Web Application Developer's Guide

VERSION 5

Borland®
JBuilder™

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

Refer to the file DEPLOY.TXT 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. 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.

JBE0050WW21004 1E0R0401
0102030405-9 8 7 6 5 4 3 2 1
PDF

Contents

Chapter 16
Deploying your web application 16-1

Chapter 17
Launching your web application with Java Web Start 17-1

Index I-1

Tables

Figures

Chapter

1

Developing web applications

Web Development is a feature of JBuilder Professional and 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 (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. See one of the following chapters for more information:

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

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

- 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 applets"

  Explains how to create applets in JBuilder and deploy them to a web server. Discusses the main issues involved in applet development and deployment and presents solutions.

- Chapter 5, "Working with servlets"

  Introduces servlets and the servlet API.

- Chapter 6, “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 7, “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 8, “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 9, “Developing JavaServer Pages”

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

- Chapter 10, “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 11, “Using InternetBeans Express”

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

- Chapter 12, “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 displays its contents, accept user input, and save it back to the database.

- Chapter 13, “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 displays its contents, accept user input, and save it back to the database.

- Chapter 14, “Configuring your web server”

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

- Chapter 15, “Working with web applications in JBuilder”

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

- Chapter 16, “Deploying your web application”

  Explains how to manage your web application’s deployment descriptors, use JBuilder’s deployment descriptor editor, and deploy your web application.

- Chapter 17, "Launching your web application with Java Web Start"

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

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/.

Contacting Borland developer support

Borland offers a variety of support options. These include free services on the Internet, where you can search our extensive information base and connect with other users of Borland products. In addition, you can choose from several categories of support, ranging from 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.borland.com Technical documents available by anonymous ftp.
Listserv	To subscribe to electronic newsletters, use the online form at: http://www.borland.com/contact/listserv.html or, for Borland's international listserver, http://www.borland.com/contact/intlist.html

World Wide Web

Check www.borland.com regularly. The JBuilder Product Team will post 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.

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 JBuilder Developer Support page at http://www.borland.com/devsupport/jbuilder/. From this site, you can also submit a feature request or view a list of bugs that have already been reported.

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) with the JBuilder documentation, 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, because it helps us to improve our product.

Documentation conventions

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

Table 1.1 Typeface and symbol conventions

Typeface	Meaning
`Monospace type`	Monospaced type represents the following: • text as it appears onscreen • anything you must type, such as "Enter `Hello World` in the Title field of the Application wizard." • file names • path names • directory and folder names • commands, such as `SET PATH, CLASSPATH` • Java code • Java data types, such as `boolean`, `int`, and `long`. • Java identifiers, such as names of variables, classes, interfaces, components, properties, methods, and events • package names • 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. For example, "Press *Esc* to exit a menu."
[ ]	Square brackets in text or syntax listings enclose optional items. Do not type the brackets.

Table 1.1 Typeface and symbol conventions (continued)

Typeface	Meaning
< >	Angle brackets in text or syntax listings indicate a variable string; type in a string appropriate for your code. Do not type the angle brackets. Angle brackets are also used for HTML tags.
...	In code examples, an ellipsis indicates code that is missing from the example. On a button, an ellipsis indicates that the button links to a selection dialog.

JBuilder is available on multiple platforms. See the table below for a description of platforms and directory conventions used in the documentation.

Table 1.2 Platform conventions and directories

Item	Meaning
Paths	All paths in the documentation are indicated with a forward slash (/). For the Windows platform, use a backslash (\).
Home directory	The location of the home directory varies by platform. • For UNIX and Linux, the home directory can vary. For example, it could be `/user/[username]` or `/home/[username]` • For Windows 95/98, the home directory is `C:\Windows` • For Windows NT, the home directory is `C:\Winnt\Profiles\[username]` • For Windows 2000, the home directory is `C:\Documents and Settings\[username]`
`.jbuilder5` directory	The `.jbuilder5` directory, where JBuilder settings are stored, is located in the home directory.
`jbproject` directory	The `jbproject` directory, which contains project, class, and source files, is located in the home directory. JBuilder saves files to this default path.
Screen shots	Screen shots reflect JBuilder's Metal Look & Feel on various platforms.

Macintosh conventions

JBuilder is designed to support Macintosh OS X so seamlessly that JBuilder will have the look and feel of a native application. The Macintosh platform has conventions of appearance and style that vary from JBuilder's own; where that happens, JBuilder supports the Mac look and feel. This means that there are some variations between what JBuilder looks like on the Mac and how it is presented in the documentation. For instance, this documentation uses the word "directory" where Mac uses the word "folder." For further information on Macintosh OS X paths, terminology, and UI conventions, please consult the documentation that comes with your OS X installation.

Chapter

2

Overview of the web application development process

Web Development is a feature of JBuilder Professional and Enterprise.

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

Applet development is a feature of all editions of JBuilder.

Table 2.1 Web application technologies

Technology	Description
Applets	A specialized kind of Java application that can be downloaded by a client browser and run on the client's machine.
Servlets	A server-side Java application which can process requests from clients.
JavaServer Pages (JSP)	An extension of servlet technology. JavaServer Pages 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 component library which provides 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.

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.

Applets

There was much ado about applets when the Java language first became available. Web technology was less developed then, and applets promised some solutions to 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 4, "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 does not 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 4, "Working with applets."

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 which supports JavaServer technology, such as the Tomcat web server which 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. Compare that to 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 16, "Deploying your web application."

For more information on Java servlets, see Chapter 5, "Working with servlets" and Chapter 6, "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.

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.

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

- Less code to write.
- Easy to incorporate existing Java beans.
- 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 don't need to embed HTML code which you intend to have the servlet generate into your Java code. Instead, discrete blocks of Java code are embedded into the HTML. With careful planning, these blocks of Java code can be cleanly separated from the HTML within the file, making the JSP more readable.

Some of the disadvantages of JSPs are:

- 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.

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 9, "Developing JavaServer Pages."

InternetBeans Express

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 11, "Using InternetBeans Express."

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, comboboxes or listboxes, 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.
- Servlets and JSPs are similar enough that deciding between them is largely a matter of personal preference.
- 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 steps are:

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. At this stage, you may want to consider creating a WebApp.
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. If you skip this step, JBuilder will automatically use Tomcat, the web server that ships with JBuilder, for compiling, running, and debugging.
Develop your application	Write the code for the application. Whether your application is composed of applets, servlets, or JavaServer Pages, using JBuilder's many tools simplifies development tasks.
Compile your application	Compile the application in the JBuilder IDE.
Run your application	Run the application in the JBuilder IDE. This will give 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.
Deploy your application	Deploy your application to the web server. The specifics of your approach to this step will depend largely on which web server you are using.
Test your application	Test your application running on the web server. This will help 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, here are some differences between the two technologies.

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 Application 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).
- 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.

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.

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.

A web application can be part of a larger distributed application, which, in turn, can be part of a 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 particular, read the sections called "The Client Tier" and "The Web Tier."

Chapter

3

Working with WebApps and WAR files

Web Development is a feature of JBuilder Professional and Enterprise.

JBuilder provides some important features for managing the structure of your web application. There are two key concepts you need to understand in order 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 advisable 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 would want to 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? 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 "The Web Application wizard" on page 3-3 and "The WebApp and its properties" on page 3-4.

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.

For more information on how JBuilder works with WAR files, see "The WAR file" on page 3-10.

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	This is a simple wizard for creating a WebApp. It allows you to specify the WebApp name, the root directory for the web application's documents, and whether or not to generate a WAR file.
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.
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 and the WebApp DD Editor are discussed in detail in the "Deployment descriptors" on page 16-4.

The Web Application wizard

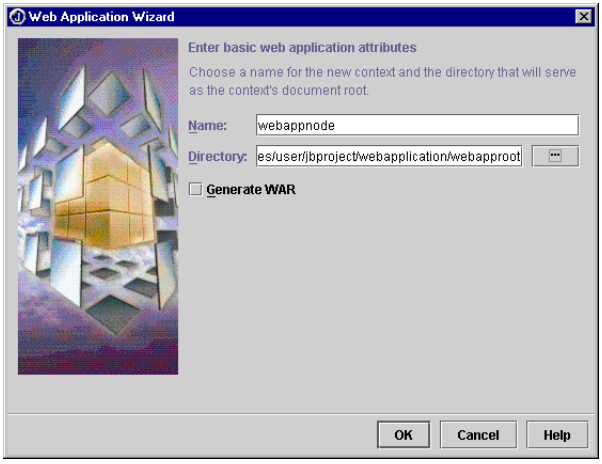
The Web Application wizard creates a new WebApp. To display the Web Application wizard, open the object gallery (File | New), click the Web tab, select Web Application, and click OK.

This wizard is very simple. It consists of one page, with two text fields and a check box. In the first text field, labeled Name, enter a name for your WebApp. In the second text field, labeled Directory, enter the location that will be your WebApp's document root. Typing a directory name here 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 is not recommended.

By checking the Generate WAR check box, you are specifying that you want a WAR file to be generated when building the project. If the check box is selected, the WAR file will have the same name as the WebApp and be placed in the directory that contains the document root directory. If you don't check Generate WAR, don't worry. You can always change your mind later on. This check box corresponds to a setting in the WebApp Properties.

You can also use the Web Application wizard to import an existing web application you may have already built. Give the existing WebApp a name, and point the Directory location to the directory containing your existing web application. If the web application is valid, it can be run from within the JBuilder IDE immediately.

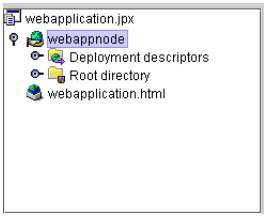
Figure 3.1 Web Application wizard



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 named 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 is not shown under the Root Directory node of the project pane because it is a protected directory that cannot be served by the web server.

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 will be created by JBuilder when you create the WebApp. Your web server may also require additional deployment descriptors which are unique to that particular web server. These can be edited in JBuilder and will be created if they are listed as required by the currently configured web server plugin. 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 the section called "Deployment descriptors" on page 16-4.

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 four tabs:

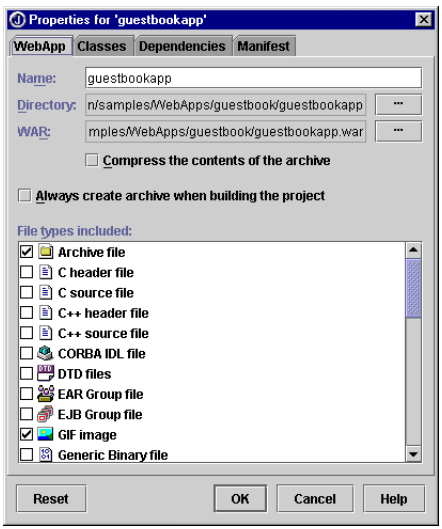
- WebApp
- Classes
- Dependencies
- Manifest

The WebApp tab

The WebApp tab 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 are two check boxes indicating whether or not to generate (or update) the WAR file whenever the project is built, and whether or not to compress the archive. The check box for creating the archive corresponds to the "Generate WAR" check box 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.

This tab also 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.

Figure 3.3 WebApp tab of WebApp Properties



The Classes tab

The Classes page controls 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.

The Classes page contains three radio buttons allowing you to select the method of handling classes and resources. The options are as follows:

Include Required Classes And Known Resources

This option copies any classes that you have specifically added to your WebApp with the Add Classes button. It also adds any classes that are used by one or more of the added classes. 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.

This option also adds known resources. Known resources are those that you specifically add to the archive with the Add Files button. If you select this option and do not add any classes or files to the list below, no classes or resources are copied.

Include Required Classes And All Resources

This option copies any classes that you have specifically added to your WebApp with the Add Classes button. It also adds any classes that are used by one or more of the added classes. 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.

This option also adds all resources in the project's source path, such as images, video clips, sound files, etc.

Always Include All Classes And Resources

This option gathers all classes on your project's outpath. The outpath is defined on the Paths page of the Project Properties dialog box. Usually, this is set to the `classes` directory of your project.

It also gathers all resources on the project's source path, also set on the Paths page of the Properties dialog box. Usually, this is set to the `src` directory of your project. Resources are files other than class files, such as images, video clips, sound files, etc.

This option is on by default. This option is the safest, as it gathers:

- All the classes used in your project
- All classes used by any added classes
- Resources used by classes in your project
- Resources you have added to your project

Caution If you select this option, 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 this option could take some time and become very large.

Add Classes

The Add Classes button displays the Select A Class dialog box, where you select a class, any number of classes, a package or any number of packages to add to your WebApp. The class does not have to be in your project's outpath. If you choose either the Include Required Classes And Known Resources or the Include Required Classes And All Resources options, JBuilder scans these added class files for additional class dependencies and puts those classes in the `WEB-INF/classes` directory.

Add Files

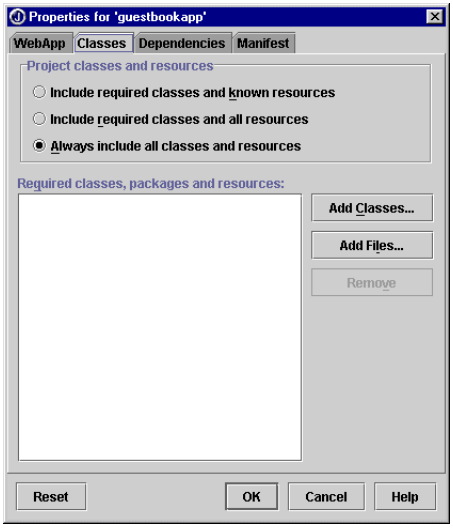
The Add Files button displays the File Open dialog box, where you choose the file or files to add to your WebApp. The file must be in your project's source path. Use this option to add miscellaneous files required by the WebApp's classes, such as `.properties` files and database drivers.

Note The Add Files dialog cannot 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.

Remove

Removes the selected class or file from the list.

Figure 3.4 Classes tab of WebApp Properties

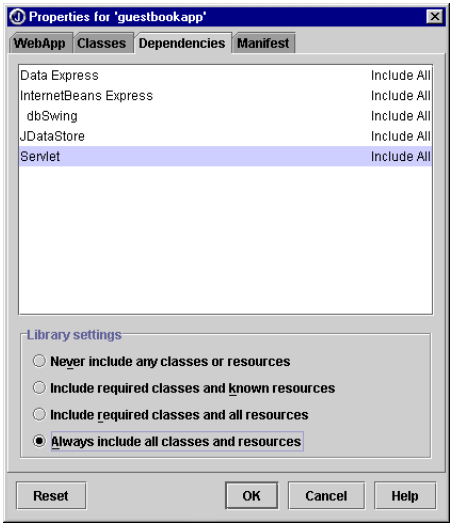


The Dependencies tab

The Dependencies tab 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 tab is Always Include All Classes And Resources - the recommended setting). The

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

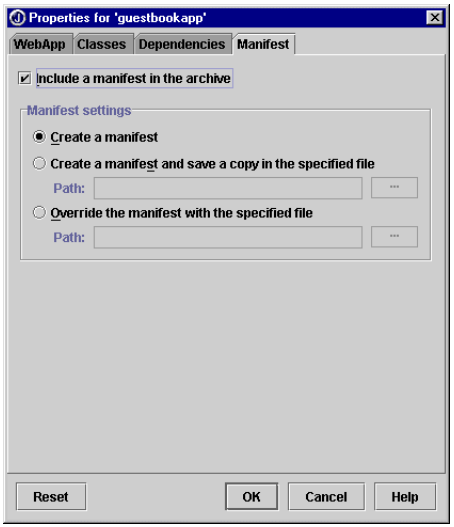
Figure 3.5 Dependencies tab of WebApp Properties



The Manifest tab

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

Figure 3.6 Manifest tab of WebApp Properties

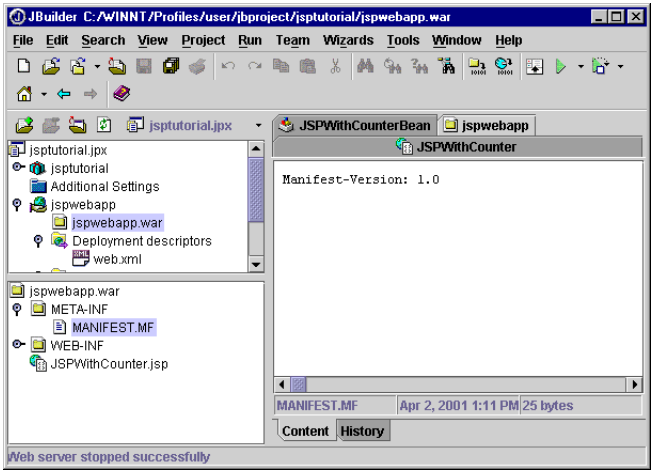


The WAR file

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.

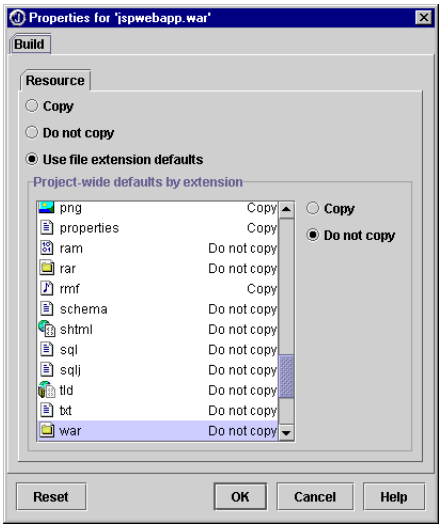
If a WAR file is present, a node representing the WAR file will appear under the WebApp node in the project tree. Opening the WAR file node displays the contents of the WAR file in the structure pane, and also displays the WAR file viewers in the AppBrowser. The WAR file's viewers consist of the Classes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

Figure 3.7 WAR file node open in JBuilder IDE



The WAR file node has a Properties dialog box, accessible by right-clicking on 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 tab which is available in the Build Properties 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 3.8 WAR file properties dialog



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

You can tell JBuilder to automatically create or update a WAR file whenever the project is built. To do this, right-click on the WebApp node in the project pane and select Properties from the menu. Click the WebApp tab of the WebApp Properties dialog box. Make sure that the Always Create Archive When Building The Project option is checked. Also 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. The servlet, JSP, and Web Start Launcher 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. In order 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 does not 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 WebApp Properties. Otherwise, the class files will not 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 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 the *Quick Start*
- The Java tutorial on "Writing Applets" at http://www.java.sun.com/docs/books/tutorial/applet/index.html
- Sun's web site at http://www.javasoft.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, which can prevent applets from performing such tasks 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 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

Below 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.

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 foo.bar.baz, 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 we gave, 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. Below are the most frequent issues you'll encounter concerning browsers.

Java Support

A common problem with applets is a JDK mismatch between your applet and the browser running it. There are many users who have not upgraded or updated their browsers. These browsers support JDK 1.1.5 code and earlier but not the newer JDKs from Sun. Swing, introduced in JDK 1.1.7, is not yet supported by most 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 and 1.3 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. Browsers that do support JDK 1.1 are Netscape 4.06+ and Internet Explorer 4.01+. JDK version support may vary by platform.

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:

- Select Communicator | Tools | Java Console in Netscape.
- 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 you have to get your clients to 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

- **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 most 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.

  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.

  From the end-user perspective, the first time they encounter a Java Plug-in enabled page, they 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, 1.2, and 1.3. 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 Professional 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 "Setting project properties" in *Building Applications with JBuilder*.

- **Write only JDK 1.1.x code and earlier**

  Writing your applets using JDK 1.1.x and earlier limits you to the functionality (and bugs) of these versions. Also, most development tools generate 1.1.x or higher code. JBuilder, for example, has many tools that generate 1.1.x or higher code which you cannot use for your applets. However, JBuilder does provide a feature that allows you to edit or switch your JDK version. See "Setting project properties" in *Building Applications with JBuilder*.

  JDK 1.02 has the best support in the browsers, but it also has different mechanisms for events than JDK 1.1.x. Most browsers now support

JDK 1.1.5 but not Swing components, which were introduced in JDK 1.1.7. If you do write your applets in newer JDK versions, use only AWT components and do not use any of the newer features, such as Swing components.

The biggest disadvantage to using JDK 1.1.x and earlier versions is that all the improvements are in later versions, so you'll eventually have to migrate and port the old code. Another disadvantage is that some browser versions that are JDK 1.1.x-compliant have more problems with JDK 1.02 code. Also, be aware that there is no debugging with JDK 1.l and earlier, because there isn't any JPDA (Java Platform Debugger Architecture) implementation.

- **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 an application-deployment technology from Sun Microsystems. It allows you to launch Java applications and applets from a link on a web page in your web browser.

  For more information on Web Start, see Chapter 17, "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

Below 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 1.1.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 4-13 and "Deploying applets in JBuilder" on page 4-23 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 4-13 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.

See also "Testing applets" on page 4-13

Security and the security manager

An important concern about running programs over the Internet is security. Users are rightly 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

- **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). These are solutions that are better suited to an intranet.

- **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 using 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, 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 Professional 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 Professional and Enterprise or the JDK **jar** tool to create the archive files.

