, and JBuilder creates the skeleton code for you.

The following table lists the more commonly used classes and interfaces in the javax.servlet.http package and provides a brief description of each.

Table 4.2 Commonly used servlet package classes and interfaces

Name	Class or Interface	Description
Cookie	Class	Creates a cookie, a small amount of information sent by a servlet to a web browser, saved by the browser, and later sent back to the server.
HttpServlet	Class	Provides an abstract class to be subclassed to create an HTTP servlet suitable for a web site.
HttpServletRequest	Interface	Extends the ServletRequest interface to provide request information for HTTP servlets.
HttpServletResponse	Interface	Extends the ServletResponse interface to provide HTTP-specific functionality in sending a response.
HttpSession	Interface	Provides a way to identify a user across more than one page request or visit to a web site and to store information about that user.
HttpSessionBindingEvent	Class	Sent to an object that implements HttpSessionBindingListener when the object is bound to or unbound from the session.
HttpSessionBindingListener	Interface	Causes an object to be notified when it is bound to or unbound from a session.

The servlet lifecycle

The javax.servlet.Servlet interface contains the servlet life-cycle methods. You implement these methods to

- Construct the servlet and initialize it with the init()method.
- Handle calls from clients to the service() method.
- Take the servlet out of service, destroy it with the destroy() method, and perform garbage-collection tasks.

Constructing and initializing the servlet

When the servlet engine starts or when a servlet needs to respond to a request, the `init()` method is called by the servlet engine to tell a servlet that it is being placed into service. The servlet engine calls `init()` only once after instantiating the servlet. The `init()` method must complete before the servlet can receive requests.

The `init()` method's `ServletConfig` parameter is an object that contains the servlet's configuration and initialization parameters. (After the servlet has been initialized, you can use `getServletConfig()` to retrieve this information.)

Once the servlet is loaded into memory, it can reside in a local file system, a remote file system, or on a network.

Handling client requests

The servlet engine calls the `service()` method to allow the servlet to respond to a request. The request and response objects are passed as parameters to the `service()` method when a client makes a request.

The servlet can also implement the `ServletRequest` and/or the `ServletResponse` interfaces to allow the servlet access to request parameters and response data. Request parameters include data or protocol methods. Response data includes response headers and status codes.

Servlets and multi-threading

Typically, servlets are multi-threaded, allowing a single servlet to handle multiple requests concurrently. As a developer, you must make any shared resources — such as files, network connections, and the servlet's class and instance variables — thread safe. For more information on threads and thread safety, see "Threading techniques" in *Getting Started with Java*.

Note that you can create a servlet with a single thread, using the `SingleThreadModel` interface. This interface allows a servlet to respond to only one request at a time. Usually, this is not practical for servlets. If a servlet is restricted to one thread, the web server will queue requests and start another instance of the servlet to service the demand.

Destroying a servlet

A servlet engine does not keep a servlet loaded for any specified period of time or for the life of the server. Servlet engines can retire servlets at any

time. Because of this, you should program your servlet so that it does not store state information. To release resources, use the `destroy()` method.

Servlet-aware HTML

Servlets can easily generate HTML-formatted text, allowing you to use servlets to dynamically generate or modify HTML pages. With servlet technology, you do not need to use scripting languages. For example, you can use servlets to personalize a user's experience on a web site by continually modifying the same HTML page.

If your web server has server-side include (SSI) functionality, you can use the `<servlet>` tag to preprocess web pages. This tag must be in a file with an `.shtml` extension. The `<servlet>` tag tells the web server that a pre-configured servlet should be loaded and initialized with the given set of configuration parameters. The output from the servlet is included in an HTML-formatted response. Like an `<applet>` tag, you can also use the `class` and `codebase` attributes to specify the servlet's locations.

An example of a `<servlet>` tag is as follows:

```
<servlet>
  codebase=""
  code="dbServlet.Servlet1.class"
  param1=in
  param2=out
</servlet>
```

JBuilder's Servlet wizard can create a `<servlet>` tag in an `.shtml` file.

HTTP-specific servlets

Servlets used with HTTP protocol (by extending `HTTPServlet`) may support any HTTP method. They can redirect requests to other locations and send HTTP-specific error messages. Additionally, they can access parameters which were passed through standard HTML forms, including the HTTP method to be performed and the URI that identifies the destination of the request:

```
String method  = request.getMethod();
String uri     = request.getRequestURI();
String name    = request.getParameter("name");
String phone   = request.getParameter("phone");
String address = request.getParameter("address");
String city    = request.getParameter("city");
```

For HTTP-specific servlets, request and response data are provided as MIME-formatted data. The servlet specifies the data type, and writes data encoded in that format. This allows a servlet to receive input data of one type and return data in the form appropriate for that request.

How servlets are used

Because of the robust servlet API functionality, servlets can be used in a variety of ways:

- As part of an order entry and processing system, working with customer, product, and inventory databases. For example, a servlet could process data, such as credit-card data, POSTed over HTTPS using an HTML `<form>` tag.
- In conjunction with an applet, as part of a corporate intranet to track employee information or salary histories.
- As part of a collaborative system, such as a messaging system, where several or more servlets are handling multiple requests concurrently.
- As part of a load-balancing process, where servlets forward requests in a chain.
- As an HTML-aware servlet, to insert formatted data into a web page from a database query or a web search, or to create individually targeted advertising banners.

Deploying servlets

Servlets can be deployed stand-alone to a production web server. They can also be deployed as a J2EE module. A J2EE module consists of one or more J2EE components of the same type (web, EJB, client etc.) that share a single deployment descriptor.

For more information and deployment tips, see Chapter 11, “Deploying your web application.”

Chapter

5

Creating servlets in JBuilder

Web Development is a feature of JBuilder Enterprise

In JBuilder Enterprise, you can use the Servlet wizard to create a Java file that extends `javax.servlet.http.HttpServlet`. The JBuilder Servlet wizard creates servlets that generate the following content types:

- HTML — HyperText Markup Language.
- XHTML — a reformulation of HTML 4.0 as an application of XML.
- WML — Wireless Markup Language.
- XML — eXtensible Markup Language.

With the Servlet wizard, you can also create standard servlets, filter servlets, or listener servlets (if your web server supports the Java Servlets v2.3 specification).

Servlet wizard options

The Servlet wizard makes creating servlets easy. To start the wizard, choose File | New. Click the Web tab of the object gallery, select Servlet and click OK. (A server must first be enabled for your project.)

Choose Servlet Name and Type page

On the first step of the Servlet wizard, you can choose the package the servlet belongs to. You can also enter the name of the servlet in the Class field. If you want to replicate the header comments added to other classes in the project (see the Project wizard online Help topic), choose the Generate Header Comments option.

The Single Thread Model option implements the `SingleThreadModel` interface. Choosing this option may make your servlet a little less efficient, as the web server will queue requests and start another instance of the servlet to service the demand.

Select the name of the WebApp you want to run this servlet under from the WebApp drop-down list. Any web content files, such as a servlet's SHTML file, are placed in the WebApp directory.

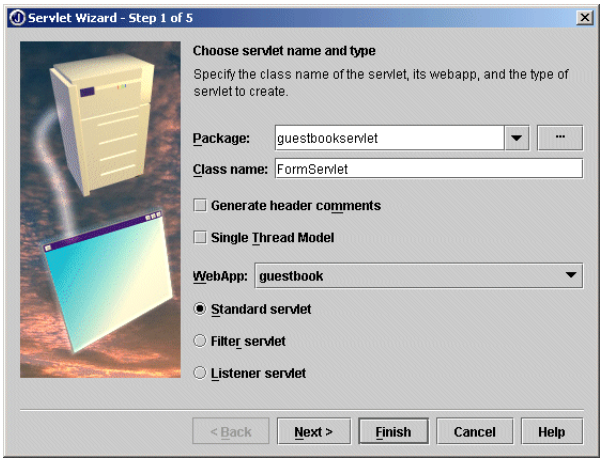
The options at the bottom of the dialog box allow you to choose the type of servlet you're creating.

Note These options are only available if your web server supports the Java Servlets v2.3 specifications. (Tomcat 4.x is the reference implementation of these specifications.) Otherwise a standard servlet is created by default.

Table 5.1 Servlet type options

Servlet Type	Description
Standard Servlet	A servlet that is not a filter or a listener. The Standard Servlet option is always available whether you select the `<default>` WebApp or a named WebApp. When this option is selected for the default WebApp, the servlet name (see "Enter WebApp Details page" on page 5-6) defaults to the simple class name of the servlet in lowercase. If the WebApp is a named WebApp, the servlet URL (also on the Enter WebApp Details page) defaults to the URL pattern: `/servletname` (all lowercase).
Filter Servlet	A servlet that acts as a filter for another servlet or for the WebApp. In addition to the name, you must choose a URL pattern for the other servlet (which must have a name). If this option is selected, the Enter WebApp Details page, where you name the servlet and specify the filter mapping, is the only other available Servlet wizard page.
Listener Servlet	A servlet that is added to the WebApp's list of listeners. If this option is selected, the Enter Listener Servlet Details page is the only other available Servlet wizard page.

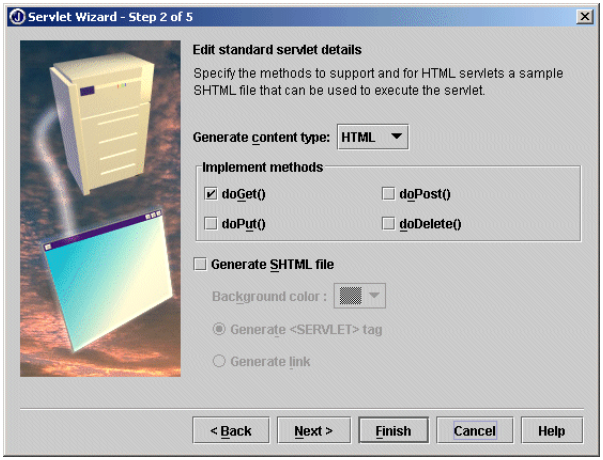
Figure 5.1 Servlet wizard — Choose Servlet Name and Type page



Enter Standard Servlet Details page

If you've selected Standard Servlet as the servlet type on the Choose Servlet Name and Type page of the Servlet wizard, the Enter Standard Servlet Details page is Step 2 of the wizard. This page allows you to select the servlet's content type, the methods to implement, and the SHTML file to generate.

Figure 5.2 Servlet wizard — Enter Standard Servlet Details page



Generate Content Type option

You use the Generate Content Type drop-down list to choose the content type for the servlet. The options include:

- HTML — HyperText Markup Language. A markup language for hypertext documents on the Internet. HTML enables the embedding of images, sounds, video streams, form fields, references to other objects with URLs, and basic text formatting.
- XHTML — A reformulation of HTML 4.0 as an application of XML. The tags and attributes are almost identical to HTML, but XHTML is a stricter, tidier version of HTML. For example, all XHTML tags are lowercase and all open tags must have a closing tag. XHTML allows web designers to move toward a more modular and extensible web based on XML while maintaining compatibility with today's HTML 4 browsers.

  In the generated `.java` class file, a line of code indicating that the DOC TYPE is XHTML is added, and looks like this:

  ```
  private static final String DOC_TYPE = "<!DOCTYPE html
    PUBLIC \"-//W3C//DTD XHTML 1.0 Strict//EN\"\n" +
    "  \"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd\">";
  ```

  Similar text is added to the top of the SHTML file, if it was generated.

- WML — Wireless Markup Language. WML files are regular text files containing XML content. Due to the restricted bandwidth and memory capacity of the mobile devices, this content has to be compiled into compact binary form before it can be presented on a Wireless Application Protocol (WAP) enabled device.

  For WML servlets, only a `.java` class file is generated. There is no option for an SHTML file with a WML-type servlet. The `.java` class file contains a line of code indicating that the DOC TYPE is WML and the CONTENT TYPE is WAP. It looks like this:

  ```
  private static final String CONTENT_TYPE = "text/vns.wap.wml";
  private static final String DOC_TYPE = "<!DOCTYPE wml
    PUBLIC \"-//WAPFORUM//DTD WML 1.2//EN\"\n" +
    "  \"http://www.wapforum.org/DTD/wml12.dtd\">";
  ```

  Code to run the servlet's GET request as a WML-servlet is added as well, and looks like this:

  ```
  out.println("<?xml version=\"1.0\"?>");
  out.println(DOC_TYPE);
  out.println("<wml>");
  out.println("<card>");
  out.println("<p>The servlet has received a GET. This is the
    reply.</p>");
  out.println("</card></wml>");
  ```

- XML — eXtensible Markup Language. A markup language that allows you to define the tags (markup) needed to identify the data and text in XML documents. For more information on JBuilder's XML support, see the "Introduction" to the *XML Developer's Guide*.

  For XML servlets, a `.java` class file is generated. The option to create an SHTML file is eliminated. The `.java` file includes code to identify the servlet as an XML-servlet. It looks like this:

  ```
  private static final String CONTENT_TYPE = "text/xml";
  ```

  The servlet's `doGet()` method is modified from the HTML version with the following code:

  ```
  out.println("<?xml version=\"1.0\"?>");
  if (DOC_TYPE != null) {
    out.println(DOC_TYPE);
  }
  ```

Implement Methods options

This area of the Servlet wizard provides options for overriding the standard servlet methods. `HttpServlet` provides an abstract class that you can subclass to create an HTTP servlet, which receives requests from and sends responses to a web client. When you subclass `HttpServlet`, you must override at least one method, usually one of the following. For more information on the methods, refer to the Servlet documentation at `http://java.sun.com/products/servlet/index.html`.

The following methods can automatically be generated in your servlet:

- `doGet()` — Allows a client to read information from the web server, passing a query string appended to a URL to tell the server what information to return. A GET also happens when you type in an address, click a link, or use a bookmark. Override this method if your servlet supports HTTP GET requests.
- `doPost()` — Allows the client to send data of unlimited length to the web server. A POST also happens when you click a browser's submit button. Override this method if your servlet supports HTTP POST requests.
- `doPut()` — Allows a client to place a file on the server and is similar to sending a file to the server by FTP. Override this method if the servlet supports HTTP PUT requests.
- `doDelete()` — Allows a client to remove a document or web page from the server. Override this method if the servlet supports HTTP DELETE requests.

SHTML File Details options

The SHTML file details options are displayed if you selected either HTML or XHTML from the Generate Content Type drop-down list. If you want to call the servlet from an HTML page, as described in "Calling a servlet from an HTML page" on page 5-12, select the Generate SHTML File option. A SHTML file, with the same name as the servlet, is added to the project. The SHTML file is placed in the WebApp directory.

If you want to run the servlet directly, as described in "Invoking a servlet from a browser window" on page 5-11, do not select the Generate SHTML File option.

Choose the SHTML file's background color from the Background Color drop-down list. To connect to the servlet from the SHTML file, you can either use either a `<servlet>` tag or an `<a href>` link tag.

- If you choose the `<servlet>` tag option, the SHTML file runs the servlet using a `<servlet>` tag. The tag contains a `codebase` and `code` property, similar to an applet. By default, the `codebase` is set to the current folder and the `code` is set to the servlet's class name.
- If you choose to run the servlet from a link, an `<a href>` tag is placed in your SHTML file. The tag links to the servlet's URL pattern. Note that you must go to the next page of the wizard in order for the URL pattern mapping to be generated. If you click Finish on this page, a link to the class name is created.

Enter WebApp Details page

This page allows you to quickly define basic WebApp mappings. (The selected WebApp is used to run or debug the servlet.) The Servlet wizard automatically adds the servlet mappings to the Servlet or Filters sections of the `web.xml` deployment descriptor file. For more information, see "Servlets" on page 11-2, or "Filters page" on page 12-4.

- If you've selected Standard Servlet as the servlet type on the Choose Servlet Name and Type page of the wizard, this is Step 3.
- If you've selected Filter Servlet, this is Step 2 and the final Servlet wizard step.

For more information on mapping servlet names and URL patterns, see "How URLs run servlets" on page 10-9.

The Name field applies to both standard and filter servlets. This field is displayed for standard servlets running in both the `<default>` and named WebApp and for filter servlets.

The wizard suggests a servlet-name that is the same as the classname in lowercase. The same name is suggested for the URI and is preceded with a forward slash (all URIs must start with a forward slash). However, you

can change these names to any other name, except an invalid one. For example, you could have the URL pattern `/myhtmlfile.html` run the servlet named `custlist` which is actually the class `package.subpackage.CustomerList`.

The entry in the Name field is the name used to identify the servlet in the `web.xml` file. The name is displayed in the Servlets node in the structure pane for the `web.xml` file. It is also used to map the URL pattern to the servlet. The URL pattern maps explicitly to a servlet via the servlet name — there is no implicit mapping between the servlet name and the URL. For example, the following URL:

```
http://localhost:8080/guestbook/formservlet
```

generates the following URI (the URI remains after the protocol, hostname, and port number are removed from the URL):

```
/guestbook/formservlet
```

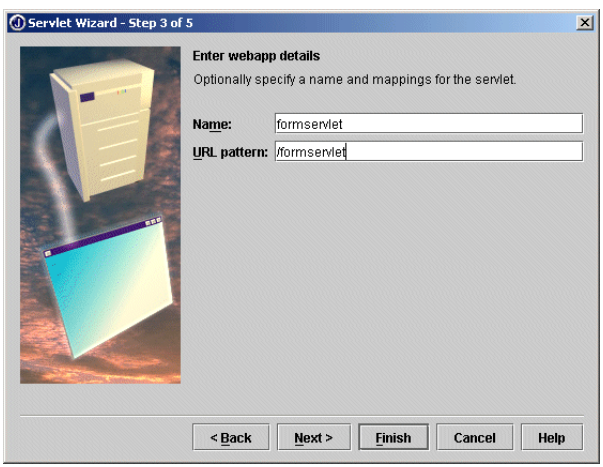
The beginning of the URI is matched against all known WebApp names. A match causes the remaining part of the URI to be directed to that particular WebApp/context.

Note If there is no WebApp/context match, then the URI is passed as-is to the default WebApp.

The URI is then compared to all known URL patterns. Those patterns are listed on the Servlets page of the WebApp DD Editor. A given pattern has a corresponding servlet name. That servlet name identifies the "servlet," which could also be a JSP file, since JSPs are compiled to servlets.

Warning URL mappings will not be generated if you press the Finish button on one of the previous wizard steps. You must use the Enter WebApp Details page to generate a URL mapping.

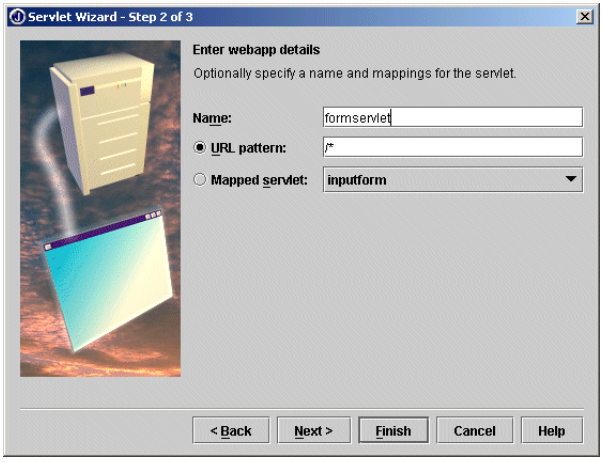
Figure 5.3 Servlet wizard — Standard servlet, Enter WebApp Details page



For filter servlets, you can choose how the servlet is mapped; it can be mapped with either the URL Pattern or the Mapped Servlet option.

The Mapped Servlet drop-down list allows you to choose another servlet in your project that is filtered by this servlet. (This field defaults to the lowercase name of the first servlet in your project, preceded by a forward slash. For example, if your project already contains `Servlet1.java` and `Servlet2.java` which are both standard servlets, the Mapped Servlet field would default to `servlet1`.) The drop-down list shows, in alphabetical order, all other servlets in the project that have been previously mapped. If there are none, the option is disabled.

Figure 5.4 Servlet wizard — Filter servlet, Enter WebApp Details page

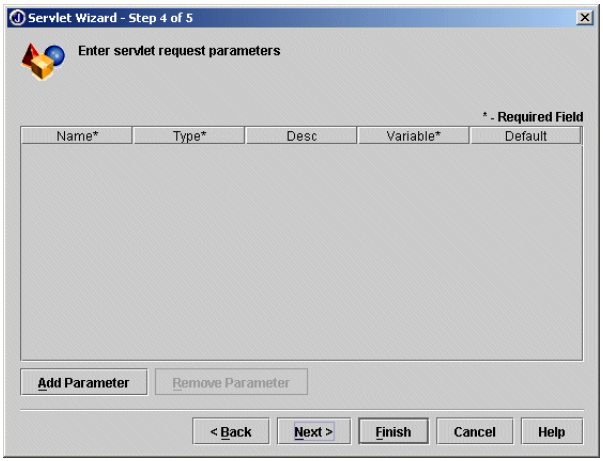


Enter Servlet Request Parameters page

The Enter Servlet Request Parameters page of the Servlet wizard is where you enter servlet parameters. Parameters are values passed to the servlet. The values might be simple input values. However, they could also affect the runtime logic of the servlet. For example, the user-entered value could determine what database table gets displayed or updated. Alternatively, a user-entered value could determine the background color of an HTML page that the servlet generates.

The servlet in Chapter 16, "Tutorial: Creating a simple servlet," uses a parameter of type `String` to allow entry of the user's name. The parameter name in the SHTML file is `UserName`; the corresponding variable, used in the servlet class file, is `userName`.

Figure 5.5 Servlet wizard — Enter Servlet Request Parameters page

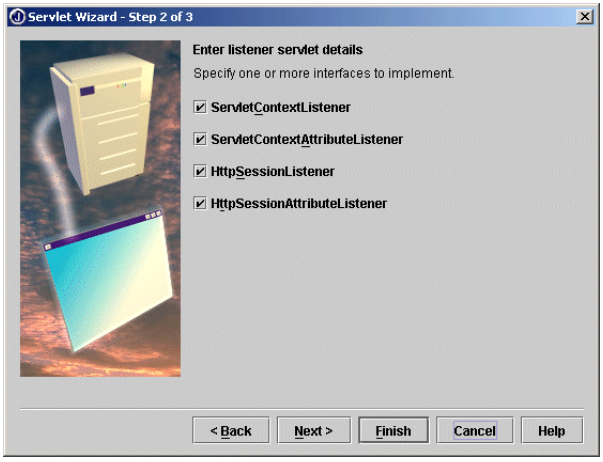


Enter Listener Servlet Details page

This page is available only if you've selected the servlet type of Listener Servlet on the Choose Servlet Name and Type page of the Servlet wizard. You use this page to implement one or more servlet listener interfaces. The corresponding methods are added to the servlet.

The Servlet wizard automatically adds the selected listeners to the Listeners section of the `web.xml` deployment descriptor file. For more information, see "Listeners page" on page 12-6.

Figure 5.6 Servlet wizard — Enter Listener Servlet Details page



Define Servlet Configuration page

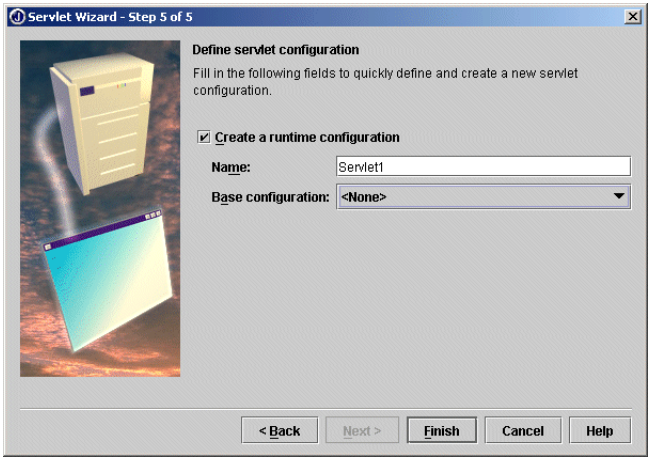
This page creates a runtime configuration for your servlet. You only need to create a configuration for the servlet that starts the flow of processing in your application, but not for those that are called by other servlets.

To create the configuration, make sure the Create A Runtime Configuration option is checked. Enter the configuration name in the Name field. You can use an existing configuration as a base — select it in the Base Configuration drop-down list.

For more information, see:

- "Creating a runtime configuration" on page 10-2
- "Setting runtime configurations" in the "Running Java programs" chapter of *Building Applications with JBuilder*

Figure 5.7 Servlet wizard — Runtime configuration page



Invoking servlets

Invoking a servlet from a browser window

A servlet can be called directly by typing its URL into a browser's location field. However, this only works when the web server has a server-invoker servlet (a servlet whose only purpose is to run other servlets). Tomcat has an invoker servlet, but many other servers do not. Using an invoker servlet is not recommended because it is not portable.

If you choose to use an invoker servlet, The general form of the servlet URL, where *servlet-class-name* corresponds to the class name of the servlet, is:

```
http://machine-name:port-number/servlet/servlet-class-name
```

For example, a URL in the format of any of the following examples should enable you to run a servlet:

- `http://localhost:8080/servlet/Servlet1` (running locally on your computer)
- `http://www.borland.com/servlet/Servlet1` (running from this URL)
- `http://127.0.0.1/servlet/Servlet1` (running from this IP address)

Note If you omit the port number, the HTTP protocol defaults to port 80. The first URL in the example above would work in the IDE if you set the runtime configuration's port to 80 and you're not already running a web server on this port. The other examples would work in a real-life situation, after the web application had been deployed to a web server.

Servlet URLs can contain queries for HTTP GET requests. For example, the servlet in Chapter 16, "Tutorial: Creating a simple servlet," can be run by entering the following URL for the servlet:

```
http://localhost:8080/servlet/simpleservlet.Servlet1?userName=Mary
```

`simpleservlet.Servlet1` is the fully qualified class name. The `?` indicates that a query string is appended to the URL. `userName=Mary` is the parameter name and value.

If `simpleservlet.Servlet1` is mapped to the URL pattern `/firstservlet`, you would enter:

```
http://localhost:8080/firstservlet?userName=Mary
```

For more information on how servlets are run, see "How URLs run servlets" on page 10-9.

Calling a servlet from an HTML page

To invoke a servlet from within an HTML page, just use the servlet URL in the appropriate HTML tag. Tags that take URLs include those that begin anchors and forms, and meta tags. Servlet URLs can be used in HTML tags anywhere a normal URL can be used, such as the destination of an anchor, as the action in a form, and as the location to be used when a meta tag directs that a page be refreshed. This section assumes knowledge of HTML. If you don't know HTML you can learn about it through various books or by looking at the *HTML 4.0 Reference Specification* on the web at `http://www.w3.org/TR/html4/#minitoc`.

For example, in Chapter 16, "Tutorial: Creating a simple servlet," the SHTML page contains an anchor with a servlet URL pattern as a destination:

```
<a href="/servlet1">Click here to call Servlet: Servlet1</a><br>
```

HTML pages can also use the following tags to invoke servlets:

- A `<form>` tag

  ```
  <form action="http://localhost:8080/servlet1  method="post">
  ```

- A meta tag that uses a servlet URL as part of the value of the `http-equiv` attribute.

  ```
  <meta http-equiv="refresh" content="4;url=http://localhost:8080/servlet1">
  ```

Note that you can substitute the servlet's fully qualified class name for the URL pattern. For example, if the servlet class name is `Servlet1`, you can use the following `<form>` tag to run the servlet:

```
<form action="http://localhost:8080/servlet/simpleservlet.Servlet1"  method="post">
```

Internationalizing servlets

Servlets present an interesting internationalization problem. Because the servlet outputs HTML source to the client, if that HTML contains characters that are not in the character set supported by the server on which the servlet is running, the characters may not be readable on the client's browser. For example, if the server's encoding is set to ISO-8859-1, but the HTML written out by the servlet contains double-byte characters, those characters will not appear correctly on the client's browser, even if the browser is set correctly to view them.

By specifying an encoding in the servlet, the servlet can be deployed on any server without having to know that server's encoding setting. Servlets can also respond to user input and write out HTML for a selected language.

The following is an example of how to specify the encoding setting in a servlet. In the Java source generated by the Servlet wizard, the `doPost()` method contains the following line:

```
PrintWriter out = response.getWriter();
```

This line can be replaced with:

```
OutputStreamWriter writer = new OutputStreamWriter(
  response.getOutputStream(), "encoding");
PrintWriter out = new PrintWriter(writer);
```

The second argument to the `OutputStreamWriter` constructor is a `String` representing the desired encoding. This string can be resourced, hard-coded, or set by a variable. A call to `System.getProperty("file.encoding")` returns a `String` representing the system's current encoding setting.

If the `OutputStreamWriter` constructor is called with only the first argument, the current encoding setting is used.

Writing a data-aware servlet

JBuilder web development technologies include the InternetBeans Express API to simplify the creation of data-aware servlets. InternetBeans Express is a set of components that read HTML forms and generate HTML from DataExpress data models, making it easier to create data-aware servlets and JSPs. The components generate HTML and are used with servlets and JSPs to create dynamic content. They feature specific hooks and optimizations when used in conjunction with DataExpress components but can be used with generic Swing data models as well.

For more information on InternetBeans Express, see Chapter 7, "Using InternetBeans Express." You can also refer to Chapter 19, "Tutorial: Creating a servlet with InternetBeans Express."

Chapter

6

Developing JavaServer Pages

Web Development is a feature of JBuilder Enterprise

JavaServer Pages (JSP) technology allows web developers and designers to rapidly develop and easily maintain information-rich, dynamic web pages that leverage existing business systems. As part of the Java family, the JSP technology enables rapid development of web-based applications that are platform independent.

In theory, JavaServer Pages technology separates the user interface from content generation, enabling designers to change the overall page layout without altering the underlying dynamic content. In practice, it takes a little planning and some coding standards to ensure that the HTML is cleanly separated from the Java code in the JSP, since they both reside in the same file. Web designers handling the HTML portion should have a minimal understanding of which tags denote embedded Java code to avoid causing problems when designing the UI.

JSP technology uses XML-like tags and scriptlets written in the Java programming language to encapsulate the logic that generates the content for the page. Additionally, the application logic can reside in server-based resources (such as JavaBeans component architecture) that the page accesses with these tags and scriptlets. Any and all formatting (HTML or XML) tags are passed directly back to the response page. By separating the page logic from its design and display and supporting a reusable component-based design, JSP technology makes it faster and easier than ever to build web-based applications.

JSP technology is an extension of the Java Servlet API. JSP technology essentially provides a simplified way of writing servlets. Servlets are platform-independent, 100% pure Java server-side modules that fit seamlessly into a web server framework and can be used to extend the capabilities of a web server with minimal overhead, maintenance, and support. Unlike other scripting languages, servlets involve no platform-specific consideration or modifications. Together, JSP

technology and servlets provide an attractive alternative to other types of dynamic web scripting/programming. JSP technology and servlets offer platform independence, enhanced performance, separation of logic from display, ease of administration, extensibility into the enterprise and most importantly, ease of use.

JSPs are very similar to ASPs (Active Server Pages) on the Microsoft platform. The main difference between JSPs and ASPs is that the objects being manipulated by the JSP are JavaBeans, which are platform independent. Objects being manipulated by the ASP are COM objects, which ties ASPs completely to the Microsoft platform.

All that is required for a JSP is a JSP technology-based page. A JSP technology-based page is a page that includes JSP technology-specific tags, declarations, and possibly scriptlets, in combination with static content (HTML or XML). A JSP technology-based page has the extension `.jsp`; this signals to the web server that the JSP technology-enabled engine will process elements on this page. A JSP can also optionally use one or more JavaBeans in separate `.java` files.

When a JSP is compiled by the JSP engine on the web server, it gets compiled into a servlet. As a developer, you usually won't see the code in the generated servlet. This means that when compiling JSPs in the JBuilder IDE, you may see error messages that refer directly to code in the generated servlet, and only indirectly to the JSP code. Keep in mind that if you get error messages when compiling your JSP, they could refer to lines of code in the generated servlet. It's easier to determine the problem in your JSP if you have an understanding of how JSPs get translated into servlets. To achieve this, you need to understand the JSP API.

When JBuilder compiles the JSP, it translates the locations in the generated servlet back into the locations in the originating JSP. This means that clicking on an error in the structure pane or message view takes you to the right place in the JSP. When the web server compiles the JSP, however, it does not do this translation. When you see a stack trace in the browser or the Web View or a stack trace in the web server's output, the file name and the line numbers are not translated.

Chapter 18, "Tutorial: Creating a JSP using the JSP wizard," shows you how to create a JSP using the JSP wizard as a starting point.

For links to web pages that contain more information on JavaServer Pages technology, see "Additional JSP resources" on page 6-13.

JSP tags

A JSP usually includes a number of specialized tags which contain Java code or Java code fragments. Here is a list of a few of the most important JSP tags:

Table 6.1 Common JSP tags

tag syntax	description
`<%` *code fragment* `%>`	Scriptlet tag. Contains a code fragment, which is one or more lines of code that would normally appear within the body of a method in a Java application. No method needs to be declared, because these code fragments become part of the `service()` method of the servlet when the JSP is compiled.
`<%!` *declaration* `%>`	Method or variable declaration. When declaring a method in this tag, the complete method must be contained in the tag. Gets compiled into a method or variable declaration in the servlet.
`<%--` *comment* `--%>`	Comment. This is a JSP style comment that doesn't get passed to the client browser. (You could also use HTML comments, but these do get passed to the client browser.)
`<%=` *expression* `%>`	Expression. Contains any valid Java expression. The result is displayed at that point on the page.
`<%@ page` *[attributes]* `%>`	Page directive. Specifies attributes of the JSP page. Directives like this and the taglib directive should be the first lines in the JSP. One of the most common attributes to specify in the page directive is an import statement. Example: `<%@ page import="com.borland.internetbeans.*" %>`
`<%@ taglib uri="`*path to tag library*`" prefix="`*tag prefix*`" %>`	Taglib directive. Makes a tag library available for use in the JSP by specifying the location of the tag library and the prefix to use in its associated tags. Directives like this and the page directive should be the first lines in the JSP.

The JSP specification also includes standard tags for bean use and manipulation. The `useBean` tag creates an instance of a specific JavaBeans class. If the instance already exists, it is retrieved. Otherwise, it is created. The `setProperty` and `getProperty` tags let you manipulate properties of the given bean. These tags and others are described in more detail in the JSP specification and user guide, which can be found at `http://java.sun.com/products/jsp/techinfo.html`.

It's important to realize that most Java code contained within JSP tags becomes part of the servlet's `service()` method when the JSP is compiled into a servlet. This doesn't include code contained in declaration tags, which become complete method or variable declarations in their own right. The `service()` method is called whenever the client does a `GET` or a `POST`.

JSP tag libraries and frameworks

JSP tag libraries and frameworks make JSP development easier by providing additional JSP tags and functionality that are not available in the JSP API.

A tag library is a collection of Java code that can be called from markup in the JSP file. A framework defines a discipline for application development. You accept a framework's development paradigm and in return you get some development help and a cleaner, more standardized application.

A JSP tag library consists of a tag library descriptor (TLD) file and tag handler classes which implement or call the functionality that's needed by the tags. A JSP tag library can also be part of a framework. A framework can contain one or more tag libraries. There can also be other types of frameworks, some of which are not designed for use with JSPs.

There are three popular frameworks that are shipped with and integrated into JBuilder and are useful for JSP development. They are described as follows.

- Struts — The Struts open source framework provides a Model 2, or Model-View Controller, approach to software design.
- JSTL (JavaServer Pages Standard Tag Library) — JSTL provides a standard way to accomplish common coding tasks using simple tags. More information is available from `http://java.sun.com/products/jsp/jstl/`.
- InternetBeans Express — InternetBeans Express is a Borland component library that makes it easier to create data-aware servlets and JSPs.

See also

- "Working with JSP tag libraries and frameworks in JBuilder" on page 6-5
- Chapter 8, "Using the Struts framework in JBuilder"
- Chapter 7, "Using InternetBeans Express"

JSPs in JBuilder

JBuilder provides a complete development system for JSPs. Here's a list of JBuilder's JSP development features.

- A JSP wizard for creating a new JSP that can optionally support one or more frameworks
- JSP tag library and framework support
- CodeInsight for completing JSP-specific tags
- Debugging within the JSP file
- Testing and running the JSP on any JSP-supporting servlet engine from within the JBuilder development environment

Note You can't open a JSP in JBuilder's designer because it isn't a `.java` file.

Working with JSP tag libraries and frameworks in JBuilder

JBuilder supports the use of JSP tag libraries and frameworks in the following ways.

- The Web Application wizard allows you to select one or more frameworks for use in your WebApp. The wizard adds the libraries for the selected framework to your WebApp, adds the required tag libraries, adds required entries to the `web.xml` file, and copies the framework's JARs to `WEB-INF/lib`. It may also add child nodes to the project pane and add configuration files if needed. All of this helps ensure that the framework will be deployed correctly and can be used when running your WebApp in the JBuilder IDE.
- The JSP wizard allows you to select one or more tag libraries to use in the JSP that the wizard generates. The wizard adds the appropriate `page` and `taglib` directives to the generated JSP.
- The Configure Libraries dialog box allows you to manage user-defined tag libraries and frameworks.

See also

- "Creating a WebApp with the Web Application wizard" on page 3-3
- "Creating a JSP using the JSP wizard" on page 6-10
- "Using the Configure Libraries dialog box to manage user-defined frameworks" on page 6-6

Using the Configure Libraries dialog box to manage user-defined frameworks

The Framework tab of the Configure Libraries dialog box (Tools | Configure Libraries) allows you to configure user-defined JSP tag libraries for use with JSPs in JBuilder. A framework consists of a tag library and taglib-specific elements that are contained in a JAR file.

The following frameworks are provided with JBuilder, and do not need to be configured:

- Struts
- JSTL
- InternetBeans Express
- Cocoon

These frameworks are automatically available from the appropriate wizards. For example, when you choose the JSP wizard, the InternetBeans, JSTL, and Struts frameworks are available from the Tag Libraries list on the Enter JSP File Details page of the wizard.

When you have properly configured your user-defined JSP tag library using the Configure Libraries dialog box, your tag library becomes available in the WebApp and JSP wizards.

Adding a user-defined JSP tag library

When you add a new user-defined JSP tag library, the tag library is automatically available in the WebApp and JSP wizards. To add a new tag library,

1. Choose Tools | Configure Libraries to open the Configure Libraries dialog box.
2. Click the New button on the lower left to display the New Library wizard.

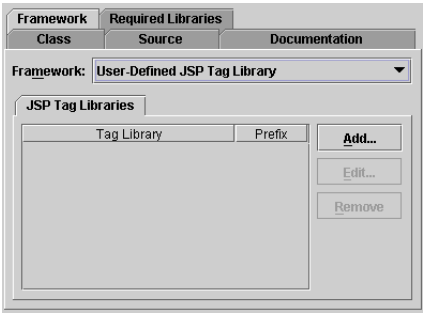
3 Enter the tab library's name in the Name field. Choose its location from the Location drop-down list.

4 Click Add to add the JAR file(s) containing your tag library. Browse to each JAR file, select it, and click OK.

5 Click OK to close the New Library wizard. The Configure Libraries dialog box is redisplayed. The new library is displayed in the assigned folder on the left of the dialog box.

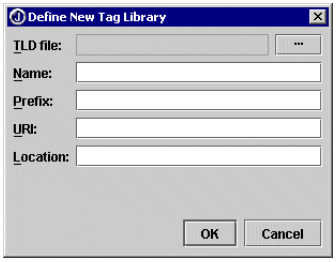
6 Click the Framework tab on the Configure Libraries dialog box.

7 Choose User-Defined JSP Tag Library from the Framework drop-down list.



The JSP Tag Libraries tab is displayed.

8 Click the Add button to display the Define New Tag Library dialog box.



9 Click the ellipsis (...) button next to the TLD File field to browse to the TLD file you want to define for your tag library. The TLD can be an external file distributed with the JAR, inside the JAR, or both. The TLD file must have a .TLD extension. It requires that actual tag handler classes be in a JAR file. Once you choose a valid TLD file, the Name, Prefix, URI and Location are automatically filled in for you from the TLD definition.

10 Click OK to close the Define New Tag library dialog box.

11 Click OK again to close the Configure Libraries dialog box.

Note You can add as many TLDs to a single framework as you like.

Editing the properties of a tag library in an existing framework

Once the framework has been defined, the most common editing task is the change the prefix for a tag library. Both the prefix and URI are used in the `taglib` tag generated by the JSP wizard. The prefix can be whatever you want as long as it doesn't contain any spaces or special characters. You should avoid using the same prefix for two different tag libraries. If you just want to edit the prefix, you can change it in the table without having to open the Edit Tag Library dialog box. Don't forget to press `Enter` to save your edit.

To edit the properties of a tag library in an existing framework,

1. Choose Tools | Configure Libraries to open the Configure Libraries dialog box.
2. On the left side of the dialog box, choose the name of the framework library that contains a tag library whose properties you want to edit.
3. Click the Framework tab.
4. From the JSP Tag Libraries tab, choose the tag library definition you want to edit.

Note If the JSP Tag Libraries tab is not displayed, it means the selected framework either does not contain JSP Tag Libraries or else they are not editable.

5. Click the Edit button. The Edit Tag Library dialog box is displayed.
6. Change the TLD File, Name, Prefix, URI, or Location and click OK.
7. Click OK again to save your changes and close the Configure Libraries dialog box.

Note You can add as many TLDs to a single framework as you like.

Removing a tag library from a framework

One reason to remove a tag library from a framework is if you never use that particular library. To remove a tag library from an existing framework,

1. Choose Tools | Configure Libraries to open the Configure Libraries dialog box.
2. On the left side of the dialog box, choose the framework library that contains a tag library you want to remove.
3. Click the Framework tab.
4. From the JSP Tag Libraries tab, choose the tag library definition you want to remove.

Note If the JSP Tag Libraries tab is not displayed, it means the selected framework either does not contain JSP Tag Libraries or else they are not editable.

5 Click Remove.

6 If you want to remove the entire library, choose the library in the left side of the dialog box and choose the Delete on the lower left.

7 Click OK to close the Configure Libraries dialog box.

Developing a JSP

The JSP wizard is an optional starting point for developing a JSP. JBuilder's editor provides syntax highlighting for JSPs. JBuilder also provides CodeInsight and ErrorInsight for Java code embedded in a JSP page.

The structure pane in the JBuilder IDE shows the tag structure within the JSP, and it also shows any HTML and Java errors in your code. These errors are very useful, for instance, they can often remind you to finish a tag that was incomplete.

The JSP wizard

JBuilder provides a JSP wizard. This is a logical starting point when developing a JSP. To find this wizard, click the Web tab of the object gallery. You must have a server selected for your project for the JSP wizard to be available.

The JSP wizard generates the skeleton of a JSP. It creates the basic files for your JSP, then you fill in the details later.

The JSP wizard creates the following files:

- A JSP. A JSP includes JSP-specific tags, declarations, and possibly scriptlets, in combination with other static (HTML or XML) tags. The JSP file has the extension `.jsp`. This file is always created by the JSP wizard.
- A Java file. This file can optionally be created by the wizard, by checking "Generate sample bean". The Java file contains one or more JavaBeans that are used by the JSP file.
- An error page JSP. The JavaServer Page wizard can optionally create an error page that displays runtime errors in a tidy and useful way.

In the JSP wizard, you can specify which WebApp your JSP is a part of, whether to set the JSP up to use one or more of several available tag libraries, and whether your JSP uses any JavaBeans. You can also create a new runtime configuration for your JSP.

See also

- JSP wizard in the online help
- Chapter 18, "Tutorial: Creating a JSP using the JSP wizard"

Creating a JSP using the JSP wizard

The JSP wizard is useful as an optional starting point for creating a JSP. Before using the JSP wizard, you should use the Web Application wizard to create a WebApp if there isn't already a WebApp in your project. To create a JSP using the JSP wizard:

1. Select File | New to open the object gallery. Click the Web tab. (Your project must first have a server enabled.)
2. Select JavaServer Page. Click OK.
3. Select a WebApp to contain the new JSP. If you have only one WebApp in your project, the WebApp drop-down list is disabled.
4. Specify whether to create a sample bean and an error page.
   - If Generate Sample Bean is checked, the wizard creates a sample JavaBean that can be used by the JSP. Checking this option adds a step to the wizard.
   - If Generate Error Page is checked, the wizard creates an error page that displays runtime errors in a tidy and useful way. Checking this option adds a step to the wizard.

   Click Next to go to the next step. You may also click Finish to skip the remaining steps, close the wizard, and create the JSP.
5. Select the background color of the JSP.
6. Check Generate Submit Form if you want the JSP wizard to generate an HTML `<form>` tag in the JSP file.
7. Select the tag libraries you want the JSP to use by checking the appropriate boxes in the Tag Libraries list. The wizard adds the appropriate `page` and `taglib` directives for the selected tag libraries to the generated JSP. Tag libraries for the InternetBeans Express, Struts, and JSTL frameworks are automatically available. Any user-defined tag libraries that you've added in the Configure Libraries dialog box are also available here.

   When you're finished selecting tag libraries, click Next to go to the next step. You may also click Finish to skip the remaining steps, close the wizard, and create the JSP.
8. Specify the Package and Class Name for the sample bean. Check Generate Header Comments if you want header comments to be generated in the sample bean. If you didn't check Generate Sample Bean in the first step of the wizard, you won't see this step.
   Click Next to go to the next step. You may also click Finish to skip the remaining steps, close the wizard, and create the JSP.

9 Use the Add Bean and Remove Bean buttons to specify JavaBeans you'd like the JSP to use. The wizard generates a `jsp:useBean` tag for each JavaBean specified.

   When you're finished specifying JavaBeans, click Next to go to the next step. You may also click Finish to skip the remaining steps, close the wizard, and create the JSP. If you didn't check Generate Sample Bean or Generate Error Page in the first step of the wizard, there are no other steps and the Next button is disabled.

10 Specify a Name and the background color for the error page. If you didn't check Generate Error Page in the first step of the wizard, you won't see this step.

   Click Next to go to the next step. You may also click Finish to skip the remaining step, close the wizard, and create the JSP.

11 Check Create A Runtime Configuration if you want to create a new Server type runtime configuration. If this option is checked, specify a Name. You may also choose a base configuration to copy if other Server type runtime configurations exist in your project.

   Click Finish to close the wizard and create the JSP.

See also

- JSP wizard in the online help
- Chapter 18, "Tutorial: Creating a JSP using the JSP wizard"
- "Using the Configure Libraries dialog box to manage user-defined frameworks" on page 6-6
- "Working with JSP tag libraries and frameworks in JBuilder" on page 6-5
- Chapter 8, "Using the Struts framework in JBuilder"
- Chapter 7, "Using InternetBeans Express"

Compiling a JSP

JSPs are an extension of the Servlet API and are compiled to servlets before they are used. This requires the compilation process to translate JSP file names and line numbers into their Java equivalents. In JBuilder, JSPs can be compiled at build-time. To enable this feature for all JSPs in your project,

1 Choose Project | Project Properties.

2 Select the Build page.

3 Select the General page on the Build page.

4 Select Check JSPs For Errors At Build-Time.

5 Click OK.

You can set this property for each JSP file in your project via the context menu for the JSP in the project pane. This allows you to exclude certain JSP files from compilation. For example, JSPs that are intended to be included in other JSPs probably would not compile successfully on their own, so you would exclude those files.

To compile your project, including any JSPs that are set to compile at build-time, select Project | Make Project or Project | Rebuild Project.

See "Compiling your servlet or JSP" on page 10-13 for more information on compiling your JSP.

Web Running a JSP

To run a JSP in the JBuilder IDE, right click the JSP in the project pane and select Web Run from the context menu. Web Run is enabled if an appropriate runtime configuration exists and the currently selected server supports the JSP/Servlet service.

See "Web running your servlet or JSP" on page 10-14 for more information on running your JSP.

Web Debugging a JSP

To debug a JSP in the JBuilder IDE, right click the JSP in the project pane and select Web Debug from the context menu. Web Debug is enabled if an appropriate runtime configuration exists and the currently selected server supports the JSP/Servlet service.

