Web Applications Developer's Guide

Innovation Release

Version 10.0

April 2017

WEBMETHODS

This document applies to webMethods Integration Server and Software AG Designer Version 10.0 and to all subsequent releases.

Specifications contained herein are subject to change and these changes will be reported in subsequent release notes or new editions.

Document ID: IS-WEBAPP-DG-100-20170419

Table of Contents

About this Guide

This guide is for developers who want to incorporate web applications into the webMethods Integration Server environment, secure and access these applications, and use the webMethods tag library for JSP to invoke Integration Server services and obtain pipeline data.This guide assumes you already know how to develop and deploy web applications.

Document Conventions

Convention	Description
Bold	Identifies elements on a screen.
Narrowfont	Identifies storage locations for services on webMethods Integration Server, using the convention *folder.subfolder:service* .
UPPERCASE	Identifies keyboard keys. Keys you must press simultaneously are joined with a plus sign (+).
Italic	Identifies variables for which you must supply values specific to your own situation or environment. Identifies new terms the first time they occur in the text.
`Monospace font`	Identifies text you must type or messages displayed by the system.
{ }	Indicates a set of choices from which you must choose one. Type only the information inside the curly braces. Do not type the { } symbols.
|	Separates two mutually exclusive choices in a syntax line. Type one of these choices. Do not type the | symbol.
[ ]	Indicates one or more options. Type only the information inside the square brackets. Do not type the [ ] symbols.
...	Indicates that you can type multiple options of the same type. Type only the information. Do not type the ellipsis (...).

Online Information

Software AG Documentation Website

You can find documentation on the Software AG Documentation website at "http://documentation.softwareag.com". The site requires Empower credentials. If you do not have Empower credentials, you must use the TECHcommunity website.

Software AG Empower Product Support Website

You can find product information on the Software AG Empower Product Support website at "https://empower.softwareag.com".

To submit feature/enhancement requests, get information about product availability, and download products, go to "Products".

To get information about fixes and to read early warnings, technical papers, and knowledge base articles, go to the "Knowledge Center".

Software AG TECHcommunity

You can find documentation and other technical information on the Software AG TECHcommunity website at "http://techcommunity.softwareag.com". You can:

- Access product documentation, if you have TECHcommunity credentials. If you do not, you will need to register and specify "Documentation" as an area of interest.
- Access articles, code samples, demos, and tutorials.
- Use the online discussion forums, moderated by Software AG professionals, to ask questions, discuss best practices, and learn how other customers are using Software AG technology.
- Link to external websites that discuss open standards and web technology.

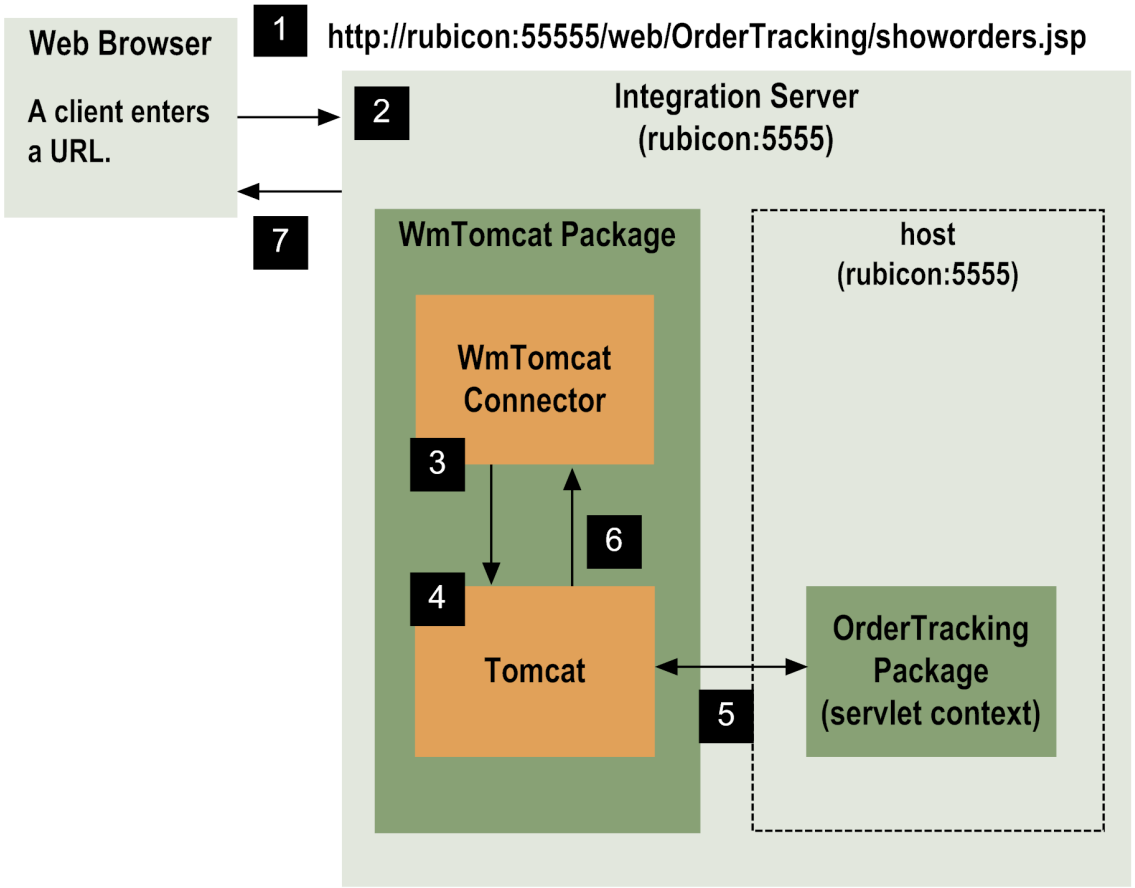
1 Concepts

How webMethods Integration Server Processes Web Applications

webMethods Integration Server supports Tomcat requests via the WmTomcat package, which is provided with Integration Server. By default, the WmTomcat package is installed with Integration Server. However, you must enable the WmTomcat package before you can begin using Integration Server to process web applications. The WmTomcat package includes Apache Tomcat.

Important: The WmTomcat package is deprecated as of Integration Server 9.10. Existing web applications based on the WmTomcat package can be migrated to webMethods Application Platform.

You can use Integration Server as a general purpose servlet container and run web applications based on servlets and JSPs. The following diagram shows how Integration Server processes Tomcat requests.



Step	Description
1	In a browser, a user enters a URL for a web page. The HTTP request is sent to Integration Server.
2	Integration Server receives control. Because the URL contains the "/web" directive, Integration Server's dispatcher forwards the request to the logic in the WmTomcat package.
3	The *WmTomcat Connector* receives control. The WmTomcat Connector is webMethods-provided logic within the WmTomcat package that connects the webMethods engine to Tomcat, which is embedded in Integration Server. The webMethods engine consists of Integration Server HTTP functionality, logging, and an engine that executes services. The WmTomcat Connector parses the URL in the format that Tomcat requires and passes control to the embedded Tomcat.
4	The embedded Tomcat receives control. Tomcat accesses the web application files, including JSPs and servlets, for the web page to display in the browser. To access the web application files, Tomcat uses the parsed URL that the WmTomcat Connector provides. The host name and port identified in the URL identifies the host with which Tomcat interacts to access the web application files. In a typical Tomcat implementation, there might be multiple hosts. However, for WmTomcat there is a single host (i.e., localhost). In the above example, the Integration Server host name and port number is rubicon:5555. The *servlet context* is a component that represents a web application. For WmTomcat, a servlet context always corresponds to exactly one Integration Server package. This package is identified in the URL (e.g., OrderTracking). Because the web application files are stored in the Integration Server file system, the Tomcat Engine can access the files from the *Integration Server_directory*\instances*instance_name* \packages *packageName* \web directory of a package. In the above example, the web application files reside in the *Integration Server_directory*\instances *instance_name* \packages\OrderTracking\web directory.
5	Control passes to the servlet context by way of the corresponding Integration Server package. The web application (e.g., showorders.jsp) that was specified in the original URL from the browser is invoked.
6	After the final web page is built and displayed in the browser, control is returned to Tomcat, which in turn returns the web page to the WmTomcat Connector.

Step	Description
7	The WmTomcat Connector returns the HTTP response to the HTTP functionality within Integration Server. Integration Server responds to the original HTTP requests with the response (i.e., the web page to display at the browser.)

Key Differences with the webMethods Tomcat Implementation

This section describes key differences in the webMethods implementation of Tomcat when compared to a typical Tomcat implementation.

Servlet Context

For WmTomcat, a servlet context always corresponds to exactly one Integration Server package. Because each context is associated with a single package, each web application is associated with a single package. The corresponding WmTomcat ROOT context package and corresponding files are in the \web directory of the WmTomcat package itself.

About Servlet Mapping

By default, Integration Server disables the Tomcat invoker servlet mapping. You can enable the servlet mapping by editing the web.xml file in the *Integration Server_directory*\instances*instance_name* \web\conf directory. However, this is a security risk as it allows disclosure of the JSP source code. Because JSP source code is sometimes sensitive (especially if it includes passwords for back-end systems), avoid disclosing the JSP source code by leaving the invoker servlet mapping disabled.

File Locations for WmTomcat

With WmTomcat, the locations where web application files and Tomcat-specific files are stored is different from the locations in a typical Tomcat installation. This section describes the locations for the following files:

- Web application files
- Class files and jar files that are shared by multiple web applications
- Tomcat configuration files
- Tomcat temporary files used when processing web application requests

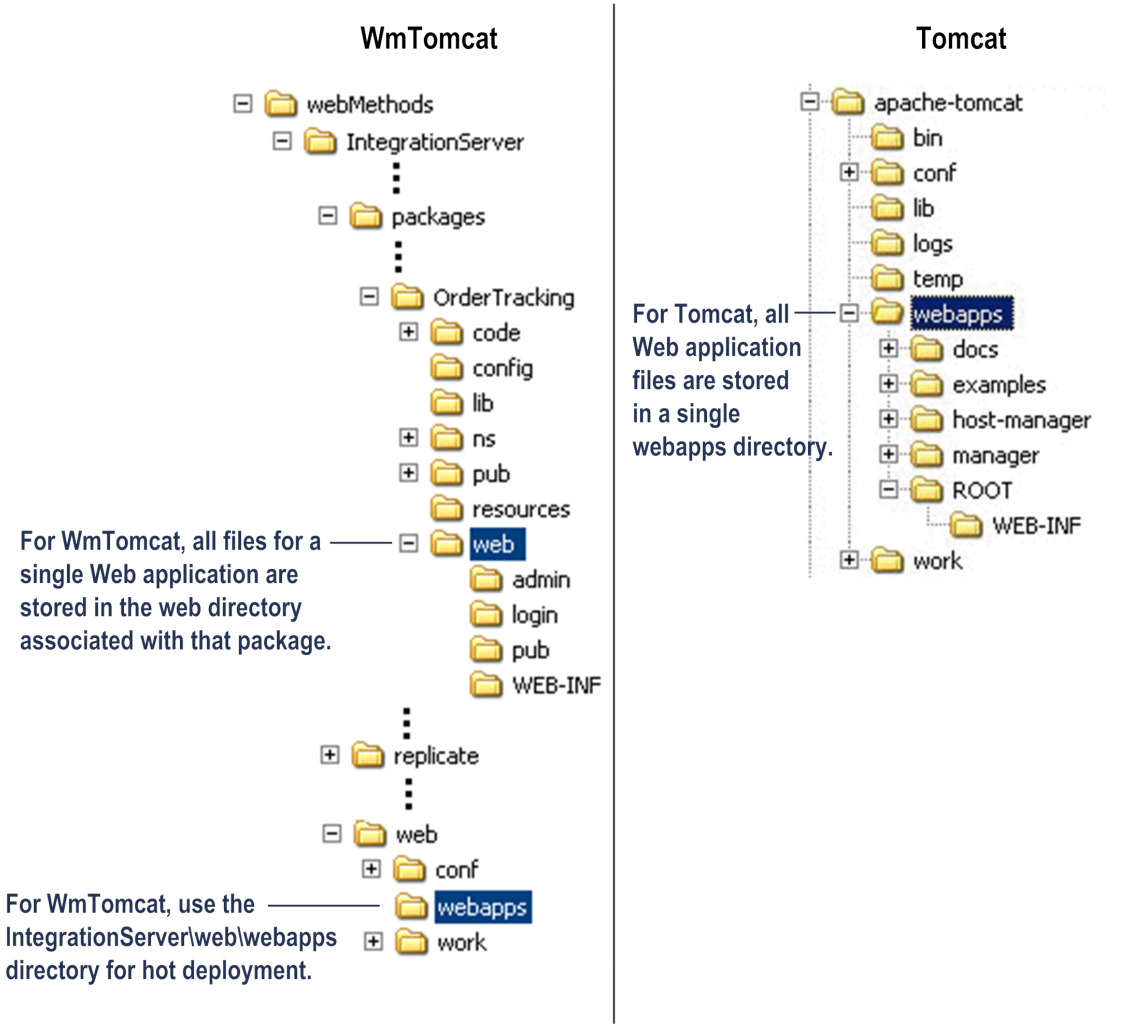
Where You Store Web Application Files

Store web application files as follows:

- If you are using WmTomcat, place all files for a single web application into the \web directory of the Integration Server package associated with the web application. For example, if you have a web application that uses the package OrderTracking, place the web application files into the *Integration Server_directory*\instances*instance_name* \packages\OrderTracking\web directory.
- If you are using hot deployment, place the war file for the web application in the *Integration Server_directory*\instances*instance_name* \web\webapps directory. WmTomcat unzips the war file and places the extracted web application files into the web directory of the package for the web application. For example, if you hot deploy the OrderTracking.war file, WmTomcat places the extracted files into the *Integration Server_directory*\instances*instance_name* \packages\OrderTracking\web directory. For more information about hot deployment, see "About Hot Deployment of the War File" on page 28.

