Development -- table of contents

JSP files

XML

Web applications

Various topics

Personalization

Pervasive computing

4.5: Employing pervasive computing

Samples

IBM WebSphere Application Server

4: Developing applications

For IBM WebSphere Application Server, applications are combinations of building blocks that work together to perform a business logic function.

Web applications are groups of one or more servlets, plus static content.

Web applications = servlets + JSP files + XML files + HTML files + graphics

WebSphere Application Server's programming model is based on Sun's Java™2 Platform, Standard Edition (J2SE) software. The J2SE environment provides the basis for building network-centric enetprise applications that run on a variety of systems. The J2SE software consists of the Java™2 SDK, Standard Edition and theJava™2 Runtime Environment (JRE), Standard Edition.

The J2SE environment provides the foundation for the J2EE model, whichpackages enterprise and Web applications into new categories of Web ArchiveResource files or WAR files and Enterprise Archive Resource files or EAR files.The J2EE model is fully implemented in WebSphere Application Server version 4.0.See the product sitefor more information.

View the supported specification levels for servlet, JSP, and EJB APIs at theWebSphere Application Server prerequisites Web site.

See article 4.1 to review the WebSphere application programming model and environment, including information on various tools to help you develop and testyour application components.

Consult sections 4.2 and 4.3 for a focus on developing Web applications

View the Related information links to help you bringthese building blocks together, adding personalization, and other features.

4.1: Programming model and environment

IBM WebSphere Application Server supports a three-tier programmingmodel in which the application server and its contents -- your applications -- reside in the middle tier.

In this multi-tiered programmingmodel, tier 0 represents Applets which run in a Web browser; tier 1, some application resources such as JSP filesand servlets, which respond to HTTP requests; tier 2, the enterprise beans that run on the EJB server; and tier 3, the databases that store thebusiness data.

This documentation is geared towards the following layered approachto application development:

1. Determine what the application should do
2. Plan the application building blocks and their interactions
3. Create the Web application building blocks
4. Combine them into a Web application with the sought features
5. Combine Web applications into enterprise applications

Application developers might specialize in areassuch as data access, Java programming, and Web page design.The layered approach provides a model allowing these programmers to collaboratein designing, implementing, deploying, and maintaining applications withmaximum efficiency.

4.1.1: Finding supported APIs and specifications

Finding supported specification levels

See the WebSphere Application Server prerequisites Web page for the supported levels of specifications such as the Java Servlet and JavaServer Pages (JSP) specifications from Sun Microsystems.

Refer to the Sun Microsystems Web site for additional information about Java specifications:

`http://java.sun.com/products`

Finding API documentation (Javadoc) pertaining to IBM WebSphere Application Server

Access the Javadoc index for the packages included with IBM WebSphere Application Server (though not necessarily produced by IBM) from the fullInfoCenter:

Index to API documentation (Javadoc)

4.1.1.1: Supported programming languages

WebSphere Application Server is designed and tested to support applications and clients based on the **Java** programming language and technologies.

The IBM WebSphere Application Server Enterprise Edition is the **recommended** solution for environments requiring C and C++ clients. The Enterprise Editionsupports CORBA, COM, and DCOM clients in addition to the Javaand browser clients supported by the Advanced and Standard Editions.

4.1.1.2: Supported XML/XSL APIs and specifications

IBM WebSphere Application Server provides document parsers, document validators, and document generators for server-side XML processing. The product supports the following XML-related recommendations:

- W3C Extensible Markup Language (XML) 1.0
- W3C Namespaces in XML (Recommendation January 14, 1999)
- W3C Level 1 Document Object Model Specification (DOM) 1.0 (Recommendation October 1, 1998)
- XSL Transformations Version 1.0
- XML Path Language Version 1.0

IBM WebSphere Application Server supports the following XML/XSL APIs:

- XML4J Version 2.0.15
- LotusXSL Version 1.0.1 or Xalan Version 1.1

XML parsing and validation support

The components of XML for Java provide support for parsing, validating, and generating XML data. The processor implements the base XML, namespace, and DOM W3C recommendations and SAX *de facto* standard. For more information, see the product Javadoc.

xml4j.jarcan be found in the product_installation_root\lib directory.

To obtain updates and source code for XML4J and other XML-related resources, visit the IBM alphaWorks site athttp://alphaworks.ibm.com/.

XSL processing support

This includes APIs for formatting and transforming XML documents at the server.

lotusxsl.jarand its open-source version, xalan.jar,can be found in the *product_installation_root*\lib directory.

To obtain updates and source code for LotusXSL, visit the IBM alphaWorks site at the URL provided previously.

4.1.2: Tools for developing Web applications

When you install IBM WebSphere Application Server from the product CD, the installation program provides options to install IBM Distributed Debugger (DD) andObject Level Trace (OLT).

In addition, the following products can help you develop components for Web applications:

- IBM VisualAge for Java, Enterprise Edition
- IBM WebSphere Studio

These products are available separately.

4.1.2.1: IBM Distributed Debugger and Object Level Trace

The IBM Distributed Debugger (DD) enables you to detect and diagnose errors in your code.Its client/server design enables you to debug programs over a network connection.You can also debug programs running on your local workstation.

Object Level Trace (OLT), which works closely with the IBM Distributed Debugger, enables you to monitor the flow of a distributed application and debug code from a single workstation.

Tips for using OLT/DD

In order to trace and debug the application server,you must install the debugger on the machine on which the application server is running. For remote tracing and debugging, you must also install the debugger on the machine from which you plan to run the OLT tool and the debugger. For example,only remote debugging is supported on Solaris, soif your application server is running on Solaris, you must install the Solaris component of the debugger on that same machine.In addition, you must install OLT and the debugger on the AIX or Windows NT (or Windows 2000) machine from which you plan to run the tools remotely.

For the latest information about OLT/DD, see the IBM Distributed Debuggerand OLTdocumentation.

4.1.2.2: IBM VisualAge for Java

VisualAge for Java Enterprise Edition provides the following tools for developing Web application components:

- JSP Execution Monitor - Enables you to monitor the execution of JSP source code, generated servlets, and HTML source code as it is generated. This tool is available for Windows NT systems.
- Servlet Launcher - Enables you to start a Web server, open your Web browser, and launch a servlet. This tool is available for AIX and Windows NT systems.
- WebSphere Test Environment - Enables you to test deployment of Web application components without a full-fledgedWebSphere Application Server installation.You can set breakpoints within servlet code, dynamically update the servlet at breakpoints, and continue running the servlet with the changes incorporated. These tasks can be performed without restarting the servlet.

For more information about this product, visit the following Web site:

http://www.ibm.com/software/ad/vajava/

More about the WebSphere Test Environment

IBM VisualAge for Java provides a subset of the WebSphere Application Serverrun-time environment in a component called the *WebSphere Test Environment* (WTE).The WebSphere Test Environment offers the following:

- A lightweight run-time environment with no dependency on WebSphere Application Server availability
- No dependency on an external database unlessentity bean support is required

As a subset of the WebSphere Application Server,the WTE does *not* offer certainfeatures that the application server product does, as follows:

- Secure Socket Layer (SSL) and secure HTTP (HTTPS).
- HTTP-style user ID/password authentication challenge.
- Administrative server and services.
- The XMLConfig tool. Older XML grammar is used in the WTE configuration.
- Personalization APIs
- Security context and API for enterprise beans.
- Security APIs for servlet sessions, or other security classestypically involved in sign-on, authentication, or authorization.
- Support for running multiple Web applicationsin addition to the default Web application

Tips for using VisualAge for Java

When you are ready to move from the WTE to deployment on the WebSphere Application Server, verify that application class paths are properly set in the new environment.

4.1.2.3: IBM WebSphere Studio

IBM WebSphere Studio Professional Edition offers the following features:

- Create Web applications for various devices, such as voice browsers and handheld devices.
- Select from two Web application models - Servlet or JSP.
- Close integration with IBM VisualAge for Java.
- Graphical display of the links between files in a project.
- Automatic updating of links whenever your files are changed or moved.
- Wizards that jump-start creation of dynamic pages that use databases and Java beans.Use the wizard output as is or tailor it to your needs.
- An import feature to quickly transfer existing Web site content into a Studio project.
- Staging and publishing your project to different (and to multiple) servers.
- The ability to archive a Web site into a single compressed file.
- Full-function visual editing of HTML and JSP files.
- Companion tools:
    - AnimatedGif Designer, for building GIF animations
    - Applet Designer, a visual authoring tool for building Java applets
    - WebArt Designer, for creating buttons, masthead images, and other graphics

For more information about this product, visit the following Web site:

`http://www.ibm.com/software/webservers/studio/index.html`

Tips for using WebSphere Studio

WebSphere Studio providesthe com.ibm.servlet.PageListServlet class to call JSP files.Servlets generated by the WebSphere Studio wizardsare subclasses of this class.Such a servlet must have an associated servlet configuration file (.servlet)that specifies all JSP files that the servlet might call.For more information, see*Servlet and JSP Programming with IBM WebSphere Studioand VisualAge for Java* (SG24-5755),available from the IBM Redbooks Web site.

4.2: Building Web applications

Different types of Web applications exist, ranging from static document Web sitesto database-backed systems.Some Web applications are front ends to traditional, non-Web applications.

This section provides considerations, instructions, and tips forcreating the building blocks that comprise Web applications.

View article 6.6.8: Administering Web applications for information on configuring such Web application settings as:

- Classpaths
- Web paths
- Welcome pages
- Servlet filtering parameters
- Context attributes

4.2.1: Developing servlets

Servlets are Java programs that build dynamic client responses, such as Web pages.Servlets receive and respond to requests from Web clients, usually across HTTP, the HyperText Transfer Protocol.

Because servlets are written in Java, they can be ported without modification to different operating systems.Servlets are more efficient than CGI programs because, unlike CGI programs, servlets are loaded into memory once, and each request is handled by a Java virtual machine thread, not an operating system process.Moreover, servlets are scalable, providing support for a multi-application server configuration.Servlets also allow you to cache data, access database information, and share data with other servlets, JSP files and (in some environments) enterprise beans.

Servlet coding fundamentals

In order to create an HTTP servlet, you should extend the `javax.servlet.HttpServlet` class and override any methods that you wish to implement in the servlet. For example, a servlet would override the `doGet` method to handle GET requests from clients.

For more information on the `HttpServlet` class and methods, review articles:

- 4.2.1.3.1: Creating HTTP Servlets
- 4.2.1.3.1.1: Overriding `HttpServlet` methods
- 4.2.1.3.2: Inter-servlet communication

The `doGet` and `doPost` methods take two arguments:

- HttpServletRequest
- HttpServletResponse

The `HttpServletRequest` represents a client's requests. This object gives a servlet access to incoming information such as HTML form data, HTTP request headers, and the like.

The `HttpServletResponse` represents the servlet's response.The servlet uses this object to return data to the client such as HTTP errors (200, 404, and others), response headers (Content-Type, Set-Cookie, and others), and output data by writing to the response's output stream or output writer.

Since `doGet` and `doPost` throw two exceptions (`javax.servlet.ServletException` and `java.io.IOException`), you must include them in the declaration. You must also import classes in the following packages:

Package names	Functions/Objects
`java.io`	PrintWriter
`javax.servlet`	`HttpServlet`
`javax.servlet.http`	`HttpServletRequest` and `HttpServletResponse`

Note: When creating your servlets, do not use the followingreserved words for the class name:

- Description
- Code
- LoadAtStartup
- UserServlet
- DebugMode
- Enabled

Some reserved words such as *UserServlet* can be used in the package names butcreate problems when used as class names.

The beginning of your servlet might look like the following example:

```
import java.io.*;import javax.servlet.*;import javax.servlet.http.*;import java.util.*;public class
MyServlet extends HttpServlet {  public void doGet(HttpServletRequest request,
HttpServletResponse response)      throws ServletException, IOException {
```

After you create your servlet, you must:

1. Compile your servlet using the `javac` command, as for example:
   ```
   javac MyServlet.java
   ```
2. Invoke your servlet using one of the methods described in article:
   4.2.4.4: Providing ways for clients to invoke applications

You can also compile your servlet using the `-classpath` option on the `javac` compiler. To access the classes that were extended, reference the`servlet.jar` file in the `<WAS_root>\lib` directory.Using this method, you issue the following command to compile your servlet:

```
javac -classpath C:\<WAS_root>\lib\servlet.jar MyServlet.java
```

Now that you successfully created, compiled, and tested your servlet on your local machine, you must install it in the WebSphere Application Server runtime. View article 6: Administer applicationsfor this information.

Servlet lifecycle

The javax.servlet.http.HttpServlet class defines methods to:

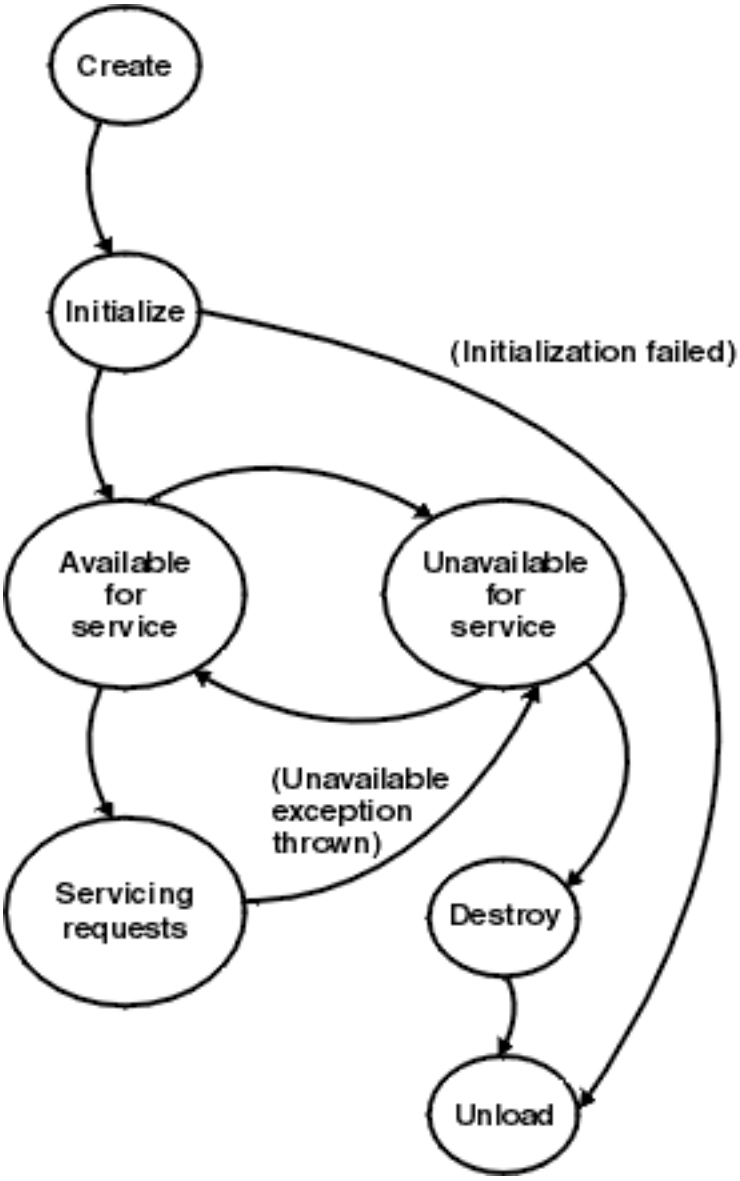
- Initialize a servlet
- Service requests
- Remove a servlet from the server

These are known as life-cycle methods and are called in the following sequence:

1. The servlet is constructed
2. It is initialized with the init method
3. Calls from clients to the service method are handled
4. The servlet is taken out of service
5. It is destroyed with the destroy method
6. The servlet is finalized and the garbage is collected.

Review article 4.2.1.1 for more life cycle information.

4.2.1.1: Servlet lifecycle



Instantiation and initialization

The servlet engine (the Application Server function that processes servlets, JSP files, and other types of server-side include coding) creates an instance of the servlet. The servlet engine creates the servlet configuration object and uses it to pass the servlet initialization parameters to the init method. The initialization parameters persist until the servlet is destroyed and are applied to all invocations of that servlet until the servlet is destroyed.

If the initialization is successful, the servlet is available for service. If the initialization fails, the servlet engine unloads the servlet. The administrator can set an application and its servlets to be unavailable for service. In such cases, the application and servlets remain unavailable until the administrator changes them to available.

