Building Web Components

Forte™ for Java™, Internet Edition, 2.0

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303
U.S.A. 650-960-1300

Part No. 806-7518-10
December 2000, Revision A

Contents

3 Programming a Web Application

4 JSP Tag Libraries

Preface

Welcome to the *Building Web Components* book of the Forte™ for Java™ programming series. This book focuses on web application development in the context of the Java 2 Platform Enterprise Edition specification (J2EE™) and its supporting technologies, which include the Java Servlet and JavaServer Pages™ (JSP) technologies.

Specifically, this book describes how to build applications consisting of components that run in a J2EE web container. These applications typically utilize Java servlets, JSP pages, JSP tag libraries, and supporting classes and files. They may access persistent data, for example, a database. They may be independent applications in which all functionality is managed by a web container. Or, they may provide primarily a user interface while depending on components in a J2EE EJB™ container for other services, such as execution of business logic and access to persistent data (this book does not address development of EJB components).

The book assumes a general knowledge of Java programming, JSP page programming, and HTML coding.

Before you read this book: The following list of documents will help you understand the concepts upon which this book is based:

- Java™ 2 Platform, Enterprise Edition Blueprints—`www.java.sun.com/j2ee/blueprints`
- *Java™ 2 Platform Enterprise Edition Specification*—`www.java.sun.com/products`
- *Java™ Servlet Specification, v2.2*—`www.java.sun.com/products/servlet/index.html`
- *JavaServer Pages™ Specification, v1.1*—`www.java.sun.com/products/jsp/index.html`

Organization of This Manual

This manual is designed to be read from beginning to end. Each chapter in the book builds upon concepts and code examples discussed in earlier chapters.

The following table briefly describes the contents of each chapter:

Chapter	Description
Chapter 1, "J2EE Web Application Concepts"	Provides an overview of the core J2EE technologies used in building web-centric Java applications.
Chapter 2, "Design and Programming Issues"	Discusses important design and programming issues relevant to J2EE web applications.
Chapter 3, "Programming a Web Application"	Discusses issues specific to programming a web application using Forte for Java.
Chapter 4, "JSP Tag Libraries"	Describes JSP tag libraries and explains how to create, package, and access them.

Conventions

This table provides information about the conventions used in this document.

Format	Description
italics	Italicized text represents a placeholder. Substitute an appropriate clause or value where you see italicized text. Italicized text is also used to designate a document title, for emphasis, or for a word or phrase being introduced.
`monospace`	Monospace text represents example code, commands that you enter on the command line, directory, file, or path names, error message text, class names, method names (including all elements in the signature), package names, reserved words, and URLs.
`monospace bold`	Monospace bold text represents user input contrasted with computer output.
ALL CAPS	Text in all capitals represents file system types (GIF, TXT, HTML and so forth), environment variables, or acronyms (FFJ, JSP).
Key+Key	Simultaneous keystrokes are joined with a plus sign. For example, Ctrl+A means press both keys simultaneously.
Key-Key	Consecutive keystrokes are joined with a hyphen. For example, Esc-S means press the Esc key, release it, then press the S key.

The Forte for Java, Internet Edition Documentation Set

Forte for Java offers a set of books delivered in Acrobat Reader (PDF) format and online help. This section provides descriptions of these documents.

Documentation Set

You can download the following documents from the Forte for Java web site:

- The Forte for Java programming series:
  - *Introduction to the Programming Series* – Introduces the two books in the Forte for Java, Internet Edition programming series.
  - *Building Web Components* – Describes how to build a web application as a J2EE web module that uses JSP pages, servlets, tag libraries, and supporting classes and files.
  - *Programming Persistence* – Describes support for different persistence programming models provided by Forte for Java: JDBC and Transparent Persistence.
- *Forte for Java, Internet Edition Tutorial*

  Provides step-by-step instructions for building a simple web application using tools introduced in Forte for Java, Internet Edition, which facilitate creating a web module, as described in the *Java™ 2 Platform Enterprise Edition Specification*.

Online Help

Online help is available inside the Forte for Java development environment. You can access it by pressing the help key (Help on Solaris, F1 on Windows and Linux), or by choosing Help>Contents from the Help menu. This action gives you a list of help topics and a search facility.

Javadoc

Javadoc documentation is available within the IDE for many Forte for Java modules. Refer to the Release Notes for instructions for installing Javadoc. When you start the IDE, you can access this Javadoc documentation by clicking on the Javadoc pane of the Explorer.

Chapter 1

J2EE Web Application Concepts

This chapter provides an overview of the core J2EE technologies used in web applications, including:

- Servlets
- JSP Pages
- Web Containers
- Web Modules

Conceptual Background

The J2EE specification defines a broad architecture that encompasses numerous component types and runtime environments for these components. In all, it defines three runtime environments—the web container, EJB container, and application client container. It also classifies its component types into categories that correspond with the containers in which they run. Thus, we have web components, EJB components, and application client components.

This chapter discusses concepts that are fundamental only to the web container and its component types. It also gives consideration to supporting classes and files that are not directly managed by the web container but that are logically part of the web application and that are deployed together with the web components.

For further information on J2EE web technologies, see:

- *Java™ 2 Platform, Enterprise Edition Blueprints*
- *JavaServer Pages Specification, version 1.1*
- *Java Servlet API Specification, version 2.2*.

Web Containers

A web container provides runtime services that support the execution of the web components of a J2EE application. These services include

- Life-cycle management, network services (by which requests and responses are sent)
- Decoding of requests and formatting of responses
- Interpreting and processing of JSP pages into servlets
- Access to the J2EE service and communication APIs, which provide for security, concurrency, transaction, and deployment

Web containers forward client requests from a web server to web components in the application and forward the client-bound responses from the web components to the web server. Web containers typically run in a web server process (as a web server plug-in) or in a J2EE application server process.

Web Modules

A web module is the smallest deployable and usable unit of web resources in a J2EE application. It corresponds to a "web application" as defined in the Java Servlet Specification version 2.2.

Web modules can be packaged and deployed as web archive (WAR) files. The format of a WAR file is identical to that of a JAR file. However, because the contents and use of a WAR file differ from that of a JAR file, WAR file names use a `.war` extension.

Structure

A web module may contain:

- Java class files for the servlets and the classes that they depend on, optionally packaged as a JAR file
- JSP pages and their helper Java classes
- JSP tag libraries (normally packaged as a JAR file)
- Static documents (for example, HTML, images, sound files, and so on)
- Applets and their class files

A web module must contain:

- A Web deployment descriptor

Web modules use a hierarchical structure for storing their resources. This structure can be represented at development time as a file system. The following diagram illustrates the web module hierarchy.

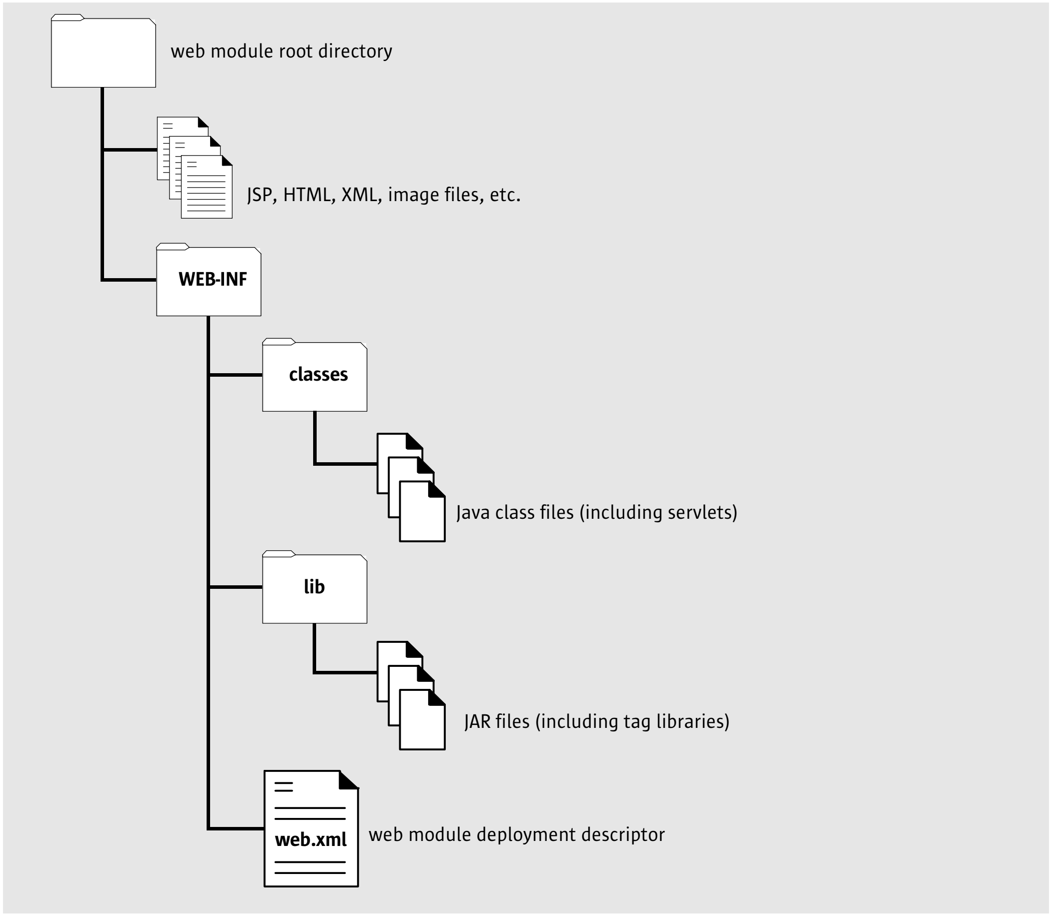

web module root directory
JSP, HTML, XML, image files, etc.
WEB-INF
classes
Java class files (including servlets)
lib
JAR files (including tag libraries)
web.xml
web module deployment descriptor

Figure 1 *Web Module Hierarchical Structure*

Runtime Representation

A web module is represented at runtime by an object implementing the `ServletContext` interface. The `ServletContext` instance provides web components with access to resources available within the web module. For example, it enables web components to log events, obtain URL references to resources, and set and store attributes that other web components in the web module can use.

A `ServletContext` instance is unique within a nondistributed web module and is shared by all web components within the web module. This object is implicitly available in JSP pages as the `application` instance variable. (This variable is always available; it does not need to be declared.)

A `ServletContext` instance, and the web module it represents, is rooted at a specific path within a web server. It could, for example, be rooted at `http://www.myStore.com/productList`. In this case, all requests starting with the `/productList` request path, known as the context path, would be routed to this `ServletContext` instance.

Web Components

Web components are server-side J2EE components. They are managed by and communicate directly with a web container. They are capable of receiving HTTP requests through the web container, processing them, and returning HTTP responses through the web container. The J2EE platform defines two web component types: servlets and JSP pages.

Servlets

Strictly speaking, a servlet is any Java class that implements `javax.servlet.Servlet`. However, most servlets in use today, and the servlets that this book refers to by way of the term servlet, are subclasses of `javax.servlet.http.HttpServlet`.

Servlets execute within a web container and are used to extend the functionality of web servers and web-enabled application servers. The Servlet API enables programmers, within their servlet code, to access HTTP requests and to generate HTTP responses as Java objects and provides many useful methods for manipulating these objects. For example, you can retrieve and set request and response parameters through simple method calls. You can also access HTTP cookies and manage user sessions through Java objects.