In a typical Tomcat implementation, you place all files for all web applications into a single directory.

The following illustrates where to store web applications for WmTomcat and where to store them in a typical Tomcat implementation.

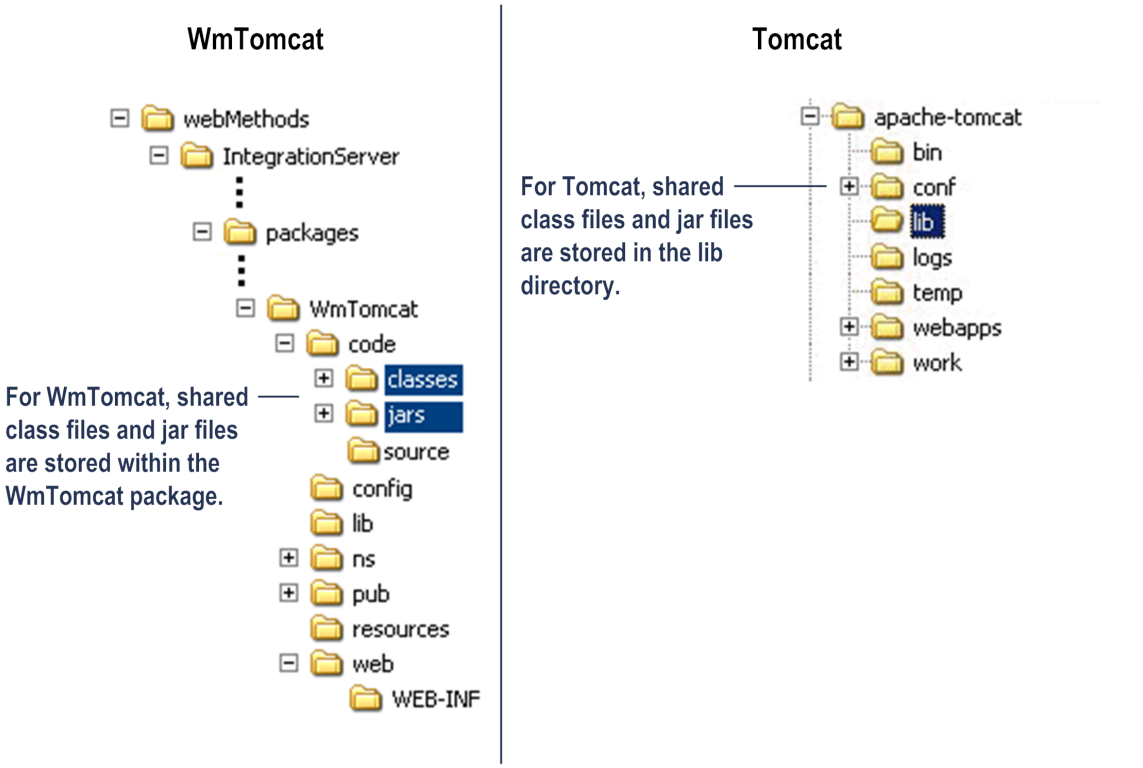


Where You Store Shared Class and Jar Files

When you use WmTomcat, place class files and jar files that are shared by multiple web applications in the following directories within the WmTomcat package:

- Place shared class files in *Integration Server_directory*\instances*instance_name*\packages\WmTomcat\code\classes
- Place shared jar files in *Integration Server_directory*\instances*instance_name*\packages\WmTomcat\code\jars

The following illustrates where to store shared class and jar files for WmTomcat and where they are stored in a typical Tomcat implementation.



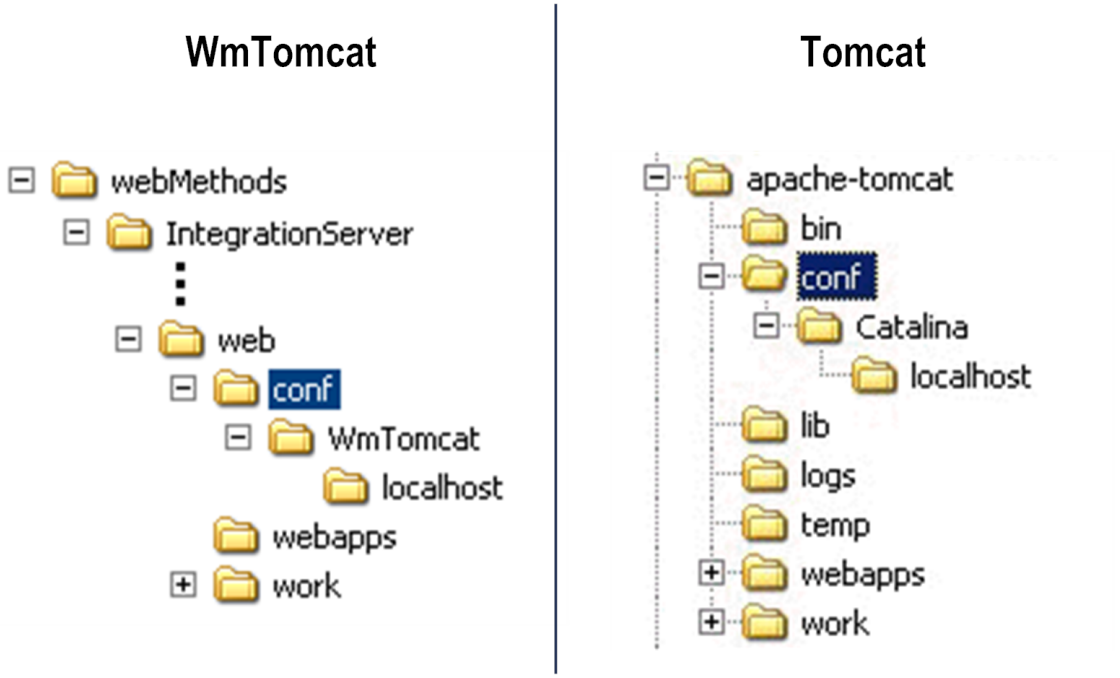
Note: The location of shared class and jar files that are used by multiple web applications is *different* from where you place shared jar files that are used by services in multiple Integration Server packages.

- **Shared class and jar files that are used by multiple web applications** must be placed in the WmTomcat package as shown above.
- **Shared jar files that are used by services in multiple IS packages** are stored within the same package as a service, but are placed in the *Integration Server_directory*\instances*instance_name* \lib\jars directory.

Where Tomcat Configuration Files Are Stored

For WmTomcat, Tomcat configuration files are stored in the *Integration Server_directory*\instances*instance_name* \web\conf directory. Tomcat administers this directory.

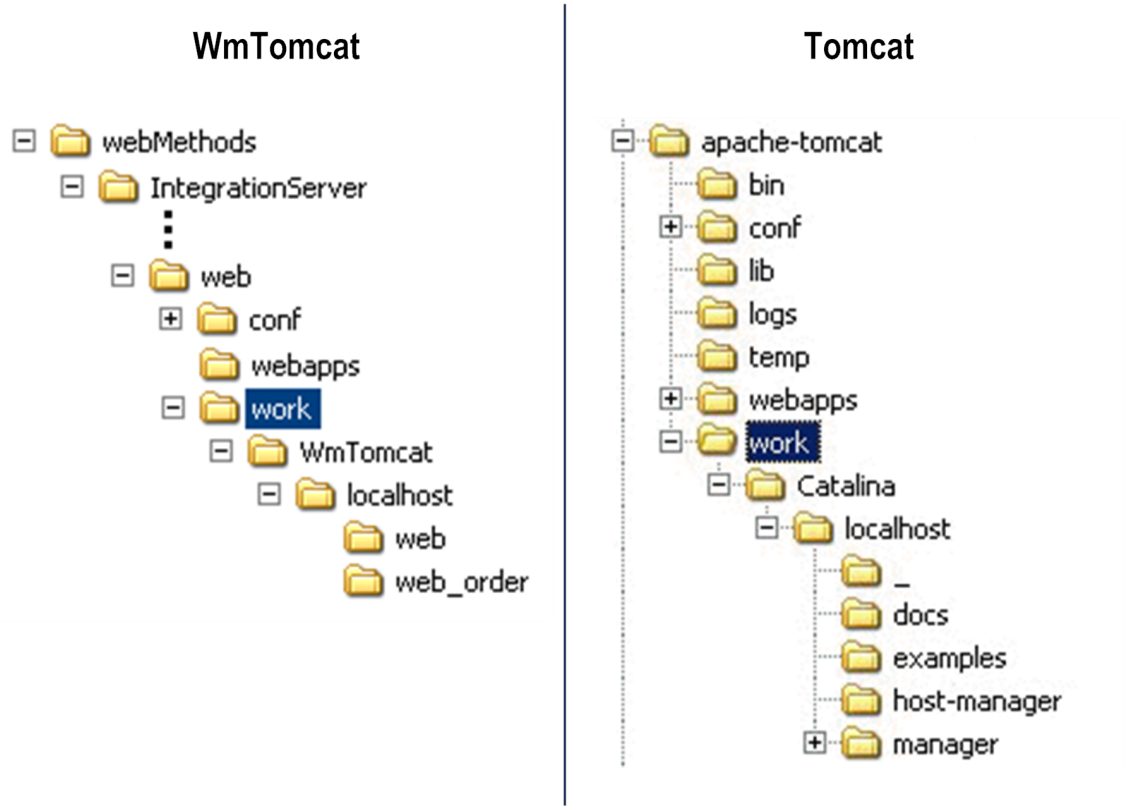
The following illustrates where Tomcat configuration files are stored for WmTomcat and where they are stored in a typical Tomcat implementation.



Where Tomcat's Working Directory Is Located

For WmTomcat, Tomcat uses the *Integration Server_directory*\instances *instance_name* \web\work directory for its working directory. Tomcat administers this directory.

The following illustrates where Tomcat configuration files are stored for WmTomcat and where they are stored in a typical Tomcat implementation.



URL for the WmTomcat ROOT Context Package

For WmTomcat, use the following URL to access the WmTomcat ROOT context package:

http://*hostname* :*portnumber* /web/

For example:

http://localhost:5555/web/

Note that the URL ends with "/web". In a typical Tomcat implementation, the "/web" element is not part of the URL for the root page.

Tip: Entering the URL to display the root page is a good test to ensure that WmTomcat is running and that the Tomcat engine is functioning correctly.