Important If you are writing JDK 1.1.1 applets with Swing components, which is not recommended due to browser issues, you will 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 4-23
- "Deploying Java programs" in *Building Applications with JBuilder*
- **jar** tool (http://java.sun.com/j2se/1.3/docs/tooldocs/tools.html#basic)

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

Below are the basic steps for testing your applet:

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
   ```

   where `<jbuilder>` is the name of the JBuilder directory. For example, `jbuilder5/`.

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:
  - Select Communicator | Tools | Java Console in Netscape.
  - Select View | Java Console in Internet Explorer.

See also "Solving Common Applet Problems" in the Java Tutorial, "Writing Applets" (http://www.java.sun.com/docs/books/tutorial/applet/problems/index.html)

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 the *Quick Start*
- "Designing a user interface" in *Building 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 4-5 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, directory name, and root, source, backup, documentation, and output paths. Change the project file name and the project directory name if you want to give the project a particular name. Leave the rest of the paths unchanged.

Note The paths for the project files are pre-set in the default project properties. Until you become a more 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>/<username>/jbproject/appletproject1/appletproject1.jpr`, the Applet wizard suggests using a package name of `appletproject1`.

For more information on packages, see "Packages" in *Building Applications with JBuilder*.

2. Click Next to continue to Step 2 of the Project wizard. Note the paths generated from Step 1:

Project path	Where the project file is saved.
Source path	Where the source files are saved.
Backup path	Where the backup files are saved.
Documentation path	Where the documentation files are saved.
Output path	Where the class files and applet HTML file are saved.

These paths and the JDK path can be edited by selecting the ellipsis button. Required libraries can also be added.

See also "Setting the JDK" and "How JBuilder constructs paths" in *Building Applications with JBuilder*

3. Click Next to continue to Step 3 where you can enter project information in the project notes HTML file.
4. Click Finish to complete the Project wizard.

The Project wizard creates two files:

- A project file (`.jpr` or `.jpx`).
- 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.

5 Next, choose File | New and choose the Web tab of the object gallery. (In JBuilder Personal, the Applet icon is on the New page.)

6 Double-click the Applet icon in the object gallery to open the Applet wizard.

7 Type a new name for the applet class if you like.

8 Select the base class you want the applet to extend: `java.applet.Applet` (AWT class) or `java.swing.JApplet` (JFC Swing class).

Important In most cases, it's best to choose `java.applet.Applet` as the base class, because most web browsers do not yet support Swing.

9 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()`.

10 Click Next to go to Step 2. 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.

11 Click Next to go to Step 3. 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 about it, 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.

12 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 Step 3:

- 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 Professional and Enterprise, an automatic source package node also appears in the project pane if the Automatic Source Packages 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 4-15
- "Tutorial: Building an applet" in the *Quick Start*

Running applets

To run an applet within a project,

1. Save the file.
2. Compile the project.
3. Do one of the following:
   - Right-click the applet's HTML file in the project pane. This file must have an `<applet>` tag. Select Run. The applet runs in Sun's **appletviewer**.
   - 
     Choose Project | Run Project or choose the Run button on the toolbar to run the applet from the main class in JBuilder's AppletTestbed.

     The main class is set on the Run page of Project Properties (Project | Project Properties).

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 in Step 1 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**

The default behavior is as follows, if you've created your applet with the Applet wizard:

- Select Run | Run Project or the Run button to run the applet from the main class in JBuilder's applet viewer, AppletTestbed.
- Right-click the applet HTML file and select Run to run the applet in Sun's **appletviewer**.

You can change the default behavior for Run | Run Project and the Run button on the Applet tab of the Run page of the Project Properties dialog

box (Project | Project Properties | Run). 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 1.1.1 did **not** include the JFC classes, you need to download the JDK 1.1.*x* specific version of JFC (Swing), `swingall.jar`, from the JavaSoft site (http://www.javasoft.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 Open Tools 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 on the Applet tab of the Run page (Project | Project Properties).
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 loads in the message pane and the applet runs in JBuilder's AppletTestbed.

Note If you're developing your applet using an older JDK, you'll need to switch to a newer JDK (JDK 1.2.2 or 1.3) 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:
  1. Compile the project.
  2. Set a breakpoint in the applet file.
  3. Right-click the applet HTML file in the project pane and choose Debug. The debugger loads in the message pane and the applet runs in the **appletviewer**.
- From the JBuilder Run menu:
  1. Set the applet's HTML file as the runnable file on the Applet tab of the Run page (Project | Project Properties).
  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 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 4.72 using the Java Plug-in and JBuilder's debugger.

To debug using JDK 1.3,

1. Download and install the plug-in for JDK 1.3 from the Sun site: http://java.sun.com/products/plugin/.
2. Download and install the HTML converter from the Sun site at http://java.sun.com/products/plugin/1.3/features.html.

   (The HTML converter version must match the Plug-in version.)
3. Follow instructions to set the debug parameters for the Java Plug-in for applets: http://java.sun.com/products/plugin/1.3/docs/debugging.html.

Note Add **-classic** as the first debug parameter in the Java Plug-in Control Panel to use the Classic VM for faster debugging.

4. Launch JBuilder and do the following:
   1. Create a simple applet with an HTML file. See "Tutorial: Building an applet" in the *Quick Start*.
   2. Compile the project.
   3. Convert the HTML file using the HTML converter. Instructions to use the HTML converter can be found at: http://java.sun.com/products/plugin/1.3/docs/htmlconv.html.
   4. Set a breakpoint in the applet file.
   5. Set the following on the Debug page of the Project Properties dialog box (Project | Project Properties):
      1. Enable remote debugging.
      2. Select the Attach option.
      3. Set the same debug parameters as you did in the Java Plug-in Control Panel. Choose one of these options:
         - Choose Transport Type `dt_shmem` and `javadebug` in the Address field.
         - Choose Transport Type `dt_socket` and enter the address you have provided in the control panel (default provided in JBuilder is 5000) in the Address field.

      For further information on remote debugging, see "Debugging distributed applications" in the *Distributed Application Developer's Guide*.

5 Start Netscape or Internet Explorer and open the HTML file. Be sure the browser has Java enabled. Refer to the browser's online help.

6 Switch to JBuilder and 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 Professional 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 (http://java.sun.com/j2se/1.3/docs/tooldocs/tools.html#basic)
- Chapter 3, "Working with WebApps and WAR files"

Chapter

5

Working with servlets

Web Development is a feature of JBuilder Professional and Enterprise.

Java servlets provide a protocol and platform-independent method for building web-based applications without the performance limitations of CGI programs. Servlets run inside web servers and, unlike applets, do not need a graphical user interface. They interact with the servlet engine running on the web server through requests and responses. A client program, which can be written in any programming language, will access the web server and make a request. The request is then processed by the web server's servlet engine, which passes it on to the servlet. The servlets 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), the iPlanet Web Server (formerly Netscape Enterprise Server) and others. Web servers with servlet engines, also known as servlet containers, can also be integrated with web-enabled application servers, such as the Borland AppServer (included with JBuilder Enterprise), the BEA WebLogic Application Server (supported in JBuilder Enterprise), IBM WebSphere (also supported in JBuilder Enterprise), the Netscape Application Server, and others.

Note JBuilder Professional and Enterprise include the Tomcat web server "in-the-box" 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. It

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 9, "Developing JavaServer Pages."

For more information on servlets, go to the following 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.

- 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
- Java Developer Connection: Servlets page at http://developer.java.sun.com/developer/technicalArticles/Servlets/index.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 7, "Tutorial: Creating a simple servlet"
- Chapter 8, "Tutorial: Creating a servlet that updates a guestbook"
- Chapter 12, "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 14, "Configuring your web server."

JBuilder Professional and Enterprise editions deliver Tomcat "in-the-box" to use as your web server, so that you can successfully develop and test your servlets and JSPs within the JBuilder development environment. To learn more about Tomcat, see the Tomcat documentation, installed in the `doc` folder of JBuilder's Tomcat installation.

Many other web servers support JavaSoft's servlet and JSP APIs. For a list of these products, see the "Servers and Engines" topic on JavaSoft'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 5.1 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.

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 will call `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, that thread must complete each task, in sequential order, before moving on to the next.

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 complete servlet support, you can use the <servlet> tag to preprocess web pages. This tag must be in a file with an `.shtml` extension. (This type of functionality is also called *Server Side Include* functionality.) 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>
```

Note that two of the servlet content types (HTML and XHTML) created by JBuilder's Servlet wizard use the <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

Typically, servlets are not deployed stand-alone to a production web server. Usually, they are deployed as a J2EE module, the basic unit of composition in a J2EE application. A J2EE module will consist of one or more J2EE components of the same type (web, EJB, client etc.) that share a single deployment descriptor. A J2EE application will consist of one or more J2EE modules.

A J2EE module deployment descriptor is an extensible XML file. The deployment descriptor file contains all of the declarative data required to deploy the components in that module. It also contains assembly instructions that describe how the components are composed into an application. It must reside in the same package as the servlet and other web components you are packaging. For more information, see Chapter 8, "Application Assembly and Deployment," of the Java 2 Platform Enterprise Edition, v1.3 Proposed Final Draft. You can download this document from http://java.sun.com/j2ee/download.html#platformspec.

You can use JBuilder to create the deployment descriptor and package your J2EE module or application for deployment. For more information, see Chapter 16, "Deploying your web application."

Chapter

6

Creating servlets in JBuilder

Web Development is a feature of JBuilder Professional and Enterprise.

In JBuilder Professional and 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 JavaServer Pages v1.2/Java Servlets v2.3 specifications).

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.

Servlet wizard - Naming and Type page

On the first step of the Servlet wizard, you can choose the package the servlet will belong 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, will be placed in the WebApp directory.

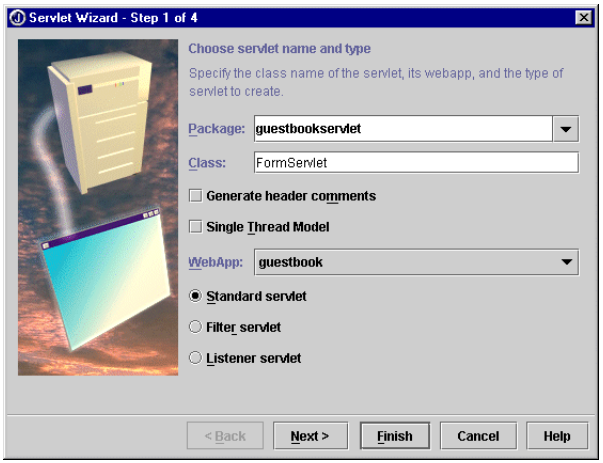
The options at the bottom of the dialog allow you to choose the type of servlet you're creating.

Note These options are only available if your web server supports the JavaServer Pages v1.2/Java Servlets v2.3 specifications. (Tomcat 4.0 is the reference implementation of these specifications.) Otherwise a standard servlet is created by default.

Table 6.1 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 "Servlet wizard - Naming Options page" on page 6-6) defaults to the simple class name of the servlet, in lowercase. If the WebApp is a named WebApp, the servlet URL (also on the Naming Options 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). This option is available only if a named WebApp is selected. If this option is selected, the Servlet wizard - Naming Options 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. This option is available only if a named WebApp is selected. If this option is selected, the Servlet wizard - Listener Servlet Details page is the only other available Servlet wizard page.

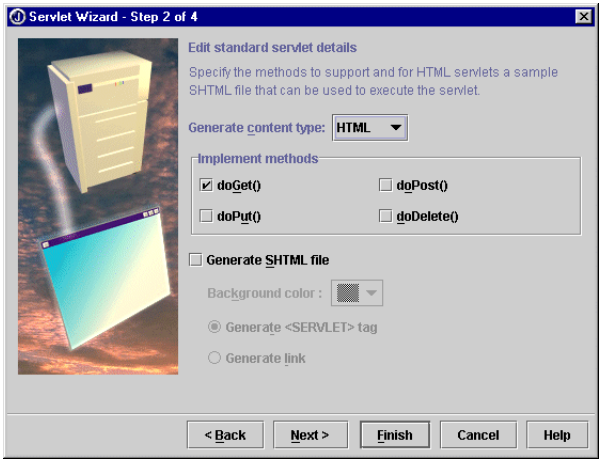
Figure 6.1 Servlet wizard - Naming and Type page



Servlet wizard - Standard Servlet Details page

If you've selected Standard Servlet as the servlet type on the Servlet wizard - Naming and Type page of the Servlet wizard, the 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 6.2 Servlet wizard - 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 "Using XML in JBuilder" in *XML Application Developer's Guide*.

  A `.java` file is generated. The step 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 these. 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 an 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 will support 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 will support 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 will support HTTP PUT requests.
- `doDelete()` - Allows a client to remove a document or web page from the server. Override this method if the servlet will support 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 6-10, select the Generate SHTML File option. An SHTML file, with the same name as the servlet, is added to the project. If you've selected a WebApp on the Servlet wizard's Naming and Type page, the SHTML file will be placed in this directory.

If you want to run the servlet directly, as described in "Invoking a servlet from a browser window" on page 6-9, 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 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 class name.

Servlet wizard - Naming Options page

This page allows you to quickly define basic WebApp mappings. (The selected WebApp is used to run or debug the servlet when you select Web Run or Web Debug.) 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 16-2 or "Filters page" on page 16-7.

- If you've selected Standard Servlet as the servlet type on the Naming 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 15-3.

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.

In this field, enter the servlet's name. For a standard servlet, this is the name that is used to run the servlet. By convention, this is always lowercase.

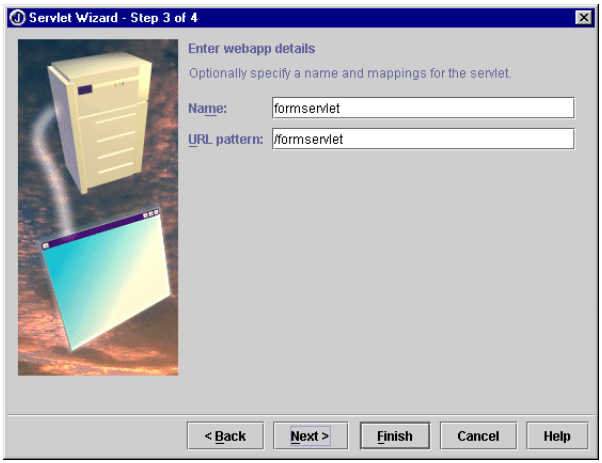
The Name represents a short-hand way of running a servlet. For example, instead of pointing your browser to the servlet's fully qualified class name running on the web server, you could simply enter: `inputform`. This could map to the servlet named `formservlet`, which in turn could be mapped to the servlet class `http://localhost:8080/servlet/FormServlet` running locally on the Tomcat web server from the JBuilder IDE.

For both standard and filter servlets, the URL Pattern field is where you choose the URL pattern that will be used to run the servlet. The URL pattern is always lowercase and preceded by a forward slash. This is useful when you're invoking the servlet directly, from a browser. (See "Invoking a servlet from a browser window" on page 6-9 for more information.)

A default URL pattern is suggested. For a standard servlet, the default pattern consists of the servlet's class name in lower case preceded by a forward slash: `/formservlet`. This would map to the servlet's fully qualified class name. If the servlet is a filter servlet, the URL Pattern field defaults to `/*`, which means that the filter is applied to all requests.

Note This field is not available if you did not select a WebApp on the Servlet wizard's Naming and Type page.

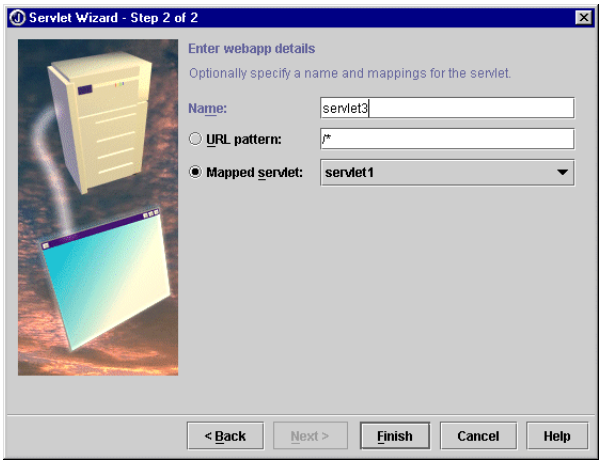
Figure 6.3 Servlet wizard - Standard servlet Naming Options page



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 will be 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 all other servlets in the project that have been previously mapped. If there are none, the option is disabled.

Figure 6.4 Servlet wizard - Filter servlet Naming Options page

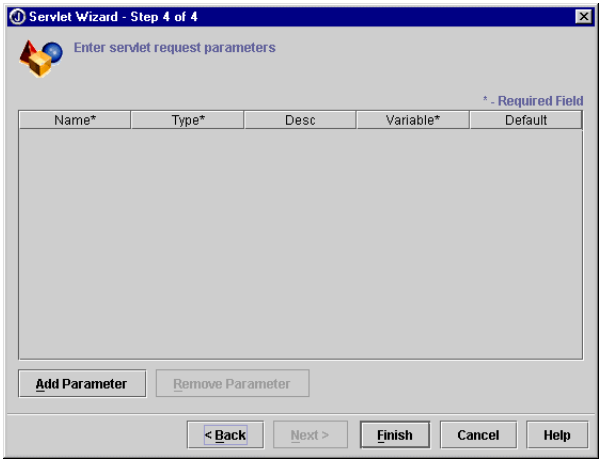


Servlet wizard - Parameters page

The 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 servlet's background color.

The servlet in Chapter 7, "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 .java file is `userName`.

Figure 6.5 Servlet wizard - Parameters page

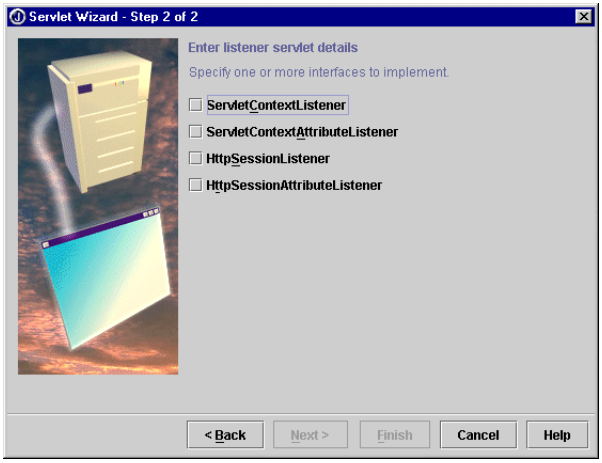


Servlet wizard - Listener Servlet Details page

This page is available only if you've selected the servlet type of Listener Servlet on the Naming and Type page of the Servlet wizard. It is Step 2, and the final Servlet wizard step for listener servlets. 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 16-8.

Figure 6.6 Servlet wizard - Listener Servlet Details page



Invoking servlets

The following topics discuss a few ways to invoke servlets:

- Invoking a servlet from a browser window
- Calling a servlet from an HTML page

Invoking a servlet from a browser window

A servlet can be called directly by typing its URL into a browser's location field. The general form of a 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 run 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 7, "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 the servlet used the name `firstservlet`, you would enter:

```
firstservlet?userName=Mary
```

For more information on how servlets are run, see "How URLs run servlets" on page 15-3.

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 3.2 Reference Specification* on the web at http://www.w3c.org/TR/REC-html32.html.

For example, in Chapter 7, "Tutorial: Creating a simple servlet," the SHTML page contains an anchor with a servlet as a destination:

```
<a href="/servlet/simpleservlet.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/servlet/simpleservlet.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/servlet/
   simpleservlet.Servlet1;">
```

Note that you can substitute the servlet's URL pattern for the fully qualified class name. For example, if you've assigned a servlet the name of `inputform` (see "Servlet wizard - Naming Options page" on page 6-6), you can use the following `<form>` tag to run the servlet:

```
<form action="inputform"  method="post">
```

Internationalizing servlets

Servlets present an interesting internationalization problem. Since 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. Servlet's 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")` will return a `String` representing the system's current encoding setting.

If the `OutputStreamWriter` constructor is called with only the first argument, the current encoding setting will be 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 11, "Using InternetBeans Express." You can also refer to Chapter 12, "Tutorial: Creating a servlet with InternetBeans Express."

Chapter

7

Tutorial: Creating a simple servlet

Web Development is a feature of JBuilder Professional and Enterprise.

This tutorial shows how to create a basic servlet that accepts user input and counts the number of visitors to a web site. You can develop and test servlets within JBuilder using the Tomcat servlet engine shipped "in-the-box" with JBuilder. This tutorial shows how to develop and run a servlet within JBuilder.

To demonstrate how to develop a Java servlet, we will build a fairly basic Hello World-type application that illustrates the general servlet framework. This servlet will display a welcome message, the user's name, and the number of connections since the servlet was started.

All servlets are built by extending a basic `Servlet` class and defining Java methods to deal with incoming connections. This sample servlet will extend the `HttpServlet` class that understands the web's HTTP protocol and handles most of the underlying "plumbing" required for a web application.

To build `SimpleServlet`, we'll use the Servlet wizard to extend the base `HttpServlet` class. We'll then define a method to output several lines of HTML, including the user's name.

For more information on servlets, read the following chapters:

- Chapter 5, "Working with servlets"
- Chapter 6, "Creating servlets in JBuilder"

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 *Building Applications with JBuilder.*

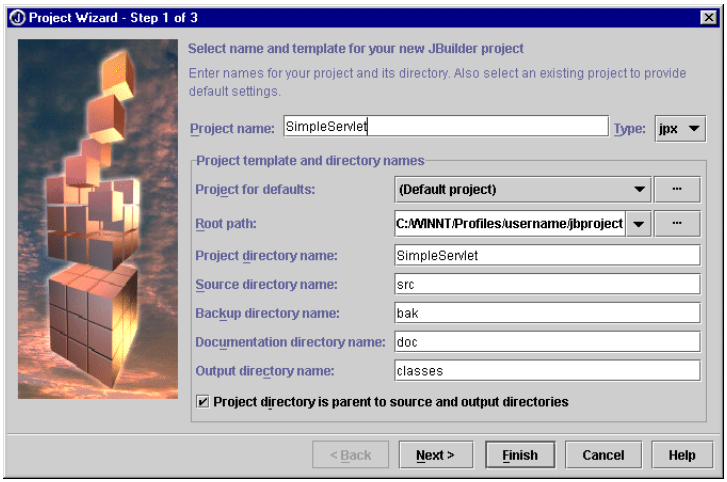
For suggestions on improving this tutorial, send email to jpgpubs@borland.com.

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. Enter `SimpleServlet` in the Project Name field.
3. Make sure the Project Directory Is Parent to Source And Output Directories option is checked.

   Step 1 of the Project wizard should look like this:



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.

In the next step, you'll create a WebApp for your servlet. Though we won't be deploying this project, in a real-life situation, you'd always want to create a WebApp.

Step 2: 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.

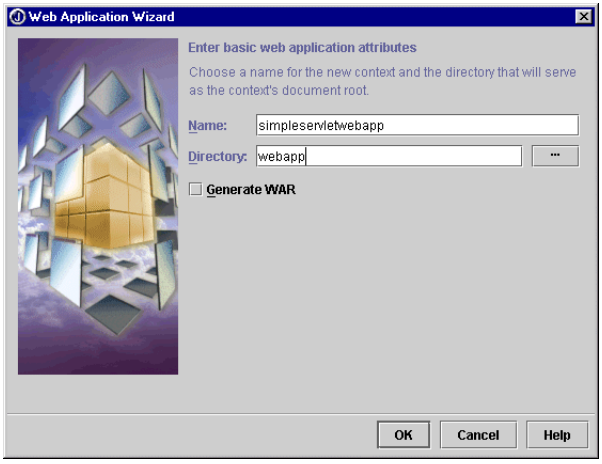
  The Web Application wizard is displayed.

2 Enter `simpleservletwebapp` in the Name field.

3 Enter `webapp` in the Directory field.

4 Make sure the Generate WAR option is not selected.

  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."

In the next step, you'll create the servlet.

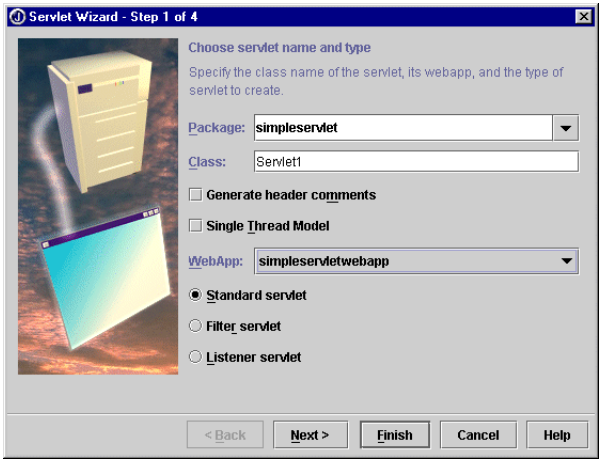
Step 3: Creating the servlet with the Servlet wizard

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 override.
- Create an SHTML file to run the servlet.
- Create parameters for the servlet.

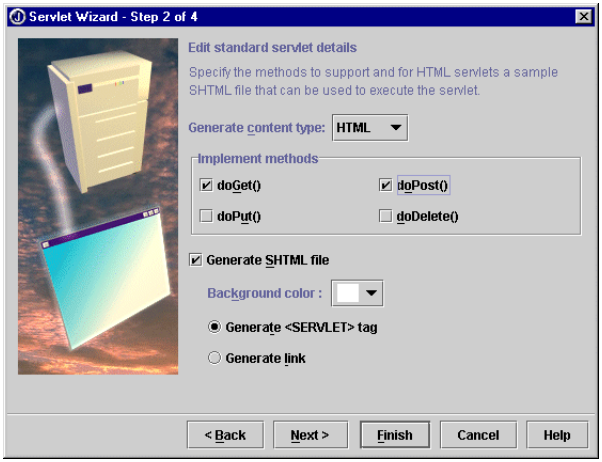
To create the servlet,

1. Choose File | New to display the object gallery.
2. Click the Web tab and choose Servlet. Click OK. Step 1 of the Servlet wizard is displayed.
3. Accept the default Package and Class names. In the WebApp drop-down list, choose `simpleservletwebapp`. Accept the other defaults. Step 1 should look like this:



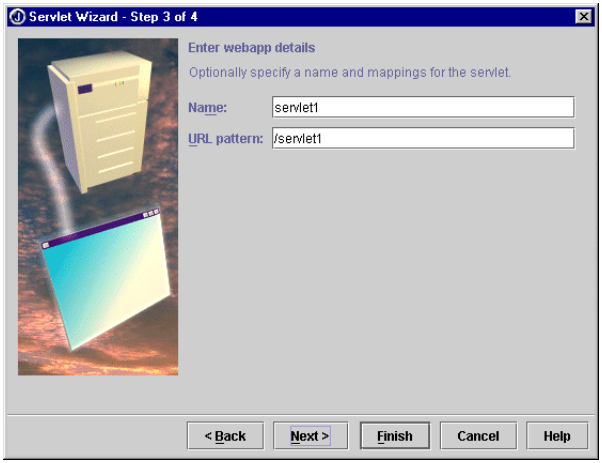
4. Click Next to go to Step 2.