See "Web debugging your servlet or JSP" on page 10-17 for more information on debugging your JSP.

Deploying a JSP

See Chapter 11, "Deploying your web application," for tips on deploying your JSP.

Additional JSP resources

For assistance with developing JSPs in JBuilder, visit the Borland newsgroup borland.public.jbuilder.servlets-jsp. All JBuilder newsgroups can be accessed from the Borland web site at http://www.borland.com/newsgroups/#jbuilder.

For more information on developing JavaServer Pages, point your browser to the following web sites. These web addresses and links were valid as of this printing. Borland doesn't maintain these web sites and isn't responsible for their content or longevity.

- Get a JSP syntax card in HTML format, from http://java.sun.com/products/jsp/tags/tags.html.
- Get a JSP syntax card in PDF format, viewable with Adobe Acrobat Reader, from http://java.sun.com/products/jsp/technical.html. This page also contains other technical resources for JSP technology.
- Get all the FAQs on JavaServer Pages at java.sun.com. Point your browser to http://java.sun.com/products/jsp/faq.html.
- The current JSP specification can be viewed from the java.sun.com web site, at http://java.sun.com/products/jsp/download.html.
- GNUJSP is a free implementation of Sun's JavaServer Pages. To learn more about the GNUJSP compiler, visit http://klomp.org/gnujsp.
- A JSP list server is maintained at JSP-INTEREST@JAVA.SUN.COM. To subscribe to the mailing list, send a message to listserv@java.sun.com with the following message body:

  ```
  subscribe jsp-interest Your Full Name
  ```

  or visit the java.sun.com web site for information.

Chapter

7

Using InternetBeans Express

Web Development is a feature of JBuilder Enterprise

InternetBeans Express technology integrates with servlet and JSP technology to add value to your application and simplify servlet and JSP development tasks. InternetBeans Express is a set of components and a JSP tag library for generating and responding to the presentation layer of a web application. It takes static template pages, inserts dynamic content from a live data model, and presents them to the client; then it writes any changes that are posted from the client back into the data model. This makes it easier to create data-aware servlets and JSPs. For example, you can use InternetBeans Express components to create a servlet that provides a form for a new user to register for site access or a JSP that displays the results of a search in a table.

InternetBeans Express contains built-in support for DataExpress DataSets and DataModules. It can also be used with generic data models and EJBs. The classes and interfaces fall into three categories:

- InternetBeans are components that deal directly with the dynamic generation of markup and the handling of HTTP request/response semantics.
- JSP tag handlers, which invoke InternetBeans internally, and their supporting infrastructure. These are used by the JSP tag extensions in the InternetBeans tag library.
- Data model support.

There is also a JSP tag library which contains JSP tag extensions used to support InternetBeans in a JSP.

Overview of InternetBeans Express classes

InternetBeans Express includes the following classes:

Table 7.1 InternetBeans Express classes

Component type	Description
 IxPageProducer	Reads and parses static HTML files so that it can later regenerate the file, inserting dynamic content from other components. Most servlets use an `IxPageProducer`, enabling the servlet to generate the entire response from a pre-designed web page, inserting dynamic data as spans of text or in controls in a form on that page.
 IxControl	A generic control that determines at runtime which type of HTML control it is replacing and emulates that control. This component can be used in place of any of the other control-specific InternetBeans. In a servlet, this component must be used with an `IxPageProducer`. You do not need an `IxPageProducer` to use this component in a JSP.
 IxTable	Generates HTML tables from data sets or table models.
 IxImage	Represents an image that is simply displayed or used as a link. If you want to use an image to submit a form, use an `IxImageButton`.
 IxLink	Represents a link. If URL rewriting is necessary, `IxLink` handles it for you by internally calling the `HttpServletResponse.encodeURL()` method.
 IxSpan	Replaces read-only content by ID attribute, probably a `<span>`, but could be another type of element. For controls in a form, use `IxControl` instead.
 IxCheckBox	Represents a check box. XHTML: `<input type="checkbox" />`
 IxComboBox	Represents a combo box. XHTML: `<select size="1" />`
 IxHidden	Represents a hidden value. XHTML: `<input type="hidden" />`
 IxImageButton	Represents an image that submits the form when clicked. Not to be confused with an image that is simply displayed or used as a link; for that, use `IxImage`. XHTML: `<input type="image" />` If the image that matches this component was the image that submitted the form, the `IxImageButton's submitPerformed()` event fires.
 IxListBox	Represents a list box. XHTML: `<select size="3" />`

Table 7.1 InternetBeans Express classes (continued)

Component type		Description
	`IxPassword`	Represents a password field. XHTML: `<input type="password" />`
	`IxPushButton`	Represents a client-side button. XHTML: `<input type="button" />`
	`IxRadioButton`	Represents a radio button. XHTML: `<input type="radio" />`
	`IxSubmitButton`	Represents a form submit button. XHTML: `<input type="submit" />` If the button that matches this component was the button that submitted the form, the `IxSubmitButton`'s `submitPerformed()` event fires.
	`IxTextArea`	Represents a text area. XHTML: `<textarea>`
	`IxTextField`	Represents an input field. XHTML: `<input type="text" />`

Using InternetBeans Express with servlets

The InternetBeans Express components simplify both the display of live data in a web page and posting of data from a web page into a database or other data model.

Displaying live web pages with servlets using InternetBeans Express

Most servlets should use an `IxPageProducer` component. This enables the servlet to generate the entire response from a pre-designed web page, inserting dynamic data as spans of text or in controls in a form on that page. This has some advantages:

- You know what the response will look like. The page can contain dummy data, which will be replaced.
- You can change that look by changing the page, without having to touch the code.

For example, when using InternetBeans Express with a servlet, you can open the servlet in the designer. A `Database` and `QueryDataSet` from the DataExpress tab of the palette can provide the data for the servlet. You can add an `IxPageProducer` from the InternetBeans tab of the palette. Set the `IxPageProducer's htmlFile` property to the file name of the pre-designed web page. When the servlet is run, the internal `HtmlParser` is invoked by the `IxPageProducer` to locate all replaceable HTML controls.

The simplest way to replace HTML controls with controls containing dynamically generated data is to use `IxControls`. You should add one `IxControl` for each HTML control on the template page which will contain data. Set each `IxControl's pageProducer` property to the `IxPageProducer`. Set the `IxControl's controlName` property to match the `name` attribute of the appropriate HTML control. Setting the `dataSet` and `columnName` properties of the `IxControl` completes the data linkage.

The `IxPageProducer.servletGet()` method is the one you will normally call to generate the page for display. This method should be called within the servlet's `doGet()` method. The body of a servlet's `doGet()` method can often be as simple as:

```
ixPageProducer1.servletGet(this, request, response);
```

This single call sets the content type of the response and renders the dynamic page into the output stream writer from the response. Note that the default content type of the response is HTML.

Internally, the `IxPageProducer.render()` method generates the dynamic version of the page, replacing the controls that have an `IxControl` assigned to them by asking the component to render, which generates equivalent HTML with the data value filled in from the current row in the dataset. You could call the `render()` method yourself, but it is simpler to call the `servletGet()` method.

Some situations where you wouldn't use an `IxPageProducer` in a servlet include:

- When you don't need the database session management and posting features of the `IxPageProducer` and simply want the page template engine, you can use the `PageProducer` instead.
- When you're using specific individual components to render HTML. For example, you can create an `IxComboBox` containing a list of countries, and use it in a servlet with hand-coded HTML.

Remember that when using InternetBeans in a servlet, usually you should use an `IxPageProducer`. When you are using `IxControls`, you must use an `IxPageProducer`.

Posting data with servlets using InternetBeans Express

Processing an HTTP POST is simple with the `IxPageProducer.servletPost()` method:

```
ixPageProducer1.servletPost(this, request, response);
```

This method should be called within the servlet's `doPost()` method, along with any other code that should be executed during the post operation.

At design-time, you should add an `IxSubmitButton` for each Submit button on the form. Add a `submitPerformed()` listener for each of the `IxSubmitButtons`. The listener should call code that is to be executed when the button is pressed. For example, a Next button should do `dataSet.next()`, and a Previous button should do `dataSet.prior()`.

At runtime, when the `servletPost()` method is called it writes the new values from the post into the corresponding InternetBeans components and transmits those values from the client side to the server side. It then fires the appropriate `submitPerformed()` event for the button that submitted the form. To actually post and save changes to the underlying dataset, you should call the dataset's `post()` and `saveChanges()` methods within the `submitPerformed()` method. The servlet's `doPost()` method can then call `doGet()` or call `IxPageProducer.servletGet()` directly to render the new page.

Parsing pages

Unlike XML, which is strict, the HTML parser is lax. In particular, HTML elements (tag) and attribute names are not case-sensitive. However, XHTML is case-sensitive; the standard names are lowercase by definition.

To make things faster, HTML element and attribute names are converted to the XHTML-standard lowercase for storage. When searching for a particular attribute, use lowercase.

When InternetBeans Express components are matched with HTML controls in the page, properties set in the InternetBeans Express component take precedence. When setting properties in the designer, you should think about whether you actually want to override a particular HTML attribute by setting its corresponding property in the component. For example, if the web page contains a `textarea` of a certain size, you probably don't want to override that size when that control is dynamically generated.

Generating tables

A fairly common and complex task is the display of data in a table using a particular format. For example, there may be certain cell groupings and alternating colors for each row.

The web page designer need only provide enough dummy rows to present the look of the table (for alternating colors, that's two rows). When the replacement table is generated by the `IxTable` component, that look will be repeated automatically.

You can set cell renderers by class or assign each column its own `IxTableCellRenderer` to allow customization of the content; for example, negative values can be made red (preferably by setting an appropriate cascading style sheets (CSS) style, not by hard-coding the color red).

For a tutorial on using InternetBeans in a servlet, see Chapter 19, "Tutorial: Creating a servlet with InternetBeans Express."

Using InternetBeans Express with JSPs

The key to using InternetBeans Express with JSPs is in the InternetBeans Express tag library, defined in the file `internetbeans.tld`. This tag library contains a set of InternetBeans tags that can be used in your JSP file whenever you want to use an InternetBeans component. These tags require very little coding, but when the JSP is processed into a servlet, they result in full-fledged InternetBeans components being inserted into the code.

To use InternetBeans Express in a JSP, you must always have one important line of code at the beginning of your JSP. It is a `taglib` directive, which indicates that the tags in the InternetBeans Express tag library will be used in the JSP and specifies a prefix for these tags. The `taglib` directive for using the InternetBeans tag library looks like this:

```
<%@ taglib uri="/internetbeans.tld" prefix="ix" %>
```

If you want to instantiate classes in your scriptlets, and don't want to type the fully-qualified class name, you can import files or packages into your JSP using a `page` directive. This `page` directive can specify that the `com.borland.internetbeans` package should be imported into the JSP. The `page` directive should look something like this:

```
<%@ page import="com.borland.internetbeans.*,com.borland.dx.dataset.*,
com.borland.dx.sql.dataset.*" %>
```

Remember that directives such as the `taglib` directive and the `page` directive must always be the very first lines in your JSP.

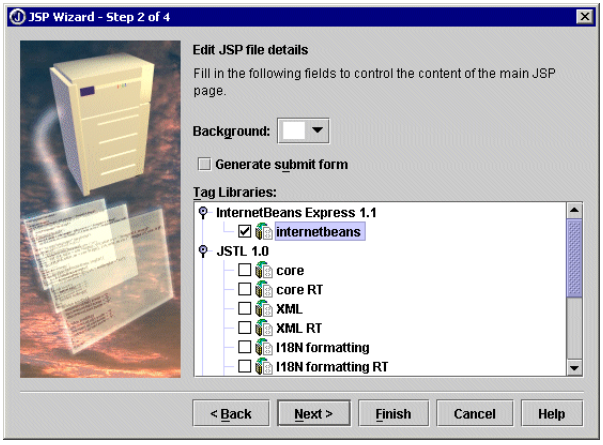
JBuilder's JSP wizard inserts a `taglib` directive and a `page` directive for you if you select the internetbeans tag library in the Edit JSP File Details step of

the wizard. The JSP wizard also completes the other necessary steps for setting up your JSP to use InternetBeans Express. These steps are as follows:

1. It adds the InternetBeans Express library to your project.
2. It sets the dependencies for the InternetBeans Express, dbSwing, and DataExpress libraries to Include All for your WebApp. This means the required `jar` files are copied to the WebApp's `WEB-INF/lib` directory when the project is compiled.
3. It adds a tag library mapping between `internetbeans.tld` and `WEB-INF/lib/internetbeans.jar` to the web.xml file.

You need to do these steps yourself if you are setting up your JSP to use InternetBeans Express without using the JSP wizard.

Here's the JSP wizard with the internetbeans tag library selected:



Here is an example of how an InternetBeans tag looks when used in your JSP:

```
<ix:database id="database1"
driver="com.borland.datastore.jdbc.DataStoreDriver"
url="jdbc:borland:dslocal:..\\guestbook\\guestbook.jds"
username="user">
</ix:database>
```

This example uses the `database` tag. If you were actually using the `database` tag in a JSP, in most cases you will want to nest other tags within this tag, such as the `query` tag. This isn't required, but it makes the JSP code more readable.

Note The `ix` prefix could be any text. It all depends on what prefix is specified in the `taglib` directive. The JSP wizard uses the `ix` prefix for the internetbeans tag library.

For a tutorial on this topic, see Chapter 20, "Tutorial: Creating a JSP with InternetBeans Express."

Table of InternetBeans tags

The tags which are included in the InternetBeans Express tag library are described in the table below. The attributes shown in bold type are required.

Table 7.2 InternetBeans Express tags

Tag name	Description	Attributes
`database`	Defines a DataExpress `Database`	• **`id`** — text used to identify this database • **`driver`** — driver property of `Database` • **`url`** — url property of `Database` • `username` — the username for the database • `password` — the password for the database
`query`	Defines a DataExpress `QueryDataSet`	• **`id`** — text used to identify this query • `database` — identifies the database to which this query belongs. This isn't required because it's implied if the query tag is nested within the database tag. If the query tag isn't nested within the database tag, this attribute needs to be specified. • **`statement`** — the SQL statement executed by this query.
`control`	Defines an InternetBeans Express `IxControl`	• `id` — text used to identify this control • `tupleModel` — the `tupleModel` for this control • `dataSet` — identifies the dataset (query) to which this control is connected. Either the **`dataSet`** or the **`tupleModel`** is required, but you can't have both. • **`columnName`** — identifies the `columnName` to which this control is connected.
`image`	Defines an InternetBeans Express `IxImage`	• `id` — text used to identify this image • `tupleModel` — the `tupleModel` for this control • `dataSet` — identifies the dataset (query) to which this image is connected. Either the **`dataSet`** or the **`tupleModel`** is required, but you can't have both. • **`columnName`** — identifies the `columnName` to which this image is connected.
`submit`	Defines an InternetBeans Express `IxSubmitButton`	• `id` — text used to identify this submit button • **`methodName`** — name of the method which will be executed when this button is pressed.
`table`	Defines an InternetBeans Express `IxTable`	• `id` — text used to identify this table • `dataSet` — identifies the dataset (query) to which this table is connected. • `tableModel` — the data model for this table. Either the **`dataSet`** or the **`tableModel`** is required, but you can't have both.

There are only six tags in the InternetBeans Express tag library, yet there are seventeen InternetBeans components. This may seem like a major limitation, but it's really not. The `control` tag maps to an `IxControl`, which delegates to all the other control-specific InternetBeans. The only InternetBeans which aren't covered by the tag library are `IxSpan` and `IxLink`. Neither of these are useful in a JSP, because you can just as easily use your own JSP expression scriptlet to do the same thing.

Of course, it's also possible to use InternetBeans directly, just like any other bean or Java class. Using the tag library is just much more convenient and it does a few extra things for you (like maintaining the session state for data entry).

Format of internetbeans.tld

It is useful to know that you can always look at the source of the `internetbeans.tld` file for hints about use of the various tags. To do this, open it in JBuilder's editor. This file cannot (and should not) be modified.

The `internetbeans.tld` file is available in `internetbeans.jar`. You don't need to be able to view the contents of `internetbeans.tld` in order to use its tags in your JSP, but if you want to view the `internetbeans.tld` file in the editor, you need to do the extra step of adding it to your project. To do this:

1. Click the Add Files/Packages button on the toolbar above the project pane.
2. In your project directory, find `internetbeans.jar`. It will be in the `WEB-INF/lib` directory of your WebApp.
3. In the directory tree, click to expand the `internetbeans.jar` node.
4. Under `com.borland.internetbeans.taglib`, locate the `internetbeans.tld` file and select it.
5. Click OK to add the file to your project.

The information at the very top of the `internetbeans.tld` file is of little interest. The information that is useful to understand begins with the first `<tag>` tag inside the file. Each `<tag>` tag represents an InternetBeans tag definition.

At the beginning of each tag definition, you see a <name> tag which indicates the name of the tag. The first one is the database tag. Nested within each tag definition, you will also see <tagclass>, <info>, and <attribute> tags. For an example of how an InternetBeans tag definition looks, see the fragment of the internetbeans.tld file which defines the submit tag below:

```
<tag>
  <name>submit</name>
  <tagclass>com.borland.internetbeans.taglib.SubmitTag</tagclass>
  <bodycontent>JSP</bodycontent>
  <info>Submit button or submit image control</info>
  <attribute>
     <name>id</name>
     <required>false</required>
     <rtexprvalue>false</rtexprvalue>
  </attribute>
  <attribute>
     <name>methodName</name>
     <required>true</required>
     <rtexprvalue>false</rtexprvalue>
  </attribute>
</tag>
```

The <tagclass> tag indicates the name of the class within the com.borland.internetbeans.taglib package which is responsible for interpreting this InternetBeans tag when it is used in a JSP. The <info> tag supplies a description of the InternetBeans tag.

The <attribute> tag describes an attribute for an InternetBeans tag. There is one <attribute> tag for each attribute. These can be thought of as the component's properties. Nested within the <attribute> tag you will see these properties. Each property has a name, a boolean value indicating whether or not it is a required property, and a boolean value indicating whether or not its value can be set using a java expression. The name is found within the <name> tag, the <required> tag indicates whether or not the property is required, and the <rtexprvalue> tag indicates whether or not the property can be set using a java expression. Those properties which can't be set using an expression require a literal value.

Chapter

8

Using the Struts framework in JBuilder

Web Development is a feature of JBuilder Enterprise

The Struts open source framework is known as the Model 2, or Model-View Controller, approach to software design. This framework evolved from the Model 1 design, JavaServer Page technology. This technology offered great advances from pure servlets, where presentation HTML was coded with lengthy `out.println` statements in `doGet()` and `doPut()` methods. JSPs offered a way to include HTML in Java code and Java code in HTML. However, these JSPs are hard to read and hard to maintain. Both web designers and developers are required to work in the same set of source files.

The Struts framework, first developed in 2001, combines the best of servlets and JSPs. The framework consists of a 3-tiered design paradigm: the Controller, Model, and the View. Struts provides its own Controller component and integrates with other technologies to provide the Model and the View.

The Controller layer controls application flow. It receives requests from a browser and determines how to process those requests. With Struts, the controller is implemented as a servlet of the class `org.apache.struts.action.ActionServlet`. The `struts-config.xml` file configures the Controller via `<action-mapping>` elements.

A business logic layer sits between the Controller and the Model. This is implemented with an `Action` class, a thin wrapper around the actual business logic. An `Action` receives the submission of a form or the request for a page, then delegates to business logic classes. Any data required by future pages is stored. The `Action` separates the business logic from the presentation and decides who is next in line for display.

The Model represents the state of the application. The business objects updates the application state. An `ActionForm` bean represents the model

state at a session-request level. The `ActionForm` class contains getter and setters for properties and a validation method. The JSP file (or the view) reads information from the `ActionForm` using JSP tags. An `ActionForm` must exist for each input form.

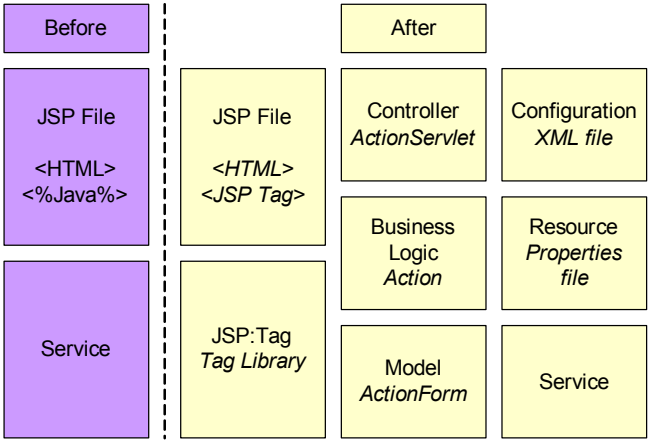
The View is a JSP file. There is no flow logic, business logic or Model information in the JSP; it is simply presentation. Struts provides a rich set of tag libraries for use in the View layer. There are four main Struts tag libraries: HTML, Bean, Logic, and Template.

- HTML tag library: Contains HTML tags that replace many of the common HTML elements.
- Bean tag library: Contains tags to display data from JavaBeans, such as data returned from a database query.
- Logic tag library: Contains tags to perform iterations and conditional processing of view data, such as presenting database data in a tabular format.
- Template tag library: Contains tags to import snippets to JSP pages, for example, a company logo and header that are common across all pages of a website.

A properties file can be used by the View layer to store strings, making internationalization simpler.

Overall, using the Struts framework significantly impacts the design of a web application. The following diagram shows the composition of a web application before using Struts and after. Notice how Struts cleanly separates the Java code from the Javascript and HTML code.

Figure 8.1 Struts — before and after



For more information on Struts, see "The Jakarta Project: Struts" at `http://jakarta.apache.org/struts/index.html`. This page provides links to the Struts Users Guide, Struts articles, sample programs, and tutorials.

Struts 1.0 and 1.1 beta releases

JBuilder 8 officially supports Struts 1.0. Although JBuilder 8 does not officially support the beta release of Struts 1.1. However, if you enable your `struts-config.xml` file for Struts 1.1 and download the Struts 1.1 JAR file to the `<jbuilder>/extras` folder, you will see support for Struts 1.1. The Struts Config Editor will display elements and attributes specific to 1.1. Additionally, if your Struts web application uses a `tileDefinitions.xml` file for presentation, a visual editor for the Tiles configuration file is available.

JBuilder tools for Struts

JBuilder provides a rich set of tools and wizards for quickly creating a Struts-enabled web application, including:

- Struts framework support
- WebApp wizard
- JSP wizard
- Action wizard
- ActionForm wizard
- JSP From ActionForm wizard
- Struts Conversion wizard
- Struts Config Editor

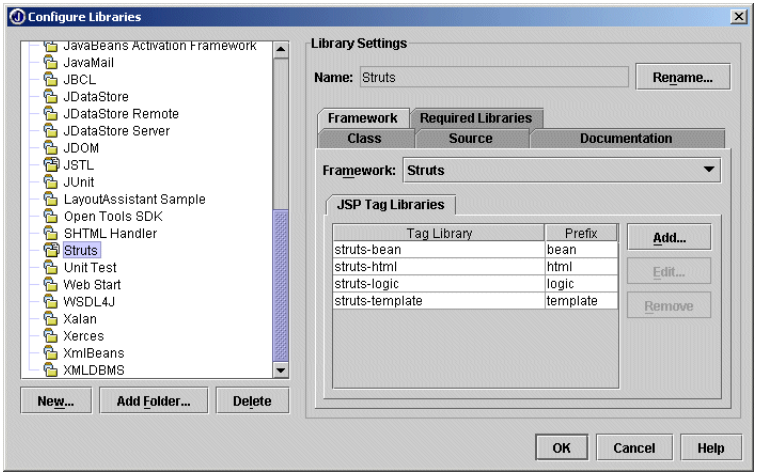
Struts framework support

The Framework tab of the Configure Libraries dialog box (Tools | Configure Libraries) allows you to configure frameworks for use with JBuilder. A framework consists of a tag library and taglib-specific elements that are contained in a JAR file. The following frameworks are provided with JBuilder, and are available from the JBuilder folder on the left side of the Configure Libraries dialog box.

- Struts
- JSTL
- InternetBeans
- Cocoon

The following figure shows the Struts framework displayed in Framework tab of the Configure Libraries dialog box.

Figure 8.2 Struts framework in Configure Libraries



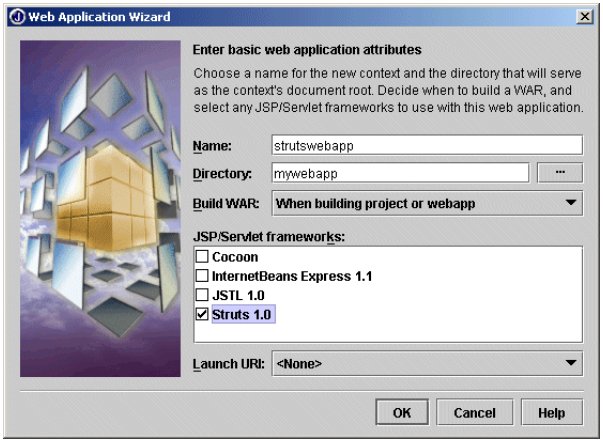
Notice how the Struts tag libraries are displayed on the JSP Tag Libraries tab of the Framework tab.

- For more information on libraries in JBuilder and the Configure Libraries dialog box, see "Working with libraries" in the "Managing paths" chapter of *Building Applications with JBuilder*.
- For more information on JSPs and tag libraries, see "Working with JSP tag libraries and frameworks in JBuilder" on page 6-5.

Struts-enabled Web Application wizard

The Web Application wizard is enabled for the Struts framework. To choose the Struts framework for your web application, choose Struts 1.0 from the JSP/Servlet Frameworks list. The Web Application wizard looks like this:

Figure 8.3 Web Application wizard



Note You can use more than one framework in a web application. For example, you can use Cocoon, Struts and the JSTL in one WebApp.

Once you choose the Struts framework for your WebApp, JBuilder adds mappings to the `web.xml` file and copies or creates files as needed. For the Struts framework, the `*.tld` files are copied into the `WEB-INF` directory and their mappings are added to `web.xml`. JBuilder creates the `struts-config.xml` file in the `WEB-INF` directory.

Some web frameworks may declare a standard welcome page that is used for the URI. In this case, the Launch URI field will be updated automatically with the name of the welcome page. For example, if you choose the Cocoon framework, the Launch URI is set to `/index.xml`.

Once your WebApp has been created, you can change the node properties, including the framework selection, by right-clicking the WebApp node in the project pane and choosing Properties. You can change the framework on the JSP/Servlet Frameworks tab of the WebApp page.

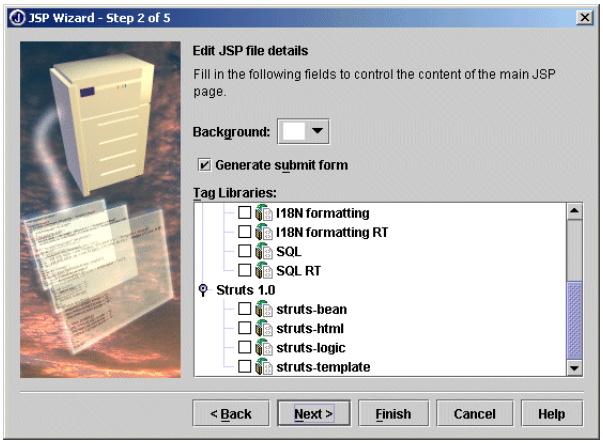
Important Creating a WebApp is the first step in building a web application. For information on the sequence of steps required to create a Struts web application, see "Creating a Struts-enabled web application in JBuilder" on page 8-16.

Struts-enabled JSP wizard

The JSP wizard is also enabled for the Struts framework. The Edit JSP File Details page of the wizard is where you choose the tag libraries to use with your JSP. In order to see the Struts tag libraries, you need to scroll the list, as shown in the following figure.

Important For information on the sequence of steps required to create a Struts web application, see "Creating a Struts-enabled web application in JBuilder" on page 8-16.

Figure 8.4 Edit JSP File Details page — JSP wizard



You select tag libraries by checking the box next to the tag library you want to use. You can choose any of the tag libraries; your choice does not have to be already enabled for your web application.

ActionForm wizard

An `ActionForm` class contains the session state of a web page. Each `ActionForm` bean is a subclass of the `ActionForm` abstract class. The `ActionForm` contains setters and getters for each of the data fields that the `ActionForm` is responsible for. This is typically all the fields in an input form on a page, but it can also span several related pages that act in a "wizard"-like way. The `ActionForm` also contains a `validate()` method that validates the data that is given to it.

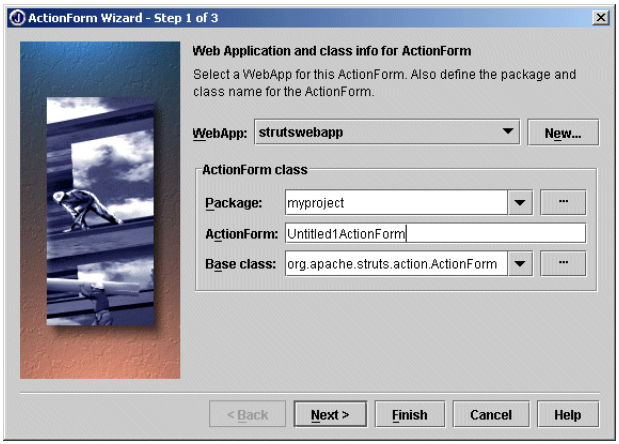
Important You create an `ActionForm` class for a JSP. To use this wizard, you must have a JSP in your project. For information on the sequence of steps required to create a Struts web application, see "Creating a Struts-enabled web application in JBuilder" on page 8-16.

The ActionForm wizard allows you to quickly and easily create an `ActionForm`. You can either add fields through the wizard or prepopulate the fields from an existing JSP page. The wizard creates the `ActionForm` class and registers it in the `web.xml` file.

Web Application And Class Info For Action Form page

The Web Application And Class Info For Action Form page of the wizard is where you specify the WebApp to use. You can choose an existing WebApp or create a new one by choosing the New button to display the Web Application wizard. You also use this page of the wizard to choose the package for the `ActionForm` class, specify the class name, and choose the base class.

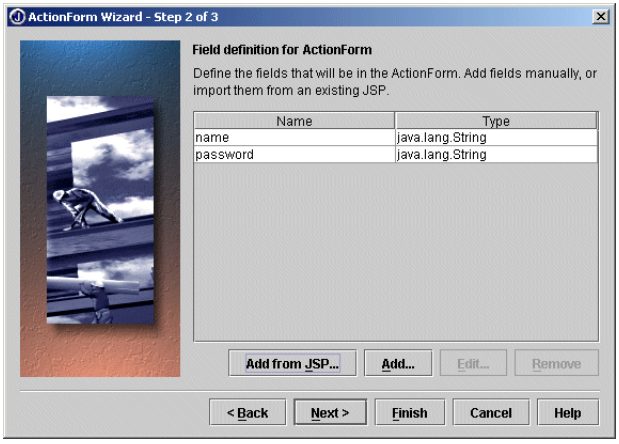
Figure 8.5 Web Application And Class Info For Action Form page — ActionForm wizard



Field Definition For ActionForm page

The Field Definition For ActionForm page allows you to define the fields the `ActionForm` will be responsible for. You can either add fields directly to the list or select a JSP from which to import all the detected fields. Fields are derived from the input tags that are discovered in the JSP.

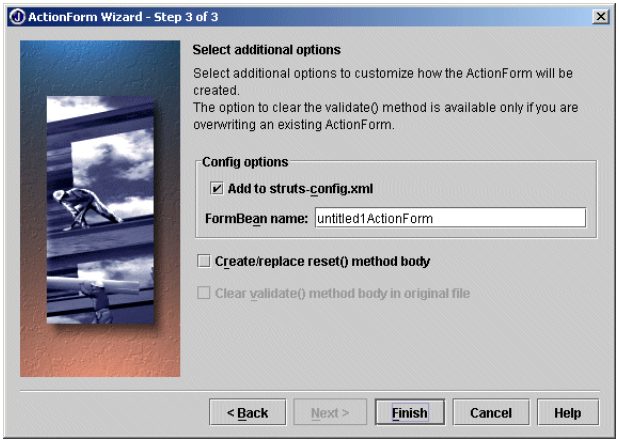
Figure 8.6 Field Definition For ActionForm page — ActionForm wizard



Select Additional Options page

The Select Additional Options page is where you choose additional configuration options. You can choose to add the `ActionForm` mapping to `struts-config.xml`. You can also choose to create and replace the `reset()` method in the `ActionForm` or clear the existing `validate()` method.

Figure 8.7 Select Additional Options page — ActionForm wizard



Note If the `ActionForm` class name you choose already exists and is a valid `ActionForm`, you will be prompted to import the existing fields from the `ActionForm`. This is useful for extending an existing form or for building a form that spans multiple JSPs. If you choose not to import existing fields, the original file will not be deleted, but all setters and getters will be removed and replaced with the fields from the final list. Any additional methods will be preserved. If fields were removed, a warning message will be displayed reminding you to fix the `validate()` method.

After you enter all the fields, the wizard will generate the `ActionForm` and register it in `struts-config.xml`.

Action wizard

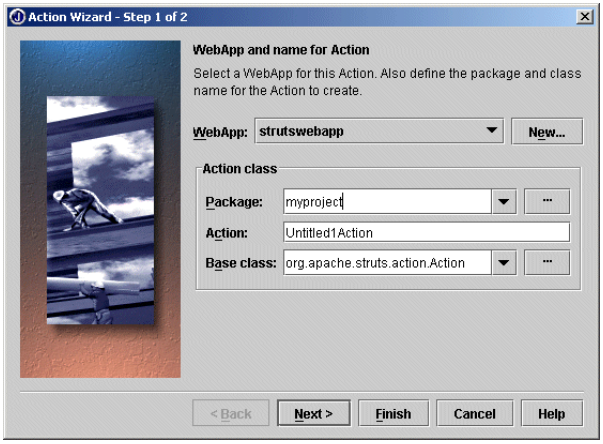
An `Action` is the control class in the Struts framework. The `Action.perform()` method is called to execute business logic. An `ActionForward` class is returned. `ActionForward` tells the Struts controller servlet the next action in the chain of events. The Action wizard quickly creates an `Action` class and registers it in `struts-config.xml`.

WebApp And Name For Action page

The WebApp And Name For Action page of the wizard allows you to specify the WebApp to use. You can choose an existing WebApp or create a new one by choosing the New button to display the Web Application wizard. You also use this page of the wizard to enter the package for the `Action` class, specify the class name, and choose the ancestor class.

Important You create an `Action` class for a `ActionForm` class. To use this wizard, you must have a JSP and `ActionForm` class in your project. For information on the sequence of steps required to create a Struts web application, see "Creating a Struts-enabled web application in JBuilder" on page 8-16.

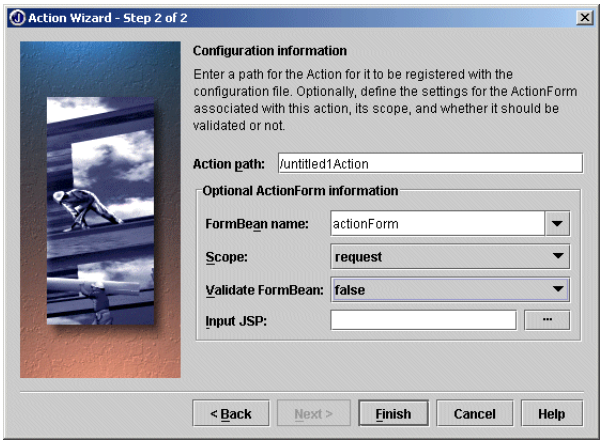
Figure 8.8 WebApp And Name For Action page — Action wizard



Configuration Information page

The Configuration Information page of the wizard is where you choose the Action path and the settings for the `ActionForm` associated with this `Action` class. You choose a context-relative path to the `Action`, the `ActionForm` to use, the scope of the `Action`, the validation for the `Action`, and the input JSP. The input JSP is where control is returned in the case of validation errors. Errors are then displayed in the `<html:errors />` tag of that JSP, usually the same JSP that originally submitted.

Figure 8.9 Configuration Information page — Action wizard



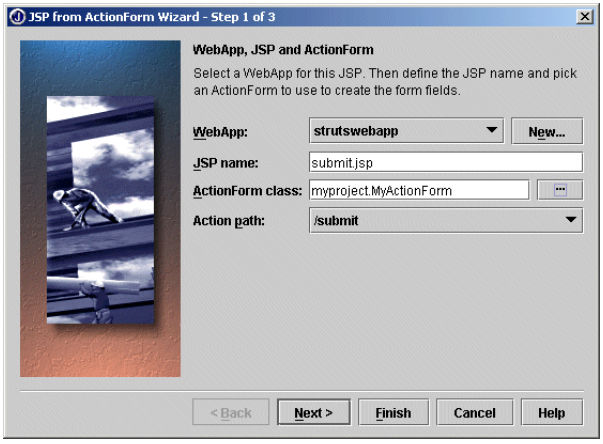
JSP From ActionForm wizard

The JSP From ActionForm takes an `ActionForm` class and creates a JSP page. The form is populated by selected fields of the `ActionForm`. You can select which fields to add to the JSP, as well as the type of tag to use for the field, i.e. text, textarea, hidden, password. This wizard is the reverse of the ActionForm Wizard.

WebApp, JSP And ActionForm page

The WebApp, JSP And ActionForm page of the wizard is where you specify the WebApp to use. You can choose an existing WebApp or create a new one by choosing the New button to display the Web Application wizard. You also use this page of the wizard to choose the JSP name, the `ActionForm` class to use to create the fields, and the context-relative action path.

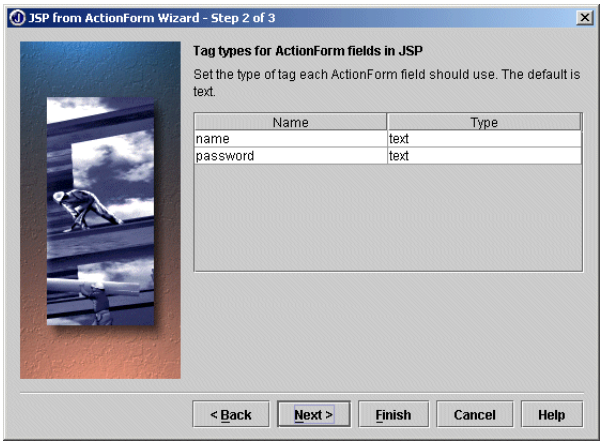
Figure 8.10 WebApp, JSP And ActionForm page — JSP From ActionForm wizard



Tag Types For ActionForm Fields In JSP page

The Tag Types For ActionForm Fields In JSP page is where you choose the type of tag each `ActionForm` field should use. The Name of the field is derived from the getters and setters in the `ActionForm` class specified on the previous page of the wizard. The default Type is set to Text; however you can change it to hidden, textarea, or password. Click in the Type field to display a drop-down list. You can also specify that this field not be used.

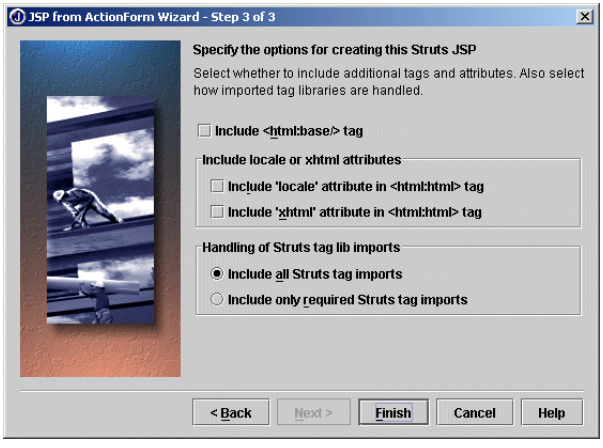
Figure 8.11 TagTypes For ActionForm Fields In JSP page — JSP From ActionForm wizard



Specify The Options For Creating This Struts JSP page

The Specify The Options For Creating This Struts JSP page allows you to select additional tags and attributes. You can choose if the Struts `<html:base/>` tag is used. You can also select whether the Struts `locale` or `xhtml` attributes are used and choose how Struts tag library imports are imported.

Figure 8.12 Specify The Options For Creating This Struts JSP page — JSP From ActionForm wizard



Struts Conversion wizard

The Struts Conversion wizard converts existing HTML or JSP pages to use Struts specific tags. The wizard will:

- Rename `.html` files to `.jsp` files if required.
- Allow you to select which of the Struts convertible tags to actually convert.
- Read and parse the existing HTML and identify the tags.
- Replace appropriate tags with Struts tags.
- Include any imports that are required by Struts.
- Write out the final file, preserving as much of the original structure and content as possible.

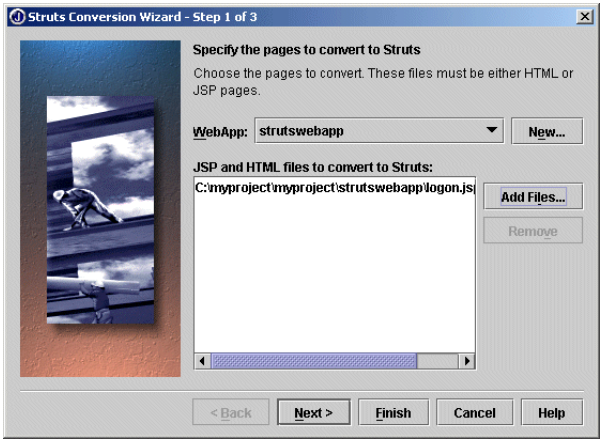
Warning The file created by the Struts Conversion wizard replaces your original file. However, you can use JBuilder's history view to recover it.

The Struts Conversion wizard is available from the Web page of the object gallery. It is also available from the editor context menu when a JSP or HTML file is open. If you choose the wizard from the context menu, the file to be converted will be listed in the JSP And HTML Files to Convert list on the first page of the wizard.

Specify The Pages To Convert To Struts page

The Specify The Pages To Convert To Struts page is where you choose the pages to convert. Each file must either be an HTML file or a JSP file. You also choose the WebApp for the files that are created by the wizard. You can specify an existing WebApp or choose a new one by clicking the New button to display the Web Application wizard.

Figure 8.13 Specify The Page To Convert To Struts page — Struts Conversion wizard



Tags To Convert page

The Tags To Convert page is where you choose the tags to convert to Struts. This page displays all the available tags to convert. You can select individual tag classes for conversion. The Select All button selects all the tags; the Select None button un-selects all tags.

Figure 8.14 Tags To Convert page — Struts Conversion wizard

Specify The Options For Converting Tags To Struts page

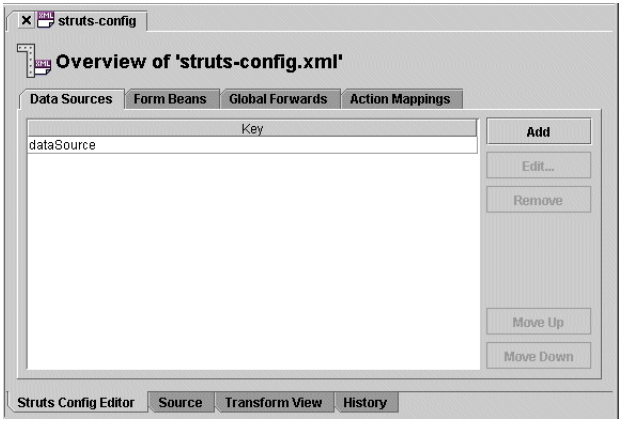
The Specify The Options For Converting Tags To Struts page allows you to select additional tags and attributes. You can choose if the Struts `<html:base />` tag is used. You can also choose to remove the `value` attributes in text input tags, as these are not used by Struts. Additionally, you can select the `locale` or `xhtml` attributes and choose what Struts tag library imports are used.

Figure 8.15 Specify The Options For Converting Tags To Struts page — Struts Conversion wizard

Struts Config Editor

The Struts Config Editor is a visual editor that allows you to edit the `struts-config.xml` file. This file is a deployment descriptor for your Struts web application and must be located in the `WEB-INF` directory. When you use JBuilder wizards to create your Struts-enabled web application, `struts-config.xml` is created for you and placed in the required location. The `struts-config.xml` file tells the controller servlet (`ActionServlet`) about your application mappings.

Figure 8.16 Struts Config Editor



For more information on the Struts Config Editor, see Chapter 13, "Editing the struts-config.xml file."

Struts framework implementations in JBuilder

As you create a Struts-enabled web application with JBuilder's tools and wizards, JBuilder seamlessly implements the Struts framework. It updates your `web.xml` file, creates a `struts-config.xml` file, copies Struts TLD files to the `WEB-INF` directory, and adds TLD mappings to the `web.xml` file.

JBuilder will:

1. Update the `web.xml` file, the web application's deployment descriptor, by:
   - Adding a servlet mapping for the control servlet (`ActionServlet`) to `web.xml`.
   - Enabling debugging at level 2.
   - Setting the configuration file value to the default, `struts-config.xml`.
   - Mapping the servlet using the `*.do` extension mapping.