URL You Use to Invoke a Web Application

For WmTomcat, the URL to invoke a web application has the following format, which includes a "/web" element:

http://*hostname* :*portnumber* /web/*packageName* /*fileName* .jsp

For example: http://localhost:5555/web/JSPSample/showorders.jsp

The “/web” element is required as a signal to Integration Server that this URL is for a web application and must be sent to the WmTomcat package for processing.

In a typical Tomcat implementation, the “/web” element is not part of the URL to invoke a web application.

How WmTomcat Executes the Tomcat Buffering Methods

Tomcat supplies methods such as getBufferSize and setBufferSize for buffering responses back incrementally to the browser. In a typical Tomcat implementation using these buffering methods, the page the user sees in the web browser is built in stages. For example, the web application logic might buffer the text back, and then shortly afterward the web application logic would buffer back the remaining graphics.

In the webMethods Tomcat implementation, web page content cannot be buffered back to the browser incrementally because Integration Server does not support incremental buffering of HTTP responses. Your web applications can still include the Tomcat buffering methods, but they will not function in the typical way. When a web application uses these buffering methods, the WmTomcat logic will accept the incremental buffers and hold incremental content in memory. After the entire web page response is complete, the WmTomcat logic will buffer the full page to the browser at one time.

Administering Your Web Applications

For WmTomcat, you use Integration Server Administrator to undeploy and delete a web application. You perform these functions using the Package Management functionality to manage the package that contains the web application.

- **To undeploy a web application**, disable the package. If you want to start using the web application again, you can enable the package.
- **To delete a web application**, delete the package. Integration Server offers a safe delete function, which allows you to restore the package if you decide you want to recover a deleted package.

2 Guidelines for Creating Web Applications

About Creating Web Applications

When you create servlets, JSPs, and supporting files for your web application, use the guidelines described in the following table:

Task	Guideline
Using a text editor to create files	Create the files using your preferred text editor or IDE.
Specifying literal text	In the JSP files, type text exactly as you want it to appear in the document that you want Integration Server to return to the client.
Invoking a service from a web application	To invoke an Integration Server service from a servlet, invoke it directly. For an example, see "Invoking an Integration Server Service from a Servlet" on page 20. To invoke an Integration Server service from a JSP, you must use the webMethods tag library for JSP.

Using the webMethods tag library for JSP	In the JSP files, use tags from the webMethods tag library for JSP to invoke Integration Server services and obtain pipeline data. To use the webMethods tag library for JSP, do the following:
	
	■ Add this directive above all tags in a JSP file:
	`<%@ taglib uri="http://webm-taglib.tld" prefix="webm" %>`
	The system uses the prefix webm in front of each tag name in the tag library descriptor file (e.g., `<webm:value>`).
	
	■ Insert tags at the points where you want their results to appear. For example, if you have a customer's last name stored in the LastName pipeline variable and want to insert the name into a table cell, your tag might look like this:
	`<TD> <webm:value variable="LastName"/> </TD>`
	
	■ Set up the web application package so you can use webMethods tags within your web application. For instructions, see "Setting Up

Task	Guideline
	a Package so a Web Application Can Use the webMethods Tags for JSP" on page 19. **Important:** Keep the number of tags in a single JSP to a minimum. A good practice is to only include the tags needed to perform a single task in each JSP. A side benefit of keeping JSPs small is that small JSPs are easier to debug and use. For a summary of webMethods tags for JSP and instructions for using them, see " webMethods Tag Library for JSP" on page 45.
Storing supporting web resource files	Create supporting web resource files for the JSP (for example, HTML pages or image and sound files) as necessary. Store these files in appropriate subdirectories, as described in "The Integration Server Package Directory Structure" on page 26.
Selecting a file extension	Save the JSP in a text file with a ".jsp" extension (for example, showorders.jsp).

Setting Up a Package so a Web Application Can Use the webMethods Tags for JSP

This section describes how to set up the web application's Integration Server package if you want to use the webMethods tags for JSP within your web application.

To set up a package so a web application can use the webMethods tags

1. If the Integration Server package does not exist, start Designer and create a new package. The package must match the name of the war file you will be using. For more information about creating packages, see *webMethods Service Development Help*.
2. If the Integration Server package already exists and the package was created before Integration Server 6.0, create a \web directory beneath the package's root directory, as follows:

   Integration Server_directory\instances*instance_name* \packages*packageName* \web
3. Make a local copy of the webMethods tag library for JSP in the package. This will improve the performance of your web application.

a. Copy the webm-taglib.tld file that is in the *Integration Server_directory* \instances *instance_name* \web directory.

b. Place the copy in the *Integration Server_directory* \instances *instance_name* \packages*packageName* \web\WEB-INF directory of the Integration Server package for the web application.

4. Update the web.xml file for the web application to identify the location of the webMethods tag library (webm-taglib.tld). The web.xml file for the web application should be stored in the following directory:

   Integration Server_directory\instances*instance_name* \packages*packageName* \web \WEB-INF

   To update the web.xml file, add the following `<taglib>` section:

```
<taglib>
      <taglib-uri>http://webm-taglib.tld</taglib-uri>
      <taglib-location>/WEB-INF/webm-taglib.tld</taglib-location>
</taglib>
```

Invoking a Service from a Web Application

You can invoke an Integration Server service either from a JSP or a servlet.

Invoking an Integration Server Service from a JSP

You can invoke an Integration Server service from a JSP using the `<webm:invoke>` tag, which is in the webMethods tag library for JSP. For more information about the `<webm:invoke>` tag, see " <webm:invoke>" on page 53.

Invoking an Integration Server Service from a Servlet

You can also invoke an Integration Server service from a servlet. The following shows a sample servlet that illustrates how to invoke an Integration Server service. It takes as input two strings (e.g., "hello" and "world"), invokes the pub.string:concat service to concatenate the two input strings, and returns as output the concatenated string (e.g., "hello world").

Note: The com.wm.app.b2b.server.Service class which is imported in the following sample servlet is located in wm-isserver.jar.

```
import java.io.IOException;
import java.io.PrintWriter;

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

import org.apache.commons.logging.Log;
```

```
import org.apache.commons.logging.LogFactory;

import com.wm.data.*;
import com.wm.app.b2b.server.Service;

public class ServiceServlet extends HttpServlet {

     // Allows you to have finer control over the logging levels
     private static Log log = LogFactory.getLog( ServiceServlet.class );

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

        PrintWriter out = response.getWriter();

        ServletContext ctx = getServletContext();
        // Will show on IS level 8
        ctx.log("logging a string in the IS server log via the Servlet API");
        log.info("Servlet logging at the IS level 4");
        log.debug("Servlet logging at the IS level 8");

        IDataCursor  cursor = idata.getCursor();
        String param1 = request.getParameter ("param1");
        String param2 = request.getParameter ("param2");

        // Pass the parameters to the pipeline
        IDataUtil.put( cursor, "inString1",  param1 );
        IDataUtil.put( cursor, "inString2",  param2 );

        try{

           idata = Service.doInvoke( "pub.string", "concat", idata );

        }catch( Throwable t){
           log.error("The service failed: " , t);
        }
        // Get the concatenated String returned by the Service
        String value = (String)IDataUtil.get( cursor, "value" );
        cursor.destroy();
        out.println("<HTML>");
        out.println("<P>Invoked the IS service <b>pub.string:concat</b> </P>");
        out.println("<UL>");
        out.println("<FONT COLOR=RED>");
        out.println("<LI>param1 = " + param1 + "</LI>");
        out.println("<FONT COLOR=BLUE>");
        out.println("<LI>param2 = " + param2 + "</LI>");
        out.println("<FONT COLOR=RED>");
        out.println("<LI>Service returned = " + value + "</LI>");
        out.println("</UL>");
        out.println("</html>");

     }
}
```

Reloading the WmTomcat Package on UNIX Platforms

At times, you might need to reload the WmTomcat package. For example, you would need to reload the package if you change the global deployment descriptor file *Integration Server_directory*\instances*instance_name* \web\conf\web.xml.

However, you cannot reload the WmTomcat package on some UNIX platforms. If the UNIX platform you are using does not permit package reloads, do one of the following:

- If you are working in a production environment (where changes are not frequent), restart Integration Server instead of reloading the package.
- If you are working in a development environment (where changes are more frequent), consider using the Java Endorsed Standards Override Mechanism, which forces the JVM to use the Xerces version. The steps for installing the Java Endorsed Standards Override Mechanism are provided below.

To install the Java Endorsed Standards Override Mechanism

1. Create the *java-home* \jre\lib\endorsed directory, where *java-home* refers to the directory where the run-time software is installed.
2. Locate the resolver.jar, xercesImpl.jar, xml-apis.jar, and xmlParserAPIs.jar files in either the *Integration Server_directory* \lib\jars directory or the *Integration Server_directory* \instances*instance_name* \lib\jars directory.
3. Copy the files listed in the previous step to the *java-home* \jre\lib\endorsed directory that you created.
4. Restart the server.

After you install the Override Mechanism, you will be able to reload the WmTomcat package in UNIX.

3 Incorporating Web Applications into Integration Server

Before You Begin

Before you run a web application within the Integration Server environment, make sure your system meets the minimum requirements specified in *System Requirements for Software AG Products*. Also make sure Integration Server has access to the files it needs to compile JavaServer Pages.

Integration Server embeds the Tomcat servlet container and JSP Engine developed by the Apache Software Foundation in the form of the WmTomcat package. Through this support, you can:

- Deploy and execute JavaServer Pages, Java servlets, and their supporting files within the Integration Server environment without having to install and configure a separate web server and servlet engine.
- Incorporate web applications into new or existing webMethods packages.

The Tomcat servlet container and JSP Engine start automatically when Integration Server starts. You do not need to start the engine manually.

Note: Integration Server supports the internationalization and localization features of the servlet and JSP standards through the WmTomcat package.

Preparing to Run Web Applications

This section describes how to prepare for web application deployment. Before running the web application, do the following:

- Enable the WmTomcat package.

  Note: An Integration Server with an enabled WmTomcat package cannot be used with webMethods Application Platform.

- Identify package dependencies. For the web applications to load correctly, the packages containing web applications must reference the classes in the WmTomcat package by identifying the WmTomcat package as a package dependency. For more information about creating a package dependency, see *webMethods Service Development Help*.
- Verify that your environment is set up correctly to run embedded Tomcat in Integration Server.

To ensure that your environment is properly set up to run embedded Tomcat in Integration Server

- Enter the following URL in your web browser: http://*hostName* :*portNumber* /web/

  For example, you might enter: http://localhost:5555/web/

This is the URL for the ROOT page of WmTomcat and will appear only if the environment is set up correctly.

What Do I Need to Deploy Web Applications?

To deploy a web application within Integration Server, you need:

- An enabled WmTomcat package on Integration Server; the package is automatically installed in a disabled state with Integration Server. You must enable the WmTomat package before running and deploying web applications.

  Note: An Integration Server with an enabled WmTomcat package cannot be used with webMethods Application Platform.

- A web application, consisting of JSPs, servlets, and supporting files or a web archive (war) file packing those files into an appropriate directory structure
- The webMethods tag library (webm-taglib.tld), if you want to use webMethods tags to invoke Integration Server services and obtain pipeline data

About War Files

You can deploy JSPs, servlets, and their associated supporting files as individual files or you can package them into a web archive (war) file. The war file format is the same as the Java archive (jar) file format. When you deploy a web application on Integration Server using a war file, Integration Server unpacks the contents of the war file the first time you invoke the web application.

You can use any compression tool to produce a war file. When you create the war file, be sure to retain the directory structure described in "The Integration Server Package Directory Structure" on page 26.

About the webMethods Tag Library for JSP

The webMethods tag library for JSP is a set of customized implementation classes. You add tags from this library to a JSP to import actions that let you call Integration Server services or retrieve data from the webMethods pipeline.

The webMethods tag library is stored in a *tag library descriptor* file. The tag library descriptor file is an XML file that maps tags and attributes to implementation classes. The webMethods tag library descriptor file, webm-taglib.tld, resides in the *Integration Server_directory*\instances*instance_name* \web directory.

You can use your own JSP tags by adding them to a separate tag library for your web application. For more information about the JSP tags in the webMethods tag library, see " webMethods Tag Library for JSP" on page 45.