Servlets are used typically to provide services such as generating dynamic content in response to a request generated by an HTML form, often accessing a data source to do so. They are also used to control application flow by enabling and disabling access to certain web resources, depending on some state that the servlet tracks. Another common use for servlets is for tracking user sessions, for example, adding and deleting items from a user’s shopping cart.

JSP Pages

A JSP page is a text-based web component that is dynamically translated into a servlet by the web container before execution.

This book uses the following terms:

- *JSP file* – the JSP text-based source file that a developer creates and edits
- *JSP implementation class* – a Java class that the web container creates by translating a JSP file
- *JSP page* – a logical term that includes both of the previous concepts and is used when it is not important or desirable to differentiate between them

From a user's perspective, a JSP page is the flip side of a servlet class—it describes how to process an HTTP request and generate an HTTP response in a presentation- and document-centric way rather than a logic-centric way. Physically, it is somewhat like a servlet turned inside out; whereas a servlet source file is generally programming code with embedded HTML, a JSP file is generally HTML with embedded programming code.

JSP Page Life Cycle

A JSP page is processed by its runtime environment—the web container—and in turn performs processing on an HTTP request and generates an HTTP response. The processes involved in this phase are JSP page translation and instantiation, request processing, and JSP page destruction.

Translation

JSP page translation refers to the process by which the web container converts a JSP file into a servlet class. The details of this process are implementation specific. In the reference implementation the JSP file is converted to a Java servlet source file and then compiled it to a class file.

The web container translates a JSP file the first time it receives a request for it. On subsequent requests for the same JSP page, the web container normally bypasses this phase. However, translation may also occur if the date on the JSP implementation class is older than the date on the JSP file.

Instantiation

When the web container receives a request for a particular JSP page, it first attempts to locate a corresponding JSP instance. If it cannot find one, it instantiates one (as part of this process, it translates the JSP file if the implementation class does not yet exist). It then calls the instance's `_jspInit` method, which corresponds to the `jspInit` method of the JSP file. You can use this method to prepare resources that your JSP pages might require.

Request Processing

The JSP page receives client requests from the web container, processes the request according to its programmed logic, and sends a response to the container. By default, each request executes in its own thread.

Destruction

The web container can reclaim resources by destroying a JSP instance. Before doing so, it calls the instance's `jspDestroy` method, which corresponds to the `jspDestroy` method of the JSP file. You can use this method to close resources that are no longer needed.

Web containers typically provide a way to limit how long a JSP instance can persist without receiving a request. After the user-specified limit, the web container calls the `jspDestroy` method.

Code Constructs in JSP Pages

A JSP page can contain template data and elements. *Elements* are constructs recognized by the web container; they provide dynamic capabilities. *Template data* are unrecognized constructs, such as HTML and XML code; these are passed through to the HTTP response verbatim. Template data is generally used to provide static content and to format dynamic data. Because HTML is passed through verbatim, coding presentation content is very natural for a web page designer.

JSP elements are grouped into three categories: directive elements, action elements, and scripting elements.

Directive Elements

Directive elements provide global declarative information about a JSP page that is unrelated to any particular request. For example, you use a directive to import packages into a page. You also use a directive to associate a page with the current HTTP session. Directives are processed at translation time. They do not write output to the HTTP response object (output written to the HTTP response object appears as text in the generated web page).

Directives are placed between <%@ and %> symbols. For example, the following `page` directive imports the `java.util` package and associates the JSP page with the current HTTP session.

```
<%@ page import="java.util.*" session="true"%>
```

The JSP Specification defines these directives: `page`, `include`, and `taglib`.

Action Elements

Action elements are XML-style tags that provide a means of working with Java objects without writing Java code. For example, you can use actions to locate and instantiate objects, and to get and set an object's properties. Actions are processed at request time. Some actions write output to the HTTP response object.

Because actions use XML syntax, they provide web page designers with a familiar paradigm for accessing dynamic data. (Even though they might not code the actions themselves, web page designers need to understand actions enough to work in a file that contains them; they might have to provide HTML formatting for actions that produce output to a web page.) Actions are also potentially easy for tools to analyze (unlike Java code).

Standard actions are actions defined by the JSP specification and implemented by the web container. The standard actions are: `forward`, `include`, `useBean`, `getProperty`, `setProperty`, `param`, and `plugin`.

The JSP specification also allows for the development of *custom actions* to provide functionality not available through standard actions. You define custom actions in an XML document called a tag library descriptor (TLD) and implement them as JavaBeans™ components. The TLD and implementing beans are conceptually one component— called a tag library.

A tag library is normally packaged as a JAR file and made available to a JSP page through a `taglib` directive in the page. You can develop your own tag libraries or obtain them from a vendor (they could, for example, be provided as part of some vendor's implementation of a web container). For more information on custom actions and tag libraries, see "JSP Tag Libraries" on page 49.

Actions are placed between < and /> symbols. The following example shows the `include` action being used to insert a JSP page named `header.jsp` into the current JSP page.

```
<jsp:include page="header.jsp" flush="true"/>
```

In the example, the prefix (`jsp`) before the colon indicates that this is a standard action. The string after the colon, in this case `include`, is the name of the action. The name-value pairs (`page="header.jsp"` and `flush="true"`) are attributes of the action.

Some actions can contain a body, that is, they have a beginning and ending tag that can enclose another action, scripting elements, or template data. For example, in the following code, the `useBean` action attempts to locate an object available by the reference `cBean` in the `application` scope and make it available locally through a scripting variable also named `cBean`. (For more information about scopes, see "Scopes and Implicit Objects" on page 18.) If the object cannot be located, the action instantiates it, using the specified `Expns.CBean` class, and makes it locally available. The two method calls contained in the body of the action (`getConnected` and `getEngine`) are invoked only if the action instantiates the `Expns.CBean` class. If the action locates an already existing instance, the two methods are not invoked.

```
<jsp:useBean id="cBean" scope="application" class="Expns.CBean">
<%
  cBean.getConnected();
  cBean.getEngine();
%>
</jsp:useBean>
```

Scripting Elements

Scripting Elements allow you to embed Java code within a JSP file. You can use these elements for programming logic and also for writing output to the HTTP response object. There are three syntactically distinct types of scripting elements—declarations, scriptlets, and expressions.

Declarations allow you to declare and initialize variables, instantiate objects, and declare methods. Declarations are processed at translation time and do not write output to the HTTP response object. Declarations are placed between `<%!` and `%>` symbols. The following example declares and initializes two `String` variables:

```
<%!
  String name = null;
  String title = null;
%>
```

Scriptlets allow you to enter any piece of valid Java code. Variables and methods declared in a declaration element are available to scriptlets in the same JSP page. A Java statement can begin in one scriptlet and end in another (interspersed, for example, with HTML code). Scriptlets are processed at request time and write output to the HTTP response object if you code them to do so. Scriptlets are placed between <% and %> symbols.

The following scriptlet example shows a Java `if` statement that spans two scriptlets and is used to conditionalize a fragment of HTML code that lies between them. The HTML code will be included in the HTTP response only if the `if` statement evaluates to `true`.

```
<% if (name.equals("Elvis Presley")){
%>
<p>Let’s hear it for Elvis!
<% title = "King";
}
%>
```

Expression elements allow you to enter any valid and complete Java expression. The web container converts an expression element to a String at request time. The resulting String is then written to the HTTP response object. Expressions are placed between <%= and %> symbols.

The following example inserts a piece of dynamic data into an HTML string.

```
<p>Hail the <%= title %>!
```

Scopes and Implicit Objects

When you instantiate an object in a JSP page you will want to make it available to other objects in your application. You may want to make it available to all objects in your application, or you may want to restrict its availability to some subset of these objects. For example, you may want to make it available only to objects associated with the current user’s HTTP session.

To enable you to control the availability of an object, the JSP specification defines a number of scopes in which you can place a reference to the object. These are the page, request, session, and application scopes. At runtime, these scopes are implemented as Java objects, as described in the following table.

Table 1 *Scopes in JSP pages*

Scope	Object Type	Description
page	javax.servlet.jsp.PageContext	Represents the current JSP page. This object is available only to JSP elements in the current page or in pages included by an `include` directive (but not pages included by an `include` action; this is because the directive is executed at page translation time, and the included pages are concatenated into the same JSP implementation class).
request	javax.servlet.ServletRequest	Represents the current HTTP request. This object is available only to JSP pages and servlets executing in the current HTTP request. For example, if one JSP page forwards to another (using a `forward` action), both pages have access to the same `ServletRequest` object.
session	javax.servlet.http.HttpSession	Represents the current user's HTTP session. This object is available only to JSP pages and servlets executing in requests associated with the current user's HTTP session.
application	javax.servlet.ServletContext	Represents the runtime web module. This object is available to all JSP pages and servlets in the web module.

You can locate or make an object available within one of these scopes with a `useBean` action. In this action you supply a `scope` attribute in order to specify the availability of the bean instance, for example:

```
<jsp:useBean id="myCart" scope="session" class="Cart">
```

Scopes (and the objects they represent) are also implicitly available to the scripting elements of a page through scripting variables that the page automatically instantiates for you. These scripting variables use the same names as the scopes they represent—`page, request`, `session`, and `application`.

For example, the following scriptlet uses the implicit `request` variable to populate the `Cart` bean we instantiated in the previous `useBean` action. It then uses the `session` variable to place the `Cart` bean in the session scope, where it will be available to other scripting elements on the page, or other pages in the same user session. Notice that we did not instantiate the `session` and `request` variables:

```
<%
 CartLineItem lineItem = new CartLineItem();
 lineItem.setID(request.getParameter("cdId"));
 lineItem.setCDTitle(request.getParameter("cdTitle"));
 lineItem.setPrice(request.getParameter("cdPrice"));
 myCart.lineItems.addElement(lineItem);
 session.putValue("myLineItems", myCart.getLineItems());
%>
```

Note By default, JSP pages have access to the session scope. However, if a page's `page` directive specifies a `session` attribute whose value is set to `false`, the page is not associated with the current HTTP session and therefore cannot access the session scope and may not reference the `session` implicit variable.

For example, the previous `useBean` action and scriptlet code samples would be illegal on a page containing the following `page` directive.

```
<%@ page session="false" %>
```

Supporting Classes, Beans, and other Files

Web components generally require additional classes, beans, HTML files, and other files to provide supporting functionality. For example, a servlet could delegate complex tasks such as screen flow management or session control to a supporting bean. A servlet could also use a bean for accessing a remote resource, such as an EJB or database, and for caching results returned by calls to such resources. Also, the JSP pages and HTML files will often reference image files and perhaps sound and video files.

Chapter 2

Design and Programming Issues

This chapter discusses web application design and programming issues relevant to J2EE applications in general and that do not pertain specifically to programming in Forte for Java.

This chapter discusses these topics:

- Functional roles of web components in J2EE Applications
- Choosing between using servlets or JSP pages
- Architecture design for accessing a data source
- Techniques for reusing code in JSP pages
- Techniques for accessing Java Objects in JSP pages
- Using a JSP Page as a layout template

Choosing Between Servlets and JSP Pages

JSP pages are servlets at runtime and possess all the capabilities inherent to this interface, foremost among these being the ability to access an HTTP request, process it, and create an HTTP response. The question arises, then, whether you should program your application using servlets or JSP pages.

Even though servlets and JSP pages possess the same basic capabilities, they are suited to playing different roles in a J2EE application, mainly because of the format of their sources. For example, a servlet is well suited for performing logic because it is a standard Java class, and you can use the standard Java programming paradigm and tools to create, edit, compile, and debug it.