Servicing requests

A client request arrives at the Application Server. The servlet engine creates a request object and a response object. The servlet engine invokes the servlet service method, passing the request and response objects.

The service method gets information about the request from the request object, processes the request, and uses methods of the response object to create the client response. The service method can invoke other methods to process the request, such as doGet(), doPost(), or methods you write.

Termination

The servlet engine invokes theservlet's destroy() method when appropriate and unloads the servlet. The Java Virtual Machine performs garbage collection after the destroy.

More on the initialization and termination phases

A servlet engine creates an instance of a servlet at the following times:

- Automatically at the application startup, if that option is configured for the servlet
- At the first client request for the servlet after the application startup
- When the servlet is reloaded

The init method executes only one time during the lifetime of the servlet. It executes when the servlet engine loads the servlet. For the Application Server Version 3, you can configure the servlet to be loaded when the application starts or when a client first accesses the servlet. The init method is not repeated regardless of how many clients access the servlet.

The destroy() method executes only one time during the lifetime of the servlet. That happens when the servlet engine stops the servlet. Typically, servlets are stopped as part of the process of stopping the application.

4.2.1.2: Servlet support and environment in WebSphere

IBM WebSphere Application Server supports the Java ServletAPI from Sun Microsystems. The product builds upon the specificationin two ways.

Article 4.2.1.2.2 describes several IBMextensions to the specification to make it easier to manage sessionstate, create personalized Web pages, generate better servlet errorreports, and access databases.

Article 4.2.1.2.3 describes some complimentary servlets includedwith the product. Add them to Web applications for extended functionality. You can use the WebSphere servlets as theyare, or use them as a basis for creating customized versions.

Beginning with version 3.5.2, WebSphere Application Server added support for the Java ServletAPI 2.2 from Sun Microsystems. See article 4.2.1.2.1afor a description of the Servlet API 2.2 specification.

4.2.1.2.1: Features of Java Servlet API 2.1

Some highlights of the Java Servlet API 2.1 are:

- A request dispatcher wrapper for each resource (servlet)

  A request dispatcher is a wrapper for resources that can process HTTP requests (such as servlets and JSPs) and files related to those resources (such as static HTML and GIFs). The servlet engine generates a single request dispatcher for each servlet or JSP when it is instantiated. The request dispatcher receives client requests and dispatches the request to the resource.

- A servlet context per application

  For the Java Servlet API 2.0, the servlet engine generated a single servlet context that was shared by all servlets. The Servlet API 2.1 provides a single servlet context per application, which facilitates partitioning applications. As explained in the description of the application programming model, applications on the same virtual host can access each other's servlet context.

- Deprecated HTTP session context

  The Servlet API 2.0 HttpSessionContext interface grouped all of the sessions for a Web server into a single session context. Using the session context interface methods, a servlet could get a list of the session IDs for the session context and get the session associated with an ID. As a security safeguard, this interface has been deprecated in the Servlet API 2.1. The interface methods have been redefined to return null.

4.2.1.2.1a: Features of Java Servlet API 2.2

WebSphere Application Server supports Java Servlet API 2.2 and JSP 1.1.

Java Servlet API 2.2 contains many enhancements intended to make servlets part of a complete application framework

These new functions in the Servlet 2.2 specification are ***SUPPORTED*** by WebSphere Application Server:

- response buffering
- WAR files (for depoyment)
- multiple error page support
- welcome file list
- new request mapping logic
- session timeout per Web application
- session scoping per Web application
- MIME mapping table per Web application
  (MIME table now exists at the VirtualHost and Web application)
- request dispatchers by servlet name
- Request dispatchers by relative path
- duplicate header support:
  `(req.getHeaders(name), resp.addHeader())`
- initialization parameters on a Web application
- internationalization improvements:
  `(getLocale(), getLocales())`

The following J2EE extensions in the Servlet 2.2 specification are ***NOT SUPPORTED***:

- J2EE security
- roles
- APIs: `isUserInRole()` and `getUserPrincipal()`
- J2EE-style Form Login
- EJB reference
- resource reference
- environment entry
- reference deployment information in `web.xml`
- security deployment information in `web.xml`
- accessing a JSP file through the URI without the .jsp extension, as for example,
  `../jsp/HitCount`
- creating a sevlet and associating a JSP file as the handler for the servlet

  The Servlet 2.2 specification allows you to associate a JSP tag to the servlet tag. However, the WebSphere Application Server WAR conversion tool does not supportthe `<jsp-file>` tag. The JSP tag association is illustrated in the following code example:

```
<servlet>     <servlet-name>JSPTest</servlet-name>
<jsp-file>/jsp/HitCount.jsp</jsp-file></servlet>/*mapping to URI*/ <servlet-mapping>
<servlet-name>JSPTest</servlet-name>
<url-pattern>/jsp/HitCount.jsp</url-pattern></servlet-mapping>
```

The Servlet 2.2 specification is available atjava.sun.com/products/servlet/index.html

No new classes were added to the Java Servlet API 2.2. specification.The following table provides more information on 27 new methods, 2 new constants and 6 deprecated methods supported by WebSphere Application Server:

New methods	Description
getServletName()	Returns the servlet's registered name
getNamedDispatcher(java.lang.String name)	Returns a dispatcher located by resource name
getInitParameter(java.lang.String name)	Returns the value for the named context parameter
getInitParameterNames()	Returns an enumeration of all the context parameter names
removeAttribute(java.lang.String name)	Added for completeness
getLocale()	Gets the client's most preferred locale

getLocales()	Gets a list of the client's preferred locales as an enumeration of locale objects
isSecure()	Returns **`true`** if the request was made using a secure channel
getRequestDispatcher(java.lang.String name)	Gets a `RequestDispatcher` using what can be a relative path
setBufferSize(int size)	Sets the minimum response buffer size
getBufferSize()	Gets the current response buffer size
reset()	Empties the response buffer, clears the response headers
isCommitted()	Returns true if part of the response has already been sent
flushBuffer()	Flushes and commits the response
setLocale(Locale locale)	Sets the response locale, including headers and charset
getLocale()	Gets the current response locale
UnavailableException(String message)	Replaces `UnavailableException(Servlet servlet, String message)`
UnavailableException(String message, int sec)	Replaces `UnavailableException(int sec, Servlet servlet, String message)`
getHeader(String message)	Returns all the values for a given header, as an enumeration of strings
getContextPath()	Returns the context path of this request
addHeader(String name, String value)	Adds to the response another value for this header name
addDateHeader(String name, long date)	Adds to the response another value for this header name
addIntHeader(String name, int value)	Adds to the response another value for this header name
getAttribute(String name)	`ObjectHttpSession.getValue(String name)`
getAttributeNames()	Replaces `String[] HttpSession.getValueNames()`
setAttribute(String name, Object value)	Replaces `void HttpSession.setValue(String name, Object value)`
removeAttribute(String name)	Replaces `void HttpSession.removeValue(String name)`
New constants	**Description**
SC_REQUESTED_RANGE_NOT_SATISFIABLE	New mnemonic for status code 416
SC_EXPECTATION_FAILED	New mnemonic for status code 417
Newly *deprecated* methods	**Description**
UnavailableException(Servlet servlet, String message)	Replaced by `UnavailableException(String message)`
UnavailableException(int sec, Servlet servlet, String message)	Replaced by `UnavailableException(string message, int sec)`
getValue(String name)	Replaced by `Object HttpSession.getAttribute(String name)`
getValueNames()	Replaced by `numeration HttpSession.getAttributeNames()`
putValue(String message, Object value)	Replaced by`void HttpSession.setAttribute(String name, Object value)`
removeValue(String message)	Replaced by `void HttpSession removeAttribute(String name)`

4.2.1.2.2: IBM extensions to the Servlet API

The Application Server includes its own packages that extend and add to the Java Servlet API. Those extensions and additions make it easier to manage session state, create personalized Web pages, generate better servlet error reports, and access databases. The Javadoc for the Application Server APIs is installed in the product *product_installation_root*\web\apidocs directory.

The Application Server API packages and classes are:

- `com.ibm.servlet.personalization.sessiontracking` package

  This Application Server extension to the Java Servlet API records the referral page that led a visitor to your Web site, tracks the visitor's position within the site, and associates user identification with the session. IBM has also added session clustering support to the API.

- `com.ibm.websphere.servlet.session.IBMSession` interface

  Extends HttpSession for session support and increased Web administrators' control in a session cluster environment.

- `com.ibm.servlet.personalization.userprofile` package

  Provides an interface for maintaining detailed information about your Web visitors and incorporate it in your Web applications, so that you can provide a personalized user experience. This information is made persistent by storing it in a database.

- `com.ibm.websphere.userprofile` package

  User profile enhancements

- `com.ibm.db` package

  Includes classes to simplify access to relational databases and provide enhanced access functions (such as result caching, update through the cache, and query parameter support).

- `com.ibm.websphere.servlet.error.ServletErrorReport` class

  A class that enables the application to provide more detailed and tailored messages to the client when errors occur. See the enhanced servlet error reporting article, 4.2.1.3.5, for details.

- `com.ibm.websphere.servlet.event` package

  Provides listener interfaces for notifications of application lifecycle events, servlet lifecycle events, and servlet errors. The package also includes an interface for registering listeners. See the package Javadoc for details.

- `com.ibm.websphere.servlet.filter` package

  Provides classes that support servlet chaining. The package includes the ChainerServlet, the ServletChain object, and the ChainResponse object. See the servlet filtering article, 4.2.1.3.4, for more details.

- `com.ibm.websphere.servlet.request` package

  Provides an abstract class, HttpServletRequestProxy, for overloading the servlet engine's HttpServletRequest object. The overloaded request object is forwarded to another servlet for processing. The package also includes the ServletInputStreamAdapter class for converting an InputStream into a ServletInputStream and proxying all method calls to the underlying InputStream. See the Javadoc for details and examples.

- `com.ibm.websphere.servlet.response` package

Provides an abstract class, HttpServletResponseProxy, for overloading the servlet engine's HttpServletResponse object. The overloaded response object is forwarded to another servlet for processing. The package includes the ServletOutputStreamAdapter class for converting an OutputStream into a ServletOutputStream and proxying all method calls to the underlying OutputStream. The package also includes the StoredResponse object that is useful for caching a servlet response that contains data that is not expected to change for a period of time, for example, a weather forecast. See the Javadoc for details and examples.

4.2.1.2.3: Using the WebSphere servlets for a head start

IBM Application Server provides internal (built-in) WebSphere servlets that you can add to your Web applications to enable optional functions.

The tables below describe each WebSphere servlet and how to use the Java console to add it to a Web application. To determine whether a WebSphere servlet currently belongs to a Web application, check the Web application configuration for the presence of the servlet by its administrative name.

Invoke servlets by class name

Objective	Invoke servlets by class or code names (such as MyServletClass)
Servlet administrative name	invoker
Servlet code	com.ibm.servlet.engine.webapp.Invoker
How to add to Web application	When using the Console -> Task -> Configure a Web application wizard, specify to serve servlets by classname. For an existing Web application, use the Console -> Tasks -> Add a servletwizard.
More information	Using the Invoker servlet is considered a security exposure that can be avoided by performingcertain administrative tasks. See the Related information for details. The default invoker's URL in Servlet 2.2 ***compliance*** modeis `/servlet/*` , not `/servlet/`.See file, New Servlet Engine option for migrating applications to Servlet 2.2, for information on the two modes: compliance versus compatibility.

Serve files without specifically configuring them

Objective	Serve HTML, servlets, or other files in the Web application document root without extra configuration steps. For HTML files, you will not need to add a pass rule to the Web server. For servlets, you will not need to explicitlyocnfigure the servlets in the WebSphere administrative domain.
Servlet administrative name	file
Servlet code	com.ibm.servlet.engine.webapp.SimpleFileServlet
How to add to Web application	When using the Console -> Task -> Configure a Web application wizard, specifyto enable the file servlet. For an existing Web application, use the Console -> Tasks -> Add a servletwizard.
More information	This servlet handles files in the application document root whose URLs are not covered by the configured servlet URLs

Enable Web applications to serve JSP files

Objective	Enable the JSP page compiler to allow Web application to handle JSP files
Servlet administrative name	See section 4.2.1.2
Servlet code	See section 4.2.1.2
How to add to Web application	When using the Console -> Task -> Configure a Web application wizard, specify a JSP level for the Web application. For an existing Web application, use the Console -> Tasks -> Add a JSP enabler wizard.
More information	Adding a JSP processor to an application is required if the Web application contains JSP files. • 4.2.1.2: JSP processors • 6.6.10: Administering JSP files

Enable an error page without having to write one

Objective	Enable error reporting through an error page, withoutwriting your own error page
Servlet administrative name	ErrorReporter
Servlet code	com.ibm.servlet.engine.webapp.DefaultErrorReporter
How to add to Web application	Configure the Web application, then add the ErrorReporter servlet byusing the Console -> Tasks -> Add a servlet wizard.
More information	4.2.1.3.5: Enhancing servlet error reporting

Enable servlet chaining

Objective	Enable a servlet chain, in which servlets forward output and responses to other servlets for processing
Servlet administrative name	Chainer
Servlet code	com.ibm.websphere.servlet.filter.ChainerServlet
How to add to Web application	Configure the Web application, then add the Chainer servlet byusing the Console -> Tasks -> Add a servlet wizard.
More information	• 4.2.1.3.4: Filtering and chaining servlets

4.2.1.2.3.1: Avoiding the security risks of invoking servlets by class name

Anyone enabling the Invoker servlet to serve servlets by their class names

Anyone enabling the "serve files by class name" function in WebSphere Application Server, should take steps to avoid potential security risks. The administrator should remain aware of each and every servlet classplaced in the classpath of an application, even if the servlets are to be invoked by their classnames.

Appending `/$/foo` to the URL allows you to access the servlet URL, butthe URL is then misunderstood by the runtime environment.This type of URL may create a security exposure.The best practice for securing servlets is to follow the Java security specifications documented in the Securing applications section.

A Web site may inadvertently include malicious HTML tags or scripts in a dynamically generated page based on unvalidated input from untrustworthy sources.By accessing a malicious URL and then accessing an application server, a usermay unknowingly execute script code on his machine that exposes the data received from the server. The browser executes the script on the user machine without the knowledge of the user.

The malicious tagsthat can be embedded in this way are <SCRIPT> and </SCRIPT>.

This problem can be prevented if the server generated pages are encoded to prevent thescripts from executing.Developers generating responses containing client data, based on servlet or JSP requests, canencode the responses using the following method:

```
com.ibm.websphere.servlet.response.ResponseUtils.encodeDataString(String)
```

Visit the Cert advisories Web sitefor more information.

Protecting servlets

To protect each servlet, the administrator needs to:

1. Configure a Web resource based on the servlet class name, such as:

   ```
   /servlet/SnoopServlet
   ```

   for SnoopServlet.class
2. Add the Web resource to the Web Path list of the Invoker servlet in the Web applicationto which the servlet belongs.
3. Use the Configure Resource Security wizard in the Java administrative console to secure the Web resource.

Also, the administrator needs to secure the Invoker servlet itself.

Details

WebSphere security is based on defining, and then securing, URIs (knownas Web resources) for servlets. This allows an administrator to applydifferent security levels to different paths for accessing the same servlet.Also, Web resources are logical designations that are not guaranteed to match servlet class names. For these reasons, actual class names are irrelevant to WebSphere security unless you explicitly specify that you want to protect the path for invoking a servlet by its class name.

When a Web application allows users to invoke servlets by class name, the administrator is able to drop servlets into a Web application without havingto explicitly define them in WebSphere systems administration.

Suppose that the WebSphere administrator drops in a servlet class tobe invoked by its class name. Even if a servlet corresponding to the same class name is defined and protected, users will be able to invoke the servlet by class name without any security checks. (The exception is if the administrator has created a Web resource corresponding to the servletclass name, as described in the above steps).

Undefined servlets remain unprotected unless steps are taken to assign secure Web resources to them based on their class names.

4.2.1.2.3b: Security risk example of invoking servlets by class name

Anyone enabling the "serve files by class name" function in WebSphere Application Server, should take steps to avoid potential security risks. The administrator should remain aware of each and every servlet classplaced in the classpath of an application, even if the servlets are to be invoked by their classnames.