To do this, JBuilder adds the following code to the top of the web.xml file:

```
<servlet>
  <servlet-name>action</servlet-name>
  <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
  <init-param>
    <param-name>debug</param-name>
    <param-value>2</param-value>
  </init-param>
  <init-param>
    <param-name>config</param-name>
    <param-value>/WEB-INF/struts-config.xml</param-value>
  </init-param>
  <load-on-startup>2</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>action</servlet-name>
  <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

Note If the web.xml file already maps a servlet named action, none of these changes are made.

2. Create WEB-INF/struts_config.xml if one doesn't already exist.
3. Copy the Struts TLD files to the WEB-INF directory.
4. Add the mappings for the TLD files to web.xml, using their locations in the WEB-INF directory. The mappings look like this:

```
<taglib>
  <taglib-uri>/WEB-INF/struts-bean.tld</taglib-uri>
  <taglib-location>/WEB-INF/struts-bean.tld</taglib-location>
</taglib>
<taglib>
  <taglib-uri>/WEB-INF/struts-html.tld</taglib-uri>
  <taglib-location>/WEB-INF/struts-html.tld</taglib-location>
</taglib>
<taglib>
  <taglib-uri>/WEB-INF/struts-logic.tld</taglib-uri>
  <taglib-location>/WEB-INF/struts-logic.tld</taglib-location>
</taglib>
<taglib>
  <taglib-uri>/WEB-INF/struts-template.tld</taglib-uri>
  <taglib-location>/WEB-INF/struts-template.tld</taglib-location>
</taglib>
```

Creating a Struts-enabled web application in JBuilder

Creating a Struts-enabled web application in JBuilder is a multi-step process. It's best to first design your application on paper. Determine the logical flow of the application. What view will be displayed to the user first? What action will follow that view? How will errors be handled? There are several comprehensive Struts tutorials on the web that can help you with application design and flow. Look at the links on the Jakarta Struts Resources Tutorials page at `http://jakarta.apache.org/struts/resources/tutorials.html`.

Once your application design is in place, you can start using JBuilder's tools and wizards to create it. The following steps provide a brief overview of creating a Struts-enabled web application in JBuilder.

1. Create a new project.
2. Use the WebApp wizard (File | New | Web tab | Web Application) to create your Struts-enabled web application. Choose the Struts framework.
3. Use the JSP wizard (File | New | Web | JavaServer Page) to create the JSP page for the application's introductory view. This is the view layer of the Struts web application.
   a. On the Declare JSP And Components page of the wizard, enter the JSP name. Do not select either of the options.
   b. On the Edit JSP File Details page, choose the background color you want for the JSP. Do not generate a Submit form. Scroll down the list of Tag Libraries until the Struts tag libraries are displayed. Choose the Struts tag libraries you will use in this JSP:
      - `struts-beans`: Tags to display data from JavaBeans, such as data returned from a database query.
      - `struts-html`: Tags that replace many of the common HTML elements.
      - `struts-logic`: Tags that perform iterations and conditional processing of view data, such as presenting database data in a tabular format.
      - `struts-template`: Tags that import snippets to JSP pages, for example, a company logo and header that are common across all pages of a website.
   c. You can click Finish on this page, you do not need to set options on other pages.

4. Add Struts-enabled tags to the JSP to design the view. For more information on building the view layer of the Struts web application, see "Building View Components" in the *Struts User's Guide* at `http://jakarta.apache.org/struts/userGuide`. The document also includes information about building forms with Struts and internationalizing your Struts web application. It additionally discusses other presentation techniques, such as includes, tiles (a feature of Struts 1.1), image rendering components, and text rendering.

Note If you don't use Struts tags in your JSP, you can use the Struts Conversion wizard to convert the HTML tags in your JSP to Struts.

5. Use the ActionForm wizard (File | New | Web tab | ActionForm) to design the `ActionForm` for the view layer.
   a. On the Web Application And Class Info For ActionForm page of the wizard, choose the web application and package for the `ActionForm`. Usually, this will be the default value. Enter the name of the class. Use a name that is similar to the JSP name.
   b. On the Field Definition For ActionForm page, choose Add From JSP if you have already added fields to the JSP. Choose the JSP from the Add JSP dialog box. The fields and their types are automatically filled in.
   c. Leave the defaults set on the Select Additional Options page. The default option automatically creates the `<form-bean>` mapping in `struts-config.xml`.
   d. Click Finish.
6. JBuilder automatically creates getters and setters in the `ActionForm`, based on the field names. You do not need to edit this file. For more information on the `ActionForm` class, see "ActionForm classes" in the *Struts User's Guide* at `http://jakarta.apache.org/struts/userGuide`.
7. Use the Action wizard (File | New | Web tab | Action) to create the `Action` for the view layer.
   a. On the WebApp and Name For Action page, accept the default web application, package, and base class. Change the `Action` name to the name you want. It should be a name that is similar to the JSP file name.

b. On the Configuration Information page, leave the Action Path set to the default. Choose the form bean from the drop-down list (this is the name assigned to the `ActionForm` you created with the ActionForm wizard). Set the Input JSP the the JSP where control is returned to in the case of validation errors. You do not need to set either the Scope or Validate FormBean options.

c. Click Finish to create the `Action` class.

8. Open the `Action` class in the editor. Add business logic to the `perform()` method. Once you add code to the method, you can delete the last line of the method, the error handling line. (This line will become unreachable, as you've implemented the method.) For more information on the `Action` class, see "Action classes" in the *Struts User's Guide* at `http://jakarta.apache.org/struts/userGuide`.

9. Add the action mapping to the JSP, in the `action` element of the `<html:form />` tag.

10. Choose File | Save All, then choose Run | Run Project to run your project.

Chapter

9

Configuring your web server

Web Development is a feature of JBuilder Enterprise

Both Java servlets and JavaServer Pages (JSP) run inside web servers. Tomcat, the JavaServer Pages/Java Servlets reference implementation, is included with JBuilder. Although it might differ from your production web server, Tomcat allows you to develop and test your servlets and JSPs within the JBuilder development environment.

JBuilder Enterprise supports the Borland Enterprise Server, Sun iPlanet, IBM WebSphere and BEA WebLogic application servers. These application servers contain web servers. However, in order to use the contained web server, you must have the corresponding application server installed and configured. You cannot use the web server outside of its application server container. Once you've configured your web application and web server, you can compile, run and debug your servlet and JSP. For more information, see Chapter 10, "Working with web applications in JBuilder."

Note In JBuilder Enterprise, you can however, use the Tomcat web server as a stand-alone web server.

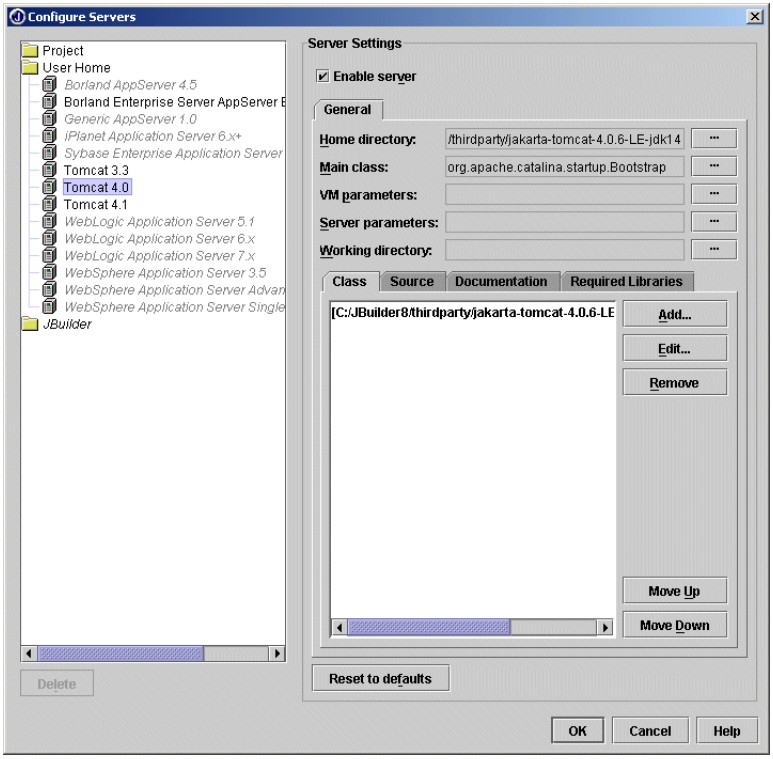
Viewing Tomcat configurations

When you install JBuilder Enterprise, three versions of Tomcat, 3.3, 4.0 and 4.1, are automatically installed in your JBuilder directory. By default, they are automatically configured. You can examine the configuration with the Configure Servers dialog box.

Tomcat 4.0 and 4.1 are the JDK 1.4 Lightweight Editions. These editions do not contain an XML parser (JDK 1.4 has one built in) and a few other publicly available components. You can download the full edition from `http://jakarta.apache.org/` then use the Configure Servers dialog box to point to them.

Note In JBuilder Enterprise, Tomcat 4.0 is the default server. To change the default for your project, see "Selecting a server for your project" on page 9-4.

1 Choose Tools | Configure Servers. The Configure Servers dialog box is displayed.



Note In the tree on the left side of the dialog box, entries in gray have not yet been configured. Entries in red are invalid.

2 Choose one of the Tomcat installations from the User Home folder. The Enable Server option at the top of the right side of the dialog box is automatically checked. You do not need to change any settings.

3 Click OK to close the dialog box.

If you have installed Tomcat to another directory, and want to run Tomcat from that directory instead of the default, you can change the settings to point to that directory. The following table explains the settings.

Table 9.1 Configure Server dialog box settings for Tomcat

Option	Description
Home directory	Tomcat's home directory. If you're reconfiguring one of the Tomcat versions installed with JBuilder, this will be in the `<jbuilder>/thirdparty` directory.
Main class	The main class for starting the Tomcat web server.
VM parameters	The parameters to pass to the Java VM running the web server.
Server parameters	The parameters to pass to the web server.
Working directory	The working directory.
Class	The location of Tomcat class files.
Source	The location of Tomcat source files.
Documentation	The location of Tomcat documentation files.
Required Libraries	The libraries that Tomcat requires.

If you'd like more information about Tomcat or would like to run it stand-alone, refer to the documentation directory of JBuilder's Tomcat installation:

- Tomcat 3.3 — `<jbuilder>/thirdparty/jakarta-tomcat-3.3.1/doc`
- Tomcat 4.0 — `<jbuilder>/thirdparty/jakarta-tomcat-4.0.6-LE-jdk14/webapps/tomcat-docs`
- Tomcat 4.1 — `<jbuilder>/thirdparty/jakarta-tomcat-4.1.12-LE-jdk14/webapps/tomcat-docs`

Note Tomcat 4.1 does not support JSP debugging.

Configuring other web servers

JBuilder supports additional web servers other than Tomcat:

- Borland Enterprise Server
- Sun iPlanet
- Sybase Enterprise Application Server
- BEA WebLogic
- IBM WebSphere

These web servers are automatically configured when you configure their corresponding application server in the Configure Servers dialog box

(Tools | Configure Servers). You cannot use these web servers outside of their application server container.

To configure the application servers that contain these web servers, see "Configuring the target application server settings" in the *Enterprise JavaBeans Developer's Guide*.

Note The Borland Enterprise Server application server uses Tomcat internally as its production web server. You do not need to configure Tomcat separately.

Selecting a server for your project

JBuilder can target multiple application servers and their contained web servers in a single project. You can choose a single application server container for all stages of EJB and web application development. You can also choose different services provided by different servers for different aspects of development. For example, you can select the EJB service provided by the Borland Enterprise Server for enterprise bean development and the JSP/Servlet service provided by Tomcat for web application development.

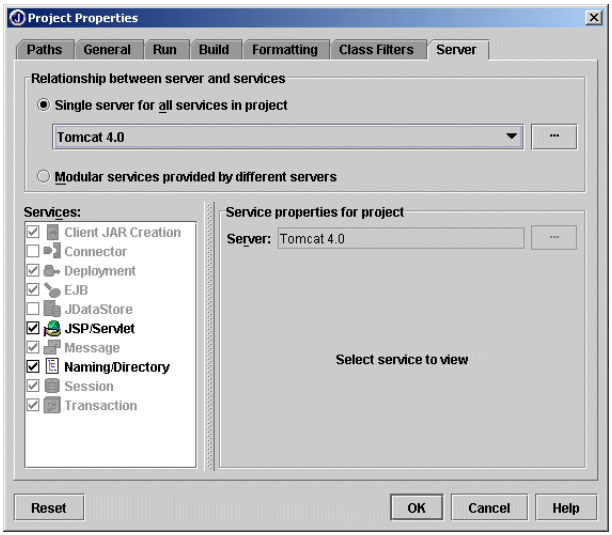
Tip If you don't have an application server installed, JBuilder will automatically target your project for Tomcat 4.0. However, if the web icons on the Web page of the object gallery are disabled, you may need to actively select a server for your project — follow the steps below.

Important The only service available for Tomcat 3.3 is the JSP/Servlet service. For Tomcat 4.0 and 4.1, the JSP/Servlet and Naming/Directory services are available.

To select one or more application server(s) (and their contained web servers) to use for your project,

1 Choose Project | Project Properties.

2 Click the Server tab. The Server page is displayed:



Note In order for an application server to display in the drop-down list at the top of the dialog box, it must be configured in the Configure Servers dialog box.

3 If you have one or more application servers installed and configured, decide whether you want to use a single application and web server for all aspects of development or different application servers to handle different areas of development.

- To use a single server,

  1 Select the Single Server For All Services In Project option and select the server from the drop-down list. This server can either be an application server that contains a web server or the Tomcat web server that is installed with JBuilder.

  2 If you want to avoid having libraries added to your project that you won't use, uncheck the check box in front of the service(s) you don't need in the Services list. If you disable services, the corresponding JBuilder features will be disabled. For example, if you turn off the JSP/Servlet service, most of the Web wizards and the JSP compilation feature are disabled.

  3 If you want to make changes to the configuration settings for the selected server, click the ellipsis button to the right of the server name and edit the settings you want on the General and/or Custom pages. Click OK when you're finished.

- To use different servers for different services,

  1. Select the Modular Services Provided By Different Servers option.
  2. If you want to avoid having libraries added to your project that you won't use, uncheck the check box in front of the service(s) you don't need in the Services list. If you disable services, the corresponding JBuilder features will be disabled. For example, if you turn off the JSP/Servlet service, most of the Web wizards and the JSP compilation feature are disabled.
  3. Update the configuration for the selected service on the right side of the dialog box. Depending on the selected server/service, this information may be able to be configured or may be read-only. For more information, see "Selecting a server" in the *Enterprise JavaBeans Developer's Guide*.
  4. To use a selected web server, click the JSP/Servlet service. In the Server drop-down list on the right slide of the dialog box, select the web server you want to use. If you want to make changes to the configuration settings for the web server, click the ellipsis button and edit the settings you want on the General page. Click OK when you're finished.

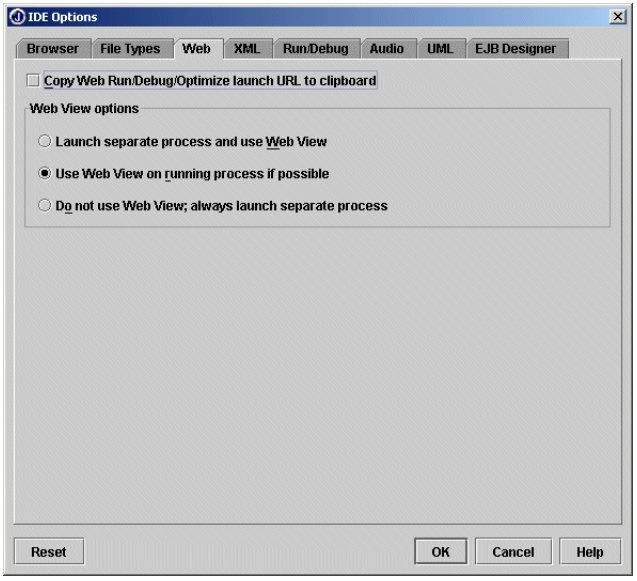
4. Click OK again to close the Servers page.

Configuring the IDE for web run/debug

Once you've set up JBuilder with a web server, you can configure options for the web server, including web view options and how the web server is launched in the IDE.

To configure how the IDE behaves when trying to Web Run or Web Debug,

1 Choose the Web tab on the IDE Options dialog box (Tools | IDE Options). The Web page looks like this:



2 Choose the Copy Web Run/Debug/Optimize Launch URL To Clipboard option to copy the URL used to launch the web application to the clipboard. This enables you to easily go to the same URL in an external browser. Set this option if you're creating a Java Web Start applet or application.

3 Choose Web View Options. These options work in conjunction with the Search For Unused Port option on the Runtime Configuration Properties dialog box when the specified port is in use by a non-web process. (See "Creating a runtime configuration" on page 10-2 for more information.)

- Choose the Launch Separate Process And Use Web View option to use both the internal web browser and an external web browser. This option automatically displays your rendered servlet or JSP in the Web View page of the content pane.
- Choose the Use Web View On Running Process If Possible option to use the internal web browser to view your web page. This option automatically displays your rendered servlet or JSP in the Web View page of the content pane. If a web server is already running, JBuilder uses the same process on the existing port. This is the default.
- Choose the Do Not Use Web View Always Launch Separate Process option when launching your web application in an external web browser.

Chapter

10

Working with web applications in JBuilder

Web Development is a feature of JBuilder Enterprise

JBuilder provides commands on the project pane's context menu that make it easy to run and debug servlets and JSPs. Web Run runs your servlet or JSP using the selected server runtime configuration. Web Debug debugs your JSP or servlet using the selected server runtime configuration with debug options, allowing you to easily step through and examine your code. Web Optimize runs the server with an optimizer/profiler like OptimizeIt (you need to have one installed). The web commands display the runtime configuration(s) available in the project.

Selecting Web Run or Web Debug runs or debugs that servlet or JSP in its WebApp context. If the JSP, HTML, or SHTML file is not in a WebApp, it cannot be web run or web debugged, and the Web Run and Web Debug commands are not displayed on the context menu. For more information about WebApps, see "The WebApp and its properties" on page 3-5.

Note If you set a Launch URI for a WebApp (in the Web Application wizard), you can run the WebApp from that specified location from the WebApp node's context menu. This way, if your application has an obvious starting point, you can run, debug, or optimize without having to drill down into the web content.

When you create a servlet or JSP using JBuilder's Servlet or JSP wizard, the Web Run and Web Debug commands are automatically enabled.

Important Applets cannot be web run or web debugged. This is because applets don't have a URL or a web context to run in. Additionally, applets run in a client browser as opposed to a server. Typically, you run an applet in Sun's AppletViewer or in JBuilder's AppletTestbed. For more information, see "JBuilder's AppletTestbed and Sun's appletviewer" on page 14-20.

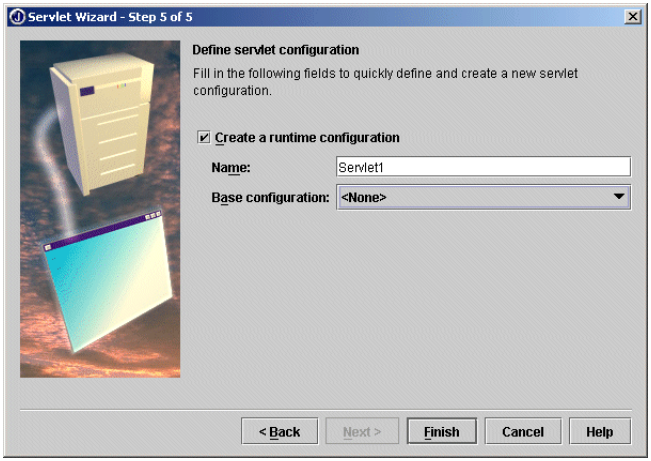
Before you can run your applet, servlet or JSP, you need to create a runtime configuration. For servlets not created with the wizard, you also need to set run properties. Then, once you compile your servlet or JSP, you can web run it.

Creating a runtime configuration

Before you can run your applet, servlet or JSP in JBuilder, you need to create a runtime configuration. The configuration sets the type of runner and determines what is run. You can also set debugging options for the configuration. There are two ways to create a configuration for an applet, JSP, or servlet — you can use the wizard to create the basic configuration and then customize it if needed. Alternatively, you can create the configuration directly using the Run | Configurations command.

Creating a runtime configuration with the wizards

The Runtime Configuration page of the Applet, JSP and Servlet wizards allow you to quickly create a runtime configuration. For the Servlet wizard, the Runtime Configuration page looks like this:



When you create an Applet runtime configuration with the wizard, the Applet runner becomes active. The applet is run in JBuilder's applet viewer, AppletTestbed, using the applet class file. (Note that you cannot Web Run or Web Debug an applet because it does not launch a server to run; it runs in a browser. Additionally, applets do not have a web application context, or WebApp.)

When you create a Servlet runtime configuration with the wizard, the Server runner becomes active. The server selected in the Project Properties Server page becomes the active web server. The URI to run is taken from

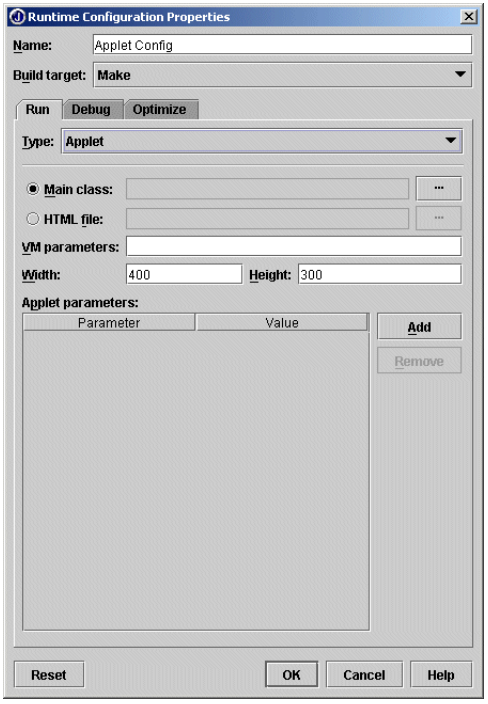
the entry in the URL Pattern field on the Enter WebApp Details page of the Servlet wizard and from the servlet's WebApp.

When you create a JSP runtime configuration with the wizard, the Server runner becomes active. The server selected in the Project Properties Server page becomes the active web server. The URI to run is taken from the WebApp and the JSP file name.

Creating an applet runtime configuration

If you have an existing program, or choose not to create a runtime configuration with the Applet wizard, you can create a configuration using the Runtime Configuration Properties dialog box. To do so,

1. Choose Run | Configurations. The Project Properties Run page is displayed.
2. Click New to create a new configuration. The Runtime Configuration Properties dialog box is displayed.



3. Enter the name of the configuration in the Name field.
4. Choose the build target to execute before running. For more information, see "Build targets" in the "Setting runtime configurations" chapter of *Building Applications with JBuilder*.

5. Choose Applet from the Type drop-down list.
6. Choose the Main Class option to run your applet using JBuilder's applet viewer, AppletTestbed. When you create your applet with the Applet wizard, it sets the main class for you. The main class must contain an `init()` method. Browse to the main class by choosing the ellipsis button and selecting the class in the Select Main Class For Project dialog box.
7. Choose the HTML File option to run your applet in Sun's **appletviewer.** The HTML file must contain the `<applet>` tag and code attribute must contain the fully qualified class name. Browse to the HTML file by choosing the ellipsis button and selecting the file in the Select HTML File To Run dialog box.
8. In the VM Parameters field, enter any parameters to pass to the Java Virtual Machine (VM). For more information on the Java VM and the options you can pass to it, see "Basic Tools: java — The launcher for Java technology applications".
9. If you selected the Main Class option to run the applet, enter the applet's initial size and any initial parameters:
   a. Enter the applet's initial width and height in the Width and Height fields.
   b. Enter any applet parameters in the Applet Parameters list. Enter the parameter name in the Name column and its initial value in the Value column.

Note If you run the applet from the HTML page, you don't need to set parameters since they are already set in the HTML page.

10. When you finish, click OK to close the Runtime Configuration Properties dialog box. The Run page of the Project Properties dialog box is displayed. Your new configuration is highlighted.
11. To make this configuration the default configuration, choose the checkbox in the Default column. (Only one configuration in the project can be the default configuration.) To add this configuration to the project pane's context menu, choose the checkbox in the Context Menu column. Click OK to close the dialog box.

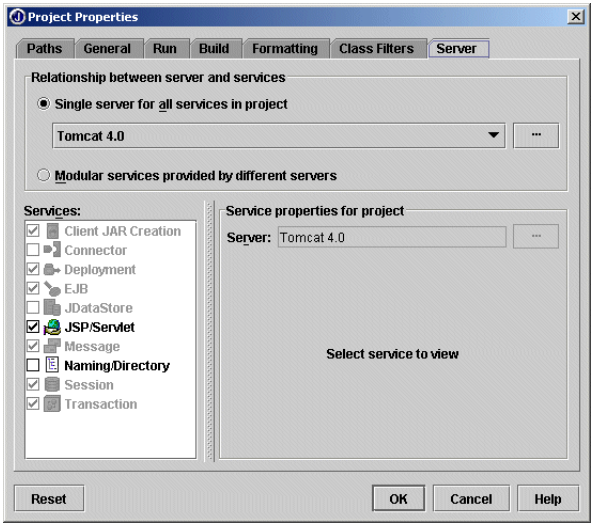
Applets cannot be web run or web debugged. This is because applets don't have a URL or a web context to run in. Additionally, applets run in a client browser as opposed to a server. Typically, you run an applet in Sun's AppletViewer or in JBuilder's AppletTestbed. For more information, see "JBuilder's AppletTestbed and Sun's appletviewer" on page 14-20.

For more information on runtime configurations, see "Setting runtime configurations" in the "Running Java programs" chapter of *Building Applications with JBuilder*.

Creating a server runtime configuration

Before configuring your server, you need to verify that a server is enabled for your project. To do so,

1 Open the Project Properties dialog box and choose the Server tab.



2 Verify that a server is selected.

- If the Single Server For All Services In Project option is selected for the project, make sure that the drop-down list is not set to `<None>`. Verify that the selected server supports the JSP/Servlet service. The check box for that service must be enabled in the tree on the left side of the page. If the server doesn't happen to support a JSP/Servlet service, that service won't be available.
- If the Modular Services Provided By Different Servers option is selected, choose the JSP/Servlet service on the left side of the dialog box page. Choose a server from the drop-down list on the right side of the page — make sure the choice is not set to `<None>`.

Note Note that the selected server may allow services to be disabled at runtime. For example, the Borland Enterprise Server will allow this. Make sure that the JSP/Servlet service has not been disabled in the runtime configuration; if it is, the web commands are not available.

3 Click OK to close the Project Properties dialog box.

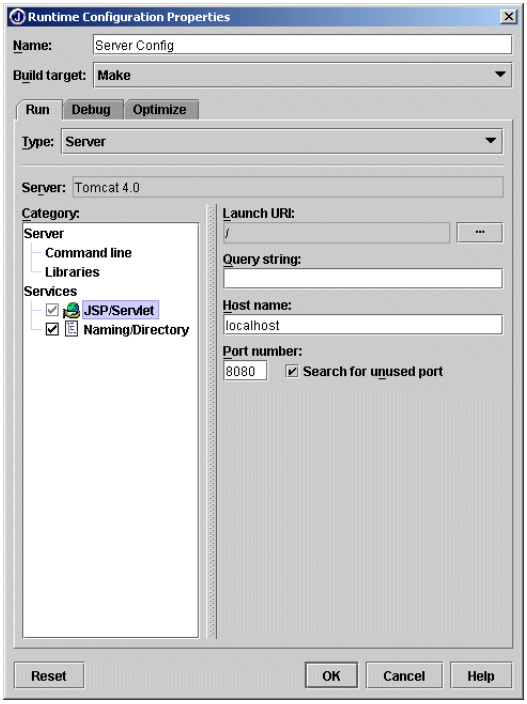
For more information on web servers, see Chapter 9, "Configuring your web server."

Once a server is enabled for your project, you can edit or create a server runtime configuration.

Important If you have two different servers in the same project, you will need two separate runtime configurations, one for each server.

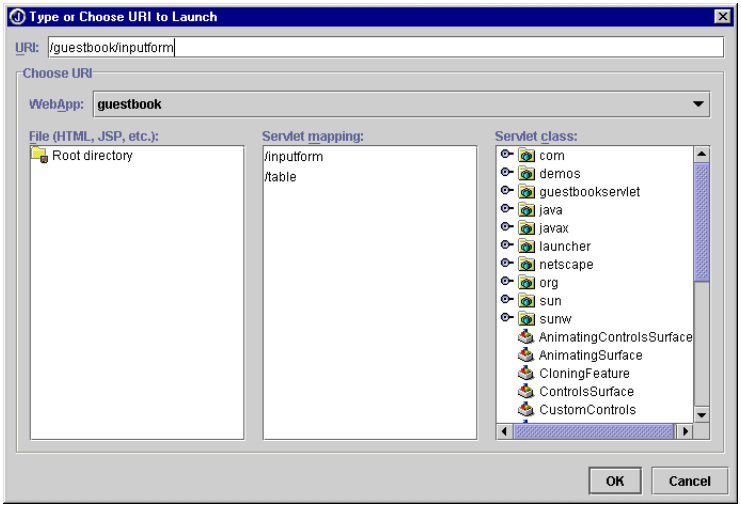
To create a server runtime configuration,

1. Choose Run | Configurations. The Project Properties Run page is displayed.
2. Click New to create a new configuration. The Runtime Configuration Properties dialog box is displayed.



3. Enter the name of the configuration in the Name field.
4. Choose the build target to execute before running. For more information, see "Build targets" in the "Setting runtime configurations" chapter of *Building Applications with JBuilder*.
5. Choose Server from the Type drop-down list.
6. Choose the JSP/Servlet service in the Category list on the left side of the dialog box.

7 To choose the servlet or JSP to launch, click the ellipsis (...) button to the right of the Launch URI field. The Type Or Choose URI To Launch dialog box is displayed. You run the servlet or JSP via a URI so that focus will switch to the web view when you use the Web Run command.



You can either directly type the URI (Universal Resource Identifier) into the URI text field at the top of the dialog box or choose the WebApp and URI from the trees at the bottom of the dialog box:

a Choose the WebApp of the URI you want to run from the WebApp drop-down list. This list displays all WebApps that are defined in your project.

b Choose the URI directly from one of the three different trees at the bottom of the Launch URI dialog box. These trees roughly correspond to the different kinds of servlet mappings. The File, or web content, tree on the left creates URIs that would probably match either extension mappings (like `*.jsp`) or not match anything and get served by the default servlet. The Servlet Mapping tree in the middle contains URL patterns for exact matches. The Servlet Class tree, displayed only if the server has a servlet invoker, creates URIs that would match the servlet invoker. (Tomcat 3.3 and 4.0 have a servlet invoker; Tomcat 4.1 does not.)

For more information about URL mappings, see "How URLs run servlets" on page 10-9.

Table 10.1 URI trees

Tree Name	Description	Example of resulting URI
File (HTML, JSP, etc.)	All HTML-type files in the selected WebApp.	`/selectedwebapp/hello.html` or `/selectedwebapp/login.jsp`
Servlet Mapping	All URL pattern values that do not contain wildcards. Allows a user to invoke a servlet or JSP by name. This value was entered into the Name/URL Pattern fields on the Servlet wizard — Enter WebApp Details page.	`/selectedwebapp/form` or `/selectedwebapp/table`
Servlet Class	All classes in the opened project; however, the classes displayed are assumed to be servlets and are executed using the web server's servlet invoker.	`/selectedwebapp/servlet/com.test.MyServlet`

The URI is appended to the `hostname:port` during a run, for example:

```
http://localhost:8080/selectedwebapp/hello.html
http://localhost:8080/selectedwebapp/login.jsp
http://localhost:8080/selectedwebapp/form
http://localhost:8080/selectedwebapp/table
http://localhost:8080/selectedwebapp/servlet/com.test.MyServlet
```

8 Click OK to close the Type Or Choose URI To Launch dialog box.

9 In the Query String field on the Runtime Configuration Properties dialog box, enter any user parameters. User parameters are a series of name/value pairs separated by an ampersand, for example, `a=1&b=2`. You can also enter a query string if the client is using the `doGet()` method to read information from the servlet or JSP. This string is appended to any URL generated for a web run and usually includes parameters for the servlet or JSP. For example, your web application may contain both a servlet (`SalesHistory.java`, given the servlet name `saleshistory`) and JSP (`showproduct.jsp`) that use a product ID number. When you run the web application, the Web Run command generates the following URLs:

```
http://localhost:8080/showproduct.jsp
http://localhost:8080/saleshistory
```

If you specify `"product=1234"` in the Query String field, the query gets added to the end of the URL:

```
http://localhost:8080/showproduct.jsp?product=1234
http://localhost:8080/saleshistory?product=1234
```

The question mark (?) separates the name of the servlet or JSP to run from the query string. This allows the servlet and JSP to ask for the parameter `product` and get back `1234`. Note that parameter values are always strings. Multiple parameters are separated with an ampersand (&), e.g. `product=1234&customer=6789`.

10 In the Host Name field, enter the name the web server should assume. Do not choose a name already in use in your sub-net. `localhost` is the default.

11 Enter the port number the web server should listen to in the Port Number field. Generally, you should use the default port number, `8080`. Change this value only if the default value is in use.

12 Choose the Search For Unused Port option to tell JBuilder to choose another port if the specified one is in use. It is useful to select this option when you are running more than one servlet or JSP, as otherwise you might get a message that the port is busy. It is also useful to check this option in the event that a user problem brings the web server down. With this option selected, you are protected if the web server is not shut down properly. This option works in conjunction with the Launch options on the IDE Options page when the specified port is in use by a non-web process. (See "Configuring the IDE for web run/debug" on page 9-7 for more information.)

To create a runtime configuration with debug options,

1 Click the Debug tab on the Runtime Configuration Properties dialog box.

2 For information on options, see "Setting debug configuration options" in "Debugging Java programs" in *Building Applications with JBuilder*.

When you're done, click OK to close the Runtime Configuration Properties dialog box. Click OK again to close the Project Properties Run page and save the changes to your runtime configuration.

How URLs run servlets

A Uniform Resource Locator (URL) is used to run a servlet. A Uniform Resource Identifier (URI) is a broader concept that includes both URLs and *request-URI-paths.* The request-URI-path follows the server name and optional port number. It starts with a forward slash.

For example, in the URL:

```
http://localhost:8080/filename.html
```

`/filename.html` is the request-URI-path.

In a non-servlet-container web server like IIS or Apache without Tomcat, the basic handling of the request-URI-path is simple. The web content is rooted in a particular directory, so the web server can resolve that path,

using the leading slash to indicate the web-root directory. It can then return the corresponding file, if it's there.

Servlet containers like Tomcat and WebLogic are more complex, but more flexible. These containers allow contexts and mappings, so that your web application can have any number of named contexts. Each context is mapped to its own root directory.

The servlet container's first job in evaluating the request-URI-path is to see if the first part of the path matches a context name. In JBuilder, these are the WebApp names. During a Web Run, those names are associated with WebApp root directories. (For more information on WebApps, see "The WebApp" on page 3-1.)

If there is a match, the first part of the request-URI-path becomes the context path. The remaining part of the path, starting with a slash, becomes the *URL-path-to-map*. If there is no match, the context path is an empty string, and the entire request-URI-path is considered the URL-path-to-map.

For example, for a project with a single WebApp named `myprojectwebapp`, the request-URI-path `/myprojectwebapp/subpackage/somename.jsp` would be evaluated as follows:

- The context path would be `/myprojectwebapp`.
- The URL-path-to-map would be `/subpackage/somename.jsp`.

However, the request-URI-path `/test/subpackage/somename.jsp` would not contain any context path in the evaluation, since the only existing WebApp is `myprojectwebapp`. In this case, the context path would be empty and the URL-path-to-map would be the entire URI: `/test/subpackage/somename.jsp`

Note that the context configuration is done in a server-specific way. However, the matching of the URL-path-to-map is done via the servlet-mapping entries in each context's standard WebApp deployment descriptor, the `web.xml` file. Each servlet-mapping has two parts: a URL-pattern and a servlet-name.

There are three special kinds of URL-patterns:

Table 10.2 URL patterns

Pattern Type	Description
Path mapping	Starts with **/** and ends with **/***
Extension mapping	Starts with ***.**
Default mapping	Only includes **/**

Note The three trees at the bottom of the Launch URI dialog box roughly correspond to the three different kinds of mappings. For specifics on how these mappings work in the Launch URI dialog box, see "Creating a server runtime configuration" on page 10-5.

All other URL-pattern strings are used for exact matches only. When matching the URL-path-to-map, an exact match is tried first. For example, if the WebApp `somewebapp` includes a URL-pattern `/test/jspname.jsp`, the corresponding servlet would be used.

If there is no exact match, a path match is attempted, starting with the longest path. In the default context, the URL-pattern `/test/*` would be the first match.

If there is no path match, then an extension match is tried. The URL-pattern `*.jsp` would match both of the following two request-URI-paths: `/testwebapp/subpackage/jspname.jsp` and `/myprojectwebapp/anyfolder/myjsp.jsp`.

Finally, if there is no extension match, the servlet mapped to `/` (the default servlet) is used.

Most web servers already have some default mappings, again done in a server-specific way. For example, `*.jsp` would be mapped to a servlet by:

1. Taking the path that was matched (e.g. `/sub/somename.jsp` or `/test/subpackage/somename.jsp`)
2. Finding the corresponding file relative to the context's web-root directory
3. Converting it to a servlet if not done already
4. Running that servlet

Another typical default mapping is `/servlet/*`, which is an *invoker servlet.* An invoker servlet:

1. Takes whatever follows `/servlet/` as a class name
2. Tries to run that class as a servlet

The default servlet (the one mapped to the URL-pattern `/`):

1. Takes the URL-path-to-map (that didn't map to anything)
2. Finds the corresponding file relative to the context's web-root directory
3. Sends it to the browser

When you create a standard servlet in the Servlet wizard, it does the minimum needed to create a mapping. For example, with the name `myproject.MyServlet`, the Servlet wizard:

1. Creates a servlet with the servlet-name `myservlet` associated with the servlet-class `myproject.MyServlet`
2. Creates the URL-pattern `/myservlet` and maps that to the servlet-name `myservlet`

If that was done for the WebApp `test`, then `/test/myservlet` would run the servlet class `myproject.MyServlet` within the WebApp context `test`. For more information on how the Servlet wizard creates a mapping, see "Enter WebApp Details page" on page 5-6.

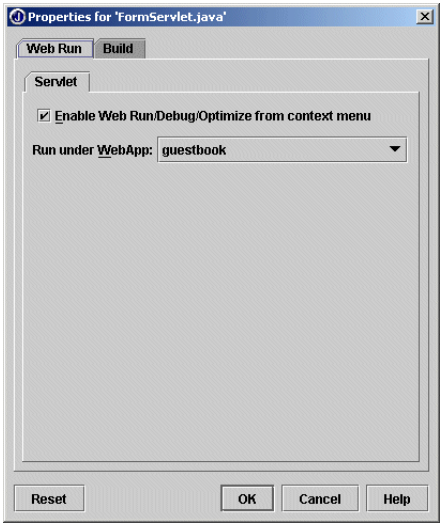
When you Web Run a servlet `.java` (or `.class`) file, the servlet invoker is used if no exact URL-pattern is found for that servlet class. For the previous example, a web run under the WebApp `test` would run `/test/myservlet/myproject.MyServlet`.

Setting run properties

If you've created a servlet by coding, not using the Servlet wizard, you need to enable the Web Run and Web Debug commands and choose the WebApp to run under. If you're using a servlet or JSP created with the Servlet or JSP wizards, you don't need to follow these steps. JBuilder has already completed this setup.

To enable web commands and check the WebApp,

1. Right-click the `.java` servlet file in the project pane.
2. Choose Properties to display the Properties dialog box.



3. Choose the Enable Web Run/Debug/Optimize From Right-Click Menu option on the Servlet tab of the Web Run page. This option displays the Web Run and Web Debug commands when you right-click the servlet file in the project pane. (If you create your file using the Servlet or JSP wizards, this option is automatically set.)

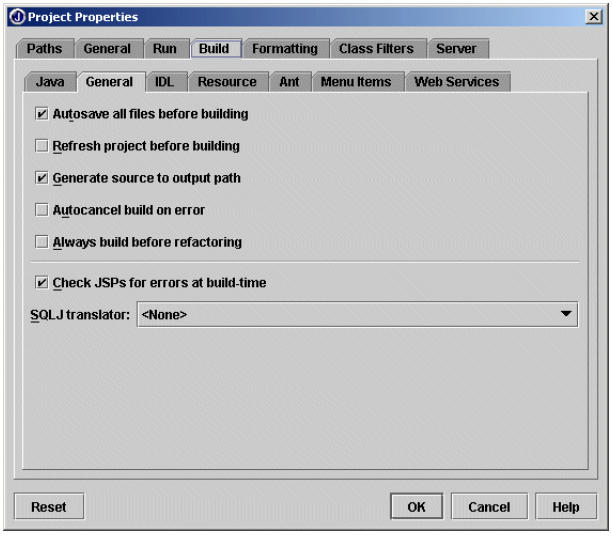
4 Choose the WebApp to run under from the Run Under Context Path drop-down list. This path is used when you select the Web Run command. It is set in the Web Application wizard.

5 Click OK to close the dialog box.

Compiling your servlet or JSP

As with any Java program, before running it, you need to compile it. You can compile the entire project with the Project | Make or Project | Rebuild commands. You can also compile the individual servlet or JSP file by right-clicking the file in the project pane and choosing Clean, Make or Rebuild. (Note that for JSPs, only the Make command is available.) Compiler errors are displayed on the Build tab of the message pane. For more information on the compiler, see the chapter called "Compiling Java programs" in *Building Applications with JBuilder*.

JSPs are an extension of the Servlet API and are compiled to servlets before they are used. This requires the compilation process to translate JSP file names and line numbers into their Java equivalents. JSPs can be compiled at build-time. To enable this feature for all JSPs in your project, choose the Check JSPs For Errors At Build-Time option on the General tab of the Project Properties Build page.



You can set this property for each JSP file in your project, so that you can exclude certain files from compilation. For example, JSPs that are intended to be included in other JSPs probably would not compile successfully on their own, so you would exclude those files.

Web running your servlet or JSP

To web run your servlet or JSP in JBuilder, right-click the servlet or JSP file in the project pane and choose Web Run. The Web Run command runs your configuration in the selected web server without debugging it. If your servlet runs from an HTML or SHTML file, right-click that file and choose Web Run.

If the Web Run or Web Debug command is grayed out on the context menu for a servlet or JSP, check the runtime configuration. To run a web application, you need to create a runtime configuration with the server type selected as the current runner. That server must support the JSP/Servlet service listed on the Project Properties Server page on the tree in the left side of the dialog box. To create a runtime configuration, see "Creating a runtime configuration" on page 10-2.

Note Applets cannot be web run or web debugged. This is because applets don't have a URL or a web context to run in. Additionally, applets run in a client browser as opposed to a server. Typically, you run an applet in Sun's AppletViewer or in JBuilder's AppletTestbed. For more information, see "JBuilder's AppletTestbed and Sun's appletviewer" on page 14-20.

Starting your web server

When you choose Web Run, JBuilder starts your web server, using:

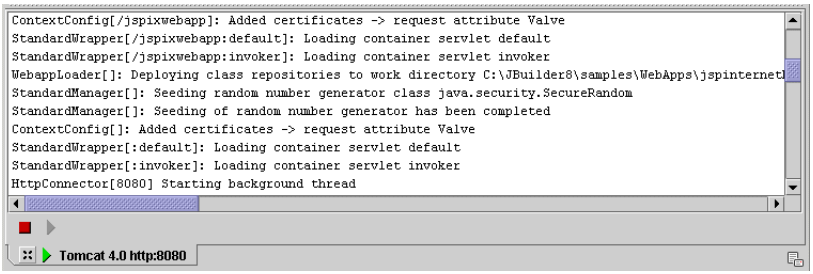
- The runtime configuration (see "Creating a runtime configuration" on page 10-2 for more information).
- The options set on the Web page of the IDE Options dialog box (Tools | IDE Options). See "Configuring the IDE for web run/debug" on page 9-7 for more information.

Messages are logged to the web server tab displayed in the message pane at the bottom of the AppBrowser. HTTP commands and parameter values are also echoed to this pane. The name of the tab will reflect the name of the web server, for example Tomcat for the Tomcat web server.

Note When you Web Run a servlet or JSP using the iPlanet web server, you'll see two new tabs in the message pane. One is for the iPlanet preparation phase, and shows the command line used to set up the web server. The other is for output from the web server itself. Both tabs are called iPlanet.

An example of output to the message pane for the Tomcat web server is displayed below:

Figure 10.1 Tomcat messages



When the Web Run starts, two new tabs are displayed in the content pane: Web View and Web View Source. Click the tab to open the web view and the web view source.

Note Web View and Web View Source tabs will not display if you have selected the Do Not Use Web View option on the Web page of the IDE Options dialog box. For more information, see "Configuring the IDE for web run/debug" on page 9-7.

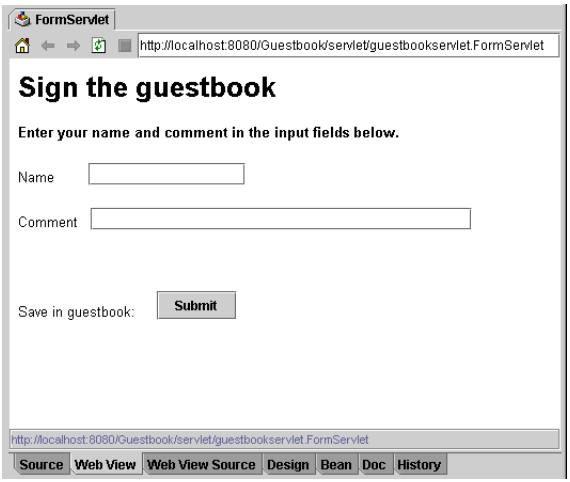
Web view

Formatted output is displayed in the web view pane of the content pane. The generated URL is displayed in the location field at the top of the web view.

The web view displays the servlet after it has been processed by the servlet engine. There may be a delay between when the servlet or JSP file is edited and when the change is shown in the web view. To see the most recent changes, select the Refresh button at the top of the web view.

Note that web view is simply a browser and is not tied to other JBuilder features. For example, you can view your formatted output in a browser external to JBuilder. Options on the Web page of the IDE Options dialog box speed up repeated use of this feature. See "Configuring the IDE for web run/debug" on page 9-7.

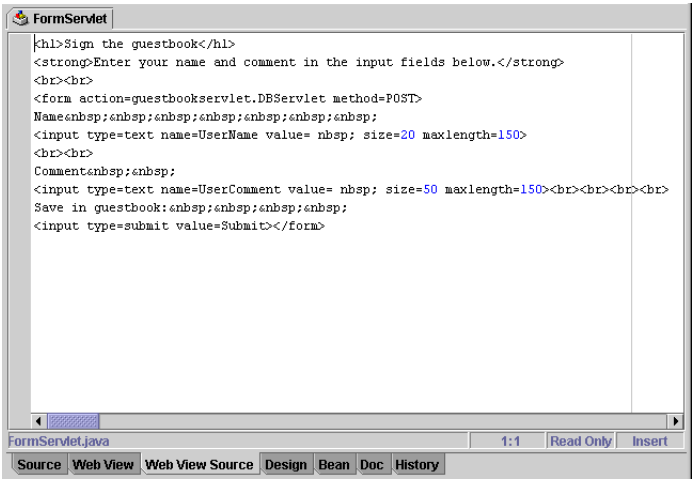
Figure 10.2 Web view output



Web view source

Raw output from the web server is displayed in the web view source pane.

Figure 10.3 Web view source



Stopping the web server


To stop the web server, click the Reset Program button on the web server tab. To start the web server again and re-run your web application, click the Restart Program button. You'll usually follow these steps when you make changes to source code, re-compile, and re-run. You don't need to close the web server pane each time you start the web server.

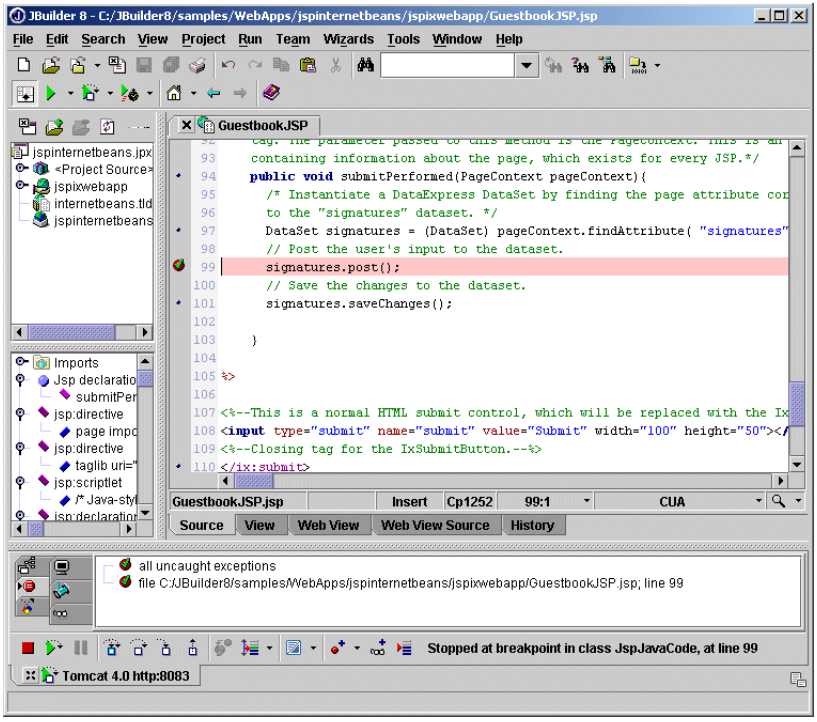
Note The first time you press the Reset Program button, it simply sends a command to the server to shutdown. In particular, if you are trying to debug your webapp's lifecycle event handlers, this would call the `Servlet.destroy()` and `ServletContextListener.contextDestroyed()` methods. If you're not debugging, the server will usually shutdown by itself (Tomcat does this in a few seconds; other server take longer). In any case, if you press the Reset Button a second time, the server process is terminated immediately.

Web debugging your servlet or JSP

To web debug, launch the server in debug mode with the Web Debug command. You can then debug any servlet or JSP that the server processes. To make the debugger stop at a specific line of code, set a breakpoint in your servlet or JSP before starting the server.

Note JBuilder provides source debugging for JSPs. This allows you to trace through your JSP code directly; you don't need to trace through the servlet that the JSP is compiled to and try to match up line numbers in order to find errors. Note that Tomcat 4.1 does not support JSP source debugging.

The Web Debug command displays the debugger in the web server pane. Both web server and debugger messages are written to the Console view of the debugger.



Note For more information on debugging, look at the following topics:

- "Debugging Java programs" in *Building Applications with JBuilder*
- "Compiling, running and debugging tutorial" in *Building Applications with JBuilder*
- "Remote debugging" in *Building Applications with JBuilder*

Chapter

11

Deploying your web application

Web Development is a feature of JBuilder Enterprise

Deploying your web application is the process of moving the web application to the web server, placing it in the correct location on the web server, and completing any other necessary steps for the web server to correctly recognize your web application. Different web servers require different steps for correct deployment of a web application. You need to consult your server's documentation for server-specific issues.

Overview

The following sections discuss some issues you will want to consider when deploying to any web server.

Archive files

Gathering your web application's files into an archive can greatly simplify deployment. Some servers require a WebApp to be packaged in a WAR (Web Archive) file. A WAR file organizes the files you need for your web application into the correct hierarchy. You can then copy the archive file to the appropriate location on your web server. This eliminates the need to copy each file individually and ensures they get copied to the proper locations.

A WAR file is a web archive. It can contain your entire web application in an appropriate structure, making that structure easier to replicate on the web server. WAR files are discussed in more detail in Chapter 3, "Working with WebApps and WAR files."

If your web application contains one or more applets, you might consider putting them in a JAR file. For more information on using JAR files to contain applets, see Chapter 14, "Working with applets."

Your web application, WAR file, or JAR file can also be packaged into an EAR file. See the online help for the "EAR wizard."

Deployment descriptors

Deployment descriptors are XML files which contain information needed by the web server about the WebApp. You will use one or more deployment descriptors if your web application contains servlets or JSPs. Check your server's documentation for more information about what deployment descriptor(s) it requires.

Applets

See "Deploying applets" on page 14-12 for information on deploying an applet.

Servlets

Servlet deployment can be tricky, because if it's not done correctly, the web server will fail to recognize your servlet. In order to simplify deployment, you should consider archiving your servlet into a WAR file. This enables you to gather all the files and resources that are needed for the servlet together in a correctly organized hierarchy prior to deployment. Then you only need to deploy the WAR file to the web server.

Regardless of the web server, successful servlet deployment requires certain information to be present in the `web.xml` deployment descriptor. Your web server could also have additional requirements. At minimum, before deploying a standard servlet, you will need to specify the following in a `servlet` element of the `web.xml` file:

- `servlet-name` — a name for the servlet
- `servlet-class` — the fully qualified class name for the servlet

Each standard servlet must also specify a `servlet-mapping` in the `web.xml`. You will need to specify the following within a `servlet-mapping` element:

- `servlet-name` — the name used in the `servlet` tag
- `url-pattern` — the URL pattern to which the servlet is mapped

A filter servlet or a listener servlet will require different tags. See "Filters page" on page 12-4 and "Listeners page" on page 12-6 for more information on the required tags for these types of servlet.

If you use JBuilder's Servlet wizard to build your servlet, the wizard will insert the required information for the servlet into the `web.xml` for you. This is true for a standard servlet, a filter servlet, or a listener servlet.

Servlets must be compiled before being deployed to the web server.

JSPs

JSPs are easier to deploy than servlets. You might want to consider archiving them into a WAR file to make deployment even easier.

JSPs are mapped by a web server in the same way that HTML files are; the server recognizes the file extension. Compilation is also handled by the web server. Remember, JSPs are compiled into servlets by the web server prior to execution.

Some JSPs use JSP tag libraries. These libraries also must be deployed to the web server, and the web server needs to know how to find them. For this reason, if you have a JSP that uses a tag library, your `web.xml` deployment descriptor will require a `taglib` element which indicates which tag library to use and where it can be found. The following information is found in the `taglib` element:

- `taglib-uri` — the URI used in the JSP to identify the tag library. This matches the taglib directive (`<%@ taglib uri="whatever" prefix="whatever">`) in the JSP.
- `taglib-location` — the actual location of the tag library.

If you use JBuilder's JSP wizard to create a JSP which uses one of the tag libraries that's recognized by the wizard, the tag library information is added to `web.xml` for you.

Testing your web application

After you have deployed your web application to the web server, you should test it to make sure that it is deployed correctly. You will want to try accessing all the pages, servlets, JSPs, and applets in your application and make sure they are working as expected. This should be done from a browser on another machine, so that you ensure the web application is accessible over the web and not just locally. You might also want to consider testing with different types of browsers, since the way your application appears in different browsers can vary, especially when using applets.

Editing deployment descriptors

Deployment descriptors are XML files which contain information needed by the web server about the WebApp. All Java-enabled web servers expect a standard deployment descriptor called `web.xml`.

If you use the Struts framework in your WebApp, a deployment descriptor called `struts-config.xml` is also required. Some other frameworks may also require their own deployment descriptors.

Some servers may also have vendor-specific deployment descriptors they use in addition to `web.xml`. For example, BEA WebLogic uses `weblogic.xml`. Check your server's documentation to find out what descriptor files it uses. Tomcat, the web server which ships with JBuilder, requires only `web.xml`.

JBuilder provides a deployment descriptor editor for `web.xml`. This is called the WebApp DD Editor. It provides a Graphical User Interface (GUI) for editing the most commonly used information in the `web.xml` file.

When the `web.xml` file is opened in the JBuilder IDE, its contents display in the structure pane, and the AppBrowser displays the WebApp DD Editor and the source editor. You can edit the `web.xml` file in either the WebApp DD Editor or the source editor. Changes made in the WebApp DD Editor will be reflected in the source, and code changes made in the source will be reflected in the WebApp DD Editor.

Caution If you enter comments in the `web.xml` file using the source editor, these will be removed if you subsequently open the file in the WebApp DD Editor.

JBuilder also provides the Struts Config Editor for editing the `struts-config.xml` deployment descriptor required by the Struts framework.

Editing vendor-specific deployment descriptors

You can also edit vendor-specific deployment descriptors in the JBuilder IDE. A vendor-specific deployment descriptor file is only recognized by JBuilder and displayed in your project if the corresponding server plug-in is properly configured.

Vendors who offer plug-ins for web servers other than Tomcat are encouraged to provide GUI editors for their vendor-specific deployment descriptors. For example, the Sybase plug-in provides a GUI editor for the `sybase-easerver_config.html` file.

If your vendor doesn't provide a GUI editor for their deployment descriptor, you can still edit the deployment descriptor's source code in JBuilder's XML source editor. To do this, you open the vendor-specific deployment descriptor and click the Source tab in the content pane.

When a vendor-specific deployment descriptor is opened in the JBuilder IDE, its contents are displayed in the structure pane, and the AppBrowser displays the source editor and the history view, along with any vendor-specific GUI editor which may be provided by your web server plug-in.

More information on deployment descriptors

For more information on deployment descriptors, and the `web.xml` deployment descriptor in particular, see the Java Servlet Specification, downloadable from `http://java.sun.com/products/servlet/`.

Chapter

12

Editing the web.xml file

The WebApp DD Editor is active when the `web.xml` file is opened in the content pane. At the same time, the structure pane shows an outline of the contents of the file. Clicking the various nodes within the structure pane displays various pages of the editor. There are 15 main nodes, and some of these have child nodes. Here is a list of the main nodes:

- WebApp Deployment Descriptor
- Context Parameters
- Filters (Servlet 2.3 specification)
- Listeners (Servlet 2.3 specification)
- Servlets
- Tag Libraries
- MIME Types
- Error Pages
- Environment Entries
- EJB References
- Local EJB References (Servlet 2.3 specification)
- Resource Manager Connection Factory References
- Resource Environment References (Servlet 2.3 specification)
- Login
- Security

Each of these nodes contain tags you can edit in the WebApp DD Editor. The WebApp DD Editor covers all the `web.xml` deployment descriptor tags in the Servlet 2.3 specification. You can also edit the source of the `web.xml` file.

Caution If you enter comments in the `web.xml` file using the source editor, these will be removed if you subsequently open the file in the WebApp DD Editor.

The tags contained in each of the WebApp DD Editor's nodes are discussed in the following sections.

WebApp DD Editor context menu

Right-clicking any of the main nodes of the WebApp DD Editor displays a context menu which allows adding a new filter node, a new servlet node, or a new security constraint node. Note that adding a filter node is only available if your web server supports the Servlet 2.3 specification, since filter servlets are new to this version of the servlet specification.

Right-clicking an existing security constraint node displays a context menu which allows adding a new web resource collection node to that security constraint or another security constraint node. There must be at least one security constraint node before a web resource collection node may be added.

The context menu for an existing servlet, filter, or web resource collection node also contains options to rename or delete the current node. The context menu for an existing security constraint node contains the option to delete the current node and when editing a Servlet 2.3 `web.xml` file it also contains the option to rename the security constraint. Renaming or deleting any node cascades the change to all relevant parts of the `web.xml` file.

WebApp Deployment Descriptor page

The main page of the WebApp DD Editor contains basic identifying information for your WebApp. Here is a list of the information you can edit on this page:

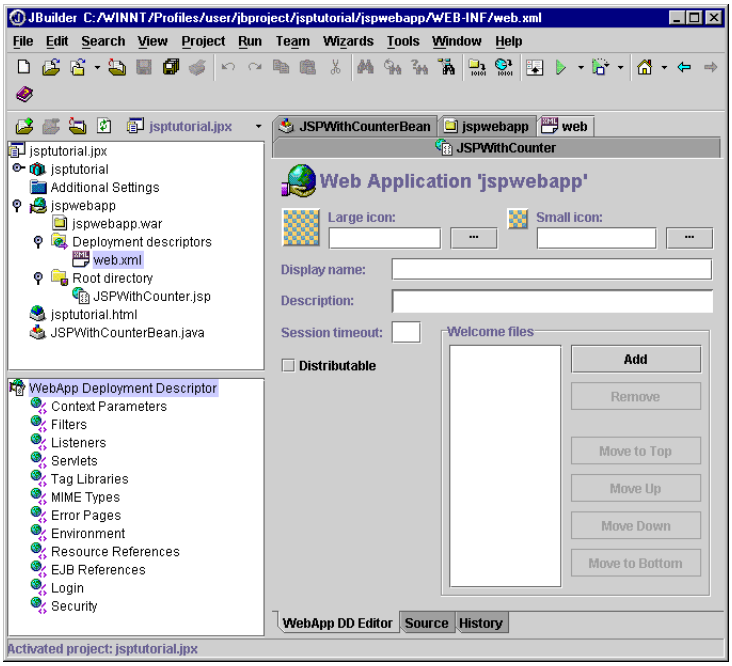
Table 12.1 WebApp Deployment Descriptor page of WebApp DD Editor

Item	Description
Large icon	Points to the location of a large icon for the WebApp (32 x 32 pixels), which must be contained within the WebApp's directory tree.
Small icon	Points to the location of a small icon for the WebApp (16 x 16 pixels), which must be contained within the WebApp's directory tree.

Table 12.1 WebApp Deployment Descriptor page of WebApp DD Editor (continued)

Item	Description
Display name	Name to be displayed for the WebApp. This can be used by servlet engine administration tools.
Description	Description of the WebApp. This can be used by servlet engine administration tools.
Session timeout	Whole number of minutes which are allowed to pass before a session times out.
Distributable	Whether the web application is deployable into a distributed (multi-VM) servlet container.
Welcome files	The file or files to be displayed when the URL points to a directory, for example: `index.html`

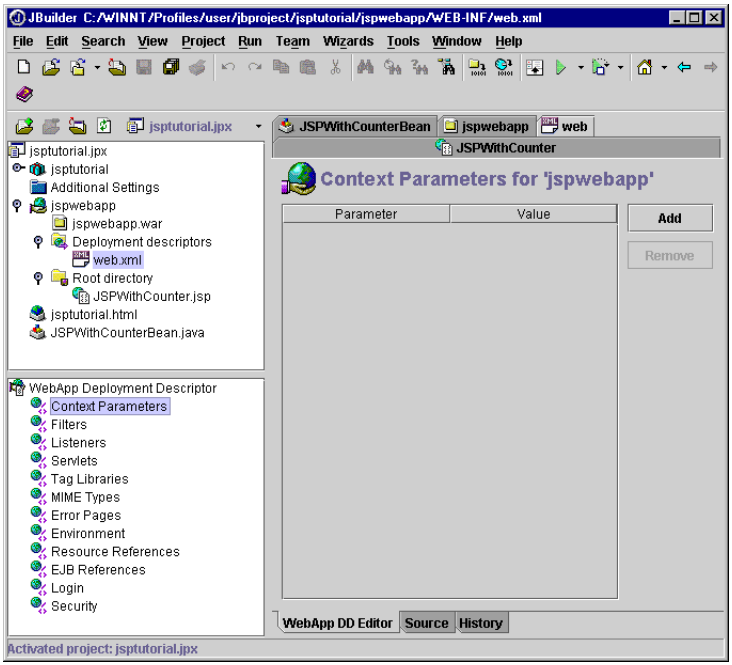
Figure 12.1 WebApp Deployment Descriptor page of WebApp DD Editor



Context Parameters page

The Context Parameters page contains a grid of initialization parameters for the entire WebApp's `ServletContext` and the values of those parameters. These parameters are accessed through the `ServletContext.getInitParameterNames` and `ServletContext.getInitParameter` methods.

Figure 12.2 Context Parameters page of WebApp DD Editor



Filters page

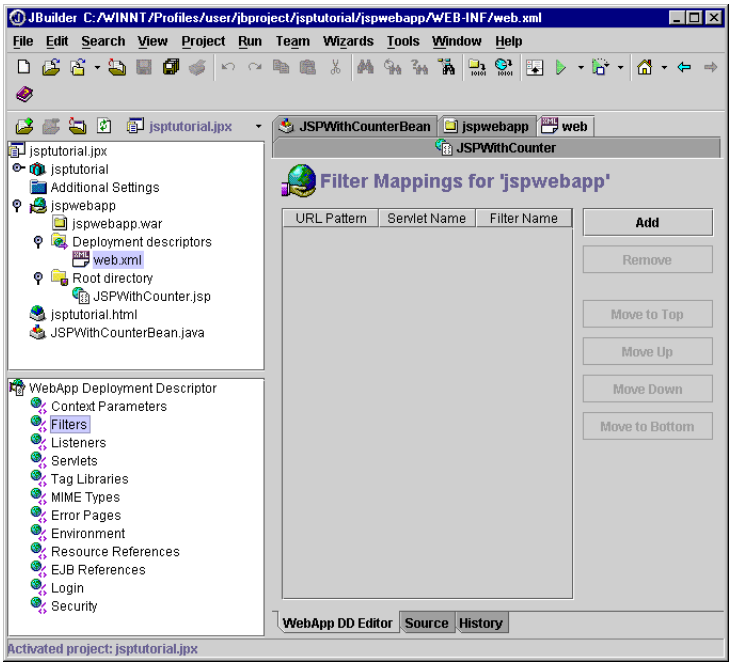
The Filters page will only be visible if your web server supports the Servlet 2.3 specification. This page contains a grid to map the filters (by `filter-name`) to either a URL pattern or a servlet name (but not both). The order of the filters is important because it is the order in which the filters will be applied. This page allows you to change the order in which the filters are applied. The following describes the information presented on the filters page:

Table 12.2 Filters page of WebApp DD Editor

Item	Description
URL Pattern	The `url-pattern` for the location of the filter. Either this or the `servlet-name` is required when deploying a filter servlet.
Servlet Name	The `servlet-name` which is used to map the filter. Either this or the `url-pattern` is required when deploying a filter servlet.
Filter Name	The `filter-name` which is used to map the filter. This is required when deploying a filter servlet.

If you use JBuilder's Servlet wizard to create a filter servlet, the wizard will add the required filter mapping for you.

Figure 12.3 Filters page of Webapp DD Editor



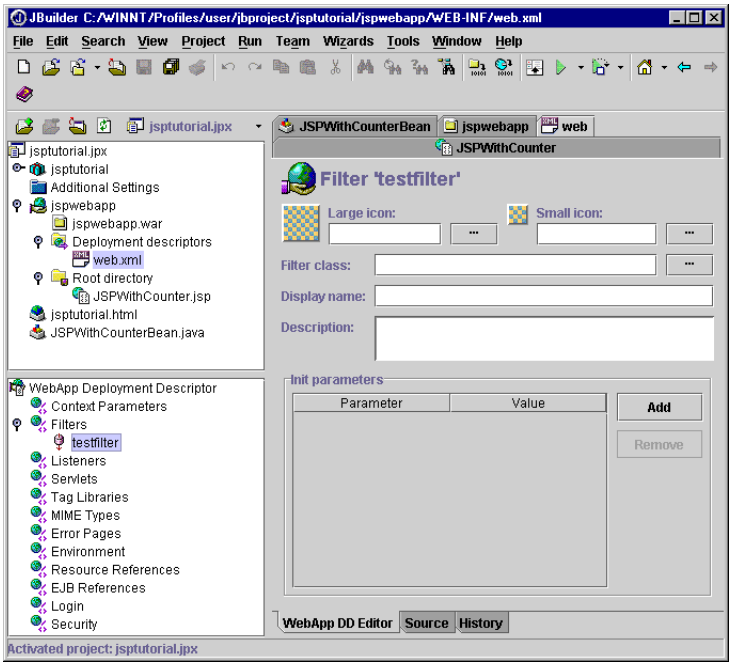
Each individual filter is listed in the structure pane as a separate child node of the Filters node. The filter's `filter-name` is displayed in the tree. You can rename or delete a filter by right-clicking the node for the individual filter and selecting Rename or Delete from the context menu. If you do rename or delete a filter, this change will be cascaded to all relevant parts of the deployment descriptor.

When an individual filter node is opened, the WebApp DD Editor displays a page for that specific filter. This page contains the following identifying information for the filter:

Table 12.3 Individual Filter page of WebApp DD Editor

Item	Description
Large icon	Points to the location of a large icon for the filter (32 x 32 pixels), which must be contained within the WebApp's directory tree.
Small icon	Points to the location of a small icon for the filter (16 x 16 pixels), which must be contained within the WebApp's directory tree.
Filter class	Fully qualified class name for the filter. This information is required when deploying a filter servlet.
Display name	Name to be displayed for the filter. This can be used by servlet engine administration tools.
Description	Description of the filter. This can be used by servlet engine administration tools.
Init parameters	Initialization parameters for the filter.

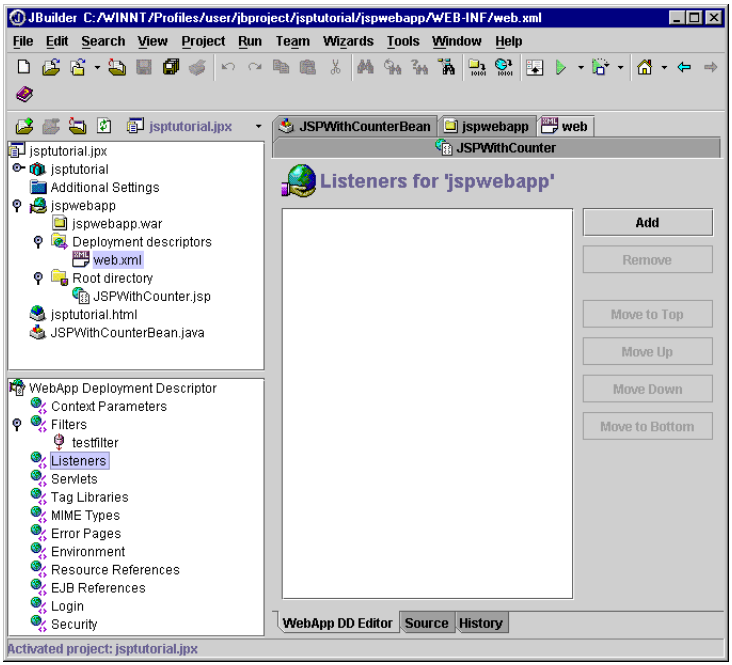
Figure 12.4 Individual filter node in Webapp DD Editor



Listeners page

The Listeners page will only be visible if your web server supports the Servlet 2.3 specification. The Listeners page has a list box of web application listener classes. The order of the listeners is important because it is the order in which the listeners will be applied. The Listeners page allows you to change the order in which the listeners are applied. The information on the Listeners page is required when deploying a listener servlet. If you use JBuilder's Servlet wizard to create a listener servlet, the servlet class will be added to the list for you.

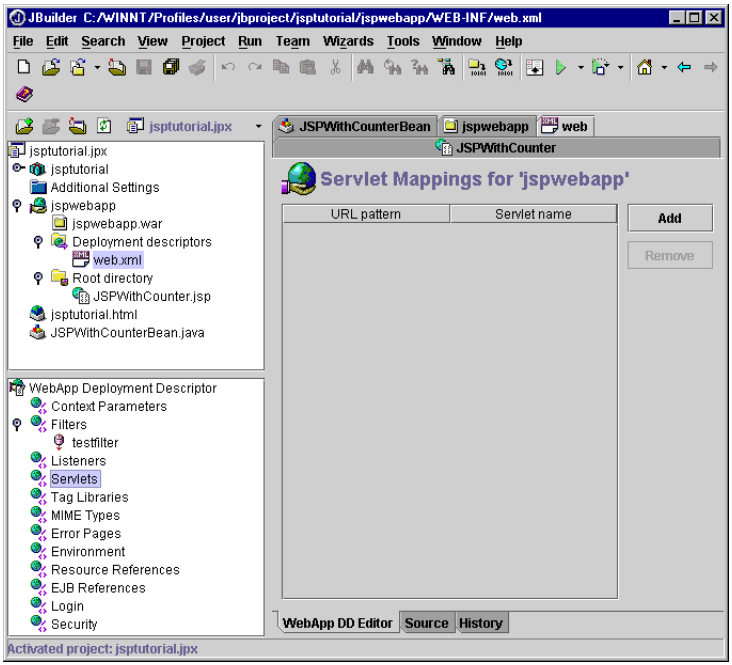
Figure 12.5 Listeners page of Webapp DD Editor



Servlets page

The Servlets page has a grid mapping URL patterns to a `servlet-name`. Note that a servlet can be mapped to more than one URL pattern. At least one servlet mapping is recommended for each servlet. If you use JBuilder's Servlet wizard to create a standard servlet, it will fill in the servlet mapping for you.

Figure 12.6 Servlets page of WebApp DD Editor



If any servlets are defined, you will find child nodes representing individual servlets underneath the Servlets page node in the structure pane. The servlet's `servlet-name` is displayed in the tree. You can rename or delete a servlet by right-clicking the node for the individual servlet and selecting Rename or Delete from the context menu. If you do rename or delete a servlet, this change will be cascaded to all relevant parts of the deployment descriptor. For example, removing a servlet also removes all its URL and filter mappings.

Keep in mind that you can also map URL patterns to JSPs. This means that an individual servlet node in the structure pane may represent a JSP or a servlet.

When an individual servlet node is opened, the WebApp DD Editor displays a page for that specific servlet. This page contains identifying information for the servlet:

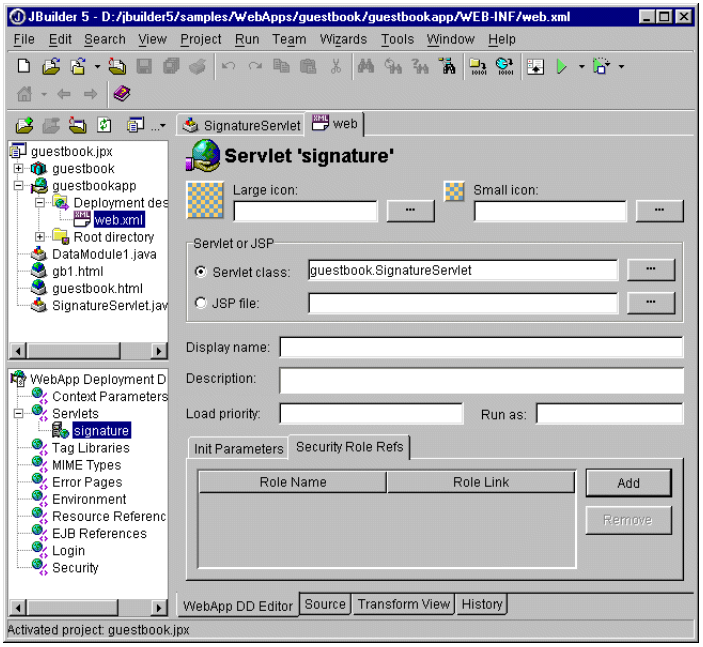
Table 12.4 Individual Servlet page of WebApp DD Editor

Item	Description
Large icon	Points to the location of a large icon for the servlet (32 x 32 pixels), which must be contained within the WebApp's directory tree.
Small icon	Points to the location of a small icon for the servlet (16 x 16 pixels), which must be contained within the WebApp's directory tree.

Table 12.4 Individual Servlet page of WebApp DD Editor (continued)

Item	Description
Servlet class	Select this radio button for a servlet. Enter the fully qualified class name for the servlet in the text field.
JSP file	Select this radio button for a JSP. Enter the path to the JSP file in the text field.
Display name	Name to be displayed for the servlet. This can be used by servlet engine administration tools.
Description	Description of the servlet. This can be used by servlet engine administration tools.
Load Priority	The `load-on-startup` priority for the servlet, in the form of a positive integer. Servlets with lower integers are loaded before those with higher integers.
Run As	Servlet 2.3 servlets can Run As another user, as one of the Role Names defined in the Security page.
Init Parameters	This page contains initialization parameters for the servlet.
Security Role Refs	This page contains a grid of security role renamings. You use this grid to map an alternate name hard-coded in the servlet to actual role names in the deployment descriptor.

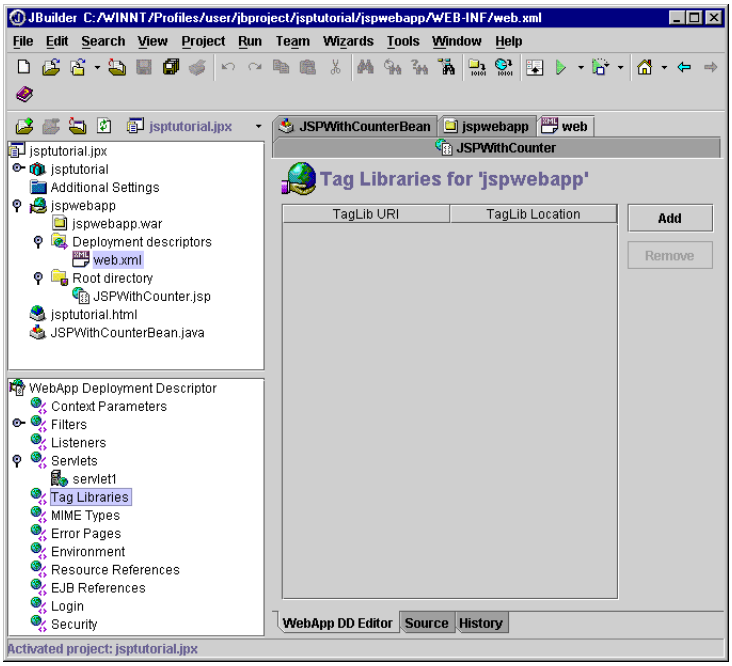
Figure 12.7 Individual servlet node in WebApp DD Editor



Tag Libraries page

The Tag Libraries page has a grid to map URIs used in a JSP with the actual locations of the Tag Library Definition (`.tld`) files. This information is required for deployment of a JSP which uses a tag library. If you use JBuilder's JSP wizard to create a JSP that uses one of the tag libraries recognized by the wizard, the tag library information is filled in for you.

Figure 12.8 Tag Libraries page in WebApp DD Editor

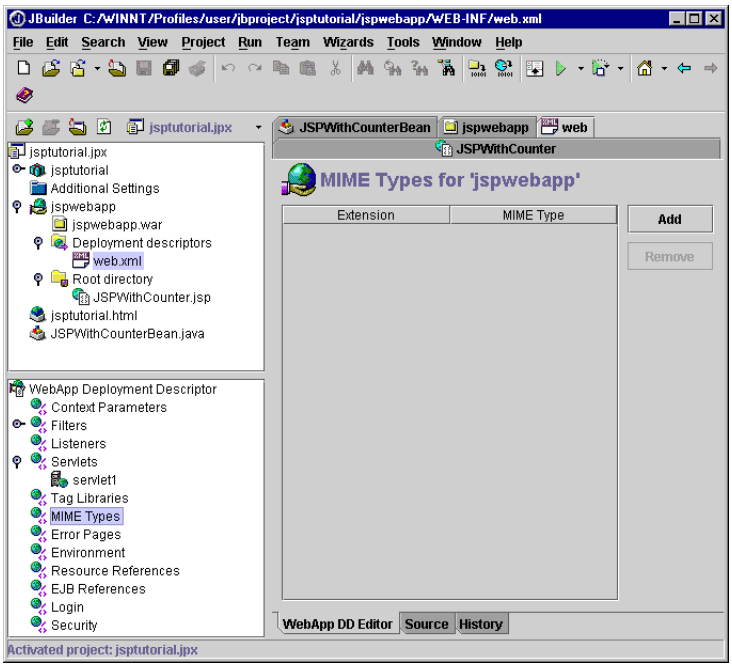


MIME Types page

The MIME Types page has a grid mapping extensions to type names. Use the MIME Types page to add custom types. Here are some sample MIME-types that are built in to many web servers:

- txt — text/plain
- html — text/html
- xml — text/xml
- pdf — application/pdf
- zip — application/x-zip-compressed
- jpeg — image/jpeg
- gif — image/gif

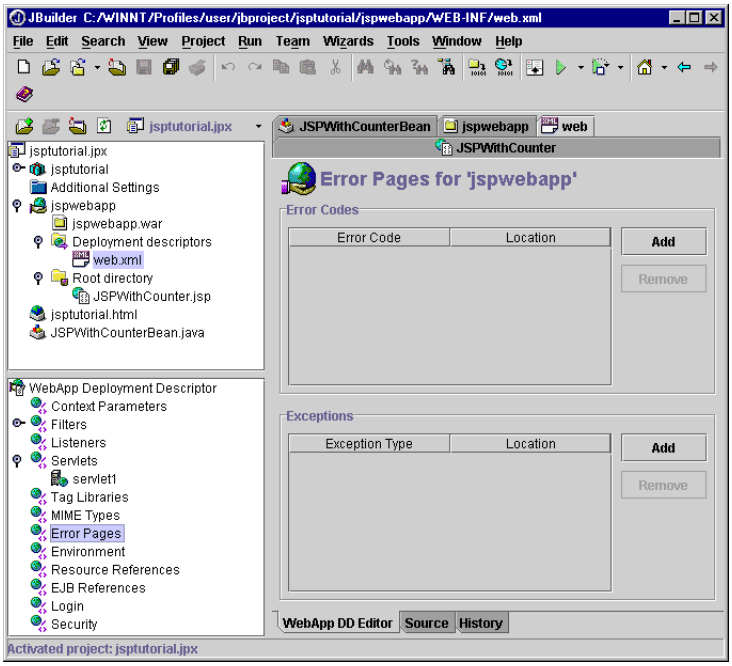
Figure 12.9 MIME Types page in WebApp DD Editor



Error Pages page

The Error Pages page has two grids, one for code numbers and one for exception class names, that are mapped to the locations of pages which should be displayed in the event of errors or exceptions.

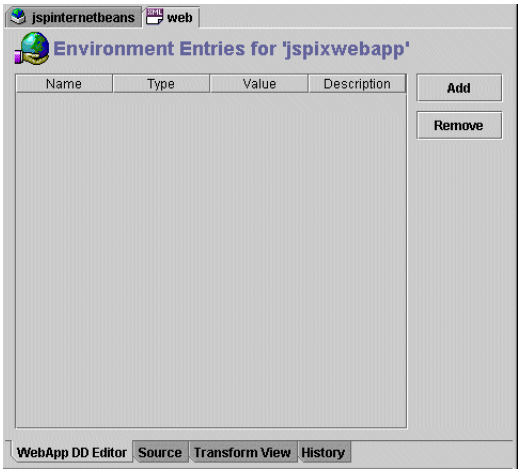
Figure 12.10 Error Pages page in WebApp DD Editor



Environment Entries page

The Environment Entries page has a grid of environment entry names, their values, and their types, plus a description text field for the currently selected entry.

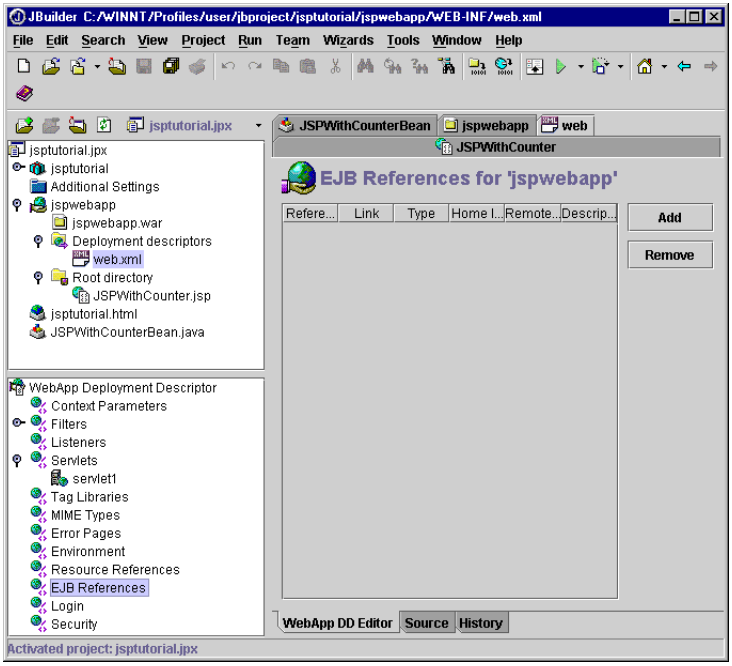
Figure 12.11 Environment page in WebApp DD Editor



EJB References page

The EJB references page has a grid of EJB names, their types, home and remote interfaces, optional `ejb-link`, and a description. This is similar to the EJB References page in the EJB DD Editor.

Figure 12.12 EJB References page in WebApp DD Editor



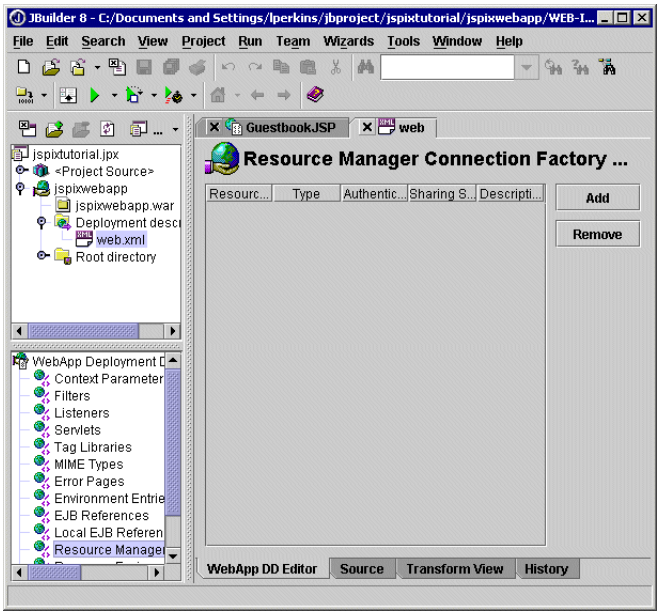
Local EJB References page

The Local EJB References page is present in the structure pane only when editing a Servlet 2.3 `web.xml` file. Its contents are similar to those on the EJB References page.

Resource Manager Connection Factory References page

The Resource Manager Connection Factory References page has a grid of resource names, their types, whether they use Container or Servlet authorization, and their sharing scope, plus a description text field for the currently selected entry.

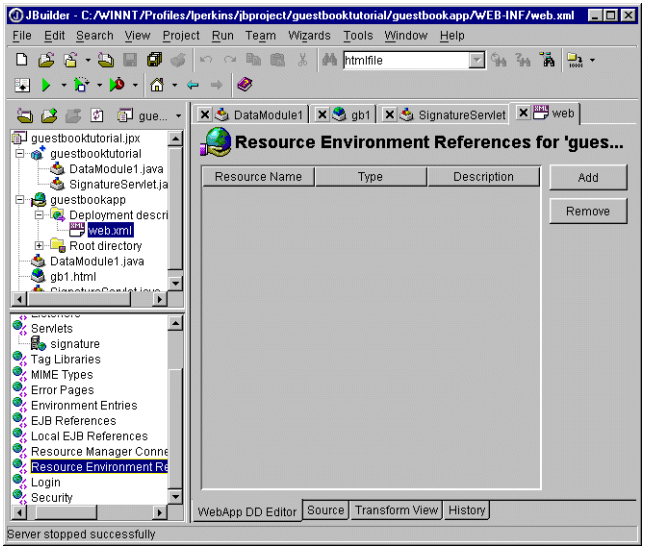
Figure 12.13 Resource Manager Connection Factory References page in WebApp DD Editor



Resource Environment References

The Resource Environment References page has a grid of resource names, their types, plus a description text field for the currently selected entry.

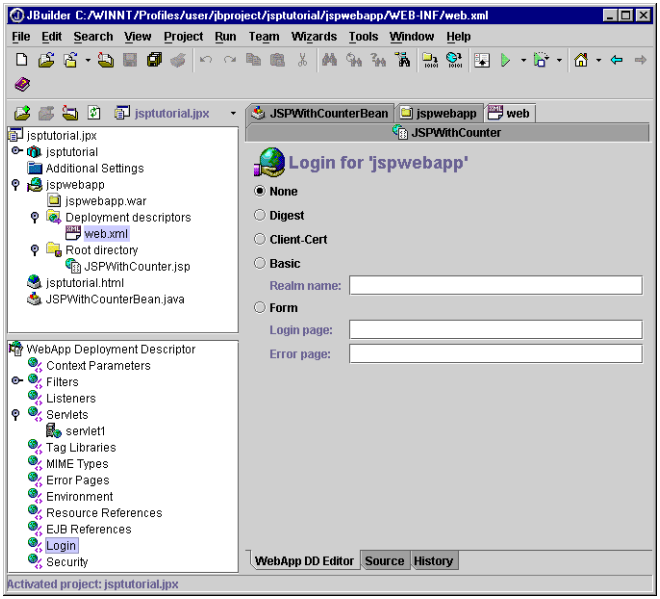
Figure 12.14 Resource Environment References page in WebApp DD Editor



Login page

The Login page displays a set of radio buttons for choosing the authentication method for the WebApp. The default is none. Other options are Digest, Client-Cert, Basic and Form. For Basic, you can specify a Realm name. For Form, you can specify a Login page and a login Error page.

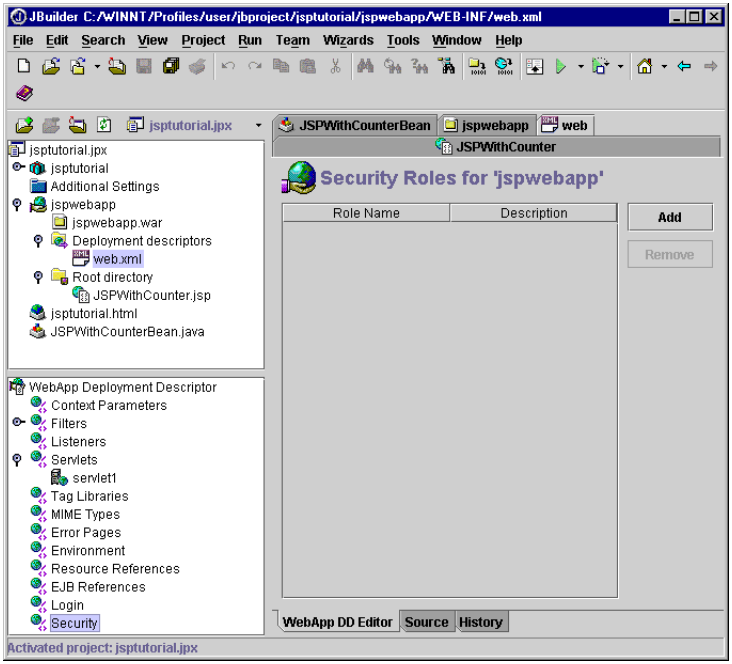
Figure 12.15 Login page in WebApp DD Editor



Security page

The Security page has a grid of role names and their descriptions.

Figure 12.16 Security page in WebApp DD Editor



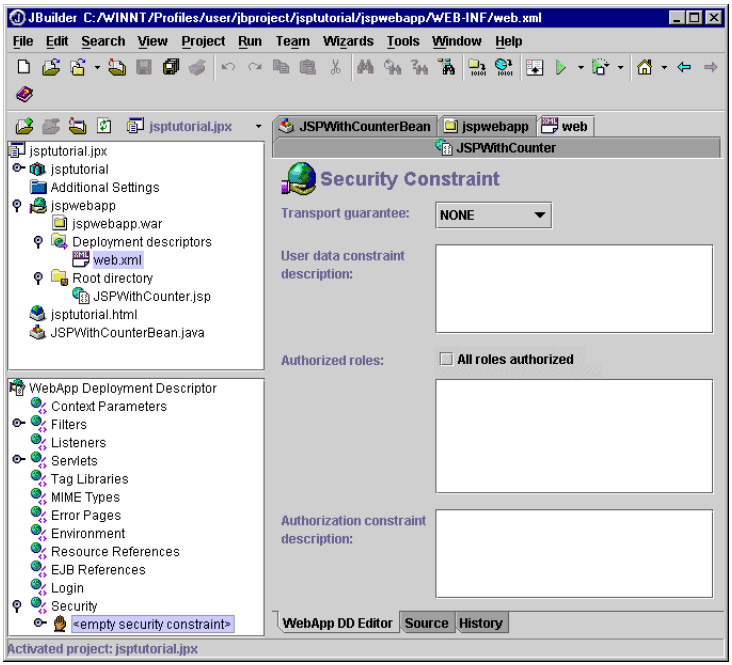
Security constraints

Each security constraint is listed as a separate child node of the Security node. You can delete a security constraint by right-clicking the node for the individual constraint and selecting Delete from the context menu. If you do delete a constraint, this change will be cascaded to all relevant parts of the deployment descriptor.

When an individual security constraint node is opened, the WebApp DD Editor displays the information for that security constraint: the transport guarantee, the role names that are authorized, and descriptions for both.

In addition to specifying authorization role names separately, the reserved role-name `*` indicates all roles. If this designation is used in addition to individual role names, the individual names are ignored. The security constraint page in the WebApp DD Editor includes an All Roles Authorized checkbox that is visible only when editing a Servlet 2.3 `web.xml` file.

Figure 12.17 Security constraint in WebApp DD Editor



Web resource collections

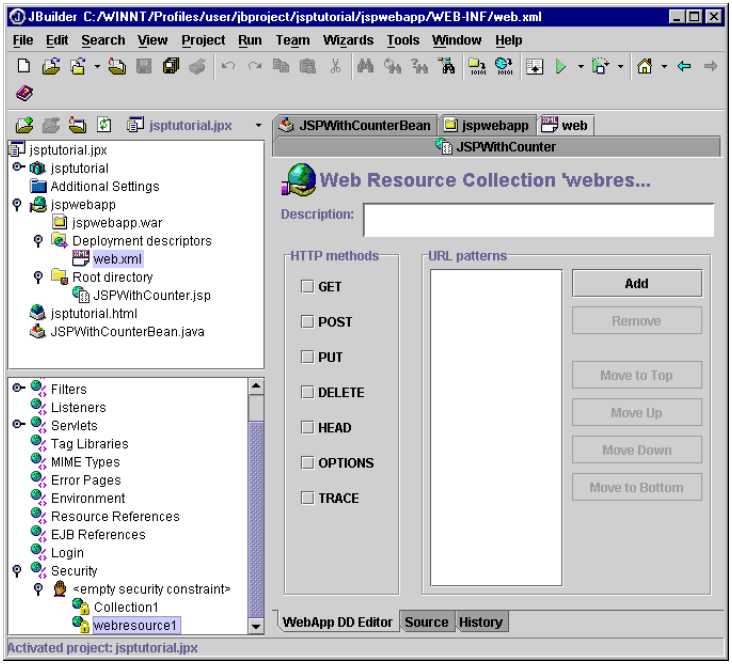
Each web resource collection's `web-resource-name` is listed as a separate child node of one or more applicable security constraints. You cannot add a web resource collection unless you already have at least one security constraint. Each security constraint must have at least one web resource collection. You can rename or delete a web resource collection by right-clicking the node for the individual collection and selecting Rename or Delete from the context menu. If you do rename or delete a web resource collection, this change will be cascaded to all relevant parts of the deployment descriptor. Deleting the last web resource collection for a constraint also deletes that constraint.

When a web resource collection node is opened, the WebApp DD Editor contains identifying information for the web resource collection:

Table 12.5 Web Resource Collection page of WebApp DD Editor

Item	Description
Description	Description of the web resource collection.
HTTP methods	A set of check boxes for the HTTP methods (such as `GET` and `POST`) that apply to the URL patterns. Selecting none of the check boxes is the same as selecting all of them.
URL patterns	A list box of the URL patterns for this web resource collection.

Figure 12.18 Web resource collection node in WebApp DD Editor



Chapter

13

Editing the struts-config.xml file

Web Development is a feature of JBuilder Enterprise

The `struts-config.xml` file is the deployment descriptor for a Struts-enabled web application. It must conform to a specific format, as described by the Document Type Definition (DTD) maintained at `http://jakarta.apache.org/struts/dtds/struts-config_1_0.dtd`. For detailed information on the most important elements of the DTD, see "The Struts Configuration File" topic in the *Struts Users Guide* at `http://jakarta.apache.org/struts/userGuide`.

You can use JBuilder's visual Struts Config Editor to edit and maintain the `struts-config.xml` file. When you open the `struts-config.xml` file in the JBuilder IDE, its contents display in the structure pane, and the AppBrowser displays the Struts Config Editor. Clicking a node in the structure pane displays the corresponding pages of the editor. There are four main pages:

- Data Sources
- Form Beans
- Global Forwards
- Action Mappings

Each of these pages configures elements and attributes of the deployment descriptor. The Struts Config Editor covers all the `struts-config.xml` deployment descriptor elements in the Struts 1.0 specification.

You can edit the `struts-config.xml` file in either the Struts Config Editor or the source editor. Changes made in the Struts Config Editor will be reflected in the source, and code changes made in the source will be reflected in the Struts Config Editor. If you enter comments in the `struts-config.xml` file using the source editor, these will be removed if you subsequently open the file in the Struts Config Editor.

Note If you use JBuilder's wizards to create the pieces of your Struts web application, you may not need to edit the `struts-config.xml` file.

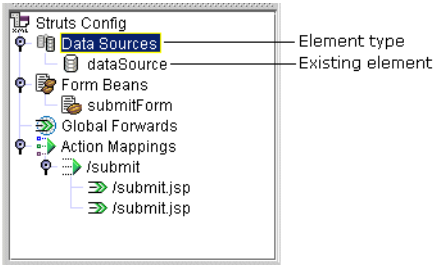
As you use JBuilder to create the pieces of your web application, the `web.xml` deployment descriptor file is automatically updated for Struts. For more information, see "Struts framework implementations in JBuilder" on page 8-14 .

Important JBuilder 8 supports Struts 1.0 and does not officially support the beta release of Struts 1.1. However, if you enable your `struts-config.xml` file for Struts 1.1 and download the Struts 1.1 JAR file to the `<jbuilder>/extras` folder, you will see support for Struts 1.1. The Struts Config Editor will display elements and attributes specific to 1.1.

Choosing a page of the Struts Config Editor

You use the structure pane to choose pages of the Struts Config Editor. To choose a page,

1. Double-click the `struts-config.xml` file in the project pane. The overview page of the editor is displayed in the content pane and its structure is displayed in the structure pane.
2. In the structure pane, click the node you want to edit. To go to an already existing element, expand the node in the structure pane, and click the element. The element and its attributes are displayed in the Struts Config Editor.



The Struts Config Editor context menu

When the Struts Config Editor is open in the content pane, you can right-click any of the nodes in the structure pane to display a context menu. Menu options vary slightly according to what's appropriate for the selected node. Commands allow adding new elements, deleting existing ones, and displaying wizards to create new classes.

Data Sources page

You use the Data Sources page to create the <data-sources> element in the struts-config.xml file. This element describes a set of JDBC 2.0 Standard Extension data source objects which are configured according to the nested <data-source> elements.

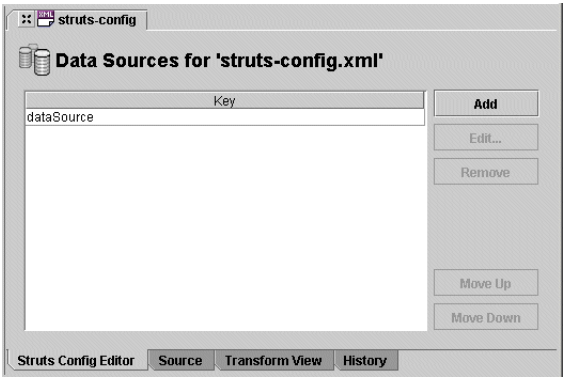
A sample entry in struts-config.xml looks like this:

```
<data-sources>
  <data-source>
     <set-property property="autoCommit" value="true"/>
     <set-property property="description" value="Sample JDataStore Configuration"/>
     <set-property property="maxCount" value="4"/>
     <set-property property="minCount" value="2"/>
     <set-property property="password" value="mypasssword"/>
     <set-property property="url"
         value="jdbc:borland:dslocal:
             C:/JBuilder/samples/WebApps/guestbook/guestbook.jds"/>
     <set-property property="user" value="myusername"/>
  </data-source>
</data-sources>
```

To display the Data Sources overview page of the Struts Config Editor, choose the Data Sources node in the structure pane. The Data Sources overview page displays existing <data_source> elements. You use this page to add and remove elements.

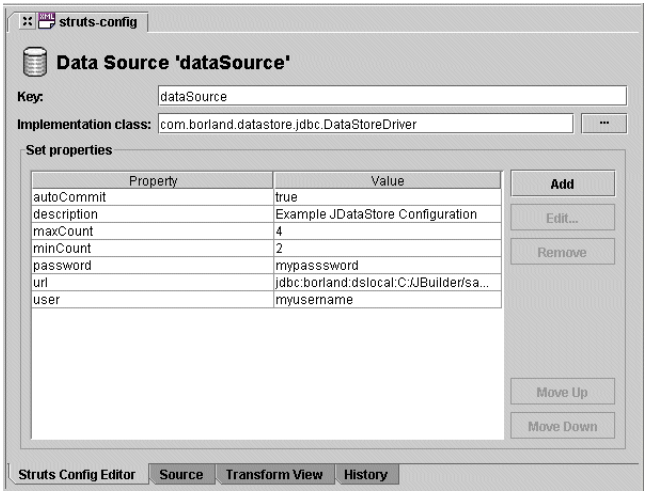
- To add a new <data-source> element, click the Add button. A new element is added with a default name of dataSource.
- To remove an element, select it and choose Remove.
- To edit the newly added element or an existing one, select it and choose Edit.

Figure 13.1 Data Sources overview page



You use the Data Sources attribute page to edit the attributes of a `<data-source>` element. The attribute page looks like this:

Figure 13.2 Data Sources attribute page



The following table explains the attributes on the page.

Table 13.1 Data Source attributes

Field Name	Attribute Name	Description
Key	`key`	Servlet context attribute key under which this data source will be stored. The default is the value specified by the string constant `Action.DATA_SOURCE_KEY`.
Implementation Class	`type`	The Java class name of the JDBC driver to be used.
Property	`property-name`, for example `autoCommit`, `loginTimeout`, `maxCount`, `minCount`, `password`, `readonly`, `url`, and `user`. Note that `password`, `readonly`, `url`, and `user` are required.	A property for this data source connection.
Value	The property value, for example `true`, `false`, a user name or password, or the URL of the data source.	The default value for the property.

Data Sources page context menu

To display the Data Sources page context menu, right-click the Data Sources node in the structure pane or expand the node and right-click an element. The available menu commands vary slightly according to what's appropriate for the selected node. The following table describes the context menu commands:

Table 13.2 Data Sources page context menu

Node	Menu command	Description
Data Sources node	Add DataSource	Displays the Data Source attribute page of the editor, where you add a new `<data-source>` element.
Data Sources element	Add DataSource	Adds a new `<data-source>` element.
Data Sources element	Delete	Deletes the selected element.

Configuring property attributes

You use the Add, Edit and Remove buttons to set and edit property attributes for the Data Source element.

- To add a property attribute, click the Add button. A property is automatically added with the default values of `property` and `value`.
- To edit a property attribute, double-click the Property column and enter a new value. You can also click the Edit button to display the Edit/Set Property dialog box, where you can change the property and value.
- To delete a property, select it and choose Delete.

Form Beans page

You use the Form Beans page to create the `<form-beans>` element of `struts-config.xml`. The `<form-beans>` element is the root of the set of form bean descriptors for your Struts web application. Nested `<form-bean>` elements describe a particular form bean, which is a JavaBean that implements the `org.apache.struts.action.ActionForm`.

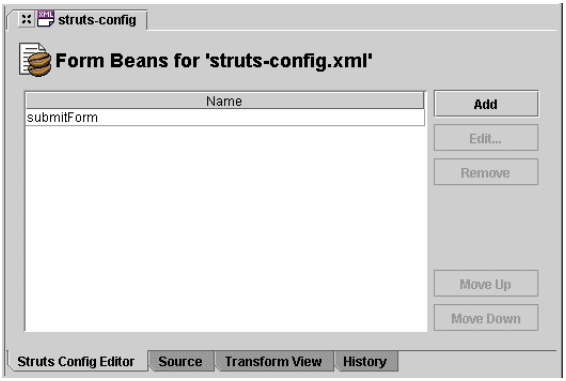
A sample entry in `struts-config.xml` looks like this:

```
<form-beans>
    <form-bean name="submitForm" type="myproject.SubmitForm" />
  </form-beans>
```

To display the Form Bean overview page of the Struts Config Editor, choose the Form Beans node in the structure pane. The Form Beans overview page displays existing `<form-bean>` elements. You use this page to add and remove elements.

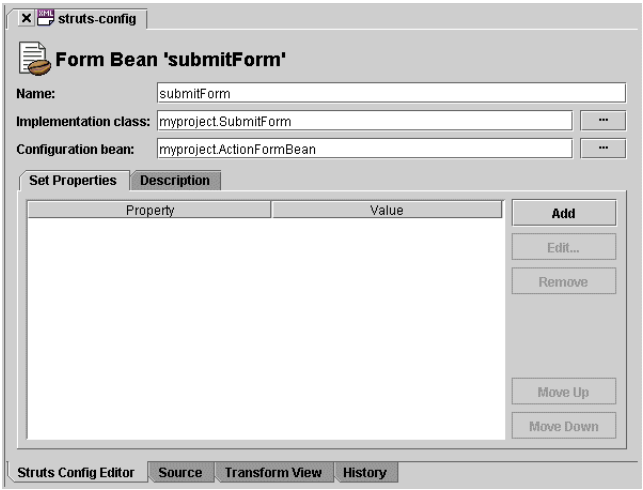
- To add a new `<form-bean>` element, click the Add button. A new element is added with a default name of `formBean`.
- To remove an element, select it and choose Remove.
- To edit the newly added element or an existing one, select it and choose Edit.

Figure 13.3 Form Beans overview page



You use the Form Bean attribute page to configure the attributes of a `<form-bean>` element. The attribute page looks like this:

Figure 13.4 Form Bean attribute page



The following table explains the attributes on the page.

Table 13.3 Form Bean attributes

Struts Config Editor Field Name	Attribute Name	Description
Name	`name`	Unique identifier of this bean, used to reference it in corresponding action mappings.
Implementation Class	`type`	Fully qualified Java class name of the implementation class to be used or generated.
Configuration Bean	`className`	Fully qualified Java class name of the `ActionFormBean` implementation class to use. Defaults to the value configured as the `formBean` initialization parameter to the Struts controller servlet.
Property (Set Properties tab)	`property-name`, for example `id`. A property is not required.	A property for this form bean.
Value (Set Properties tab)	The default property value, for example `ID` for the `id` property.	The default value.
Large Icon (Description tab)	`large-icon`	Specifies the location, relative to the Struts configuration file, of a resource containing a large (32x32 pixel) icon image, suitable for use in GUI tools.
Small Icon (Description tab)	`small-icon`	Specifies the location, relative to the Struts configuration file, of a resource containing a small (16x16 pixel) icon image, suitable for use in GUI tools.
Display Name (Description tab)	`display-name`	Contains a short (one line) description of the surrounding element, suitable for use in GUI tools.
Description (Description tab)	`description`	Contains descriptive (paragraph length) text about the surrounding element, suitable for use in GUI tools.

Form Beans page context menu

To display the Form Beans page context menu, right-click the Form Beans node in the structure pane or expand the node and right-click an element. The available menu commands vary slightly according to what's appropriate for the selected node. The following table describes the context menu commands:

Table 13.4 Form Beans page context menu

Node	Menu command	Description
Form Beans node	Add Form Bean	Displays the Form Bean attribute page, where you add a `<form-bean>` element.
Form Beans element	Add Form Bean	Adds a new `<form-bean>` element.
Form Beans element	ActionForm Wizard	Displays the ActionForm wizard, where you create a new `ActionForm` class.
Form Beans element	Delete	Deletes the selected element.

Configuring property attributes

You use the Add, Edit and Remove buttons to set and edit property attributes on the Set Properties tab.

- To add a property attribute, click the Add button. A property is automatically added with the default values of `property` and `value`.
- To edit a property attribute, double-click the Property or Value column and enter a new value. You can also click the Edit button to display the Edit/Set Property dialog box, where you can change the property and value.
- To delete a property, select it and choose Delete.

Global Forwards page

You use the Global Forwards page to create the `<global-forwards>` element of `struts-config.xml`. The `<global-forwards>` element configures the global mappings of logical names (used within the application) to mappable resources (identified by context-relative URI paths). The individual `<forward>` elements in the `<global-forwards>` element describe a mapping of a logical name (used within the application) to a mappable resource identified by a context-relative URI path.

Note A `<global-forwards>` with a particular name can be locally overridden by defining a `<forward>` of the same name within an `<action>` element.

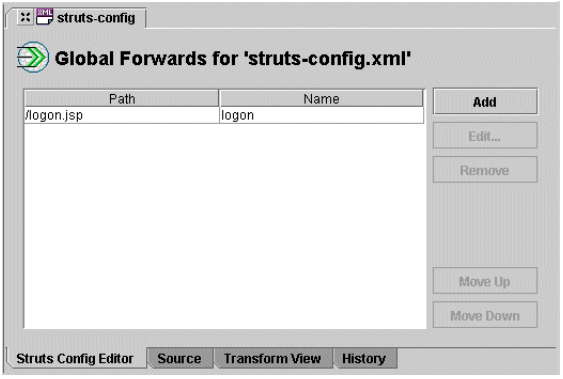
A sample entry in `struts-config.xml` looks like this:

```
<global-forwards>
  <forward name="logon" path="/logon.jsp" redirect="false" />
</global-forwards>
```

To display the Global Forwards overview page of the Struts Config Editor, choose the Global Forwards node in the structure pane. The Global Forwards overview page displays existing `<forward>` elements. You use this page to add and remove elements.

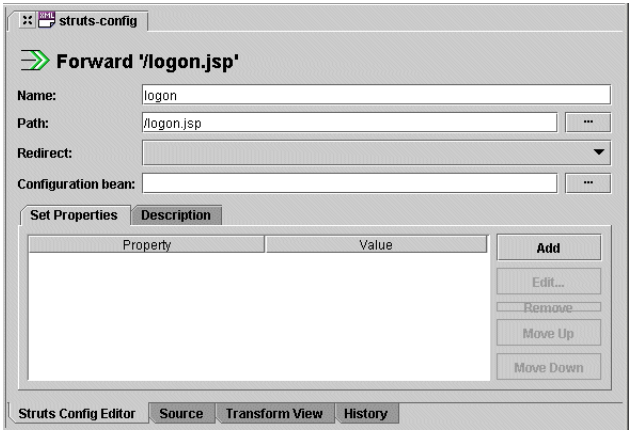
- To add a new `<forward>` element, click the Add button. A new element is added with a default name of `forward`.
- To remove an element, select it and choose Remove.
- To edit the newly added element or an existing one, select it and choose Edit.

Figure 13.5 Global Forwards overview page



You use the Forward attribute page to configure the attributes of an element. The attribute page looks like this:

Figure 13.6 Forward attribute page



The following table explains the attributes on the page.

Table 13.5 Forward attributes

Field Name	Attribute Name	Description
Name	`name`	Unique identifier of this `forward`, used to reference it in application action classes.
Path	`path`	The context-relative path of the mapped resource.
Redirect	`redirect`	Set to `true` if `sendRedirect()` should be used to forward to this resource, or `false` in order to use `RequestDispatcher.forward()` instead.
Configuration Bean	`className`	Fully qualified Java class name of the `ActionForward` implementation class to use. Defaults to the value configured as the `forward` initialization parameter to the Struts controller servlet.
Property (Set Properties tab)	`property-name`, for example `id`. A property is not required.	A property for this forward.
Value (Set Properties tab)	The default property value, for example `ID` for the `id` property.	The default value.
Large Icon (Description tab)	`large-icon`	Specifies the location, relative to the Struts configuration file, of a resource containing a large (32x32 pixel) icon image, suitable for use in GUI tools.
Small Icon (Description tab)	`small-icon`	Specifies the location, relative to the Struts configuration file, of a resource containing a small (16x16 pixel) icon image, suitable for use in GUI tools.
Display Name (Description tab)	`display-name`	Contains a short (one line) description of the surrounding element, suitable for use in GUI tools.
Description (Description tab)	`description`	Contains descriptive (paragraph length) text about the surrounding element, suitable for use in GUI tools.

Global Forwards page context menu

To display the Global Forwards page context menu, right-click the Global Forwards node in the structure pane or expand the node and right-click an element. The available menu commands vary slightly according to what's appropriate for the selected node. The following table describes the context menu commands:

Table 13.6 Global Forwards page context menu

Node	Menu command	Description
Global Forwards node	Add Forward	Displays the Forward attribute page, where you add a `<global-forward>` element.
Global Forwards element	Add Forward	Adds a new `<global-forward>` element.
Global Forwards element	Delete	Deletes the selected element.

Configuring property attributes

You use the Add, Edit and Remove buttons to set and edit property attributes on the Set Properties tab.

- To add a property attribute, click the Add button. A property is automatically added with the default values of `property` and `value`.
- To edit a property attribute, double-click the Property or Value column and enter a new value. You can also click the Edit button to display the Edit/Set Property dialog box, where you can change the property and value.
- To delete a property, select it and choose Delete.

Action Mappings page

You use the Action Mapping page to create the `<action-mappings>` element of `struts-config.xml`. The `<action-mappings>` element configures the mappings from submitted request paths to the corresponding `Action` classes that should be used to process these requests.

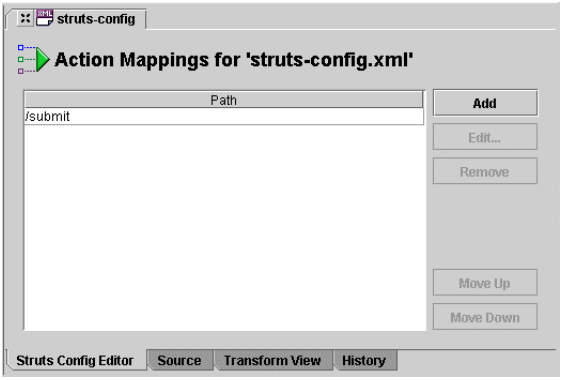
A sample entry in `struts-config.xml` looks like this:

```
<action-mappings>
  <action name="submitForm" type="myproject.SubmitAction" input=
     "/submit.jsp" scope="request" path="/submit">
     <forward name="success" path="/submit.jsp" />
     <forward name="failure" path="/submit.jsp" />
  </action>
</action-mappings>
```

To display the Action Mappings overview page of the Struts Config Editor, choose the Action Mappings node in the structure pane. The Action Mappings overview page displays existing `<action>` elements. You use this page to add and remove elements.

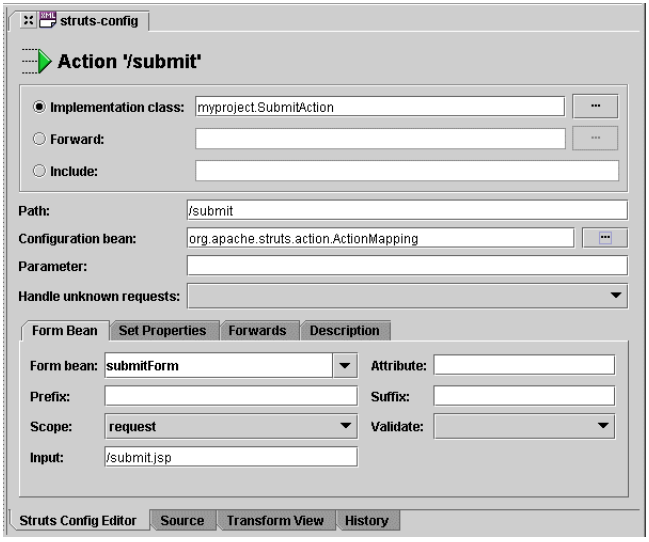
- To add a new `<action>` element, click the Add button. A new element is added with a default name of `/action`.
- To remove an element, select it and choose Remove.
- To edit the newly added element or an existing one, select it and choose Edit.

Figure 13.7 Action Mapping overview page



You use the Action attribute page to configure the attributes of an element. The attribute page looks like this:

Figure 13.8 Action attribute page



The following table explains the attributes on the page:

Table 13.7 Action attributes

Field Name	Attribute Name	Description
Implementation Class	`type`	Fully qualified Java class name of the `Action` class to be used to process requests for this mapping. If this field is selected, the Forward and Include fields are not available.
Forward	`forward`	Context-relative path of the servlet or JSP resource that will process this request, instead of instantiating and calling the `Action` class specified in the Implementation Class field. If this field is selected, the Implementation Class and Include fields are not available.
Include	`include`	Context-relative path of the servlet or JSP resource that will process this request, instead of instantiating and calling the `Action` class specified in the Implementation Class field. If this field is selected, the Implementation Class and Forward fields are not available.
Path	`path`	The context-relative path of the submitted request, starting with a forward slash ("/"), and without the filename extension if extension mapping is used.
Configuration Bean	`className`	Fully qualified Java class name of the `ActionMapping` implementation class to use. Defaults to the value configured as the `mapping` initialization parameter to the Struts controller servlet.
Parameter	`parameter`	General purpose configuration parameter that can be used to pass extra information to the `Action` selected by this mapping.
Handle Unknown Requests	`unknown`	Set to `true` if this action should be configured as the default for this application, to handle all requests not handled by another action. Only one action can be defined as a default within a single application.
Form Bean (Form Bean tab)	`name`	Name of the form bean, if any, that is associated with this action.
Prefix (Form Bean tab)	`prefix`	Prefix used to match request parameter names to form bean property names, if any. Optional if the Form Bean field is used, otherwise, not available.
Scope (Form Bean tab)	`scope`	Identifier of the scope (`request` or `session`) within which the form bean is accessed, if any. Optional if the Form Bean field is used, otherwise, not available.

Table 13.7 Action attributes (continued)

Field Name	Attribute Name	Description
Input (Form Bean tab)	`input`	Context-relative path of the input form to which control should be returned if a validation error is encountered. Required if the Form Bean field is used and the input bean returns validation errors. Optional if the Form Bean field is used and the input bean does not return validation errors. Not allowed if the Form Bean field is not used.
Attribute (Form Bean tab)	`attribute`	Name of the request-scope or session-scope attribute under which this form bean is accessed, if it is other than the name entered in the Form Bean field. Optional if the Form Bean field is used, otherwise, not available.
Suffix (Form Bean tab)	`suffix`	Suffix used to match request parameter names to form bean property names, if any. Optional if the Form Bean field is used, otherwise, not available.
Validate (Form Bean tab)	`validate`	Set to `true` if the `validate()` method of the form bean should be called prior to calling this action, or set to `false` if you do not want validation performed.
Property (Set Properties tab)	`property`	Specifies the name and value of an additional JavaBeans configuration property whose setter method will be called by the object that represents the surrounding element. This is especially useful when an extended implementation class (with additional properties) is configured on the `<global-forwards>` or `<action-mappings>` elements.
Value (Set Properties tab)	`value`	String representation of the value to which this property will be set, after suitable type conversion.
Path (Forwards tab)	`path`	The context-relative path of the mapped resource. Logically overrides a global `forward` of the same name.
Name (Forwards tab)	`name`	The unique identifier for this `forward`. Logically overrides a global `forward` of the same name.
Large Icon (Description tab)	`large-icon`	Specifies the location, relative to the Struts configuration file, of a resource containing a large (32x32 pixel) icon image, suitable for use in GUI tools.
Small Icon (Description tab)	`small-icon`	Specifies the location, relative to the Struts configuration file, of a resource containing a small (16x16 pixel) icon image, suitable for use in GUI tools.

Table 13.7 Action attributes (continued)

Field Name	Attribute Name	Description
Display Name (Description tab)	`display-name`	Contains a short (one line) description of the surrounding element, suitable for use in GUI tools.
Description (Description tab)	`description`	Contains descriptive (paragraph length) text about the surrounding element, suitable for use in GUI tools.

Action Mappings page context menu

To display the Action Mappings page context menu, right-click the Action Mappings node in the structure pane or expand the node and right-click an element. The available menu commands vary slightly according to what's appropriate for the selected node. The following table describes the context menu commands:

Table 13.8 Action Mappings page context menu

Node	Menu command	Description
Action Mappings node	Add Action	Displays the Action attribute page, where you add an `<action>` element.
Action Mappings node	Action Wizard	Displays the Action wizard, where you create a new `Action` class.
Action Mappings element	Add Action	Adds an `<action>` element.
Action Mappings element	Add Forward	Adds a `<global-forward>` element to the selected action.
Action Mappings element	Delete	Deletes the selected element.

Configuring property attributes

You use the Add, Edit and Remove buttons to set and edit property attributes on the Set Properties tab.

- To add a property attribute, click the Add button. A property is automatically added with the default values of `property` and `value`.
- To edit a property attribute, double-click the Property or Value column and enter a new value. You can also click the Edit button to display the Edit/Set Property dialog box, where you can change the property and value.
- To delete a property, select it and choose Delete.

Chapter

14

Working with applets

Applet development is a feature of all editions of JBuilder

If you have already tried to write applets and deploy them to a web site, you've probably discovered that, compared to applications, applets require dealing with additional issues to make them work. Deploying and testing applets outside the IDE is where difficulty usually begins, and if you have not handled some basic applet issues correctly, your applet will not work properly. To succeed, you need to know how applets work and how all the pieces fit together, especially when it comes to deploying and uploading them to an external web site.

See also

- "Tutorial: Building an applet" in *Introducing JBuilder*
- The Java tutorial on "Writing Applets" at `http://java.sun.com/docs/books/tutorial/applet/index.html`
- Sun's web site at `http://java.sun.com/applets/index.html`
- The Java FAQ at `http://www.afu.com/javafaq.html`
- Charlie Calvert's "Java Madness, Part II: Applets" at `http://homepages.borland.com/ccalvert/JavaCourse/index.htm`
- Rich Wilkman's Java Curmudgeon web site at `http://formlessvoid.com/jc/applets/`
- John Moore's "Applet design and deployment" at `http://www.microps.com/mps/p_appletdesign.html`

How do applets work?

Applets are Java programs that are stored on an Internet/intranet server. They are downloaded to multiple client platforms where they are run in a Java Virtual Machine (JVM) provided by the browser running on the client machine. This delivery and execution is done under the supervision of a security manager. A security manager can prevent applets from performing unwanted tasks, such as formatting the hard drive or opening connections to "untrusted" machines.

When the browser encounters a web page with an applet, it starts the JVM and provides it with the information located in the `<applet>` tag. The class loader inside the JVM looks to see what classes are needed for the applet. As part of the class loading process, the class files are run through a verifier which makes sure they are valid class files and not malevolent code. Once verified, the applet runs. This is, of course, a simplified view of the process.

This delivery mechanism is the primary value of applets. However, there are some issues unique to applet development that need to be addressed to ensure successful implementation.

The <applet> tag

Everything an applet needs to know at runtime is determined from the contents of the `<applet>` tag in the applet's HTML file. The tag attributes tell it what class to run, which (if any) archives it should search for classes, and the location of these archives and/or classes.

Unlike Java applications, which use an environment variable called `CLASSPATH` to search for classes, applets use only the `codebase` attribute in the `<applet>` tag to specify where to look for classes needed by the applet.

The key to running an applet is its ability to find the classes it needs.

Sample <applet> tag

Here is a simple `<applet>` tag that uses the most common attributes.