Servlets are not well suited for generating web pages, however, for two reasons:

- Web page designers are typically not programmers and are not accustomed to working in Java source files.
- Coding HTML in a servlet is cumbersome because it requires the embedding of HTML code within numerous `println` statements; the `println` statements require escape sequences for many of the HTML characters.

JSP syntax makes the coding of HTML natural, because HTML is coded in a JSP file exactly the way it is coded in an HTML file (the same is true for XML). For this reason JSP pages are ideal for generating presentation content.

The guiding principle in choosing between servlets and JSP pages, therefore, is to use JSP pages for generating HTML and to use servlets for programming the logic that determines the behavior of an application. You can, however, also modularize logic into classes, beans, or tag libraries and access it from servlets and JSP pages.

This document may use either the term JSP page or servlet, depending on which of these web components is more appropriate for a particular function, but it does so with the understanding that JSP pages and servlets are potentially interchangeable.

Web Components in J2EE Applications

A web component can perform any of the following functional roles in an application:

- Front component – In a model-view-controller (MVC) type of application, this component is at the front of the controller structure; it receives the HTTP request from the web server. It can process the request or simply redirect it to another component for processing. The MVC architecture typically uses only one front component that receives all HTTP requests. Other architectures may use multiple front components, for example one front component for each presentation component. A front component can be implemented as either a servlet or JSP page.

- Logic component – This type of component performs some type of logic processing, for example, application flow management, fetching data, or processing business logic. Such components are most often implemented as a servlet, some other class, a bean, or a tag library, although JSP pages can also be used for this purpose.
- Presentation component – This type of component generates part or all of the HTTP response sent to the client. JSP pages are usually the best choice for this function, although servlets can also be used.

Figure 2 shows the relationship of these functional roles.

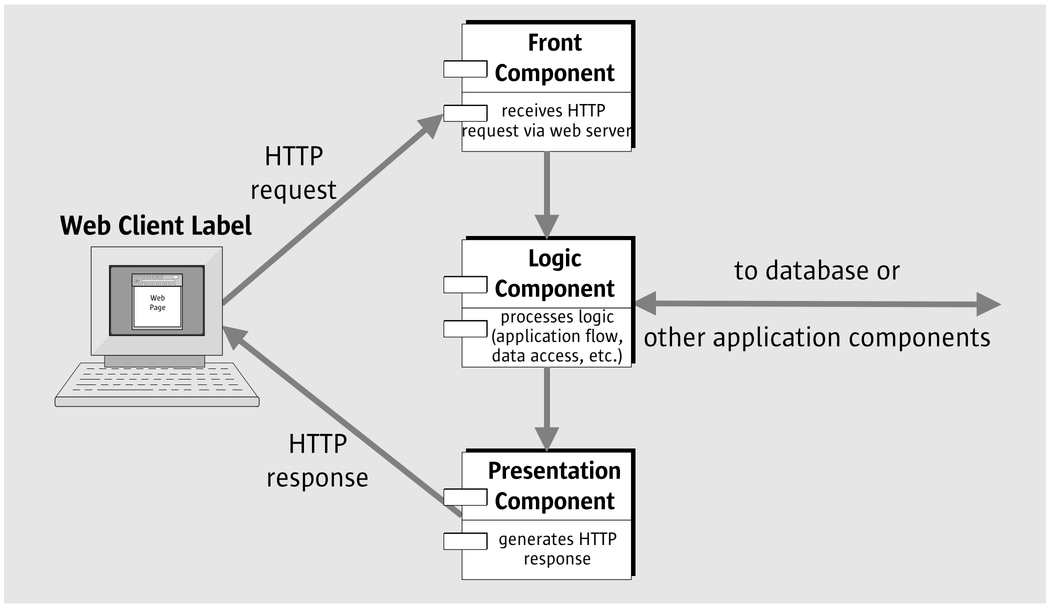


Figure 2 *Potential Uses for Web Components*

It is possible to implement all these roles using a single web component, and in a simple application, this might be the best way to do it. However, complex applications will benefit from separating these roles into individual web components. For example, application flow, data source access, and presentation can be independently edited and optimized if they are modularized. Modularization also makes it easier to divide development tasks between the members of a team, allowing them to specialize in their areas of strength.

Using single-role JSP pages also allows for code reuse and hence centralization of functionality. For example, by separating application-flow logic from presentation, you can centralize the code that controls application flow into a single component. This centralization makes it easier to comprehend and change the application flow when needed.

Designing Data Source Access

J2EE technology provides a flexible architecture for the dynamic retrieval, update, and presentation of data. These types of actions make logical candidates for code reuse because application developers routinely re-implement the code that performs these actions. Instead of re-implementing such code, developers can encapsulate their code into separate servlets, classes, beans, or tag libraries. Such code can be accessed from a JSP page using one of these means:

- JSP scripting elements
- Standard JSP tags
- Custom JSP tags

Applications that access dynamic data revolve around three major concepts: data source, data object, and data presentation.

- A *data source* is the location from which a web component retrieves data for display, or the location to which it sends data updates. The data source could be a database, an EJB method, a JavaBean method, or a `PersistenceManager` method, in the case of Transparent Persistence. The data source returns a data object For more information on Transparent Persistence, see *Programming Persistence*.
- A *data object* is usually a Java object, but can also be an XML document or some other format of a database result set. Note that if the data object is a Java object, it could be typed as a Java class or a Java interface. It could also represent a single business object (such as a customer), a Java collection (such as a list of orders), or a complex Java object graph (such as a customer with a list of outstanding orders).
- A *data presentation* is the specific format used for presenting the data object to the user. Some example data presentation styles are “form,” “table,” “bulleted list of links,” or “master/detail.”

The diagrams in Figure 3, Figure 4, and Figure 5 illustrate these concepts. Figure 3 shows the simplest implementation, in which a single web component both accesses the data source and generates the presentation for the returned data object.

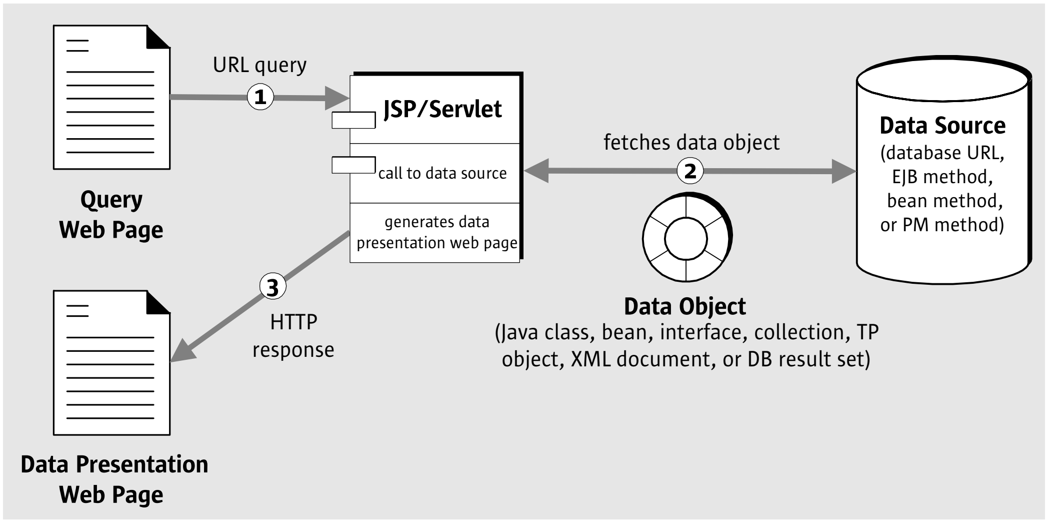

URL query
1
JSP/Servlet
call to data source
generates data presentation web page
fetches data object
2
Data Source
(database URL, EJB method, bean method, or PM method)
Query Web Page
3
HTTP response
Data Object
(Java class, bean, interface, collection, TP object, XML document, or DB result set)
Data Presentation Web Page

Figure 3 *Query and Presentation Using Single Web Component*

Figure 4 shows an implementation that allows for more reuse. It uses a servlet to access the data source and a JSP page to generate the presentation of the data object.

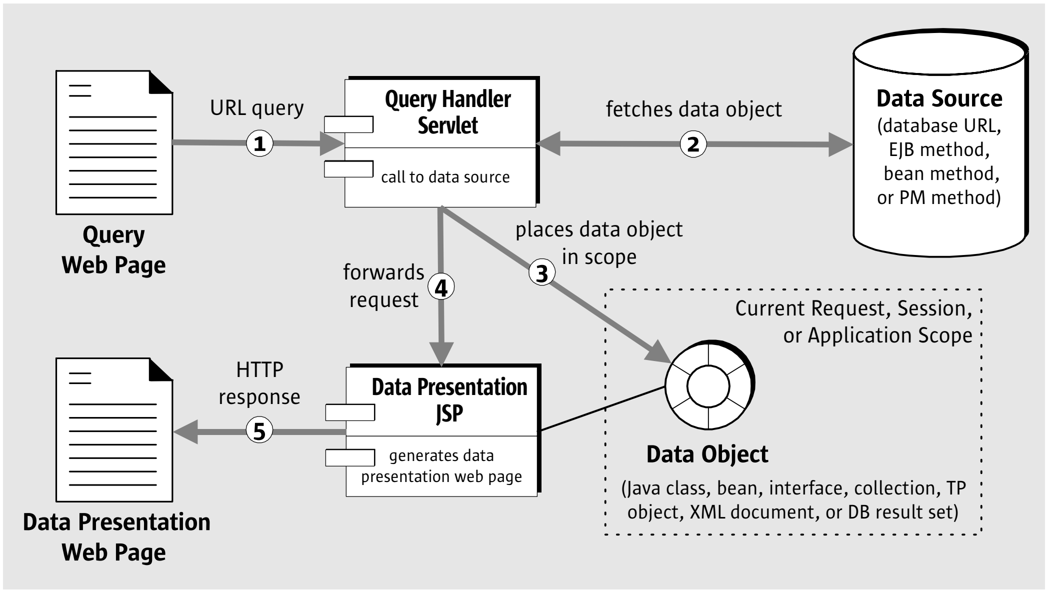

URL query
1
Query Handler Servlet
call to data source
fetches data object
2
Data Source
(database URL, EJB method, bean method, or PM method)
Query Web Page
places data object in scope
3
forwards request
4
Current Request, Session, or Application Scope
HTTP response
5
Data Presentation JSP
generates data presentation web page
Data Object
(Java class, bean, interface, collection, TP object, XML document, or DB result set)
Data Presentation Web Page

Figure 4 *Query and Presentation Using Two Web Components*

Figure 5 shows an implementation of an update action, in which a servlet updates the data source and a JSP page generates a presentation that displays an update status message. This example could have many possible variations, for example, the update status message could be an HTML file. In another variation, the update handler servlet could also query the data source and retrieve the updated data to display to the user.