 A Web site may inadvertently include malicious HTML tags or scripts in a dynamically generated page based on unvalidated input from untrustworthy sources.By accessing a malicious URL and then accessing an application server, a usermay unknowingly execute script code on his machine that has full access to the data and resources on that machine. The browser executes the script on the user machine without the knowledge of the user.

The malicious tagsthat can be embedded in this way are <SCRIPT> and </SCRIPT>.

This problem can be prevented if the server generated pages are encoded to prevent thescripts from executing.Developers generating responses containing client data, based on servlet or JSP requests, canencode the response data using the following method:

`com.ibm.websphere.servlet.response.ResponseUtils.encodeDataString(String)`

Visit the Cert advisories Web sitefor more information.

4.2.1.3: Servlet content, examples, and samples

Click the related topics to focus on particular aspects of servletdevelopment, including example and sample code.

4.2.1.3.1: Creating HTTP servlets

To create an HTTP servlet, as illustrated in ServletSample.java:

1. Extend the HttpServlet abstract class.
2. Override the appropriate methods. The ServletSample overrides the doGet() method.
3. Get HTTP request information, if any.

   Use the HttpServletRequest object to retrieve data submitted through HTML forms or as query strings on a URL. The ServletSample example receives an optional parameter (myname) that can be passed to the servlet as query parameters on the invoking URL. An example is:

   `http://your.server.name/application_URI/ServletSample?myname=Ann`

   The HttpServletRequest object has specific methods to retrieve information provided by the client:

   - getParameterNames()
   - getParameter(java.lang.String name)
   - getParameterValues(java.lang.String name)
4. Generate the HTTP response.

   Use the HttpServletResponse object to generate the client response. Its methods allow you to set the response headers and the response body. The HttpServletResponse object also has the getWriter() method to obtain a PrintWriter object for sending data to the client. Use the print() and println() methods of the PrintWriter object to write the servlet response back to the client.

4.2.1.3.1.1: Overriding HttpServlet methods

HTTP servlets are specialized servlets that can receive HTTP client requests and return a response. To create an HTTP servlet, subclass the HttpServlet class. A servlet can be invoked by its URL, from a JavaServer Page (JSP), or from another servlet.

Methods to override

The `javax.servlet.http.HttpServlet` class contains the init, destroy, and service methods. The init and destroy methods are inherited, while the service methodimplementation is specific to HttpServlet. The method behaviors are described below; however, you might want to override methods in order to provide specialized behavior in your servlet.

- **init**

  The default init method is usually adequate but can be overridden with a custom init method, typically to register application-wide resources. For example, you might write a custom init method to load GIF images only one time, improving the performance of servlets that return GIF images and have multiple client requests. Other examples are initializing a database connection and registering servlet context attributes.

  The Java Servlet API 2.1 provides a new init method: `init()`, the no argument init method that is inherited from the superclass GenericServlet. The GenericServlet also implements the ServletConfig object. The benefit is that when you use the no-argument init method in your servlet, you do not need to call `super.init(config)`. The reason is that servlet engines that implement the Servlet API 2.1 call the servlet's `init(ServletConfig config)` method behind the scenes. In turn, the GenericServlet calls the servlet's init() method.

  If a servlet exception is thrown inside the init method, the servlet engine will unload the servlet. The init method is guaranteed to complete before the service method is called.

- **destroy**

  The default destroy method is usually adequate, but can be overridden.Override the destroy method if you need to perform actions during shutdown. For example, if a servlet accumulates statistics while it is running, you might write a destroy() method that saves the statistics to a file when the servlet is unloaded. Other examples are closing a database connection and freeing resources created during the initialization.

  When the server unloads a servlet, the destroy method is called after all service method calls complete or after a specified time interval. Where threads have been spawned from within service method and the threads have long-running operations, those threads may be outstanding when the destroy method is called. Because this is undesirable, make sure those threads are ended or completed when the destroy method is called.

- **service**

  The service method is the heart of the servlet. Unlike the init and destroy methods, it is invoked for each client request. In the HttpServlet class, the service method already exists. The default service function invokes the doXXX method corresponding to the method of the HTTP request. For example, if the HTTP request method is GET, doGet method is called by default. Because the HttpServlet.service method checks the HTTP request method and calls the appropriate handler method, it is usually not desirable to override the service method. Rather, override the appropriate doXXX methods that the servlet supports.

4.2.1.3.2: Inter-servlet communication

There are three types of servlet communication:

- Accessing data within a servlet's scope
- Forwarding a request and including a response from another servlet using the RequestDispatcher
- Application-to-application communication via the ServletContext

Sharing data within scope

JavaServerPages (JSPs) use this method to share data through beans. The ability of servlets to share data depends on the scope of the bean. The possible scopes are request, session, and application.

Forwarding and including data

For session-scoped data and attributes, use the HttpSession.setAttribute and getAttribute methods to set and get attributes in the HttpSession object. Session-scoped beans and objects bound to a session are examples of session-scoped objects.

For the Servlet API 2.1, an HttpSession object is only accessible to the Web applications and servlets that are a part of that session. In the Servlet API 2.1, a servlet cannot determine the ID of another session and request its SessionContext, because the HttpSessionContext and related methods are deprecated (returns null).

For application-scoped data, use the RequestDispatcher's forward and include methods to share data among applications. The forward method sends the HTTP request from one servlet to a second servlet for additional processing. The calling servlet adds the URL and request parameters in its HTTP request to the request object passed to the target servlet. The forwarding servlet must not have committed any output to the client. The target servlet generates the response and returns it to the client.

The include method enables a receiving servlet to include another servlet's response data in its response. The included servlet cannot set response headers. The receiving servlet can fully access the request object but can only write data to the ServletOutputStream or PrintWriter of the response object. If the servlets use session tracking, you must create the session outside of the included servlet. The RequestDispatcher.forward method is similar in function to the HttpServiceResponse.callPage method previously supported for JSP development.

Application-to-application communication

Web applications share data through the ServletContext. A Web application has a single servlet context. A ServletContext object is accessible to any Web application associated with a virtual host. Servlet A in application A can obtain the ServletContext for application B in the same virtual host. After Servlet A obtains the servlet context for B, it can access the request dispatcher for servlets in application B and call the getAttribute and setAttribute methods of the servlet context. An example of the coding in Servlet A is:

```
appBcontext = appAcontext.getContext("/appB");
appBcontext.getRequestDispatcher("/servlet5");
```

4.2.1.3.2.1: Forwarding and including data (request and response)

When the servlet engine calls the service method of an HTTP servlet, it passes two objects as parameters:

- HttpServletRequest (the Request object)
- HttpServletResponse (the Response object)

The servlet communicates with the server andultimately with the client through these objects. The servlet reads the Request object from a ServletInputStream. The servlet can invoke the Requestobject's methods to get information about the client environment, theserver environment, and any information provided by the client (for example,form information set by GET or POST).The servlet invokes the Response object's setter methods to return the client response. However, if the servlet is part of a servlet chain, it might pass its response object to another servlet for further processing.

4.2.1.3.2.2: Example: Servlet communication by forwarding

In this example, the forward method is used to send a message to a JSP file (a servlet) that prints the message. The forwarding servlet code is:

```
import java.io.*;import javax.servlet.*;import javax.servlet.http.*;public class UpdateJSPTest extends HttpServlet{   public void doGet (HttpServletRequest req, HttpServletResponse res) throws ServletException, IOException     {        String message = "This is a test";  req.setAttribute("message", message);        RequestDispatcher rd = getServletContext().getRequestDispatcher("/Update.jsp");        rd.forward(req, res);      }}
```

The JSP file is:

```
<html><head></head><body><h1><servlet code=UpdateJSPTest></servlet></h1><%    String message = (String) request.getAttribute("message");    out.print("message: <b>" + message + "</b>");%><p><ul><% for (int i = 0; i < 5; i++)        {       out.println ("<li>" + i);  }%></ul></body></html>
```

4.2.1.3.3: Using page lists to avoid hard coding URLs

IBM WebSphere Application Server supports page lists, which allow applicationdevelopers to prevent hard-coding URLs in servlets and JSP files. To learn howpage lists work, and their advantages, see the page lists descriptioncited in the Related information below.

Use IBM WebSphere Studio to develop (1) servlets that support page lists, and(2) their accompanying .servlet configuration files that specify the page lists.Alternatively, use materials supplied by IBM WebSphere Application ServerVersion 3.x to manually create the two items.

Regardless of how you obtain them, servlets and their .servlet configurationfiles can be deployed in an IBM WebSphere Application Server environment.

See the Related information for instructions for using .servletconfiguration files obtained from either Studio or WebSphere ApplicationServer.

4.2.1.3.3.1: Obtaining and using servlet XML configuration files (.servlet files)

The IBM WebSphere Studio provides wizards that generate servlets withaccompanying XML servlet configuration files (.servlet files).

If you do not have access to the Studio, you can manually implementXML servlet configuration files. The servlet must also be modified orcreated to support the use of a .servlet file for its configuration.

Using .servlet files from IBM WebSphere Studio

1. Use IBM WebSphere Studio to create a servlet and .servlet files. See theStudio documentation for instructions.
2. Deploy the compiled servlet and its XML servlet configuration file on the applicationserver.

Using manually configured .servlet files

1. Create or obtain a servlet that extends the PageListServlet class.
2. Use the XMLServletConfig class to create an XML servlet configuration file for the servlet instance.
3. Deploy the compiled servlet and its XML servlet configuration file.

4.2.1.3.3.1.1: Extending PageListServlet

IBM WebSphere Application Server supplies the PageListServlet, the superclass of servlets that load pages contained in the page list element (<page-list>) of an XML servlet configuration file.

Implement a servlet that supports the use of XML configuration files(.servlet files) and page lists by extending the PageListServlet class.

The PageListServlet has a callPage() method that invokes a JavaServer Page in response to an HTTP request for a page in the page list.

The PageListServlet.callPage() method receives as input:

- A page name from the page-list element of the XML configuration file
- The HttpServletRequest object
- The HttpServletResponse object

In structuring the servlet code, keep in mind that the PageListServlet.callPage() method is not an exit. Any servlet code that follows the callPage() method invocation will be run after the invocation.

See the Related information for an example of a servlet that extends thePageListServlet.

4.2.1.3.3.1.1.1: Example: Extending PageListServlet

SimplePageListServlet is an example of a servlet that extends the PageListServlet class and uses its callPage() method to invoke a JSP:

```
public class SimplePageListServlet extends com.ibm.servlet.PageListServlet {   public void
service(HttpServletRequest req, HttpServletResponse resp)   throws ServletException, IOException   {
try{         setRequestAttribute("testVar", "test value", req);
setRequestAttribute("otherVar", "other value", req);         String pageName =
getPageNameFromRequest(req);         callPage(pageName, req, resp);      }      catch(Exception e){
handleError(req, resp, e);      }   }}
```

4.2.1.3.3.1.2: Using XMLServletConfig to create .servlet configuration files

IBM WebSphere Application Server supplies the XMLServletConfig class for creating XML servlet configuration files (*servlet_instance_name*.servlet files).

Write a Java program that uses the XMLServletConfig class to generate a servlet configuration file. The XMLServletConfig class provides methods for setting and getting the file elements and their contents.

See the comments in the XMLServletConfig class for an explanation of how to use it.

4.2.1.3.3.1.3: XML servlet configuration file syntax (.servlet syntax)

Each XML configuration file must be a well-formed XML document. The files are not validated against a Document Type Definition (DTD). This article describes the syntax, as illustrated by theexample cited in Related information.

For the Application Server to use an XML servlet configuration file to load a servlet instance, the file must contain at least the code element. For a PageListServlet, the XML configuration file must contain at least the code element and the page-list element.

Although there is no DTD, it is recommended that all elements appear in the order shown in the example. The elements (also known as *tags*) are:

Tag	Description
servlet	The root element of an XML configuration file. The XMLServletConfig class automatically generates this element.
code	The class name of the servlet (without the .class extension), even if the servlet is in a JAR file
description	A user-defined description of the servlet
init-parameter	The attributes of this element specify a name-value pair to be used as an initialization parameter. A servlet can have multiple initialization parameters, each within its own init-parameter element.
page-list	Identifies the JavaServer pages to be called depending on the path information in the HTTP request. The page-list element can contain the following child elements: ● default-page: Contains a uri element that specifies the location of the page to be loaded, if the HTTP request does not contain path information ● error-page: Contains a uri element that specifies the location of the page to be loaded, if the handleError() method sets the request attribute "error" ● page: Contains a uri element that specifies the location of the page to be loaded if the HTTP request contains the page name. A page-list element can contain multiple page elements.

4.2.1.3.3.1.4: Example: XML servlet configuration file

```
<?xml version="1.0"?><servlet>  <code>SimplePageListServlet</code>  <description>Shows how to use
PageListServlet class</description>  <init-parameter name="name1" value="value2"/>  <page-list>
<default-page>      <uri>/index.jsp</uri>    </default-page>    <error-page>
<uri>/error.jsp</uri>    </error-page>    <page>      <uri>/TemplateA.jsp</uri>
<page-name>page1</page-name>    </page>    <page>      <uri>templateB.jsp</uri>
<page-name>page2</page-name>    </page>  </page-list></servlet>
```

4.2.1.3.4: Filtering and chaining servlets

The Application Server supports two kinds of filtering:

- *MIME-based filtering* involves configuring the servlet engine to forward HTTP responses with the specified MIME type to the designated servlet for further processing.
- Servlet chaining involves defining a list (a sequence) of two or more servlets such that the request object and the ServletOutputStream of the first servlet is passed to the next servlet in the sequence. This process is repeated at each servlet in the list until the last servlet returns the response to the client.

4.2.1.3.4.1: Servlet filtering with MIME types

To configure MIME filters, use an administrative client to configure recognized MIME types for virtualhosts containing servlets.

4.2.1.3.4.2: Servlet filtering with servlet chains

To configure a servlet chain, use the administrative console to:

- Define the following initialization parameter and value for the ChainerServlet:

Parameter	Value
chainer.pathlist	*/first_servlet_URL /next_servlet_URL*

The chainer.pathlist is a space-delimited list of servlet URLs. For example, if you want the sequence of servlets to be three servlets that you added to the examples application (servletA, servletB, servletC), specify:

Parameter	Value
chainer.pathlist	/servletA /servletB /servletC

- To invoke a servlet chain, invoke the servlet URL of the ChainerServlet in your application. instance, for example, http://*your.server.name*/webapp/example/abc.

4.2.1.3.5: Enhancing servlet error reporting

A servlet can report errors by:

- Calling the ServletResponse.sendError method
- Throwing an uncaught exception within its service method

The enhanced servlet error reporting function in IBM WebSphere Application Server provides an easier way to implement error reporting. The error page (a JSP file or servlet) is configured for the application and used by all of the servlets in that application. The new mechanism handles caught and uncaught errors.

To return the error page to the client, the servlet engine:

1. Gets the ServletContext.RequestDispatcher for the URI configured for the application error path.
2. Creates an instance of the error bean (type `com.ibm.websphere.servlet.errorServletErrorReport`). The bean scope is request, so that the target servlet (the servlet that encountered the error) can access the detailed error information.

For the Application Server, the ServletResponse.sendError() method has been overriden to provide the functionality previously described. The overriden method is shown below:

```
public void sendError(int statusCode, String message){   ServletException e = new
ServletErrorReport(statusCode, message);   request.setAttribute(ServletErrorReport.ATTRIBUTE_NAME,
e);   servletContext.getRequestDispatcher(getErrorPath()).forward(request, response);}
```

4.2.1.3.5.1: Public methods of the ServletErrorReport class

To create an error JSP or servlet, you need to know the public methods of the `com.ibm.websphere.servlet.error.ServletErrorReport` class (the error bean), which are:

```
public class ServletErrorReport extends ServletException{       //Get the stacktrace of the error as a string   public String getStackTrace()   //Get the message associated with the error.    //The same message is sent to the sendError() method.   public String getMessage()      //Get the error code associated with the error. //he same error code is sent to the sendError() method. //This will also be the same as the status code of the response.        public int getErrorCode()       //Get the name of the servlet that reported the error   public String getTargetServletName()}
```

4.2.1.3.5.2: Example: JSP file for handling application errors

As illustrated in the following code example, specify "ErrorReport" for the id value.The error page loads an instance of code from the request space named "ErrorReport"to read the properties. If the default scope (scope="page") is used, a new instance of the code is created and the properties are blank.

```
<html><jsp:useBean id="ErrorReport"
class="com.ibm.websphere.servlet.error.ServletErrorReport"scope="request" /><head>   <title>
ERROR: <%= ErrorReport.getErrorCode() %>   </title></head><body><H1>  This error occured while
processing the servlet named:   <%= ErrorReport.getTargetServletName() %></H1><B>My Message: </B><%=
ErrorReport.getMessage() %><BR><BR><B>My StackTrace: </B><%= ErrorReport.getStackTrace()
%><BR></body></html>
```

If you do not want to write your own error, consider adding the optional internal servlet, com.ibm.servlet.engine.webapp.DefaultErrorReporter, to your Web application.