About the Deployment Descriptor File

The deployment descriptor file contains configuration settings for a web application (for example, defining /servlet as an invoke path for servlets, identifying ".jsp" as a file extension for JSPs, and specifying security constraints). Integration Server provides a global deployment descriptor file that defines the configuration settings for all web applications that run within the Integration Server environment. The global deployment descriptor file, named web.xml, resides in the *Integration Server_directory*\instances*instance_name*\web\conf directory.

If you want to override the default Integration Server web application settings or use the webMethods tag library, you can create your own deployment descriptor file within a web application. For information about overriding the default Integration Server web application security, see "About Securing the Web Application" on page 39.

The Integration Server Package Directory Structure

For Integration Server to recognize a web application, the application (either deployed as individual files or packaged into a war file) must reside in the appropriate subdirectories within an Integration Server package. The \web subdirectory within a package (that is, *Integration Server_directory*\instances*instance_name*\packages*packageName*\web) is the top-level, or *parent*, directory of a web application in Integration Server.

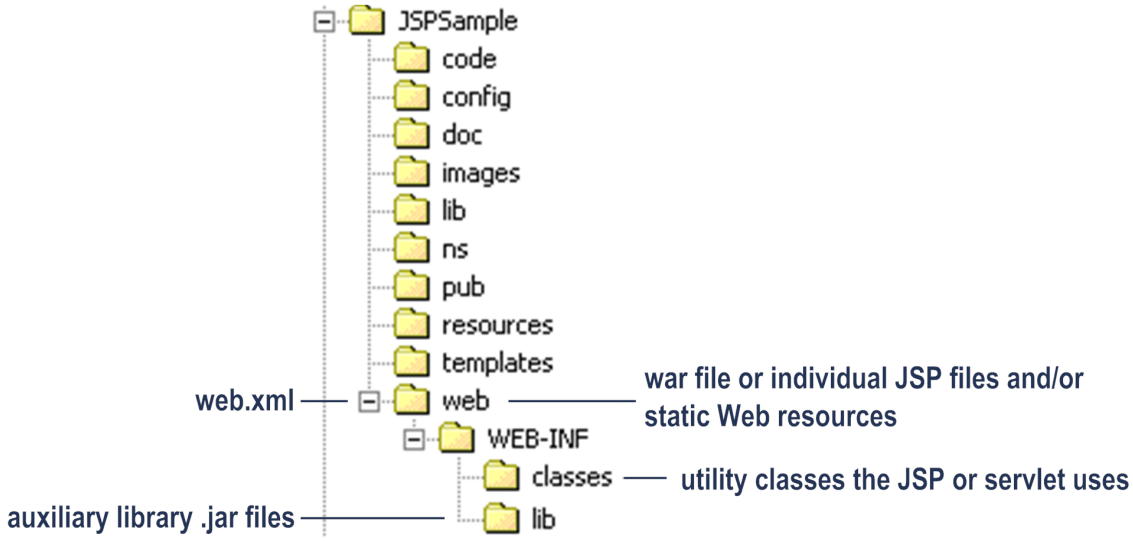
The first time you access a web application, Integration Server:

1. Looks in the \web subdirectory for the application's files
2. Unpacks the files, if you packaged them using a war file
3. Compiles the JSPs into Java servlets, if necessary

The compilation step might cause a slight delay the first time you access a JSP from a browser. This step occurs once per server and is repeated only if you modify the JSP. Integration Server does not recompile JSPs if you reload the package or restart Integration Server.

Note: Packages created in Integration Server before version 6.0 do not have a \web subdirectory. You can deploy web applications in these packages by creating this subdirectory beneath the package's root directory. For more information, see "About Deploying Web Applications" on page 27.

The following diagram illustrates a typical web application directory structure within an Integration Server package.



Integration Server stores the files associated with a web application in the Integration Server package directory structure as follows:

In this directory within an Integration Server package...	Integration Server stores...
\web	■ Web application files (also referred to as the application's *parent directory*)
\web*subdirectory*	■ JavaServer Pages and static web resources (for example, HTML pages and image or sound files)
\web\WEB-INF	■ Web application deployment descriptor file (web.xml) ■ Optional classes directory to hold servlet and utility classes ■ Optional lib directory to contain jar libraries (for example, classes or resources used by the JSPs or servlets) **Note:** Use the \web\WEB-INF directory to store files that you do not want users to be able to access.

About Deploying Web Applications

You can deploy a web application in the following ways:

- Hot deploy the war file that contains the web application files. For more information and instructions, see "About Hot Deployment of the War File" on page 28.
- Place the war file that contains the web application files in the \web directory of the Integration Server package for the web application. For instructions, see "Placing the War File in the Package \web Directory" on page 30.
- Place the web application files (JSP, supporting files, and subdirectories of supporting files) in the \web directory of the Integration Server package for the web application. For instructions, see "Placing Web Application Files in the Package \web Directory" on page 31.

Note: You can publish and subscribe to packages that contain web applications. When you create a package release, Integration Server includes the contents of the package's \web subdirectory. When you install a package on another server, the installation process also installs its associated web application files. The new server will execute the web application as long as the server meets the requirements described in "Before You Begin" on page 24.

About Hot Deployment of the War File

Hot deployment allows you to place a war file in a single directory and have WmTomcat unpack the web application files to the proper directory and update the application without having to manually reload the package. Before you use hot deployment, you must perform some setup tasks.

Setting Up Hot Deployment for a Development Environment

To set up hot deployment for your development environment, you schedule the service that deploys a war file to regularly execute.

To set up hot deployment for a development environment

- Create a user task to regularly execute the wm.tomcat.admin:hotDeploy service. In a development environment when you will be updating web applications frequently, a good interval for Integration Server to execute this service is every 10 seconds. For information about how to create a user task, see *webMethods Integration Server Administrator's Guide*.

  Setting up the user task is not needed in a production environment because web applications should not be changing frequently. If the user task is not scheduled, you can manually invoke the wm.tomcat.admin:hotDeploy service.

Hot Deploying a War File

Hot deployment is accomplished by executing the wm.tomcat.admin:hotDeploy service. When this service executes, it checks the *Integration Server_directory*\instances *instance_name* \web\webapps directory.

- **When you place a new war file in the directory**, if an Integration Server package named the same as the war file does not exist, the hotDeploy service creates a new package. Be sure the name of your war file contains only characters that are valid for a package name; for more information, see *webMethods Service Development Help*. The hotDeploy service then unpacks the web application files to the \web directory of the Integration Server package for the web application.
- **When you place an updated war file in the directory**, the hotDeploy service unpacks the web application files from the updated war file into the \web directory of the Integration Server package, replacing all files in the \web directory that have the same name as files in the war file.
- **When you delete a war file from the directory**, the hotDeploy service takes no action. It does *not* delete the web application from the \web directory of the Integration Server package for the web application. The web application will still be available. If you want to delete the web application, see "Deleting Web Applications" on page 32.

To hot deploy a web application

1. If the Integration Server package for the web application already exists, ensure the name of the war file exactly matches the name of the package (including uppercase and lowercase letters). If the package does not already exist, the wm.tomcat.admin:hotDeploy service will create it during hot deployment.
2. If your web application uses tags from the webMethods tag library for JSP within your web application's JSP files to invoke Integration Server services and obtain pipeline data, be sure your web application has access to the tag library. Do the following:
   a. Copy the webm-taglib.tld file that is located in the *Integration Server_directory* \instances*instance_name* \web directory.
   b. If the Integration Server package for the web application already exists, place the copy in the *Integration Server_directory* \instances*instance_name* \packages *packageName* \web\WEB-INF directory of the package. If the Integration Server package does not exist, include the webm-taglib.tld file in your war file.
   c. Update the web.xml file for the web application (which should also be in the WEB-INF directory) to identify the location of the webMethods tag library (webm-taglib.tld).

      To update the web.xml file, add the following `<taglib>` section:

      ```
      <taglib>
           <taglib-uri>http://webm-taglib.tld</taglib-uri>
           <taglib-location>/WEB-INF/webm-taglib.tld</taglib-location>
      </taglib>
      ```
3. Place the war file for the web application in the *Integration Server_directory* \instances *instance_name* \web\webapps directory.

   Be sure to retain the appropriate directory structure for the JSPs and their supporting files, as described in "The Integration Server Package Directory Structure" on

page 26. In the war file, do not include the \web directory itself. Include only the files and subdirectories that reside beneath the \web directory.

4. Execute the wm.tomcat.admin:hotDeploy service.

   - **The wm.tomcat.admin:hotDeploy service is automatically executed** if you created a user task to have Integration Server regularly execute thewm.tomcat.admin:hotDeploy service. No further action is needed. After placing the war file in the *Integration Server_directory*\instances*instance_name* \web\webapps directory, the next time the wm.tomcat.admin:hotDeploy service executes, the hotDeploy service deploys the web application.
   - **Manually execute the wm.tomcat.admin:hotDeploy service** if you did not create a user task to execute the service. To manually execute the service do the following:

     i. From the Integration Server Administrator, in the **Packages** menu of the navigation panel, click **Management**.

     ii. In the list of packages, click **WmTomcat**.

     iii. Click **Browse services in WmTomcat**.

     iv. Click **wm.tomcat.admin:hotDeploy**.

     v. Click **Test hotDeploy**.

     vi. Click **Test (without inputs)**.

Placing the War File in the Package \web Directory

Important: Either deploy a web application as a single war file or as individual files (as described in "Placing Web Application Files in the Package \web Directory" on page 31. Do not mix individual files and war files within a package's web application directory structure.

To deploy by placing a war file into the \web directory of a package

1. Prepare the Integration Server package for the web application:

   a. If the Integration Server package does not exist, start Designer and create a new package. The package name does not have to match the name of the war file you will be using. For more information about creating packages, see *webMethods Service Development Help*.

   b. If the Integration Server package already exists and the package was created before Integration Server 6.0, create a \web directory beneath the package's root directory, as follows:

      Integration Server_directory\instances*instance_name* \packages *packageName* \web

   c. If you are using tags from the webMethods tag library for JSP within your web application's JSP files to invoke Integration Server services and obtain pipeline

data, see "Setting Up a Package so a Web Application Can Use the webMethods Tags for JSP" on page 19.

2. Copy the war file into the package's \web directory.

   Be sure to retain the appropriate directory structure for the JSPs and their supporting files, as described in "The Integration Server Package Directory Structure" on page 26. In the war file, do not include the \web directory itself. Include only the files and subdirectories that reside beneath the \web directory.

3. Reload the package. When you reload the package, WmTomcat unpacks the web application files the package's \web directory and deletes the war file.

Placing Web Application Files in the Package \web Directory

Important: Either deploy a web application as individual files or as a single war file (as described in "About Hot Deployment of the War File" on page 28 and "Placing the War File in the Package \web Directory" on page 30. Do not mix individual files and war files within a package's web application directory structure.

To deploy by placing web application files into the \web directory of a package

1. Prepare the Integration Server package for the web application:

   a. If the Integration Server package does not exist, start Designer and create a new package. The package name does not have to match the name of the JSP you will be using. For more information about creating packages, see *webMethods Service Development Help*.

   b. If the Integration Server package already exists and the package was created before Integration Server 6.0, create a \web directory beneath the package's root directory, as follows:

      Integration Server_directory\instances*instance_name* \packages *packageName* \web

   c. If you are using tags from the webMethods tag library for JSP within your web application's JSP files to invoke Integration Server services and obtain pipeline data, see "Setting Up a Package so a Web Application Can Use the webMethods Tags for JSP" on page 19.

2. Copy the web application files to the appropriate subdirectories beneath the package's \web directory. For example, to publish a web application in the Orders package, you would copy the files to the *Integration Server_directory* \instances *instance_name* \packages\Orders\web directory.

   Be sure to retain the appropriate directory structure for the JSPs and their supporting files, as described in "The Integration Server Package Directory Structure" on page 26.

3. Reload the package.

Undeploying Web Applications

To undeploy a web application, you can disable the Integration Server package in which the web application resides.

To undeploy a web application

1. From the Integration Server Administrator, in the **Packages** menu of the navigation panel, click **Management**.
2. Disable the package that contains the web application.

For more information about how to disable an Integration Server package, see *webMethods Integration Server Administrator's Guide*.