5 On Step 2 of the wizard, select the `doPost()` method. Make sure the `doGet()` method is selected. Select the Generate SHTML File option and the Generate <SERVLET> Tag option. Step 2 should look like this:



6 Click Next to go to Step 3.

7 Accept the default Name and URL Pattern on Step 3 of the wizard. Step 3 will look like this:



8 Click Next to go to Step 4.

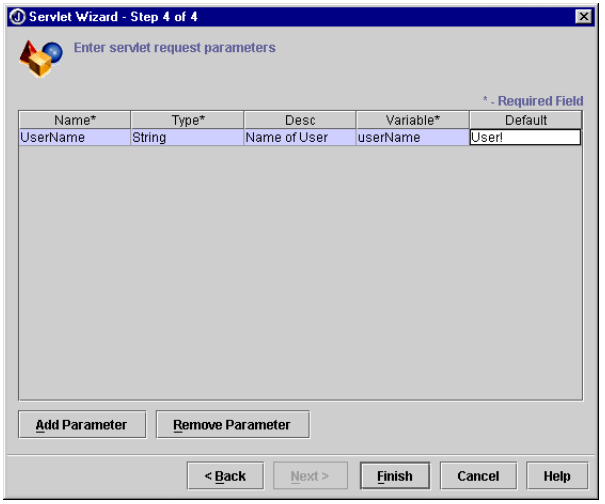
9 Click the Add Parameter button to create a new servlet parameter. This parameter contains the name entered into the servlet's text entry field.

The following table describes the fields and required values. You need to enter the values that are in the Value column of the following table.

Table 7.1 Servlet wizard parameter options

Parameter Name	Value	Description
Name (required)	`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 (required)	`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 (required)	`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, Step 4 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 Web Run and Web Debug commands are enabled for the servlet and its SHTML file.

11 Choose File | Save All to save your work.

In the next step, you'll add code to `Servlet1.java`.

Step 4: 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 and use the Search | Find command to find the comment `/**Initialize global variables**/` near the top of the file. Immediately before that line, enter 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 the reply.
   ```

   Immediately after that line of code add the following code:

   ```
   out.println("<p>Thanks for visiting, ");
   out.println(request.getParameter("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.

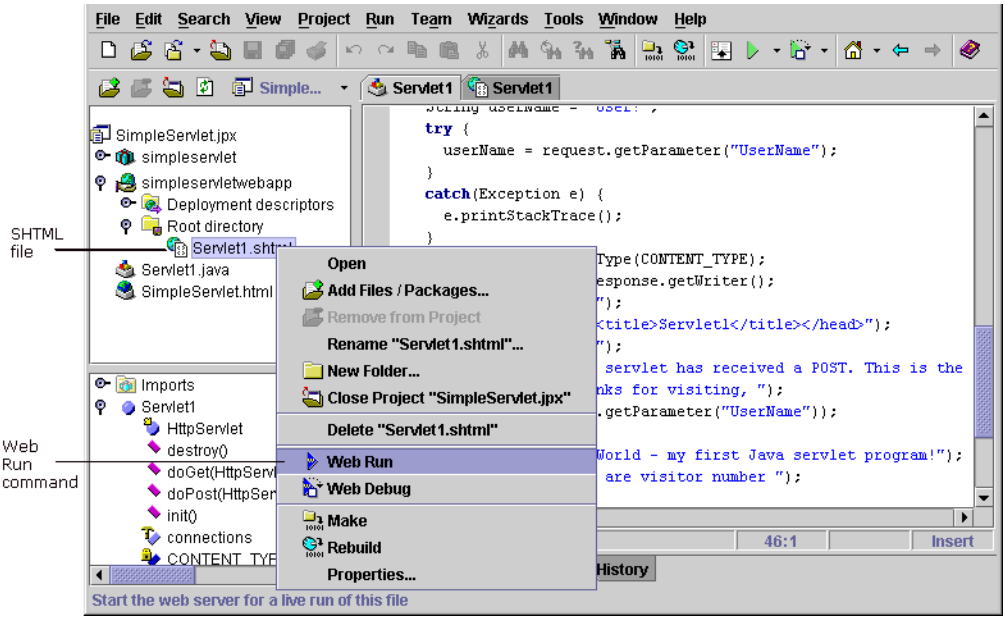
In the next step, you'll compile and run the servlet.

Step 5: 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.

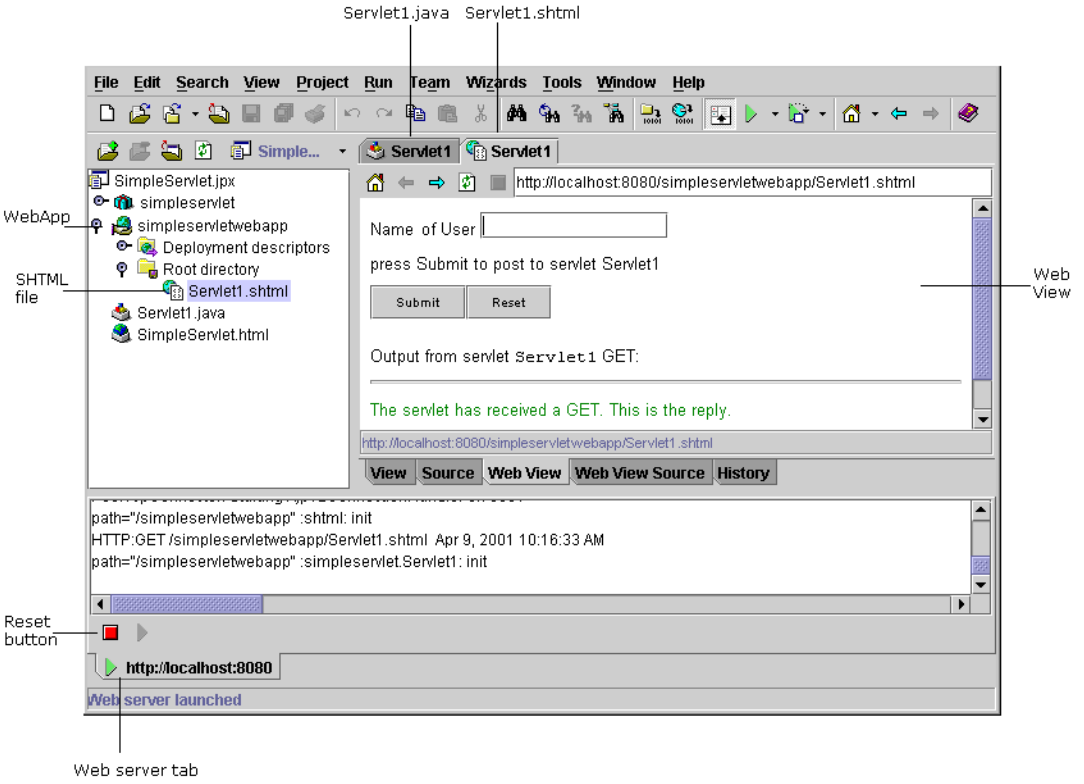
3 Choose Web Run.



Note You can also run servlets directly by right-clicking the `java` file in the project pane, and selecting Web Run. In this example, we 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.

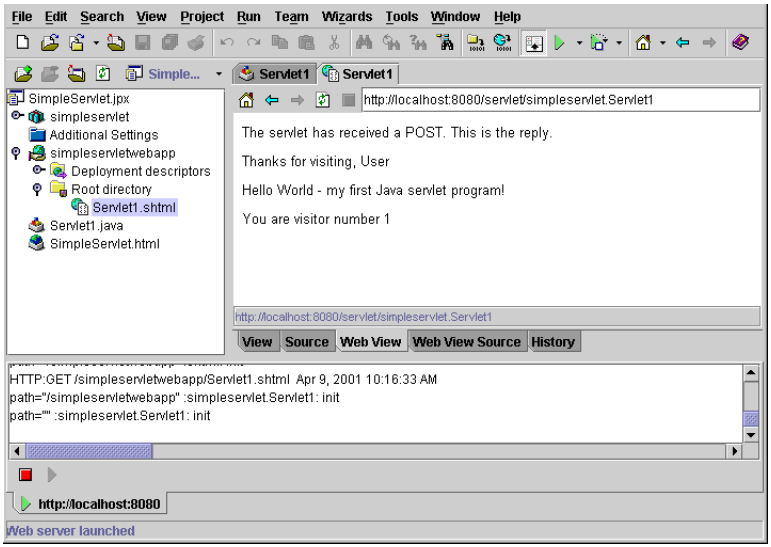
Running the SHTML file starts Tomcat, the web server that comes "in-the-box" with JBuilder. The output from Tomcat is displayed in the message pane. HTTP commands and parameter values are also echoed to the output pane. Two new tabs appear in the content pane for the servlet: Web View and Web View Source. The running servlet displays in the web view. It looks like this:

Figure 7.1 Servlet running in the web view



To run the servlet, enter a name 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 in the next figure.

Figure 7.2 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. Enter 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.

Congratulations! You have completed your first servlet program. For a more advanced servlet that connects to a database, see Chapter 8, "Tutorial: Creating a servlet that updates a guestbook."

Chapter

8

Tutorial: Creating a servlet that updates a guestbook

Web Development is a feature of JBuilder Professional and 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.

You will run the runnable class, `FormServlet`, from the WebApp `guestbook`, using the URL pattern `/inputform` instead of the class name. You will call `DBServlet` from `FormServlet's` <form> tag using the servlet name `table` instead of the class name.

This tutorial assumes that you are familiar with servlets and with JBuilder's JDataStore and DataExpress technologies. For more information on servlets, read the following chapters:

- Chapter 5, "Working with servlets"
- Chapter 6, "Creating servlets in JBuilder"

To get started, you can also work through a less complex "Hello World" servlet - see Chapter 7, "Tutorial: Creating a simple servlet" for more

information. For more information about the JBuilder's database functionality, see the *Database Application Developer's Guide*. For more information about JDataStore, see the *JDataStore Developer's Guide*.

Note This tutorial assumes that you have entered your licensing information into the JDataStore License Manager. For more information, see "Using JDataStore for the first time" in the *JDataStore Developer's Guide*.

The completed classes for this tutorial can be found in the `samples/WebApps/GuestbookServlet` directory of your JBuilder installation.

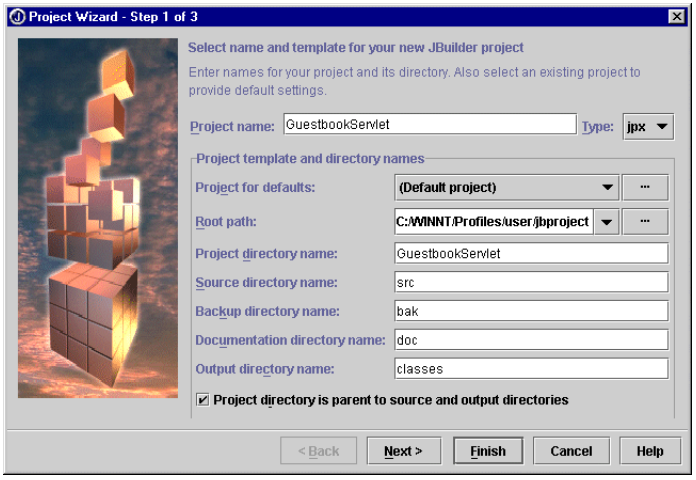
For suggestions on improving this tutorial, send email to jpgpubs@borland.com.

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. Enter `GuestbookServlet` in the Project Name field.
3. Make sure the Project Directory Is Parent To Source And Output Directories option is checked.

   Step 1 of the Project wizard should look like this:



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: 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.

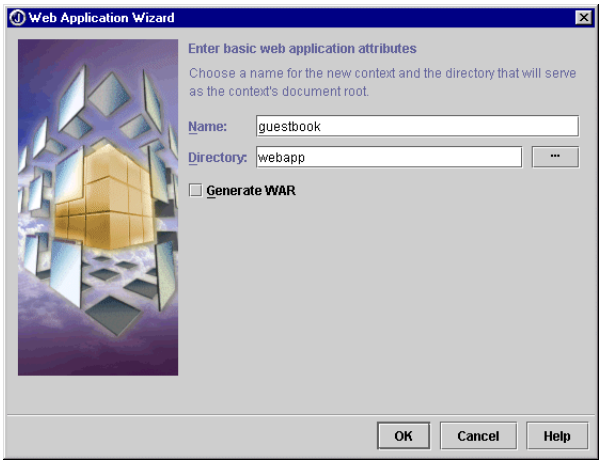
  The Web Application wizard is displayed.

2 Enter `guestbook` in the Name field.

3 Enter `webapp` in the Directory field.

4 Make sure the Generate WAR option is not selected.

  The Web Application wizard should look similar to this:



5 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 and the Root Directory nodes.

For more information on WebApps, see Chapter 3, "Working with WebApps and WAR files."

Step 3: 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`,

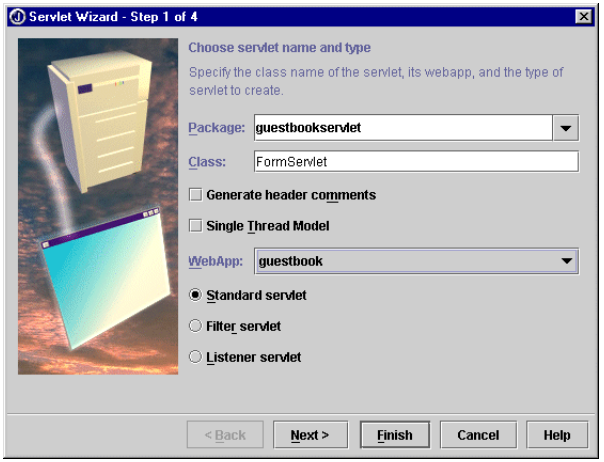
1. Choose File | New to display the object gallery. Click the Web tab and choose Servlet. Click OK.

   The Servlet wizard is displayed.

2. Leave the package name set to `guestbookservlet`. Change the Class field to `FormServlet`.

3. Make sure the Generate Header Comments and Single Thread Model options are not selected. Choose `guestbook` from the WebApp drop-down list. (You just created this WebApp in the previous step.) Leave the Standard Servlet option selected.

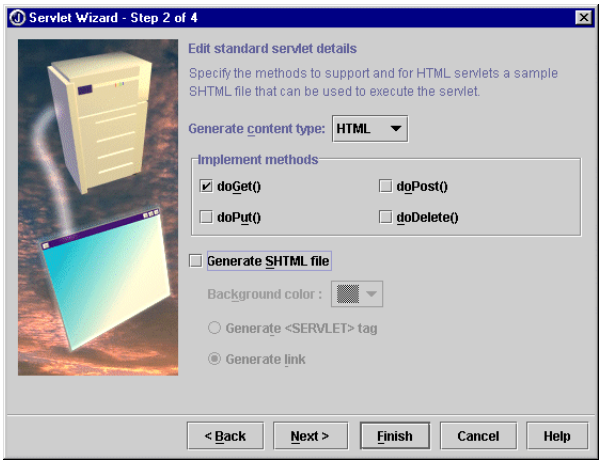
   Step 1 of the wizard should look like this:



4. Click Next to go to Step 2 of the wizard.

5. Make sure that the Content Type option is set to HTML and that the `doGet()` method is selected. Do not select the Generate SHTML File option.

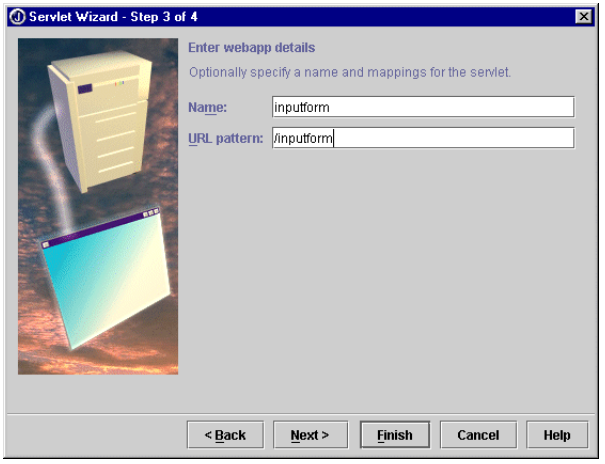
When you're finished, Step 2 of the wizard should look similar to this:



6 Click Next to go to Step 3 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 will be used to run the servlet instead of the class name.

Step 3 of the wizard should look like this:



8 Click Finish to create the servlet. You do not need to add parameters in Step 4 of the wizard.

To create `DBServlet.java`,

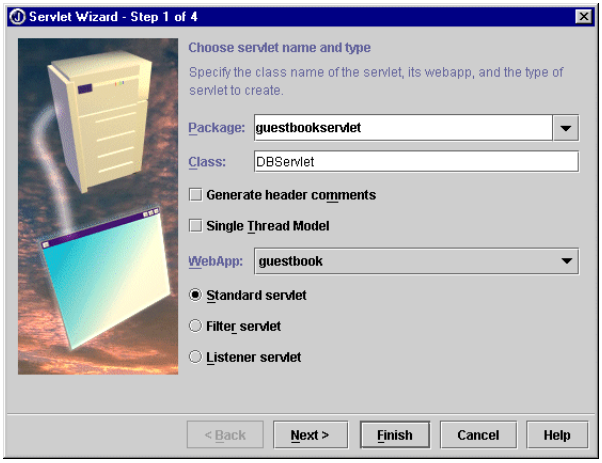
1 Choose File | New to display the object gallery. Click the Web tab and choose Servlet. Click OK.

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.

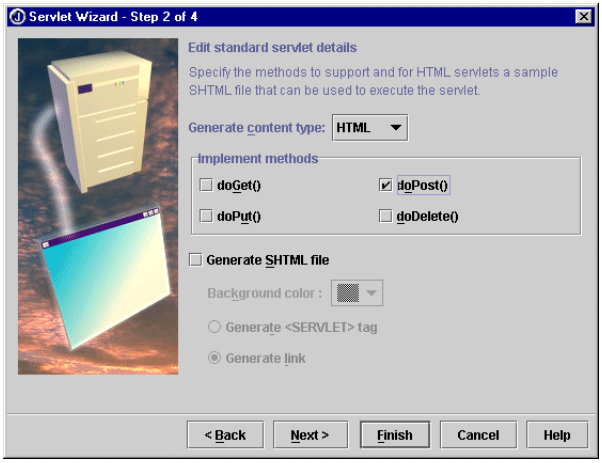
Step 1 of the wizard should look like this:



4 Click Next to go to Step 2 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.

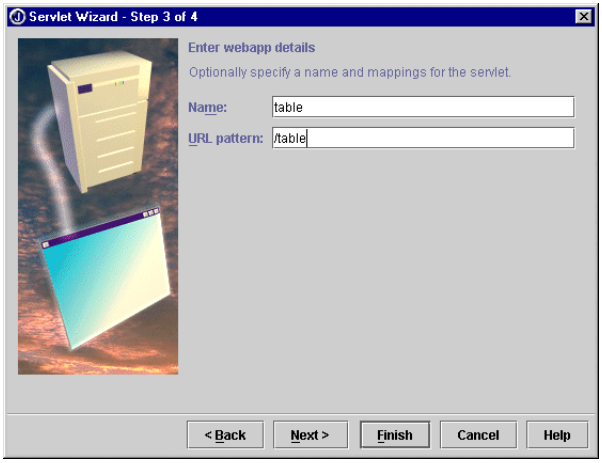
When you're finished, Step 2 of the wizard should look similar to this:



6 Click Next to go to Step 3 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.

Step 3 of the wizard should look similar to this:



8 Click Finish to create the servlet. You do not need to add parameters on Step 4 of the wizard.

In the next step, you'll create the data module that holds the business logic.

Step 4: Creating the data module

To create the data module that connects to the Guestbook JDataStore and performs the update tasks,

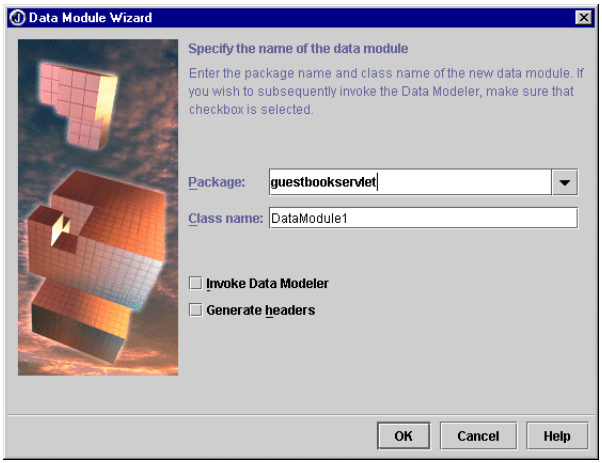
1 Choose File | New to display the object gallery. On the New page, choose Data Module and click OK.

The Data Module wizard is displayed.

2 Leave the package name set to `guestbookservlet`. Keep the default Class Name of `DataModule1`.

3 Make sure the Invoke Data Modeler and Generate Headers options are not selected.

When you're finished, the Data Module wizard will look similar to this:



4 Click OK to create the data module.

5 Choose File | Save All or click  on the toolbar to save your work.

In the next step, you'll add the database components to the data module.

Step 5: 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 "JBuilder's visual design tools" in *Building 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.

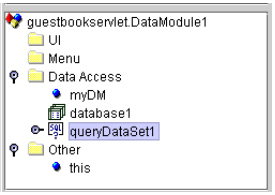
To add data access components to the data module,

1 Choose the DataExpress tab on the component palette.

2 Click the Database icon. Then click in the component tree to add the database component to the data module.

3 Click the QueryDataSet icon. Then click in 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 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.

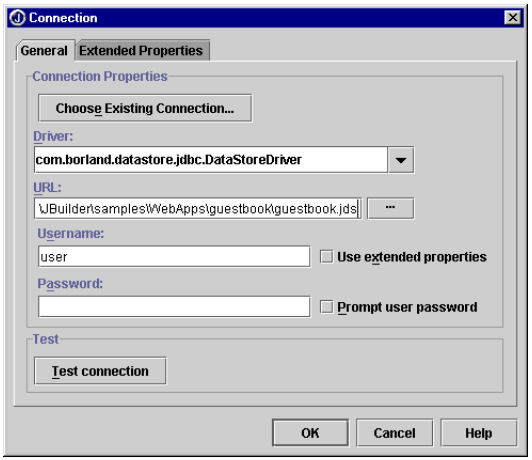
5 Click the Local DataStore Database option.

6 Choose 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, enter `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` will display to the right of the button. If the connection is not successful, an error message will attempt 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, enter:

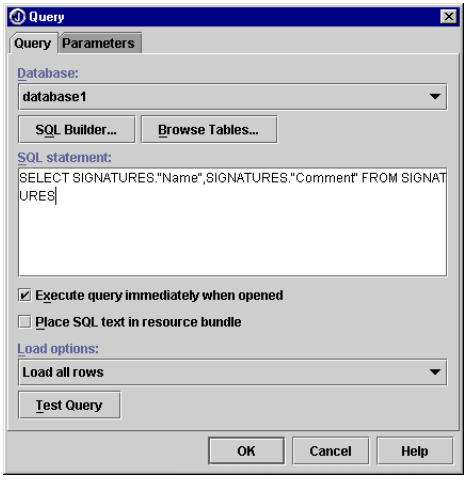
```
SELECT SIGNATURES."Name",SIGNATURES."Comment" FROM SIGNATURES
```

This query loads all the values in the Name and Comment field 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` will display to the right of the button. If the query cannot be executed, an error message will attempt to explain why the query failed.