```
<applet
   codebase = "."
   archive = "test.jar"
   code = "test.Applet1.class"
   name = "TestApplet"
   width = 400
   height = 300
   vspace = 0
   hspace = 0
   >
   <param name = "param1"  value = "xyz">
</applet>
```

<applet> tag attributes

The following table describes the most common attributes and parameters used in the <applet> tag. Notice that some items are required.

Table 14.1 <applet> tag attributes

Item	Description
codebase	Allows you to specify a different location for your classes or archive files than the location of the HTML file containing the <applet> tag. This path is relative to the location of the HTML file and is often limited to subdirectories of the location of the HTML file. The codebase is similar to a CLASSPATH in that it tells the browser where to search for the classes. For example, if the applet classes are stored in a classes subdirectory, the codebase attribute is codebase = "classes". A value of "." specifies the same directory as the HTML file running the applet. **Important:** This attribute is required if the class or archive files are in a different directory than the applet's HTML file.
archive	Used to identify one or more archive files (ZIP, JAR, or CAB) that contain the classes needed for the applet. Archive files must be placed in the directory specified by codebase. You can have multiple archive files and list them with commas: archive="archive1.jar, archive2.jar". **Important:** This attribute is required if your applet is deployed to archive files. **Note:** Some of the older browsers only support ZIP archives.
code (required)	The fully qualified name of the applet class that contains the init() method. For example, if the class Applet1.class is part of a package called test, then code would be code="test.Applet1.class".
name (optional)	A string that describes your applet. This string shows up in the status bar of the browser.
width/height (required)	Defines the size in pixels allocated to the applet in the browser. This information is also passed to the applet's layout manager.
hspace/vspace (optional)	Represents the horizontal or vertical padding in pixels around the applet. This is useful if you have text on the web page that wraps around the applet or if you have more than one applet in the page.
param (optional)	Allows you to specify parameters that can be read by the <applet> tag. These are similar to the argument list used by main() in applications. Parameters have two parts, name and value, both quoted strings. name is used inside the applet when you want to request a value. You must handle any conversion from String to some other data type in your applet code. Be sure to match the case between the HTML parameter and the applet code that requests it.

Common mistakes in the <applet> tag

Most problems with the `<applet>` tag are due to the following errors:

- A missing `</applet>` tag

  If you receive an error message that says there's no `</applet>` tag on the HTML page, check if there is a closing tag.

- Forgetting that Java is case-sensitive

  The case is extremely important. Typically, class names use mixed case, and directories and archives are all lowercase. The class, archive, and directory names in the `<applet>` tag must exactly match the case of those on the web server. If the case is not exactly the same, you'll see "can't find xyz.class" errors.

  If your class is in package `com.mypackage`, any directories created to support that class must be created with and referenced in lowercase.

  Moving files between Windows 95/98 and Windows NT and the Internet introduces increased chances for case-sensitivity errors.

  Once you've posted your files to the Web, check to make sure that files, directories, and archives have the same uppercase and lowercase values as your local machine. Also, if you have archived all your class files, make sure that the archiving tool has preserved the case.

- Using the "short name" of the class in the `code` attribute

  You need to use the fully qualified name because that is the actual name of the class. If the class name is `Applet1.class` and is in package `test`, then you must reference it correctly in the `code` attribute as `test.Applet1.class`.

- An incorrect `codebase` value

  The `codebase` is resolved against the directory containing the HTML file, effectively forming a classpath entry. The `codebase` value is not a URL or some other type of reference. In the simple applet example in the previous section, `codebase = "."` was used. This specifies that files needed by the applet will be found in either the same directory as the HTML file, or, if no archive is specified, the files will be found in package appropriate subdirectories relative to the one containing the HTML file.

- A missing `archive` attribute

  If your applet is deployed in one or more JAR or ZIP files, you must add the `archive` attribute to the HTML file.

- Missing quotes around the values used for `code`, `codebase`, `archive`, `name`, and so on.

  For improved XHTML compatibility, it is recommended to also use quotes around numbers.

Browser issues

One of the primary tasks in getting applets to work at runtime is solving the various issues with the browsers that host the applets. Browsers vary in the level of JDK support and each browser approaches Java implementation differently. The most frequent issues you'll encounter concerning browsers include Java support, getting the preferred browser to the end user, multiple browser support, and differences in Java implementation.

Java support

A common problem with applets is a JDK version mismatch between your applet and the browser running it. There are many users who have not upgraded or updated their browsers. These browsers may not support the newer JDKs from Sun. Swing, introduced in JDK 1.1.7, may not be supported by some browsers. The most common error will be `NoClassDefFoundError` on some class in the `java.*` packages because the ClassLoader determined it needed a Java class or method that the browser JDK does not support.

JDK 1.2, 1.3, and 1.4 are still not fully supported in all browsers. While it is true that recent versions of the browsers support Java, you should know what specific version of Java is supported in the browsers that will run your applet. This is important so you can do whatever is necessary to create a match between the version of the JDK used in your applet and the JDK supported by the browsers.

To find out what JDK version the browser supports, check the Java Console in the browser or check the browser's web site.

To open the Java Console, do one of the following:

- Select Communicator | Tools | Java Console in older versions of Netscape or Tasks | Tools | Java Console in more recent versions.
- Select View | Java Console in Internet Explorer.

The Java™ Plug-in from Sun easily solves the problem of JDK mismatch by permanently downloading the same version of the JDK to all the end-users' machines.

Getting the preferred browser to the end user

Initially, your clients must install a minimum version of a browser. If they already have their favorite browser installed, you will need to convince them that any inconvenience caused by getting updates is offset by the value of your applet. Each browser upgrade requires downloading the browser update and/or installing a patch.

Supporting multiple browsers

If you've chosen to support more than one browser with your applet, you will have to support them in the field as well. This means that if users have browser-related trouble using your applet, they are going to call you for support.

Differences in Java implementation

There are guidelines and specifications for what browsers must provide but implementation of these and any extensions is left to the browser manufacturer.

One difference among browsers is in their implementation of the security manager and security levels. What is allowed in one browser at "medium" security may not be allowed in the other. Adjustment of security is done on the client side, and each browser does it differently. For example, the mechanism for relaxing some security for an applet is signing. Signing an applet and getting the browser to accept that signature allows you more flexibility in your applet. The mechanism for this is built into Java with the **Javakey** utility. Unfortunately, browsers aren't required to use this mechanism. In fact, some use their own proprietary mechanisms for signing which only work in their browser.

Another difference is that individual browser manufacturers make some adjustments to core Java functionality, including leaving some of it out. This can result in unexpected behavior at runtime. For example, you could have paint calls that do not happen, leaving the applet in an odd state, or even invisible. You can try to find a way to code around specific cases, but it is not always possible.

Also, there are often variations in Java support on different platforms from within the same browser. For example, a multi-threaded applet may run on one platform, but because of problems in the threading model on a different platform, threads may run sequentially on a different platform. Often, browsers are not updated across platforms at the same time, and a browser update for a less popular platform may be released later.

These differences increase the amount of time you have to spend testing and debugging in a particular browser.

Solutions to browser issues

This section offers solutions for problems you might encounter when running your applet in a browser.

- Use the Java Plug-in

  Most of the browser issues mentioned above can be solved by using the Java Plug-in, found at `http://java.sun.com/products/plugin/`. This plug-in provides up-to-date JDK archives, which is beneficial if your applet uses Swing (JFC).

  You do have to modify your HTML files for the applet to use the plug-in. Sun provides a utility, the HTML Converter, that does the conversion quickly and easily. For more information on the HTML Converter, see `http://java.sun.com/j2se/1.4/docs/guide/plugin/developer_guide/html_converter.html`.

  Using the Java Plug-in is the only way to get the same JDK into the different browsers. You also typically get a JDK that is more current than the browser's JDK. There are some extensions that browser manufacturers have added that may not work the same with the plug-in.

  The first time an end-user encounters a Java Plug-in enabled page, he or she will be prompted to install the plug-in. Once the plug-in is installed, the client machine will have an official VM from Sun and the latest JDK (including Swing) installed locally. This will only be used in "plug-in" aware pages, but it does cut down on the overhead of delivering the latest JDK technology to the browser.

Important There are several plug-in versions: JDK 1.1.x, 1.2x, 1.3x and 1.4. You must use the version that matches the JDK used by your applet. In addition, the HTML Converter version must match the plug-in version. Also note that you can have only one version of the plug-in on the client machine.

- Use the same version of the JDK supported by the browsers

  You can avoid many problems by writing applets using the same JDK that the browsers support. JBuilder SE and Enterprise allow you to switch the JDK version you are using for development. Although JBuilder Personal does not support JDK switching, you can edit the current JDK. See Tools | Configure JDKs and "Editing the JDK" in the "Creating and managing projects" chapter of *Building Applications with JBuilder*.

- Install the non-core Java classes on the clients

  Each browser has a directory that Java uses for `CLASSPATH`-based searches, so if you store the large archives that your applet uses locally, your client won't have to download them. This is usually only possible in an intranet environment or in Internet situations where you have a controlled client set (such as in a subscription-based service). This does, however, make future code updates more difficult.

- Select one browser

  Make your users use only one browser to minimize the need for special code and maintenance. This is usually only possible in intranet situations controlled by IS policy.

- Use Java Web Start

  Java Web Start is a deployment technology from Sun Microsystems. It allows you to launch Java applets (and applications) from a link on a web page in your web browser. Java Web Start solves the JDK version mismatch problem. As long as the browser has the Java Web Start plug-in, it can automatically download the appropriate JDK to run the applet.

  For more information on Web Start, see Chapter 15, "Launching your web application with Java Web Start," and visit the Java Web Start page at `http://java.sun.com/products/javawebstart/`.

Additional tips for making applets work

Here are a few more tips for avoiding problems when developing and deploying applets:

- Place files in the correct location

  A common problem with a deployed applet is placing files in the wrong location (for example, "can't find xyz.class"). Everything is relative to the HTML file that launches the applet. Without a `CLASSPATH` to help find applet classes, the applet relies on the `codebase` and `code` attributes in the HTML file to locate the classes it needs to run. Therefore, before beginning deployment, it is important to have the files in the correct location.

  You'll find the deployment process easier if you make your project working environment reflect the reality of a deployed applet, bringing your development a little closer to instant deployment. For example, when the `codebase` value is ".", the browser looks for the applet's class and archive files in the same directory as the HTML file. Also, if the class is in a package, the browser constructs a directory path underneath the HTML file's directory according to the package structure. If you use JBuilder's Applet wizard, this file structure is

created automatically by the wizard. Once you have your applet built and running in this situation, you can archive everything into one JAR or ZIP file. Then, test your applet outside JBuilder in the browsers.

Tip You can use a copy of the actual HTML page that is on your site in the local project, instead of a simpler test one. That makes the external testing a little more realistic. When you're satisfied, upload the entire directory to your web site.

- Use packages

  You should always use packages to organize similar classes when coding in Java. This makes them easier to find and makes deployment much simpler. Packages also help avoid conflicts with class names, because the package name is added before the class name in Java.

- Use the latest browser

  Know the browsers on which your applet will run. If you are using current tools like JBuilder, you are likely to be generating JDK 1.3.x or later code and will need the latest browser(s) to run your applets. Make sure you provide the latest patch or plug-in to match the browser's JDK version with the applet's. See Java Plug-in found at `http://java.sun.com/products/plugin/`.

- Match case

  Remember that Java is case-sensitive. Check that all case in class, package, archive, and directory names in the `<applet>` tag exactly match the names on the server.

- Use archive files

  Archives simplify deployment, because you only have to upload the HTML file and your archive file(s). They also speed up the downloading process to the client; one compressed file is downloaded rather than each individual, uncompressed class file. Always check the directory structure and the case of file names in the archive file for accuracy. See "Deploying applets" on page 14-12 and "Deploying applets in JBuilder" on page 14-23 for more information on creating archive files.

Important Older versions of some browsers do not support multiple archives and support uncompressed ZIP files only.

- Deploy everything

  Unless you are writing something that uses only core Java classes, you have to deploy all of the class files needed by your applet to the Web. Of course, you still need to deploy everything to the right location. See "Deploying applets" on page 14-12 for more information.

- Retest the applet after deploying it to the server

  It is not sufficient to test the applet locally. You must also test it in multiple browsers after deploying it to the server. This is critical to ensure that all the classes you need for the applet are deployed properly to the server, that the `<applet>` tag and server deployment match, and to find any browser-specific problems.

  For more information, see "Testing applets" on page 14-13.

Security and the security manager

An important concern about running programs over the Internet is security. Users are extremely cautious about downloading and running unknown programs on their machines without a guarantee of security to prevent things like programs deleting files or uploading personal information from their computers.

Security is advantageous for the developer because without it, many end users would not allow applets to run at all. However, it also has several disadvantages for the developer, especially if the developer is unaware of the restrictions when the project is started. Many activities that you take for granted in an application will be denied in an applet.

The sandbox

Applets address this concern by running all untrusted code in a safe environment called the *sandbox*. While an applet is in the sandbox, it is under the watchful eye of a *security manager*. The security manager protects the client machine from harmful programs that might delete files, read secure information, or do anything that is a violation of the security model being used. The restrictions placed by the security manager, in turn, limit what an applet can do.

Understanding these limitations before writing your applets can save time and money. Applets, like any tool, accomplish certain tasks very well. But, trying to use an applet in the wrong context only leads to problems.

Applet restrictions

By default, all applets are "untrusted" and are blocked from specific activities, such as

- Reading and writing from the local disk

  You can't read files, check for their existence, write files, rename files, and so on.

- Connecting to another machine other than the one from which the applet came

  This makes it difficult to provide data from a database that lives on a different machine than the web server.

There are other activities taken for granted by application developers, such as running local system programs to save time, which are not allowed in applets. Any good Java book will list the restrictions placed on applets.

If you find that you are trying to bypass the security manager at any point in your planning or development, consider using an application instead. Sun has done a very good job of defining their security model and identifying the key bits of information, or data types, on which entire sets of functionality depend. As a result, it is very difficult to trick the security model.

Solutions to security problems

To manage security problems, you can

- Sign the applet

  This may allow you to relax some of the restrictions placed by the security manager. It has a few disadvantages, one of which is that there is no standard signing mechanism that works in all browsers. Check the web site for your particular browser for more information on signing applet archive files. See "Code signing for Java applets" at `http://www.suitable.com/Doc_CodeSigning.shtml` for more information on signing applets.

- Have your users change the security model on their side

  The browsers have settings that can be used to adjust the security model. These settings range from very simple choices between "high," "medium," and "low" to fairly flexible schemes. Unfortunately, these settings are global for all applets. While the user may trust your applet not to do harm under relaxed security, it is often difficult to convince them to do so for all applets that they may encounter.

- Use different technologies to get around the security manager

  For example, if you simply need to write back to the server, write server-side Java applications, such as servlets, that your applet talks to. This server-side code doesn't have the limitations of the applet and can be fairly powerful. Server-side Java applications require access to the physical server or a very flexible and understanding ISP (Internet Service Provider).

- Recognize tasks which are not suited for applets

  Gathering data from forms on the Web and collecting that data is probably better suited for servlets or JavaServer Pages (JSPs). Complex operations, such as using databases, require additional software on the server such as JDBC. It's a more complex solution, but without it, "normal" database actions, such as posting new data, are not possible in applets. If you are writing a servlet or JSP which uses JDBC to connect to a database, you should consider InternetBeans Express to make the servlet or JSP data-aware. See the remaining chapters in this book for more information on these topics.

- Consider writing an application, servlet, or JSP rather than an applet

  The advantages of Java applications, including full access to the Java language and no security issues, may far outweigh the benefits of applets in some situations. Servlets and JSPs also have advantages over applets. They don't rely on the client browser JDK, because Java is performed on the server side. Also, servlets and JSPs are often faster because they don't require a download to the client browser like applets do.

Using third-party libraries

Third-party libraries are a great benefit for the developer. Often, these libraries contain a variety of components that save time and money. However, it is important to weigh the advantages of the component libraries to the disadvantages to your end user.

With applets, all referenced classes have to be sent across the Internet to the client machine. If the third-party library has a large interwoven architecture, it can substantially increase the applet's file size. You should also be aware of the redistribution license for the library. Some libraries require you to redistribute the library in its entirety (not just the pieces you use).

Remember The larger the applet, the longer it takes to download and run.

Deploying applets

Deployment is the process of gathering the applet's classes and their dependents in the correct hierarchy for deployment to a server. JBuilder SE and Enterprise provide the Archive Builder for deployment within JBuilder's IDE after applet development. Sun provides the **jar** tool in the JDK.

There are several things you can do to make deployment easier:

- Develop your applet in a local environment that is the mirror image of your web site directory structure.
- Use archive files.
- Use the Archive Builder in JBuilder SE and Enterprise or the JDK **jar** tool to create the archive files.

Important If you are writing JDK 1.1.x applets with Swing components, which is not recommended due to browser issues, you also need to download and deliver the JFC (Swing) classes, `swingall.jar`, as they were not delivered as part of the JDK.

See also

- "Deploying applets in JBuilder" on page 14-23
- "Deploying Java programs" in *Building Applications with JBuilder*
- **jar** tool in the JDK documentation
- "Java Web Start and JBuilder" on page 15-4

Testing applets

The main purpose of an IDE is to enable you to develop and test your code locally. When you run an application or an applet in JBuilder, if the program needs a particular class, JBuilder goes to great lengths to find that class. This actually allows you to focus on developing your application or applet and making sure it works.

However, it does not give you accurate runtime behavior for applets. To properly test an applet, you need to test it with the applet deployed to the web server in its actual running environment. This is the only way to make sure all the classes the applet needs are available on the server running it.

When you run your applet in a browser, the applet needs to have enough information inside the `<applet>` tag of the HTML file to allow the applet loader to find all the classes, images, and other assorted files your applet needs. This environment is the entire "real world" of your applet and is the only place the applet gets the information it needs for finding the files.

For the first tests of your deployed applet, consider using the JDK **appletviewer** test "browser." This command-line tool included in the JDK provides useful displays of error messages and stack traces. As demonstrated in the following steps, remove the `CLASSPATH` in the testing session.

Basic testing steps

The basic steps for testing your applet are as follows:

1. Deploy the necessary applet files to the server.

   Check the following:

   - The files are transferred to the web server in binary mode when using an FTP client.
   - The files are in the proper locations *relative to the HTML file* that launches the applet and the `codebase` value is correct.
   - The case of the names on the server match the original names.
   - The `<applet>` tag contains the `archive` attribute with the archive file name(s) if the applet is deployed to a JAR or ZIP file.
   - The directory structure matches your package structure.

   Deployment is discussed in depth in "Deploying Java programs" in *Building Applications with JBuilder*.

2. Open a command-line window.
3. Remove any `CLASSPATH` you may have set for that session. **appletviewer** knows where to find the core Java files.
4. Change to the directory containing the HTML file and the JAR file (or class files).
5. Run **appletviewer**.

   ```
   <jbuilder>/<jdk>/bin/appletviewer TestApplet.html
   ```

   `<jbuilder>` is the name of the JBuilder directory and `<jdk>` is the name of the JDK directory. For example, `JBuilder8/jdk1.4/`.

Note If JBuilder is on another drive, include the drive letter. For Windows, use a backslash (\).

If this is the first time you've run **appletviewer**, you'll get a license agreement screen. Click Yes.

If all is well, you will see your applet running in **appletviewer**. If not, look in **appletviewer**'s console to read the error messages.

- If there is a "can't find class" error, verify your deployment.
- If there is a runtime exception, such as `NullPointerException`, debug your code to find the error.

Testing in the browsers

Always test your applet in the browsers after deploying it to the server. This is critical to ensure that all the classes you need for the applet are deployed properly to the server and that the `<applet>` tag and server deployment match.

Here are a few tips for testing in the browser:

- Be sure the browser has Java enabled. Refer to the browser's online help for details.
- Don't use the browser's Reload or Refresh button after recompiling and deploying your revised applet. The browser may continue to load the old applet version from the cache.
- Use the Java Console to read the browser's error messages. To open the Java Console, do one of the following:
  - Select Communicator | Tools | Java Console in older versions of Netscape or Tasks | Tools | Java Console in more recent versions.
  - Select View | Java Console in Internet Explorer.

For more information, see "Solving Common Applet Problems" in the Java Tutorial "Writing Applets."

JBuilder and applets

JBuilder provides a variety of tools for developing your applet:

- An Applet wizard for quickly creating an applet.
- A designer for visually designing an applet user interface.
- JBuilder's AppletTestBed for running and debugging applets.
- Sun's **appletviewer** for running and debugging applets.

For more information on creating applets in JBuilder, see

- "Tutorial: Building an applet" in *Introducing JBuilder*
- "Introducing the Designer'' in *Designing Applications with JBuilder*

Creating applets with the Applet wizard

JBuilder provides the Applet wizard to generate the basic code for an applet. To create an applet using the Applet wizard complete the following steps.

Caution If you're creating an applet for the Web, see "Browser issues" on page 14-5 for information on browser and JDK compatibility issues before designing your applet.

1. Choose File | New Project to create a new project for your applet. The Project wizard appears, suggesting a default project name and directory. Change the project file name and the project directory if you want to give the project a particular name and location. Click the Generate Project Notes File option to create a descriptive HTML file for your project.

   Note The paths for the project files are pre-set in the default project properties. Unless you are an advanced Java user, it's best to leave these unchanged. For more information on the default project properties, see "Creating and managing projects" in *Building Applications with JBuilder*.

   The name of the package for the project is derived from the project file name and is displayed in the Applet wizard. For example, if you create a project called `<home>/jbproject/AppletProject`, the Applet wizard suggests using a package name of `appletproject`.

   For more information on packages, see "Packages" in the "Creating and managing projects" chapter of *Building Applications with JBuilder*.

   a. Click Next to continue to the Project Paths page of the Project wizard. Note the paths:

Output path	Where the class files and applet HTML file are saved.
Backup path	Where the backup files are saved.
Working directory	The project's working directory.
Source path	Where the source and test files are saved — Default is where the source files are saved, Test is where unit testing files are saved.
Documentation path	Where the documentation files are saved.

   The Output, Backup and Working Directory paths and the JDK path can be edited by selecting the ellipsis button. The Source, Test and Documentation paths can be editing by selecting the path and then selecting the Edit button. Required libraries can also be added.

See also

- "Setting the JDK" in *Building Applications with JBuilder*
- "How JBuilder constructs paths" in *Building Applications with JBuilder*

b Click Finish to complete the Project wizard.

The Project wizard creates a project file (`.jpx`). If you selected the Generate Project Notes File option on the first page of the wizard, an HTML file with a name matching the project that contains the default project information. You can edit this to record any pertinent information about the project you want to display.

2 Next, choose File | New and choose the Web tab of the object gallery.

3 Double-click the Applet icon in the object gallery to open the Applet wizard.

a Do not change the suggested package name. Type a new name for the applet class if you like.

b Select the base class you want the applet to extend: `java.applet.Applet` (AWT class) or `java.swing.JApplet` (JFC Swing class).

c Check any of the remaining options you want:

Generate Header Comments	Uses information from the project file as header comments at the top of the applet class file.
Can Run Standalone	Creates a `main()` function so you can test the applet without its being called from an HTML page.
Generate Standard Methods	Creates stubs for the standard applet methods: `start()`, `stop()`, `destroy()`, `getAppletInfo()`, and `getParameterInfo()`.

d Click Next to go to the Applet Parameters page. In this step, you can add parameters. From this information, the wizard generates `<param>` tags within the `<applet>` tag of the applet HTML file and parameter-handling code in the new applet java file.

Fill in one row for each parameter you wish to have.

- To add a new parameter, click the Add Parameter button.
- To select a cell, click it or use the keyboard navigation arrows to move to it.
- To enter a value in a cell, type in a value, or select one if a drop-down list exists.

- To remove a parameter, click in any cell of the parameter row, then click the Remove Parameter button.

The parameter field definitions are as follows:

Name	A name for the parameter. This will be used to provide the `name` attribute in the `<param>` tag in the HTML file and to provide the `name` parameter of the corresponding `getParameter()` call in the Java source.
Type	The type of variable that will be inserted into the Java source code of your applet for holding the value of the parameter coming in from the HTML page.
Desc	A brief description of the parameter. This will be used to describe the parameter when external tools query the applet for what parameters it supports. An example of such a tool is the Applet | Info Browser in **appletviewer.**
Variable	The name of the variable that will be inserted into the Java source code of your applet for holding the value of the parameter coming in from the HTML page.
Default	The default value for the parameter. This is the value that the Java source code in this applet uses if a future HTML file that uses this applet doesn't have a `<param>` tag for this parameter. For an HTML file to provide this parameter, the `name` attribute in the `<param>` tag must exactly match what you've entered in the Name column. Note that this matching is case-sensitive.

e Click Next to go to the HTML Details page. If you don't want to generate the HTML page automatically, you can uncheck the Generate HTML Page option. Otherwise, leave it checked and enter information, such as the title of the page, the name of the applet, the size of the applet on the page, and the `codebase` location. By default, the Applet wizard saves the HTML file in the output directory with the compiled classes.

f Click Next to go to the Define Applet Configuration page.

- In JBuilder SE and Enterprise, check the Create A Runtime Configuration option and enter a name for the configuration in the Name field. For more information, see "Creating an applet runtime configuration" on page 10-3.
- In JBuilder Personal, make sure Overwrite Existing Configuration is checked. Enter a name for the configuration in the Name field.

g Choose Finish to generate the applet `.java` and HTML file.

The Applet wizard creates two files if you selected the Generate HTML Page option on the HTML Details page:

- An HTML file containing an `<applet>` tag referencing your applet class. This is the file you should select to run or debug your applet.
- A Java class that extends `Applet` or `JApplet` and that is designable with the UI designer.

Note In JBuilder SE and Enterprise, an automatic source package node also appears in the project pane if the Enable Source Packages Discovery and Compilation option is enabled on the General page of the Project Properties dialog box (Project | Project Properties).

See also

- "Creating applets with the Applet wizard" on page 14-16
- "Tutorial: Building an applet" in *Introducing JBuilder*

Running applets

To run an applet within a project,

1. Save all files.
2. Compile the project.
3. Do one of the following:

- Choose Run | Run Project or choose the Run button on the toolbar to run the applet from the main class in JBuilder's AppletTestbed. (In JBuilder SE and Enterprise, choose the runtime configuration if more than one exists.)
- Right-click the applet's HTML file in the project pane. This file must have an `<applet>` tag. Select Run to run the applet in Sun's **appletviewer**. (In JBuilder SE and Enterprise, choose the runtime configuration if more than one exists.)

Note You can also select the applet `.java` file if it has a `main()` method and choose Run. You can create an applet with a `main()` method by selecting the Can Run Standalone option on the Enter Applet Class Details page of the Applet wizard.

The applet compiles and runs if there are no errors. The build progress is displayed in a dialog box and the message pane displays any compiler errors. If the program compiles successfully, the classpath is displayed in the message pane and the applet runs.

JBuilder's AppletTestbed and Sun's appletviewer

There are two ways to run an applet in JBuilder:

- JBuilder's AppletTestbed
- Sun's **appletviewer**

If you've created your applet with the Applet wizard, the default behavior is as follows:

- Select Run | Run Project or the Run button to run the applet from the main class in JBuilder's applet viewer, AppletTestbed. (In JBuilder SE and Enterprise, choose the runtime configuration if more than one exists.)
- Right-click the applet HTML file and select Run to run the applet in Sun's **appletviewer**. (In JBuilder SE and Enterprise, choose the runtime configuration if more than one exists.)

You can change the default behavior for Run | Run Project and the Run button by customizing the applet runtime configuration. There are two choices for running an applet in JBuilder:

- Main class — uses JBuilder's AppletTestbed
- HTML file — uses Sun's **appletviewer**

Main class

When you select a main class to run the applet, it runs in JBuilder's applet viewer, AppletTestbed. When you create your applet using the Applet wizard, it sets the main class for you automatically. The selected class must contain an `init()` method.

HTML file

When you select an HTML file to run the applet, it runs in Sun's **appletviewer**. The HTML file must contain the `<applet>` tag and the `code` attribute must contain the fully qualified class name. If the class file is not located in the same directory as the HTML file, the `codebase` attribute must specify the location of the class file in relation to the HTML file.

Running JDK 1.1.x applets in JBuilder

If you run your JDK 1.1.x applet from its main class in JBuilder, the applet is run using JBuilder's AppletTestbed, which requires JFC (Swing) to launch. Because JDK 1.1.x did **not** include the JFC classes, you need to download the JDK 1.1.x-specific version of JFC (Swing), `swingall.jar` from `http://java.sun.com.products`. Then, create a library for the JFC classes and add the library to the project (Tools | Configure Libraries).

Running JDK 1.2 applets in JBuilder

In order to run an applet on Solaris or Linux from within JBuilder, you must add the OpenTools SDK library to your project. Failing to add this library can lead to an exception about a `NoClassDefFoundError:AppletTestbed`. This affects some of the applet samples, including the Primes Swing sample.

Debugging applets

You can debug applets in JBuilder's IDE just as you would debug any Java program. Just as there are two ways to run applets in JBuilder, you can also debug applets two ways: JBuilder's AppletTestbed and Sun's **appletviewer**.

To debug with JBuilder's AppletTestbed, you must run the main applet class which contains the `init()` method:

1. Set the applet's main class, which contains the `init()` method, as the runnable class (use a runtime configuration.)
2. Compile the project.
3. Set a breakpoint in the applet file.
4.  Choose Run | Debug Project or click the Debug button on the toolbar. The debugger takes control.

Note If you're developing your applet using an older JDK, you'll need to switch to a newer JDK (JDK 1.2.2 and above) to debug. Earlier JDKs do not support JPDA debugging API (Java Platform Debugger Architecture) that JBuilder uses. See "Setting the JDK" in *Building Applications with JBuilder*.

For detailed instructions on debugging in JBuilder, see "Debugging Java programs" in *Building Applications with JBuilder*.

To debug with Sun's **appletviewer**, you must run the HTML file using one of these methods:

- From the project pane, follow these steps:
  a. Compile the project.
  b. Set a breakpoint in the applet file.
  c. Right-click the applet HTML file in the project pane and choose Debug. The debugger takes control.

- From the JBuilder Run menu, follow these steps:
  - **a** Set the applet's HTML file as the runnable file by modifying the runtime configuration.
  - **b** Compile the project.
  - **c** Set a breakpoint in the applet file.
  - **d** Choose Run | Debug Project or click the Debug button on the toolbar. The debugger loads in the message pane and the applet runs in Sun's **appletviewer**.

Debugging applets in the Java Plug-in

This is a feature of JBuilder Enterprise

You can also debug applets from within Internet Explorer 5 or Netscape Navigator 6 and above using the Java Plug-in and JBuilder's debugger.

To debug using JDK 1.4,

1. Download and install the plug-in for JDK 1.4 from the Sun site: `http://java.sun.com/products/plugin/`.
2. Set the following runtime parameters in the Java Runtime Parameters field on the Advanced tab of the Java Plug-in Control Panel:

   ```
   -Xdebug -Xnoagent -Djava.compiler=NONE
       -Xrunjdwp:transport=dt_socket,server=y,address=3999,suspend=n
   ```

   For more information on debugging in the plug-in, see: `http://java.sun.com/j2se/1.4/docs/guide/plugin/developer_guide/debugger.html#how`.
3. Launch JBuilder and do the following:
   - **a** Create a new project.
   - **b** Use the Applet wizard to create a simple applet that runs from an HTML file. Make sure the applet uses Swing for its base class (`javax.swing.JApplet`). Add a `JPanel` to the applet, and set the layout to `GridBagLayout`. Add a `JButton` button to the panel, and add an event to the button that changes the button text.
   - **c** Edit the applet's runtime configuration and set the following options on the Debug page of the Runtime Properties dialog box:
     1. Check the Enable Remote Debugging check box.
     2. Select the Attach option.
     3. Make sure the Host Name field is set to `localhost`.
     4. Make sure the Transport Type is set to `dt_socket` and the Address is set to `3999`.

     For further information on remote debugging, see "Remote debugging" in *Building Applications with JBuilder*.

4. Compile the project.
5. Run the HTML Converter and convert the applet's HTML file (in your project's `classes` directory) to the format required for the Java Plug-in. For more information on the HTML converter and on running it from the Sun site at `http://java.sun.com/j2se/1.4/docs/guide/plugin/developer_guide/html_converter.html`.

Note The HTML converter version must match the Plug-in version.

6. Start Netscape or Internet Explorer and open the HTML file. Be sure the browser has Java enabled. Refer to the browser's online help for more information.
7. Switch to JBuilder and set a breakpoint in the applet file. Start debugging your project. The debugger should start up successfully. Switch back to the browser and refresh the page. In JBuilder, the debugger should stop at the breakpoint in the applet.

Deploying applets in JBuilder

JBuilder SE and Enterprise have an Archive Builder that can create the ZIP and JAR archive files for you. You can also create JAR files manually with Sun's **jar** tool provided with the JDK. There are a number of ZIP tools that create ZIP files, but be sure to use a version that accepts long file names.

If you have a large web application with servlets, JSPs, applets, and other web content, you might want to use a WAR file, a Web application archive file. Your web application, WAR file, or JAR file can also be packaged into an EAR file.

See also

- "Using the Archive Builder" in *Building Applications with JBuilder*
- "Deploying Java programs" in *Building Applications with JBuilder*
- **jar** tool at `http://java.sun.com/j2se/1.3/docs/tooldocs/tools.html#basic`
- Chapter 3, "Working with WebApps and WAR files"

Chapter

15

Launching your web application with Java Web Start

Web Development is a feature of JBuilder Enterprise

Java Web Start is a deployment technology from Sun Microsystems. It allows you to launch any Java applet or application from a link on a web page in your web browser. If the applet or application is not present on your computer, Java Web Start downloads all the necessary files and caches them locally so the program can be relaunched from an icon on your desktop or from the web page link.

Java Web Start is the reference implementation of the Java Network Launching Protocol (JNLP) technology. This technology defines a standard file format that describes how to launch a JNLP application. The JBuilder Web Start Launcher wizard generates a JNLP file for you, as well as an HTML homepage for your application.

For more information on Web Start, go to the Java Web Start page at `http://java.sun.com/products/javawebstart/`. You can also look at:

- The Java Web Start *Developer's Guide* at `http://java.sun.com/products/javawebstart/docs/developersguide.html`
- "Frequently Asked Questions" at `http://java.sun.com/products/javawebstart/faq.html`

Considerations for Java Web Start applications

Generally, developing an application for deployment with Java Web Start is the same as developing a stand-alone application. The entry point for the application is the `main()` method and the application must be delivered as a JAR file or a set of JAR files. However, all application resources must be called using the `getResource` mechanism. For more information, see "Application Development Considerations" in the Java Web Start *Developer's Guide* at `http://java.sun.com/products/javawebstart/docs/developersguide.html#dev`.

A special consideration for running applications over the Internet is security. Users are cautious about downloading and running programs on their computers without a guarantee of security to prevent programs deleting files or uploading personal information.

Java Web Start addresses this concern by running all untrusted code in a restricted environment called the *sandbox*. While the application is in the sandbox, Java Web Start can promise that the application cannot compromise the security of local files or files on the network.

Java Web Start also supports digital code signing. This technology allows Java Web Start to verify that the contents of a JAR file have not been modified since they were signed. If verification fails, Java Web Start will not run the application. For more information about Java Web Start and security, see "Security And Code Signing" in the "Application Development Considerations" section of the Java Web Start *Developer's Guide* at `http://java.sun.com/products/javawebstart/docs/developersguide.html#dev`.

The JNLP API provides additional file handling services for running in a restricted execution environment. These classes replace ordinary file download, open, write, and save operations. For example, you would use methods in `javax.jnlp.FileOpenService` to import files from the local disk, even for applications running in the sandbox.

Table 15.1 Overview of JNLP API

Name	Methods for
`BasicService`	Querying and interacting with the environment.
`ClipboardService`	Accessing the system-wide clipboard.
`DownloadService`	Allowing an application to control how its resources are cached.
`FileOpenService`	Importing files from the local disk.
`FileSavService`	Exporting files from the local disk.
`PrintService`	Accessing the printer.
`PersistenceService`	Storing data locally.
`FileContents`	Encapsulating the name and contents of a file.

For more information, see "JNLP API Examples" in the Java Web Start *Developer's Guide* at `http://java.sun.com/products/javawebstart/docs/developersguide.html#api`.

Installing Java Web Start

JBuilder's installation of JDK 1.4 includes an executable that installs Web Start. Run the executable to complete the Web Start installation.

On Windows platforms, follow these steps:

1. Navigate to the `jdk1.4\jre` directory in your JBuilder installation.
2. Run `javaws-1_2-windows-i586-i.exe`.
3. Accept the license agreement.
4. You will be asked for a directory to install to. The default value is the directory you last used for Web Start or `\Program Files\Java Web Start`. If you accept the default, you will have to modify JBuilder's Web Start library definition, as outlined in "Modifying JBuilder's Web Start library definition" on page 15-4. It is easier to change the install directory to `<jbuilder>\jdk1.4\jre\javaws`, where `<jbuilder>` is the directory you installed JBuilder to.
5. Uncheck the Create Shortcuts For All Users option if desired.
6. Choose Next to create the Web Start files in the specified location.
7. The installation program may ask you to shut down your web browser so it can be configured to support Web Start.
8. Installation will create a Web Start program group and optionally add an icon on the desktop.

On Linux or Solaris, follow these steps:

1. Navigate to the `jdk1.4/jre` directory under your JBuilder install directory.
2. Unzip the archive file to create `install.sh` and the related text files.
3. Run `install.sh`.
4. Accept the license agreement.
5. When asked for the location of your JDK, enter `<jbuilder>/jdk1.4/jre/javaws` where `<jbuilder>` is the directory you installed JBuilder to.
6. When you choose Next, a `javaws` directory will be created in the specified location and the Web Start files are created in it.

Modifying JBuilder's Web Start library definition

If you installed Web Start as recommended above, you don't need to modify the library definition. Otherwise:

1. Select Tools | Configure Libraries to open the Configure Libraries dialog.
2. In the tree on the left, select the Web Start library.
3. Select the Classes tab on the right and click the Add button.
4. Browse to the location where you installed `javaws.jar`, select it, and click OK.
5. Click OK to finish defining the library.

Note You don't need the Web Start library in every Web Start-enabled application or applet. You only need to add the library to your project when you're using the JNLP API. In this case, you should also get the developer's download, containing additional jars and documentation, from the Web Start web site at `http://java.sun.com/products/javawebstart/`.

Important Some extra setup is required if you are using Netscape 6. For more information, go to the topic called "Using Java Web Start Technology with Netscape™ 6 Web Browsers " in the *Java Web Start Installation Guide* at `http://java.sun.com/products/javawebstart/docs/installguide.html`.

Web Start and JDK 1.3 or 1.2

If your project uses JDK 1.3 or 1.2, you need to download and install Java Web Start separately. To do this, go to the Java Web Start page at `http://java.sun.com/products/javawebstart/` and click the "Get Java WebStart Now!" button. (Note that the installation depends on the platform.) Once you install Java Web Start on your computer, it is ready to run with your default browser. You do not need to configure either Java Web Start, JBuilder, or your web browser: the three will run seamlessly.

Java Web Start and JBuilder

JBuilder provides a number of features that can turn your applet or stand-alone application into a Web Start applet or application. To do this, you'll follow these general steps:

1. Set up a web server, for example, Tomcat 4.0, for your project. For more information on web servers, see Chapter 9, "Configuring your web server."

2 Create a WebApp with the Web Application wizard. For more information on WebApps, see Chapter 3, "Working with WebApps and WAR files."

3 Create the applet or application's JAR file with the Archive Builder, using the following options on Steps 1 and 2 of the wizard. The options on the remaining steps can be left at the default settings.

Table 15.2 Archive Builder options

Archive Builder	Option
Step 1	Archive type — Web Start Applet or Web Start Application
Step 2	WebApp — The WebApp to put the JAR file in Name — The name of the archive node File — The name of the JAR file to place in the WebApp's directory structure. The default is in the root of the WebApp.

For more information on creating JAR files, see "Using the Archive Builder" in *Building Applications with JBuilder*.

4 Build the project to create the JAR file.

5 Use the Web Start Launcher wizard (File | New | Web page) to create the applet or application's JNLP file and homepage. Enter the following options:

Table 15.3 Web Start Launcher wizard options

Web Start Launcher	Option
Step 1 — Enter required information	Name — Name for Web Start applet or application. WebApp — Name of the WebApp holding the JAR file JAR File — Name and path to JAR file. Main Class — For applets, the applet class. For applications, the class containing `main()` method. Create Homepage — Creates a homepage. Leave checked.
Step 2 — Enter applet information (Displayed for applets only)	Applet Name — Name of applet. Document Base — Root of applet. For more information, see "Root directory" on page 3-5. Width — The applet width (in pixels). Height — The applet height (in pixels).
Step 3 — Enter descriptive information	Title — Application title. Vendor — Company name. Description — Application description. Allow Offline Usage — Check to launch from desktop.

6 Create a new Server runtime configuration. See "Creating a server runtime configuration" on page 10-5 . For the Launch URI, choose the HTML file created by the Web Start Launcher. This file will be located in the root directory of your web application.

7 Right-click the application's HTML file (created by the Web Start Launcher wizard) and choose Web Run "Name of server configuration." The HTML file is located in the Root Directory folder of the WebApp node in the project pane.

8 Copy the application URL from the URL field at the top of the Web View. You can set this option automatically on the Web page of the IDE Options dialog box (Tools | IDE Options).

9 Paste the URL into your external browser.

10 Click the link to your application on the web page.

Each of these steps is outlined in greater detail in Chapter 21, "Tutorial: Running the CheckBoxControl sample application with Java Web Start."

Note Using Java Web Start with applets solves the browser mismatch problem. As long as the browser has the Java Web Start plugin, it can automatically download the appropriate JDK to run the applet. The main class for a Java Web Start applet is the actual applet class to run. The Applet information page gathers a few values that are necessary to simulate the applet running in a web page, where an applet would expect to be running (when in fact it is being hosted by the Java Web Start plugin).

The application's JAR file

JBuilder's Archive Builder allows you to choose a JAR archive type of a Web Start Applet or Web Start Application. The Archive Builder places the resulting JAR file in the selected web application directory (WebApp), so it can be served by the web server.

The application's JNLP file and homepage

JBuilder's Web Start Launcher wizard creates your application's HTML homepage and JNLP file. The wizard also allows you to specify the application's title, the name of the company that created the application, and a description. This information is displayed when Java Web Start launches your application. Additionally, you can use the wizard to allow the application to be started offline, from an icon on the desktop.

Note The Web Start Launcher wizard assumes you've already created and built your application's JAR file.

Warning The Web Start Launcher wizard gives the same name to the JNLP and HTML files. If the name entered in the Name field on Step 1 of the wizard matches the name of an existing HTML or JNLP file in your project, you are asked if you want to overwrite the existing file.

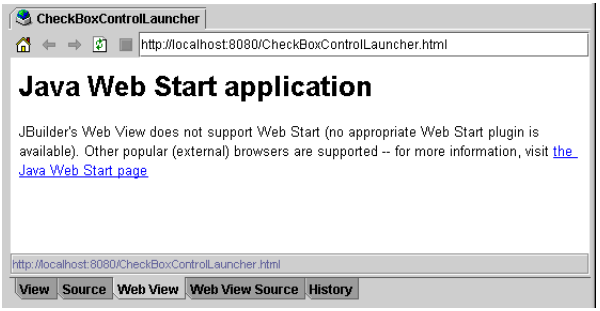
The JNLP file is an XML document. The elements in the file describe application features, such as the application name, vendor, and

homepage; as well as JNLP features. For more information, see "JNLP File Syntax" in the Java Web Start *Developer's Guide* at `http://java.sun.com/products/javawebstart/docs/developersguide.html`.

Note Before you deploy your web start application, you must change the `codebase` attribute in the JNLP file. This attribute is automatically generated as `localhost:8080`. You'll need to change it to match your web server. Java Web Start 1.2 now includes a servlet to automatically generate the appropriate codebase as part of the Developer's kit. See `http://java.sun.com/products/javawebstart/1.2/docs/downloadservletguide.html` for more information.

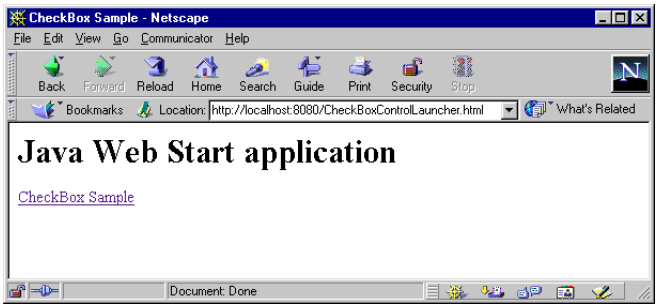
Your application's homepage is an HTML file that contains both JavaScript and VBScript code and HTML tagging. The script determines if you are running the HTML file within JBuilder or from an external web browser. If you're in JBuilder, you'll see a message in the web view explaining that you need Java Web Start to run this application. The web view looks like this:

Figure 15.1 Web view for Java Web Start



If you're in the external browser, you'll see a link to your application (if you have already installed Web Start). Click the link to launch Java Web Start and run your application. The external browser looks like this:

Figure 15.2 External browser for Java Web Start



Important Java Web Start applications cannot be launched from within JBuilder. To launch your application, you need to paste the application URL into your external browser.

Chapter

16

Tutorial: Creating a simple servlet

Web Development is a feature of JBuilder Enterprise

This tutorial shows how to develop and test a servlet within JBuilder, using the Tomcat servlet engine. The servlet accepts user input and counts the number of visitors to a web site. The servlet, generated with the Servlet wizard, displays a form for submitting a user name. When the form is submitted, the servlet is modified to display the user name and a counter for the number of visitors.

All servlets are built by extending the basic `Servlet` class and defining Java methods to deal with incoming connections. This sample servlet extends the `HttpServlet` class that understands the web's HTTP protocol and handles most of the underlying "plumbing" required for a web application.

For more information on servlets, read the following chapters:

- Chapter 4, "Working with servlets"
- Chapter 5, "Creating servlets in JBuilder"

For more information on web applications, read Chapter 3, "Working with WebApps and WAR files."

This tutorial assumes you are familiar with Java and with the JBuilder IDE. For more information on Java, see *Getting Started with Java.* For more information on the JBuilder IDE, see "The JBuilder environment" in *Introducing JBuilder*.

See the Tutorials section in the JBuilder Quick Tips for useful information about viewing and printing tutorials. The Accessibility options section in the JBuilder Quick Tips contains tips on using JBuilder features to improve JBuilder's ease of use for people with disabilities.

For information on documentation conventions used in this tutorial and other JBuilder documentation, see "Documentation conventions" on page 1-4.

Step 1: Creating the project

To develop the sample Hello World servlet in JBuilder, you first need to create a new project. To do this,

1. Select File | New Project to display the Project wizard.
2. Type `SimpleServlet` in the Name field.
3. Make sure the Generate Project Notes File option is checked.
4. Click Finish to close the Project wizard and create the project. You do not need to make any changes to the defaults on Steps 2 and 3 of the wizard.

   The project file `SimpleServlet.jpx` and the project's HTML file are displayed in the project pane.

Step 2: Selecting a server

In this step, you will select the Tomcat 4.0 server to use as the server for this project.

1. Select Project | Project Properties. The Project Properties dialog box appears.
2. Click the Server tab.
3. Make sure the Single Server For All Services In Project radio button is selected.
4. Make sure that Tomcat 4.0 is selected in the server drop-down list.
5. Click OK to close the dialog box.

In the next step, you'll create a WebApp for your servlet. Though you won't be deploying this project, in a real-life situation, you'd always want to create a WebApp.

Step 3: Creating the WebApp

When developing web applications, one of your first steps is to create a WebApp, the collection of your web application's web content files. To create a WebApp,

1. Choose File | New to display the object gallery. Click the Web tab and choose Web Application. Click OK.

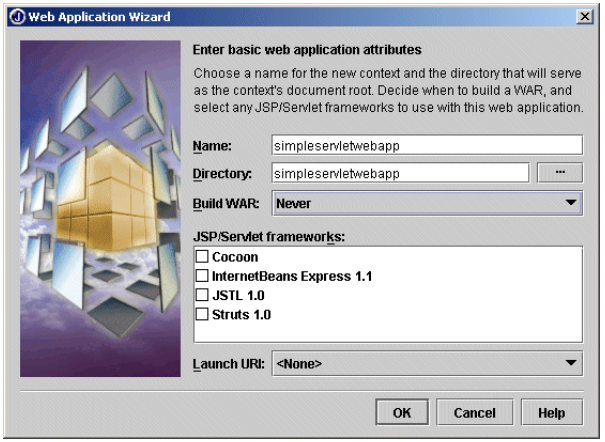
   The Web Application wizard is displayed.

2 Type `simpleservletwebapp` in the Name field. The Directory field is filled in as you type.

3 Change the Build WAR option to Never.

4 Do not choose any of the JSP/Servlet frameworks.

The Web Application wizard should look similar to this:



5 Click OK to close the wizard and create the WebApp.

The WebApp `simpleservletwebapp` is displayed in the project pane as a node. Expand the node to see the Deployment Descriptor and the Root Directory nodes.

For more information on WebApps, see Chapter 3, "Working with WebApps and WAR files."

Step 4: Creating the servlet with the Servlet wizard

In this step, you'll create the servlet using the Servlet wizard. You'll use the wizard to:

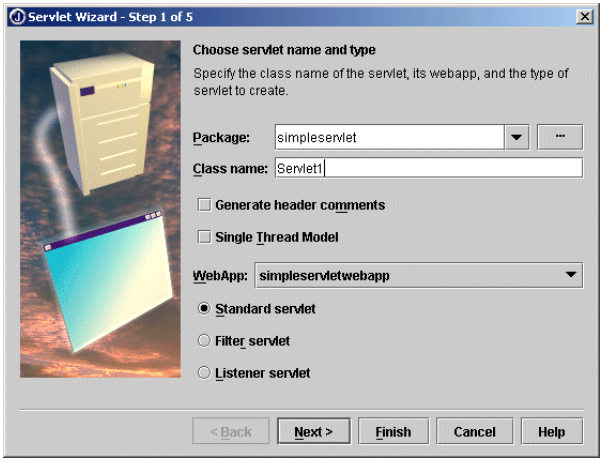
- Enter the servlet's class name.
- Choose the type of servlet and its content type.
- Choose the HTTP methods to implement.

- Create an SHTML file to run the servlet.
- Create parameters for the servlet.

For more information about options, click the Help button on the wizard.

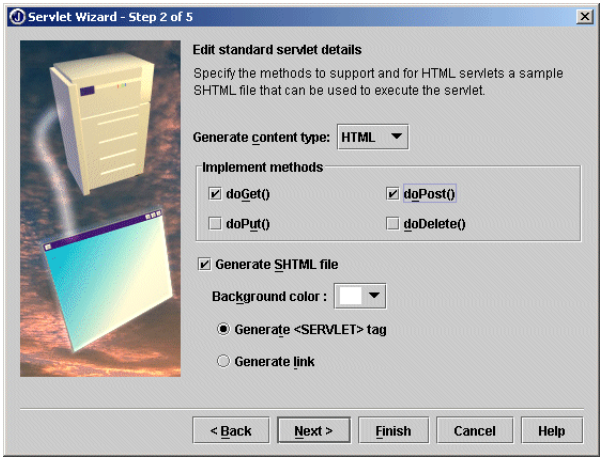
To create the servlet,

1. Choose File | New to display the object gallery.
2. Click the Web tab and choose Servlet. Click OK. The Choose Servlet Name and Type page of the Servlet wizard is displayed.
3. Accept all defaults. The Choose Servlet Name and Type page looks like this:



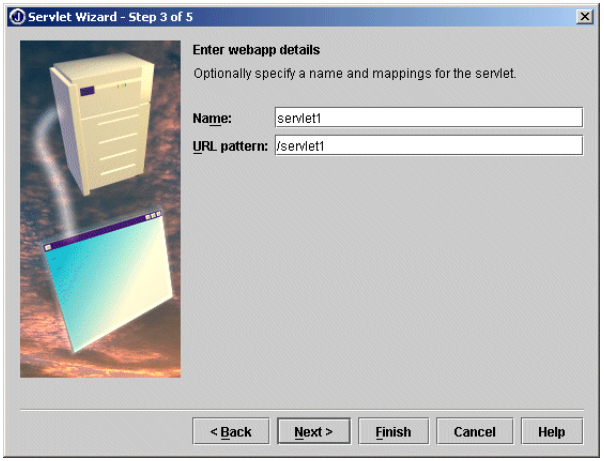
4. Click Next to go to the Enter Standard Servlet Details page.
5. On the Enter Standard Servlet Details page of the wizard, make sure HTML is selected in the Generate Content Type drop-down list. In the Implement Methods section, select the `doPost()` method. Make sure the `doGet()` method is also selected. Select the Generate SHTML File and

Generate <SERVLET> Tag options. The Enter Standard Servlet Details page should look like this:



6 Click Next to go to the Enter WebApp Details page.

7 Accept the default Name and URL Pattern. The Enter WebApp Details page looks like this:



8 Click Next to go to the Enter Servlet Request Parameters page.

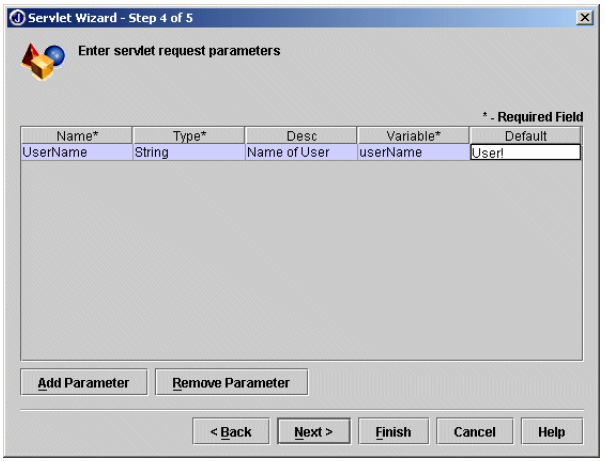
9 Click the Add Parameter button to create a new servlet parameter. This parameter contains the name entered into the servlet's text entry field.

The following table describes the fields and values to use for this tutorial. Enter the values that are in the Value column of the following table.

Table 16.1 Servlet wizard parameter options

Parameter Name	Value	Description
Name*	`UserName`	The parameter used in the `<form>` tag of the SHTML file. It holds the String that the user enters into the form's text entry field.
Type*	`String`	The Java language type of the variable. (This is the default setting and is already selected.)
Desc	`Name of User`	The comment that is added to your servlet source code.
Variable*	`userName`	The name of the variable used in `Servlet1.java` that holds the name of the user passed to it by the SHTML file.
Default	`User!`	The default value of the variable `userName`.

When you're finished, the Enter Servlet Request Parameters page of the wizard will look like this:



10 Click Finish to create the servlet.

The files `Servlet1.java` and `Servlet1.shtml` are added to the project. Note that `Servlet1.shtml` was added to the Root Directory node of the WebApp `simpleservletwebapp`. The Servlet library is added to the Required Libraries list on the Paths page of the Project Properties dialog box (Project | Project Properties). The Servlet wizard also automatically creates a runtime configuration so the servlet can be web run in the IDE. For more information, see "Creating a runtime configuration" on page 10-2.

11 Choose File | Save All to save your work.

Step 5: Adding code to the servlet

In this step, you'll add code to `Servlet1.java`. This code creates a counter for the number of times the page has been visited and displays the count.

1 Open `Servlet1.java` in the editor. Find the comment:

```
// Initialize global variables
```

near the top of the file. Immediately before that line, type the following line of code:

```
int connections = 0;
```

This line of code creates the variable `connections` and initializes it to zero.

2 Search for the line of code that contains the string:

```
The servlet has received a POST. This is the reply.
```

Immediately after that line of code add the following code:

```
out.println("<p>Thanks for visiting, ");
out.println(userName);
out.println("<p>");
out.println("Hello World - my first Java servlet program!");
out.println("<p>You are visitor number ");
out.println(Integer.toString(++connections));
```

These lines of code get the `UserName` parameter and display it in an `out.println` statement. The code then increments the number of visitors and displays it.

3 Choose File | Save All to save your work.

Step 6: Compiling and running the servlet

To compile and run the servlet,

1 Choose Project | Make Project "SimpleServlet.jpx."

2 Right-click `Servlet1.shtml` in the project pane. (It is in the Root Directory node of the `simpleservletwebapp` node.)

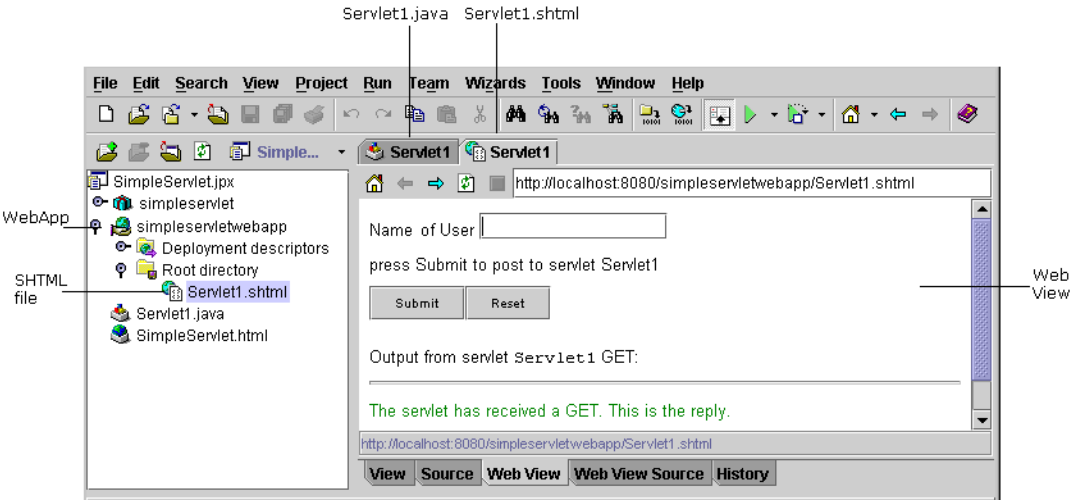
3 Choose the Web Run command.

Note You can also run servlets directly by right-clicking the `.java` file in the project pane, and selecting Web Run. In this example, you are running the servlet from the SHTML file because that is where the parameter input fields and Submit button are coded, based on our selections in the Servlet wizard.

Running the SHTML file starts Tomcat, JBuilder's default web server. The output from Tomcat is displayed in the message pane. HTTP commands

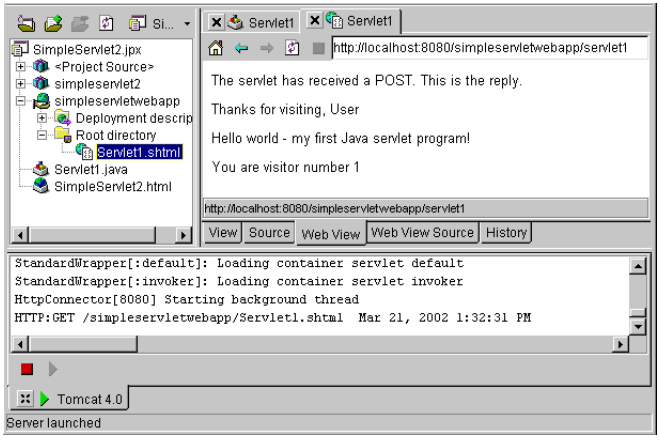
and parameter values are also echoed to the output pane. Two new tabs appear in the content pane for the servlet: Web View and Web View Source. The running servlet displays in the web view. It looks like this:

Figure 16.1 Servlet running in the web view



To run the servlet, type a name, such as `User` in the text box, then click the Submit button. The `doPost()` method is called, and the response is displayed in the web view, as shown below.

Figure 16.2 Servlet running after name submitted



To run the servlet again and see the number of connections increase, click the back arrow at the top of the web view. Type another user name and click the Submit button. When the reply from the `doPost()` method is displayed, you'll see that the number of connections has increased.

To stop the web server, click the Reset Program button directly above the web server tab. If you make changes to your source code, you should stop the web server before re-compiling and re-running.

You have completed your first servlet program. For a more advanced servlet that connects to a database, see Chapter 17, "Tutorial: Creating a servlet that updates a guestbook."

Chapter

17

Tutorial: Creating a servlet that updates a guestbook

Web Development is a feature of JBuilder Enterprise

This tutorial shows how to create a servlet that accepts user input and saves the data to a JDataStore database.

When you complete this tutorial, your project will contain the following classes:

- `FormServlet.java` — This is the runnable class in the program. Its `doGet()` method displays an input form using an HTML `<form>` tag. The servlet posts the user values (via parameters) to `DBServlet`.
- `DBServlet.java` — This servlet passes parameter values (in its `doPost()` method) to `DataModule1`. Code in the `doGet()` method renders the Guestbook JDataStore as an HTML table.
- `DataModule1.java` — The data module that contains the program's business logic. Code in the data module connects to the Guestbook JDataStore, updates it, and saves changes.

This tutorial assumes that you are familiar with servlets and with JBuilder's JDataStore and DataExpress technologies. For more information on servlets, see

- Chapter 4, "Working with servlets"
- Chapter 5, "Creating servlets in JBuilder"

To get started, you can also work through a less complex "Hello World" servlet — see Chapter 16, "Tutorial: Creating a simple servlet." For more information about JBuilder's database functionality, see the *Database Application Developer's Guide*. For more information about JDataStore, see the *JDataStore Developer's Guide.*

The completed sources for this tutorial can be found in the `samples/WebApps/GuestbookServlet` directory of your JBuilder installation.

See the Tutorials section in the JBuilder Quick Tips for useful information about viewing and printing tutorials. The Accessibility options section in the JBuilder Quick Tips contains tips on using JBuilder features to improve JBuilder's ease of use for people with disabilities.

For information on documentation conventions used in this tutorial and other JBuilder documentation, see "Documentation conventions" on page 1-4.

Step 1: Creating the project

To develop the sample Guestbook servlet in JBuilder, you first need to create a new project. To do this,

1. Select File | New Project to display the Project wizard.
2. Type `GuestbookServlet` in the Name field.
3. Make sure the Generate Project Notes File option is checked.
4. Click Finish to close the Project wizard and create the project. You do not need to make any changes to the defaults on Steps 2 and 3 of the wizard.

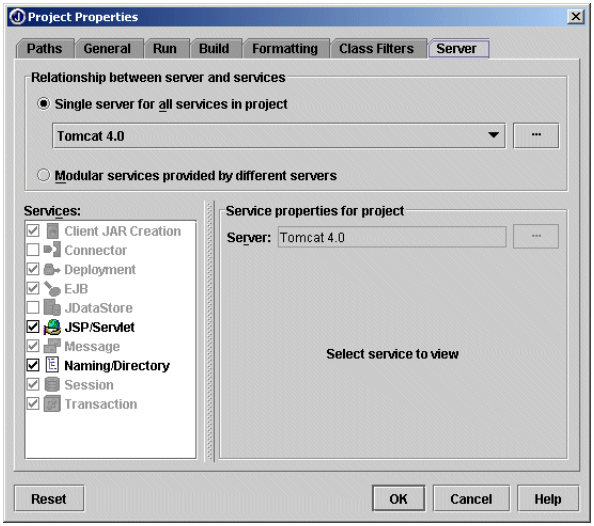
   The project file `GuestbookServlet.jpx` and project's HTML file are displayed in the project pane.

Step 2: Selecting a server

In this step, you will make sure the Tomcat 4.0 server is selected as the server to use for this project.

1. Select Project | Project Properties. The Project Properties dialog box is displayed.
2. Click the Server tab.
3. Make sure the Single Server For All Services In Project radio button is selected.

4 Make sure that Tomcat 4.0 is selected in the server drop-down list. The Server tab should look similar to this:



5 Click OK to close the dialog box.

Step 3: Creating the WebApp

When developing web applications, one of your first steps is to create a WebApp, the collection of your web application's web content files. To create a WebApp,

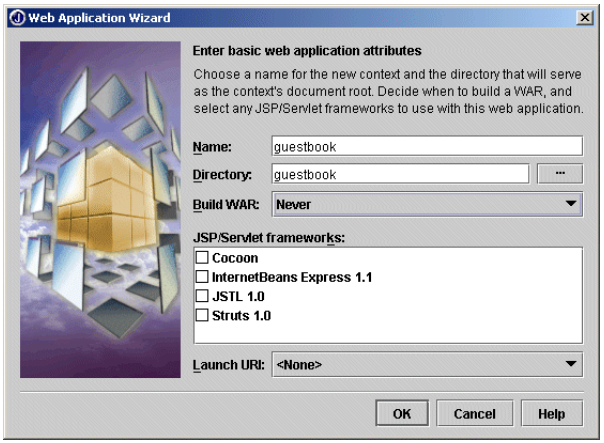
1 Choose File | New to display the object gallery. Click the Web tab and choose Web Application. Click OK.

The Web Application wizard is displayed.

2 Type `guestbook` in the Name field. The Directory field is automatically filled in as you type.

3 Change the Build WAR option to Never. Don't choose a JSP/Servlet framework.

The Web Application wizard should look similar to this:



4 Click OK to close the wizard.

The WebApp `guestbook` is displayed in the project pane as a node. Expand the node to see the Deployment Descriptor sand the Root Directory nodes.

For more information on WebApps, see Chapter 3, "Working with WebApps and WAR files."

Step 4: Creating the servlets

In this step, you'll create the two servlets in the project:

- `FormServlet.java` — This is the runnable class in the program. Its `doGet()` method displays an input form using an HTML `<form>` tag. The servlet posts the user values (via parameters) to `DBServlet`.
- `DBServlet.java` — This servlet passes parameter values (in its `doPost()` method) to `DataModule1`. Code in the `doGet()` method renders the Guestbook JDataStore as an HTML table.

To create `FormServlet.java`,

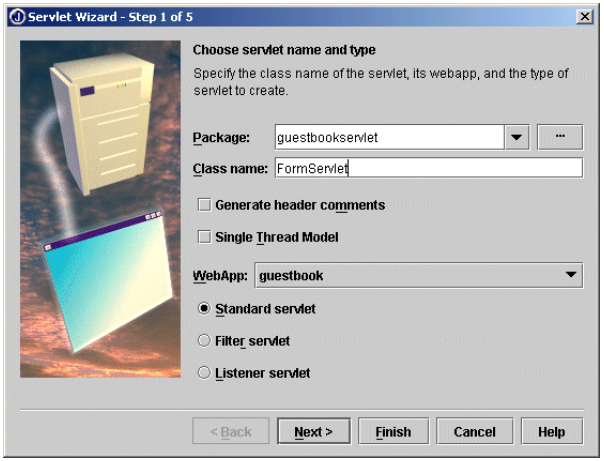
1. Choose File | New to display the object gallery. Click the Web tab and choose Servlet. Click OK.

   The Choose Servlet Name and Type page of the Servlet wizard is displayed.

2. Leave the package name set to `guestbookservlet`. Change the Class Name field to `FormServlet`.

3. Make sure the Generate Header Comments and Single Thread Model options are not selected. Verify that `guestbook` is selected in the WebApp drop-down list. (You just created this WebApp in the previous step.) Leave the Standard Servlet option selected.

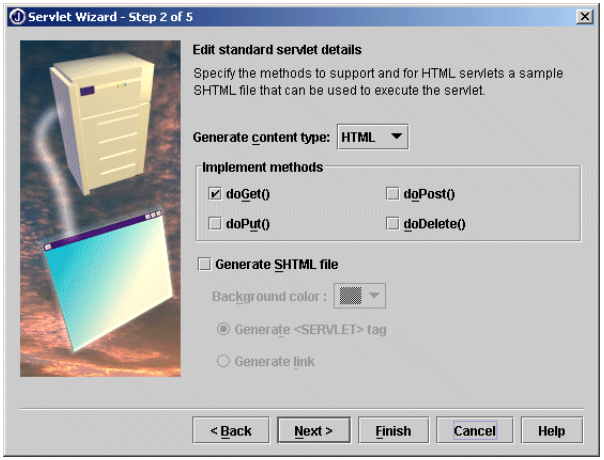
   The Choose Servlet Name and Type page of the wizard should look like this:



4. Click Next to go to the Enter Standard Servlet Details page of the wizard.

5 Make sure that the Content Type option is set to HTML and that the `doGet()` method is selected. Uncheck the Generate SHTML File option.

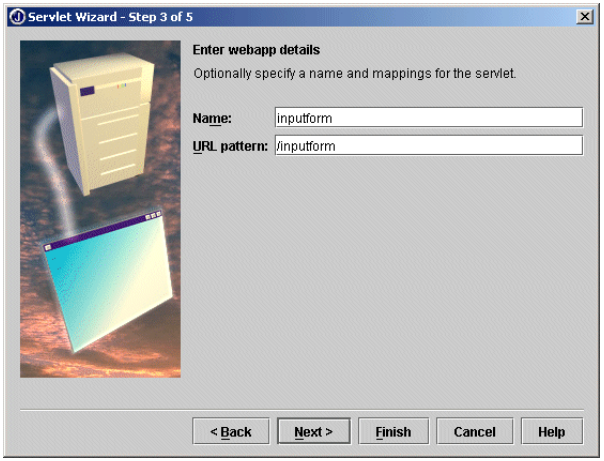
When you're finished, the Enter Standard Servlet Details page of the wizard should look like this:



6 Click Next to go to the Enter WebApp Details page of the wizard.

7 Change the default value in the Name field to `inputform`. Change the value in the URL Pattern field to `/inputform`. Make sure the entries are all lowercase. The URL Pattern entry is used to run the servlet instead of the class name.

The Enter WebApp Details page should look like this:



8 Click Finish to create the servlet. You do not need to set other options.

9 Choose File | Save All or click on the toolbar to save your work.

To create `DBServlet.java`,

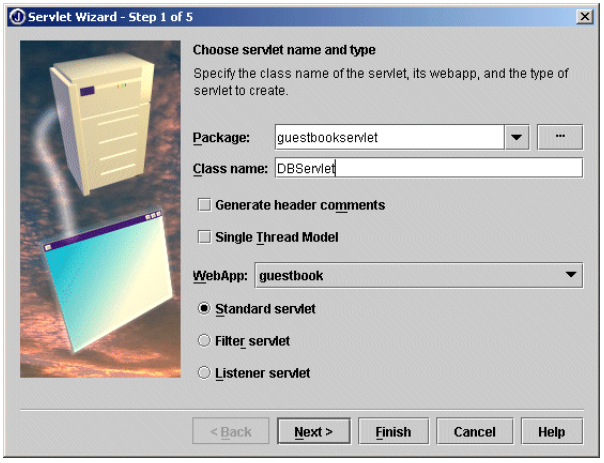
1 Choose File | New to display the object gallery. Click the Web tab and choose Servlet. Click OK.

The Choose Servlet Name and Type page of the Servlet wizard is displayed.

2 Leave the package name set to `guestbookservlet`. Change the Class field to `DBServlet`.

3 Make sure the Generate Header Comments and Single Thread Model options are not selected. The WebApp `guestbook` should be selected in the WebApp drop-down list. Leave the Standard Servlet option selected.

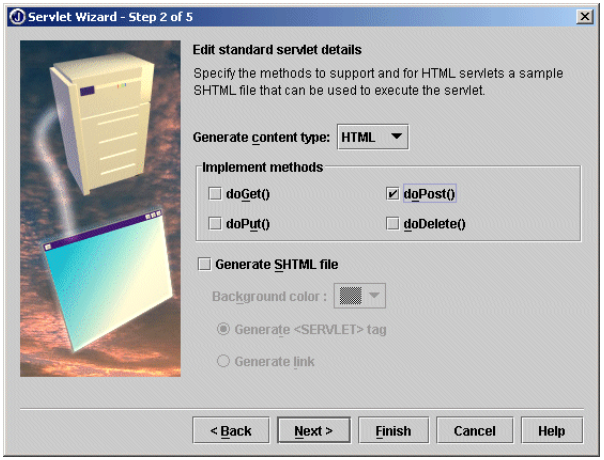
The Choose Servlet Name and Type page of the wizard should look like this:



4 Click Next to go to the Enter Standard Servlet Details page of the wizard.

5 Make sure the Content Type is set to HTML. Unselect the `doGet()` method. Select the `doPost()` method instead. Do not select the Generate SHTML File option.

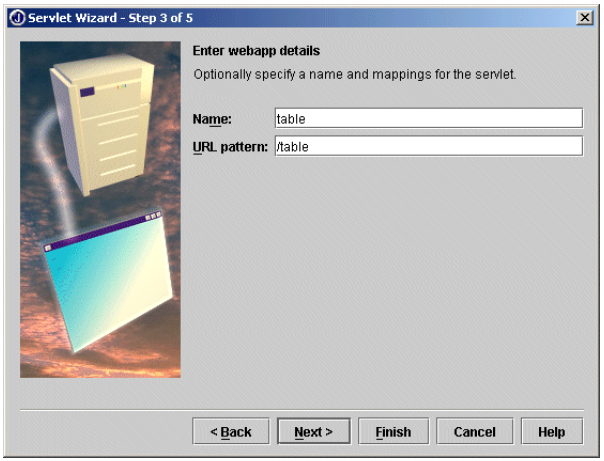
When you're finished, the Enter Standard Servlet Details page should look like this:



6 Click Next to go to the Enter WebApp Details page of the wizard.

7 Change the default value in the Name field to `table`. Change the value in the URL Pattern field to `/table`. Make sure the entries are all lowercase. You will use the name `table` instead of the class name `DBServlet` in `FormServlet's` HTML `<form>` tag.

The Enter WebApp Details page should look like this:



8 Click Finish to create the servlet. You do not need to set other options.

9 Choose File | Save All or click on the toolbar to save your work.

Step 5: Creating the data module

To create the data module that connects to the Guestbook JDataStore and performs the update tasks,

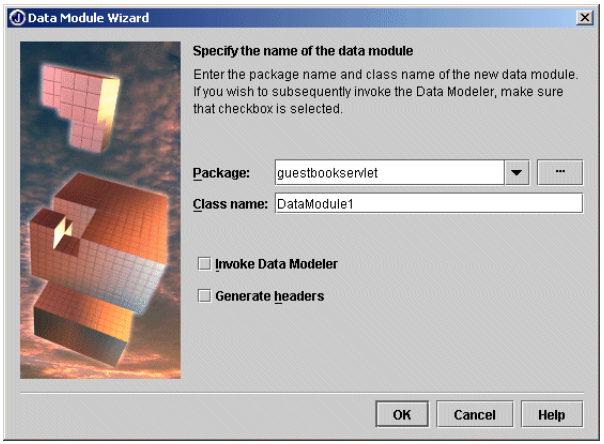
1. Choose File | New to display the object gallery. On the General page, select Data Module and click OK.

   The Data Module wizard is displayed.

2. Leave the package name set to `guestbookservlet`. Keep the default Class Name of `DataModule1`.

3. Uncheck the Invoke Data Modeler and Generate Headers options.

   When you're finished, the Data Module wizard will look like this:



4. Click OK to create the data module.

5. Choose File | Save All or click on the toolbar to save your work.

6. Choose Project | Make Project "GuestbookServlet.jpx" to compile the project and create class files.

Step 6: Adding database components to the data module

In this step, you'll use JBuilder's UI Designer to add database components to the data module. For more information about the designer, see "Introducing the designer" in *Designing Applications with JBuilder*. For more information about database components, see "Connecting to a database" in the *Database Application Developer's Guide*.

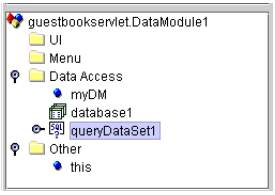
To open the designer, double-click `DataModule1.java` in the project pane. Then, click the Design tab at the bottom of the content pane.

To add data access components to the data module,

1. Choose the DataExpress tab on the component palette.
2. Click the Database icon. Then click the component tree in the structure pane to add the database component to the data module.

3. Click the QueryDataSet icon. Then click the component tree.

   When you're finished, the component tree should look like this:



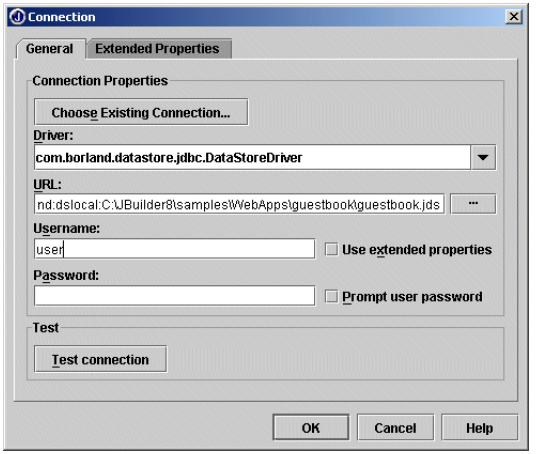
To connect to the Guestbook JDataStore,

1. Choose `database1` in the component tree.
2. Click the `connection` property in the Inspector, then click the ellipsis button to the right of the property name.

   This opens the Connection dialog box.
3. On the General page of the Connection dialog box, make sure the Driver is set to `com.borland.datastore.jdbc.DataStoreDriver`.
4. Click the ellipsis button to the right of the URL field to display the Create URL For DataStore dialog box. You use this dialog box to choose the JDataStore to connect to.
5. Select the Local DataStore Database option.
6. Click the ellipsis button to browse to the Guestbook JDataStore (`guestbook.jds`) in the `samples/WebApps/guestbook` directory of your JBuilder installation. Click Open to select the JDataStore.
7. Click OK to close the Create URL For Datastore dialog box.

8 On the General page of the Connection dialog box, type `user` in the Username field.

The Connection dialog box should look similar to this:



9 Click the Test Connection button to test the connection to the Guestbook JDataStore. If the connection is successful, the word `Success` displays to the right of the button. If the connection is not successful, an error message attempts to explain why the connection failed.

10 Click OK to close the Connection dialog box.

JBuilder adds the `database1.setConnection()` method to the `jbInit()` method of the data module.

To access data in the Guestbook, you need to define a query. For more information about queries, see "Querying a database" in the *Database Application Developer's Guide*. To create the query in JBuilder,

1 Click the `queryDataSet1` component in the component tree.

2 Click the `query` property in the Inspector and click the ellipsis button.

The Query dialog box is displayed.

3 On the Query page, choose `database1` from the Database drop-down list.

4 In the SQL Statement box, type:

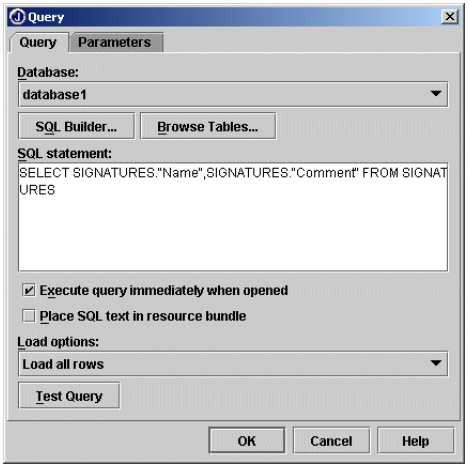
```
SELECT SIGNATURES."Name",SIGNATURES."Comment" FROM SIGNATURES
```

This query loads all the values in the Name and Comment fields in the SIGNATURES table of the Guestbook JDataStore. When you enter the query, use the capitalization displayed above.

5 Make sure Execute Query Immediately When Opened is selected.

6 In the Load Options drop-down list, make sure the Load All Rows option is selected.

The Query page of the Query dialog box should look like this:



7 Click the Test Query button to test the query. If the query is successful, the word `Success` displays to the right of the button. If the query cannot be executed, an error message attempts to explain why the query failed.

8 Click OK to close the Query dialog box.

JBuilder adds the `queryDataSet1.setQuery()` method to the `jbInit()` method.

Note You may see the following message displayed in the Designer tab of the message pane:

```
Failed to create live instance for variable 'myDM'
guestbookservlet.DataModule1
```

For now, you can ignore this message. You'll fix this in a later step. Click the X on the Designer tab at the bottom of the AppBrowser to close it.

9 Click on the toolbar to save your work.

Step 7: Creating the data connection to the DBServlet

In this step, you'll create a data connection to the `DBServlet`. The connection allows the servlet to pass data to the data module.

1. Double-click `DBServlet.java` in the project pane to open it in the editor.
2. Choose Project | Make Project "GuestbookServlet.jpx" to compile the project.
3. Choose Wizards | Use DataModule to open the Use DataModule wizard.
4. Keep the default DataModule class name. Change the Field Name do `dm`. Leave the instance declaration to `Share (static) instance of dataModule`.
5. Click OK to close the wizard. The wizard adds the following line of code to the `DBServlet` class declaration:

```
DataModule1 dm;
```

It also adds the following constructor:

```
public DBServlet() {
    try {
      jbInit();
    }
    catch(Exception e) {
      e.printStackTrace();
    }
}
```

Following the constructor, the following code is added:

```
private void jbInit() throws Exception {
    dm = guestbookservlet.DataModule1.getDataModule();
}
```

6. Click the Save All icon on the toolbar to save your work.

Now that the servlet is connected to the data module, you need to make both servlets and the data module do something. From this point on in our tutorial, you'll be entering code directly into the editor. The first step will be to create an input form in `FormServlet`.

Step 8: Adding an input form to FormServlet

In this step, you'll add code to the `doGet()` method of `FormServlet`. This code creates a form using the HTML `<form>` tag. The form reads in two values, `UserName` and `UserComment`, entered by the user. This data is posted to `DBServlet`, the servlet that communicates with the data module.

A `<form>` tag is a standard HTML tag that creates an input form to gather data from and display information to a user. The tag contains `action` and `method` attributes. These attributes tell the servlet what to do when the form's Submit button is pressed. In our tutorial, the `action` attribute calls `DBServlet`. The `method` attribute posts the `UserName` and `UserComment` parameters to `DBServlet`.

To add code to `FormServlet`,

1. Double-click `FormServlet` in the project pane to open it in the editor. (It may already be open.)
2. Find the `doGet()` method near the top of the file.

Tip You can search by positioning the cursor in the structure pane and typing `doGet`.

3. Remove all the lines that start with `out.println`.
4. Add the following lines of code to the `doGet()` method, after `PrintWriter out = response.getWriter();`.

```
out.println("<html><body>");
out.println("<h1>Sign the guestbook</h1>");
out.println("<strong>Enter your name and comment in the input fields
    below.</strong>");
out.println("<br><br>");
out.println("<form action=table method=POST>");
out.println("Name<br>");
out.println("<input type=text name=UserName value=\"\"
    size=20 maxlength=150>");
out.println("<br><br>");
out.println("Comment<br>");
out.print("<input type=text name=UserComment value=\"\"
    size=50 maxlength=150>");
out.println("<br><br><br><br>");
out.print("<input type=submit value=Submit>");
out.println("</form>");
out.println("</html></body>");
```

Tip You can copy and paste this code directly in the editor, or copy it from the sample in the `samples/WebApps/GuestbookServlet` folder of your JBuilder installation.

5. Click the Save All icon on the toolbar to save your work.

Step 9: Adding code to the DBServlet doPost() method

In this step, you'll add code to the DBServlet's doPost() method that

- Reads in the UserName and UserComment parameters from FormServlet.
- Calls the DataModule method that updates the Guestbook JDataStore, passing UserName and UserComment parameter values.
- Calls the data module method that saves changes to the JDataStore.

To do this,

1. Double-click DBServlet in the project pane to open it in the editor. (It may already be open.)
2. Find the doPost() method.
3. Remove the following line of code from the doPost() method:

```
out.println("<p>The servlet has received a POST. This is the reply.</p>");
```

4. Insert the following lines of code, keeping the cursor at the location where you just removed code:

```
String userName = request.getParameter("UserName");
String userComment = request.getParameter("UserComment");
dm.insertNewRow(userName, userComment);
dm.saveNewRow();
doGet(request, response);
```

Tip You can copy and paste this code directly in the editor, or copy it from the sample in the samples/WebApps/GuestbookServlet folder of your JBuilder installation.

The first two lines of code get the values in the UserName and UserComment parameters that are passed in from FormServlet. The next lines call two methods in the data module:

- insertNewRow() — inserts the new Name and Comment values into the last row of the table.
- saveNewRow() — saves the changes in the Guestbook JDataStore.

The last line calls the servlet's doGet() method which renders the Guestbook table in HTML.

Note These methods have not been added to the data module yet, so you will see errors in the Errors folder of the structure pane. You can ignore these errors for now.

5. Click the Save All icon on the toolbar to save your work.

Step 10: Adding code to render the Guestbook SIGNATURES table

In this step, you'll add a `doGet()` method to `DBServlet` that renders the Guestbook SIGNATURES table in HTML. Both existing rows and the new row are displayed.

1 Insert the following code after `DBServlet's doPost()` method:

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
  response.setContentType(CONTENT_TYPE);
  PrintWriter out = response.getWriter();
  out.println("<html>");
  out.println("<body>");
  out.println("<h2>" + dm.getQueryDataSet1().getTableName() + "</h2>");
  Column[] columns = dm.getQueryDataSet1().getColumns();
  out.println ("<table border = 1><tr>");
  for (int i=1; i < columns.length; i++) {
    out.print("<th>" + columns[i].getCaption() + "</th>");
  }
  out.println("</tr>");
  dm.getQueryDataSet1().first();
  while (dm.getQueryDataSet1().inBounds()) {
    out.print("<tr>");
    for (int i = 1; i < columns.length; i++) {
      out.print ("<td>" + dm.getQueryDataSet1().format(i) + "</td>");
    }
    out.println("</tr>");
    dm.getQueryDataSet1().next();
  }
  out.println("</table>");
  out.println("</body>");
  out.println("</html>");
  }
```

Tip You can copy and paste this code directly in the editor, or copy it from the sample in the `samples/WebApps/GuestbookServlet` folder of your JBuilder installation.

2 Add the following packages to the list of `import` statements at the top of the file. This ensures that the servlet will compile.

```
import com.borland.dx.dataset.*;
import com.borland.dx.sql.dataset.*;
import com.borland.datastore.*;
```

Tip You can use MemberInsight (*Ctrl+H*) to assist you in completing your `import` statements.

Note Ignore any errors you see in the Errors folder of the structure pane.

3 Click Save All on the toolbar to save your work.

What the doGet() method does

The `doGet()` method you just added renders the Guestbook SIGNATURES table in HTML. It cycles through the rows in the JDataStore table and displays them in the web browser.

The following lines of code contain the standard servlet method declaration, set up the servlet content type, and define the output writer:

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
  response.setContentType(CONTENT_TYPE);
  PrintWriter out = response.getWriter();
```

The next two lines of code set up the output as HTML and start the HTML page.

```
out.println("<html>");
out.println("<body>");
```

The following line of code prints the name of the JDataStore table, SIGNATURES, at the top of the HTML page. The code uses the `queryDataSet1.getTableName()` method to get the table name.

```
out.println("<h2>" + dm.getQueryDataSet1().getTableName() + "</h2>");
```

The next line calls the `queryDataSet1.getColumns()` method to get the column names, and return them as an array.

```
Column[] columns = dm.getQueryDataSet1().getColumns();
```

The following line creates the table with the `<table>` tag and creates the first row of the table.

```
out.println ("<table border = 1><tr>");
```

Then, the code uses a `for` loop to cycle through the column names in the array of columns, retrieve the column captions, and display each caption in a table row. In this tutorial, the program is only displaying the second and third columns of the JDataStore. It is not displaying the first column, the internal row number.

```
for (int i = 1; i < columns.length; i++) {
  out.print ("<th>" + columns[i].getCaption() + "</th>");
}
```

The line following the `for` block closes the table row.

```
out.println("</tr>");
```

The next line positions the cursor on the first row of the JDataStore.

```
dm.getQueryDataSet1().first();
```

The `while` loop cycles through the JDataStore and displays data in the second and third columns of the table. The first time through, it displays the data in the first row. Then, the `next()` method positions the cursor on the next row of the JDataStore. The `while` loop continues displaying data

while the cursor is in bounds, that is while the `inBounds()` method reports that the navigation falls between the first and last record visible to the cursor. When this condition is not met, the table and the HTML page are closed.

```
while (dm.getQueryDataSet1().inBounds()) {
  out.print("<tr>");
    for (int i = 1; i < columns.length; i++) {
      out.print ("<td>" + dm.getQueryDataSet1().format(i) + "</td>");
  out.println("</tr>");
  dm.getQueryDataSet1().next();
}
out.println("</table>");
out.println("</body>");
out.println("</html>");
```

Step 11: Adding business logic to the data module

You're almost done. Right now, there's still no code to write the newly added data to the Guestbook JDataStore and save it. That code will be added to `DataModule1`. This code will open the data set, insert the new row (using the `userName` and `userComment` strings passed in from `DBServlet`), and save the new row to the JDataStore.

Follow these steps to add business logic to the data module:

1. Double-click `DataModule1.java` in the project pane to open it in the editor. (It may already be open in the UI designer; if so, click the Source tab at the bottom of the content pane.)
2. Find the `jbInit()` method, using the Search | Find command. Add the following code before the method's closing curly brace:

   ```
   queryDataSet1.open();
   ```

   This code opens the dataset. The dataset must be open before you can insert or save data. In the data module, the dataset is opened right after the code that connects to the database and sets up the query.
3. Add code for the method that inserts a new row. To add a row, you need to create a `DataRow` object, then pass data from the `userName` and `userComment` parameters into the `DataRow`. You'll add this method after the `jbInit()` method. Simply move the cursor down a line, past the method's closing curly brace, and press `Enter` a few times. Add the following method:

   ```
   public void insertNewRow(String userName, String userComment) {
     try {
       DataRow dataRow1 = new DataRow(queryDataSet1, new String[]
       { "Name", "Comment"});
       dataRow1.setString("Name", userName);
       dataRow1.setString("Comment", userComment);
       queryDataSet1.addRow(dataRow1);
     }
   ```

```
    catch (DataSetException ex) {
      ex.printStackTrace();
    }
  }
```

The first line of the method creates a new `DataRow` object that holds the new Name and Comment values. The second and third rows pass the values in the `userName` and `userComment` parameters into the Name and Comment fields. The last row adds the `DataRow` object to the dataset.

4 Add the following method to save the new row to the dataset after the `insertNewRow()` method:

```
public void saveNewRow() {
  try {
    database1.saveChanges(queryDataSet1);
  }
  catch (DataSetException ex) {
    ex.printStackTrace();
  }
}
```

5 Click the Save All icon on the toolbar to save your work.

You have now added all the code to the program. In the next step, you'll compile and run it.

Step 12: Compiling and running your project

Before compiling and running, you need to check dependencies for the WebApp and edit the runtime configuration.

To check the WebApp dependencies,

1 Right-click the `guestbook` WebApp in the project pane and choose Properties.

2 Choose the Dependencies page in the Properties dialog box.

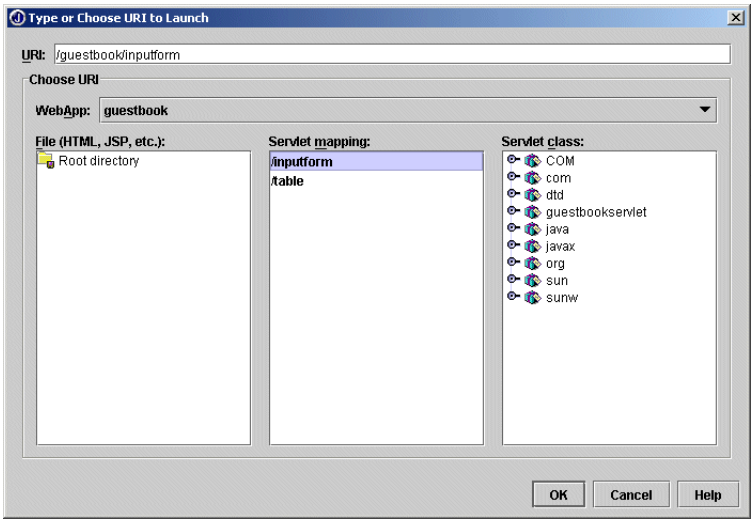
Make sure the library settings for the Data Express and JDataStore Server libraries on the right side of the page are set to Include All. If they are set to another option, choose the Always Include All Classes And Resources option at the bottom of the page for each library.

3 Click OK to close the dialog box.

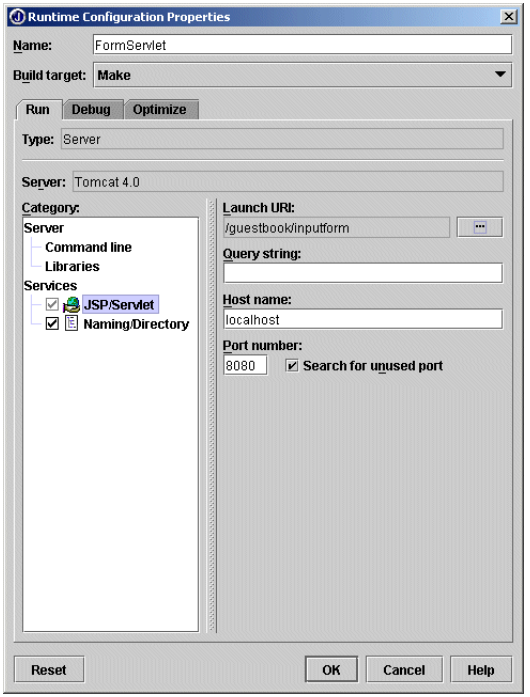
To edit the runtime configuration,

1. Choose Run | Configurations to display the Run page of the Project Properties dialog box. Choose the FormServlet configuration and click Edit to display the Runtime Configuration Properties dialog box.
2. Choose the JSP/Servlet service in the tree on the left side of the page.
3. Click the ellipsis button to the right of the Launch URI field to display the Type Or Choose URI To Launch dialog box where you choose the name of the servlet to launch.
4. Choose `/inputform` in the Servlet Mapping directory tree in the middle of the dialog box. The URI field at the top of the dialog box now contains: `/guestbook/inputform`. This is the name of the WebApp you created in the Web Application wizard, followed by the servlet's name.

   The Type Or Choose URI To Launch dialog box should look like this:



5 Click OK to close the Type Or Choose URI To Launch dialog box. The Runtime Configuration Properties dialog box should look similar to this:



6 Click the OK button two more times to close the Runtime Configuration Properties and Project Properties dialog boxes.

7 Click the Save All icon on the toolbar to save your work.

To compile and run your project,

1 Choose Project | Make Project "GuestbookServlet.jpx."

2 Choose Run | Run Project.

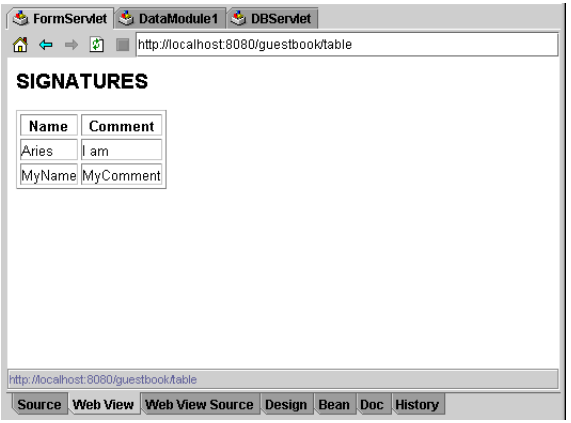
Output from the Tomcat web server is displayed in the message pane.

3 `FormServlet's` input form is displayed in the web view. The URI is `/guestbook/inputform/` and matches what you selected in the URI Launch dialog box.

4 Type `MyName` in the Name field and `MyComment` in the Comment field.

5 Click the Submit button.

The Guestbook SIGNATURES table is rendered in HTML. `MyName` and `MyComment` are displayed in the last row of the table. Note that the URI has changed to `http://localhost:8080/guestbook/table`, indicating that the program is running `DBServlet`.



For more information on URLs, URIs, and servlets, see "How URLs run servlets" on page 10-9.

6 You can click the back arrow to the left of the URL Location field to return to the input form and enter another name and comment.

7 Click the Reset Program button directly above the web server tab to stop the web server. You must stop the web server before you compile and run the servlet again, after making changes.

Note You can open the Guestbook JDataStore in the JDataStore Explorer (Tools | JDataStore Explorer) to verify that the new data was saved to the table.

You have completed the tutorial. You now know how to create an HTML input form for use in a servlet, pass a parameter from one servlet to another, connect a servlet to a data module, pass parameters from a servlet to a data module, and use a data module to update a JDataStore.

Chapter

18

Tutorial: Creating a JSP using the JSP wizard

Web Development is a feature of JBuilder Enterprise

This tutorial walks you through developing a JavaServer Page (JSP) using JBuilder's JSP wizard. This JSP takes text input, displays the text as output when the Submit button is clicked, and uses a JavaBean to count the number of times the web page is visited.

The JSP wizard is a good starting point for developing JSPs. It doesn't generate a complete application, but it does take care of all the tedious details required to get your application up and running. You get to this wizard by selecting New from the File menu, clicking the Web tab, then selecting JavaServer Page. For complete information on the options in the JSP wizard, see the topic on the JSP wizard in the online help.

For development testing purposes in this tutorial, you'll use Tomcat. This tutorial uses Tomcat because it is included with JBuilder and does not require additional setup. Tomcat is the reference implementation of the Java Servlet and JavaServer Pages Specifications. This implementation can be used in the Apache Web Server as well as in other web servers and development tools. For more information about Tomcat, check out `http://jakarta.apache.org`.

This tutorial assumes you are familiar with Java and with the JBuilder IDE. For more information on Java, see *Getting Started with Java.* For more information on the JBuilder IDE, see "The JBuilder environment" in *Introducing JBuilder*.

See the Tutorials section in the JBuilder Quick Tips for useful information about viewing and printing tutorials. The Accessibility options section in the JBuilder Quick Tips contains tips on using JBuilder features to improve JBuilder's ease of use for people with disabilities.

For information on documentation conventions used in this tutorial and other JBuilder documentation, see "Documentation conventions" on page 1-4.

Step 1: Creating a new project

1. Select File | New Project to display the Project wizard.
2. In the Name field, enter a project name, such as `jsptutorial`.
3. Click Finish to close the Project wizard and create the project. You do not need to make any changes to the defaults on Steps 2 and 3 of the wizard.

A new project is created.

Step 2: Selecting a server

In this step, you will select the Tomcat 4.0 server to use as the server for this project.

1. Select Project | Project Properties. The Project Properties dialog box appears.
2. Click the Server tab.
3. Make sure the Single Server For All Services In Project radio button is selected.
4. Make sure that Tomcat 4.0 is selected in the server drop-down list.
5. Click OK.

Step 3: Creating a new WebApp

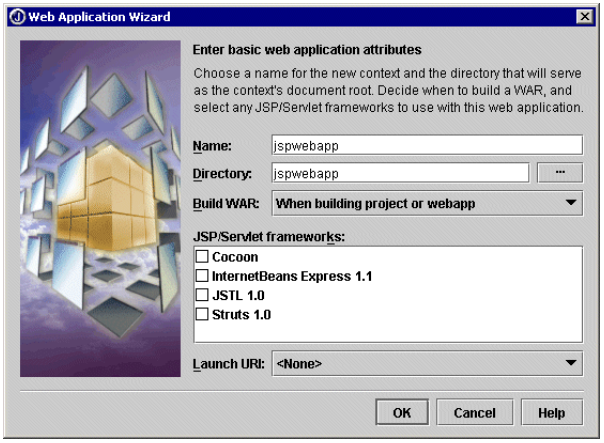
This step is optional, but advisable. You can use the default WebApp, but it's often less confusing to create a WebApp with a custom name. For more information on WebApps and WAR files, see Chapter 3, "Working with WebApps and WAR files."

1. Select File | New.
2. Click the Web tab. Select Web Application.
3. Click OK. The Web Application wizard appears.
4. Enter a name for the WebApp, such as `jspwebapp`. The Directory field is automatically filled in with the same name.

5 Leave the default setting for Build WAR, although you won't really need a WAR file since you probably won't want to actually deploy this tutorial application.

6 Leave all the JSP/Servlet frameworks unselected and don't specify a Launch URI.

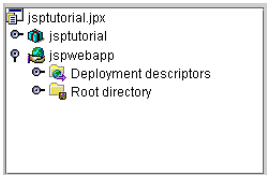
The wizard should look something like this:



7 Click OK to close the wizard.

A WebApp node, `jspwebapp` is displayed in the project pane. Expand the node to see the Root Directory and Deployment Descriptors nodes.

Figure 18.1 WebApp node in project pane



Step 4: Creating the JSP

In this step, you will use the JSP wizard to create the skeleton of a JSP.

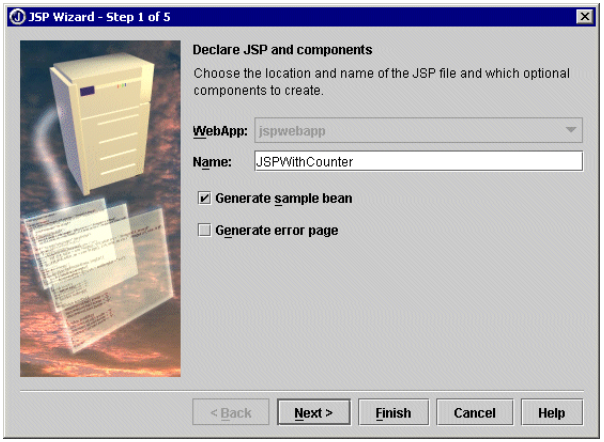
1 Select File | New.

2 Click the Web tab. Select JavaServer Page.

3 Click OK. The JSP wizard appears.

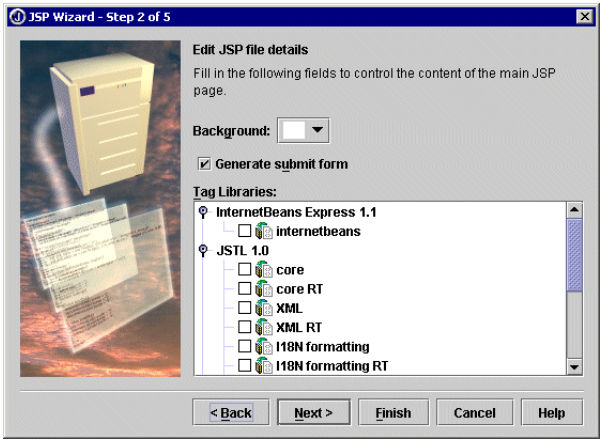
4 In the Name field, enter `JSPWithCounter`. This is the name of the JSP.

5 Check Generate Sample Bean and uncheck Generate Error Page. The wizard looks similar to this:



6 Click Next.

7 Check Generate Submit Form. The wizard looks similar to this:



8 Click Finish to accept all the default settings on the remaining pages of the wizard.

A `JSPWithCounter.jsp` file is added to the root directory of your WebApp. Expand the Root Directory node in the project pane to see it. A `JSPWithCounterBean.java` sample bean is also added to your project. This bean is called by the JSP. The JSP wizard also automatically creates a runtime configuration so that the JSP can be run in the IDE. For more information about runtime configurations, see "Setting runtime configurations" in *Building Applications with JBuilder*. For more information about creating a run configuration with a wizard, see "Creating a runtime configuration with the wizards" on page 10-2.

Step 5: Adding functionality to the JavaBean

At this point, a JSP and a JavaBean that can be used by the JSP have been created.

The next step in this tutorial is to create a method to count the number of hits to a web page.

1 Double-click `JSPWithCounterBean.java` in the project pane.

2 Modify the source code as follows, adding the code in **bold** to the existing code:

```
package jsptutorial;

public class JSPWithCounterBean {
  /**initialize variable here*/
  private int myCount=0;
  private String sample = "Start value";
  // Access sample property
  public String getSample() {
    return sample;
  }
  // Access sample property
  public void setSample(String newValue) {
    if (newValue!=null) {
      sample = newValue;
    }
  }
/**New method for counting number of hits*/
  public int count() {
    return ++myCount;
  }
}
```

3 Choose File | Save All to save your work.

Step 6: Modifying the JSP code

In this step, you'll modify the JSP code to count the number of hits.

1 Double-click `JSPWithCounter.jsp` in the project pane to open it in the editor. Remember it is in the Root Directory node of the WebApp.

2 Modify the generated file as follows, adding the code in **bold**. You can use CodeInsight and JSP source highlighting to help with coding.

```
<html>
<head>
<title>
JspWithCounter
</title>
</head>
```

```
<jsp:useBean id="jSPWithCounterBeanId" scope="session"
   class="jsptutorial.JSPWithCounterBean" />
<jsp:setProperty name="jSPWithCounterBeanId" property="*" />
<body>
<h1>
JBuilder Generated JSP
</h1>
<form method="post">
<br>Enter new value:  <input name="sample"><br>
<br><br>
<input type="submit" name="Submit" value="Submit">
<input type="reset" value="Reset">
<br>
Value of Bean property is: <jsp:getProperty name="JSPWithCounterBeanId"
   property="sample" />
<p>This page has been visited: <%= jSPWithCounterBeanId.count() %> times.</p>
</form>
</body>
</html>
```

3 Select File | Save All to save your work.

The line of code you just added uses a JSP expression tag to call the `count()` method of the `JSPWithCounterBean` class and insert the returned value in the generated HTML. For more information on JSP tags, see "JSP tags" on page 6-3.

Notice the `<jsp:useBean/>`, `<jsp:setProperty/>`, and `<jsp:getProperty/>` tags in the code above. These were added by the JSP wizard. The `useBean` tag creates an instance of the `JSPWithCounterBean` class. If the instance already exists, it is retrieved. Otherwise, it is created. The `setProperty` and `getProperty` tags let you manipulate properties of the bean.

The rest of the code that was generated by the JSP wizard is just standard HTML.

Step 7: Running the JSP

To run the JSP, the runtime properties for the project must be set correctly in Project | Project Properties | Run. In this tutorial, the runtime properties were already set by the JSP wizard, so you can go ahead and run the JSP.

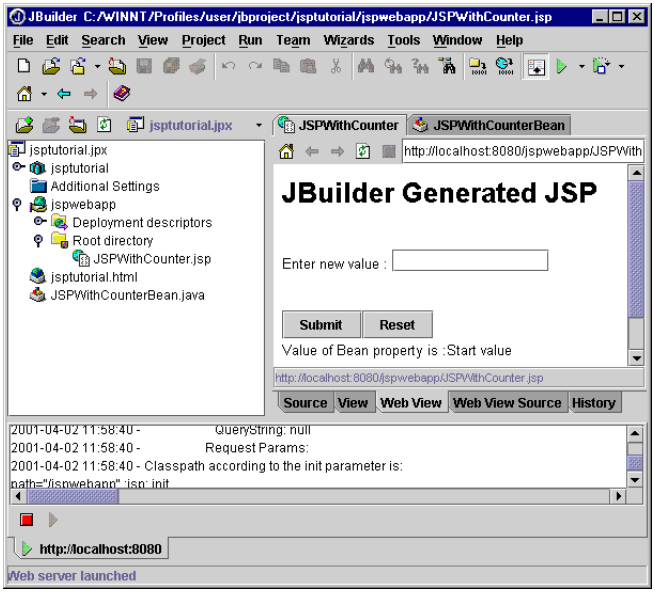
1 Right-click `JSPWithCounter.jsp` in the Root Directory of the WebApp node. Select Web Run Using "JSPWithCounter" from the menu.

Note The Web Run command includes the name of a runtime configuration, in this case, "JSPWithCounter". This runtime configuration was automatically created by the JSP wizard. By default, the JSP wizard names the runtime configuration the same name as the JSP. For more information about runtime configurations, see "Setting runtime configurations" in *Building Applications with JBuilder*.

The project compiles and runs. Compilation errors are displayed in the message pane. If there are errors, refer to "Web debugging your servlet or JSP" on page 10-17.

If there are no errors, the web server is started and two new tabs, the Web View and Web View Source, appear in the content pane. Tomcat, which is installed with JBuilder, is a servlet engine that supports servlets and JSP files. The web view is a web browser which displays output from the running JSP. The Web View Source tab displays the actual HTML code which has been dynamically generated by the JSP. If successful, the running JSP looks like this:

Figure 18.2 JSP in web view



The web view of the content pane displays the JSP. For local testing, the URL points to localhost:8080, which is where Tomcat is running. To test the JSP:

1. Enter any text in the text field.
2. Click the Submit button. The value you entered is displayed below the buttons and the page counter is incremented.

The output/log data from Tomcat appears on a new tab in the message pane. Output from servlets or beans, as well as HTTP commands and parameter values, are echoed to the message pane. The run configuration for a JSP may be edited or a new one may be created on the Run page of the Project Properties dialog box (Project | Project Properties). For more information about setting runtime properties for your JSP, see "Creating a server runtime configuration" on page 10-5. JBuilder uses port number 8080 by default. If port 8080 is in use, by default JBuilder will search for an unused port.

Using the Web View

The web view of the content pane displays the JSP file after it has been processed by the JSP engine. In this case the JSP engine is Tomcat. The web view behaves differently than the View tab. In the web view, there may be a delay between when the JSP file is edited and when the change is shown in the web view. To see the most recent changes to a JSP file, select the Refresh button in the web view's toolbar, just as you would in any web browser.

If you were debugging the JSP, you could press F9 to return the display to the web view.

Debugging the JSP

JSP debugging is a feature of JBuilder Enterprise

JSP's are compiled to servlets. In JBuilder, you can debug Java code snippets in the original JSP file, as opposed to debugging the corresponding generated Java servlet. For more information on debugging your JSP, see "Web debugging your servlet or JSP" on page 10-17.

Deploying the JSP

For deployment onto a production web server, consult the documentation for that web server for information on how to deploy JSPs to it. For general information on deploying JSPs, see Chapter 11, "Deploying your web application."

According to the JSP FAQ at `http://java.sun.com/products/jsp/faq.html`, there are a number of JSP technology implementations for different web servers. The latest information on officially-announced support can be found at java.sun.com.

Chapter

19

Tutorial: Creating a servlet with InternetBeans Express

Web Development is a feature of JBuilder Enterprise

This tutorial teaches you how to build a servlet using InternetBeans. When you are finished with the tutorial, you will have a servlet which uses a `DataModule` to query a table in a JDataStore, displays guest book comments in an `IxTable`, and allows visitors to the site to enter their own comments and see them displayed in the guest book. A finished version of the application created in this tutorial can be found in `<JBuilder>/samples/WebApps/guestbook`.

This tutorial assumes you are familiar with Java and Java servlets, the JBuilder IDE, and JDataStore. For more information on Java, see *Getting Started with Java.* For more information on Java servlets, see Chapter 4, "Working with servlets." For more information on the JBuilder IDE, see "The JBuilder environment" in *Introducing JBuilder*. For more information on JDataStore, see *JDataStore Developer's Guide.*

See the Tutorials section in the JBuilder Quick Tips for useful information about viewing and printing tutorials. The Accessibility options section in the JBuilder Quick Tips contains tips on using JBuilder features to improve JBuilder's ease of use for people with disabilities.

For information on documentation conventions used in this tutorial and other JBuilder documentation, see "Documentation conventions" on page 1-4.

Step 1: Creating a new project

1. Select File | New Project to display the Project wizard.
2. In the Name field, enter a Project name, such as `guestbooktutorial`.
3. Click Finish to close the Project wizard and create the project. You do not need to make any changes to the defaults on Steps 2 and 3 of the wizard.

A new project is created.

Step 2: Selecting a server

In this step, you will select the Tomcat 4.0 server to use as the server for this project.

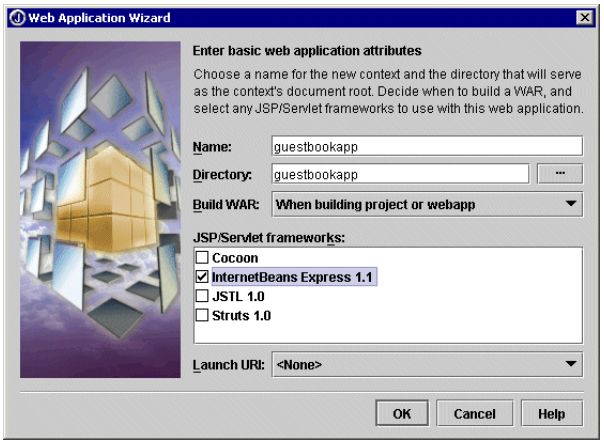
1. Select Project | Project Properties. The Project Properties dialog box appears.
2. Click the Server tab.
3. Make sure the Single Server For All Services In Project radio button is selected.
4. Make sure that Tomcat 4.0 is selected in the server drop-down list.
5. Click OK.

Step 3: Creating a new WebApp

This step is optional, but advisable. You can use the default WebApp, but it's often less confusing to create a WebApp with a custom name. For more information on WebApps and WAR files, see Chapter 3, "Working with WebApps and WAR files."

1. Select File | New.
2. Click the Web tab of the object gallery. Select Web Application.
3. Click OK. The Web Application wizard appears.
4. Enter a name for the WebApp, such as `guestbookapp`. The Directory field is automatically filled in as you type.
5. Leave the default setting for Build WAR, although you won't really need a WAR file since you probably won't want to actually deploy this tutorial application.
6. Check the InternetBeans Express 1.1 JSP/Servlet framework. Don't specify a Launch URI.

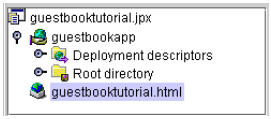
7 The wizard should look something like this:



8 Click OK.

A WebApp node, `guestbookapp`, is displayed in the project pane. Expand the node to see the Root Directory and Deployment Descriptors nodes.

Figure 19.1 WebApp node in project pane



Step 4: Creating the servlet

In this step, you will create the servlet with the Servlet wizard.

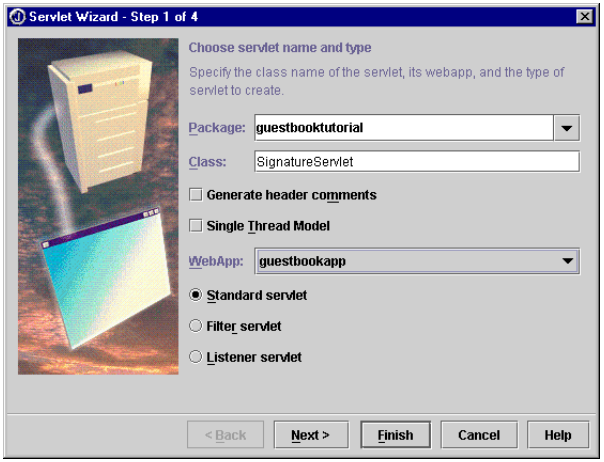
1 Select File | New.

2 Click the Web tab of the object gallery. Select Servlet.

3 Click OK. The Servlet wizard appears.

4 Enter a name for the class: `SignatureServlet`

5 Select `guestbookapp` for the WebApp, if it's not already selected. The wizard should look something like this:

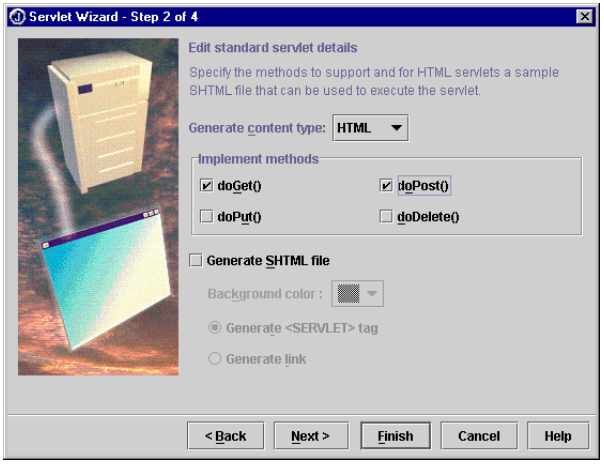


6 Click Next to proceed to the next step of the wizard.

7 Make sure the Generate Content Type option is set to HTML.

8 Make sure the methods `doGet()` and `doPost()` are checked.

9 Make sure Generate SHTML File is not checked. The wizard should look something like this:



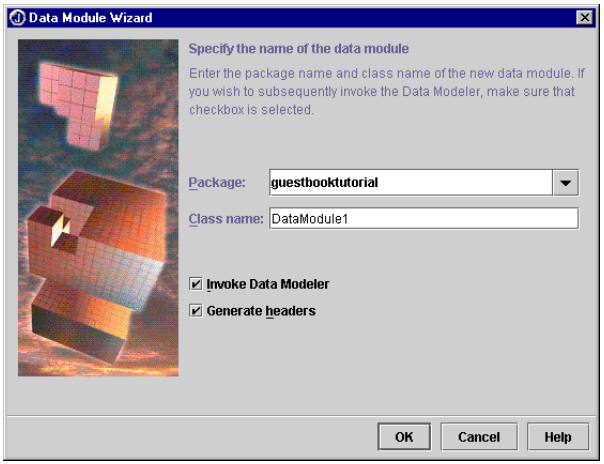
10 Click Finish. A `SignatureServlet.java` file is added to your project.

11 Click the Save All button on the toolbar to save your work.

Step 5: Creating the data module

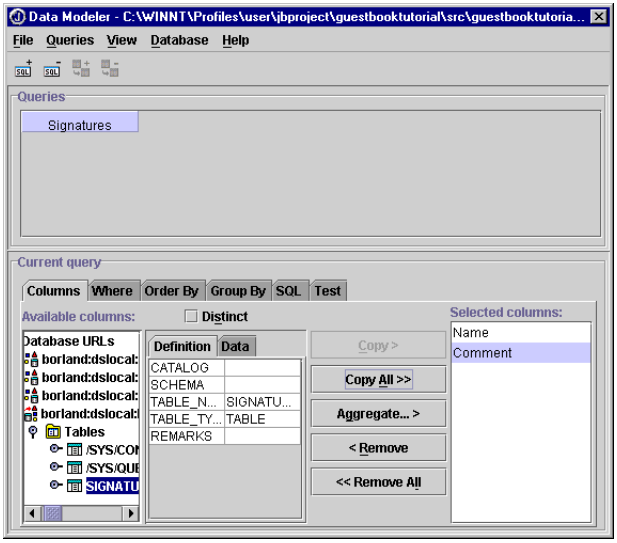
In this step, you'll use the Data Module wizard to create the data module that holds the database connection logic.

1. Select File | New.
2. Select Data Module from the General page of the object gallery.
3. Click OK. The Data Module wizard appears.



4. Leave the Package and Class Name fields set to the defaults.
5. Make sure the Invoke Data Modeler option is checked.
6. Click OK. The Data Modeler appears.
7. Go to the Database menu and select Add Connection URL.
8. Select `com.borland.datastore.jdbc.DataStoreDriver` from the driver drop-down list.
9. In the URL field, enter or browse to the path to the `guestbook.jds` file. It's found in the `<JBuilder>/samples/WebApps/guestbook` folder. Click OK to close the Create URL for Datastore dialog box.
10. Click OK again. The new Database URL is added to the Database URLs list on the lower left of the Data Modeler and is selected.
11. Double-click the URL and enter `user` in the login dialog box. A password isn't necessary. Click OK to close the dialog box.
12. Open the list of tables by clicking the Tables node of the Available Columns tree.
13. Select the SIGNATURES table by clicking it.

14 Click the Copy All button. The Data Modeler should look similar to this:



15 Choose Save from the Data Modeler File menu.

16 Choose Exit from the Data Modeler File menu. The `DataModule1.java` file is updated with the required connection information.

17 Click the Save All button on JBuilder's toolbar to save your work.

Step 6: Designing the HTML template page

In this step, you'll create an HTML page which can be used by InternetBeans as a template for the layout of the dynamic data.

1 Click the Add Files/Packages button on the project toolbar.

2 Click the Project button on the Explorer tab of the Add Files Or Packages To Project dialog box.

3 Select the directory for the WebApp (i.e. `guestbookapp`)

4 Type `gb1.html` in the File Name field.

5 Click OK.

6 Click OK again to create the file.

7 Double-click the file in the project pane to open it. The empty HTML file opens.

8 Click the Source tab to open the HTML source. It's blank at this point.

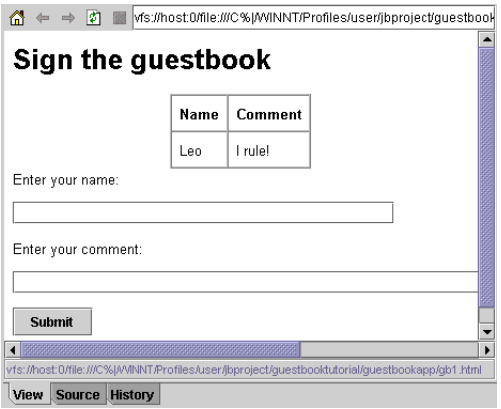
9 Type the following HTML code into the file. You can also copy and paste it from this tutorial.