Note: If your Integration Server package contains services, IS document types, or other Integration Server elements that other applications or services use, rather than disable the package, you will need to:

1. Save a copy of the web application files (that is, the individual JSPs and supporting files) that are in the \web directory of the Integration Server package.
2. Delete the web application files from the \web directory of the package.
3. Reload the package.

When you want to make the web application available again, use the saved files and redeploy the web application files following the directions in "About Deploying Web Applications" on page 27.

Deleting Web Applications

To delete a web application, you can delete the Integration Server package in which the web application resides.

To delete a web application

1. From the Integration Server Administrator, in the **Packages** menu of the navigation panel, click **Management**.
2. Delete the package that contains the web application. When you delete the package, WmTomcat takes the following actions:
   - Stops and removes the Tomcat context that corresponds to the package
   - Unloads the package
   - Deletes the files in the following directories, where *PackageName* is the name of the package for the web application:

- *Integration Server_directory*\instances*instance_name* \web\work\WmTomcat \localhost\web_*PackageName* \
- *Integration Server_directory*\instances*instance_name* \web\conf\WmTomcat \localhost\web#*PackageName* .xml

- Deletes the following war file if it exists:

  Integration Server_directory\instances*instance_name* \web\webapps

For more information about how to delete an Integration Server package, see *webMethods Integration Server Administrator's Guide*.

Note: If your Integration Server package contains services, IS document types, or other Integration Server elements that other applications or services use, rather than delete the package, you will need to:

1. Delete the web application files from the \web directory of the package.
2. Reload the package.

About Invoking Web Applications

You can invoke a web application directly from a browser or from within an HTML form. Depending on how security was configured for the application, you might need to provide a user name and password to access the application.

Note: If you receive a message about your browser's language setting, right-click anywhere in the browser window, click **Encoding**, and then click **Unicode (UTF#8)**.

Invoking a JSP from a Browser

To invoke a web application from a browser, use the following URL format:

http://*hostName* :*portNum* /web/*packageName* /*fileName* .jsp

Parameter	Description
hostName	Host name or IP address of the Integration Server on which the JSP resides.
portNum	Port number on which Integration Server listens for HTTP requests. The default is 5555.
packageName	Name of the package to which the JSP belongs. *packageName* must match the package directory in which the JSP resides within *Integration Server_directory*\instances

Parameter	Description
	instance_name \packages on the server. If you do not specify a package name, the server looks for the JSP in the Default package. This parameter is case sensitive.
fileName .jsp	Name of the file containing the JSP. This file name must have a ".jsp" extension and it must reside within the \web directory under the package directory named in *packageName* . If the JSP resides in a subdirectory of \web, include the name of that subdirectory in the file name (see example below). This parameter is case sensitive. **Note:** If you are using a war file, supply the name of the JSP file here, not the name of the war file.

Examples

- The following URL retrieves showorders.jsp from a package named ORDER_TRAK on a server named rubicon:

  http://rubicon:5555/web/ORDER_TRAK/showorders.jsp

- The following URL retrieves showorders.jsp from the STATUS subdirectory in a package named ORDER_TRAK on a server named rubicon:

  http://rubicon:5555/web/ORDER_TRAK/STATUS/showorders.jsp

- The following URL retrieves showorders.jsp from the Default package on a server named rubicon:

  http://rubicon:5555/web/showorders.jsp

Note: If you provide a URL to a directory that does not contain a welcome file (for example, index.html), Integration Server displays a listing of the files in that directory. You can click a file to run the application associated with the file.

Invoking a JSP from within an HTML Form

Typing the JSP's URL on the address line in your browser is one way to invoke a web application. However, when you use JSPs to build a user interface, you will often invoke JSPs from HTML forms. For example:

```
<%@ page language="java" contentType="text/html" %>
<html>
<h1>Sample Form</h1>
<p>
<form name="edit" method="post" action="sampleFormAction.jsp">
Your Name
<input name="Name">
```

```
<input type=SUBMIT value="Go!">
</form>
</html>
```

In this example, the JSP called sampleFormAction.jsp executes when the user clicks **Go!**. For more information about accessing the form input variables, see the description of the `<webm:value>` tag in " webMethods Tag Library for JSP" on page 45.

Invoking a Servlet

To invoke a servlet, use the following URL format:

http://*hostName* :*portNum* /web/*packageName* /*servletName*

Parameter	Description
hostName	Host name or IP address of the Integration Server on which the servlet resides.
portNum	Port number on which Integration Server listens for HTTP requests. The default is 5555.
packageName	Name of the package to which the servlet belongs. *packageName* must match the package directory in which the servlet resides within *Integration Server_directory* \instances *instance_name* \packages on the server. If you do not specify a package name, the server looks for the servlet in the Default package. This parameter is case sensitive.
servletName	Servlet name.

For example, if a servlet is named ServiceServlet and resides in the ORDER_TRAK package on a server named rubicon, you would enter the following URL:

http://rubicon:5555/web/ORDER_TRAK/ServiceServlet

To see a sample servlet that illustrates how to invoke an Integration Server service, see "Invoking an Integration Server Service from a Servlet" on page 20.

About Testing the Web Application

Testing a web application consists of invoking the web application in a browser, troubleshooting errors, and editing and redeploying the application if necessary.

Troubleshooting Errors

Errors might arise at four different points in the development and execution of a web application:

- Compilation errors might occur as you develop the application.
- Errors might occur when you invoke the application.
- Run-time exceptions might occur when you invoke the application.
- Errors might occur when you reload the package containing the application.

Integration Server logs these errors to its server log. The WmTomcat package directs Tomcat internal messages and any servlet and JSP error messages to the Integration Server server log.

For complete information about Integration Server logs, see *webMethods Audit Logging Guide* and *webMethods Integration Server Administrator's Guide*.

Note: To minimize the number of servlet exceptions a web application will produce, be sure your system is set up properly. See "Before You Begin" on page 24 and "Preparing to Run Web Applications" on page 24 for more information.

Including Tomcat Information in the Server Log

To include Tomcat information in the server log

1. In Integration Server Administrator, go to the **Settings > Logging** page and click **Edit Logging Settings**.
2. The amount of information the server log contains depends on the logging level for the server log. The setting of the logging level determines whether a level of Tomcat message is or is not logged. In the **Level of Logging** list, click the amount of information you want Integration Server to record. The levels of Tomcat messages map to the logging levels for the server log as follows:

Tomcat Message Level	Integration Server Logging Level
Fatal	Fatal
Error	Error
Warning	Warn
Information	Info

Tomcat Message Level	Integration Server Logging Level
Debug	Debug
Trace	Trace

The default logging level for the server log is Info. When you are testing JSPs, you might want to set the logging level higher.

Note: If you set your logging level to Debug or higher, you might receive large numbers of Tomcat messages in your server log.

3. In the **Facilities** list, choose to include information for these two facilities:

Facility	Description
0072 Reporter	Details about the tags that Integration Server executes. The webMethods tag library for JSP is a thin wrapper of the Integration Server Reporter module. If errors result from the tags, you should first attempt to troubleshoot the Reporter module. To do so, you will need to raise the log level of the 0072 Reporter facility.
0100 Web Container	Information about Tomcat, WmTomcat, and your web application.

4. Click **Save Changes**.

Sending Messages Produced by the Jakarta commons-logging Sent to Integration Server

To have log messages produced by the Jakarta commons-logging sent to the Integration Server log

- Copy the commons-logging.properties file from *Integration Server_directory* \instances*instance_name* \packages\WmTomcat\resources into the following directory:

  Integration Server_directory\instances*instance_name* \packages *PackageName* \resources

  where *PackageName* is the name of the package that contains your web application that uses the Jakarta commons-logging API.

Viewing the Server and Error Logs

To view the server log and the error log

- In Integration Server Administrator, go to the **Logs > Server** page to view the server log.
- In Integration Server Administrator, go to the **Logs > Error** page to view the error log. In the error log, the system sets the **Service Name** column to the last service Integration Server invoked. If Integration Server has not invoked a service, the **Service Name** is NULL.

Editing and Redeploying a Web Application

Following are tips for editing and redeploying web applications in Integration Server packages.

When you...	You must...
Create a file in a package's \web subdirectory	Reload the package and then provide the appropriate URL in a browser to invoke the file.
Change a JSP, an HTML page, or an image or sound file	Reload the page's URL in the browser. Integration Server automatically recompiles the JSP and runs the resulting servlet.
Add a JSP or other resource file, class, or jar file	Provide the appropriate URL in a browser to invoke the file.
Change a package's deployment descriptor file	Reload the package.
Change the global deployment descriptor file	Reload the WmTomcat package.
Change the security constraints	Launch a new browser window to discard your previous credentials.
Replace a war file	To use hot deploy: 1. Place the war file in the *Integration Server_directory*\instances*instance_name*\web\webapps directory. 2. If you do not have a user task set up to automatically execute the

When you...	You must...
	wm.tomcat.admin:hotDeploy service, manually execute this service. For instructions, see the step about executing the wm.tomcat.admin:hotDeploy service in "Hot Deploying a War File" on page 28. To manually redeploy a war file into the \web directory of the web application's Integration Server package: 1. Delete the corresponding, unpacked war files from the \web directory of the web application's package. 2. Place the updated war file in the \web directory of the web application's package. 3. Reload the package.

About Securing the Web Application

Integration Server uses Access Control Lists (ACLs) and the global deployment descriptor file to authenticate and authorize access to all web applications deployed within the Integration Server environment. These security mechanisms allow you to make access to a web application as restrictive (for example, restrict access to only certain people) or as liberal (for example, allow access to anyone) as you need. By default, Integration Server uses the "Administrators" ACL.

Note: For more information about specifying JSP security constraints, see the Java Servlet standards.

The Global Deployment Descriptor File

Integration Server uses the global deployment descriptor file in *Integration Server_directory*\instances*instance_name* \web\conf\web.xml to specify default configuration settings for all web applications you deploy on Integration Server. These settings include:

- Definitions for the built-in servlets that serve all web applications, include servlets that handle requests for static resources, compile and execute JSPs, and process Server Side Include (SSI) directives and Common Gateway Include (CGI) scripts
- Default mappings for the servlets, JSP file name extensions, and MIME (content-type) properties
- Session timeout value

- Names of welcome files
- Security constraints

You can customize the global deployment descriptor file as needed. The changes you make to this file apply to all web applications on the current Integration Server instance.

You can also augment or override the defaults in the global deployment descriptor file for selected web applications. To do so, you update the deployment descriptor file for each application and include the appropriate security sections for the application. Integration Server processes the global deployment descriptor file first and then processes the application's deployment descriptor file.

Note: Modifying the global deployment descriptor file requires that you reload the WmTomcat package. Special conditions apply to UNIX platforms; for more information, see "Reloading the WmTomcat Package on UNIX Platforms" on page 22.

Recommendations for Setting Up Security in a Deployment Descriptor File

This section describes how you might want to set up your deployment descriptor file to secure web applications. Add two security sections to the deployment descriptor file:

- **A <security-constraint> section** to specify who is authorized to access the web application. For more information, see "Setting Up Web Application Authorization" on page 41.
- **The <login-config> section** validates a user's name and password when the user invokes the web application. By default, Integration Server displays a standard browser authentication screen. For more information, see "Setting Up Web Application Authentication" on page 44.

The following example shows a global deployment descriptor file that restricts access to all web applications that reside on the server to users belonging to the "Administrators" ACL. It also indicates that authentication should be performed using standard (basic) browser authentication.

Security sections in a deployment descriptor

Setting Up Web Application Authorization

Add a `<security-constraint>` section to the deployment descriptor to set up authorization of web applications. Security you set in the global deployment descriptor file applies to all web applications.

You can set security for all web applications by specifying security restraints in the global deployment descriptor file. You can override or augment the global security restraints for a particular web application by specifying different constraints in the deployment descriptor file for the package that is associated with the web application. The deployment descriptor file for a specific web application is located at *Integration Server_directory*\instances*instance_name* \packages\PackageName\web \WEB-INF\web.xml.

Important: Protection of web applications is as crucial as protecting DSPs and Integration Server services. If your web application does not provide its own authentication and authorization, you should add a `<security constraint>` section to either the global deployment descriptor file or to the web.xml file of the package containing the web application.