8 Click OK to close the Query dialog box.

9 Click the Source tab to switch back to the editor.

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 will fix this in a later step. Right-click the Designer tab at the bottom of the AppBrowser and choose Remove "Designer" Tab to remove the tab.

10 Choose File | Save All or click  on the toolbar to save your work.

JBuilder adds the `queryDataSet1.setQuery()` method to the `jbInit()` method.

In the next step, you'll set up the data connection to the `DBServlet`.

Step 6: Creating the data connection to 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 Find the line of code in the class definition that reads:

```
private static final String CONTENT_TYPE="text/html"
```

3 Position the cursor after this line of code and add the following line of code:

```
DataModule1 dm = guestbookservlet.DataModule1.getDataModule();
```

4 Click the Save All icon on the toolbar to save your work.

Now that the servlet is connected to the data module, we 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 7: Adding an input form to FormServlet

In this step, you will add code to the doGet() method of FormServlet. This code will create a form, using the HTML <form> tag. The form will read in two values - UserName and UserComment - entered by the user. This data will be 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.
2. Find the doGet method. It is near the top of the file.

Tip You can search by positioning the cursor in the structure pane and typing doGet.

3. Remove the following lines of code from the doGet() method:

```
out.println("<font color=\"green\">);
out.println("<p>The servlet has received a GET. This is the reply.</p>");
out.println("</font>");
```

4. Add the following lines of code to the doGet() method, between the open and close <body> tags:

```
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>");
```

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.

In the next step, you'll add code that connects DBServlet to the data module.

Step 8: Adding code to connect DBServlet to the data module

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.
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. Add the following lines of code to the doPost() method:

```
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.

5. Click the Save All icon on the toolbar to save your work.

In the next step, you'll add code to DBServlet that renders the Guestbook table, including the newly added row, in HTML.

Step 9: 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.

Insert the following code after the servlet'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.queryDataSet1.getTableName() + "</h2>");
  Column[] columns = dm.queryDataSet1.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.queryDataSet1.first();
  while (dm.queryDataSet1.inBounds()) {
    out.print("<tr>");
    for (int i = 1; i < columns.length; i++) {
      out.print ("<td>" + dm.queryDataSet1.format(i) + "</td>");
    }
    out.println("</tr>");
    dm.queryDataSet1.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.

In order to ensure that this servlet will compile, check the `import` statements at the top of the file. Add the following packages to the list of imports:

```
import com.borland.dx.dataset.*;
import com.borland.dx.sql.dataset.*;
import com.borland.datastore.*;
```

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 method declaration:

```
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.queryDataSet1.getTableName() + "</h2>");
```

The next line calls the `queryDataSet1.getColumns()` method to get the column names, and return them as an array.

```
Column[] columns = dm.queryDataSet1.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, we are only displaying the second and third columns of the JDataStore. We are 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.queryDataSet1.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.queryDataSet1.inBounds()) {
  out.print("<tr>");
    for (int i = 1; i < columns.length; i++) {
      out.print ("<td>" + dm.queryDataSet1.format(i) + "</td>");
  out.println("</tr>");
  dm.queryDataSet1.next();
}
out.println("</table>");
out.println("</body>");
out.println("</html>");
```

Step 10: Adding business logic to the data module

We're almost done. Right now, the program doesn't do anything because 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.
2. Before you can insert or save data, you must open the dataset. In our data module, we'll open it right after the code that connects to the database and sets up the query. To do this, find the `jbInit()` method, using the Search | Find command. Add the following code before the method's closing curly brace:

   ```
   queryDataSet1.open();
   ```

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 After the `insertNewRow()` method, add the following method to save the new row to the dataset:

```
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 11: Compiling and running your project

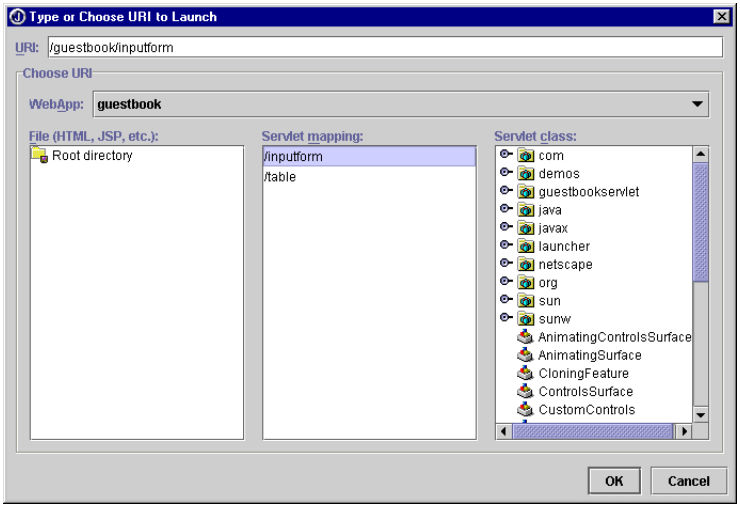
To set run properties for the project,

1 Choose Project | Project Properties. Click the Run tab and then click JSP/Servlet tab of the Run page.

2 Click the ellipsis button to the right of the Launch URI button to display the Type Or Choose URI To Launch dialog box. You will choose the name of the servlet to launch. This is the name you entered into the Name field on Step 3 of the Servlet wizard for `FormServlet`.

3 In the Servlet Mapping directory tree in the middle of the dialog box, choose `/inputform`. The URI field at the top of the dialog box will contain: `/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 similar to this:



4 Click OK to close the Type Or Choose URI To Launch dialog box, then OK again to close the Project Properties dialog box.

5 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.

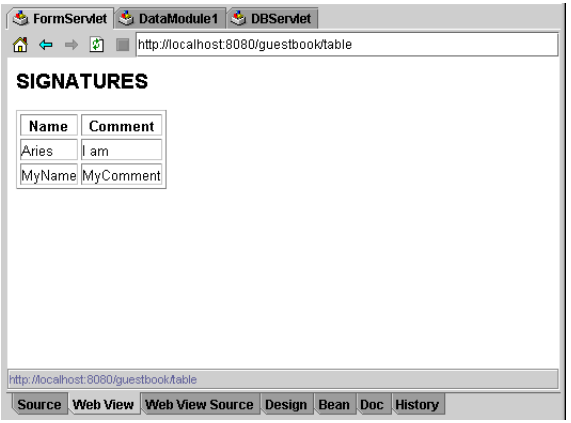
The Tomcat web server is displayed in the message pane.

3 `FormServlet's` input form is displayed in the web view.

4 Enter `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 is `/guestbook/inputform/` and matches what you selected in the URI Launch dialog box. If you ran the servlet by right-clicking `FormServlet.java` and choosing Web Run, you'd see a different URL: `guestbook/servlet/guestbookservlet.FormServlet`. For more information on URLs, URIs, and servlets, see "How URLs run servlets" on page 15-3.

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 To stop the web server, click the Reset Program button directly above the web server tab. 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 to verify that the new data was saved to the table.

Congratulations! 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

9

Developing JavaServer Pages

Web Development is a feature of JBuilder Professional and 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, all 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 that offers 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 other 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 will be easier to figure out what the problem is 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.

Chapter 10, "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 the topic "Additional JSP resources" on page 9-5.

The JSP API

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:

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 does not 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 does not 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`.

JSPs in JBuilder

JBuilder provides a complete development system for JSPs, including a JSP wizard for creating a new JSP, CodeInsight for JSP-specific tags, debugging within the JSP file, and testing and running the JSP on the Tomcat servlet engine from within the JBuilder development environment.

The JSP wizard

JBuilder provides a JSP wizard. This is a logical starting point when developing a JSP. This wizard can be found by clicking the Web tab of the object gallery.

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

In the JSP wizard, you can specify which WebApp your JSP is a part of, whether to set the JSP up to use the InternetBeans tag library, and whether your JSP uses any JavaBeans. For more information on the features of the JSP wizard, see the topic on the JSP wizard in the online help.

Chapter 10, "Tutorial: Creating a JSP using the JSP wizard" shows you how to create a JSP using the JSP wizard as a starting point.

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 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.

Compiling a JSP

See "Compiling your servlet or JSP" on page 15-2 for information on compiling your JSP.

Running a JSP

See "Running your servlet or JSP" on page 15-5 for information on running your JSP.

Debugging a JSP

See “Debugging your servlet or JSP” on page 15-13 for information on debugging your JSP.

Deploying a JSP

See “Deploying your web application” on page 16-1 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 does not maintain these Web sites and can not be responsible for their content or longevity.

- Get a JSP syntax card in HTML format, from http://www.java.sun.com/products/jsp/tags/tags.html.
- Get a JSP syntax card in PDF format, viewable with Adobe Acrobat Reader, from http://www.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://www.java.sun.com/products/jsp/faq.html.
- The current JSP specification can be viewed from the JavaSoft Web site, at http://www.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 Web site with many links to JSP topics, entitled “Web Development with JavaServer Pages”, is http://www.burridge.net/jsp/jspinfo.html.
- 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 JavaSoft Web site (http://www.javasoft.com/) for information.

Chapter

10

Tutorial: Creating a JSP using the JSP wizard

Web Development is a feature of JBuilder Professional and Enterprise.

This tutorial walks you through developing a 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 will not generate a complete application, but it will 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, we use Tomcat. 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.

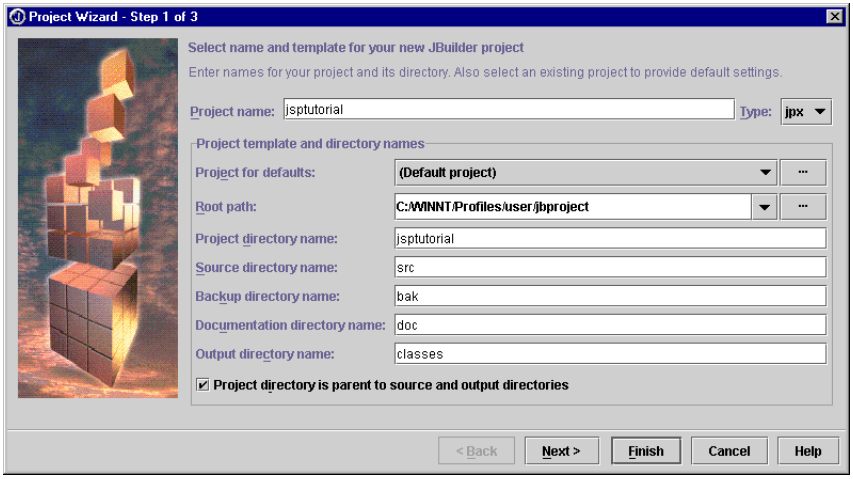
This tutorial assumes you are familiar with Java and with the JBuilder IDE. For more information on Java, see *Getting Started with Java.* For more information on the JBuilder IDE, see "The JBuilder environment" in *Building Applications with JBuilder.*

For suggestions on improving this tutorial, send email to jpgpubs@borland.com.

Step 1: Creating a new project

1. Select File | New Project to display the Project wizard.
2. In the Project Name field, enter a Project name, such as `jsptutorial`.
3. Click Finish.
4. A new project is created, containing an HTML file for describing the project.

Figure 10.1 Project wizard



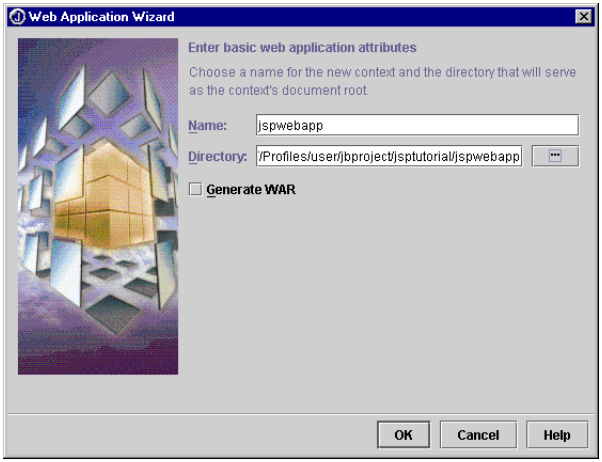
Step 2: Creating a new WebApp

This step is optional, but advisable. 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 WebApp wizard appears.
4. Enter a name for the WebApp, such as `jspwebapp`.
5. Click the ellipsis button to the right of the Directory field.
6. Enter a directory name for the WebApp's root directory, such as `jspwebapp`.
7. Click OK.
8. Click Yes to create the directory.

9 Leave Generate WAR unchecked, since you probably won't want to actually deploy this tutorial application.

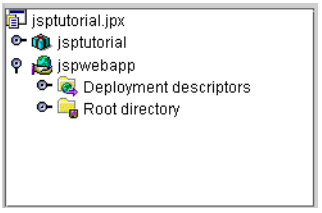
10 The wizard should look something like this:



11 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 10.2 WebApp node in project pane



Step 3: Using the JSP wizard

1 Select File | New.

2 Click the Web tab. Select JavaServer Page.

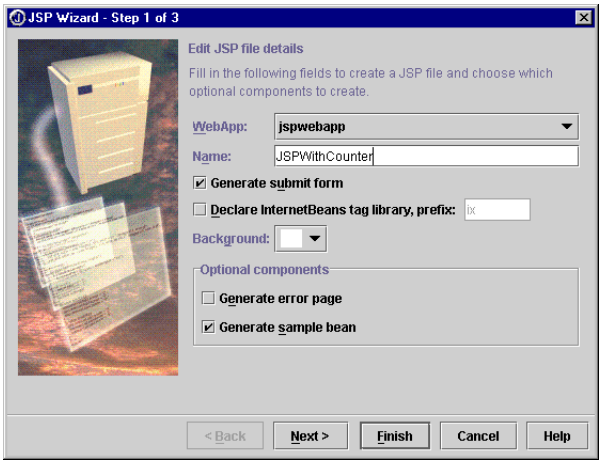
3 Click OK. The JSP wizard appears.

4 Select the WebApp for the JSP from the WebApp drop-down list. You will want to select `jspwebapp` if you created this WebApp in step 2.

5 Enter a name for the JSP: `JSPWithCounter`

6 Click Finish to accept all the default settings.

Figure 10.3 JSP 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.

Step 4: 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.

Double-click `JSPWithCounterBean.java` in the project pane. Modify the source code as follows, entering 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;
  }

}
```

Step 5: Modifying the JSP code

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. Select the Source tab. You can use CodeInsight and JSP source highlighting to help with coding. Modify the generated file as follows, adding the code in **bold**. JBuilder's CodeInsight will activate to help with the syntax.

```
<html>
<head>
<jsp:useBean id="JSPWithCounterBeanId" scope="session"
   class="jsptutorial.JSPWithCounterBean" />
<jsp:setProperty name="JSPWithCounterBeanId" property="*" />
<title>
JspWithCounter
</title>
</head>
<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>
```

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 "The JSP API" on page 9-3.

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.

Select File | Save All to save your work.

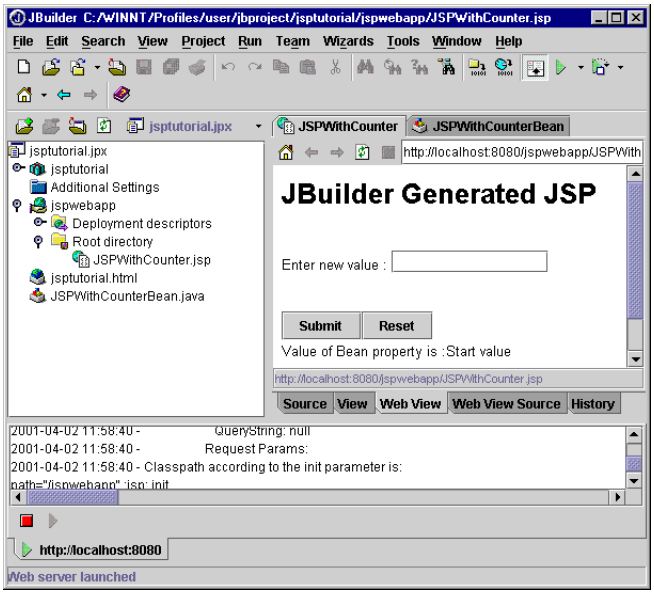
Step 6: Running the JSP

In order to run the JSP, the runtime properties for the project must be set correctly. In this tutorial, the runtime properties were already set by the JSP wizard, so you can go ahead and run the JSP. Right-click the JSP file and select Web Run from the menu.

The project compiles and runs. Compilation errors are displayed in the message pane. If there are errors, refer to the topic "Debugging your servlet or JSP" on page 15-13.

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. JBuilder's default web server is Tomcat, 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 10.4 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, enter a value in the text field, then 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 will appear 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. Run time properties of the Web server may be set by selecting Project | Project Properties, and selecting JSP/Servlet on the Run tab. The port number is the port on which the web server will run. The default is 8080. If port 8080 is in use, by default JBuilder will search for an unused port.

For more information on setting run parameters for your servlet or JSP, see "Setting run parameters for your servlet or JSP" on page 15-9.

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 will behave 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'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 "Debugging your servlet or JSP" on page 15-13.

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 16, "Deploying your web application."

According to the JSP FAQ at http://www.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 www.java.sun.com.

Chapter

11

Using InternetBeans Express

Web Development is a feature of JBuilder Professional and 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

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 will use a `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 checkbox. XHTML: `<input type="checkbox" />`
 IxComboBox	Represents a combobox. 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 will fire.
 IxListBox	Represents a listbox. XHTML: `<select size="3" />`

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 radiobutton. 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 will fire.
 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 will 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 will now be 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 a 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 lower-case by definition.

To make things faster, HTML element and attribute names will be 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, their attributes are merged, with properties set in the InternetBeans Express component taking 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 designer creates 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 12, "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.

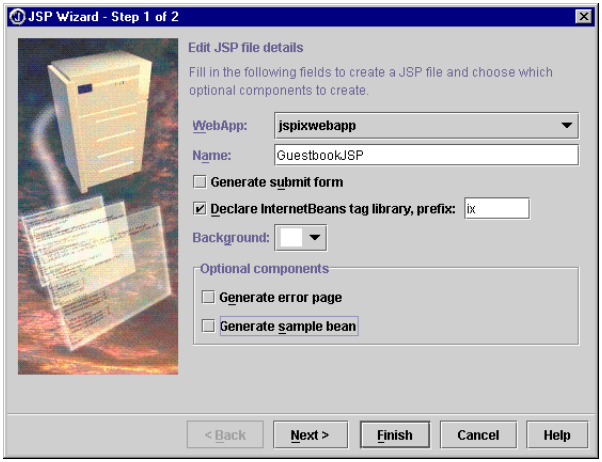
JBuilder's JSP wizard inserts a `taglib` directive and a `page` directive for you if you check the Declare InternetBeans Tag Library, Prefix: option. Type in the prefix you want to use next to this option. This prefix will then be used with every InternetBeans tag in your JSP to identify it as an InternetBeans

tag. If this option is checked, the JSP wizard also completes the other necessary steps for adding the InternetBeans library to your project. These steps are as follows:

- It copies the entire `internetbeans.jar` file into the WebApp's `/WEB-INF/lib` directory
- It adds a mapping between the `internetbeans.tld` file and `internetbeans.jar` to the web.xml file. There is a standard location in any tag library JAR (`/META-INF/taglib.tld`) that contains the tag library descriptor. That copy is used for JSP compiling and running.

The JSP wizard also adds an `internetbeans.tld` file to the project root. It actually points to that file inside the copied JAR. Because it's inside the JAR, it is read-only.

The Declare InternetBeans Tag Library option is displayed in the JSP wizard as follows:



Here is an example of how an InternetBeans tag will look 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. Note that the `ix` prefix could be any text. It all depends on what prefix you specified in the JSP wizard. 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 is not required, but it makes the JSP code more readable.

For a tutorial on this topic, see Chapter 13, "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.

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` • `password`
query	Defines a DataExpress `QueryDataSet`	• **`id`** - text used to identify this query • `database` - identifies the database to which this query belongs. This is not required because it's implied if the query tag is nested within the database tag. If the query tag is not 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.

Tag name	Description	Attributes
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 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 its 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

12

Tutorial: Creating a servlet with InternetBeans Express

Web Development is a feature of JBuilder Professional and 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, with the JBuilder IDE, and with JDataStore. For more information on Java, see *Getting Started with Java.* For more information on Java servlets, see "Working with servlets" in the *Web Application Developer's Guide.* For more information on the JBuilder IDE, see "The JBuilder environment" in *Building Applications with JBuilder.* For more information on JDataStore, see the *JDataStore Developer's Guide*.

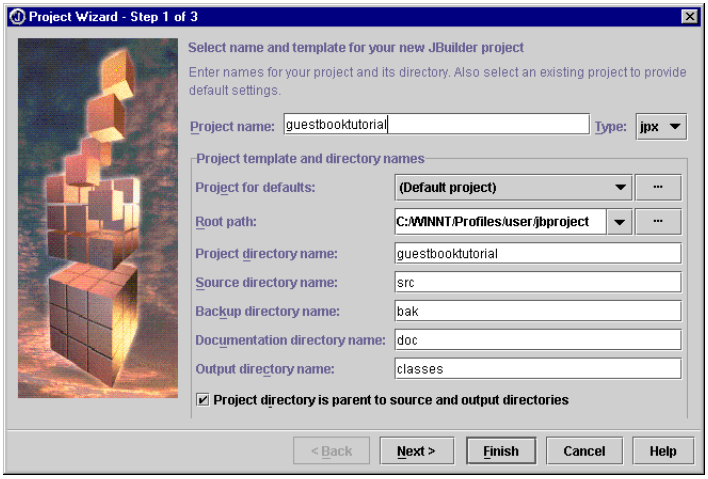
Note This tutorial assumes that you have entered your licensing information into the JDataStore License Manager. For more information, see "Using JDataStore for the first time" in the *JDataStore Developer's Guide.*

For suggestions on improving this tutorial, send email to jpgpubs@borland.com.

Step 1: Creating a new project

1. Select File | New Project to display the Project wizard.
2. Enter a Project name, such as `guestbooktutorial`.
3. Click Finish.
4. A new project is created, containing an HTML file for describing the project.

Figure 12.1 Project wizard



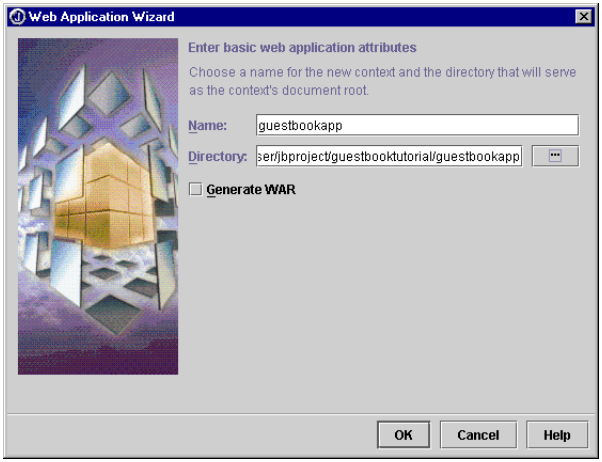
Step 2: Creating a new WebApp

This step is optional, but advisable. 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 WebApp wizard appears.
4. Enter a name for the WebApp, such as `guestbookapp`.
5. Click the ellipsis button to the right of the Directory field.
6. Enter a directory name for the WebApp's root directory, such as `guestbookapp`.
7. Click OK.
8. Click Yes to create the directory.

9 Leave Generate WAR unchecked, since you probably won't want to actually deploy this tutorial application.

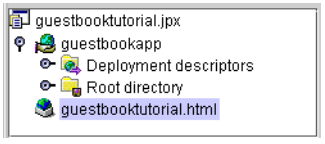
10 The wizard should look something like this:



11 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 12.2 WebApp node in project pane



Step 3: Using the Servlet wizard

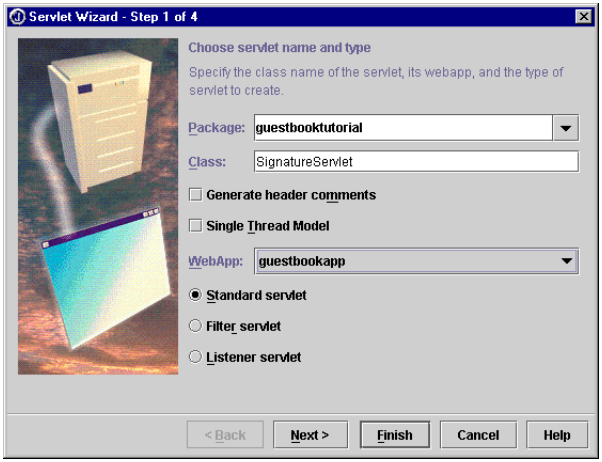
1 Select File | New.

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. The wizard should look something like this:

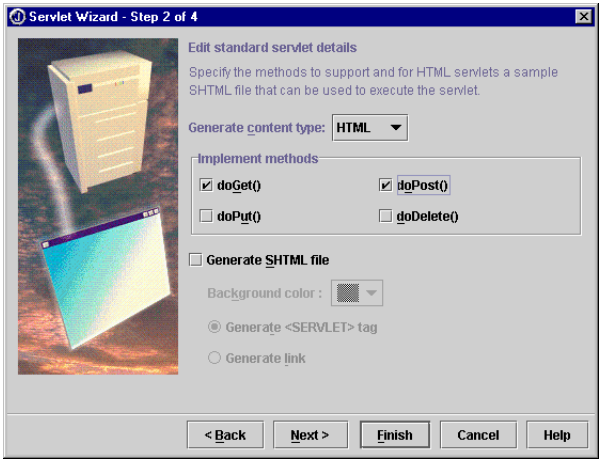


6 Click Next to proceed to Step 2 of the wizard.

7 Make sure Generate Content Type 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:

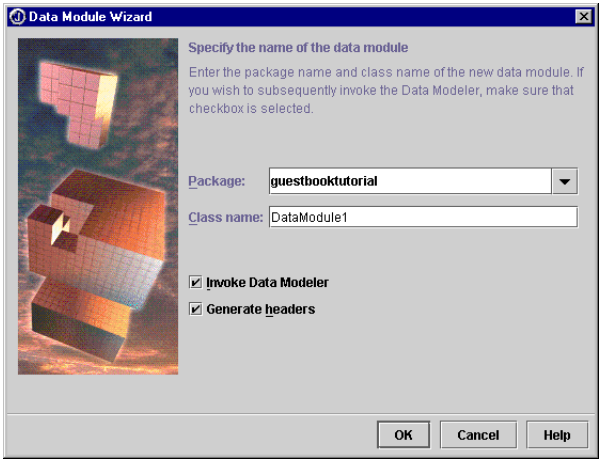


10 Click Finish. A `SignatureServlet.java` file is added to your project.

11 Click the Save All button on the toolbar.

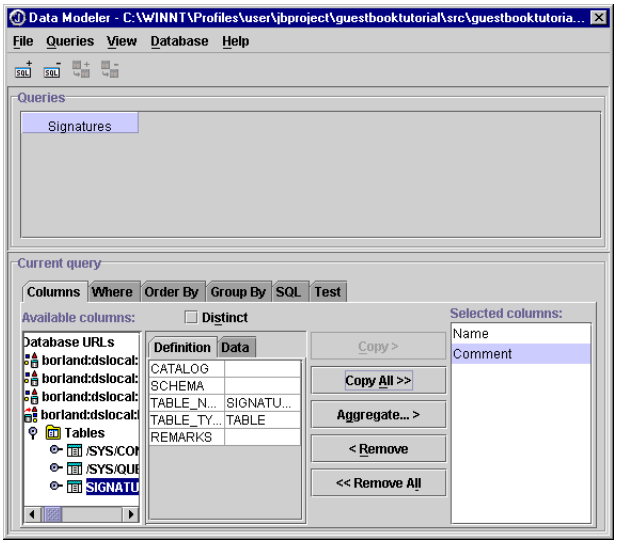
Step 4: Creating the DataModule

1. Select File | New.
2. Select Data Module from the New page of the object gallery.
3. Click OK. The Data Module wizard appears.



4. Make sure Invoke Data Modeler is checked.
5. Click OK. The Data Modeler appears.
6. Go to the Database menu and select Add Connection URL.
7. For the driver, select `com.borland.datastore.jdbc.DataStoreDriver` from the drop-down list.
8. For the URL, type the path to the `guestbook.jds` file found in the `<JBuilder>/samples/WebApps/guestbook` folder.
9. Click OK. A new Database URL is added.
10. Connect to the new Database URL by clicking on it and entering a `user` in the login dialog. A password is not necessary.
11. Open the list of tables by clicking on the Tables node of the Available Columns tree.
12. Select the SIGNATURES table.

13 Click the Copy All button. The wizard should look something like this:



14 Choose Save from the File menu.

15 Choose Exit from the File menu. The `DataModule1.java` file is updated with the required connection information.

16 Click the Save All button on the toolbar.

Step 5: Designing the HTML template page

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 in a File name: `gb1.html`

5 Click OK.

6 Click OK again to create the file.

7 Double-click the file in the project pane to open it. You will see a warning message indicating that the file cannot be opened. This is because the file has not yet been saved. You can safely ignore this warning.

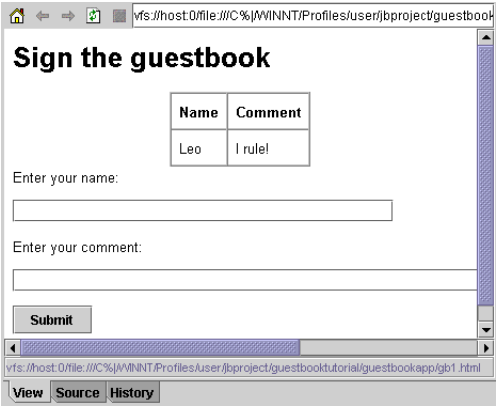
8 Click the Source tab to open the HTML source. It will be blank at this point.

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>
```