```
<html>
<head>

<title>Guestbook Signatures</title>

</head>

<body>

<h1>Sign the guestbook</h1>

<table id="guestbooktable" align="CENTER" cellspacing="0"
    border="1" cellpadding="7">
<tr>
<th>Name</th><th>Comment</th>
</tr>
<tr>
<td>Leo</td><td>I rule!</td>
</tr>
</table>

<form method="POST">
<p>Enter your name:</p>

<input type="text" id="Name" name="Name" size="50">

<p>Enter your comment:</p>

<input type="text" id="Comment" name="Comment" size="100">
<p>
<input type="submit" name="submit" value="Submit"></p>
</form>

</body>
</html>
```

Notice that the `<table>` tag contains dummy data. This data will be replaced with live data from the JDataStore when the servlet is run. This dummy data provides an indication of how the live data should look when it is rendered. For more information on how InternetBeans Express renders tables, see "Generating tables" on page 7-6.

10 Click the Save All button on the toolbar.

11 Click the View tab. The HTML should look like this in the View:



Step 7: Connecting the servlet to the DataModule

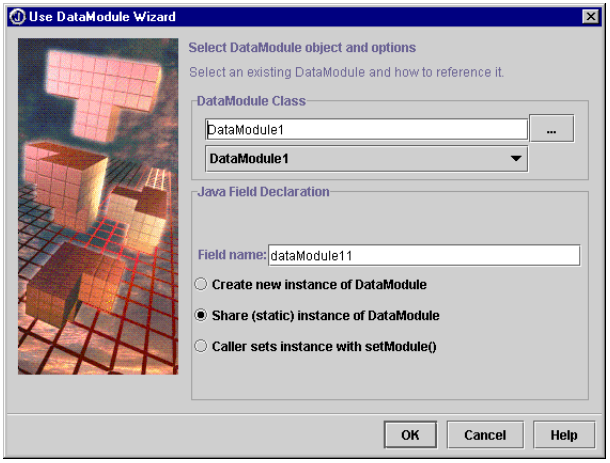
In this step you'll add a line of code to the servlet which enables it to use the `DataModule`.

1 Select Project | Make Project "guestbooktutorial.jpx". This builds the project so the `DataModule1.class` file is created.

2 Open the `SignatureServlet.java` file in the editor.

3 Select Wizards | Use DataModule. The Use DataModule wizard opens. The `DataModule1` class is already selected.

4 Make sure Share (Static) Instance Of DataModule is checked. The wizard should look something like this:



5 Click OK. A line of code is added to the `jbInit()` method to associate the `DataModule` with the servlet.

Step 8: Designing the servlet

In this step, you will use the designer to add InternetBeans components to the servlet. These components won't be visible in the designer, because the GUI for the servlet is actually in the HTML file. However, the properties of the components will be visible in the Inspector. When the servlet is run, the InternetBeans components you add in this step will replace the dummy data in the HTML file with data from the JDataStore.

1 Make sure `SignatureServlet.java` is open in the editor.

2 Click the Design tab to open the JBuilder designer.

3 Select the InternetBeans tab of the component palette.

4 Select the `IxPageProducer` icon, and drop an `IxPageProducer` into the servlet by clicking in the designer.

5 Set the properties of the `IxPageProducer` as follows:

Property	Value
`dataModule`	`DataModule11`
`htmlFile`	`gb1.html` — setting this property automatically fills in the `rootPath` property

6 Select the `IxControl` icon in the palette. Drop three `IxControls` into the servlet by clicking in the designer.

When you want to drop multiple instances of a control from the component palette into the designer, press *Shift* and click the icon for the control. This causes the control to remain selected. When you are finished with that control, click the selection tool on the component palette to deselect the choice.

7 Select `ixControl1` and set its properties as follows in the order shown:

Property	Value
`dataSet`	`Signatures`
`columnName`	`Name`
`pageProducer`	`ixPageProducer1`
`controlName`	`Name`

8 Select `ixControl2` and set its properties as follows:

Property	Value
dataSet	Signatures
columnName	Comment
pageProducer	ixPageProducer1
controlName	Comment

9 Select the `IxTable` icon from the palette. Drop an `IxTable` into the servlet by clicking in the designer.

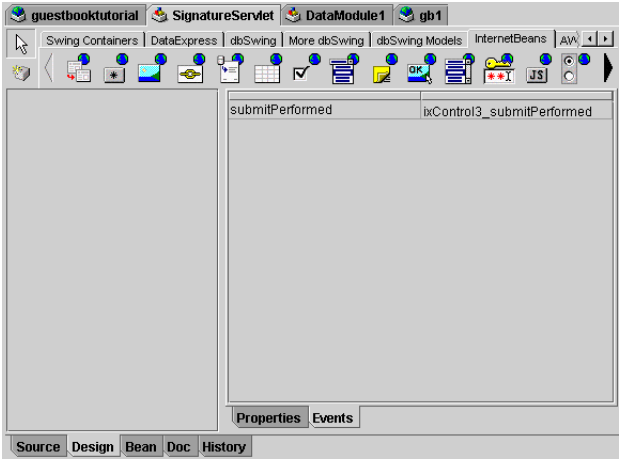
10 Set the properties of `ixTable1` as follows:

Property	Value
pageProducer	ixPageProducer1
dataSet	Signatures
elementId	guestbooktable

11 Select `ixControl3` and set its properties as follows:

Property	Value
pageProducer	ixPageProducer1
controlName	submit

12 Click the Events tab in the property inspector. The UI should look similar to this:



13 Click once in the `submitPerformed` property in the Inspector to set the `submitPerformed()` event to `ixControl3_submitPerformed`. This adds an event listener to `ixControl3`. Press Enter to generate the `ixControl3_submitPerformed()` method. This opens the editor and places the cursor in the new method.

14 Click the Save All button on the toolbar.

Step 9: Editing the servlet

1 Make sure the `SignatureServlet.java` file is open in the editor.

2 Remove the wizard-generated body of the servlet's `doGet()` method.

3 Remove the wizard-generated body of the servlet's `doPost()` method.

4 Type the following line of code into the body of the `doGet()` method:
`ixPageProducer1.servletGet(this, request, response);`
The `doGet()` method is now complete. Often calling the `servletGet()` method of the `IxPageProducer` is all you need to do here.

5 Type the following lines of code into the body of the `doPost()` method:

```
DataModule1 dm =
    (DataModule1)
        ixPageProducer1.getSessionDataModule(request.getSession());
dm.getSignatures().insertRow(false);
ixPageProducer1.servletPost(this, request, response);
doGet(request, response);
```

When the form is posted, this code gets a per-session instance of the `DataModule`, inserts an empty row, calls `IxPageProducer.servletPost()` to fill in the empty row with the values the user typed, then calls `doGet()` again to display the data that was posted.

6 Next you need to fill in the body of the `ixControl3_submitPerformed()` method. This method is called by the `servletPost()` method. Type the following code into the body of the `ixControl3_submitPerformed()` method:

```
DataModule1 dm =
    (DataModule1) ixPageProducer1.getSessionDataModule(e.getSession());
dm.getSignatures().post();
dm.getSignatures().saveChanges();
```

This code gets a per-session instance of the `DataModule` and posts and saves the user's input to the JDataStore. Note that this per-session instance is different from the shared instance stored in the variable `dataModule11`.

7 Click the Save All button on the toolbar.

Step 10: Setting dependencies for the WebApp

In this step, you'll set the dependencies of the WebApp to make sure the necessary libraries are included.

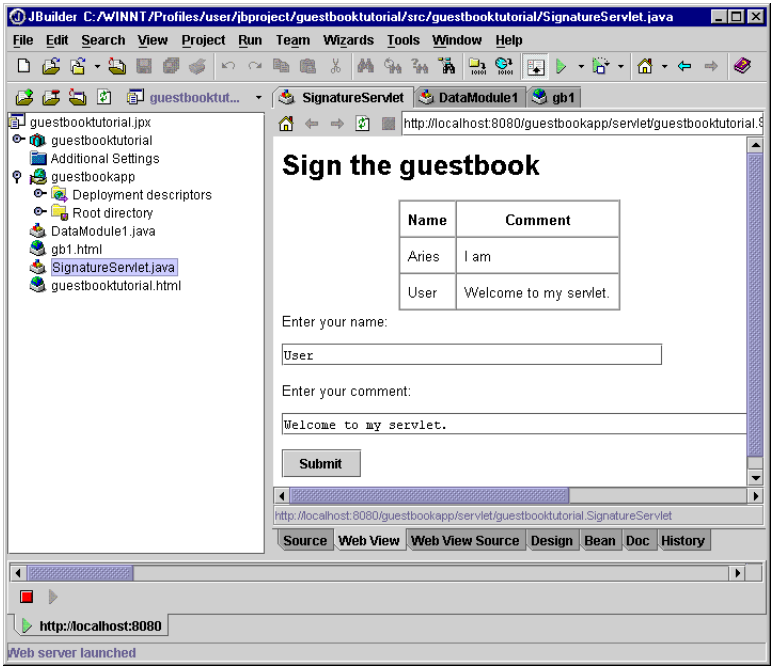
1. Right-click the `guestbookapp` node in the project pane.
2. Select Properties from the context menu for the WebApp.
3. Click the Dependencies tab of the Properties dialog box.
4. Make sure the dependencies for the following libraries are set to Include All:
   - Data Express
   - JDataStore Server
   - InternetBeans Express
   - dbSwing
5. Click OK.

Step 11: Running the servlet

In this step you'll run the servlet and test it.

1. Right-click the `SignatureServlet.java` file in the project pane.
2. Select Web Run Using "SignatureServlet" from the menu. The servlet runs in the JBuilder IDE.
3. Test the servlet by removing the existing values from the Name and Comment fields. Enter your name and comment and click Submit. Your

name and comment are displayed in the table and saved to the JDataStore.



4 Stop the servlet by clicking the Reset Program button on the Web Server tab in the message pane.

Deploying the servlet

For information on deploying your servlet, see Chapter 11, "Deploying your web application."

Chapter

20

Tutorial: Creating a JSP with InternetBeans Express

Web Development is a feature of JBuilder Enterprise

This tutorial teaches you how to build a JSP containing InternetBeans. When you are finished with the tutorial, you will have a JSP which queries a table in a JDataStore, displays guest book comments in an `IxTable`, and allows visitors to the site to enter their own comments and see them displayed in the guest book. A finished version of the application created in this tutorial can be found in `<jbuilder>/samples/WebApps/jspinternetbeans`.

This tutorial assumes you are familiar with Java and JavaServer Pages (JSP), the JBuilder IDE, and JDataStore. For more information on Java, see *Getting Started with Java.* For more information on JavaServer Pages, see Chapter 6, "Developing JavaServer Pages." For more information on the JBuilder IDE, see "The JBuilder environment" in *Introducing JBuilder.* For more information on JDataStore, see *JDataStore Developer's Guide*

See the Tutorials section in the JBuilder Quick Tips for useful information about viewing and printing tutorials. The Accessibility options section in the JBuilder Quick Tips contains tips on using JBuilder features to improve JBuilder's ease of use for people with disabilities.

For information on documentation conventions used in this tutorial and other JBuilder documentation, see "Documentation conventions" on page 1-4.

Step 1: Creating a new project

1. Select File | New Project to display the Project wizard.
2. In the Name field, Enter a project name, such as `jspixtutorial`.
3. Click Finish to close the Project wizard and create the project. You do not need to make any changes to the defaults on Steps 2 and 3 of the wizard.

A new project is created.

Step 2: Selecting a server

In this step, you will select the Tomcat 4.0 server to use as the server for this project.

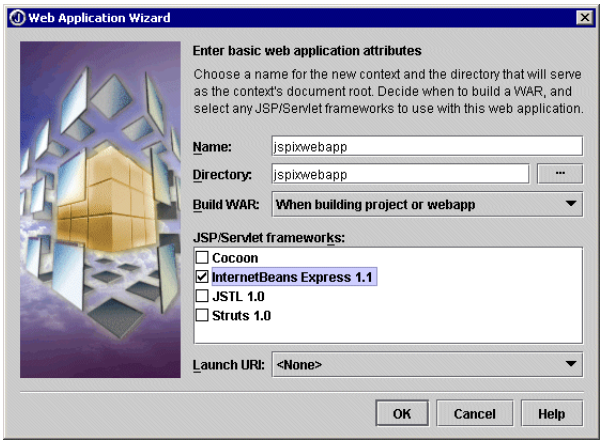
1. Select Project | Project Properties. The Project Properties dialog box appears.
2. Click the Server tab.
3. Make sure the Single Server For All Services In Project radio button is selected.
4. Make sure that Tomcat 4.0 is selected in the server drop-down list.
5. Click OK.

Step 3: Creating a new WebApp

This step is optional, but advisable. You can use the default WebApp, but it's often less confusing to create a WebApp with a custom name. For more information on WebApps and WAR files, see Chapter 3, "Working with WebApps and WAR files."

1. Select New from the File menu.
2. Click the Web tab of the object gallery. Select Web Application.
3. Click OK. The Web Application wizard appears.
4. Enter a name for the WebApp, such as `jspixwebapp`. The Directory field is automatically filled in as you type.
5. Leave the default setting for Build WAR, although you won't really need a WAR file since you probably won't want to actually deploy this tutorial application.
6. Check the InternetBeans Express 1.1 JSP/Servlet framework. Don't specify a Launch URI.

The wizard should look something like this:



7 Click OK.

A WebApp node, `jspixwebapp` is displayed in the project pane. Expand the node to see the Root Directory and Deployment Descriptors nodes.

Figure 20.1 WebApp node in project pane

Step 4: Using the JSP wizard

In this step, you will create the skeleton of a JSP using the JSP wizard.

1 Select File | New.

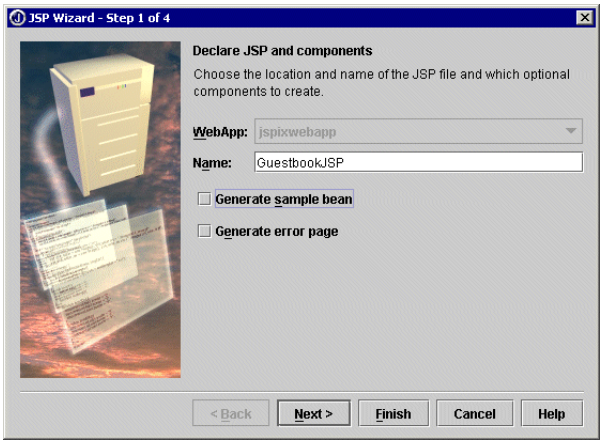
2 Click the Web tab. Select JavaServer Page.

3 Click OK. The JSP wizard appears.

4 In the name field, enter a name for the JSP: `GuestbookJSP`

5 Uncheck Generate Sample Bean.

The JSP wizard should look similar to this:

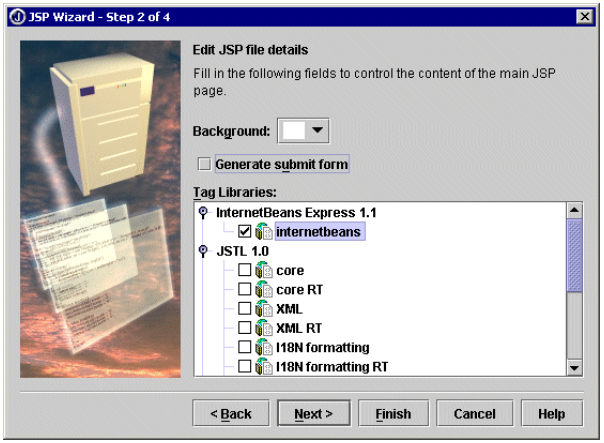


6 Click Next.

7 Uncheck Generate Submit Form.

8 Check internetbeans under InternetBeans Express 1.1 in the Tag Libraries tree.

The JSP Wizard should look similar to this:



9 Click Finish. A `GuestbookJSP.jsp` file is added to the Root Directory node of your WebApp in the project pane. Expand the Root Directory node to see the file. The JSP contains the page directive and taglib directive required for using the InternetBeans tag library. The JSP wizard takes care of the necessary steps for adding the InternetBeans library to your project, as described in "Using InternetBeans Express with JSPs" on page 7-6. The JSP wizard also automatically creates a runtime configuration so that the JSP can be run in the IDE. For more information about runtime configurations, see "Setting runtime

configurations" in *Building Applications with JBuilder*. For more information about creating a run configuration with a wizard, see "Creating a runtime configuration with the wizards" on page 10-2.

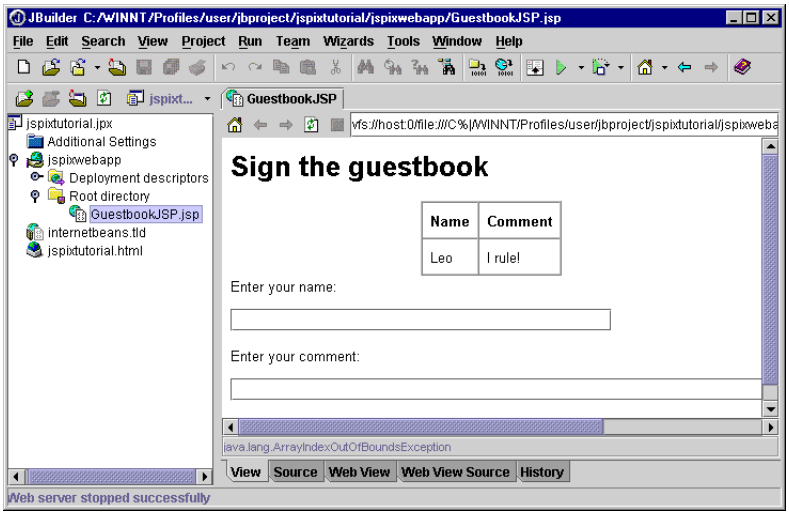
Step 5: Designing the HTML portion of the JSP

In this step you will add some HTML code to the `GuestbookJSP.jsp` file. The HTML you add provides a template for the way the data is displayed when the JSP is running.

1. Open the `GuestbookJSP.jsp` file in the editor, if it's not already open. It's in the Root Directory node of the WebApp in the project pane.
2. Change the contents of the `<title>` tag to read `JSP/InternetBeans Tutorial`.
3. Change the contents of the `<h1>` tag to read `Sign the Guestbook`
4. Type the following HTML code into the body of the file, below the `</h1>` tag:

```
<table id="guestbooktable" align="CENTER" cellspacing="0"
    border="1" cellpadding="7">
<tr>
<th>Name</th><th>Comment</th>
</tr>
<tr>
<td>Leo</td><td>I rule!</td>
</tr>
</table>

<form method="POST">
<p>Enter your name:</p>

<input type="text" name="Name" size="50">

<p>Enter your comment:</p>

<input type="text" name="Comment" size="100">
<p>
<input type="submit" name="submit" value="Submit"></p>
</form>
```

When you are finished, the HTML should look like this in the View tab:



Click the Save All button on the toolbar.

Step 6: Adding the InternetBeans database tag

In this step you will add an InternetBeans `database` tag to the JSP. The `database` tag provides connection information for a data source.

1. If you switched to the View tab in the previous step, switch back to the editor.
2. Add the opening `database` tag shown in **bold**. Change the value of the `url` attribute of the `database` tag to point to the `guestbook.jds` JDataStore in `<jbuilder>\samples\WebApps\guestbook`.

```
<h1>
Sign the guestbook
</h1>

<ix:database id="database1"
    driver="com.borland.datastore.jdbc.DataStoreDriver"
        url="jdbc:borland:dslocal:jbuilder\\samples\\WebApps\\guestbook\\
    guestbook.jds"
username="user">

<table id="guestbooktable" align="CENTER" cellspacing="0"
    border="1" cellpadding="7">
```

3. Add the closing `database` tag shown in **bold**.

```
</form>
</ix:database>
</body>
```

Click the Save All button on the toolbar.

Step 7: Adding the InternetBeans query tag

In this step you will add a `query` tag to the JSP. The `query` tag specifies an SQL query statement.

1 Add the opening `query` tag shown in **bold**.

```
<ix:database id="database1"
driver="com.borland.datastore.jdbc.DataStoreDriver"
url="jdbc:borland:dslocal:jbuilder\\samples\\WebApps\\guestbook\\
guestbook.jds"
username="user">

<ix:query id="signatures" statement="select * from signatures">

<table id="guestbooktable" align="CENTER" cellspacing="0" border="1"
cellpadding="7">
```

Note that you are nesting the `query` tag within the `database` tag. This is so that the `database` attribute of the `query` tag doesn't need to be specified, since it's implied. It also makes the code more elegant.

2 Add the closing `query` tag shown in **bold**.

```
</ix:query>

</ix:database>
```

Click the Save All button on the toolbar.

Step 8: Adding the InternetBeans table tag

1 Add the opening and closing `table` tags shown in **bold**.

```
<ix:query id="signatures" statement="select * from signatures">

<ix:table dataSet="signatures">

<table id="guestbooktable" align="CENTER" cellspacing="0" border="1"
cellpadding="7">
<tr>
<th>Name</th><th>Comment</th>
</tr>
<tr>
<td>Leo</td><td>I rule!</td>
</tr>
</table>

</ix:table>

<form method="POST">
```

Note that you are wrapping the HTML `table` tag in the InternetBeans `table` tag. This allows the InternetBeans `IxTable` to implicitly understand which table it is replacing. The InternetBeans `table` tag is nested within the InternetBeans `query` tag. This isn't required, because the table's `dataSet` attribute makes the relationship clear. Nesting the InternetBeans table within the `query` tag like this just makes the code more elegant.

2 Click the Save All button on the toolbar.

Step 9: Adding the InternetBeans control tags

Now it's time to add the two `control` tags for the two text input fields. The `control` tags allow dynamic data to be supplied in place of HTML controls which are used as placeholders for the data. Add the tags shown in **bold**.

```
<form method="POST">
<p>Enter your name:</p>

<ix:control dataSet="signatures" columnName="Name">

<input type="text" name="Name" size="50">

</ix:control>

<p>Enter your comment:</p>

<ix:control dataSet="signatures" columnName="Comment">

<input type="text" name="Comment" size="100">

</ix:control>

<p>
```

Note that you are wrapping each of the HTML text `input` tags in an InternetBeans `control` tag. This allows the InternetBeans `IxControls` to implicitly understand which text input fields they are replacing.

Click the Save All button on the toolbar.

Step 10: Adding the InternetBeans submit tag

Add the opening and closing `submit` tags shown in **bold**.

```
<input type="text" name="Comment" size="100">
</ix:control>

<p>
<ix:submit methodName="submitPerformed">

<input type="submit" name="submit" value="Submit"></p>

</ix:submit>

</form>
```

Note that you are wrapping the HTML submit `input` tag in an InternetBeans `submit` tag. This allows the InternetBeans `IxSubmitButton` to implicitly understand which submit button it is replacing.

Click the Save All button on the toolbar.

Step 11: Adding the submitPerformed() method

We're not done with the submit button yet. You still have to add the method which will be executed when the button is pushed. You'll do that in this step. Add the code shown in **bold**.

```
<ix:submit methodName="submitPerformed">

<%!
    public void submitPerformed(PageContext pageContext){
      DataSet signatures = (DataSet) pageContext.findAttribute( "signatures" );
      signatures.post();
      signatures.saveChanges();
    }
%>

<input type="submit" name="submit" value="Submit"></p>

</ix:submit>
```

The `submitPerformed()` method is contained within a JSP declaration tag. This method declaration doesn't have to be nested within the InternetBeans `submit` tag, but it is an elegant way of writing the code. The parameter passed to this method is the `PageContext`. This is an object containing information about the page, which exists for every JSP. The method retrieves a DataExpress `DataSet` by finding the page attribute corresponding to the "signatures" dataset. It then posts the user's input to the dataset, and saves the changes to the dataset.

Click the Save All button on the toolbar.

Step 12: Adding code to insert a row

There is still one more piece of code you need to add before the JSP will work properly. When the form is posted, you need to add an empty row to the dataset to contain the user's input. Add the code shown in **bold**.

```
</ix:table>

<%
   signatures.insertRow(false);
%>

<form method="POST">
```

This last Java code fragment may look a little confusing, because it doesn't appear to be enclosed in a method declaration. It actually is. When the JSP gets compiled this will become part of the `service()` method in the generated servlet (which you can't see, but it's still there). Any line of code within a JSP scriptlet tag such as this will become part of the `service()` method.

This code fragment inserts a row in the dataset just before the form is displayed. The form displays empty values. Then when the form is posted the data is written to the empty row before calling the `submitPerformed()` method.

Click the Save All button on the toolbar.

Step 13: Adding the JDataStore Server library to the project

This project requires the JDataStore Server library. To add this library to the Project Properties:

1. Select Project Properties from the Project menu.
2. Click the Paths tab.
3. Click the Required Libraries tab.
4. Click Add.
5. Select JDataStore Server from the JBuilder folder. Click OK.
6. Click OK again to close the Project Properties dialog box. At this point you should make sure the dependencies for the libraries in your project are set correctly. To do this:
7. Right click the `jspixwebapp` node in the project pane and select Properties from the context menu.
8. Click the Dependencies tab of the Properties dialog box.

9 Make sure the dependencies for the following libraries are set to Include All:
   - InternetBeans Express
   - dbSwing
   - Data Express
   - JDataStore Server

10 If any of these libraries are not set to Include All, select the library, then select the Always Include All Classes And Resources radio button. The text next to the library name changes to Include All. Once all the library dependencies are set correctly click OK.

Click the Save All button on the toolbar.

Step 14: Running the JSP

Now it's time to run and test the JSP.

1 Make sure the Root Directory node of the WebApp is expanded in the project pane.

2 Right-click `GuestbookJSP.jsp` in the project pane.

3 Select Web Run Using "GuestbookJSP" from the menu.

   Tomcat is started and the JSP runs within the JBuilder IDE.

Note The Web Run command includes the name of a runtime configuration, in this case, "GuestbookJSP". This runtime configuration was automatically created by the JSP wizard. By default, the JSP wizard names the runtime configuration the same name as the JSP. For more information about runtime configurations, see "Setting runtime configurations" in *Building Applications with JBuilder*.

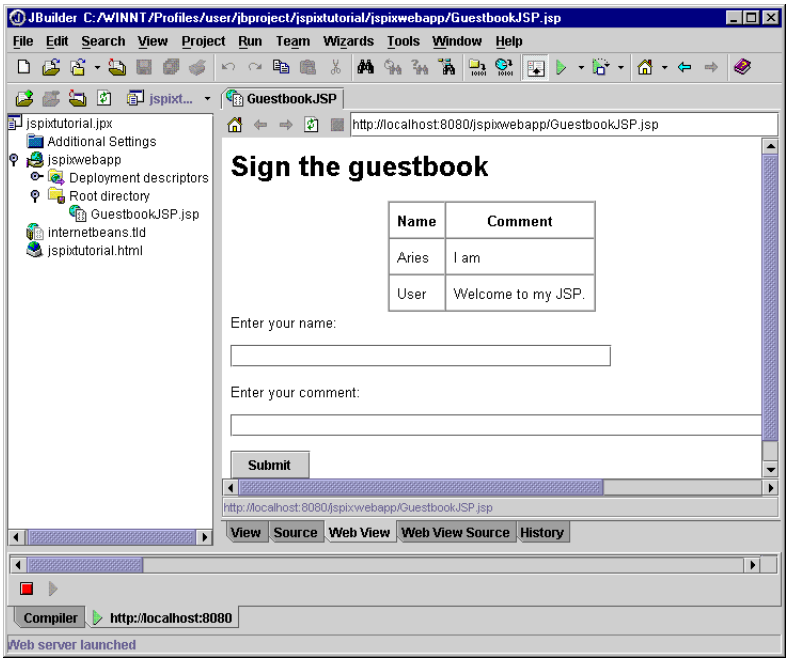
4 Select the URL shown at the top of the Web View and copy it. Paste it into the URL field in the full-featured browser of your choice and go to the URL.

Note JBuilder's Web View offers some browser capabilities, but it is not a full-featured browser. For best results, copy the URL of a running web application to an external browser and run it there.

5 Enter your name and comment in the JSP running in your external browser.

6 Click the Submit button in the external browser. Your name and comment are added to the table (and stored in the JDataStore).

Figure 20.2 JSP running in the Web View



Deploying the JSP

JSPs are easier to deploy than servlets. This is because a web server finds them in the same way it finds HTML files. You don't have to do special installation, because it's up to the web server to know what to do with the JSP. For more information on deploying your JSP, see Chapter 11, "Deploying your web application."

Chapter

21

Tutorial: Running the CheckBoxControl sample application with Java Web Start

This tutorial walks you through the steps of launching a Swing-based sample application with Web Start. The sample, `CheckBoxControl`, is located in the `samples/Swing` directory of your JBuilder installation. This tutorial assumes that Java Web Start is installed on your computer. For installation instructions, see "Installing Java Web Start" on page 15-3.

For information about JBuilder and Java Web Start, see Chapter 15, "Launching your web application with Java Web Start."

See the Tutorials section in the JBuilder Quick Tips for useful information about viewing and printing tutorials. The Accessibility options section in the JBuilder Quick Tips contains tips on using JBuilder features to improve JBuilder's ease of use for people with disabilities.

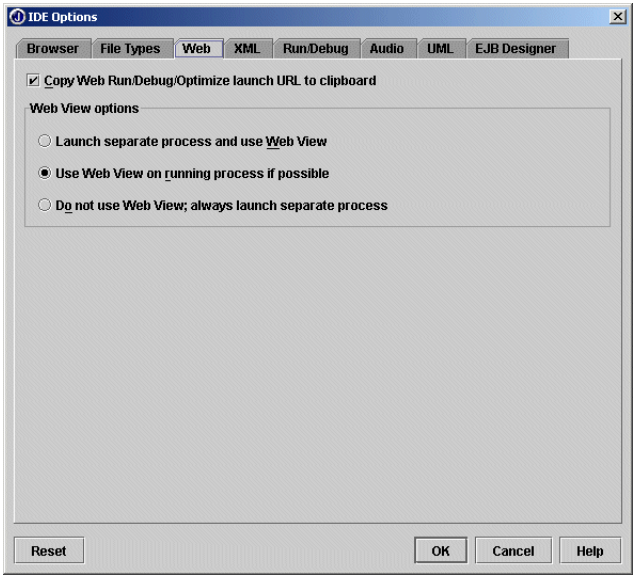
For information on documentation conventions used in this tutorial and other JBuilder documentation, see "Documentation conventions" on page 1-4.

Step 1: Opening and setting up the project

First, you'll open the project in JBuilder and set web view IDE options. To do this,

1. Choose File | Open Project to display the Open Project dialog box.
2. In the Open Project dialog box, click the Samples button. Browse to `Swing/CheckBoxControl`. Click `CheckBoxControl.jpx` and click OK to open the project.
3. Choose Tools | IDE Options to open the IDE Options dialog box. On the Web page, make sure the Copy Web Run/Debug Launch URL To Clipboard option is selected. This option copies the URL generated by a web run into the clipboard, so you can paste it directly into an external browser.

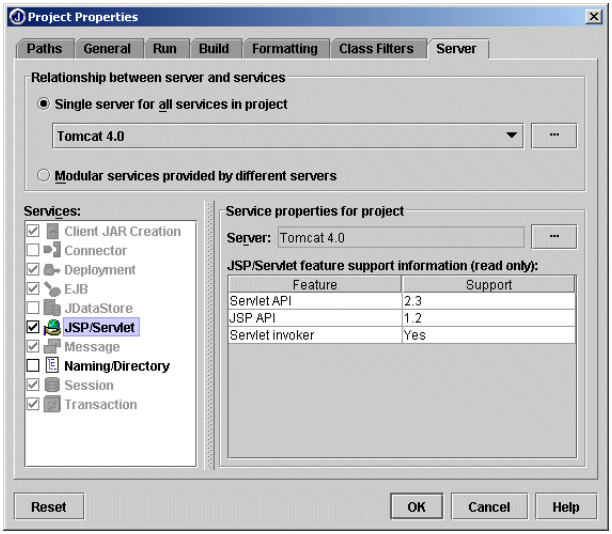
   The Web page of the IDE Options dialog box looks like this:



4. Click OK to close the dialog box.
5. Choose Project | Project Properties to open the Project Properties dialog box. Choose the Server tab.
6. Choose Tomcat 4.0 from the Single Server For All Services In Project drop-down list.

7 Make sure only the JSP/Servlet service is selected in the Services list on the left side of the dialog box.

The Server page, with the JSP/Servlet service selected, looks like this:



8 Click OK to close the dialog box.

To see what this application does, you can run it by choosing Run | Run Project. The application demonstrates how to use Swing's CheckBox control. (It is not using Web Start at this point.)

Note that this application does not contain any file handling operations, recommended for running with Java Web Start. A WebStart application, because it's deployed like an applet, doesn't have the right to do file handling operations unless you digitally sign the JAR file or use the JNLP classes. For more information about Java Web Start and security issues, see "Considerations for Java Web Start applications" on page 15-2.

Step 2: Creating the application's WebApp

To create a WebApp, use the Web Application wizard. For more information on WebApps, see Chapter 3, "Working with WebApps and WAR files."

To create the application's WebApp,

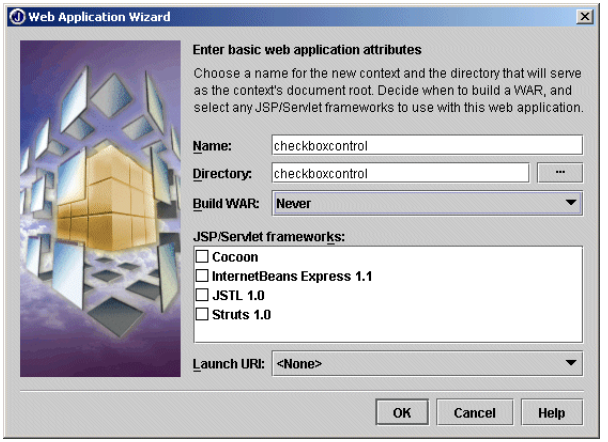
1 Choose File | New to display the object gallery. On the Web page, choose Web Application and click OK.

The Web Application wizard is displayed.

2 Enter `checkboxcontrol` in the Name field. The Directory field is filled in as you type.

3 Set the Build WAR option to Never. Do not choose a JSP/Servlet framework.

The Web Application wizard should look like this:



4 Click OK to close the wizard.

The WebApp `checkboxcontrol` is displayed in the project pane as a node. Expand the node to see the Deployment Descriptor and the Root Directory nodes.

Step 3: Creating the application's JAR file

In order to launch an application as a Web Start application, you need to create a JAR file. You use JBuilder's Archive Builder to create JAR files:

1 Compile the project. Choose Project | Make Project "CheckBoxControl.jpx."

2 Choose Wizards | Archive Builder.

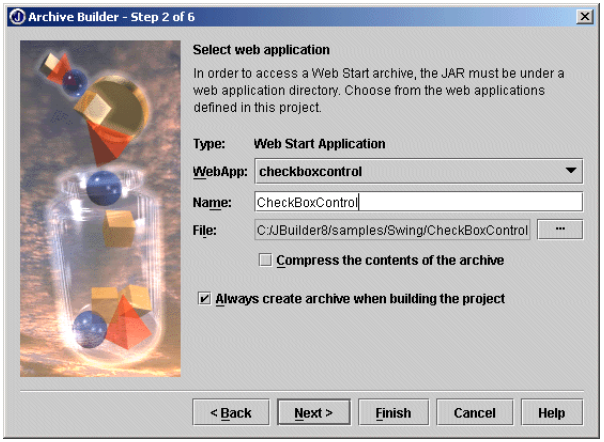
3 Change the Archive Type set to Web Start Application on the Select An Archive Type page of the Archive Builder. Click Next.

4 Make sure `checkboxcontrol` is selected in the WebApp drop-down list on the Select Web Application page.

5 Change the Name to `CheckBoxControl`.

The File field has been filled in for you. The JAR file name is based on the project name. The JAR file is placed in the `samples/Swing/CheckBoxControl/webapp` folder of your project.

Step 2 of the Archive Builder looks like this:



6 Click Finish to create the archive and close the wizard. You do not need to change any options on the remaining steps of the wizard.

7 Choose File | Save All.

8 Choose Project | Make Project "CheckBoxControl.jpx" to create the JAR file.

The wizard creates an archive node and displays it in the project pane. The archive will be built each time you build the project.

Step 4: Creating the application's homepage and JNLP file

In this step, you'll use the Web Start Launcher wizard to create the application's homepage and JNLP file. The homepage is an HTML file that you load into your external web browser. It contains a link to your application — when you click the link, the JNLP file instructs Java Web Start to launch your application.

To create these files,

1 Choose File | New to display the object gallery.

2 Choose Web Start Launcher and click OK on the Web page.

The Web Start Launcher wizard is displayed.

3. Enter `CheckBoxControlLauncher` in the Name field.

   This option names the HTML file and the JNLP file.

4. Make sure `checkboxcontrol` is selected in the WebApp drop-down list.

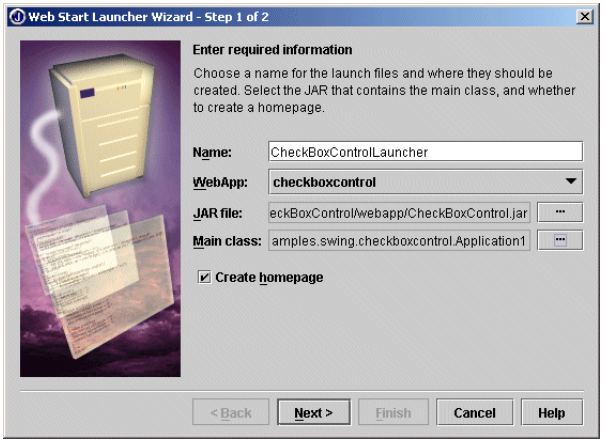
5. Click the ellipsis (...) button to the right of the JAR File field. This opens the Choose JAR For WebStart dialog box where you choose the name of the JAR file you created with the Archive Builder. This is `CheckBoxControl.jar`. It is in the `CheckBoxControl/webapp` directory. Select the JAR file in the Choose JAR For WebStart dialog box, then click OK to close it.

6. Click the ellipsis (...) button to the right of the Main Class field if it is not already filled in. This displays the Select Main Class dialog box. Choose the Browse tab. Expand the `com` folder at the top of the dialog box to choose `com.borland.samples.swing.checkboxcontrol.Application1`. Click OK to close the dialog box.

7. Make sure the Create Homepage option is checked on the Web Start Launcher wizard. This option creates the HTML file that launches the application.

Warning If the name entered in the Name field matches the name of an existing HTML or JNLP file in your project, you are asked if you want to overwrite the existing file.

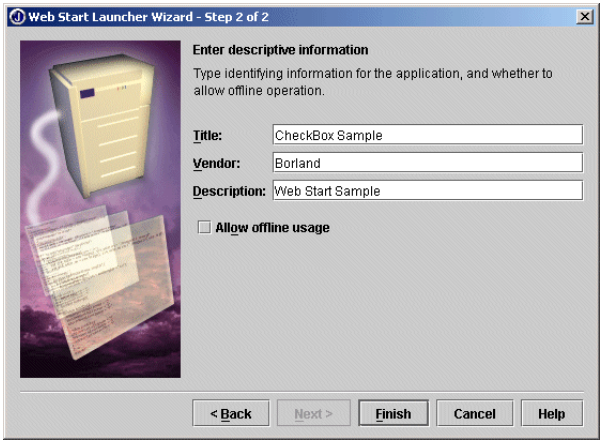
The Enter Required Information page of the Web Start Launcher wizard looks like this:



8. Click Next.

9 Enter `CheckBox Sample` in the Title field. Enter `Borland` in the Vendor field and `Web Start Sample` in the Description field. Make sure the Allow Offline Usage option is not selected.

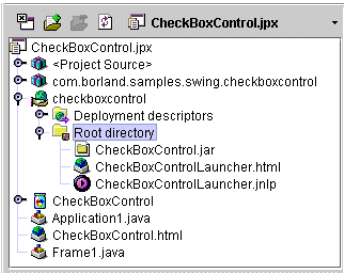
The Enter Descriptive Information page of the Web Start Launcher wizard looks like this:



10 Click Finish.

The wizard creates an HTML file and a JNLP file called `CheckBoxControlLauncher` and places them in root directory of your web application. To see these files in the project pane, expand the Root Directory node of the `checkboxcontrol` WebApp node.

The project pane should look similar to this:

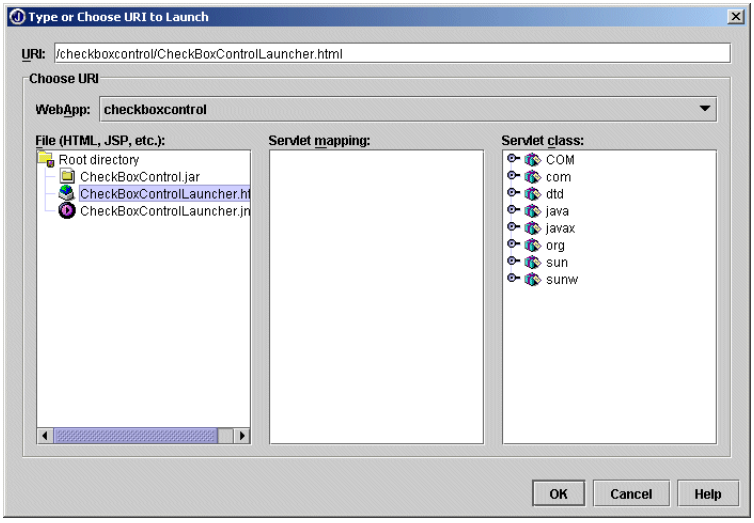


Note You can open these files in the editor; however, do not change them.

Step 5: Creating a server runtime configuration

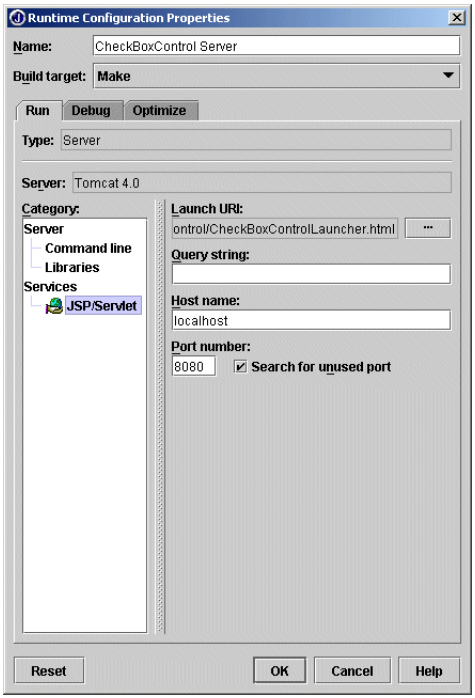
In this step, you will create a server runtime configuration. This allows the sample to run using a server.

1. Select Run | Configurations. The Run page of the Project Properties dialog box appears.
2. Click the New button to display the Runtime Configuration Properties dialog box.
3. Enter `CheckBoxControl Server` in the Name field.
4. Choose Server from the Type drop-down list.
5. Click JSP/Servlet in the Category list on the left side of the dialog box.
6. Click the ellipsis (...) button next to the Launch URI dialog box to display the Type Or Choose Launch URI dialog box.
7. In the File list on the left side of the dialog box, choose `CheckBoxControlLauncher.html`. Note that this file is in the root directory of the web application. The Type Or Choose Launch URI dialog box will look like this:



8. Click OK to close the dialog box.

9 The Runtime Configuration Properties dialog box will look like this:



10 Click OK two times to close the Runtime Configuration Properties and the Project Properties dialog boxes.

For more information, see "Creating a server runtime configuration" on page 10-5.

Step 6: Launching the application

This step tells you how to launch your application with Web Start. To do this,

1 Right-click `CheckBoxControlLauncher.html` in the project pane and choose Web Run Using "CheckBoxControl Server." (It is in the Root Directory folder of the WebApp node in the project pane.)

JBuilder compiles files and starts the Tomcat web server. Because the JNLP file specifies that this application is to be run with Web Start,

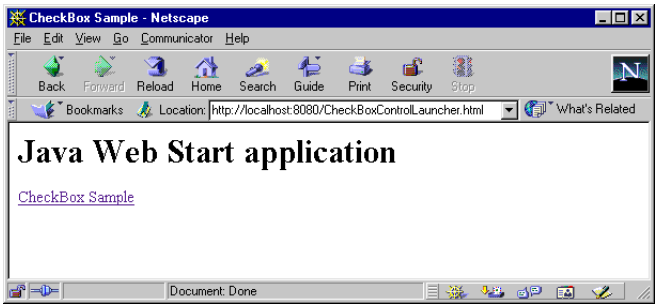
JBuilder displays a warning message in the web view. The web view looks like this:

2 Position the cursor in the Location field of your external browser and press *Ctrl+V*. This copies the Web Run URL from the clipboard. JBuilder copied this URL into the clipboard, based on your selection on the Web page of the IDE Options dialog box. (You selected this option in an earlier step of this tutorial.) Press Enter to go to the URL.

Important

Some extra setup is required if Netscape 6 is your external browser. For more information, go to the topic called "Using Java Web Start Technology with Netscape™ 6 Web Browsers " in the *Java Web Start Installation Guide* at `http://java.sun.com/products/javawebstart/docs/installguide.html`.

Your web browser displays the application's homepage, `CheckBoxControlLauncher.html`. The web page contains a link to the application.



3 Click the link on the web page. Java Web Start loads and launches the application. The application is now running from a link in an external web browser. Note that the splash screen displays information you entered into the Web Start Launcher wizard.

4 Choose File | Exit to exit the sample application. To stop the web server, choose the Reset Program button on the Web Server tab.

In this tutorial, you learned how to set up a project for a Web Start application, use the Web Start Launcher wizard to create the application's homepage and JNLP file, and launch the application with Web Start.

Index

A

B

C

D

E

F

G

H

I

J

L

M

N

P

Q

R

S

T

U

V

W

X

Z