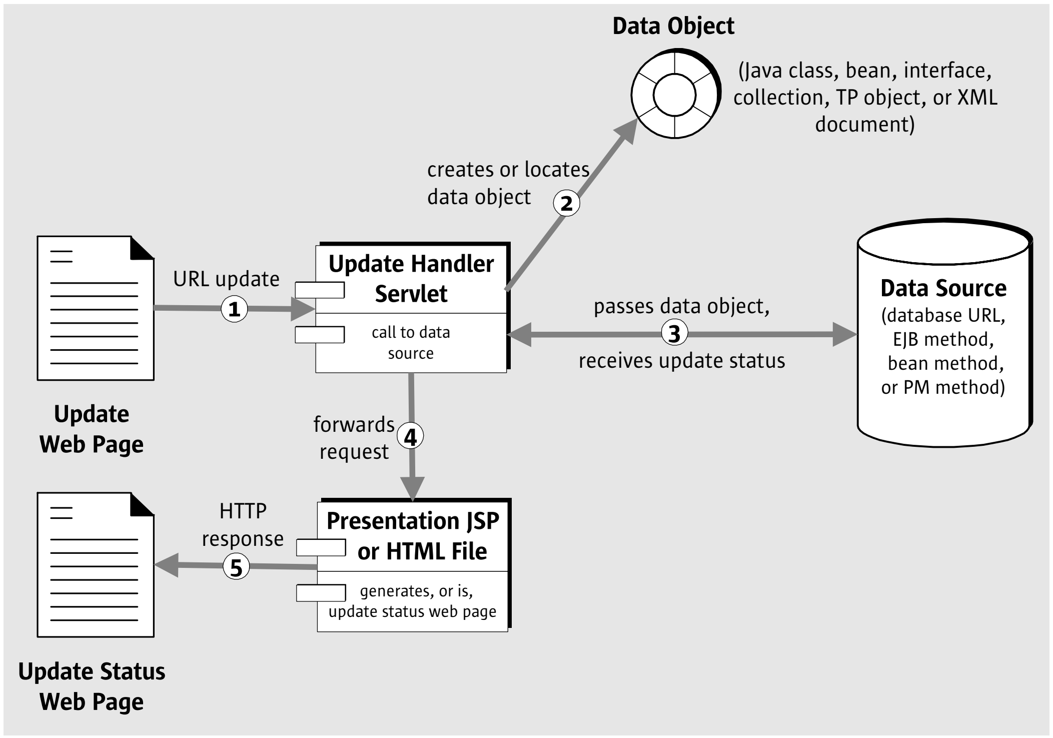

Data Object
(Java class, bean, interface, collection, TP object, or XML document)
creates or locates data object
2
Update Handler Servlet
call to data source
URL update
1
passes data object,
3
receives update status
Data Source
(database URL, EJB method, bean method, or PM method)
Update Web Page
forwards request
4
HTTP response
5
Presentation JSP or HTML File
generates, or is, update status web page
Update Status Web Page

Figure 5 *Update and Presentation Using Two Web Components*

The concepts of data source, data object, and data presentation are interrelated, but their relationships are not rigidly defined. For example, a given type of data object can be displayed using a number of different data presentations, as shown in Table 2.

Table 2 *Presentation Possibilities for a Data Object*

Data Object	Data Presentation
single customer	detailed form layout summary form layout
collection of customer objects	tabular layout form layout (with navigation buttons) list of links

Similarly, a given type of data object can be retrieved from any number of data sources. You could, for example, retrieve a customer object from a database, an EJB method, a PersistenceManager, or a method on a JavaBeans component. Furthermore, the data source does not determine the data presentation. For example, you can format a collection as a table regardless of whether the collection was retrieved from a database, a method on a JavaBeans component, an EJB method, or a TP PersistenceManager.

Programming JSP Pages

This section discusses JSP programming topics.

Code Reuse Through Forwards and Includes

JSP technology provides for reuse of JSP pages through `include` and `forward` actions and `include` directives.

An `include` directive inserts a specified file, verbatim, into the file containing the directive.

For example:

```
<%@ include file="myFile.jsp" %>
```

In this example, the inclusion is performed at translation time and the code is inserted into the compiled implementation class. Because the inclusion occurs at translation time, you can not make it dependent on a reference that is resolved at request time. In other words, the value of the `file` attribute (which in the example is `myFile.jsp`) is interpreted literally. This restriction means that you can not use an expression for the value of this attribute because expressions are evaluated at request time.

The `include` action is similar to the `include` directive, except that it performs its inclusion at request time. You use the action's `page` attribute to specify the file that you want to include. This attribute can accept an expression as its value, which means that you can determine at runtime which file to include. For example, you could obtain the value of a request parameter and use it to compute the name of the file to include.

The following example shows an `include` action that uses an expression in this way.

```
<jsp:include page="<%= myFile %>" flush="true"/>
```

Note Only some attributes accept an expression as a value (consult the JSP Specification for a complete list). The value for such an attribute must be either an expression or a literal ASCII string. You cannot mix the two. For example, the following `include` action is invalid:

```
<jsp:include page="<%= myFile %>.html" flush="true"/> INVALID
```

In JSP 1.1, the `include` action requires a `flush` attribute set to `true`.

The `forward` action is similar to the `include` action, except that whereas the `include` action forwards execution (along with the request) to another page and then returns to the original page, the `forward` action terminates execution of the current JSP page and forwards execution (along with the request) to another page. The `forward` action uses the same `page` attribute as the `include` action but does not require the `flush` attribute, as shown in the following example:

```
<jsp:forward page="<%= myFile %>" />
```

Accessing Java Objects

JSP technology provides two means for working with Java objects: actions and scripting elements. These two types of JSP elements represent two different programming paradigms.

Actions enable you to access and manipulate Java objects without using Java code directly in a JSP page. For example, the standard action, `useBean`, enables you to locate an existing bean object in a specified scope (or instantiate one if it doesn't already exist) and make it available to other actions or scripting elements on the page. For example, the following code sample locates an object of the class `com.xyz.Cart` that is available by the name `myCart` on the current `session` object; if the object does not exist, it creates one and adds it to the `session` object. It then makes this object locally available by the name `myCart`.

```
<jsp:useBean id="myCart" class="com.xyz.Cart" scope="session" />
```

Scripting elements allow you to write Java code directly in a JSP page. With scripting elements, you have unlimited access to the Java API and any custom classes you need.

The following sample code performs the same work as the previous action example.

```
<%
  Cart myCart = (Cart)session.getValue("myCart");
    if (myCart==null) {
      myCart = new Cart();
      session.setValue("myCart", myCart);
    }
%>
```

Even though actions and scripting elements essentially represent different programming paradigms, you can mix them in a single JSP page. For example, in the previous two code samples, we instantiated a `myCart` object. If we wanted to access a piece of data from this object and display it on a web page, we could do so using a `getProperty` action or an expression (one of the three types of scripting elements), regardless of whether we used an action or scriptlet to instantiate the object.

The following code sample uses a `getProperty` action to get the `myCart` object's user name and write it to the HTTP response, along with some HTML formatting:

```
<tr>
  <td>User Name:</td>
  <td><jsp:getProperty name="myCart" property="userName" /></td>
</tr>
```

You could use an expression, as in the following code sample, to achieve the same result as the previous `getProperty` action:

```
<tr>
  <td>User Name:</td>
  <td><%= myCart.getUserName() %></td>
</tr>
```

Using JSP Pages as Layout Templates

JSP `include` actions and `include` directives provide a powerful yet simple means for controlling the look and feel of an application. These elements can enable a single JSP page to serve as a template that defines the layout for multiple screens. (A screen is the sum total HTML sent to a client for any single request.)

Figure 6 shows a common layout used for web pages. It has four sections: a header, side bar, body, and footer.

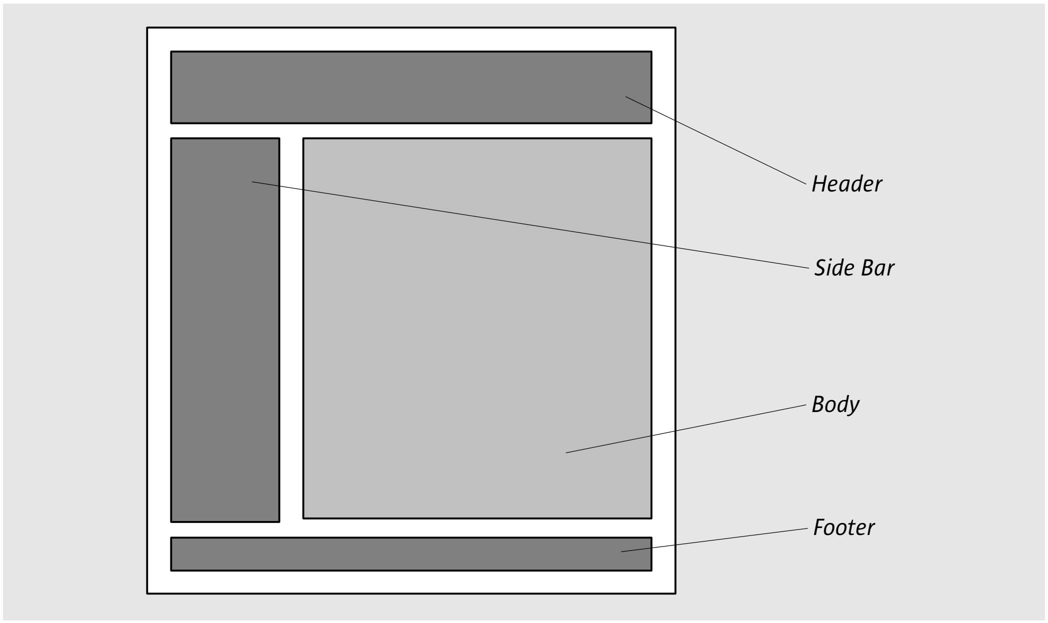


Figure 6 *Elements of a Typical Web Page Layout*

A template of this type defines the layout of a web page but typically does not provide content. The content is provided by additional JSP pages and/or HTML files through the use of `include` directives or `include` actions in the template. Content can be determined dynamically at request time by using an `include` action with an expression as the value of its page attribute.

For example:

```
<jsp:include page="<%= currentScreen %>" flush="true" />
```

A template typically declares both the start and end of the HTML document as well as the start and end of the table that defines the overall grid layout. It's good practice to design included files to describe a complete HTML element, with both its start tag and end tag (if an end tag is required). By following this principle, you have a better chance of being able to view your JSP files in a web browser or manipulate them with a tool, even though they might not be complete HTML documents. For example, many web browsers correctly display a document containing a validly constructed HTML table, even though the start and end tags that declare the document as an HTML document are missing.

Chapter 3

Programming a Web Application

This chapter assumes that you have finished designing your application and are now ready to begin programming.

J2EE web applications may consist of one or more web modules. This chapter begins by providing a high-level view of developing web modules in Forte for Java. This high-level view ties together the disparate tasks you will have to perform in developing your application. It then delves into details on individual programming tasks.

Web Module Development Work Flow

This section gives you an overview of the workflow involved in programming a web module using Forte for Java. The overview is intentionally simplistic and does not attempt to describe iterations of coding and testing. It more importantly lists the major tasks the development team will undertake, and gives a logical order in which these tasks could be performed. Each task references a section later in the chapter that provides more detailed information on the task. Forte for Java on-line help also provides information on most of these tasks.

1 If your web module requires a JDBC™ database driver, copy it to the `lib/ext` directory of your Forte for Java installation directory.

Placing the driver in this directory adds it to the Forte for Java internal classpath, which enables you to test your application with your database.

Note Adding the database driver to your system classpath variable is not an alternative to this step. You must add the driver to the `lib/ext` directory.

2 Create a web module (see "Creating a Web Module" on page 36).

Note Although you can create and test run simple applications without creating a web module, it is highly recommended that create your application within a web module (for more information on this recommendation, see "Creating a Web Module" on page 36).