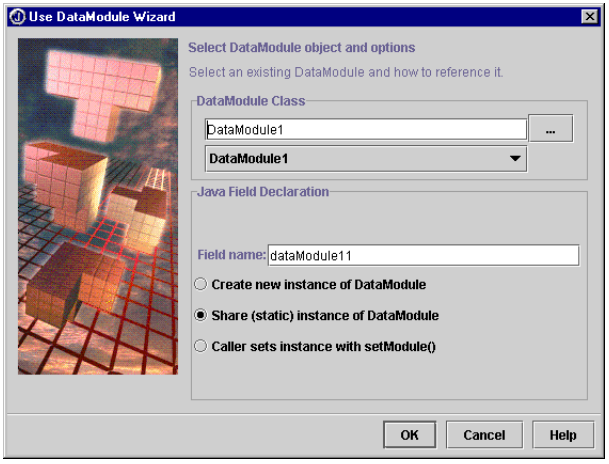
Notice that the `<table>` tag contains dummy data. This data will be replaced with live data from the JDataStore when the servlet is run. This dummy data provides an indication of how the live data should look when it is rendered. For more information on how InternetBeans Express renders tables, see "Generating tables" on page 11-6.

Click the Save All button on the toolbar. Click the View tab. The HTML should look like this in the View:



Step 6: Connecting the servlet to 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.
4. Click the ellipsis button next to the DataModule class field.
5. Select `DataModule1` from the `guestbooktutorial` package.
6. Make sure Share (static) Instance of DataModule is checked. The wizard should look something like this:



7. Click OK.
8. A line of code is added to the `jbInit()` method to associate the `DataModule` with the servlet.

Step 7: Designing the servlet

In this step, you will use the designer to add InternetBeans components to the servlet. These components will not 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 the `SignatureServlet.java` file 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 `ixPageProducer.dataModule` property to `DataModule11`.

6 Set the `ixPageProducer.htmlFile` property to point to the `gb1.html` file (including the path).

7 Select the `IxControl` icon in the palette. Drop an `IxControl` into the servlet by clicking in the designer.

8 Drop two more `IxControls` into the servlet by repeating the previous step twice.

9 Select `ixControl1`.

10 Set the `ixControl1.dataSet` property to the Signatures dataset shown in the drop-down list.

11 Set the `ixControl1.columnName` property to “Name”.

12 Set the `ixControl1.pageProducer` property to `ixPageProducer1`.

13 Set the `ixControl1.controlName` property to “Name”.

14 Select `ixControl2`.

15 Set the `ixControl2.dataSet` property to the Signatures dataset shown in the drop-down list.

16 Set the `ixControl2.columnName` property to “Comment”.

17 Set the `ixControl2.pageProducer` property to `ixPageProducer1`.

18 Set the `ixControl2.controlName` property to “Comment”.

19 Select the `IxTable` icon from the palette. Drop an `IxTable` into the servlet by clicking in the designer.

20 Set the `ixTable1.pageProducer` property to `ixPageProducer1`.

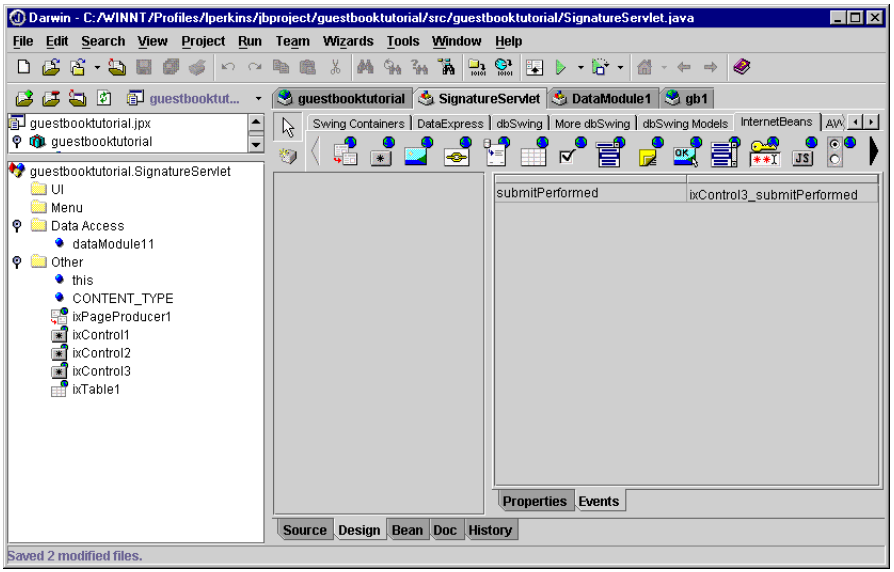
21 Set the `ixTable1.dataSet` property to the Signatures dataset shown in the drop-down list.

22 Set the `ixTable1.elementId` property to “guestbooktable”.

23 Select `ixControl3`.

24 Set the `ixControl3.pageProducer` property to `ixPageProducer1`.

25 Set the `ixControl3.controlName` property to "submit".



26 Click the Events tab in the property inspector.

27 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 Source editor and places the cursor in the new method.

28 Click the Save All button on the toolbar.

Step 8: 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`.

Step 9: Running the servlet

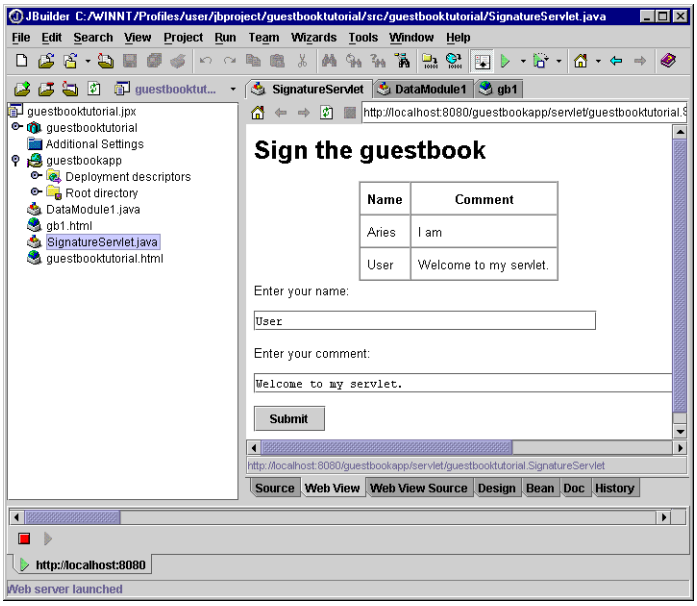
1 Right-click the `SignatureServlet.java` file in the project pane.

2 Select Web Run from the menu.

3 The servlet runs in the JBuilder IDE.

4 Test the servlet. Remove the existing values from the Name and Comment fields. Enter your name and comment and clicking Submit.

5 Your name and comment are displayed in the table, and saved to the JDataStore.



6 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 16, “Deploying your web application.”

Chapter

13

Tutorial: Creating a JSP with InternetBeans Express

Web Development is a feature of JBuilder Professional and 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), with the JBuilder IDE, and with JDataStore. For more information on Java, see *Getting Started with Java.* For more information on JavaServer Pages, see Chapter 9, "Developing JavaServer Pages." For more information on the JBuilder IDE, see "The JBuilder environment" in *Building Applications with JBuilder.* For more information on JDataStore, see *JDataStore Developer's Guide.*

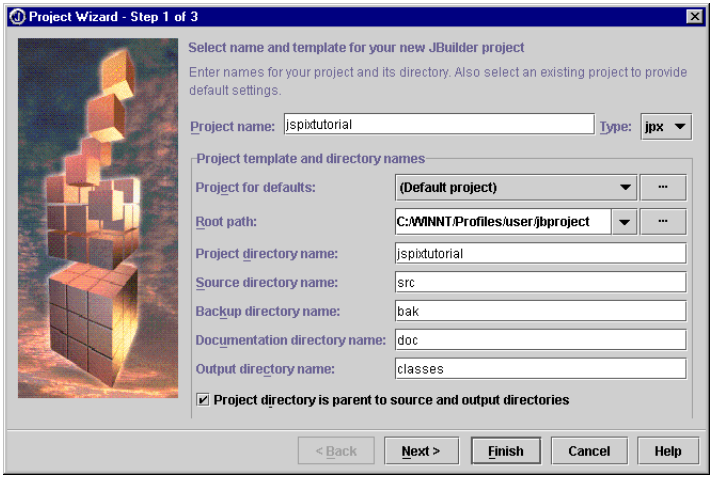
Note This tutorial assumes that you have entered your licensing information into the JDataStore License Manager. For more information, see "Using JDataStore for the first time" in the *JDataStore Developer's Guide.*

For suggestions on improving this tutorial, send email to jpgpubs@borland.com.

Step 1: Creating a new project

1. Select File | New Project to display the Project wizard.
2. Enter a Project name, such as `jspixtutorial`.
3. Click Finish.
4. A new project is created, containing an HTML file for describing the project.

Figure 13.1 Project wizard

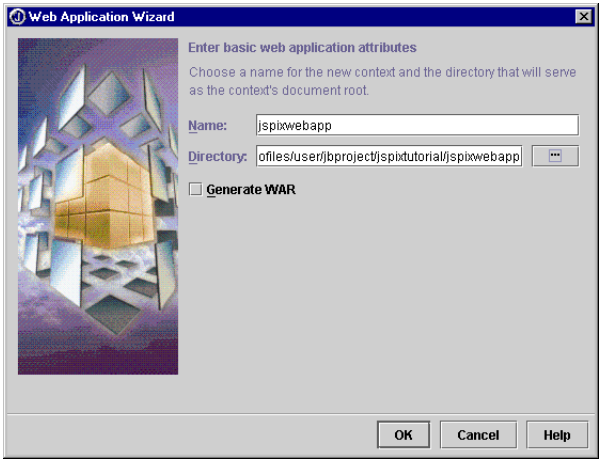


Step 2: Creating a new WebApp

This step is optional, but advisable. 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 WebApp wizard appears.
4. Enter a name for the WebApp, such as `jspixwebapp`.
5. Click the ellipsis button to the right of the Directory field.
6. Enter a directory name for the WebApp's root directory, such as `jspixwebapp`.
7. Click OK.
8. Click Yes to create the directory.

9 Leave Generate WAR unchecked, since you probably won't want to actually deploy this tutorial application. The wizard should look something like this:



10 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 13.2 WebApp node in project pane

Step 3: 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 Select the WebApp from the drop-down list. Select `jspixwebapp` if you created this WebApp in step 2.

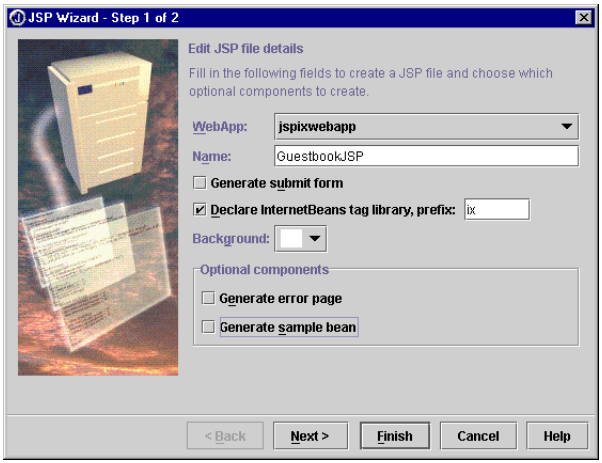
5 Enter a name for the JSP: `GuestbookJSP`

6 Uncheck Generate Submit Form.

7 Check Declare InternetBeans Tag Library, leaving the default prefix of `ix`.

8 Uncheck Generate Sample Bean. The JSP wizard should look like this:

Figure 13.3 JSP wizard



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 also takes care of the necessary steps for adding the InternetBeans library to your project, as described in "Using InternetBeans Express with JSPs" on page 11-6.

Step 4: Designing the HTML portion of the JSP

1 Open the `GuestbookJSP.jsp` file in the editor.

2 Change the contents of the `<title>` tag to read `JSP/InternetBeans Tutorial`.

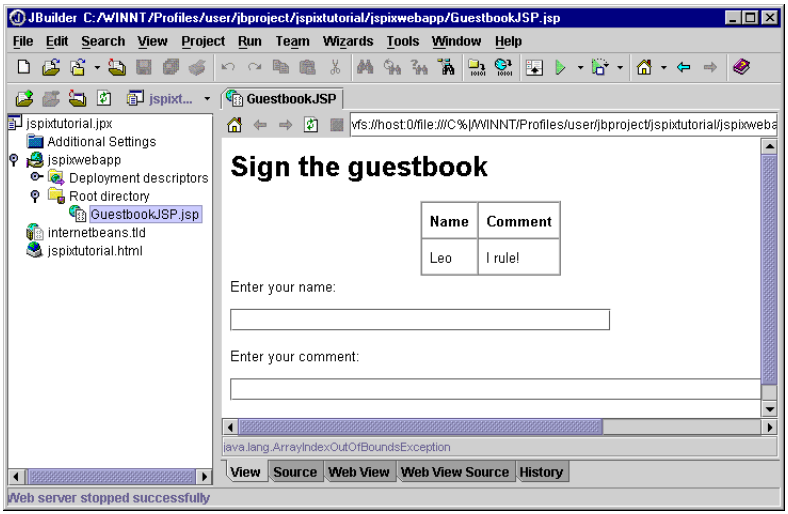
3 Change the contents of the `<h1>` tag to read `Sign the Guestbook`

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:



Step 5: Adding the InternetBeans database tag

1. 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:C:\\JBuilder\\samples\\WebApps\\guestbook\\guestbook.jds"
  username="user">

<table id="guestbooktable" align="CENTER" cellspacing="0" border="1" cellpadding="7">
```

2. Change the path to the `guestbook.jds` as needed in the `url` attribute. The `guestbook.jds` can be found in the `<jbuilder>/samples/WebApps/guestbook` folder.

3. Add the closing `database` tag shown in **bold**.

```
</form>

</ix:database>

</body>
```

Step 6: Adding the InternetBeans query tag

1 Add the opening `query` tag shown in **bold**.