Some web applications provide their own authentication and require anonymous access to the application. In these cases, omit the `<security-constraint>` section from both the global deployment descriptor file and from the application's deployment descriptor file so that users are not prompted for a password. Omitting the `<security-constraint>` section entirely is not recommended except in these special circumstances. Ensure the other applications on your server are adequately protected and plan this approach carefully with your company's security advisors.

To set up web application authorization

1. If you want to set up global security for all web applications, do the following:

Note: If you do not set up global security, skip step 1. However, be sure the web.xml files in each package contain their own `<security-constraint>` sections. Doing so prevents unauthorized access to the web applications contained within them.

a. Create a `<security-constraint>` section in the global deployment descriptor file (*Integration Server_directory* \instances*instance_name* \web\conf\web.xml).

b. Define the `<security-constraint>` section as follows:

Change this sub-element...	To reflect the...
`<web-resource-name>`	Name for the collection of URLs to which you are restricting access. Use this name to reflect the type of security you are applying to the URLs deployed in the web application (for example, `Administrators` or `No Authentication`).
`<url-pattern>`	Pattern, or path, of the URLs to be protected by the role you specify in `<role-name>`.
`<role-name>`	Name of a valid Integration Server ACL whose members can access the URLs specified in `<url-pattern>`.

If you plan to override or augment your global security settings for some web applications, be careful how you set up your URL patterns. If the global web.xml file specifies a "superset" URL pattern (for example, /*), a package's web.xml file will not override this security constraint.

Examples:

- To protect all files in a web application by the same ACL, specify a URL pattern of /* and a role name that matches the ACL you want to use. In this example, all files in the web application are protected by the Administrators ACL:

```
    <security-constraint>
        <web-resource-collection>
            <web-resource-name>All
Restricted</web-resource-name>
            <url-pattern>/*</url-pattern>
        </web-resource-collection>
        <auth-constraint>
             <role-name>Administrators</role-name>
       </auth-constraint>
    </security-constraint>
```

- To protect selected files in the web application, specify multiple `<security-constraint>` sections. You can use multiple URL patterns and multiple role names in the same `<security-constraint>` section:

```
    <!-- Protect files in admin/* and private/* by Administrators ACL.-->
    <security-constraint>
        <web-resource-collection>
            <web-resource-name>Administrators</web-resource-name>
            <url-pattern>/admin/*</url-pattern>
            <url-pattern>/private/*</url-pattern>
        </web-resource-collection>
        <auth-constraint>
             <role-name>Administrators</role-name>
        </auth-constraint>
    </security-constraint>
    <!--Protect files in internal/* by Developers/Administrators ACLs.-->
    <security-constraint>
        <web-resource-collection>
            <web-resource-name>Internal</web-resource-name>
            <url-pattern>/internal/*</url-pattern>
        </web-resource-collection>
        <auth-constraint>
            <role-name>Developers</role-name>
            <role-name>Administrators</role-name>
        </auth-constraint>
    </security-constraint>
    <!-- For all other files, allow anyone to view                -->
    <!-- with no authentication                                   -->
    <security-constraint>
        <web-resource-collection>
            <web-resource-name>Public</web-resource-name>
            <url-pattern>/*</url-pattern>
        </web-resource-collection>
    </security-constraint>
```

c. Save and close the web.xml file.

d. Reload the WmTomcat package.

2. If you want to set up security for a specific web application, do the following:

a. Create a `<security-constraint>` section in the deployment descriptor file for the web application (*Integration Server_directory* \instances *instance_name* \packages\PackageName\web\WEB-INF\web.xml).

b. Define the `<security-constraint>` section. For more information about the tags to use, refer to the previous step.

c. Save and close the web.xml file.

d. Reload the package.

e. Repeat step 2 for each web application for which you want to override or augment global security.

3. Test your web applications to ensure the authorization is set properly. For more information, see "About Testing the Web Application" on page 35. Because your browser might cache previous credentials, you might have to close your browser and reopen it to reflect the new security settings.

Setting Up Web Application Authentication

By default, Integration Server uses a basic authentication setting that invokes a browser-based screen to prompt for user name and password. You can substitute the default authentication screen with your own HTML form-based login page for one or more web applications.

To set up web application authentication

- To specify an authentication method other than basic authentication, specify an authentication method of FORM in the `<login-config>` section of the web.xml file.

> **Note:** If the web application invokes services, Integration Server uses the credentials the user supplies to access the application to determine access to the invoked services.

A webMethods Tag Library for JSP

Overview

The webMethods tag library for JSP is a set of customized implementation classes. You use tags in this library in JSPs to import actions that let you call Integration Server services and retrieve data from the Integration Server pipeline.

The examples shown in this appendix assume a pipeline that looks as follows:

Key	Value
submitter	Mark Asante
shipNum	991015-00104
shipDate	10/15/99
carrier	UPS
serviceLevel	Ground
arrivalDate	10/18/99
items	

Key	Value
qty	10
stockNum	BK-XS160
description	Extreme Spline 160 Snowboard- Black
orderNum	GSG-99401088
status	Partial Order
qty	15
stockNum	WT-XS160
description	Extreme Spline 160 Snowboard- White
orderNum	GSG-99401088

Key	Value	
	status	Complete
supplierInfo	**Key**	**Value**
	companyName	Bitterroot Boards, LLC
	streetAddr1	1290 Antelope Drive
	streeAddr2	
	city	Missoula
	state	MT
	postalCode	59801
	supplierID	BRB-950817-001
	phoneNum	406-721-5000
	faxNum	406-721-5001
	email	Shipping@BitterrootBoards.com
buyerInfo	**Key**	**Value**
	companyName	Global Sporting Goods, Inc.
	accountNum	
	phoneNum	(216) 741-7566
	faxNum	(216) 741-7533
	streetAddr1	10211 Brookpark Road
	streetAddr2	
	city	Cleveland
	state	OH

Key	Value	
	postalCode	22130
	email	Receiving@GSG.com
backitems	SL-XS170 Extreme Spline 170 Snowboard- Silver	
	BL-KZ111 Kazoo 111 Junior Board- Blue	
	BL-KZ121 Kazoo 121 Junior Board- Blue	

webMethods Tag Library Summary

The following is a summary of the tags in the webMethods tag library. Each tag is described in more detail later in this appendix

Note: Tags are case sensitive. In your JSP, you must type them exactly as shown in this appendix (for example, `<webm:comment>`, not `<WEBM:COMMENT>` or `<Webm:Comment>`).

For information about how Integration Server identifies the tag prefix (webm), see "About Creating Web Applications" on page 18.

Use this tag...	To...
`<jsp:include>`	Insert a text file in the JSP. At run time, Integration Server inserts the contents of the specified file in the JSP and processes any tags the file contains. For more information, see " <jsp:include>" on page 50.
`<webm:comment>`	Add a comment to the JSP. For more information, see " <webm:comment>" on page 51.

`<webm:ifvar>`
`<webm:then>`
`<webm:else>`
	Execute a block of code in the JSP if a specified condition is met. The condition can specify either that a certain variable exists or has a particular value in the Integration Server pipeline. You can also define a different block of code to execute if the specified condition is not met. For more information, see " <webm:ifvar>" on page 51.

Use this tag...	To...
<webm:invoke> <webm:onError> <webm:onSuccess>	Call an Integration Server service. You can also define a block of code to execute if the service completes without errors and a block of code to execute if the service fails. Integration Server runs the specified service at run time and returns the results of the service to the JSP. For more information, see " <webm:invoke>" on page 53.
<webm:loop> <webm:loopsep>	Execute a block of code in the JSP once for each element in a specified Document, Document List, or String List in the Integration Server pipeline. You can insert a specified character sequence between results from a <webm:loop> tag in the HTML page produced from the JSP. For more information, see " <webm:loop>" on page 55.
<webm:nl>	Generate a new line character in the HTML page produced from the JSP. For more information, see " <webm:nl>" on page 58.
<webm:rename>	Rename or copy a variable in the Integration Server pipeline. For more information, see " <webm:rename>" on page 59.
<webm:scope>	Limit the variables in the Integration Server pipeline that are available to a specified block of code in the JSP. For more information, see " <webm:scope>" on page 60.
<webm:switch> <webm:case>	Execute one of multiple blocks of code, based on the value of a variable in the Integration Server pipeline. For more information, see " <webm:switch>" on page 62.
<webm:sysvar>	Insert a special variable or server property into the HTML page produced from the JSP. For more information, see " <webm:sysvar>" on page 64.
<webm:usePipeline>	Make the current Integration Server pipeline available in Java code as an IData variable

Use this tag...	To...
	called webm_pipe. For more information, see "<webm:usePipeline>" on page 65.
<webm:value>	Insert the values of one or more variables from the Integration Server pipeline into the HTML page produced from the JSP. For more information, see "<webm:value>" on page 65.

<jsp:include>

You use the <jsp:include> tag to insert a text file in the JSP. At run time, Integration Server inserts the contents of the specified file in the JSP and processes any tags the file contains.

Tip: If you use JSPs extensively, you might want to build a library of standard "code fragments" that you reference using <jsp:include> tags.

Syntax

For interpreted pages:

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

For static pages:

```
<%@ include file="file"%>
```

Arguments

Argument	Description
file.extension	Text file to insert. If the text file is not in the same directory as the JSP that references it, specify the path to the file relative to the JSP.

Examples

These examples insert the TMPL_ShipToBlock.html file into the JSP.

- The TMPL_ShipToBlock.html file resides in the same directory as the JSP.

```
<webm:scope recordName="buyerInfo">
<p>Shipped To:<br>
<jsp:include page="TMPL_ShipToBlock.html" flush="true"/>
</p>
</webm:scope>
```

- The TMPL_ShipToBlock.html file resides in a subdirectory named StandardJSPs within the directory in which the JSP resides.

```
<webm:scope recordName="buyerInfo">
<p>Shipped To:<br>
<jsp:include page="StandardJSPs/TMPL_ShipToBlock.html" flush="true"/>
</p>
</webm:scope>
```

- The TMPL_ShipToBlock.html file resides in the JSP's parent directory.

```
<webm:scope recordName="buyerInfo">
<p>Shipped To:<br>
<jsp:include page="../TMPL_ShipToBlock.html" flush="true"/>
</p>
</webm:scope>
```

<webm:comment>

You use the `<webm:comment>` tag to add a comment to the JSP.

Syntax

```
<webm:comment>comment</webm:comment>
```

Example

```
<webm:comment>
Use this JSP to generate an order list from any document that contains
a purchase item number, quantity, description, and PO number.
</webm:comment>
```

Note: You can also use the HTML comment syntax `<!-- comment -->` to include comments in JSP code.

<webm:ifvar>

You use the `<webm:ifvar>` tag to execute a block of code in the JSP if a specified condition is met. The condition can be either that a certain Integration Server pipeline variable exists or has a particular value.

You can also define a different block of code to execute if the specified condition is not met

Use `<webm:then>` to define the block of code to execute if the condition is met. Use `<webm:else>` to define the block of code to execute if the condition is not met.

Syntax

```
<webm:ifvar variable="variable" [isNull="true"] [notEmpty="true"]
      [equals="any_string"] [vequals="ref_variable"] [matches="regular_exp"]>
    <webm:then>block_of_code</webm:then>
    [<webm:else>block_of_code</webm:else>]</webm:ifvar>
```

Arguments

Argument	Description
variable	Pipeline variable to evaluate. **Example** `<webm:ifvar variable="carrier">`
`[isNull="true"]`	Specifies that the condition is true if the variable exists but its value is null. **Example** `<webm:ifvar variable="carrier" isNull="true">`
`[notEmpty="true"]`	Specifies that the condition is true if the variable contains one or more characters, and is false if the value is null. For string variables only. **Example** `<webm:ifvar variable="carrier" notEmpty="true">`
`[equals="`*`any_string`*`"]`	Specifies that the condition is true if the variable matches the string you specify in *any_string* . *any_string* is case sensitive. For example, FedEx does not match Fedex or FEDEX. **Example** `<webm:ifvar variable="carrier" equals="FedEx">`

`[vequals="`*`ref_variable`*`"]`	Specifies that the condition is true if the variable matches the value of the pipeline variable you specify in *ref_variable* .
	Example
	`<webm:ifvar variable="supplierInfo/state"`
	`vequals="buyerInfo/state">`

`[matches="`*`regular_exp`*`"]`	Specifies that the condition is true if the variable matches the regular expression *regular_exp* . **Example** In this example, the condition is true if the value of the pipeline variable carrier starts with UPS: `<webm:ifvar variable="carrier" matches="UPS*">`

Example

If a variable named *AuthCode* exists in the pipeline, this code inserts a success message into the HTML page produced from the JSP. The success message includes the current date and the value of the pipeline variable *AuthCode* . If the variable does not exist, the code inserts an error message.

```
<webm:ifvar variable="AuthCode">
    <webm:then>
         <p>Authorization code received <webm:sysvar variable="date">
         <webm:value variable="AuthCode"></p>
    </webm:then>
    <webm:else>
        <p>Error: Authorization code does not exist.</p>
    </webm:else>
</webm:ifvar>
```

<webm:invoke>

You use the `<webm:invoke>` tag to call an Integration Server service. You can also define a block of code to execute if the service completes without errors and a block of code to execute if the service fails. Integration Server runs the specified service at run time and returns the results of the service to the JSP.