3 Create the JSP pages required for your web module.

These go in the root (top-level) directory of the web module (see "Creating JSP Pages" on page 38).

4 Create or import the servlets, classes, and beans required for your web module.

These go in the `/WEB-INF/Classes` directory of the web module unless they are packaged as JAR files, in which case they go in the `/WEB-INF/lib` directory (see "Creating Servlets, Classes, and Beans" on page 41).

Note Classes developed using Transparent Persistence are an exception to this rule. You must develop any persistence-capable classes that your web module depends on outside the web module. After you package the persistence-capable classes as a JAR file, place them in your web module's `/WEB-INF/lib` directory. You can then test your application or package it as a WAR file.

For information on Transparent Persistence, see *Programming Persistence* in the Forte for Java programming series. For information on packaging web modules, see "Packaging and Deploying a Web Module" on page 48.

5. Create and/or add any tag libraries your JSP pages depend on.

   You normally develop a tag library within a separate web module that is specifically for that purpose. You then package it as a JAR file and place it in the `lib` directory of the web module containing the dependent JSP pages (see "JSP Tag Libraries" on page 49).

6. Test run your application in the development environment, optionally monitoring it with the HTTP transaction monitor (for information on test running an application, see "Test Running an Application" on page 42; for information on using the HTTP transaction monitor, refer to on-line help).

7. Debug your application (for information on using the debugger, refer to on-line help).

8. Configure the web module (see "Configuring the Web Module Deployment Descriptor" on page 44).

9. Package the web module as a WAR file and deploy it (see "Packaging and Deploying a Web Module" on page 48)

Creating a Web Module

A web module is a J2EE *deployment* construct and is not directly related to how you *develop* your application—there is no J2EE requirement that you develop a web application using a web module directory structure. However, you should develop your web application as a web module for these reasons:

- Your application's files must eventually be part of a web module in order to package them as a WAR file. If you do not develop them in that structure, you will have to manually place them in a web module before you can package and then deploy your application.
- If you create a web component outside of a web module, Forte for Java invisibly creates a web module for it and deploys it to its internal web container. However, this feature is currently suitable for testing only simple applications. Developing your application as a web module ensures that you can test run it in the development environment.

Web Modules in Forte for Java

In Forte for Java, a web module is represented in the Explorer as a mounted file system that conforms to the structure of a WAR file (for more information on this structure, see "Structure" on page 11).

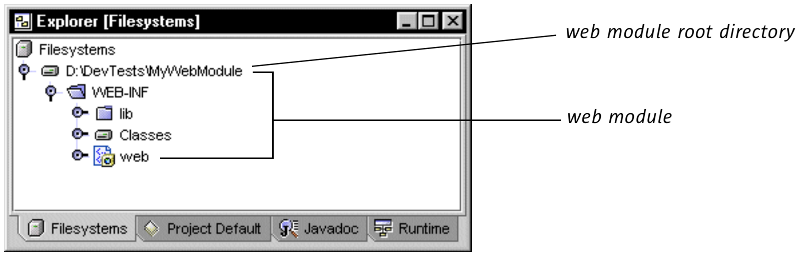


Figure 7 *Web Module mounted in the Explorer*

You mount a web module in the Explorer exactly as you would mount any other file system (see on-line help for information on mounting file systems). But, you must mount the web module *itself*. If you mount a directory that *contains* a web module, rather than the web module itself (in other words if you mount a web module in such a way that it is a subdirectory of a mounted file system), Forte for Java will not properly recognize the web module. This means you will not be able to perform some operations normally associated with a web module.

Although a web module is not an object type in a programming sense, it is treated as an object type in the Explorer. This means, for example, that it has properties that you can set in its Properties window and a set of commands available in its pop-up menu that pertain to the web module. It also means, that like any other object type in the Explorer, you create it from a template.

You can create a web module in one of two ways:

- You can create a web module as a new directory.
- You can convert an already existing directory into a web module.

Note If you have an existing directory structure that conforms to that of a web module, you can mount and use it in the Explorer as a web module without converting it. Forte for Java will recognize such a directory as a web module by its structure.

➤ **To create a new directory as a web module:**

1. From the main window, choose File>New.

   The Template Chooser wizard opens.
2. Open the JSP & Servlet template folder, select the Web Module template, and then click Next.

   The Web Module dialog box opens.
3. Click the ellipsis button (marked with ...).

   A file chooser opens.
4. Navigate to the location in which you want to create the new directory, and then click the new folder icon.
5. Locate the new folder (it is entitled New Folder, but it is not selected; you may have to scroll to find it).
6. Slowly double-click the new folder's title to make it editable. Then type a name for the folder and press Return.
7. Verify that the File Name field indicates the folder's new name (you might have to select a different folder and then reselect the new folder).
8. Click Add.

   Focus returns to the Web Module dialog box.
9. Verify that the Directory field indicates the correct directory, and then click Finish.

   The web module is created and mounted in the Explorer.

➤ **To convert an existing directory into a web module:**

1. In the Explorer, choose Mount Directory from the pop-up menu of the Filesystems icon.

   A file browser opens.
2. Navigate to the directory you want to convert into a web module and click Mount.

   The directory is mounted in the Explorer as a file system.
3. From the directory's pop-up menu, choose Tools>Convert Filesystem into Web Module.
4. Click OK in the confirmation dialog box that opens.

   The directory is converted to a web module.

Creating JSP Pages

You can create a JSP page in one of two ways:

- You can create a JSP page using the template chooser.
- You can create a JSP page by generating it from a Dreamweaver template. For information on this topic, see "To generate a JSP page from a Dreamweaver template:" on page 40.

Note For most situations, it is recommended that you create your JSP pages within a web module. You should create them in the root directory of your web module or in some subdirectory that you have created in the root directory. Do not place them within the `WEB-INF` directory. For more information on this topic, see "Creating a Web Module" on page 36.

➤ **To create a JSP page using the Template Chooser:**

1. In the Explorer, find the directory in which you want to create the JSP page.
2. From the directory's pop-up menu, select New>JSP & Servlet and then choose one of the JSP templates: either JSP (HTML), JSP (Plain Text), or JSP (XML).
3. In the Name field of the wizard that opens, type a name for your JSP page and click Finish.

   The JSP page is created and opens in the Source Editor.

Working With Dreamweaver Templates

Forte for Java enables you to work with Macromedia™ Dreamweaver™ templates in these ways:

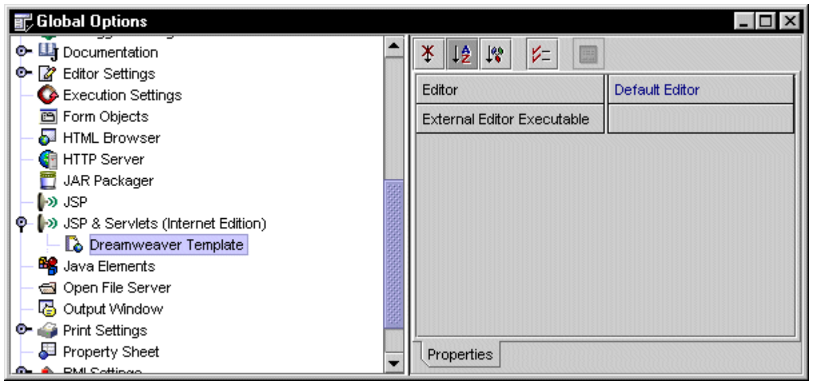
- You can open and edit Dreamweaver templates in the Source Editor.
- You can configure Forte for Java to open Dreamweaver templates in an editor of your choice.
- You can generate a JSP page from a Dreamweaver template.

Forte for Java recognizes Dreamweaver templates as an individual file type (Dreamweaver templates use a `.dwt` extension). This support enables you to open and edit Dreamweaver templates in the Source Editor. By default, a Dreamweaver template will open in the Source Editor when you double-click it in the explorer.

You can reconfigure Forte for Java to open Dreamweaver templates in an external editor of your choice. The following procedure provides an example of how to do this.

➤ **To configure Forte for Java to open Dreamweaver templates in the Dreamweaver application:**

1. Click Tools>Global Options.
2. In the left pane of the Global Options window, open the node entitled JSP & Servlets (Internet Edition).
3. Click on the Dreamweaver Template icon.



Editor	Default Editor
External Editor Executable	

4. In the right pane, click the Editor field and choose External Editor from the drop-down list.
5. Click the Eternal Editor Executable field and then the customize button marked by the ellipsis (...).

6. In the file browser that opens, navigate to and select your Dreamweaver executable and click Select.
7. Close the Global Options window.

   When you double-click a Dreamweaver template, the Dreamweaver application launches and opens the template.

➤ **To generate a JSP page from a Dreamweaver template:**

1. In the explorer, select the Dreamweaver template.
2. From its pop-up menu, choose Save Template as JSP.
3. In the dialog box that opens, type a name for the JSP page in the File Name field and click OK.

The resulting JSP page has the same contents as the Dreamweaver template.

Creating Servlets, Classes, and Beans

As with other object types, Forte for Java provides templates for creating servlets, classes, and beans.

Note For most situations, it is recommended that you create these object types within a web module (for more information on this topic, see "Creating a Web Module" on page 36). You should create them in the `WEB-INF/classes` directory of your web module. This directory is included in Forte for Java's internal classpath when the web module is mounted in the Explorer.

➤ **To create a servlet:**

1. In the Explorer, locate the `WEB-INF/classes` directory.
2. From this directory's pop-up menu, select New>JSP & Servlet>Servlet.
3. In the Name field of the wizard that opens, type a name for your servlet and click Finish.

   The servlet is created and opens in the Source Editor.

For information on creating other classes and beans, refer to on-line help.

Test Running an Application

Forte for Java enables you to test run your application from within the IDE by transparently deploying it to the internal web server upon execution. This functionality makes iterative testing during the development cycle quick and easy. You can test run applications that are composed of either single or multiple web modules.

Test Running a Single Web Module

➤ **To test run an application composed of a single web module:**

1. Select your web module in the explorer and from its pop-up menu choose Build All.

   This action ensures that you have saved all the files and compiled all the classes and components in your web module.

2. Select the servlet or JSP page that is the starting point of your application, and from its pop-up menu, choose Execute or Execute (restart server) depending on the following conditions:

   a. If the internal web server is not running or you have not made changes to any classes (including servlets), you may choose Execute. For example, if the web server is running but you have edited only JSP pages and HTML files, you may choose Execute. The updated pages will be reloaded.

   b. Otherwise, choose Execute (restart server). If you are unsure, choose this command.

   This action switches your view in the IDE to the Running workspace and launches the internal web browser.

Test Running Multiple Web Modules

You can also test run applications that are composed of multiple web modules. To do this you must perform these tasks:

1. Create a web server configuration file. The internal web server will use this file to determine which web modules to load and to map the web modules to URIs.
2. Add the web modules that you want to run to the web server configuration file.
3. Configure the executor property for the servlets and JSP pages in your web modules.
4. Execute the servlet or JSP page that is the starting point of your application

The remainder of this section explains these tasks in detail.

➤ **To create a web server configuration file:**

1 In the Explorer, mount or create the directory that will contain the web server configuration file.

Although you may use one of your web module directories, it is recommended that you use a different directory because you would not normally want to include this file in any of the WAR files produced when you package your web modules.