4.2.1.3.6: Serving servlets by classname

IBM WebSphere Application Server provides a WebSphere servletthat you can add to your Web applications. Web applications thatcontain the servlet can serve servlets by theservlet classnames (such as MyServletClass). No additional stepsare required.

See the details and instructions.

4.2.1.3.7: Serving all files from application servers

IBM WebSphere Application Server provides a WebSphere servletthat you can add to your Web applications. Web applications thatcontain the servlet can serve HTML, eliminating the need to adda pass rule to the Web server for serving the same HTML files. No additional steps are required.

See the details and instructions.

4.2.1.3.8: Obtaining the Web application classpath from within a servlet

To have a servlet or JSP-generated servlet detect the classpathof the Web application to which it belongs, get the

`com.ibm.websphere.servlet.application.classpath`

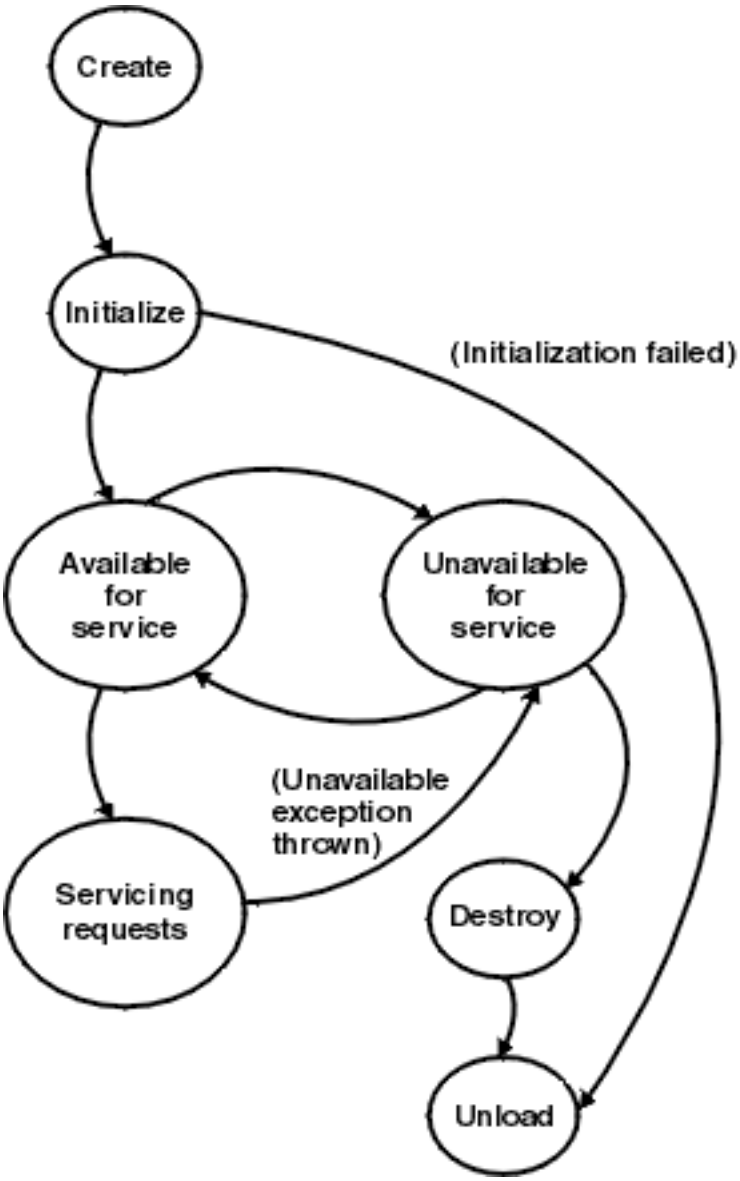
attribute from the ServletContext.

4.2.2: Developing JSP files

If JSP files are fairly new to you, consider reading about their lifecycle and access model. When you are ready to begin writing JSP files, see the article featuring JSP file content. Review the support and environment article for topics such as JSP processors and APIs, recommended development tools, and batch compiling.

4.2.2.1: JavaServer Pages (JSP) lifecycle

JSP files are compiled into servlets. After a JSP is compiled, its lifecycle is similar to the servlet lifecycle:



Java source generation and compilation

When a Web container receives a request for a JSP file, it passes the request to the JSP processor .

If this is the first time the JSP file has been requested or if the compiled copy of the JSP file is not found, the JSP compiler generates and compiles a Java source file for the JSP file. The JSP processor puts the Java source and class file in the JSP processor directory.

By default, the JSP syntax in a JSP file is converted to Java code that is added to the service() method of the generated class file. If you need to specify initialization parameters for the servlet or other initialization information, add the method directive set to the value `init`.

Request processing

After the JSP processor places the servlet class file in the JSP processor directory, the Web container creates an instance of the servlet and calls the servlet service() method in response to the request. All subsequent requests for the JSP are handled by that instance of the servlet.

When the Web container receives a request for a JSP file, the engine checks to determine whether the JSP file (.jsp) has changed since it was loaded. If it has changed, the Web container reloads the updated JSP file (that is, generates an updated Java source and class file for the JSP). The newly loaded servlet instance receives the client request.

Termination

When the Web container no longer needs the servlet or a newinstance of the servlet is being reloaded, the Web container invokes theservlet's destroy() method. The Web container can also call the destroy() method if the engine needs to conserve resources or a pending call to a servlet service() method exceeds the timeout. The Java Virtual Machine performs garbage collection after the destroy.

4.2.2.1a: JSP access models

JSP files can be accessed in two ways:

- The browser sends a request for a JSP file.

  The JSP file accesses beans or other components that generate dynamic content that is sent to the browser,as shown:

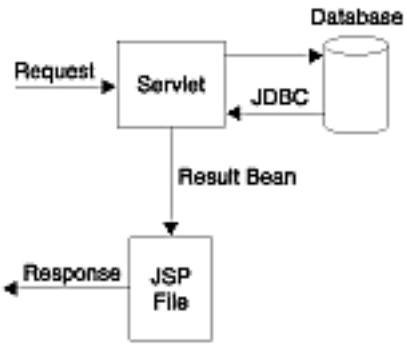
  Request for a JSP file

  When the Web server receives a request for a JSP file, the server sends therequest to the application server. The application server parses theJSP file and generates Java source, which is compiled and executed as aservlet.

- The request is sent to a servlet that generates dynamic content and calls a JSP file to send the content to the browser, as shown:

  Request for a servlet

  This access model facilitates separating content generation from content display.

  The application server supplies a set of methods in the HttpServiceRequest object and the HttpServiceResponse object. These methods allow an invoked servlet to place an object (usually a bean) into a request object and pass that request to another page (usually a JSP file) for display. The invoked page retrieves the beanfrom the request object and generates the client-side HTML.

4.2.2.2: JSP support and environment in WebSphere

IBM WebSphere Application Server supports the JSP 1.1 Specification from Sun Microsystems. If you are going to develop new JSP files for use with IBMWebSphere Application Server, it is recommended you use JSP 1.1.

All APIs described in this section are supported at the JSP 1.1 level.

It also supports the JSP .91 and JSP 1.0 Specification.Please consult the Related information for the necessary migration of JSP .91 APIs that are deprecated in Version 3.5.

4.2.2.2.1: JSP support for separating logic from presentation

Two interfaces support the JSP technology. These APIs provide a way to separate content generation (business logic) from thepresentation of the content (HTML formatting).

This separation enablesservlets to generate content and store the content (for example, in a bean) inthe request object. The servlet that generated the context generates aresponse by passing the request object to a JSP file that contains the HTMLformatting. The <BEAN> tag provides access to the businesslogic.

The <useBEAN> tag provides access to the businesslogic.

Goal	Interface
Set attributes in the request object.	javax.servlet.http.HttpServletRequest.setAttribute()
Forward a response object to another servlet or JSP file.	javax.servlet.http.RequestDispatcher.forward()

In IBM WebSphere Application Server Version 2.0x, theseinterfaces had different names. You might need to migrate codethat is calling the old interfaces. See the Related informationfor details.

4.2.2.2.2: JSP processors

IBM WebSphere Application Server provides a JSP processor for each supported levelof the JSP specification, .91 and 1.0. Each JSP processor is a **servlet** that you canadd to a Web application to handle all JSP requests pertaining to the Web application.

When you install the Application Server product on a Web server, the Web server configuration is set to pass HTTP requests for JSP files (files with the extension .jsp) to the Application Server product.

By specifying either a .91, 1.0 or 1.1 JSP Enabler for each Web application containingJSP files, you configure Web applications to pass JSP files in the Web application folder to the JSP processor correspondingto the JSP specification level of the JSP files.

The JSP processor creates and compiles a servlet from each JSP file. The processor produces these files for each JSP file:

- .java file, which contains the Java language code for the servlet
- .class file, which is the compiled servlet

The JSP processor puts the .java, and the .class filein a path specific to theprocessor (see below). The .java and the .class file have the same filename. The processor uses a naming convention that includes adding underscore characters and a suffix to the JSP filename.

For example, if the JSP filename is `simple.jsp`, the generated files are `_simple_xjsp.java` and `_simple_xjsp.class`.

Like all servlets, a servlet generated from a JSP file extends javax.servlet.http.HttpServlet. The servlet Java code contains import statements for the necessary classes and a package statement, if the servlet class is part of a package.

If the JSP file contains JSP syntax (such as directives and scriptlets), the JSP processor converts the JSP syntax to the equivalent Java code. If the JSP file contains HTML tags, the processor adds Java code so that the servlet outputs the HTML character by character.

JSP 1.0 processor

Processor servlet name	JSP Servlet
Class name and location	com.sun.jsp.runtime.JspServlet in jsp10.jar
Where processor puts output	*product_installation_root*\temp*servlet_host_name**app_name***???????**

For example, if the JSP file is in:

```
c:\WebSphere\AppServer\hosts\default_host\examples\web
```

the .java and .class files are put in:

```
c:\WebSphere\AppServer\temp\default_host\examples\???????
```

JSP .91 processor

Processor servlet name	PageCompileServlet

Class name and location	com.ibm.servlet.jsp.http.pagecompile.PageCompileServlet inibmwebas.jar
Where processor puts output	*product_installation_root*\temp*servlet_host_name**app_name*\pagecompile where *product_installation_root* is the path where the Application Server is installed and *app_name* is the name of the application root folder.

For example, if the JSP file is in:

```
c:\WebSphere\AppServer\hosts\default_host\examples\web
```

the .java and .class files are put in:

```
c:\WebSphere\AppServer\temp\default_host\examples\pagecompile
```

4.2.2.2.3: Java Server Page attributes

Use the WebSphere Application Assembly Tool (AAT) to set the following Java Server Page attributes. The JSP attributes are storedin the IBM extensions document for Web module, `ibm-web-ext.xmi`.

JSP file attribute names	JSP file attribute values (Default values are in bold text)	Purpose
keepgenerated	true | **false**	If true, the generated `.java`file will be kept. If the value is false, the `.java` file isnot saved.
dosetattribute	true | **false**	By default, JSP files using the "usebean" tag withScope="session" do not always work properly when session persistence is enabled.
scratchdir	***product_installation_root*\temp**	Set `scratchdir` to a valid drive and directory which the JSP enginewill use to store the generated `.class` and `.java` files.
jsp.repeatTag.ignoreException	true | **false**	In previous releases, the <tsx:repeat> tagwould iterate until one of the following conditions was met: 1. The end value was reached 2. An `ArrayIndexOutofBoundsException` was thrown Other types of exceptions were caught but not thrown, which could result in numerous errors being returned to the browser. In version 4.0, the default behavior will now stop therepeat tag iterations when any exception is thrown. To reinstate the old behavior, set this parameter's valueto **true**.
defaultEncoding	Name of the desired character set. The value of the system property**file.encoding** is the default.	Use this parameter to set the encoding for JSP pages. If a JSP page contains a `contentType` directive that defines an alternative character set, that character set is used instead of the `defaultEncoding` parameter's value. The order of precedence is: 1. The JSP page's `contentType`directive's charset. 2. The `defaultEncoding` parameter's value. 3. The System property `file.encoding` value 4. ISO-8859-1

4.2.2.2.4: Batch Compiling JSP files

As an IBM enhancement to JSP support, IBM WebSphere Application Server provides a batch JSP compiler. Use this function to batch compile your JSP files and thereby enable faster responses to the initial client requests for the JSP files on your production Web server.

It is best to batch compile all of the JSP files associated with an application. Batch compiling saves system resources and provides security on the application server by specifying if and/or when the server is to check for a classfile or recompile a JSP file. The application server will monitor the compiled JSP file for changes, and will automatically recompile and reload the JSP file whenever the application server detects that the JSP file has changed. By modifying this process, you can eliminate time- and resource-consuming compilations and ensure that you have control over the compilation of your JSP files. It is also useful as a fast way to resynchronize all of the JSP files for an application.

The process of batch compiling JSP files is different for JSP 0.91 files and JSP 1.0 files. Consult the page corresponding to the JSP level for your files.

4.2.2.2.4.1: Compiling JSP .91 files as a batch

To use the JSP batch compiler for JSP .91 files:

1. Add the following JAR files (found in the Application Server lib directory) to your system classpath:
   - ibmwebas.jar (contains the batch compiler classes)
   - servlet.jar (contains the Java Servlet 2.1 APIs)
2. At an operating system command prompt, enter the following command on a single line:

   `java com.ibm.servlet.jsp.http.pagecompile.jsp.tsx.batch.JspBatch -s` *`sourceRootDir`* `-t` *`targetRootDir`* `-c` *`classPath`* `-l` *`libDirectory`* `-v`

   where:
   - **sourceRootDir**

     The root directory of the paths where the batch JSP compiler will search JSP source files to process. The compiler processes all files with the extension .jsp that are in the source root and its subdirectories.
   - **targetRootDir**

     The root directory of the path where you want the compiler to place the resulting .java and .class files. The non-batch, JSP 0.91 processor (PageCompileServlet) places the .java and .class files in the path:

     `product_installation_root``\temp\`*`servlet_host_name`*`\`*`app_name`*`\pagecompile`

     where *product_installation_root* is the path where the Application Server is installed and *app_name* is the name of the application root folder. It is recommended that you specify that path for the target root if you are batch compiling JSPs to run on your production Application Server. However, if you are batch compiling on a different system and plan to move them to the Application Server later, you can specify any valid target root directory.

     If any of the .class files have package names, those names will become the names of subdirectories under the target root. For example, if the .class name is security.login.login.class and the target root is `d:\WebSphere\AppServer\temp\default_host\examples\pagecompile`, the batch compiler places the .java and .class files in `d:\WebSphere\AppServer\temp\default_host\examples\pagecompile\security\login` directory.
   - **classPath**

     An optional parameter that is the fully-qualified path for the classes and Java archives that the compiled classes need. If those resources are in multiple paths, use the semicolon character (;) to separate the path names. You do not need to specify the Application Server JAR files on this parameter.
   - **libDirectory**

     The fully-qualified path to the Application Server ibmwebas.jar (contains the JSP batch compiler and related JSP classes) and servlet.jar (contains the Java Servlet 2.1 APIs). The default path is *product_installation_root*\lib.
   - **-v**

     An optional parameter that causes more trace and progress messages to be displayed.

All of the command parameters, except -v, are required.