Use the `<webm:onSuccess>` tag to define the block of code to execute if the service completes without errors. Use the `<webm:onError>` tag to define the block of code to execute if the service fails. Within the `<webm:onError>` tag block of code, you can use these pipeline variables:

Key	Description
errorClass	Exception name.
errorMessage	Error message text.
localizedMessage	Localized error message text that uses the client's locale.
errorService	Service that failed.
errorInputs	Inputs to the service that failed.
errorOutputs	Outputs of the service that failed. Could be null.

Syntax

```
<webm:invoke serviceName="service">
    <webm:onSuccess>block_of_code</webm:onSuccess>
    <webm:onError>block_of_code</webm:onError>
</webm:invoke>
```

Arguments

Argument	Description
service	Fully qualified name of the service to call.

Example

This code calls the Integration Server service orders:getShipInfo, which gets shipping information from a back-end system and builds an HTML form that allows the user to edit or cancel an order. If the service fails, error information is displayed.

```
<webm:invoke serviceName="orders:getShipInfo">
<h2>Shipping Details for Order  <webm:value variable="/oNum"/></h2>
<p>Date Shipped: <webm:value variable="shipDate"/><br>
Carrier: <webm:value variable="carrier"/>
<webm:value variable="serviceLevel"/></p>
<hr>
<webm:ifvar variable="shipDate" notempty="true">
<form action="http://rubicon:5555/orders/editShipInfo.dsp"
    method="get">
<p><b>Change this Shipment:</b></p>
<p><input type="RADIO" name="action" value="edit">
    Edit Shipment Details</p>
<p><input type="RADIO" name="action" value="cancel">
    Cancel this shipment</p>
    <input type="SUBMIT" value="Submit">
    <input type="HIDDEN" name="oNum"
    value="<webm:value variable='/oNum'/>"
</form>
    <hr>
</webm:ifvar>
<p><a href="http://rubicon:5555/orders/getorder.dsp
?action=showorder&oNum=<webm:value variable='/oNum'/>View Entire Order</a></p>
<webm:onError>
<hr>
<p><font color="#FF0000">Integration Server could not process
    your request because the following error occurred. Contact
    your server administrator.</font></p>
<table width="50%" border="1">
    <tr>
        <td><b>Service</b></td>
        <td><webm:value variable="errorService"/></td>
    </tr>
    <tr>
        <td><b>Error</b></td>
        <td><webm:value variable="errorMessage"/></td>
     </tr>
</table>
<hr>
</webm:onError>
</webm:invoke>
```

<webm:loop>

You use the <webm:loop> tag to execute a block of code once for each element in a specified Document, Document List, or String List in the Integration Server pipeline.

If you do not specify a Document, Document List, or String List to loop over, Integration Server will loop over the entire pipeline.

You can use the <webm:loopsep> tag to insert a specified character sequence between results from a <webm:loop> tag in the HTML page produced from the JSP. (<webm:loopsep> does not insert the character sequence after the result produced by the last iteration of the loop.)

At run time, Integration Server loops over the pipeline data as follows:

For this data structure...	Integration Server loops over…
Pipeline	Each key in the pipeline.
Document	Each key in the Document.
Document List	Each Document in the Document List.
String List	Each String in the String List.

Syntax

For the pipeline:

```
<webm:loop loopStruct="true" [excludePrivate="true"]>
block_of_code
    [<webm:loopsep sepString="separator_string"/>]
</webm:loop>
```

For a Document:

```
<webm:loop variable="loop_variable" loopStruct="true" [excludePrivate="true"]>
block_of_code
    [<webm:loopsep sepString="separator_string"/>]
</webm:loop>
```

For a Document List or String List:

```
<webm:loop variable="loop_variable" [loopStruct="true"]>
block_of_code
    [<webm:loopsep sepString="separator_string"/>]
</webm:loop>
```

Arguments

Argument	Description
`[variable="loop_variable"]`	Document, Document List, or String List to loop over. If you do not specify this option, Integration Server loops over the entire pipeline. **Example** `<webm:loop variable="items">`
`[loopStruct="true"]`	Tells Integration Server to loop over each key in the pipeline, Document, Document List, or String List. **Example** `<webm:loop variable="items" loopStruct="true">` If you want to dynamically access the names of the keys in the structure over which you are looping, use the predefined *$key* keyword. If you have nested loops and the outer loop is looping over a structure whose contents are dynamic, and you want the inner loop to loop over the keys in the current key of the outer loop, set variable in the inner loop to #$key keyword. You most often use this keyword to process a set of Documents contained within another Document. For an example of using these keywords, see the "Examples" section, below.
`[excludePrivate="true"]`	Tells Integration Server to skip keys whose names begin with $. **Example** `<webm:loop variable="items" excludePrivate="true">`

Examples

- This code puts shipping information for each Document in the Document List *items* in the HTML page produced from the JSP.

```
<p>This shipment contains the following items:</p>
<webm:loop variable="items">
    <p>
    <webm:value variable="qty"/>
    <webm:value variable="stockNum"/>
    <webm:value variable="description"/>
    <webm:value variable="status"/>
    </p>
```

```
  </webm:loop>
```

- This code lists each element in the String List *backItems* on separate lines in the HTML page and indicates that the elements are on backorder.

```
<p>The following items are backordered</p>
<p>
<webm:loop variable="backitems">
    <webm:value/><BR>
</webm:loop>
</p>
```

- This code lists the elements in the String List *backItems* with commas between the elements and indicates that the elements are on backorder.

```
<p>The following items are backordered</p>
<p>
<webm:loop variable="backitems">
    <webm:value/>
    <webm:loopsep sepString=","/>
</webm:loop>
</p>
```

- This code displays the names and values of the keys in the Integration Server pipeline in the HTML page.

```
<webm:loop loopStruct="true">
    <webm:value variable="$key"/><br>
    <webm:value/><br>
</webm:loop>
```

- This code loops over the keys in the Document named *Addresses*, whose contents are not known until run time, and inserts each name and its associated address into the HTML page produced from the JSP. The basic structure of the Document looks like this:

Key	Value	
Global Sporting Goods, Inc.	**Key**	**Value**
	streetAddr1	10211 Brookpark Road
	streetAddr2	
	city	Cleveland
	state	OH
	postalCode	22130
Fitness Shoes	**Key**	**Value**
	streetAddr1	2549 Oak Avenue

Key	Value	
	streetAddr2	
	city	Silver Spring
	state	MD
	postalCode	20905

.

.

.

Key	Value	
Yoga Wear	**Key**	**Value**
	streetAddr1	6666 Maple Street
	streetAddr2	Suite 100
	city	New York
	state	NY
	postalCode	11302

```
<p>Customer Addresses:</p>
<p>
<webm:loop variable="Addresses" loopStruct="true">
     <webm:value variable="$key"/>
     <webm:loop variable="#$key" loopStruct="true">
        <webm:value variable="streetAddr1"/>
        <webm:value variable="streetAddr2"/>
        <webm:value variable="city"/>
        <webm:value variable="state"/>
        <webm:value variable="postalCode"/>
    </webm:loop>
</webm:loop>
</p>
```

<webm:nl>

You use the `<webm:nl>` tag to generate a new line character in the HTML page produced from the JSP.

Use this tag when you want to preserve the ending of a line that ends in a tag. The tag does not insert the HTML line break
 code. It merely inserts a line break character, which Integration Server treats as white space. If you do not explicitly insert this tag on such lines, Integration Server drops the new line character following that tag.

Syntax

```
<webm:nl/>
```

Example

This code inserts the values of the variables *carrier* , *serviceLevel* , and *arrivalDate* on separate lines in the HTML page.

```
<hr>
<p>Shipping Info:
<webm:value variable="carrier"/><webm:nl/>
<webm:value variable="serviceLevel"/><webm:nl/>
<webm:value variable="arrivalDate"/><webm:nl/></p>
<hr>
```

<webm:rename>

You use the <webm:rename> tag to rename or copy a variable in the Integration Server pipeline.

Syntax

```
<webm:rename sourceVar="source_variable" targetVar="target_variable"
 [copy="true"]/>
```

Arguments

Argument	Description
source_variable	Pipeline variable to rename or copy. *source_variable* can reside in any existing scope or Document.
target_variable	Pipeline variable to which to rename or copy *source_variable* . *target_variable* must be in the current scope. If *target_variable* does not exist, Integration Server creates it. If *target_variable* already exists, Integration Server overwrites it.
[copy="true"]	Tells Integration Server to copy *source_variable* to *target_variable* . By default, Integration Server renames *source_variable* to *target_variable* , by copying *source_variable* to *target_variable* and then deleting *source_variable* .

Example

This code renames the pipeline variable *state* in the Document *buyerinfo* to *ST*.

```
<webm:scope recordName="buyerInfo">
    <webm:rename sourceVar="state" targetVar="ST"/>
    <jsp:include page="TMPL_ShipToBlock.html" flush="true"/>
</webm:scope>
```

<webm:scope>

You use the `<webm:scope>` tag to limit the variables in the Integration Server pipeline that are available to a specified block of code in the JSP.

The specified scope remains in effect until Integration Server encounters the next `</webm:scope>` tag.

Syntax

```
<webm:scope [recordName="document"]
[options="param(name='value')
param(stringList[]='value1', 'value2',...'valuen')
rparam(document={name1='value1';name2='value2';...namen='valuen'})
rparam(documentList[]={name1='value1';name2='value2';...namen='valuen'}|
{name1='value1';name2='value2';...namen='valuen'})"]>  block_of_code
</webm:scope>
```

Arguments

You can use any of the following options with this tag. If you specify multiple options, use a space to separate the options.

Important: If you set the value of a variable with these options, the value you specify will replace the current value of that variable.

If the value of a variable contains spaces, enclose the value within single quotes.

When you use the `<webm:scope>` tag in a JSP, the entire tag must appear on one line.