```
<ix:database id="database1"
driver="com.borland.datastore.jdbc.DataStoreDriver"
url="jdbc:borland:dslocal:C:\\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 does not 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>
```

Step 7: Adding the InternetBeans table tag

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 is not 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.

Step 8: Adding the InternetBeans control tags

Now it's time to add the two control tags for the two text input fields. 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.

Step 9: 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. We're not done with the submit button yet. We still have to add the method which will be executed when the button is pushed. We'll do that in the next step.

Step 10: Adding the submitPerformed() method

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 does not 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.

Step 11: Adding code to insert a row

There is still one more piece of code we need to add before the JSP will work properly. When the form is posted, we 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.

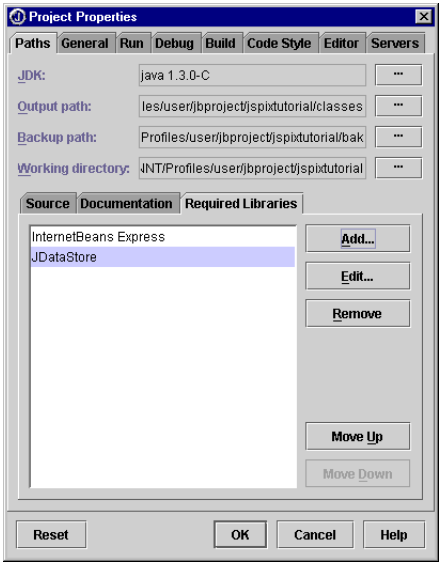
This code fragment inserts a row in the dataset just before the form is displayed. The form displays empty values. Then when the form is posted the data is written to the empty row before calling the `submitPerformed()` method.

Step 12: 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 dialog box from the Project menu.
2. Click the Paths tab.
3. Click the Required Libraries tab.
4. Click Add.
5. Select JDataStore Server.
6. Click OK.
7. Click OK again to close the Project Properties dialog box.

Figure 13.4 Required Libraries tab of Project Properties



Step 13: Running the JSP

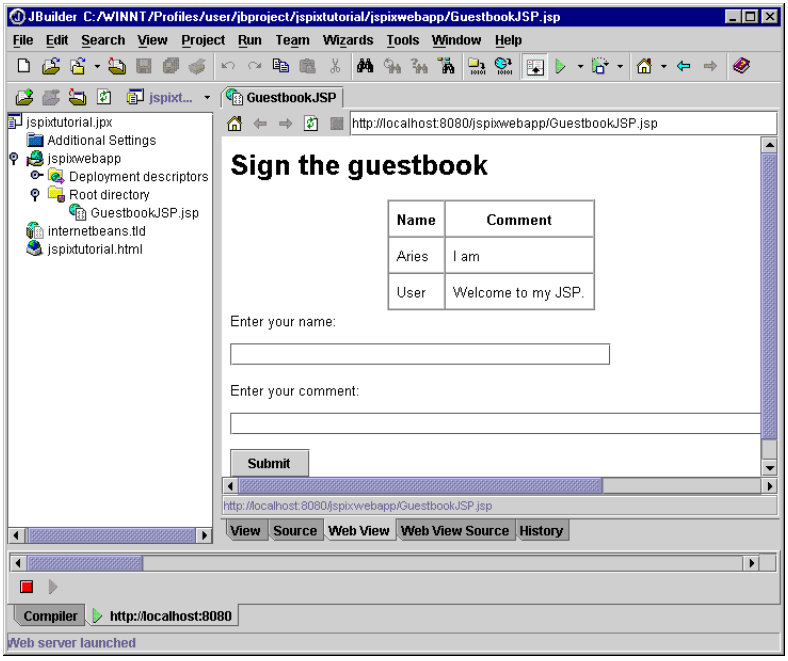
Now it's time to run and test the JSP.

1. Expand the Root Directory node of the WebApp in the project pane.
2. Right-click the `GuestbookJSP.jsp` file in the project pane.
3. Select Web Run from the menu.

   Tomcat is started and the JSP runs within the JBuilder IDE.
4. Enter your name and comment.
5. Click the Submit button.

   Your name and comment are added to the table (and stored in the JDataStore).

Figure 13.5 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 16, "Deploying your web application."

Chapter

14

Configuring your web server

Web Development is a feature of JBuilder Professional and Enterprise.

Both Java servlets and JavaServer Pages (JSP) run inside web servers. Tomcat, the JavaServer Pages/Java Servlets reference implementation, is included "in-the-box" 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 provides plugins for the following web servers:

- Tomcat 3.1
- Tomcat 3.2
- Tomcat 4.0
- Borland Application Server 4.5 (Enterprise only)
- WebLogic 5.1 (Enterprise only)
- WebLogic 6.0 (Enterprise only)

Of these web servers, only Tomcat 3.2 is included with JBuilder. Once you have installed a third-party web server, the plugin allows you to configure that web server for use with JBuilder.

After you've configured your web application and web server, you can compile, run and debug your servlet and JSP. For more information, see Chapter 15, "Working with web applications in JBuilder."

Configuring Tomcat

When you install JBuilder Professional or Enterprise, Tomcat is automatically installed in your JBuilder directory. Paths and libraries are automatically set up for you.

If you want to use Tomcat as provided, you do not have to change any configuration settings. However, if you'd like to examine, and possibly change settings, follow these steps:

Note These steps work for configuring Tomcat 3.1, 3.2 or 4.0.

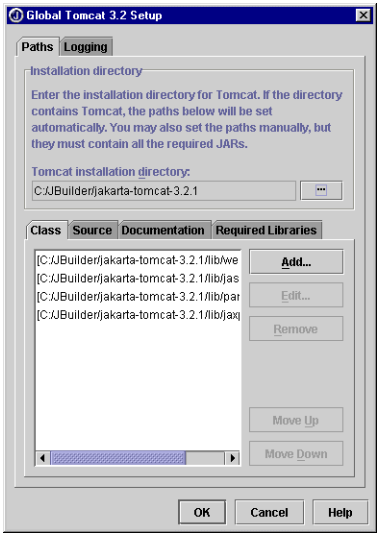
1. Choose Project | Project Properties. On the Project Properties dialog box, choose the Servers tab (Enterprise) or the Web Server tab (Professional).

Note Enterprise users can change the selected application server by clicking the ellipsis button to the right of the AppServer field. This may affect your web server setting. For more information, see "Setting up JBuilder for web servers other than Tomcat" on page 14-4.

2. Choose the Setup button to the right of the Server field.

   The Global Tomcat Setup dialog box is displayed.

3. To set installation options, choose the Paths page. For Tomcat 3.2, the Paths page looks like this:



Note If you are using the version of Tomcat installed with JBuilder, do not change any of these settings. If you want to change settings, follow these steps:

1. Enter the name of the directory where Tomcat is installed in the Tomcat Installation Directory field. To browse to the location, choose the ellipsis button.

2 In the bottom of the dialog box, choose the location of Tomcat's class files, source files, documentation files and required libraries:

Table 14.1 Tomcat setup options

Tab	Description
Class	The location of Tomcat class files. You must include the following `jar` files, or the Web Run/Web Debug commands will fail: `webserver.jar`, `jasper.jar`, `parser.jar`, and `jaxp.jar`. In a standard installation, these files are in the `/lib` directory of JBuilder's Tomcat folder.
Source	The location of Tomcat source files. In a standard installation, these files are in the `/src` directory of JBuilder's Tomcat folder.
Documentation	The location of Tomcat documentation files. In a standard installation, this tab is left blank.
Required Libraries	The libraries that Tomcat requires. In a standard installation, this is the `Servlet` library.

Use the Add, Edit and Remove buttons to update entries in the dialog box.

4 To set log file options, choose the Logging page on the Global Tomcat Setup dialog box. For Tomcat 3.2, the Logging page looks like this:

This page contains three areas. To set log file options for the main server engine, use settings in the Tomcat area of the dialog box. To set options for the servlet container, use settings in the Servlet area. For JSP container log file options, go to the Jasper area.

Table 14.2 Tomcat log file options

Field	Description
Level	The level of informational message the web server will display. Choose from displaying fatal errors, error messages, warning messages, debug messages or informational messages.
Message View	Displays messages in the message pane.
File	The name of the log file. Files are written to the the `log` directory. The path is relative to the project directory.

Note If you'd like more information about Tomcat or would like to run it stand-alone, refer to the `/doc` directory of JBuilder's Tomcat installation.

Setting up JBuilder for web servers other than Tomcat

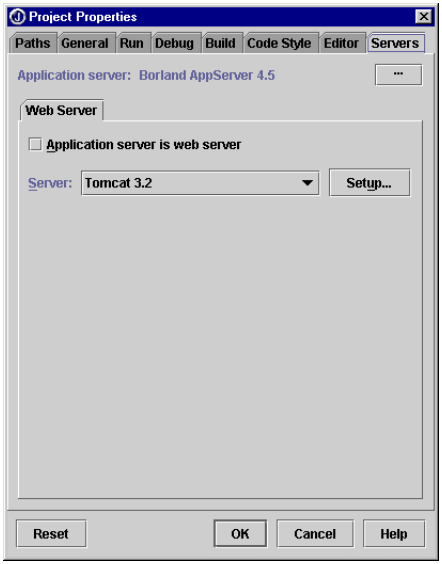
You also use the Project Properties dialog box to choose a web server other than Tomcat for use with JBuilder. Note that the steps for JBuilder Enterprise and JBuilder Professional users differ slightly. Follow the directions in the appropriate section.

- If you want the web server selection to apply to just the current project, choose Project | Project Properties to display the Project Properties dialog box.
- If you want the selection to apply to all projects, choose Project | Default Project Properties to display the Default Project Properties dialog box.

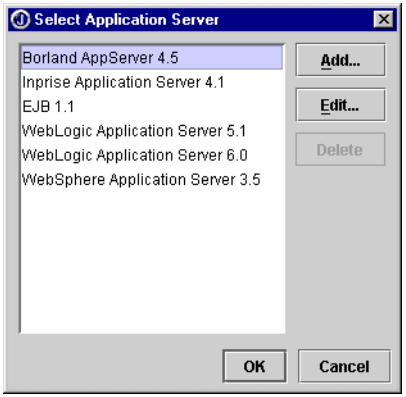
Setting up JBuilder for web servers other than Tomcat (Enterprise users)

When the Project Properties dialog box is open, follow these steps:

1 Choose the Servers tab. The page looks like this:



2 If the application server is also a web server, choose the application server you want to use by clicking the ellipsis button to the right of the Application Server field. The Select Application Server dialog box is displayed:



Select the application server you want and click OK to close the dialog box. For more information on configuring application servers, see "Setting up the target application server" in Part II, "Enterprise JavaBeans Developer's Guide" of the *Enterprise Application Developer's Guide*.

3 If the selected application server is also a web server, select the Application Server Is Web Server option. No more setup is required.

Note This option is enabled only if the application server plugin has registered a web server plugin.

4 To select a web server, choose one from the Server drop-down list. Tomcat is the default web server. For web servers other than Tomcat, choose the Setup button to display server-specific configuration UI.

JBuilder provides plugins for the following web servers:

- Tomcat 3.1
- Tomcat 3.2
- Tomcat 4.0
- Borland Application Server 4.5 (uses Tomcat 3.2 as its web server)
- WebLogic 5.1 (is its own web server)
- WebLogic 6.0 (is its own web server)

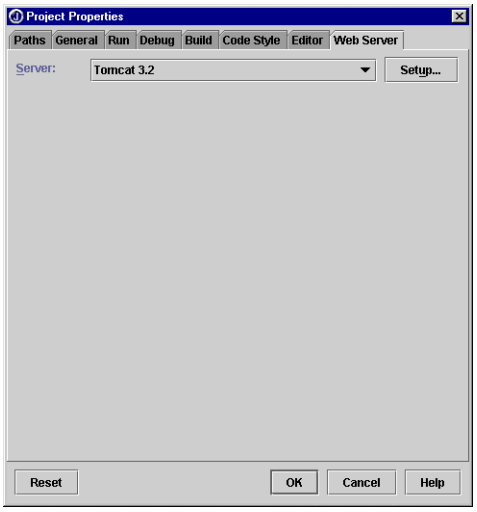
If you've purchased a third-party web server not on this list, you need to contact the vendor in order to obtain the plugin. You can also write a plugin using the OpenTools API. For more information, see "Creating your own web server plugin" on page 14-9.

If a choice in the drop-down list is in red, it is available, but not on the classpath. Click the Setup button to configure the web server.

Setting up JBuilder for web servers other than Tomcat (Professional users)

When the Project Properties dialog box is open, follow these steps:

1 Choose the Web Server tab. The page looks like this:



2 From the Server drop-down list, choose the web server you want to use. By default, this is Tomcat.

3 Choose the setup button to display the Global Tomcat Setup dialog box. (For more information, see "Configuring Tomcat" on page 14-2) For web servers other than Tomcat, the Setup button displays server-specific configuration UI.

Configuring the selected web server

Once you've set up JBuilder with a web server, you can configure options for the web server, including web view options, the name of the web server's host computer, its port number, and how the web server is launched.

Setting web view options

To configure the display of the web view and choose how the web server is launched, choose the Web tab on the IDE Options dialog box (Tools | IDE Options). The Web page looks like this:

1 Choose the Copy Web Run/Debug Launch URL To Clipboard option to copy the URL used to launch the web application to the clipboard. This will enable 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.

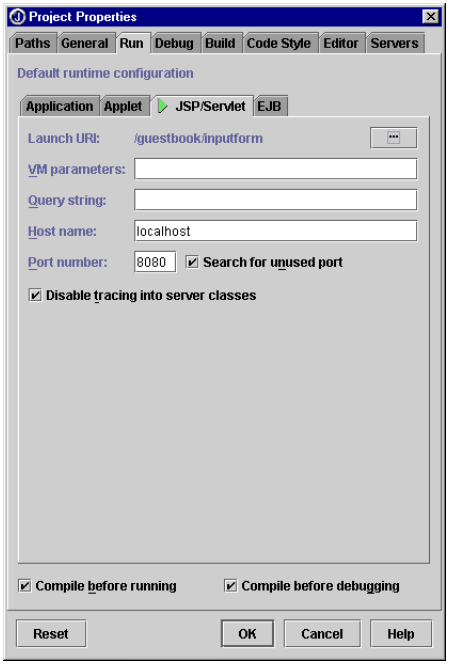
2 Choose Web View Options at the bottom of the page. These options work in conjunction with the Search For Unused Port option on the

JSP/Servlet Run page (Project Properties dialog box) when the specified port is in use by a non-web process. (See "Setting web run options" on page 14-8 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 and in the external web browser.
- Choose 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 will use 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.

Setting web run options

To set options for the web run, choose the JSP/Servlet tab on the Run page of the Project Properties dialog box (Project | Project Properties). The JSP/Servlet Run page looks like this:



1. Enter the name the web server should assume in the Host Name field. Do not choose a name already in use in your sub-net. `localhost` is the default.
2. Enter the port number the web server should listen to in the Port Number field. Use the default port number, `8080`. Change this value only if the default value is in use.
3. Choose the Search For Unused Port option to tell JBuilder to choose another port if the specified one is in use. (The port is only searched for the first time a web run is requested.) 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 will be 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 "Setting web view options" on page 14-7 for more information.)
4. Choose the Disable Tracing To Server Classes option to prevent tracing into server-side classes. For more information about tracing, see "Controlling which classes to trace into" in the "Debugging Java programs" chapter of *Building Applications with JBuilder*.

Note Other options on this page - Launch URI, VM Parameters, and Query String - apply to the runnable JSP or servlet. For more information, see "Setting run parameters for your servlet or JSP" on page 15-9.

Creating your own web server plugin

By default, JBuilder provides Tomcat for running JSPs and servlets. Other web servers may be configured to work with JBuilder. This support can be provided by the web server vendor, a third party, or you can write your own through JBuilder OpenTools API.

An OpenTool web server plugin has to perform the following tasks:

- Register as an OpenTool
- Setup the web server
- Start and stop the web server

This section provides a high-level introduction to the web server plugin OpenTools API. For more detailed information, see the OpenTool `servlet` and `jsp` package documentation.

To get started with JBuilder OpenTools, see the following documents:

- "JBuilder OpenTools Basics" in *Developing OpenTools*
- "JBuilder OpenTools Introduction" in *Developing OpenTools*

For an example of a web server plugin, open the Tomcat project in the `samples/OpenToolsAPI/web/tomcat` directory of your JBuilder installation.

Register as an OpenTool

The setup class for your web server plugin needs to identify a `ServerSetup` implementation as an OpenTool using the `initOpenTool()` method. In the `initOpenTool()` method, it must register an instance of itself with the `ServerManager`. For example, in `Tomcat32Setup.java` (the source file in the sample that sets up Tomcat 3.2 for JBuilder) the registration code looks like this:

```
public static void initOpenTool( byte majorVersion, byte minorVersion ) {
    ServerManager.registerServer( SERVER_NAME, new Tomcat32Setup() );
}
```

Setup the web server

The class that implements the web server plugin needs to implement the `ServerSetup` interface. This interface provides and verifies the setup for the web server and acts as a factory for corresponding `ServerStarter` objects. The implementation must verify that:

- Every time the Web Run or Web Debug commands are called, the server is configured.
- The Servlet API, the web server, the JSP container, and XML support are in the path

The `ServerSetup` class provides a variety of methods for configuration and verification. In the sample, the bulk of the class `Tomcat32Setup.java` provides code that set ups the web server.

Start and stop the web server

Your web server plugin needs to provide code that starts and stops the web server. The `ServerStarter` interface maintains and returns run-time specific information for a particular run of a web server/servlet engine. Each `ServerStarter` object is associated with a `ServerSetup` object, and may delegate to it for items that do not change. The code that starts the web server must:

- Prepare the server for execution by doing server-specific configuration for a particular run.
- Return the main class, any arguments, any VM parameters and the VM working directory.
- Shut down the server and do any required clean up, such as deleting configuration files.

In the sample, the class `Tomcat32Starter.java` provides code that starts and stops the web server.

JSP considerations

There are numerous, additional considerations to consider when creating a plugin. Some of these issues include compiling JSPs to servlets, debugging JSPs, and declaring a JSP tab library. For more information, see the `JSP package` in the OpenTools API Reference documentation.

GUI deployment descriptor editor

Another thing a plugin is encouraged to do, but does not have to, is provide a GUI editor for any server-specific deployment descriptors. For more information, see “Editing vendor-specific deployment descriptors” on page 16-17.

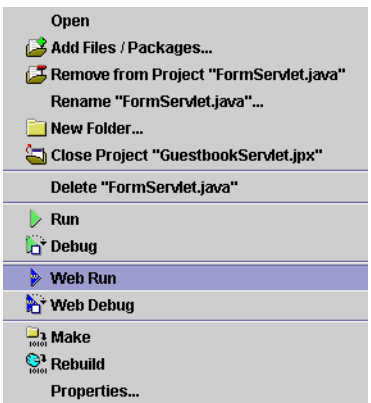
Chapter

15

Working with web applications in JBuilder

Web Development is a feature of JBuilder Professional and Enterprise.

JBuilder provides two commands on the project pane's right-click menu, Web Run and Web Debug, that make it easy to run and debug servlets and JSPs. Web Run runs your web application using the selected web server. Web Debug debugs your JSP or servlet while running on the web server window, allowing you to easily step through and examine your code.



Selecting Web Run or Web Debug will run or debug 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 right-click menu. For more information about WebApps, see "The WebApp and its properties" on page 3-4.

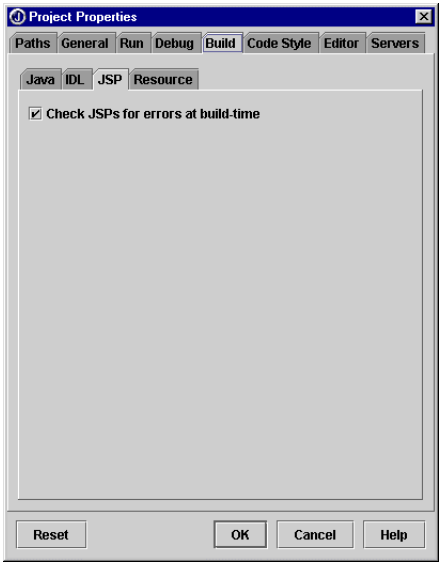
When you create a servlet or JSP using JBuilder's Servlet or JSP wizard, the Web Run and Web Debug commands are automatically enabled.

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 Make or Rebuild. Compiler errors are displayed on the Compiler tab of the message pane.

For more information on the compiler, see the chapter called "Compiling Java programs" in the online Help book *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. Starting with JBuilder 5, 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 JSP tab of the Build page of the Project Properties dialog box (Project | Project Properties).



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.

How URLs run servlets

A URL (Uniform Resource Locator) is used to run a servlet. A URI (Uniform Resource Identifier) 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 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/sub/somename.jsp` would be evaluated as follows:

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

However, the request-URI-path `/test/subtest/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/subtest/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 15.1 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 "Setting run parameters for your servlet or JSP" on page 15-9.

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/jspname.jsp/*` 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/subtest/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:

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

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

- Takes whatever follows `/servlet/` as a class name
- Tries to run that class as a servlet

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

- Takes the URL-path-to-map (that didn't map to anything)
- Finds the corresponding file relative to the context's web-root directory
- 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:

- Creates a servlet with the servlet-name `myservlet` associated with the servlet-class `myproject.MyServlet`
- 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 "Servlet wizard - Naming Options page" on page 6-6.

When you Web Run a servlet `.java` (or `.class`) file, the servlet invoker is used, under the specified WebApp; e.g. `/test/servlet/myproject.MyServlet`.

Running your servlet or JSP

To 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 program in the selected web server without debugging it.

Note If your servlet runs from an HTML or SHTML file, right-click that file and choose Web Run.

Note that you can also create a runtime configuration to run your servlet or JSP. This is more flexible and more permanent than using the Web Run command for each run. To create a runtime configuration,

1. Choose Run | Configurations. In the Runtime Configurations dialog box, choose New to display the Runtime Properties dialog box.
2. In the Configuration Name field, enter a name for your configuration.
3. Choose the JSP/Servlet tab. Enter the run parameters for the configuration. For more information on these options, see "Setting run parameters for your servlet or JSP" on page 15-9.
4. Click OK two times to close the Runtime Properties and Runtime Configurations dialog boxes.
5. To run the configuration, click the down arrow to the right of the Run button on the main toolbar. Choose the configuration you want to run from the drop-down list.

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

Starting your web server

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

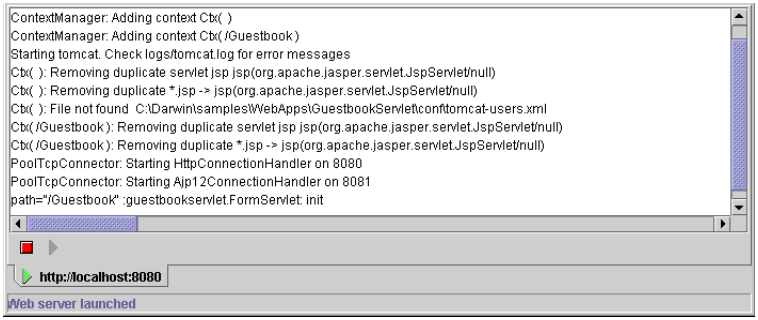
- The parameters set on the Run and Servers pages of the Project Properties dialog box.
- The properties set on the file's Java File Properties dialog box.
- The options set on the Web page of the IDE Options dialog box (Tools | IDE Options).

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 echoed to this pane.

For example, for the Tomcat web server, the following information is displayed:

- Class path
- Home directory
- Context log path
- Tomcat startup message
- Servlet or JSP startup message
- The port number on which the web server is running
- The path to the servlet or JSP

Figure 15.1 Tomcat startup messages

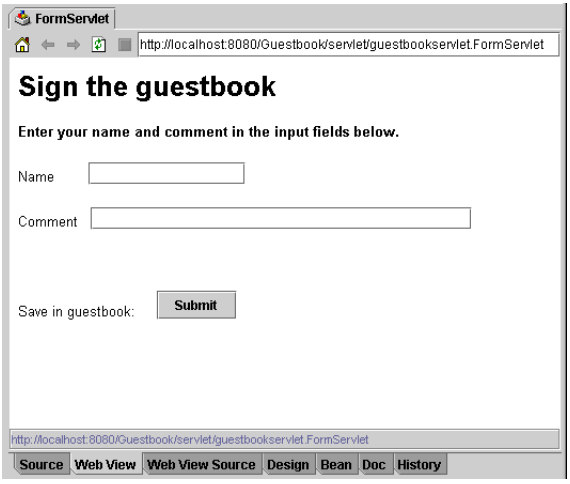


When the Web Run starts, two new tabs are displayed in the content pane: Web View and Web View Source. Click the tab to open the web view and the web view source.

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.

Figure 15.2 Web view output

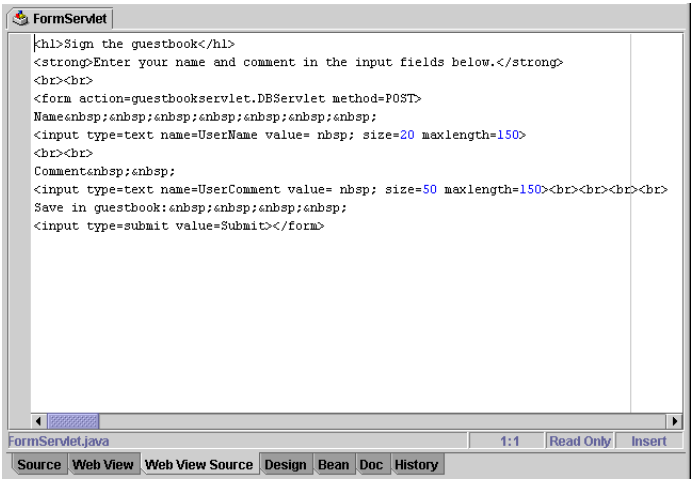


The web view displays the servlet after it has been processed by the servlet engine. There may be a delay between when the servlet or JSP file is edited, and when the change is shown in the web view pane. To see the most recent changes, select the Refresh button at the top of the web view.

Web view source

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

Figure 15.3 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.

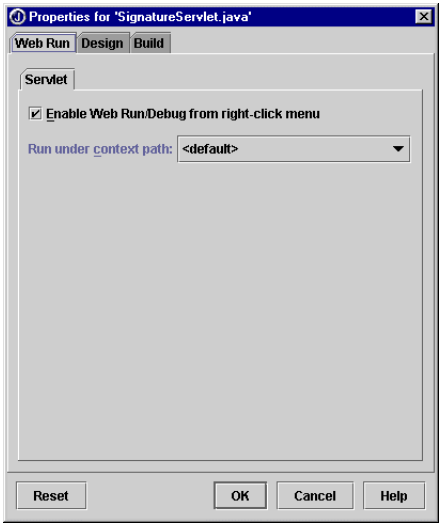
Enabling web commands

If you've created a servlet from scratch, not using the Servlet wizard, you need to enable the Web Run and Web Debug commands. If you're using a JSP or a servlet created with the JSP or Servlet wizard, you don't need to follow these steps. JBuilder has already enabled these commands for you.

To enable the web commands for a servlet,

1. Right-click the `.java` servlet file in the project pane.
2. Choose Properties to display the Java File Properties dialog box.
3. Choose the Servlet tab on the Web Run page.
4. Choose the Enable Web Run/Debug From Right-Click Menu option.

   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.)



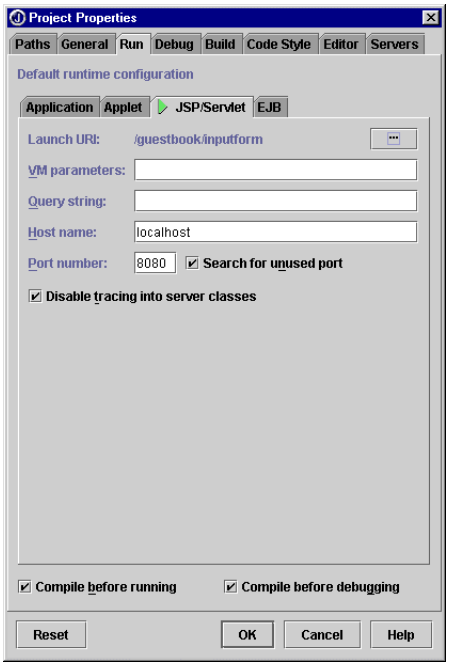
5. Click OK to close the dialog box.

Note Because JSPs have a `.jsp` file extension, the IDE can determine that web commands should be enabled. However, as mentioned above, the JSP must be part of a WebApp in order for the commands to be available.

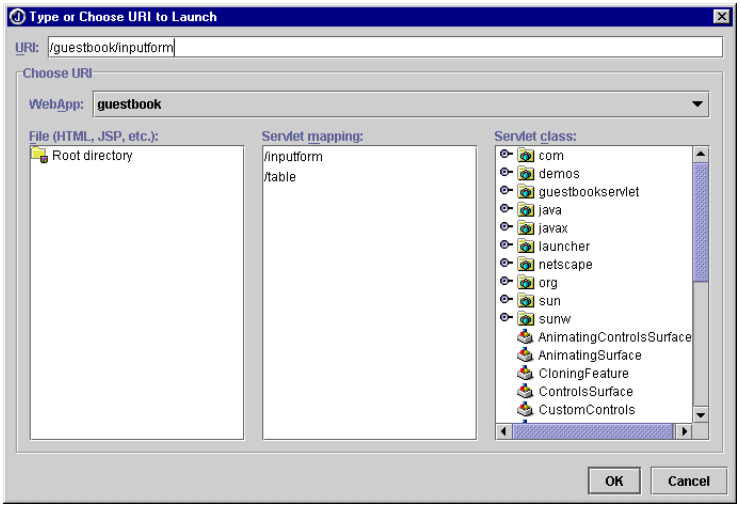
Setting run parameters for your servlet or JSP

Run parameters tell the IDE which file or class to run after the web server starts. You can also enter any parameters to pass to the VM as well as a query string to pass to the client. To set runtime parameters,

1. Choose Project | Project Properties to display the Project Properties dialog box.
2. Select the Run tab. Choose the Servlet/JSP tab. Note that the Run Process icon is displayed on the tab, showing the currently active process. The Run page looks like this:



3 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 use the Type Or Choose URI To Launch dialog box to launch a specific part of a web application. 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:

- 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.
- The three different trees at the bottom of the Launch URI dialog box 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 on the right creates URIs that would match the servlet invoker.

  For more information about URL mappings, see “How URLs run servlets” on page 15-3.

Table 15.2 URI dialog box 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 - Naming Options 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
```