Example

Suppose you want to precompile the JSP files associated with the `examples` application, one of the two applications installedwith the application server. If the JSP files are in the path:

`d:\WebSphere\AppServer\hosts\default_host\examples\web`

and you want the compiled files to be placed in:

`d:\WebSphere\AppServer\temp\default_host\examples\pagecompile`

the command would be (typed on a single line):

```
java com.ibm.servlet.jsp.http.pagecompile.jsp.tsx.batch.JspBatch -s
d:\WebSphere\AppServer\hosts\default_host\examples\web -t
d:\WebSphere\AppServer\temp\default_host\examples\pagecompile -c
d:\WebSphere\AppServer\hosts\default_host\examples\servlets;d:\devcntr\website -l
d:\WebSphere\AppServer\lib
```

4.2.2.2.4.2: Compiling JSP 1.0 files as a batch

To use the JSP batch compiler for JSP 1.0 files, enter the following command on a single line at an operating system command prompt:

```
JspBatchCompiler -adminNodeName <node name> [ -serverName <server name>
                 [-application <application name> [-filename <filename>]]]
                 [-keepgenerated <true|false>]
```

where:

- **adminNodeName**

  This is the name of the node as shown on the Adminstrative Console.

- **serverName**

  [Optional: may only be used if *adminNodeName* is set] This is the name of the Application Server in the WebSphere environment on which you wish to perform this action. Unless you have set up other servers, this will be "Default Server" [Note that from the command-line, you will need to include quote marks around the name of the server if that name comprises two or more words separated by spaces. You do not have to do this if you use the *batchcompiler.config* file described below.]

- **application**

  [Optional: may only be used if *serverName* is set] The name of a particular web application, should you wish to compile only those JSP files under that application.

- **filename**

  [Optional: may only be used if *application* is set] The name of a single file in the web application you selected above, should you wish to compile only a single JSP file in an application.

- **keepgenerated**

  [Optional] If set to "yes" this will keep the generated .java files used for compilation on your server. By default, this is set to "no" and the .java files are all erased after the class files have been compiled.

- **nameServiceHost**

  [Optional] If specified, this parameter and the **nameServicePort** parameter are used in a Model/Clone environment to designate the hostname and port number of the Admin Server to be used in accessing the WebSphere Application Server configuration.

- **nameServicePort**

  [Optional] If specified, this parameter and the **nameServiceHost** parameter are used in a Model/Clone environment to designate the hostname and port number of the Admin Server to be used in accessing the WebSphere Application Server configuration.

In lieu of specifying the parameters in a command line, you may specify them in the *batchcompile.config* file, located in the WebSphere Application Server bin directory. No quotation marks are necessary for any of the variables if you use this file. Any values you enter on the command-line will override the values specified inthe *batchcompile.config* file.

Example

Suppose you want to precompile the JSP files associated with the `examples` application, shipped with WebSphere Application Server. Issue the following command in the appserver bin directory:

```
D:\WebSphere\AppServer\bin>JspBatchCompiler.bat -adminNodeName mynode    -serverName "Default
Server" -application examples
```

You should receive the following response from the server

```
Server name: Default Server
Application Name: examples
   JSP version: 1.0
   docRoot: d:\WebSphere\AppServer\hosts\default_host\examples\web
   Application Classpath: d:\WebSphere\AppServer\hosts\default_host\examples\servlets;
   Application output dir: d:\WebSphere\AppServer/temp/default_host/examples
   URL: .jsp
   URL: .jsv
   URL: .jsw
Attempting to compile: d:\WebSphere\AppServer\hosts\default_host\examples\web\debug_error.jsp
Compilation successful
Attempting to compile: d:\WebSphere\AppServer\hosts\default_host\examples\web\HelloHTML.jsp
Compilation successful
    .       .       .Attempting to compile:
d:\WebSphere\AppServer\hosts\default_host\examples\web\StockQuoteWMLRequest.jspCompilation
successful
Attempting to compile:
d:\WebSphere\AppServer\hosts\default_host\examples\web\StockQuoteWMLResponse.jspCompilation
```

```
successful
```

If you look in the appserver temp directory, you should see a directory named examples. All of the compiled class files for the examples application will be in this directory.

4.2.2.3: Overview of JSP file content

JSP files have the extension .jsp. A JSP filecontains any combination of the following items. Click an item to learn about its syntax. To learn how to put it all together, see the Related information for examples, samples, and additional syntax references.

JSP syntax

Syntax format	Details
Directives	Use JSP directives (enclosed within <%@ and %>) to specify: • Scripting language being used • Interfaces a servlet implements • Classes a servlet extends • Packages a servlet imports • MIME type of the generated response  See Sun's JSP Syntax Referencefor JSP 1.1 syntax descriptions and examples.
Class-wide variable and method declarations	Use the <%! *declaration(s)* %> syntax to declareclass-wide variables and class-wide methods for the servlet class.
Inline Java code (scriptlets), enclosed within <% and %>	You can embed any valid Java language code inline withina JSP file between the <% and %> tags. Suchembedded code is called a *scriptlet*. If you do not specify the method directive, the generated code becomes the body of the service method. An advantage of embedding Java coding inline in JSP files is that the servlet does not have to be compiled in advance, and placed on the server. Thismakes it easier to quickly test servlet coding.
Variable text, specified using IBM extensions for variable data (JSP .91 or JSP 1.0)or Java expressions enclosed within <%= and %>	The IBM extensions are the more user-friendly approach to putting variable fields on your HTML pages. A second method for adding variable data is to specify a Java language expression that is resolved when the JSP file is processed. Use the JSP expression tags <%= and %>. The expression is evaluated, converted into a string, and displayed. Primitive types, such as int and float, areautomatically converted to string representation.
<BEAN> tag	Use the <BEAN> tag to create an instance of a bean that will be accessed elsewhere within the JSP file. Then use JSP tags to access the bean.
JSP tags for database access(JSP .91) or (JSP 1.1)	The IBM extensions make it easy for non-programmers to create Web pages that access databases.

HTML tags

A JSP file can contain any valid HTML tags. View article 0.70: What is HTML? for more informationon HTML. Refer to your favorite HTMLreference for a description of HTML tags.

<SERVLET> tags

Using the <SERVLET> tag is one method for embedding a servletwithin a JSP file.

NCSA tags

You might have legacy SHTML files that contain NCSA tags for server-side includes. If the IBM WebSphere Application Server Version 3.5 supports the NCSAtags in your SHTML files, you can convert the SHTML files to JSP files andretain the NCSA tags.

4.2.2.3.1: JSP syntax: JSP directives

Use JSP directives (enclosed within <%@ and %>) to specify:

- Scripting language being used
- Interfaces a servlet implements
- Classes a servlet extends
- Packages a servlet imports
- MIME type of the generated response

For more information on the JSP 1.1 technologies, view the Tomcatdocumentation at the Sun™ site.

The general syntax of the JSP directive is:

`<%@ directive_name ="value" %>`

where the valid directive names are:

- **language**

  The scripting language used in the file. At this time, the onlyvalid value and the default value is `java` (the Java programminglanguage). The scope of this directive is the JSP file.When used more than once, only the first occurrence of the directive issignificant. An example:

  ```
  <%@ language ="java" %>
  ```

- **method**

  The name of the method generated by the embedded Java code(scriptlet). The generated code becomes the body of the specifiedmethod name. The default method is `service`. When usedmore than once, only the first occurrence of the directive issignificant. An example:

  ```
  <%@ method ="doPost" %>
  ```

- **import**

  A comma-separated list of Java language package names or class names thatthe servlet imports. This directive can be specified multiple timeswithin a JSP file to import different packages. An example:

  ```
  <%@ import ="java.io.*,java.util.Hashtable" %>
  ```

- **content_type**

  The MIME type of the generated response. The default value is`text/html`. This information is used to generate the response header. When used more than once, only the first occurrence of thisdirective is significant. This directive can be used to specify the character set in which the pageis to be encoded. An example:

  ```
  <%@ content_type ="text/html; charset=iso-8859-1" %>
  ```

- **implements**

  A comma-separated list of Java language interfaces that the generatedservlet implements. You can use this directive more than once within aJSP file to implement different interfaces. An example:

  ```
  <%@ implements ="javax.servlet.http.HttpSessionContext" %>
  ```

- **extends**

  The name of the Java language class that the servlet extends. Theclass must be a valid class and does not have to be a servlet class.The scope of this directive is the JSP file. When used morethan once, only the first occurrence of the directive is significant.An example:

```
<%@ extends ="javax.servlet.http.HttpServlet" %>
```

4.2.2.3.2: JSP syntax: Class-wide variables and methods

Use the <SCRIPT> and </SCRIPT> tags to declareclass-wide variables and class-wide methods for the servlet class. Thegeneral syntax is:

```
<script runat=server>// code for class-wide variables and methods</script>
```

The attribute `runat=server` is required and indicates that thetag is for server-side processing. An example of specifying class-widevariables and methods:

```
<script runat=server>// class-wide variables    init i = 0;     String foo = "Hello";// class-wide
methods      private void foo() {// code for the method}  </script>
```

4.2.2.3.3: JSP syntax: Inline Java code (scriptlets)

You can embed any valid Java language code inlinebetween the <% and %> tags. Suchembedded code is called a *scriptlet*. If you do not specify the method directive, the generated code becomes the body of the service method.

The scriptlet can use a set of predefined variables thatcorrespond to essential servlet, output, and input classes:

- **request**

  The servlet request class defined byjavax.servlet.http.HttpServletRequest

- **response**

  The servlet response class defined byjavax.servlet.http.HttpServletResponse

- **out**

  The output writer class defined by java.io.PrintWriter. The content written to the writer is the client response.

- **in**

  The input reader class defined by java.io.BufferedReader

An example:

```
<%foo = request.getParameter("Name");out.println(foo);%>
```

Be sure to use the braces characters, { }, to enclose if, while, and for statements even if the scope contains a single statement. You can enclose the entire statement with a single scriptlet tag. However, if you use multiple scriptlet tags with the statement, be sure to place the opening brace character, {, in the same statement as the if, while, or for keyword. The following examples illustrate these points. The first example is the easiest.

```
<%for (int i = 0; i < 1; i++)   {   out.println("<P>This is written when " + i + " is < 1</P>");
}%>...<% for (int i = 0; i < 1; i++) {                                    %><%
out.println("<P>This is written when " + i + " is < 1</P>"); %><% }
%>...<% for (int i = 0; i < 1; i++)      {
%><%    out.println("<P>This is written when " + i + " is < 1</P>"); %><%    }
%>
```

4.2.2.3.4: JSP syntax: Java expressions

To specify a Java language expression that is resolvedwhen the JSP file is processed, use the JSP expression tags <%= and%>. The expression is evaluated, converted into a string,and displayed. Primitive types, such as int and float, areautomatically converted to string representation. In this example, foois the class-wide variable declared in the class-wide variables and methods example:

```
<p>Translate the greeting <%= foo %>.</p>
```

When the JSP file is served, the text reads: `Translate the greeting Hello.`

4.2.2.3.5: JSP syntax: useBean tag

The `<jsp:useBean>` tag locates a Bean or creates an instance of a Bean if it does not exist.

JavaBeans can be class files, serializedbeans, or dynamically generated by a servlet.A JavaBean can even be a servlet (that is, provide a service). If aservlet generates dynamic content and stores it in a bean, the bean can thenbe passed to a JSP file for use within the Web page defined by thefile.

See Sun's JSP Syntax Referencefor JSP 1.1 syntax descriptions and examples.

4.2.2.3.5.1: JSP syntax: <jsp:useBean> tag

Use the <jsp:useBean> tag to locate or instantiate a JavaBeans component. The syntax for the <jsp:useBean> tag is:

```
<jsp:useBean
  id="beanSomeName"
 scope="page|request|session|applicaton"
{ class="package_class" |
 type ="package_class" |
 class="package_class" type ="package_class" |
 beanName="{package.class| <%= expression%>}" type ="package_class"
}
{ />|
 > other elements
 </jsp:useBean>
}
```

See Sun's JSP syntax referencefor a description of the <jsp:useBean> attributes and examples.

4.2.2.3.5.1a: JSP .91 syntax: <BEAN> tag syntax

```
<bean name="bean_name" varname="local_bean_name"   type ="class_or_interface_name"
introspect="yes|no"    beanName="ser_filename" create="yes|no"   scope="request|session|userprofile"
></bean>
```

where the attributes are:

- **name**

  The name used to look up the bean in the appropriate scope (specified bythe scope attribute). For example, this might be the session key valuewith which the bean is stored. The value is case-sensitive.

- **varname**

  The name used elsewhere within the JSP file to refer to the bean.This attribute is optional. The default value is the value of the nameattribute. The value is case-sensitive.

- **type**

  The name of the bean class file. This name is used todeclare the bean instance in the code. The default value is the typeObject. The value is case-sensitive.

- **introspect**

  When the value is `yes`, the JSP processor examines all requestproperties and calls the set property methods (passed in the BeanInfo) thatmatch the request properties. The default value of this attribute is`yes`.

- **beanName**

  The name of the bean class file, the bean package name, or theserialized file (.ser file) that contains the bean. (This nameis given to the bean instantiator.) This attribute is used only whenthe bean is not present in the specified scope and the create attribute is setto `yes`. The value is case-sensitive.

  The path of the file must be specified in the Web application classpath.

- **create**

  When the value is `yes`, the JSP processor creates an instance ofthe bean if the processor does not find the bean within the specifiedscope. The default value is `yes`.

- **scope**

  The lifetime of the bean. This attribute is optional and thedefault value is `request`. The valid values are:

  - request - The bean is added to the request object by a servlet thatinvokes the JSP file using the APIs described in JSP API.If the bean is not part of the request context, the bean is created and stored in the request contextunless the create attribute is set to `no`.
  - session - If the bean is present in the current session, the bean isreused. If the bean is not present, it is created and stored as part ofthe session if the create attribute is set to `yes`.
  - userprofile - This attribute value is an IBM extension to JSP 0.91 and causes the user profile to be retrieved from the servlet requestobject, cast to the specified type, and introspected. If a type is notspecified, the default type is

    `com.ibm.websphere.UserProfile`

    .The create attribute is ignored.

4.2.2.3.5.2: JSP syntax: Accessing bean properties

After specifying the `<jsp:useBean>` tag, you can access the bean at any pointwithin the JSP file using the `<jsp:getProperty>` tag.

For a description of the `<jsp:getProperty>` tag attributesand for coding examples, see Sun's JSP Syntax Reference

4.2.2.3.5.2a: JSP .91 syntax: Accessing bean properties

After specifying the <BEAN> tag, you can access the bean at any pointwithin the JSP file. There are three methods for accessing beanproperties:

- Using a JSP scriptlet
- Using a JSP expression
- Using the <INSERT> tag (as described in the JSP .91 tags for variable data)

An example:

```
<!-- The bean declaration --> <bean name="foobar" type="FooClass" scope="request" >    <param
name="fooProperty" value="fooValue"></bean> <!-- Later in the file, some HTML content that includes
JSP syntax that calls a method of the bean --> <p>The name of the row is <%= foobar.getRowName()
%>.</p>
```

4.2.2.3.5.3: JSP syntax: Setting bean properties

You can set bean properties by using the `<jsp:setProperty>` tag. The `<jsp:setProperty>` tag specifies a list of properties and the corresponding values. The properties areset after the the bean is instantiated using the `<jsp:useBean>` tag.

You must declare the bean with `<jsp:useBean>` before you can set a property value.

See the Sun's JSP syntax referencefor `<jsp:setProperty>` syntax details and examples.

4.2.2.3.5.3a: JSP .91 syntax: Setting bean properties

You can set the bean properties by using the <PARAM> tag within the <BEAN> tag. The <PARAM> tag specifies a list of properties and the corresponding values. The properties areautomatically set in the bean using introspection. The properties areset once when the bean is instantiated. The <PARAM> tag syntax is:

`<PARAM name="property_name" value="property_value">`

This syntax is an IBM extension to the JSP 0.91 <PARAM> tag. The IBM syntax is consistent with the syntax of the <PARAM> tag used within the <SERVLET> and <APPLET> tags.

In addition to using the <param> tag to set bean properties,there are three other methods:

- Specifying query parameters when requesting the URL of the JSPfile that contains the bean. The introspect attribute must be set toyes. An example:

  `http://www.myserver.com/signon.jsp?name=jones&password=d13x`

  where the bean property name will be set to `jones`.
- Specifying the properties as parameters submitted through an HTML<FORM> tag. The JSP method directive must be set to`post`. The action attribute is set to the URL of the JSP filethat invokes the bean. The introspect attribute must be set toyes. An example:

  ```
  <form action="http://www.myserver.com/SearchSite.jsp" method="post">  <input type="text" name="Search for: ">  <input type="submit"></form>
  ```
- Using JSP syntax to set the bean property

4.2.2.3.5a: JSP .91 syntax: BEAN tags

Use the <BEAN> tag to create an instance of a bean that will beaccessed elsewhere within the JSP file. Then use JSP tags for variable data (such as the <INSERT> tag described later in this document) to access the bean.