2 From the directory's pop-up menu, choose New>JSP & Servlet>ServerConfiguration.

➤ **To add a web module to a web server configuration file:**

1 Mount your web module in the Explorer, if it is not already mounted.

2 Select your web server configuration file, and from its pop-up menu choose Add Web Module.

3 In the Add Web Module dialog box, select one of your web modules from the drop-down list, and then type a URI for this web module into the Mapping field, for example, **`/webModuleA`**.

This mapping is used to access components and files in the web module. For example, if you use the mapping `/webModuleA`, and your web module contains a JSP page named `myJSP.jsp` at the root level, you would use the URI `/webModuleA/myJSP.jsp` to access that JSP page from a JSP page in another module in your web server configuration.

➤ **To configure the executor property for a web module:**

1 Open the properties window for a JSP or servlet in the web module.

2 Click the Execution tab.

3 Click the Executor field and then its customizer button, marked with an ellipsis (...).

The Property Editor for the Executor opens.

4 In the Executor Property Editor, click the Context Execution Mode field and then its customizer button.

The Property Editor for the Context Execution Mode opens.

5 In the Context Execution Mode Property Editor, click the Use Server Configuration radio button.

6 Click the Browse button and choose the web server configuration file that you want to use.

7 Click OK in the Executor Property Editor and Context Execution Mode Property Editor.

➤ **To execute your application:**

1 Select the servlet or JSP page that is the starting point of your application.

2 From its pop-up menu, choose Execute or Execute (restart server) depending on the conditions explained in Step 2 of the procedure "To test run an application composed of a single web module:" on page 42.

Configuring the Web Module Deployment Descriptor

All web modules contain a deployment descriptor in the form of an XML file named `web.xml` located in the web module's `WEB-INF` directory. The deployment descriptor provides configuration information to the web module's deployment environment—the web container. It provides information such as:

- Initialization parameters for the `ServletContext` object (which is the runtime representation of the web module)
- Definitions of servlets and JSP pages and their mapping to URIs
- Mapping of tag libraries to URIs
- MIME type mappings
- Session timeout interval
- A list of welcome files
- Mappings of error codes and exceptions to resources
- Security configuration

In Forte for Java, you can configure the deployment descriptor in two ways:

- You can browse the elements of the deployment descriptor in the Explorer and edit them through their properties windows.
- You can open the deployment descriptor file in the source editor and edit it manually.

Mapping Servlets and JSP Pages

The ability to map servlets and JSP pages to any URI is powerful feature of the deployment descriptor. It enables you to reference these components without having to hard-code their locations in your programming code.

Creating such a mapping entails two tasks:

- creating and configuring a `servlet` element to represent your component in the deployment descriptor
- creating and configuring a `servlet-mapping` element to define the actual mapping

The following tutorial shows you how to use the Explorer to create and map a JSP page to a URI and then access that page using the URI.

➤ **To create and map the JSP page to a URI and access it:**

1. Create a web module.

   For an example of how to do this, see Step 1 of the procedure "To create the tag library:" on page 59.

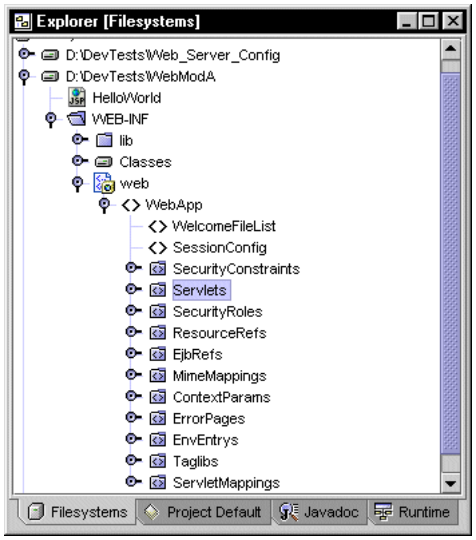
2. In the root directory of the web module, create a JSP page named `HelloWorld`.

   For an example of how to do this, see "To create a JSP page using the Template Chooser:" on page 38.

3. In the Source Editor, edit your JSP page by adding a line containing the following text after the HTML <body> tag.

```
<p>Hello World!
```

4. In the Explorer, select the Servlets folder inside the deployment descriptor as shown:

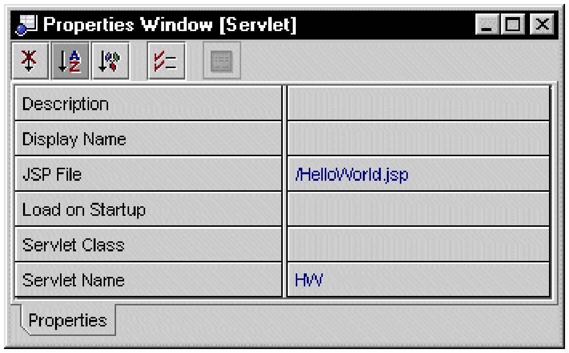


5. From its pop-up menu, select New Servlet.

   This action creates a Servlet element inside the Servlets folder.

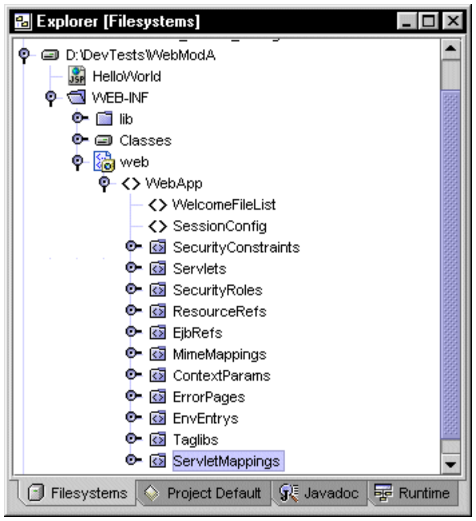
6. Open the Servlets folder, locate the Servlet element, and open its Properties window.

7 Delete the default value entered for the Servlet Class property, and then set the JSP File property to `/HelloWorld.jsp` and the Servlet Name property to `HW` as shown in the following figure.



Description	
Display Name	
JSP File	/HelloWorld.jsp
Load on Startup	
Servlet Class	
Servlet Name	HW

8 In the Explorer, select the ServletMappings folder inside the deployment descriptor as shown:

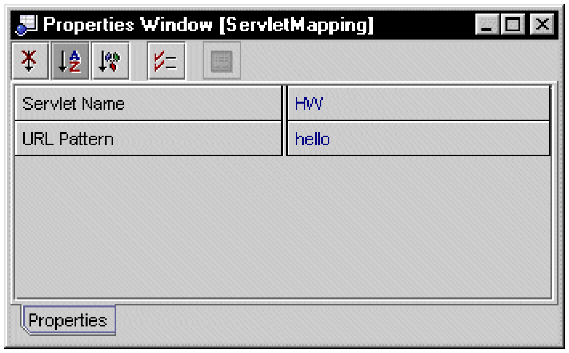


9 From its pop-up menu, select New ServletMapping.

This action creates a ServletMapping element inside the ServletMappings folder.

10 Open the ServletMappings folder, locate the ServletMapping element, and open its Properties window.

11 Set the Servlet Name property to HW and the URL Pattern property to hello, as shown in the following figure.



Servlet Name	HW
URL Pattern	hello

12 In the root directory of the web module, create another JSP page named StartHere.

13 In the Source Editor, edit this JSP page by adding a line containing the following text after the HTML <body> tag.

```
<jsp:include page="hello" flush="true" />
```

14 Select the StartHere JSP page in the Explorer, and from its pop-up menu choose Execute.

A web browser should open and display the text Hello World!.

Packaging and Deploying a Web Module

Forte for Java enables you to package your web modules as WAR files. The WAR file format helps to simplify archiving and deployment of your applications. All J2EE-compliant web containers are capable of running web modules in this format.

Note If your web module is to include persistence-capable classes, you must develop these classes outside the structure of your web module, package them as a JAR file, and place them in the web module's `lib` directory. After performing this task, you can package the web module.

➤ **To package a web module as a WAR file:**

1 In the Explorer, select the root directory of your web module.

2 Choose Tools>Package WAR File.

A file browser opens.

3 Navigate to the directory in which you want to generate the WAR (if necessary, use the file browser's new folder icon to create a directory).

4 In the File Name field, type a name for the WAR file (you are not required to include a `.war` extension to the file name).

5 Click OK.

The WAR file is generated in your chosen directory.

Deployment

Forte for Java does not provide deployment specific tools.

➤ **To deploy your application:**

1 Move the web module(s) that constitute your application to the server on which they will run.

In most cases you will want to package each web module as a WAR file before moving it. Whether or not you should package your web modules depends on the requirements of your server's web container.

2 Install and reconfigure your web modules according to your server's documentation.

Chapter 4

JSP Tag Libraries

The chapter introduces you to JSP tag libraries. It provides a conceptual overview of the structure and workings of tag libraries, explains how to create, package, and use tag libraries in the Forte for Java development environment.

About JSP Tag Libraries

The JSP specification defines a set of code constructs called actions that you can use for coding dynamic behavior into your JSP pages. For example, the `include` action includes a specified JSP page into the current page. J2EE web containers implement the functionality for the standard set of actions defined in the JSP specification. (For more information on JSP actions, see "Code Constructs in JSP Pages" on page 15.)

The JSP specification also defines a mechanism by which you can extend the standard set of actions by creating your own custom actions. By creating custom actions, you can modularize and encapsulate functional units of code within your application and make your code more reusable. With proper design, you can also cleanly separate logic from formatting, thereby eliminating, or at least reducing, the amount of Java code used in your JSP pages.

Custom actions are also commonly referred to as custom tags. However, the term custom action generally refers to the code construct used in a JSP page, whereas the term custom tag generally refers to the code that implements the functionality of a custom action.

A *tag library* is a collection of related custom tags. A tag library consists of a tag library descriptor (TLD), which is an XML document that describes the tags in the library, and the tag handlers that implement the tag library's functionality. A *tag handler* is a bean that implements the functionality for a single tag. The TLD maps each tag to its implementing tag handler. Figure 8 illustrates this architecture.

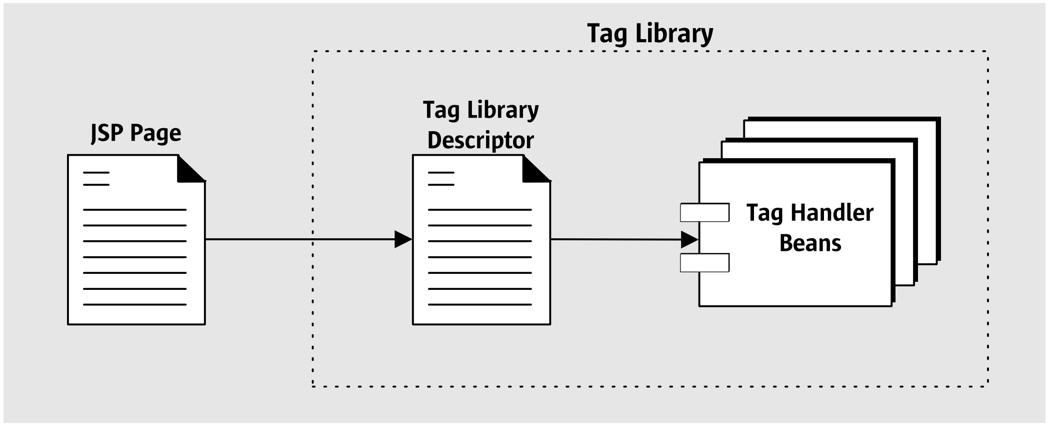


Figure 8 *Tag Library Architecture*

Forte for Java provides all the tools you need for creating your own tag libraries:

- Templates for creating TLDs
- A tool for generating tag handlers from a TLD
- A tool for packaging a tag library as a JAR file

For an example of how to use these facilities, see "Developing a Custom Tag Library—Tutorials" on page 59.