4. Click OK to close the Type Or Choose URI To Launch dialog box.
5. In the VM Parameters field, enter any parameters to pass to the Java Virtual Machine (VM) compiler. For more information on the Java VM compiler and the options you can pass to it, see "Basic Tools: java - The launcher for Java technology applications" at http://java.sun.com/products/jdk/1.2/docs/tooldocs/tools.html.
6. In the Query String field, 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 will generate the following URLs:

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

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

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

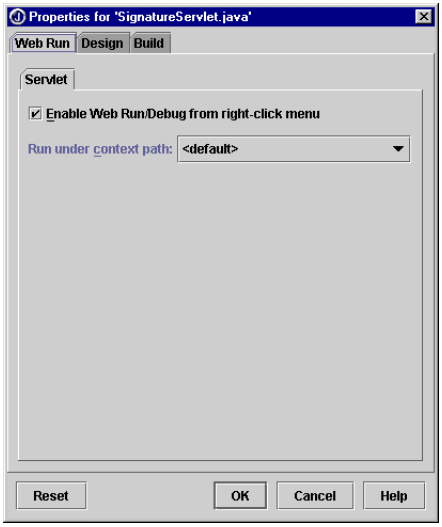
The question mark (`?`) separates the name of the JSP or servlet to run from the query string. This allows the JSP and servlet to ask for the parameter `product` and get back `1234`. Note that parameter values are always strings. Multiple parameters are separated with an ampersand (`&`), e.g. `product=1234&customer=6789`.

Note The remaining fields on this page - Host Name, Port Number, Search For Unused Port and Disable Tracing Into Server Classes - are specific to your web server. These options are explained in the "Setting web run options" on page 14-8.

Setting run properties for a servlet

To set run properties for a servlet,

1. Right-click the servlet you're going to run and choose Properties. The Java File Properties dialog box is displayed.
2. Choose the Servlet tab on the Web Run page. The Servlet page is displayed.



3. Make sure the Enable Web Run/Debug From Right-Click Menu option is enabled. This allows you to run your servlet or JSP in a web server in JBuilder. Note that you can always run from the Launch URI dialog box.

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.

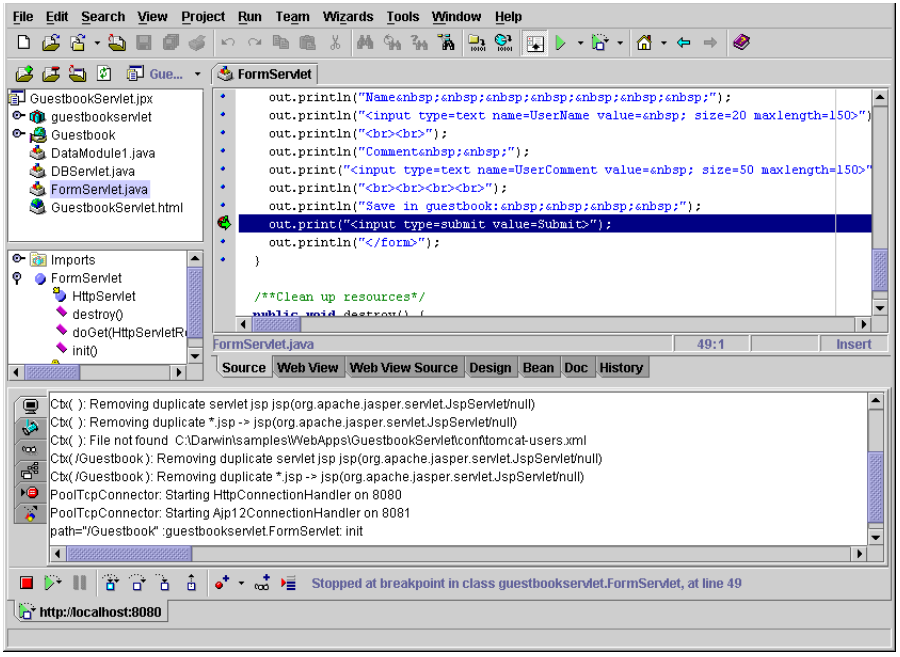
5 Click OK to close the dialog box.

Debugging your servlet or JSP

To debug your servlet or JSP, right-click the file you want to debug and choose the Web Debug command. If you create a runtime configuration (see "Running your servlet or JSP") you can use the debug commands (Debug, Step Over, or Step Into) on the Run menu. To make the debugger stop at a specific line of code, set a breakpoint in your servlet or JSP. The editor is automatically displayed when the breakpoint is hit.

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.

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



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

- "Debugging Java programs" in *Building Applications with JBuilder*
- "Compiling, running and debugging tutorial"
- "Debugging distributed applications" in Part III, "Distributed Application Developer's Guide" in the *Enterprise Application Developer's Guide*

Chapter

16

Deploying your web application

Web Development is a feature of JBuilder Professional and 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 will 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. An archive file, such as a WAR or a JAR file, should organize 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 that they all end up in 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 4, "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 probably 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 4-13 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 the sections on the "Filters page" and the "Listeners page" in the WebApp DD Editor 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
- `tablib-location` - the actual location of the tag library

If you use JBuilder's JSP wizard to create a JSP which uses the InternetBeans tag library, the InternetBeans 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.

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`. Some servers may also have vendor-specific deployment descriptors they use in addition to `web.xml`. For example, 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 are displayed in the structure pane, and the AppBrowser displays the WebApp DD Editor, the Source editor, and the History view. 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. Keep in mind, however, that if you enter comments in the `web.xml` file, these will be removed if you subsequently open the file in the WebApp DD Editor.

The WebApp DD Editor

The WebApp DD Editor is active when the `web.xml` file is opened in the content pane. At the same time, the structure pane will show an outline of the contents of the file. Clicking on the various nodes within the structure pane displays various pages of the editor. There are 12 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
- Resource References
- EJB References
- 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. 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 brings up 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 brings up 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.

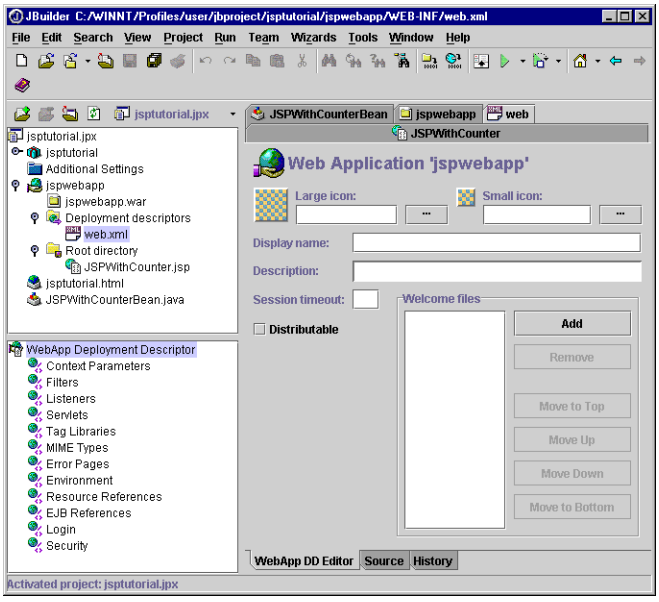
The context menu for an existing servlet, filter, or web resource collection node also contains options to rename or delete the current node. The context menu for an existing security constraint node contains the option to delete the current node (it does not contain a rename option, since security constraints do not have names). 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:

Item	Description
Large icon	Points to the location of a large icon for the WebApp (32 x 32 pixels), which should 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 should be contained within the WebApp's directory tree.
Display name	Name to be displayed for the WebApp.
Description	Description of the WebApp.
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`

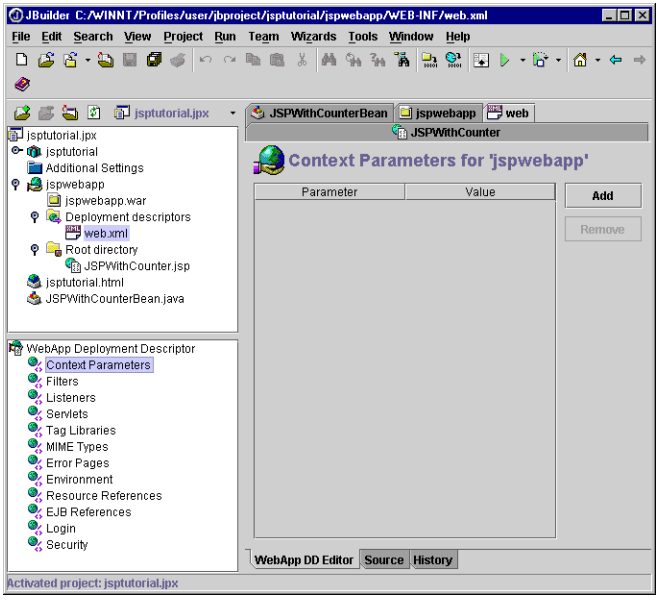
Figure 16.1 WebApp Deployment Descriptor page of WebApp DD Editor



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.

Figure 16.2 Context Parameters page of WebApp DD Editor



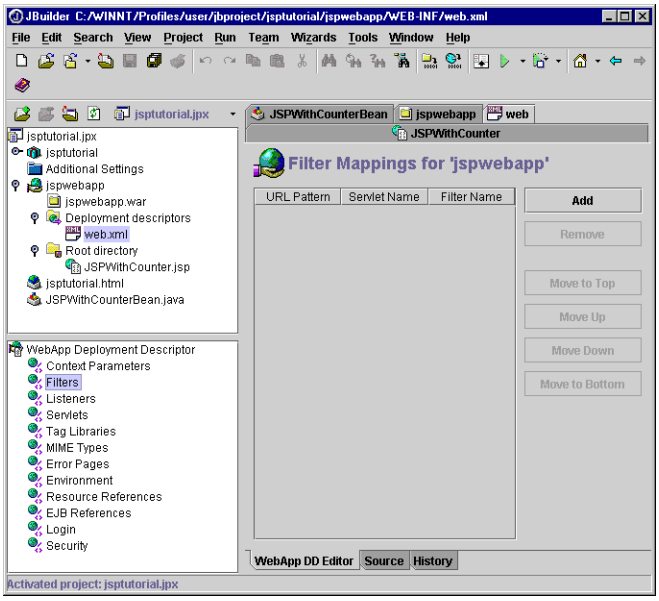
Filters page

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:

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 16.3 Filters page of Webapp DD Editor

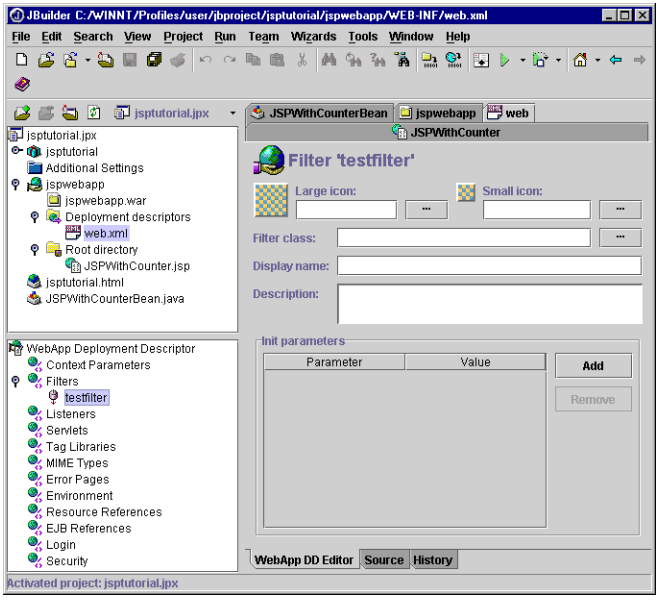


Each individual filter is listed in the structure pane as a separate child node of the Filters node. The filter's `filter-name` is displayed in the tree. You can rename or delete a filter by right-clicking the node for the individual filter and selecting Rename or Delete from the context menu. If you 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:

Item	Description
Large icon	Points to the location of a large icon for the filter (32 x 32 pixels), which should 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 should 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.
Description	Description of the filter.
Init parameters	Initialization parameters for the filter.

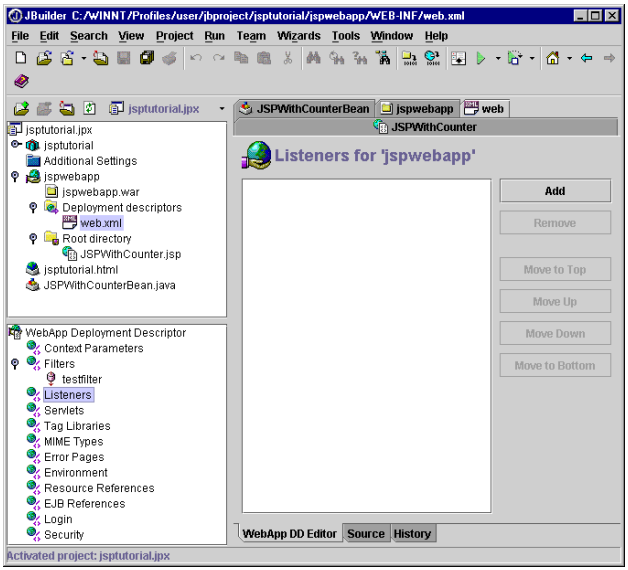
Figure 16.4 Individual filter node in Webapp DD Editor



Listeners page

The Listeners page will only be visible if your web server supports the Servlet 2.3 specification. The Listeners page has a listbox of web application listener bean classes. This information 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.

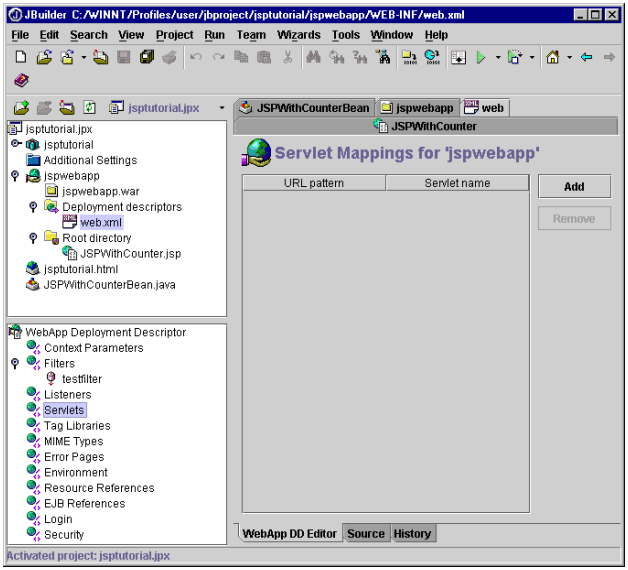
Figure 16.5 Listeners page of Webapp DD Editor



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 16.6 Servlets page of WebApp DD Editor



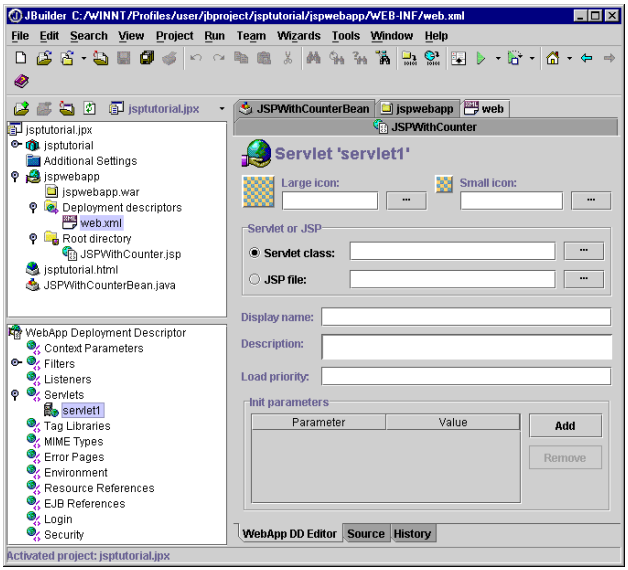
If any servlets are defined, you will find child nodes representing individual servlets underneath the Servlets page node in the structure pane. The servlet's `servlet-name` is displayed in the tree. You can rename or delete a servlet by right-clicking the node for the individual servlet and selecting 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:

Item	Description
Large icon	Points to the location of a large icon for the filter (32 x 32 pixels), which should 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 should be contained within the WebApp's directory tree.
Servlet class	Select this radiobutton for a servlet. Enter the fully qualified class name for the servlet in the text field.
JSP file	Select this radiobutton for a JSP. Enter the path to the JSP file in the text field.
Display name	Name to be displayed for the servlet.
Description	Description of the servlet.
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.
Init parameters	Initialization parameters for the servlet.

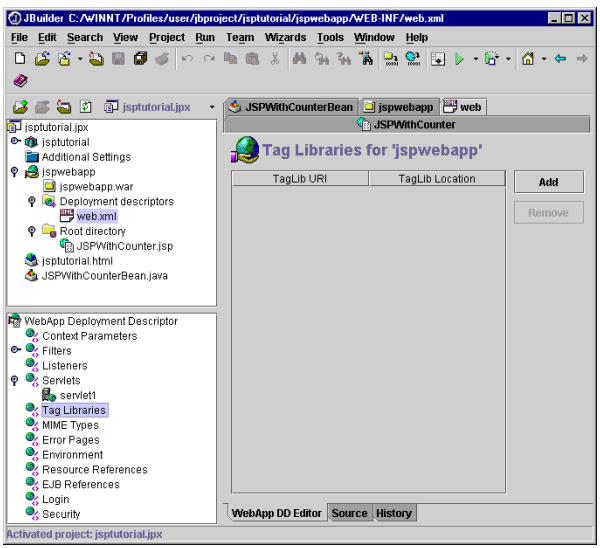
Figure 16.7 Individual servlet node in WebApp DD Editor



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 the InternetBeans tag library, the tag library information for the InternetBeans tag library is filled in for you by the wizard.

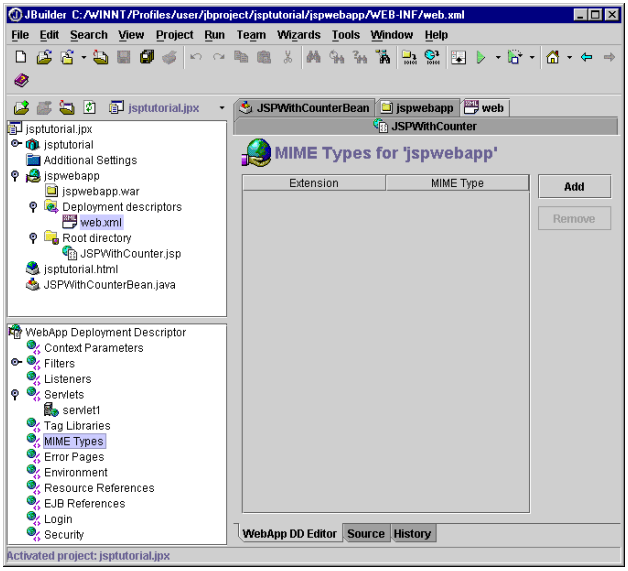
Figure 16.8 Tag Libraries page in WebApp DD Editor



MIME Types page

The MIME Types page has a grid mapping extensions to type names.

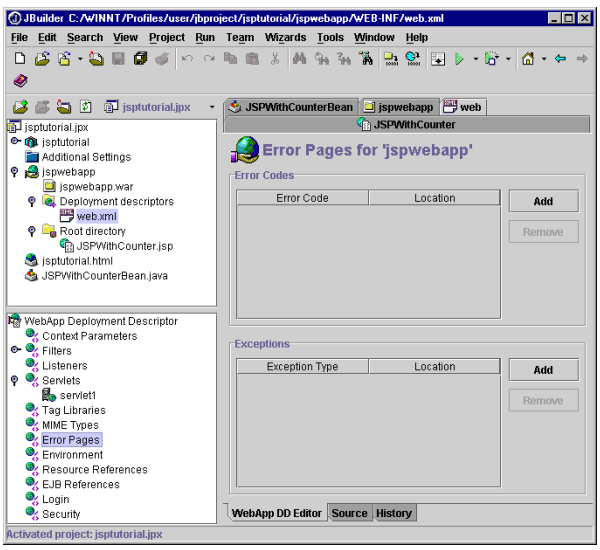
Figure 16.9 MIME Types page in WebApp DD Editor



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.