The JavaBeans can be class files, serializedbeans, or dynamically generated by a servlet.A JavaBean can even be a servlet (that is, provide a service). If aservlet generates dynamic content and stores it in a bean, the bean can thenbe passed to a JSP file for use within the Web page defined by thefile.

4.2.2.3.6: Supported NCSA tag reference

The product supports the following NCSA tags through their use in JSP files:

- config
- echo var=*variable* (see below)
- exec
- filesize
- include
- lastmodified
- Commands for formatting size and date outputs

For the echo command, the product supports thestandard server-side include (SSI) environment variables and Common GatewayInterface (CGI) environment variables.

The SSI environment variables

Variable	Description
DATE_GMT	The current date and local time zone in Greenwich mean time (GMT)
DATE_LOCAL	The current date and local time zone
DOCUMENT_NAME	The current filename
DOCUMENT_URI	The path to the document (such as, /docs/tutorials/index.shtml)
QUERY_STRING_UNESCAPED	The unescaped version of any search query the client sent, with all shellspecial characters escaped with the \ character
LAST_MODIFIED	The last date the current document was changed

CGI environment variables

Variable	Description
AUTH_TYPE	The protocol-specific authentication method used to validate the user, ifthe server supports user authentication and the script is protected
CONTENT_LENGTH	The length of the content, as specified by the remote host
CONTENT_TYPE	The data content type for queries that have information attached(such as HTTP POST and PUT)
GATEWAY_INTERFACE	The revision level of the CGI specification to which the server complies
PATH_INFO	The extra path information given by the client in this request. Theextra information follows the virtual pathname of the CGI script.
PATH_TRANSLATED	The server provides a translated version of PATH_INFO, which takes the pathand performs any virtual-to-physical mapping.
QUERY_STRING	The information that follows the ? symbol in the URL request for a script
REMOTE_HOST	The hostname of the remote host sending the request. If the serverdoes not have this information, the server should set REMOTE_ADDR and leaveREMOTE_HOST unset.
REMOTE_ADDR	The IP address of the remote host sending the request
REMOTE_IDENT	If the HTTP server supports RFC 931 identification, the remote usernameretrieved from the server

REMOTE_USER	The username used for authentication, if the server supports userauthentication and the script is protected
REQUEST_METHOD	The method with which this request was made. Methods include HTTP, GET, HEAD,POST, and so on
SCRIPT_NAME	The virtual path to the script being run. This variable is used forself-referencing URLs
SERVER_NAME	The IP address, hostname, or Domain Name Server (DNS) alias of the server
SERVER_PORT	The port number to which the request was sent
SERVER_PROTOCOL	The name and revision level of the protocol used to format this request
SERVER_SOFTWARE	The name and version of the server answering the request

4.2.2.3.7: IBM extensions to JSP syntax

Refer to the Sun JSP Specification for the base JavaServer Pages (JSP) APIs. IBMWebSphere Application Server Version 3.5 provided several extensions to the base APIs.The backward compatibility of the JSP 1.1 specification to JSP 1.0 allows users to invoke these APIs without modification.

The extensions belong to these categories:

Extension	Use
Syntax for variable data	Put variable fields in JSP files and have servlets and JavaBeans dynamicallyreplace the variables with values from a database when the JSP output is returned tothe browser
Syntax for database access	Add a database connection to a Web page and then use that connection to query or updatethe database. The user ID and password for the database connection can be provided by theuser at request time, or can be hardcoded within the JSP file.

Scope of variables: Because the values specified by syntax apply onlyto the JSP file in which thesyntax is embedded, identifiers and other tag data can be accessed only withinthe page.

See the Related information for syntax details.

4.2.2.3.7.1: JSP syntax: Tags for variable data

The variable data syntax enables you to put variable fields in your JSP file and have your servlets and JavaBeansdynamically replace the variables with values from a database when the JSP output is returned to the browser.

The table summarizes the tags. Click a tag to link to its syntax description.

Goal	Tag	Details
Get the value of a bean to display in a JSP.	<tsx:getProperty>	This IBM extension of the Sun JSP <jsp:getProperty> tag implements all of the <jsp:getProperty> function and adds the ability to introspect a database bean that was created using the IBM extension <tsx:dbquery> or <tsx:dbmodify>. **Note:** You cannot assign the value from this tag toa variable. The value, generated as output from this tag, is displayed in the Browser window.
Repeat a block of HTML tagging that contains the <tsx:getProperty> syntax and the HTML tags for formatting content.	<tsx:repeat>	Use the <tsx:repeat> syntax to iterate over a database query results set. The <tsx:repeat> syntax iterates from the start value to the end value until one of the following conditions is met: ● The end value is reached. ● An exception is thrown. The output of a <tsx:repeat> block is buffered until the block completes. If an exception is thrown before a block completes, no output is written for that block.

4.2.2.3.7.1.1: JSP syntax: <tsx:getProperty> tag syntax and examples

```
<tsx:getProperty name="bean_name"  property="property_name" />
```

where:

- **name**

  The name of the JavaBean declared by the `id` attribute of a <tsx:dbquery> syntax within the JSP file. See <tsx:dbquery> for an explanation. The value of this attribute is case-sensitive.

- **property**

  The property of the bean to access for substitution. The value ofthe attribute is case-sensitive and is the locale-independent name of theproperty.

Examples

```
<tsx:getProperty name="userProfile" property="username" /><tsx:getProperty name="request"
property=request.getParameter("corporation") />
```

In most cases, the value of the property attribute will be just theproperty name. However, to access the request bean or access a property of a property(sub-property), you specify the full form of the property attribute.The full form also gives you the option to specify an index for indexedproperties. The optional index can be a constant (such as 2) or anindex like the one described in <tsx:repeat>. Some examples of using the full form of the property attribute:

```
<tsx:getProperty name="staffQuery" property=address(currentAddressIndex) /><tsx:getProperty
name="shoppingCart" property=items(4).price /><tsx:getProperty name="fooBean"
property=foo(2).bat(3).boo.far />
```

4.2.2.3.7.1.2: JSP syntax: <tsx:repeat> tag syntax

`<tsx:repeat index=`*`name`* `start="`*`starting_index`*`" end="`*`ending_index`*`"></tsx:repeat>`

where:

- **index**

  An optional name used to identify the index of this repeat block.The value is case-sensitive and its scope is the JSP file.

- **start**

  An optional starting index value for this repeat block. The defaultis 0.

- **end**

  An optional ending index value for this repeat block. The maximumvalue is 2,147,483,647. If the value of the **end** attribute is less thanthe value of the **start** attribute, the **end** attribute is ignored.

4.2.2.3.7.1.2a: JSP syntax: The repeat tag results set and the associated bean

The <tsx:repeat> iterates over a results set. The results set is contained within a JavaBean. The bean can be a static bean (for example, a bean created by using the IBM WebSphere Studio database wizard) or a dynamically generated bean (for example, a bean generated by the <tsx:dbquery> syntax). The following table is a graphic representation of the contents of a bean, myBean:

	col1	col2	col3
row0	friends	Romans	countrymen
row1	bacon	lettuce	tomato
row2	May	June	July

Some observations about the bean:

- The column names in the database table become the property names of the bean. The section <tsx:dbquery> describes a technique for mapping the column names to different property names.
- The bean properties are indexed. For example, myBean.get(Col1(row2)) returns `May`.
- The query results are in the rows. The <tsx:repeat> iterates over the rows (beginning at the start row).

The following table compares using the <tsx:repeat> to iterate over static bean versus a dynamically generated bean:

Static Bean Example	**<tsx:repeat> Bean Example**

myBean.class
`// Code to get a connection// Code to get the data  Select * from myTable;// Code to close the connection`
JSP file
`<tsx:repeat index=abc>  <tsx:getPropery name="myBean"  property="col1(abc)" /></tsx:repeat>`
ⓘ
• The bean (myBean.class) is a static bean.
• The method to access the bean properties is myBean.get(*property*(*index*)).
• You can omit the property index, in which case the index of the enclosing <tsx:repeat> is used. You can also omit the index on the <tsx:repeat>.
• The <tsx:repeat> iterates over the bean properties row by row, beginning with the start row.
	JSP file
	`<tsx:dbconnect id="conn"userid="alice"passwd="test"url="jdbc:db2:sample"driver="COM.ibm.db2.jdbc.app.DB2Dri`
	`><tsx:dbquery id="dynamic" connection="conn" >  Select * from myTable;</tsx:dbquery><tsx:repeat index=abc>`
	`name="dynamic"  property="col1(abc)" /></tsx:repeat>`
	ⓘ
	• The bean (dynamic) is generated by the <tsx:dbquery> and does not exist until the syntax is executed.
	• The method to access the bean properties is dynamic.getValue("*property*", *index*).
	• You can omit the property index, in which case the index of the enclosing <tsx:repeat> is used. You can also omit the index on the <tsx:repeat>.
	• The <tsx:repeat> syntax iterates over the bean properties row by row, beginning with the start row.

Implicit and explicit indexing

Examples 1, 2, and 3 show how to use the <tsx:repeat>. Theexamples produce the same output if all indexed properties have 300 or fewerelements. If there are more than 300 elements, Examples 1 and 2 willdisplay all elements, while Example 3 will show only the first 300elements.

Example 1 shows implicit indexing with the default start and default endindex. The bean with the smallest number of indexed properties restricts the number of times the loop will repeat.

```
<table><tsx:repeat>  <tr><td><tsx:getProperty name="serviceLocationsQuery" property="city" /></tr></td>  <tr><td><tsx:getProperty
name="serviceLocationsQuery" property="address" /></tr></td>  <tr><td><tsx:getProperty name="serviceLocationsQuery"
property="telephone" /></tr></td></tsx:repeat></table>
```

Example 2 shows indexing, starting index, and ending index:

```
<table><tsx:repeat index=myIndex start=0 end=2147483647>  <tr><td><tsx:getProperty name="serviceLocationsQuery"
property=city(myIndex) /></tr></td>  <tr><td><tsx:getProperty name="serviceLocationsQuery" property=address(myIndex) /></tr></td>
<tr><td><tsx:getProperty name="serviceLocationsQuery" property=telephone(myIndex) /></tr></td></tsx:repeat></table>
```

Example 3 shows explicit indexing and ending index with implicit startingindex. Although the index attribute is specified, the indexed propertycity can still be implicitly indexed because the `(myIndex)` is not required.

```
<table><tsx:repeat index=myIndex end=299>  <tr><td><tsx:getProperty name="serviceLocationsQuery" property="city" /t></tr></td>
<tr><td><tsx:getProperty name="serviceLocationsQuery" property="address(myIndex)" /></tr></td>  <tr><td><tsx:getProperty
name="serviceLocationsQuery" property="telephone(myIndex)" /></tr></td></tsx:repeat></table>
```

Nesting <tsx:repeat> blocks

You can nest <tsx:repeat> blocks. Each block is separatelyindexed. This capability is useful for interleaving properties on twobeans, or properties that have sub-properties. In the example, two<tsx:repeat> blocks are nested to display the list of songs on each compactdisc in the user's shopping cart.

```
<tsx:repeat index=cdindex>  <h1><tsx:getProperty name="shoppingCart" property=cds.title /></h1>  <table>  <tsx:repeat>
<tr><td><tsx:getProperty name="shoppingCart" property=cds(cdindex).playlist />    </td></tr>  </table>
</tsx:repeat></tsx:repeat>
```

4.2.2.3.7.2: JSP syntax: Tags for database access

Beginning with IBM WebSphere Application Server Version 3.x, the JSP 1.0 supportwas extended to provide syntaxfor database access. The syntax makes it simple to add a database connectionto a Web page and then use that connection to query or update the database.The user ID and password for the database connection can be provided by theuser at request-time or hard coded within the JSP file.

The table summarizes the tags. Click a tag to link to its syntax description.

Goal	Tag	Details and examples
Specify information needed to make a connection to a JDBC or an ODBC database.	<tsx:dbconnect>	The <tsx:dbconnect> syntax does not establish the connection. Instead,the <tsx:dbquery> and <tsx:dbmodify> syntax are used to referencea <tsx:dbconnect> in the same JSP file and establish the connection. When the JSP file is compiled into a servlet, the Java processor addsthe Java coding for the <tsx:dbconnect> syntax to the servlet'sservice() method, which means a new database connection is created for eachrequest for the JSP file.
Avoid hard coding the user ID and password in the<tsx:dbconnect>.	<tsx:userid> and <tsx:passwd>	Use the <tsx:userid> and <tsx:passwd> to acceptuser input for the values and then add that data to the request object. The request objectcan be accessed by a JSP file (such as the JSPEmployee.jsp example) that requests the databaseconnection. The <tsx:userid> and <tsx:passwd> must be used within a<tsx:dbconnect> tag.
Establish a connection to a database, submit database queries, and return the results set.	<tsx:dbquery>	The <tsx:dbquery>: 1. References a <tsx:dbconnect> in the same JSP file and uses the information it provides to determine the database URL and driver. The user ID and password are also obtained from the <tsx:dbconnect> if those values are provided in the <tsx:dbconnect>. 2. Establishes a new connection 3. Retrieves and caches data in the results object 4. Closes the connection (releases the connection resource)

Establish a connection to a database and then add records to a database table.	<tsx:dbmodify>	The <tsx:dbmodify>: 1. References a <tsx:dbconnect> in the same JSP file and uses the information provided by that to determine the database URL and driver. The user ID and password are also obtained from the <tsx:dbconnect> if those values are provided in the <tsx:dbconnect>. 2. Establishes a new connection 3. Updates a table in the database 4. Closes the connection (releases the connection resource) **Examples:** Basic example
Display query results.	<tsx:repeat> and <tsx:getProperty>	The <tsx:repeat> loops through each of the rows in the query results.The <tsx:getProperty> uses the query results object (for the <tsx:dbquery>syntax whose identifier is specified by the <tsx:getProperty> bean attribute)and the appropriate column name (specified by the <tsx:getProperty> propertyattribute) to retrieve the value. **Note:** You cannot assign the value from the <tsx:getProperty> tag toa variable. The value, generated as output from this tag, is displayed in the Browser window. **Examples:** Basic example

4.2.2.3.7.2.1: JSP syntax: <tsx:dbconnect> tag syntax

```
<tsx:dbconnect id="connection_id"       userid="db_user" passwd="user_password"
url="jdbc:subprotocol:database" driver="database_driver_name"
jndiname="JNDI_context/logical_name"></tsx:dbconnect>
```

where:

- **id**

  A required identifier. The scope is the JSP file. This identifier is referenced by the connection attribute of a <tsx:dbquery> tag.

- **userid**

  An optional attribute that specifies a valid user ID for the database to be accessed. If specified, this attribute and its value are added to the request object.

  Although the userid attribute is optional, the userid must be provided. See <tsx:userid> and <tsx:passwd> for an alternative to hard coding this information in the JSP file.

- **passwd**

  An optional attribute that specifies the user password for the userid attribute. (This attribute is not optional if the userid attribute is specified.) If specified, this attribute and its value are added to the request object.

  Although the passwd attribute is optional, the password must be provided. See <tsx:userid> and <tsx:passwd> for an alternative to hard coding this attribute in the JSP file.

- **url** and **driver**

  To establish a database connection, the URL and driver must be provided.

  The Application Server Version 3 supports connection to JDBC databases and ODBC databases.

JDBC database

For a JDBC database, the URL consists of the following colon-separated elements: jdbc, the sub-protocol name, and the name of the database to be accessed. An example for a connection to the Sample database included with IBM DB2 is:

```
url="jdbc:db2:sample"driver="COM.ibm.db2.jdbc.app.DB2Driver"
```

ODBC database

Use the Sun JDBC-to-ODBC bridge driver included in the Java Development Kit (JDK) oranother vendor's ODBC driver.

The url attribute specifies the location of the database. The driver attribute specifies the name of the driver to be used to establish the database connection.