Argument Description

- Specifies the document to use as the current scope.

  ```
  [recordName="document"]
  ```

- Adds to the current scope a String named *name* whose value is *value*.

  ```
  [options="param(name='value')"]
  ```

  Example

  ```
  <webm:scope options="param(csClass=Gold) param(csName='Joe Smith')"/>
  ```

- Adds to the current scope a String List named *stringList* whose values are *value1 value2 ...valuen* .

```
[options="param(stringList[]='value1', 'value2',...
'valuen')"]
```

Example

```
<webm:scope options="param(shipPoints[]=BOS,LAX,NYC,PHL)
param(warehouseLoc[]='Los Angeles',Philadelphia)"/>
```

- Adds to the current scope a Document named document whose variables are name and whose values are *value* .

```
[options="rparam(document={name1='value1';name2='value2';...
 namen='valuen'})"]
```

Example

```
<webm:scope options="rparam(custServiceInfo={csClass=Gold;
csPhone='800-444-2300';csRep='Ann Johnson'})
rparam(buyerInfo={buyerName='Joe Smith';buyerPhone='800-333-1234'})"/>
```

- Adds to the current scope a Document List named *documentList* whose variables are *name* and whose values are *value* .

```
[options="rparam(documentList []={name1='value1'; name2='value2';...
 namen='valuen'}|{name1='value1';name2='value2';...namen='valuen'})"]
```

Example

```
<webm:scope options="rparam(custServiceCtrs[]=
{csName=Memphis;csPhone='800-444-2300'}|
{csName=Troy;csPhone='800-444-3300'}|
{csName=Austin;csPhone='800-444-4300'})
rparam(custServiceReps[]={csRep='Ann Johnson';csExt=27}|
{csRep='John Jones';csExt=28}|
{csRep='Chris Smith';csExt=29})"/>
```

Examples

- This code sets the scope to the Document *buyerInfo* and inserts the information in the scope into the HTML page produced by the JSP.

```
<webm:scope recordName="buyerInfo">
    <p>Shipped To:<br>
        <webm:value variable="companyName"/><br>
        <webm:value variable="streetAddr1"/><br>
        <webm:value variable="streetAddr2"/><br>
        <webm:value variable="city"/>
        <webm:value variable="state"/>
        <webm:value variable="postalCode"/>
    </p>
</webm:scope>
```

- This code sets the scope to the Document *buyerInfo* , adds variables named *buyerClass* and *shipPoint* to the scope, and inserts the information in the scope into the HTML page produced by the JSP.

```
<webm:scope recordName="buyerInfo"
 options="param(buyerClass=Gold) param(shipPoint='BWI Hub')">
    <p>Shipped To:<br>
        <webm:value variable="companyName"/><br>
        <webm:value variable="streetAddr1"/><br>
```

```
        <webm:value variable="streetAddr2"/><br>
        <webm:value variable="city"/>
        <webm:value variable="state"/>
        <webm:value variable="postalCode"/>
    </p>
    <hr>
    <p>Point of Departure: <webm:value variable="shipPoint"/><br>
    Customer Class: <webm:value variable="buyerClass"/></p>
</webm:scope>
```

- This code sets the scope to the Document *buyerInfo* , adds variables named *buyerClass* and *shipPoint* from a Document named *shipInfo* to the scope, and inserts the information in the scope into the HTML page produced by the JSP.

```
<webm:scope recordName="buyerInfo"
options="rparam(shipInfo={buyerClass=Gold;shipPoint='BWI Hub'})">
    <p>Shipped To:<br>
        <webm:value variable="companyName"/><br>
        <webm:value variable="streetAddr1"/><br>
        <webm:value variable="streetAddr2"/><br>
        <webm:value variable="city"/>
        <webm:value variable="state"/>
        <webm:value variable="postalCode"/>
    </p>
    <hr>
    <p>Point of Departure: <webm:value
    variable="shipInfo/shipPoint"/><br>
    Customer Class: <webm:value variable="shipInfo/buyerClass"/></p>
</webm:scope>
```

- This code sets the scope to the Document *buyerInfo* , adds a variable named *shipPoints* to the scope, adds variables named *name* and *ssid* from a Document List called *custInfo* , and inserts the information in the scope into the HTML page produced by the JSP.

```
<webm:scope recordName="buyerInfo"
options="param(shipPoints[]=DC,VA,LA,MD)
rparam(custInfo[]={name=Joe;ssid=ssid1}|{name=John;ssid=ssid2})">
<h3>Ship Points: </h3>
    <webm:loop variable="shipPoints">
    <webm:value /><br>
    </webm:loop>
<h3>Customers: </h3>
    <webm:loop variable="custInfo">
    <webm:value variable="name" />, <webm:value variable="ssid" />   <br>
    </webm:loop>
</webm:scope>
```

<webm:switch>

You use the `<webm:switch>` tag to execute one of multiple blocks of code, based on the value of a variable in the Integration Server pipeline.

Use `<webm:case>` to define each value and the associated block of code to execute. Integration Server evaluates `<webm:case>` tags in the order they appear in the JSP. When a case is true, Integration Server executes the associated block of code, then exits the `<webm:switch>` structure.

If you want to define a default case (that is, the case to execute if the specified variable does not exist or if none of the other cases is true), specify a `<webm:case>` tag with no *switch_value* . The default case must be the last `<webm:case>` tag in the `<webm:switch>` tag.

Syntax

```
<webm:switch variable="variable">
    <webm:case value="switch_value1">block_of_code</webm:case>
    [<webm:case value="switch_value2">block_of_code</webm:case>...
    <webm:case value="switch_valuen">block_of_code</webm:case>]
    [<webm:case>block_of_code</webm:case>]
</webm:switch>
```

Arguments

Argument	Description
variable	Pipeline variable whose value to evaluate.
switch_value	Value of *variable* that triggers Integration Server to execute the associated block of code. *switch_value* must match the value of *variable* exactly. *switch_value* is case sensitive. For example, FedEx does not match Fedex or FEDEX.

Examples

- This code inserts different paragraphs into the HTML page produced from the JSP, based on the value in the pipeline variable *carrier* .

```
<webm:switch variable="carrier">
    <webm:case value="FedEx">
        <p>Shipped via Federal Express <webm:value="serviceLevel"/>
        <webm:value="trackNum"/></p>
    </webm:case>
    <webm:case value="UPS">
        <p>Shipped via UPS <webm:value="serviceLevel"/></p>
    </webm:case>
    <webm:case value="Freight">
        <p>Shipped via <webm:value="transCompany"/><br>
           FOB: <webm:value="buyerInfo/streetAddr1"/><br>
          <webm:value="buyerInfo/streetAddr2"/><br>
          <webm:value="city"/>, <webm:value="state"/>
          <webm:value="postalCode"/></p>
    </webm:case>
</webm:switch>
```

- This code calls different Integration Server services based on the value of the pipeline variable *action* .

```
<html>
    <head>
    <title>Order Tracking System</title>
    </head>
    <body bgcolor="#FFFFCC">
    <h1>Order Tracking System</h1>
```

```
    <hr>
    <webm:switch variable="action">
        <webm:case value="shipinfo"/>
            <webm:invoke serviceName="orders:getShipInfo"/>
            .
            .
            .
        <webm:case value="showorder"/>
            <webm:invoke serviceName="orders:orders:getOrderInfo"/>
            .
            .
            .
            .
        <webm:case value="showinvoice"/>
            <webm:invoke serviceName="orders:orders:getInvoices"/>
            .
            .
            .
    </webm:switch>
    <hr>
    <jsp:include page="stdFooter.txt" flush="true"/>
    </body>
</html>
```

<webm:sysvar>

You use the `<webm:sysvar>` tag to insert a special variable or server property into the HTML page produced from the JSP.

Syntax

```
<webm:sysvar variable="system_variable"/>
```

Arguments

Argument	Description
system_variable	System variable or property to insert, as follows:

Value	Description
`host`	Name of the Integration Server that processed the JSP.
`date`	Current date, in the format "Weekday Month Day HH:MM:SS Locale Year" (for example, Fri Aug 12 04:15:30 Pacific 1999).
`property(property)`	Current value of the Integration Server property *property* (for example,

Argument	Description
	watt.server.port) or any Java system property (for example, java.home). See *webMethods Integration Server Administrator's Guide* for a list of Integration Server properties.

Examples

- This code inserts the name of the Integration Server that processed the JSP into the HTML page.

```
Response generated by host <webm:sysvar variable="host"/>
```

- This code inserts the value of the Integration Server property watt.server.port, which identifies Integration Server's main HTTP listening port, into the HTML page:

```
<p>
<webm:sysvar variable="host"/> was listening on
<webm:sysvar variable="property(watt.server.port)"/>
</p>
```

<webm:usePipeline>

You use the `<webm:usePipeline>` tag to make the current Integration Server pipeline available to the JSP in Java code as an IData variable named *webm_pipe* .

Syntax

```
<webm:usePipeline>block_of_code</webm:usePipeline>
```

Example

This code prints the contents of the current pipeline in one long string to the HTML page produced from the JSP.

```
<webm:usePipeline>
<% System.out.println ("pipeline is:" + webm_pipe.toString()); %>
</webm:usePipeline>
```

<webm:value>

You use the `<webm:value>` tag to insert the values of one or more variables from the Integration Server pipeline into the HTML page produced from the JSP. You can also use the tag within a loop to insert the value of the loop's current key. To do so, specify the tag without any arguments (that is, `<webm:value/>`).

Syntax

Separate nested fields with a forward slash (/).

```
<webm:value [variable="variable[/subvariable1/subvariable2/...
/subvariablen]"] [null="any_string"] [empty="any_string"] [index="n"]
[encode="code"] [decode="code"] [decimalShift="X"] [decimalShow="Y"]/>
```

Arguments

Argument	Description
[variable="*variable* [/*subvariable1* / *subvariable2* /... /*subvariable3*]"]	Pipeline variable whose value to insert. Integration Server retrieves *variable* from the current scope. If you do not specify *variable* and you are inside a loop, Integration Server returns the value of the current key. To select a variable outside the current scope, use this syntax: **To do this...** / **Use this syntax...** Insert the value of *variable* from the initial scope: */variable* Insert the value of *variable* from the parent of the current scope: *../variable* Insert the value of *subvariablen* from *variable*: *variable* /subvariable1/ *subvariable2* /.../ *subvariable3* If you do not specify *variablen* , Integration Server assumes the current element within the current scope.
`[null="any_string"]`	When *variablen* is null, inserts *any_string* . **Example** `<webm:value variable="carrier" null="No Carrier Assigned"/>`
`[empty="any_string"]`	When *variablen* contains an empty string, inserts *any_string* . **Example** `<webm:value variable="description" empty="Description Not Found"/>`
`[index="n"]`	Inserts the nth element of the Document List or String List specified by *variable* .

Argument	Description
	Example `<webm:value variable="backItems" index="1"/>`
`[encode="code"]`	Encodes the contents of *variable* before inserting it, where *code* specifies the encoding system to apply, as follows: **To encode the value using... / Set code to...** XML encoding — xml Base-64 encoding — b64 URL encoding — url
`[decode="code"]`	Decodes the contents of *variable* before inserting it, where *code* specifies the decoding system to apply, as follows: **To encode the value using... / Set code to...** Base-64 encoding — b64 URL encoding — url
`[decimalShift="X"]`	Shifts the decimal point in the value of *variable* to the right *X* positions before inserting the value. For decimal values only.
`[decimalShow="Y"]`	Truncates the value of *variable* to *Y* positions after the decimal before inserting the value. For decimal values only.

Examples

- This code calls the orders:getOrderInfo service and inserts the results of the service into the HTML page produced from the JSP.

```
<webm:invoke serviceName="orders:getOrderInfo"><br>
<p><webm:value variable="buyerInfo/companyName"/><br>
<webm:value variable="buyerInfo/acctNum"/>
    <p>This shipment contains the following items</p>
    <table width="90%" border="1">
        <tr>
            <td>Number</td><td>Qty</td>
            <td>Description</td><td>Status</td>
```

```
        </tr>
        <tr>
        <webm:loop variable="items">
            <td><webm:value variable="stockNum"/></td>
            <td><webm:value variable="qty"/></td>
            <td><webm:value variable="description"/></td>
            <td><webm:value variable="status"/></td>
        </tr>
        </webm:loop>
    </table>
</webm:invoke>
```

- This code loops over the pipeline and inserts the names and values of the keys in the pipeline into the HTML page.

```
<webm:loop loopStruct="true">
    <webm:value variable="$key"/><br>
    <webm:value/><br>
</webm:loop>
```

- This code inserts the contents of the variable carrier into the HTML page. If carrier is null or empty, the code inserts the string “UPS”.

```
<webm:value variable="carrier" null="UPS" empty="UPS"/>
```

DSP Equivalents

The webMethods tags for JSP that Integration Server supports are similar to webMethods DSP tags. The following table lists the tags in the webMethods tag library and their DSP equivalents. For more information about these tags and their arguments and options, see *Dynamic Server Pages and Output Templates Developer's Guide*.

webMethods Tag for JSP	Equivalent DSP Tag
include	include
webm:case	case
webm:comment	comment
webm:else	else
webm:ifvar	ifvar
webm:invoke	invoke
webm:loop	loop
webm:loopsep	loopsep

webMethods Tag for JSP	Equivalent DSP Tag
webm:nl	nl
webm:onError	onerror
webm:onSuccess	None
webm:rename	rename
webm:scope	scope
webm:switch	switch
webm:sysvar	sysvar
webm:then	None. DSPs execute the block of code immediately following the ifvar tag.
webm:usePipeline	None. By default, the pipeline is in the scope of every service invocation.
webm:value	value

Index

Symbols