In addition to supporting the development of tag libraries, Forte for Java lets you to import third-party tag libraries and access them from your JSP pages. Forte for Java also provides several prepackaged tag libraries. These tag libraries enable you to:

- Access and perform operations on data sources using JDBC or Transparent Persistence.
- Iterate through rows and fields in a JDBC `ResultSet`; objects and their fields in a `Vector`, `Collection`, `List`, `Iterator`, or `Enumeration`; or elements (and their fields, if the element is an object) of a Java array.
- Conditionalize parts of a JSP page (using if/else logic).

For more information on these tag libraries, refer to online help.

Tag Library Descriptor

A tag library descriptor (TLD) is an XML document that defines a tag library. The web container uses a tag library's TLD to interpret custom actions on JSP pages that reference that tag library through a `taglib` directive. At the highest level, the TLD defines specifics of the tag library as a whole, such as its version number and the version number of its intended web container. At a lower level, it defines each tag in the library.

Forte for Java enables you to create and edit TLDs without writing XML code. You create a TLD from one of the TLD templates provided by Forte for Java. After you have created a TLD, you can edit it from the explorer through menu commands and through the properties and customizer windows of the TLD and its elements.

For example, you can define a tag in a TLD by selecting the TLD in the explorer and choosing Add Tag in the pop-up menu. You can then define an attribute of that tag by selecting the tag and choosing Add Attribute. These actions create tag and attribute elements with default values. After creating such elements, you can use their customizer windows to edit them. See "Developing a Custom Tag Library—Tutorials" on page 59 for an example of creating and editing a TLD.

Tag Handlers

A tag handler is a bean that implements the functionality of a custom action. There is a one-to-one correspondence between a custom action and a tag handler.

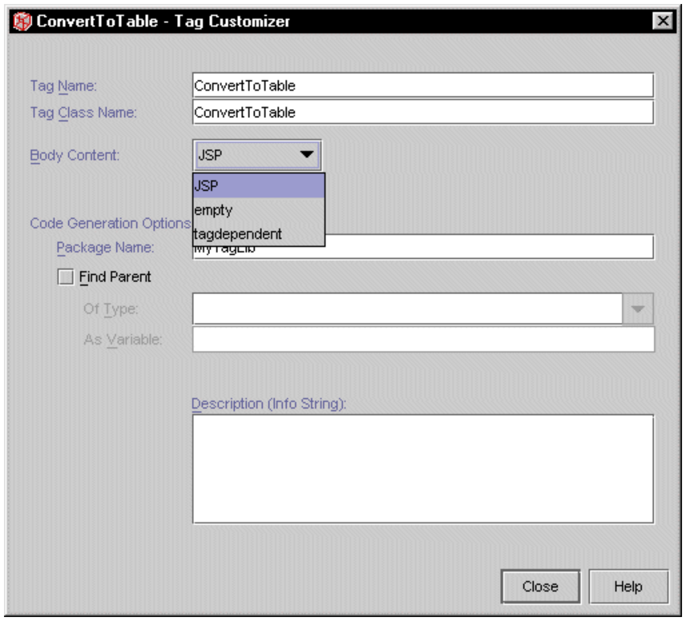
Custom Actions with Bodies

Custom actions, in principle, can contain bodies. That is, they can have beginning and ending tags that enclose other actions, scripting elements, or plain text.

For example, this sample custom action contains a body composed of plain text:

```
<mt:convertToTable>
type distance / a 30,000 / g 5,500 / z 200
</mt:convertToTable>
```

Whether or not a particular custom action can contain a body depends on how it is defined in the TLD. The Body Content field in the Tag Customizer lets you specify how the body is handled (you can access this window from the pop-up menu of the custom action's tag handler). As shown in Figure 9, you can choose one of these values: JSP, empty, or tagdependent.


ConvertToTable - Tag Customizer

Tag Name: ConvertToTable
Tag Class Name: ConvertToTable
Body Content: JSP

Figure 9 *Tag Customizer Window*

The following table explains the meaning of each of these choices.

Table 3 *Meaning of Body Content Field in Tag Customizer Window*

Body Content field	Meaning
JSP	Body content is optional. The web container *evaluates JSP elements* and then passes the body to the tag handler. The tag handler processes the body and writes output to the `out` object according to your programming logic.
empty	Body content is not permitted.
tagdependent	Body content is optional. The web container *does not evaluate JSP elements* but does pass the body to the tag handler. The tag handler processes the body and writes output to the `out` object according to your programming logic.

All tag handlers implement `javax.servlet.jsp.tagext.Tag`. Tag handlers that do not accept or process a body need only implement this interface. Tag handlers that process a body must also implement `javax.servlet.jsp.tagext.BodyTag`. This interface provides additional methods for handling this processing.

Generated Tag Handlers

In Forte for Java, you generate tag handlers from a TLD. These generated tag handlers implement the interface(s) appropriate for their corresponding custom actions, as defined in the TLD (either the `Tag` interface or both the `Tag` and `BodyTag` interfaces). Additionally, all of the tag handlers' required class members (fields, methods, and properties) are generated for you. The exact list of class members depends on your TLD, but will always include the methods required by the interface(s) that your tag handler implements.

The specific class members generated depend on the interface(s) your tag handlers implement, and also on the attributes and scripting variables that you have declared in your TLD. For example, if you declare an attribute named `myAttribute` in your TLD, a property named `myAttribute` will be generated in the tag handler.

Methods Generated

The following table lists the methods that Forte for Java creates when you generate tag handlers. They are listed according the interface(s) the tag handler implements. Methods used to get and set properties are not listed.

Table 4 *Generated Methods in Tag Handlers*

Interface	Method

`Tag`	`doEndTag`
	`doStartTag`
	`otherDoEndTagOperations`
	`otherDoStartTagOperations`
	`shouldEvaluateRestOfPageAfterEndTag`
	`theBodyShouldBeEvaluated`
	`theBodyShouldBeEvaluatedAgain`

`BodyTag`	All of the methods generated for the Tag interface plus the following:
	`doAfterBody`
	`writeTagBodyContent`

Regenerating Tag Handlers

To develop your tag library, you add programming logic to the tag handlers to provide the functionality your custom actions require. During the course of your development, you might have to add additional attributes or scripting variables to your TLD. If this is the case, you will then need to regenerate your tag handlers so that the corresponding class members are created. When you do this, some of the tag handler' s methods are regenerated and some are left untouched.

Forte for Java regenerates the methods `doStartTag`, `doEndTag`, and `doAfterBody`. The source editor does not permit you to edit these methods because your changes would be overwritten when you regenerate tag handlers.

Instead of editing the methods that get regenerated, place your custom code in methods that these regenerated methods call. For example, the `doStartTag` method calls the `otherDoStartTagOperations` and `theBodyShouldBeEvaluated` methods. The JSP specification indicates that you should use the `doStartTag` method for processing that needs to be performed at the beginning of the tag, before the body of the tag is evaluated.

This method is also used to return a Boolean to indicate whether the body *should* be evaluated. In Forte for Java, use the `otherDoStartTagOperations` method for the processing that needs to be performed at the beginning of the tag, and use the `theBodyShouldBeEvaluated` method to return the Boolean. Code that you place in these two methods will not be affected by regeneration.

The following table indicates which methods are regenerated and which methods you may edit.

Table 5 *Editable Methods in Tag Handlers*

Do not edit these methods	Put your custom code in these methods instead
`doEndTag`	`otherDoEndTagOperations` `shouldEvaluateRestOfPageAfterEndTag`
`doStartTag`	`otherDoStartTagOperations` `theBodyShouldBeEvaluated`
`doAfterBody`	`writeTagBodyContent` `theBodyShouldBeEvaluatedAgain`

Accessing a Tag Library

You use the functionality of a tag library by coding custom actions in a JSP page. For the custom actions to access the tag library, the JSP page must declare the tag library with a `taglib` directive.

For example:

```
<%@taglib uri="/WEB-INF/lib/myTagLib.jar" prefix="mt" %>
```

The `uri` attribute of a `taglib` directive references either the tag library descriptor (TLD) or, as in the previous example, a JAR file containing both the TLD and the tag handler beans. You must place the `taglib` directive before any custom actions that use the tag library.

The previous example's `uri` attribute specifies a hard-coded path relative to the root of the web module (the leading slash denotes the web module root). However, it is also possible to specify this attribute in a more abstract manner that permits it to be configured at a post-development date. To do this, you must create a `taglib` element in the web module deployment descriptor (the `web.xml` file). You then configure this `taglib` element so that it maps a URI to the physical location of your TLD or tag library JAR file.

For example, the following `taglib` element makes the TLD located at `/WEB-INF/tlds/myTagLib.tld` accessible through the URI `myTags`:

```
<taglib>
 <taglib-uri>myTags</taglib-uri>
 <taglib-location>/WEB-INF/tlds/myTagLib.tld</taglib-location>
</taglib>
```

For an example of how Forte for Java facilitates this mapping procedure, see "Developing a Custom Tag Library—Tutorials" on page 59.

With the previous mapping declared, you could make the tag library accessible to a JSP page by placing the following `taglib` directive in the JSP page:

```
<%@taglib uri="myTags" prefix="mt" %>
```

You must place the `taglib` directive somewhere before the first custom action that uses the tag library.

If your `taglib` directive references a TLD file rather than a tag library JAR file, as would be likely during tag library development, you must ensure that the TLD specifies the class names of the tag handlers and that the tag handlers are in your classpath. Forte for Java performs both these tasks for you automatically when you generate tag handlers.

You use the `prefix` attribute of a `taglib` directive to specify an identifier by which you refer to the tag library from custom actions coded in the JSP page. For example the following custom action (presumed to be in the same JSP page as the preceding `taglib` directive) uses the prefix `mt` to refer to the tag library. The string `table` specifies the tag handler that will process this custom tag.

```
<mt:table results="productDS"/>
```

The mapping between the tag name (in this case, `table`) and the tag handler bean is specified in the TLD file. You can edit this mapping in the Tag Customizer window, which is accessible in the explorer from the tag's pop-up menu.

Custom actions may create objects and make them available in the JSP page as scripting variables. Scripting variables can be accessed by other actions and scripting elements on the JSP page.

Developing a Custom Tag Library— Tutorials

This section presents three short tutorials that demonstrate how to perform several important tasks involved in developing a tag library.

- The first tutorial walks you through creating a simple "Hello World" tag library and accessing it from a JSP page.
- The second tutorial shows you how to add an attribute to your tag library and regenerate the tag handler bean.
- The third tutorial shows you how to package your tag library as a JAR file and then access it from a JSP page.

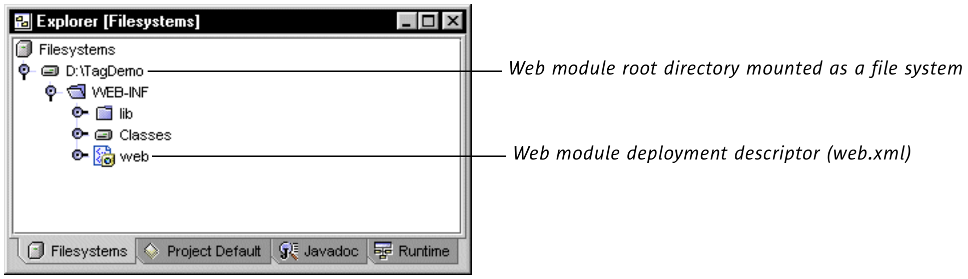
Creating a Tag Library— Tutorial

In this tutorial, you will create a tag library and, within it, create a single tag that outputs the string `Hello World`. You will also create and execute a JSP page that accesses your library and displays the string.