If the database is an ODBC database, you can use an ODBC driver or the Sun JDBC-to-ODBC bridge included with the JDK. If you want to use an ODBC driver, refer to the driver documentation for instructions on specifying the database location (the url attribute) and the driver name.

In the case of the bridge, the url syntax is jdbc:odbc:*database*. An example is:

```
url="jdbc:odbc:autos"driver="sun.jdbc.odbc.JdbcOdbcDriver"
```

To enable the Application Server to access the ODBC database, use the ODBC Data Source Administrator to add the ODBC data source to the System DSN configuration. To access the

ODBC Administrator, click the ODBC icon on the Windows NT Control Panel.

- **jndiname**

  An optional attribute that identifies a valid context in the Application Server JNDI naming context and the logical name of the data source in that context. The context is configured by the Web administrator using an administrative client such as the WebSphere Administrative Console.

  If the jndiname is specified, the JSP processor ignores the driver and url attributes on the <tsx:dbconnect> tag.

An empty element (such as <url></url>) is valid.

4.2.2.3.7.2.2: JSP syntax: <tsx:userid> and <tsx:passwd> tag syntax

```
<tsx:dbconnect id="connection_id"       <font color="red"><userid></font><tsx:getProperty
name="request" property=request.getParameter("userid") /><font color="red"></userid></font>   <font
color="red"><passwd></font><tsx:getProperty name="request" property=request.getParameter("passwd")
/><font color="red"></passwd></font>   url="protocol:database_name:database_table"
driver="JDBC_driver_name"> </tsx:dbconnect>
```

where:

- **<tsx:getProperty>**

  This syntax is a mechanism for embedding variable data. See JSP syntax for variable data.

- **userid**

  This is a reference to the request parameter that contains the userid. The parameter must have already been added to the request object that was passed to this JSP file. The attribute and its value can be set in the request object using an HTML form or a URL query string to pass the user-specified request parameters.

- **passwd**

  This is a reference to the request parameter that contains the password. The parameter must have already been added to the request object that was passed to this JSP. The attribute and its value can be set in the request object using an HTML form or a URL query string to pass user-specified values.

4.2.2.3.7.2.3: JSP syntax: <tsx:dbquery> tag syntax

```
<%-- SELECT commands and (optional) JSP syntax can be placed within the tsx:dbquery. --%><%-- Any other syntax, including HTML comments, are not valid. --%><tsx:dbquery id="query_id" connection="connection_id" limit="value" ></tsx:dbquery>
```

where:

- **id**

  The identifier of this query. The scope is the JSP file. This identifier is used to reference the query, for example, from the <tsx:getProperty> to display query results.

  The id becomes the name of a bean that contains the results set. The bean properties are dynamic and the property names are the names of the columns in the results set. If you want different column names, use the SQL keyword for specifying an alias on the SELECT command. In the following example, the database table contains columns named FNAME and LNAME, but the SELECT statement uses the AS keyword to map those column names to FirstName and LastName in the results set:

  ```
  Select FNAME, LNAME AS FirstName, LastName from Employee where FNAME='Jim'
  ```

- **connection**

  The identifier of a <tsx:dbconnect> in this JSP file. That <tsx:dbconnect> provides the database URL, driver name, and (optionally) the user ID and password for the connection.

- **limit**

  An optional attribute that constrains the maximum number of records returned by a query. If the attribute is not specified, no limit is used. In such a case, the effective limit is determined by the number of records and the system caching capability.

- SELECT command and JSP syntax

  The only valid SQL command is SELECT because the <tsx:dbquery> must return a results set. Refer to your database documentation for information about the SELECT command. See other sections of this document for a description of JSP syntax for variable data and inline Java code.

4.2.2.3.7.2.3a: Example: JSP syntax: <tsx:dbquery> tag syntax

In the following example, a database is queried for data about employees in a specified department. The department is specified using the <tsx:getProperty> to embed a variable data field. The value of the field is based on user input.

```
<tsx:dbquery id="empqs" connection="conn" >select * from Employee where WORKDEPT='<tsx:getProperty
name="request" property=request.getParameter("WORKDEPT") />'</tsx:dbquery>
```

4.2.2.3.7.2.4: JSP syntax: <tsx:dbmodify> tag syntax

```
<%-- Any valid database update commands can be placed within the DBMODIFY tag. --><%-- Any other syntax, including HTML comments, are not valid. --><tsx:dbmodify connection="connection_id"></tsx:dbmodify>
```

where:

- **connection**

  The identifier of a <DBCONNECT> tag in this JSP file. The <DBCONNECT> tag provides the database URL, driver name, and (optionally) the user ID and password for the connection.

- Database commands

  Valid database commands. Refer to your database documentation for details

4.2.2.3.7.2.4a: Example: JSP syntax: <tsx:dbmodify> tag syntax

In the following example, a new employee record is added to a database. The values of the fields are based on user input from this JSP and referenced in the database commands using <tsx:getProperty>.

```
<tsx:dbmodify connection="conn" >insert into EMPLOYEE
(EMPNO,FIRSTNME,MIDINIT,LASTNAME,WORKDEPT,EDLEVEL)values('<tsx:getProperty name="request"
property=request.getParameter("EMPNO") />','<tsx:getProperty name="request"
property=request.getParameter("FIRSTNME") />','<tsx:getProperty name="request"
property=request.getParameter("MIDINIT") />','<tsx:getProperty name="request"
property=request.getParameter("LASTNAME") />','<tsx:getProperty name="request"
property=request.getParameter("WORKDEPT") />',<tsx:getProperty name="request"
property=request.getParameter("EDLEVEL") />)</tsx:dbmodify>
```

4.2.2.3.7.2.5a: Example: JSP syntax: <tsx:repeat> and <tsx:getProperty> tags

```
<tsx:repeat><tr>        <td><tsx:getProperty name="empqs" property="EMPNO" />   <tsx:getProperty
name="empqs" property="FIRSTNME" />    <tsx:getProperty name="empqs" property="WORKDEPT" />
<tsx:getProperty name="empqs" property="EDLEVEL" />     </td></tr></tsx:repeat>
```

4.2.2.3.8: IBM extensions to JSP .91 syntax

Refer to the Sun JSP .91 specification for the base JavaServer Pages (JSP)APIs. IBM WebSphere Application Server provides several extensions to the baseAPIs.

For JSP .91, the extensions belong to these categories:

Extension	Use
Syntax for variable data	Put variable fields in JSP files and have servlets and JavaBeans dynamically replace the variables with values from a database when the JSP output is returned to the browser.
Syntax for database access	Add a database connection to a Web page and then use that connection to query or update the database. The user ID and password for the database connection can be provided by the user at request time, or can be hardcoded within the JSP file.

Scope of variables: Because the values specified by syntax apply only to the JSP file in which the syntax is embedded, identifiers and other tag data can be accessed only within the page.

See the Related information for syntax details.

4.2.2.3.8.1: JSP .91 syntax: Tags for variable data

The variable data syntax enables you to put variable fields on your HTML page and have your servlets and JavaBeans dynamically replace the variables with values from a database when the JSP output is returned to the browser.

The table summarizes the tags. Click a tag to link to its syntax description.

Goal	Tag	Details
Embed variables in a JSP file	<INSERT>	This is the base tag for specifyingvariable fields.
Repeating a block of HTML tagging that contains the <INSERT> tags and the HTML tags for formatting content	<REPEAT>	Use the <REPEAT> tag to iterate over a database query results set. The <REPEAT> tag iterates from the start value to the end value until one of the following conditions is met: ● The end value is reached. ● An ArrayIndexOutofBoundsException is thrown. The output of a <REPEAT> block is buffered until the block completes. If an exception is thrown before a block completes, no output is written for that block.

The above tags are designed to pass entact through HTML authoringtools. Each tag has a corresponding end tag. Each tag iscase-insensitive, but some of the tag attributes are case-sensitive.

4.2.2.3.8.1.1: JSP .91 syntax: <INSERT> tag syntax

```
<insert requestparm=pvalue requestattr=avalue bean=name
property=property_name(optional_index).subproperty_name(optional_index)
default=value_when_null></insert>
```

where:

- **requestparm**

  The parameter to access within the request object. This attributeis case-sensitive and cannot be used with the bean and propertyattributes.

- **requestattr**

  The attribute to access within the request object. The attributewould have been set using the setAttribute method. This attribute iscase-sensitive and cannot be used with the bean and propertyattributes.

- **bean**

  The name of the JavaBean declared by a <BEAN> tag within the JSPfile. The value of this attribute is case-sensitive.

  When the bean attribute is specified but the property attribute is notspecified, the entire bean is used in the substitution. For example, ifthe bean is type String and the property is not specified, the value of thestring is substituted.

- **property**

  The property of the bean to access for substitution. The value ofthe attribute is case-sensitive and is the locale-independent name of theproperty. This attribute cannot be used with the requestparm andrequestattr attributes.

- **default**

  An optional string to display when the value of the bean property isnull. If the string contains more than one word, the string must beenclosed within a pair of double quotes (such as "HelpDesk number").The value of this attribute is case-sensitive. If a value is notspecified, an empty string is substituted when the value of the property isnull.

Use the alternate syntax instead if you need to embed the INSERT tag withinanother HTML tag.

4.2.2.3.8.1.1a: JSP .91 syntax: Alternate syntax for the <INSERT> tag

The HTML standard does not permit embedding HTML tags within HTML tags. Consequently, you cannot embed the <INSERT> tag within another HTML tag, for example, the anchor tag (<A>). Instead, use the alternate syntax.

To use the alternate syntax:

1. Use the <INSERT> and </INSERT> to enclose the HTML tag inwhich substitution is to take place.
2. Specify the bean and property attributes:
    - To specify the bean and property attributes, use the form:

      `$(bean=`*b* `property=`*p* `default=`*d*`)`

      where *b*, *p*, and *d* are values as described forthe <INSERT> tag.
    - To specify the requestparm attribute, use the form

      `$(requestparm=`*r* `default=`*d*`)`

      where *r* and *d* are values as described for the <INSERT> tag.
    - To specify the requestattr attribute, use the form

      `$(requestattr=`*r* `default=`*d*`)`

      where *r* and *d* are values as described for the <INSERT> tag.

4.2.2.3.8.1.1b: Example: JSP .91 syntax: INSERT tag syntax

Regular syntax

```
<insert bean=userProfile property=username></insert><insert requestparm=company default="IBM
Corporation"></insert><insert requestattr=ceo default="Company CEO"></insert><insert
bean=userProfile property=lastconnectiondate.month></insert>
```

In most cases, the value of the property attribute will be just theproperty name. However, you access a property of a property(sub-property) by specifying the full form of the property attribute.The full form also gives you the option to specify an index for indexedproperties. The optional index can be a constant (such as 2) or anindex like the one described in <REPEAT> tag. Some examples of using the full form of the property attribute:

```
<insert bean=staffQuery property=address(currentAddressIndex)></insert><insert bean=shoppingCart
property=items(4).price></insert><insert bean=fooBean property=foo(2).bat(3).boo.far></insert>
```

Alternate syntax

```
<insert>  <img src=$(bean=productAds property=sale default=default.gif)></insert> <insert>  <a
href="http://www.myserver.com/map/showmap.cgi?country=$(requestparm=country
default=usa)&city$(requestparm=city default="Research Triangle Park")  &email=$(bean=userInfo
property=email)>Show map of city</a></insert>
```

4.2.2.3.8.1.2: JSP .91 syntax: <REPEAT> tag syntax

`<repeat index=`*`name`* `start=`*`starting_index`* `end=`*`ending_index`*`></repeat>`

where:

- **index**

  An optional name used to identify the index of this repeat block.The value is case-sensitive and its scope is the JSP file.

- **start**

  An optional starting index value for this repeat block. The defaultis 0.

- **end**

  An optional ending index value for this repeat block. The maximumvalue is 2,147,483,647. If the value of the end attribute is less thanthe value of the start attribute, the end attribute is ignored.

4.2.2.3.8.1.2a: JSP .91 syntax: <REPEAT> tag results set and the associated bean

The <REPEAT> tag iterates over a results set. The results set is contained within a JavaBean. The bean can be a static bean (for example, a bean created by using the IBM WebSphere Studio database wizard) or a dynamically generated bean (for example, a bean generated by the<DBQUERY> tag). The following table is a graphic representation of the contents of a bean, myBean:

	col1	col2	col3
row0	friends	Romans	countrymen
row1	bacon	lettuce	tomato
row2	May	June	July

Some observations about the bean:

- The column names in the database table become the property names of the bean. The section <DBQUERY> tag describes a technique for mapping the column names to different property names.
- The bean properties are indexed. For example, myBean.get(Col1(row2)) returns `May`.
- The query results are in the rows. The <REPEAT> tag iterates over the rows (beginning at the start row).

The following table compares using the <REPEAT> tag to iterate over static bean versus a dynamically generated bean:

Static Bean Example	**<DBQUERY> Bean Example**

myBean.class
`// Code to get a connection// Code to get the data  Select * from myTable;// Code to close the connection`
JSP file
`<repeat index=abc>  <insert bean="myBean"  property="col1(abc)">  </insert></repeat>`
- The bean (myBean.class) is a static bean.
- The method to access the bean properties is myBean.get(*property*(*index*)).
- You can omit the property index, in which case the index of the enclosing <REPEAT> tag is used. You can also omit the index on the <REPEAT> tag.
- The <REPEAT> tag iterates over the bean properties row by row, beginning with the start row.
	JSP file
	`<dbconnect id="conn"userid="alice"passwd="test"url="jdbc:db2:sample"driver="COM.ibm.db2.jdbc.app.DB2Driver"></dbconnect id="dynamic" connection="conn" >  Select * from myTable;</dbquery><repeat index=abc>  <insert bean="dynamic" property="col1(abc)">  </insert></repeat>`
	- The bean (dynamic) is generated by the <DBQUERY> tag and does not exist until the tag is executed.
	- The method to access the bean properties is dynamic.getValue("*property*", *index*).
	- You can omit the property index, in which case the index of the enclosing <REPEAT> tag is used. You can also omit the index on the <REPEAT> tag.
	- The <REPEAT> tag iterates over the bean properties row by row, beginning with the start row.

Implicit and explicit indexing

Examples 1, 2, and 3 show how to use the <REPEAT> tag. Theexamples produce the same output if all indexed properties have 300 or fewerelements. If there are more than 300 elements, Examples 1 and 2 willdisplay all elements, while Example 3 will show only the first 300elements.

Example 1 shows implicit indexing with the default start and default endindex. The bean with the smallest number of indexed properties restricts the number of times the loop will repeat.

```
<table><repeat>  <tr><td><insert bean=serviceLocationsQuery property=city></insert></tr></td>  <tr><td><insert
bean=serviceLocationsQuery property=address></insert></tr></td>  <tr><td><insert bean=serviceLocationsQuery
property=telephone></insert></tr></td></repeat></table>
```

Example 2 shows indexing, starting index, and ending index:

```
<table><repeat index=myIndex start=0 end=2147483647>  <tr><td><insert bean=serviceLocationsQuery
property=city(myIndex)></insert></tr></td>  <tr><td><insert bean=serviceLocationsQuery
property=address(myIndex)></insert></tr></td>  <tr><td><insert bean=serviceLocationsQuery
property=telephone(myIndex)></insert></tr></td></repeat></table>
```

The JSP compiler for the Application Server Version 3 is designed to prevent the ArrayIndexOutofBoundsException with explicit indexing. Consequently, you do not need to place JSP variable data syntax before the <INSERT> tag to check the validity of the index.

Example 3 shows explicit indexing and ending index with implicit startingindex. Although the index attribute is specified, the indexed propertycity can still be implicitly indexed because the `(myIndex)` is not required.

```
<table><repeat index=myIndex end=299>  <tr><td><insert bean=serviceLocationsQuery property=city></insert></tr></td>
<tr><td><insert bean=serviceLocationsQuery property=address(myIndex)></insert></tr></td>  <tr><td><insert
bean=serviceLocationsQuery property=telephone(myIndex)></insert></tr></td></repeat></table>
```

Nesting <REPEAT> tags

You can nest <REPEAT> blocks. Each block is separatelyindexed. This capability is useful for interleaving properties on twobeans, or properties that have sub-properties. In the example, two<REPEAT> blocks are nested to display the list of songs on each compactdisc in the user's shopping cart.

```
<repeat index=cdindex>  <h1><insert bean=shoppingCart property=cds.title></insert></h1>  <table>  <repeat>    <tr><td><insert
bean=shoppingCart property=cds(cdindex).playlist></insert>    </td></tr>  </table>  </repeat></repeat>
```

4.2.2.3.8.2: JSP .91 syntax: JSP tags for database access

The Application Server Version 3.5 extends JSP 0.91 support by providing a set of tags for database access. These HTML-like tags make it simple to add a database connection to a Web page and then use that connection to query or update the database. The user ID and password for the database connection can be provided by the user at request time or hardcoded within the JSP file.