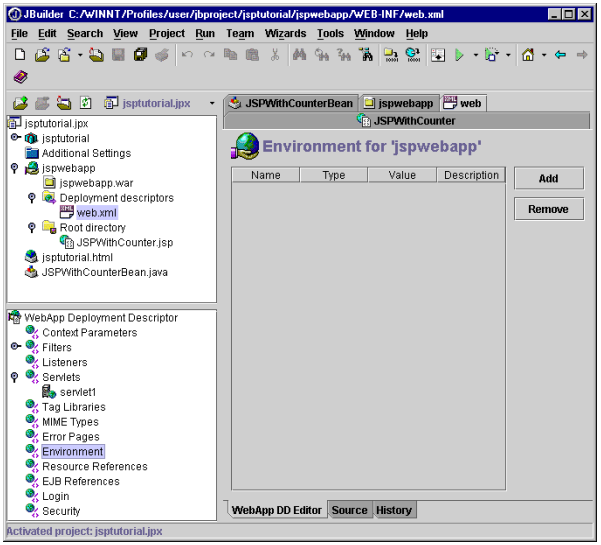
Figure 16.10 Error Pages page in WebApp DD Editor



Environment page

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

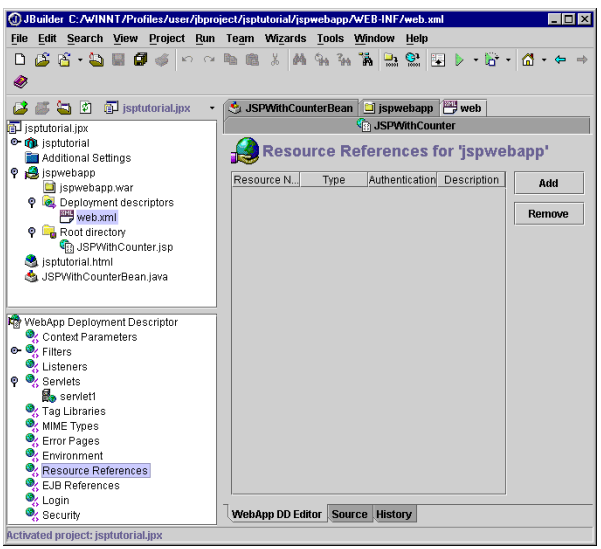
Figure 16.11 Environment page in WebApp DD Editor



Resource References page

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

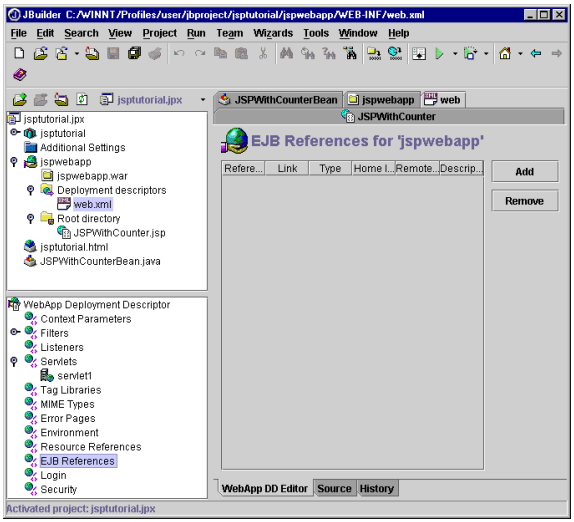
Figure 16.12 Resource References page in WebApp DD Editor



EJB References page

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

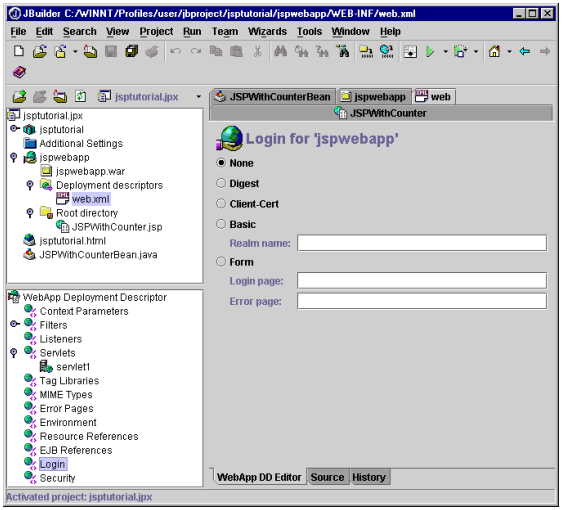
Figure 16.13 EJB References page in WebApp DD Editor



Login page

The Login page displays a set of radiobuttons 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.

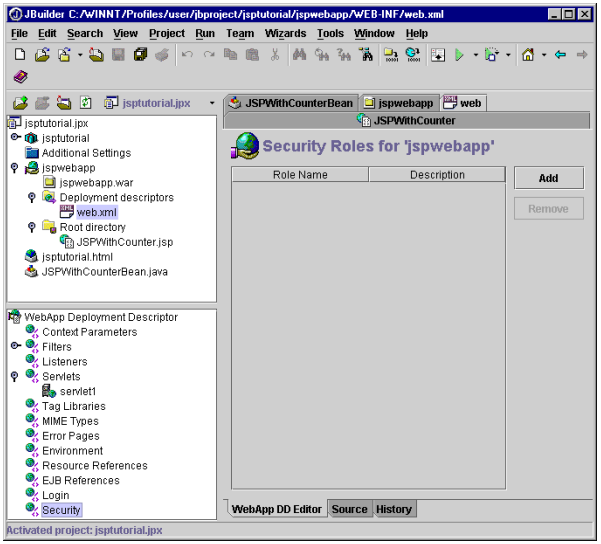
Figure 16.14 Login page in WebApp DD Editor



Security page

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

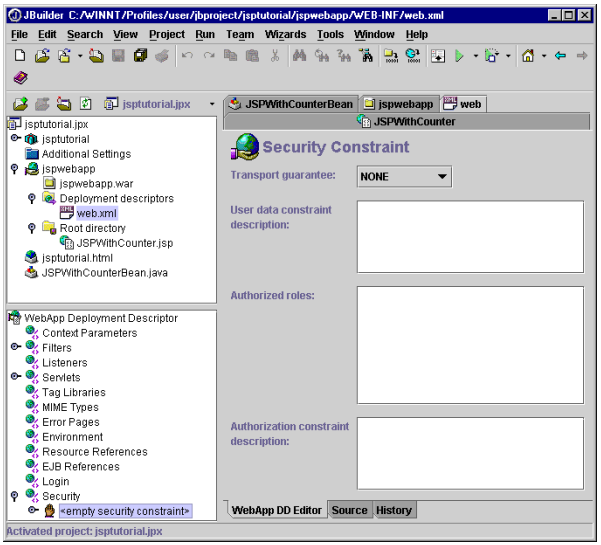
Figure 16.15 Security page in WebApp DD Editor



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.

Figure 16.16 Security constraint in WebApp DD Editor

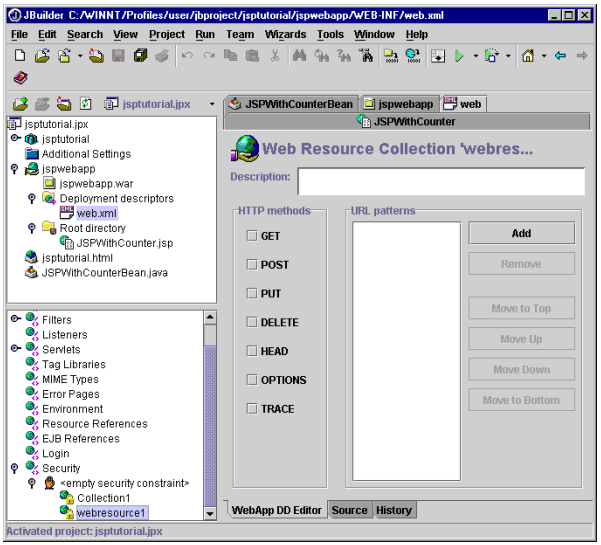


Each web resource collection's `web-resource-name` is listed as a separate child node of one or more applicable security constraints. You cannot add a web resource collection unless you already have at least one security constraint. Each security constraint must have at least one web resource collection. 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:

Item	Description
Description	Description of the web resource collection.
URL patterns	A listbox of the URL patterns for this 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.

Figure 16.17 Web resource collection node in WebApp DD Editor



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 plugin is properly configured. For more information on configuring your web server plugin, please see “Setting up JBuilder for web servers other than Tomcat” on page 14-4.

Vendors who offer plugins for web servers other than Tomcat are encouraged to provide GUI editors for their vendor-specific deployment descriptors. If your vendor doesn’t provide a GUI editor for their deployment descriptor, you can still edit the deployment descriptors source code in JBuilder’s XML source editor. To do this, you open the vendor-specific deployment descriptor and click on 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 plugin.

For information on creating a custom plugin for a web server, see “Creating your own web server plugin” on page 14-9.

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/aboutJava/communityprocess/first/jsr053/index.html.

Chapter

17

Launching your web application with Java Web Start

Web Development is a feature of JBuilder Professional and Enterprise.

Java Web Start is a new application-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 application is not present on your computer, Java Web Start downloads all the necessary files and caches them locally so the application 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

For more information on JBuilder and Web Start, see "Java Web Start and JBuilder" on page 17-3.

Considerations for Java Web Start applications

Generally, developing an application for deployment with 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 17.1 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

Java Web Start is not bundled with JBuilder. For download and installation instructions, go to the Java Web Start page at http://java.sun.com/products/javawebstart/ and click the "Download Now" icon. Note that the installation will depend on your computer's operating system. Once you've installed Java Web Start, 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 stand-alone application into a Web Start application. To do this, you'll follow these general steps:

1. Create a WebApp with the Web Application wizard.

   For more information on WebApps, see Chapter 3, "Working with WebApps and WAR files."

2. Create the 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 17.2 Archive Builder options

Archive Builder	Option
Step 1	Archive type - Web Start Applet or Web Start Application
Step 2	Name - JAR file name
	File - 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 the online Help book *Building Applications with JBuilder.*

3. Build the project to create the JAR file.

4 Create the application's JNLP file and homepage, using the following options on the Web Start Launcher wizard:

Table 17.3 Web Start Launcher options

Web Start Launcher	Option
Step 1	Name - Name for application. JAR File - Name and path to JAR file. Main Class - Class containing `main()` method. Create Homepage - Leave checked.
Step 2	Title - Application title. Vendor - Company name. Description - Application description. Allow Offline Usage - Check to launch from desktop.

5 Right-click the application's HTML file (created by the Web Start Launcher wizard) and choose Web Run.

6 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).

7 Paste the URL into your external browser.

8 Click the link to your application on the web page.

Each of these steps is outlined in greater detail in the "Tutorial: Running the CheckBoxControl sample application with Java Web Start" on page 17-6.

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.

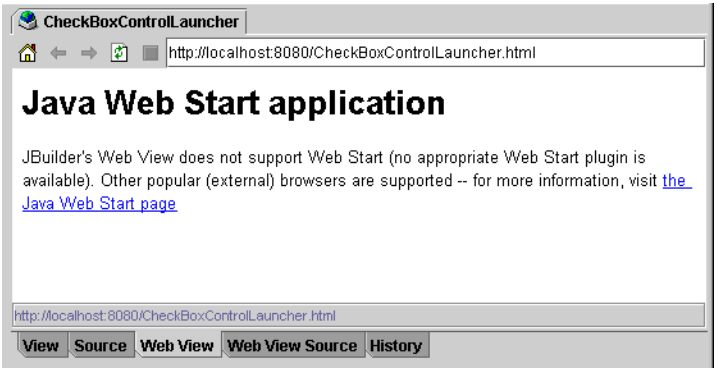
Warning The Web Start Launcher wizard gives the same name to the JNLP and HTML files. If the name entered in the Name field on Step 1 of the wizard

matches the name of an existing HTML or JNLP file in your project, you will be asked if you want to overwrite the existing file.

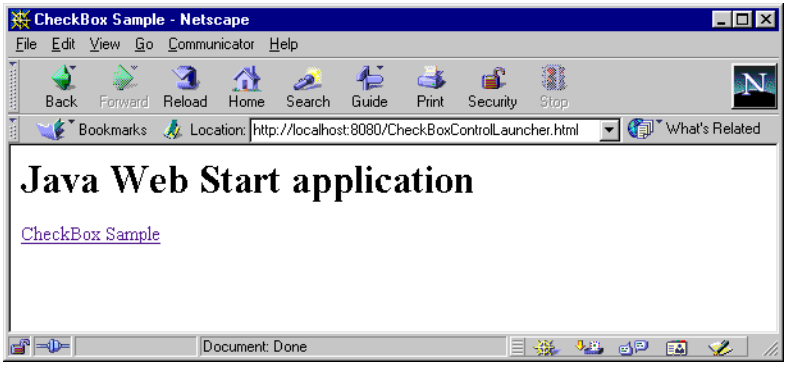
The JNLP file is an XML document. The elements in the file describe application features, such as the application name, vendor, and homepage; as well as JNLP features. For more information, see "JNLP File Syntax" in the Java Web Start *Developer's Guide* at http://java.sun.com/products/javawebstart/docs/developersguide.html.

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.

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 will look like this:



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 will look like this:



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 launcher.

Tutorial: Running the CheckBoxControl sample application with Java Web Start

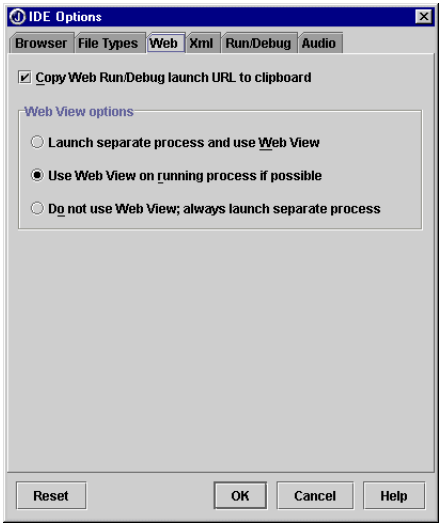
This section walks you through the steps of launching a Swing-based sample application with Web Start. The sample, `CheckBoxControl`, is located in the `samples/Swing` directory of your JBuilder installation.

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.jpr` 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 will look like this:



4. Choose File | Save All to save your work.

To see what this application does, you can run it by choosing Project | Make Project "CheckBoxControl.jpr" to compile it, then Run | Run Project to run it. The application demonstrates how to use Swing's CheckBox control.

Note that this application does not contain any file handling operations. If it did, we would have to either digitally sign the JAR file, or rewrite the file handling operations using classes in the JNLP library. For more information about Java Web Start and security issues, see "Considerations for Java Web Start applications" on page 17-2.

In the next step, you'll use the Web Application wizard to create the application's WebApp.

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.

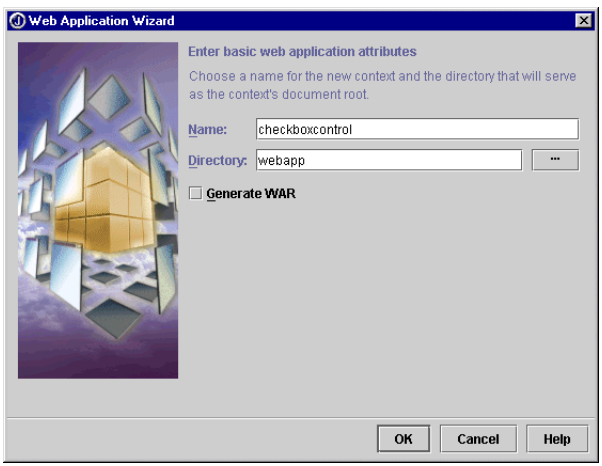
   The Web Application wizard is displayed.

2. Enter `checkboxcontrol` in the Name field.

3. Enter `webapp` in the Directory field.

4. In the Web Application wizard, make sure the Generate WAR option is not selected.

   The Web Application wizard should look similar to this:



5 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.

In the next step, you'll use the Archive Builder to create the application's JAR file.

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 If you haven't yet compiled the project, do so now. Choose Project | Make Project "CheckBoxControl.jpr."

2 Choose Wizards | Archive Builder.

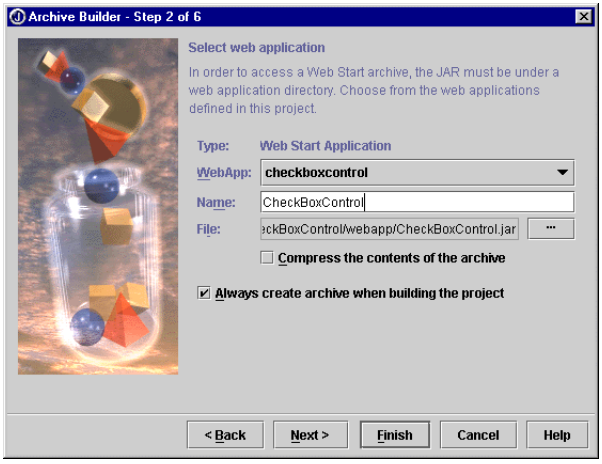
3 On Step 1 of the Archive Builder, change the Archive Type set to Web Start Application. Click Next to go to Step 2.

4 On Step 2, make sure `checkboxcontrol` is selected in the WebApp drop-down list.

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 will look like this:



6 Click Finish to create the archive and close the wizard. You do not have to change any options on the remaining steps of the wizard.

7 Choose File | Save All.

8 Choose Project | MakeProject "CheckBoxControl.jpr" 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.

In the next step, you'll use the Web Start Launcher wizard to create the application's homepage and JNLP file.

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 On the Web page, choose Web Start Launcher and click OK.

The Web Start Launcher wizard is displayed.

3 In the Name field, enter `CheckBoxControlLauncher`.

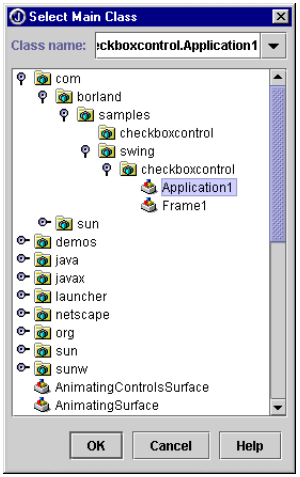
This option names the HTML file and the JNLP file.

4 Choose `checkboxcontrol` from 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 If the Main Class field is not already filled in, click the ellipsis button to the right of the field. This displays the Select Main Class dialog box. Expand the `com` folder at the top of the dialog box to choose

`com.borland.samples.swing.checkboxcontrol.Application1`. The Main Class dialog box will look like this:

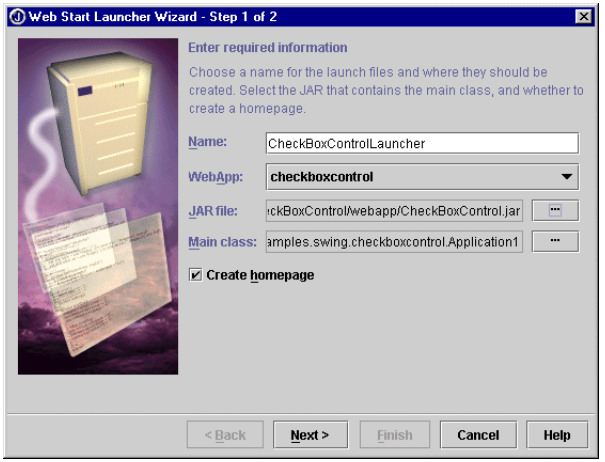


7 Click OK to close the dialog box.

8 On the Web Start Launcher wizard, make sure the Create Homepage option is checked. 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 will be asked if you want to overwrite the existing file.

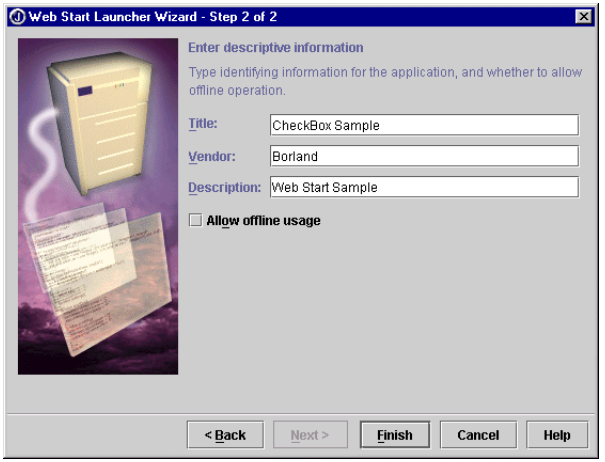
Step 1 of the Web Start Launcher will look like this:



9 Click Next to go to Step 2 of the wizard.

10 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.

Step 2 of the Web Start Launcher wizard will look like this:



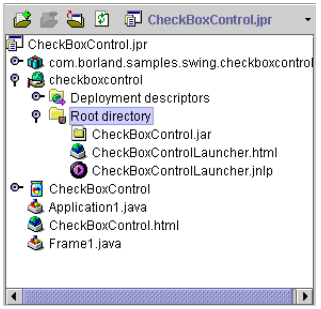
11 Click Finish.

12 Choose File | Save All to save your work.

The wizard creates an HTML file and a JNLP file called `CheckBoxControlLauncher` and places them in the `webapp` folder of your project; that is in the application's WebApp. To see these files in the project pane, expand the `Root Directory` node of the `checkboxcontrol` WebApp node.

Note The Root Directory is referring to the root directory of your WebApp, that is the `webapp` directory.

The project pane should look similar to this:



Note You can open these files in the editor; however, do not change them.

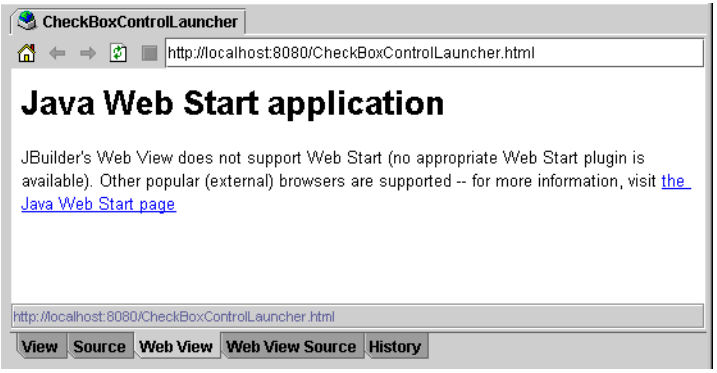
In the next step, you'll start the web server and launch your application with Web Start.

Step 5: Launching the application

This step tells you how to launch your application with Web Start. To do this,

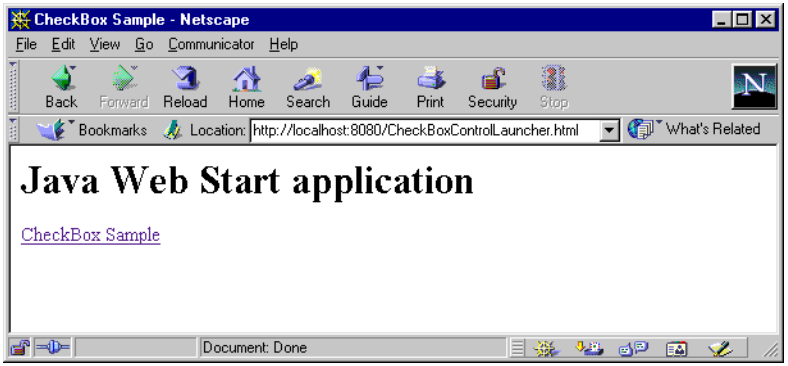
1. Right-click `CheckBoxControlLauncher.html` in the project pane and choose Web Run.

   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. In your external browser, position the cursor in the Location field 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.

3. Your web browser displays the application's homepage, `CheckBoxControlLauncher.html`. The web page contains a link to the application.



4. Click the link on the web page. Java Web Start loads and launches the application. Note that the splash screen displays information you entered into the Web Start Launcher wizard.
5. To exit the sample application, choose File | Exit. To stop the web server, choose the Reset Program button on the Web Server tab.

Congratulations! The application is now running from a link in an external web browser. 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

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

X

Z