➤ **To create the tag library:**

1 Create a new web module. For information on how to do this, see "Creating a Web Module" on page 36.

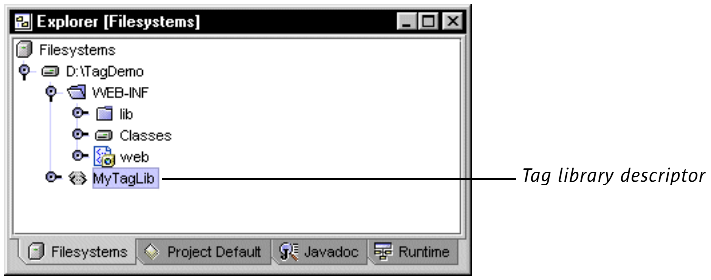
This operation creates a web module directory structure, as shown in the following figure.



2 Create a tag library descriptor in your web module and name it `MyTagLib`.

To do this, select the web module's root directory in the explorer, and choose New>JSP & Servlet>TagLibraries>Blank TagLibrary from its pop-up menu. Type **`MyTagLib`** into the name field of the wizard that opens, and click Finish.

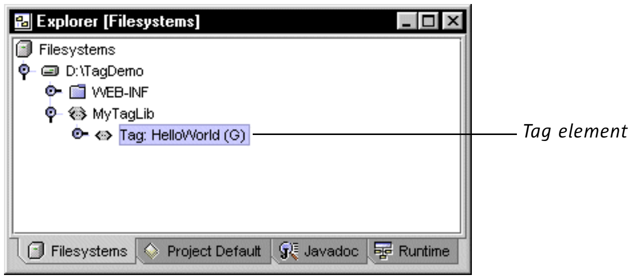
This operation creates a tag library descriptor and opens the Tag Library Customizer. (Close the Tag Library Customizer.) The following figure shows the resulting tag library descriptor:



3 Add a tag element named `HelloWorld` to your tag library descriptor, and specify `HelloWorldTag` as its handler class.

To do this, select your tag library descriptor in the explorer (the `MyTagLib` node), and choose Add Tag from its pop-up menu. In the dialog box that opens, type **`HelloWorld`** in the Tag Name field. In the Tag Class Name field, type **`HelloWorldTag`** (tabbing out of the Tag Name field does this automatically). Click Close.

The following figure shows the newly created tag element:

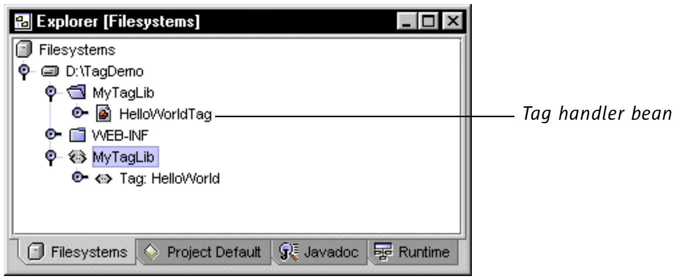


The G in parentheses following the tag name indicates that changes have been made to the tag since the last time its tag handler was generated (because we have not yet generated the tag handler).

4 Generate a tag handler bean.

To do this, select your tag library descriptor in the explorer, and choose Generate Tag Handlers from its pop-up menu.

This operation generates a package named `MyTagLib` in the root directory of the web module. This package contains the tag handler bean `HelloWorldTag`, as shown in the following figure.

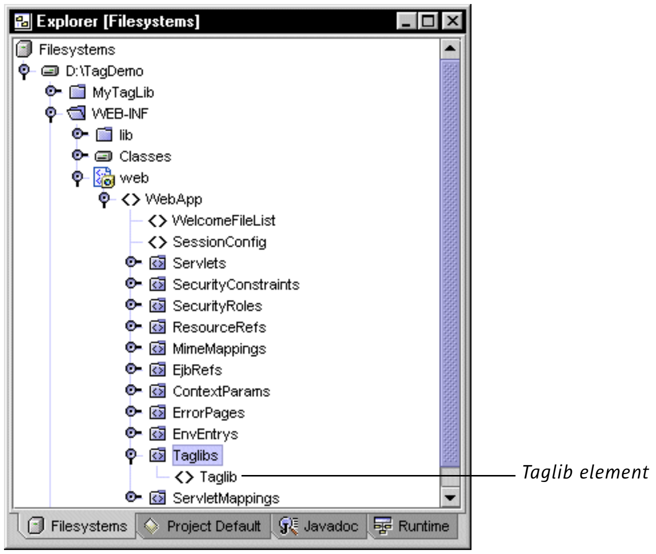


5 Modify the `otherDoStartTagOperations` method of the `HelloWorldTag` bean by adding this code to it and then compiling:

```
try{
 JspWriter out = pageContext.getOut();
 out.println("Hello World");
}
catch (Exception e){
 System.out.println(e);
}
```

6 Add a `taglib` element to your web module deployment descriptor (web.xml).

To do this, open the web module's `WEB-INF` directory, then the `web` node, and finally the `WebApp` node. Select the `Taglibs` node, and from its pop-up menu choose New Taglib.



7 Map the location of the tag library descriptor to the URI `myTags`.

To do this, open the properties window for the Taglib element you created in the previous step. Set the Taglib Location field to `/MyTagLib.tld` and the Taglib URI field to `myTags`.

This operation makes the tag library accessible to a JSP page through the URI `myTags`.

Note The leading slash in the Taglib Location field (`/MyTagLib.tld`) denotes the root of the web module.

8 Create a new JSP page and name it `TestCustomTag`.

To do this, select the root directory of your web module in the explorer and choose New>JSP & Servlet>JSP(HTML) from its pop-up menu. Type **`TestCustomTag`** in the name field of the wizard that opens. Click Finish.

9 Add the following code to your JSP page on the line after the HTML `<body>` tag:

```
<%@taglib uri="myTags" prefix="mt" %>
<mt:HelloWorld />
```

10 Restart the server and execute the JSP page.

To do this, choose Execute (restart server) from the pop-up menu of the JSP page.

The web browser should display a page that reads `Hello World`.

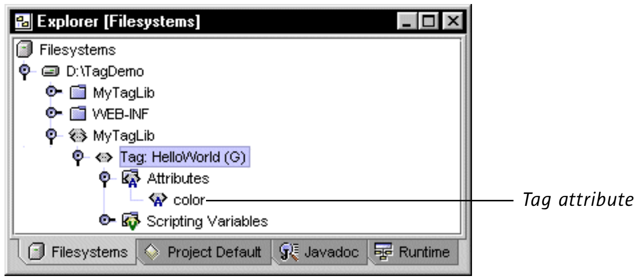
Adding an Attribute to a Tag Handler— Tutorial

The following tutorial starts where the previous tutorial left off. It shows you how to add an attribute to your "Hello World" tag that controls the color in which its output is displayed by your web browser. As part of the procedure, you learn how to regenerate tag handler beans.

➤ **To add the attribute to your tag handler:**

1 Add an attribute named `color` to your `HelloWorld` tag.

To do this, select the `HelloWorld` tag in the explorer and choose Add Tag Attribute from its pop-up menu. When the Tag Attribute Customer window opens, type **`color`** into the Name field and click Close.



2. Regenerate the `HelloWorld` tag handler.

   To do this, select the `MyTagLib` tag library descriptor in the explorer, and choose Generate Tag Handlers from its pop-up menu.

   This operation generates a property named `color` on the tag handler, and three corresponding class members: a field named `color` and methods named `getColor` and `setColor`.

3. Modify the `otherDoStartTagOperations` method of the tag handler so that the text it outputs is colored according to the value assigned to the `color` attribute.

   To do this, modify the `println` statement to read as follows and then compile the class:

   ```
   out.println("<p><font color=" + getColor() + ">Hello World</font>");
   ```

4. Modify the `HelloWorld` action in the `TestCustomTag` JSP page so that its output is colored red.

   To do this, change the action to read as follows and then compile the JSP page:

   ```
   <mt:HelloWorld color="red"/>
   ```

5. Restart the server and execute the JSP page.

   To do this, choose Execute (restart server) from the pop-up menu of the JSP page.

   Your web browser should now display `Hello World` in red.

Packaging a Tag Library and Accessing the JAR— Tutorial

The following tutorial shows you how to package the tag library you developed in the previous tutorials and then access it from your JSP page.

➤ **To package and access your tag library:**

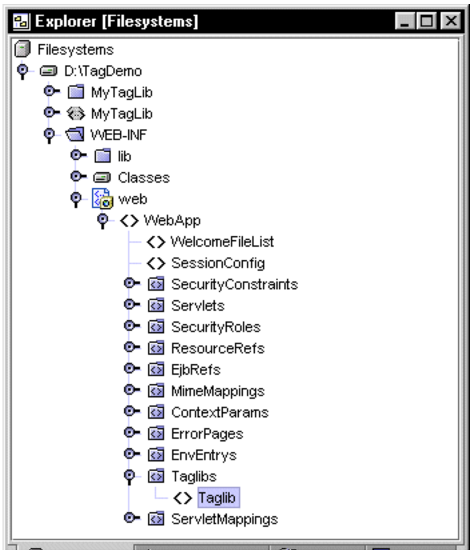
1. From the pop-up menu on the `MyTagLib` tag library descriptor, choose Create Tab Library JAR.

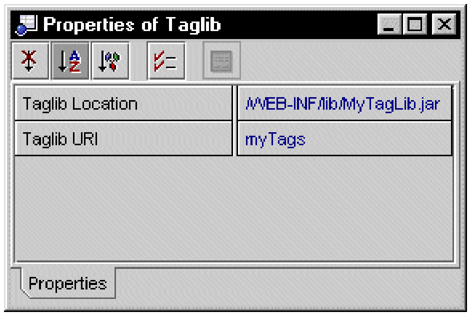
   This operation creates a JAR file named `MyTagLib.jar` in the web module's root directory (the file extension is not displayed by the explorer).

2. Use the Cut and Paste commands on the pop-up menu of the JAR file to move the JAR file into the `WEB-INF/lib` directory.

3 Modify the `Taglib` element in the deployment descriptor so that it maps to the `MyTagLib` JAR file rather than the `MyTagLib` directory in which you developed the tag library.

To do this, open the web module's `WEB-INF` directory, then the `web` node, the `WebApp` node, and finally the `Taglibs` node. Open the Properties window for the `Taglib` element and type **`/WEB-INF/lib/MyTagLib.jar`** into the Taglib Location field, as shown in the following figure.





Taglib Location	/WEB-INF/lib/MyTagLib.jar
Taglib URI	myTags

Note When mapping a `taglib` element to a tag library that is packaged as a JAR file, do not specify the location of the tag library descriptor. Specify only the location of the JAR file. The location of the tag library descriptor within the JAR file is known by the web container.

4 Restart the server and execute the JSP page.

To do this, choose Execute (restart server) from the pop-up menu of the JSP page.

As in the previous tutorial, your web browser should display `Hello World` in red.

Index