The table summarizes the tags. Click a tag to link to its syntax description.

Goal	Tag	Details and examples
Specify information needed to make a connection to a JDBC or an ODBC database	<DBCONNECT>	The <DBCONNECT> tag does not establish the connection. Instead, the <DBQUERY> and <DBMODIFY> tags are used to reference a <DBCONNECT> tag in the same JSP file and establish the connection. When the JSP file is compiled into a servlet, the Java processor adds the Java coding for the <DBCONNECT> tag to the servlet's service() method, which means a new database connection is created for each request for the JSP file. **Examples:** Employee.jsp example
Avoid hard coding the user ID and password in the <DBCONNECT> tag	<USERID> and <PASSWD>	Use the <USERID> and <PASSWD> tags to accept user input for the values and then add that data to the request object where it can be accessed by a JSP file (such as the Employee.jsp example) that requests the database connection. The <USERID> and <PASSWD> tags must be used within a <DBCONNECT> tag. **Examples:** None
Establish a connection to a database, submit database queries, and return the results set.	<DBQUERY>	The <DBQUERY> tag: 1. References a <DBCONNECT> tag in the same JSP file and uses the information provided by that tag to determine the database URL and driver. The user ID and password are also obtained from the <DBCONNECT> tag if those values are provided in the <DBCONNECT> tag. 2. Establishes a new connection 3. Retrieves and caches data in the results object 4. Closes the connection (releases the

		connection resource) **Examples:** Basic example Employee.jsp EmployeeRepeatResults.jsp
Establish a connection to a database and then add records to a database table.	<DBMODIFY>	The <DBMODIFY> tag: 1. References a <DBCONNECT> tag in the same JSP file and uses the information provided by that tag to determine the database URL and driver. The user ID and password are also obtained from the <DBCONNECT> tag if those values are provided in the <DBCONNECT> tag. 2. Establishes a new connection 3. Updates a table in the database 4. Closes the connection (releases the connection resource) **Examples:** Basic example EmployeeRepeatResults.jsp
Display query results	<REPEAT> and <INSERT> tags	The <REPEAT> tag loops through each of the rows in the query results. The <INSERT> tag uses the query results object (for the <DBQUERY> tag whose identifier is specified by the <INSERT> bean attribute) and the appropriate column name (specified by the <INSERT> property attribute) to retrieve the value. **Examples:** Basic example

4.2.2.3.8.2.1: JSP .91 syntax: <DBCONNECT> tag syntax

```
<dbconnect id="connection_id"   userid="db_user" passwd="user_password"
url="jdbc:subprotocol:database" driver="database_driver_name"   jndiname="JNDI_context/logical_name"
xmlref="configuration_file"></dbconnect>
```

where:

- **id**

  A required identifier for this tag. The scope is the JSP file. This identifier is referenced by the connection attribute of the <DBQUERY> tag.

- **userid**

  An optional attribute that specifies a valid user ID for the database to be accessed. If specified, this attribute and its value are added to the request object.

  Although the userid attribute is optional, the userid must be provided. See <USERID> and <PASSWD> for an alternative to hardcoding this information in the JSP file.

- **passwd**

  An optional attribute that specifies the user password for the userid. (This attribute is not optional if the userid attribute is specified.) If specified, this attribute and its value are added to the request object.

  Although the passwd attribute is optional, the password must be provided. See <USERID> and <PASSWD> for an alternative to hardcoding this attribute in the JSP file.

- **url** and **driver**

  To establish a database connection, the URL and driver must be provided. If these attributes are not specified in the <DBCONNECT> tag, the xmlref attribute or the jndiname attribute must be specified.

  The Application Server Version 3 supports connection to JDBC databases and ODBC databases. When connecting to an ODBC database, you can use the Sun JDBC-to-ODBC bridge driver included in the Java Development Kit (JDK) or another vendor's ODBC driver.

  The url attribute specifies the location of the database. The driver attribute specifies the name of the driver to be used to establish the database connection.

  For a connection to a JDBC database, the URL consists of the following colon-separated elements: jdbc, the sub-protocol name, and the name of the database table to be accessed. An example for a connection to the Sample database included with IBM DB2 is:

  ```
  url="jdbc:db2:sample"driver="COM.ibm.db2.jdbc.app.DB2Driver"
  ```

  If the database is an ODBC database, you can use an ODBC driver or the the Sun JDBC-to-ODBC bridge included with the JDK. If you want to use an ODBC driver, refer to the driver documentation for instructions on specifying the database location (the url attribute) and the driver name.

  In the case of the bridge, the url syntax is jdbc:odbc:*database*. An example is:

  ```
  url="jdbc:odbc:autos"driver="sun.jdbc.odbc.JdbcOdbcDriver"
  ```

  > To enable the Application Server to access the ODBC database, use the ODBC Data Source Administrator to add the ODBC data source to the System DSN configuration. To access the ODBC Administrator, click the ODBC icon on the Windows NT Control Panel.
  >
  > If your JSP accesses a different JDBC or ODBC database than the one the Application Server uses for its repository, ensure that you add the JDBC or ODBC driver for the other database to the Application Server's classpath.

- **jndiname**

  An optional attribute that identifies a valid context in the Application Server JNDI naming context and the logical name of the data source in that context. The context is configured by the Web administrator using an administrative client such as the WebSphere Administrative Console.

  If the jndiname is specified, the JSP processor ignores the driver and url attributes on the <DBCONNECT> tag or in the file specified by the xmlref tag.

- **xmlref**

  A file (in XML format) that contains the URL, driver, user ID, password information needed for a connection. This mechanism provides Web administrators an alternative method for specifying the user ID and password. It is an alternative to hardcoding the information in a <DBCONNECT> tag or reading the information from the request object parameters. This is useful when third-party vendors develop your JSP files and when you need to make quick changes or test an application with a different data source.

  When the JSP compiler processes the <DBCONNECT> tag, it reads all of the specified tag attributes. If any of the required attributes are missing, the compiler checks for an xmlref attribute. If the attribute is specified, the compiler reads the configuration file.

  The xmlref takes precedence over the <DBCONNECT> tag. For example, if the <DBCONNECT> tag and the xmlref file include values for the URL and the the driver, the values in the xmlref file are used.

The configuration file can have any filename and extension that is valid for the operating system. Place the file in the same directory as the JSP that contains the referring <DBCONNECT> tag. An example of a configuration file is:

```
<?xml version="1.0" ?><db-info>  <url>jdbc:odbc:autos</url>  <user-id>smith</user-id>
<dbDriver>sun.jdbc.odbc.JdbcOdbcDriver</dbDriver>  <password>v598m</password>
<jndiName>jdbc/demo/sample</jndiName></db-info>
```

All of the elements shown in the example XML file need to be specified. However, an empty element (such as <url></url>) is valid.

4.2.2.3.8.2.2: JSP .91 syntax: <USERID> and <PASSWD> tag syntax

```
<dbconnect id="connection_id"   <font color="red"><userid></font><insert
requestparm="userid"></insert><font color="red"></userid></font>       <font
color="red"><passwd></font><insert requestparm="passwd"></insert><font color="red"></passwd></font>
url="protocol:database_name:database_table"     driver="JDBC_driver_name"> </dbconnect>
```

where:

- **<INSERT>**

  This tag is a JSP tag for including variable data. See JSP tags for variable data.

- **userid** tag

  This is a reference to the request parameter that contains the userid. The parameter must have already been added to the request object that was passed to this JSP file. The attribute and its value can be set in the request object using an HTML form or a URL query string to pass the user-specified request parameters.

  See the Login.jsp and the Employee.jsp examples for an illustration of how to set the USERID and PASSWD using parameters in the request object. The request parameters are set using an HTML form (Login.jsp). In the Employee.jsp, the values of the parameters are passed as hidden form values to the EmployeeRepeatResults.jsp.

- **passwd** tag

  This is a reference to the request parameter that contains the password. The parameter must have already been added to the request object that was passed to this JSP. The attribute and its value can be set in the request object using an HTML form or a URL query string to pass user-specified values.

4.2.2.3.8.2.3: JSP .91 syntax: <DBQUERY> tag

```
<!-- SELECT commands and (optional) JSP syntax can be placed within the DBQUERY tag. --><!-- Any other syntax, including HTML comments, are not valid. --><dbquery id="query_id" connection="connection_id" limit="value" ></dbquery>
```

where:

- **id**

  The identifier of this query. The scope is the JSP file. This identifier is used to reference the query, for example, from the <INSERT> tag to display query results.

  The id becomes the name of a bean that contains the results set. The bean properties are dynamic and the property names are the names of the columns in the results set. If you want different column names, use the SQL keyword for specifying an alias on the SELECT command. In the following example, the database table contains columns named FNAME and LNAME, but the SELECT statement uses the AS keyword to map those column names to FirstName and LastName in the results set:

  ```
  Select FNAME, LNAME AS FirstName, LastName from Employee where FNAME='Jim'
  ```

- **connection**

  The identifier of a <DBCONNECT> tag in this JSP file. That <DBCONNECT> tag provides the database URL, driver name, and (optionally) the user ID and password for the connection.

- **limit**

  An optional attribute that constrains the maximum number of records returned by a query. If the attribute is not specified, no limit is used and the effective limit is determined by the number of records and the system caching capability.

- SELECT command and JSP syntax

  Because the <DBQUERY> tag must return a results set, the only valid SQL command is SELECT. Refer to your database documentation for information about the SELECT command. See other sections of this document for a description of JSP syntax for variable data and inline Java code.

4.2.2.3.8.2.3a: Example: JSP .91 syntax: <DBQUERY> tag syntax

In the following example, a database is queried for data about employees in a specified department. The department is specified using the <INSERT> tag to embed a variable data field. The value of that field is based on user input.

```
<dbquery id="empqs" connection="conn" >select * from Employee where WORKDEPT='<INSERT
requestparm="WORKDEPT"></INSERT>'</dbquery>
```

4.2.2.3.8.2.4: JSP .91 syntax: <DBMODIFY> tag syntax

```
<!-- Any valid database update commands can be placed within the DBMODIFY tag. --><!-- Any other
syntax, including HTML comments, are not valid. --><dbmodify connection="connection_id" ></dbmodify>
```

where:

- **connection**

  The identifier of a <DBCONNECT> tag in this JSP file. That <DBCONNECT> tag provides the database URL, driver name, and (optionally) the user ID and password for the connection.

- Database commands

  Refer to your database documentation for valid database commands.

In the following example, a new employee record is added to a database. The values of the fields are based on user input from this JSP and referenced in the database commands using <INSERT> tags.

```
<dbmodify connection="conn" >insert into EMPLOYEE
(EMPNO,FIRSTNME,MIDINIT,LASTNAME,WORKDEPT,EDLEVEL)values        ('<INSERT
requestparm="EMPNO"></INSERT>',       '<INSERT requestparm="FIRSTNME"></INSERT>',     '<INSERT
requestparm="MIDINIT"></INSERT>',      '<INSERT requestparm="LASTNAME"></INSERT>',     '<INSERT
requestparm="WORKDEPT"></INSERT>',     <INSERT requestparm="EDLEVEL"></INSERT>)</dbmodify>
```

The EmployeeRepeatResults.jsp example illustrates this tag.

Displaying query results

To display the query results, use the <REPEAT> and <INSERT> tags. The <REPEAT> tag loops through each of the rows in the query results. The <INSERT> tag uses the query results object (for the <DBQUERY> tag whose identifier is specified by the <INSERT> bean attribute) and the appropriate column name (specified by the <INSERT> property attribute) to retrieve the value. An example is:

```
<repeat><tr>    <td><INSERT bean="empqs" property="EMPNO"></INSERT>     <INSERT bean="empqs"
property="FIRSTNME"></INSERT>      <INSERT bean="empqs" property="WORKDEPT"></INSERT>      <INSERT
bean="empqs" property="EDLEVEL"></INSERT>       </td></tr></repeat>
```

JSP 0.91 APIs and migration

Two interfaces support the JSP 0.91 technology. TheseAPIs provide a way to separate content generation (business logic) from thepresentation of the content (HTML formatting). This separation enablesservlets to generate content and store the content (for example, in a bean) inthe request object. The servlet that generated the context generates aresponse by passing the request object to a JSP file that contains the HTMLformatting. The <BEAN> tag provides access to the businesslogic.

The interfaces that supported JSP 0.91 for the Application Server Version 3 are:

- javax.servlet.http.HttpServletRequest.setAttribute()

  Supports setting attributes in the request object. For the Application Server Version 2, this interface was com.sun.server.http.HttpServiceRequest.setAttribute().

- javax.servlet.http.RequestDispatcher.forward()

  Supports forwarding a response object to another servlet or JSP. For the Application Server Version 2, this interface was com.sun.server.http.HttpServiceResponse.callPage().

4.2.2.3.8.2.4a: Example: JSP .91 syntax: <DBMODIFY> tag syntax

In the following example, a new employee record is added to a database. The values of the fields are based on user input from this JSP and referenced in the database commands using <INSERT> tags.

```
<dbmodify connection="conn" >insert into EMPLOYEE
(EMPNO,FIRSTNME,MIDINIT,LASTNAME,WORKDEPT,EDLEVEL)values        ('<INSERT
requestparm="EMPNO"></INSERT>',       '<INSERT requestparm="FIRSTNME"></INSERT>',     '<INSERT
requestparm="MIDINIT"></INSERT>',      '<INSERT requestparm="LASTNAME"></INSERT>',     '<INSERT
requestparm="WORKDEPT"></INSERT>',     <INSERT requestparm="EDLEVEL"></INSERT>)</dbmodify>
```

4.2.2.3.8.2.5a: Example: JSP .91 syntax: <INSERT> and <REPEAT> tags

```
<repeat><tr>    <td><INSERT bean="empqs" property="EMPNO"></INSERT>     <INSERT bean="empqs"
property="FIRSTNME"></INSERT>      <INSERT bean="empqs" property="WORKDEPT"></INSERT>      <INSERT
bean="empqs" property="EDLEVEL"></INSERT>       </td></tr></repeat>
```

4.2.2.3a: JSP examples

The example JSP application accesses the Sample database that you can install with IBM DB2. The example application includes:

JSPLogin.jsp	An interface for logging in to the application
JSPEmployee.jsp	A dialog for querying and updating database records
JSPEmployeeRepeatResults.jsp	A dialog for displaying update confirmations and query results

JSP code example - a login

```
<HTML><HEAD><TITLE>JSP:  Login into the Employee Records
Center</TITLE></HEAD><BODY><H1><CENTER>Login into the Employee Records Center</CENTER></H1><FORM
NAME="LoginForm" ACTION="employee.jsp" METHOD="post"
ENCODE="application/x-www-form-urlencoded"><P>To login to the Employee Records Center, submit a
validuserid and password to access the Sample database installed under IBM DB2.</P><TABLE><TR
VALIGN=TOP ALIGN=LEFT><TD><B><I>Userid:</I></B></TD><TD><INPUT TYPE="text" NAME="USERID"
VALUE="userid"><BR></TD></TR><TR VALIGN=TOP ALIGN=LEFT><TD><B><I>Password:</I></B></TD><TD><INPUT
TYPE="password" NAME="PASSWD" VALUE="password"></TD></TR></TABLE><INPUT TYPE="submit" NAME="Submit"
VALUE="LOGIN"></FORM><HR></BODY></HTML>
```

JSP code example - view employee records

```
<HTML><HEAD><TITLE>JSP:  Add and View Employee Records</TITLE></HEAD><BODY><H1><CENTER>Add and View
Employee Records</CENTER></H1><% String userID    = request.getParameter("USERID"); %><% String
passWord  = request.getParameter("PASSWD"); %><%-- Get a connection to the Sample DB2 database using
parameters from Login.jsp --%><tsx:dbconnect id="conn"  url="jdbc:db2:sample"
driver="COM.ibm.db2.jdbc.app.DB2Driver"><tsx:userid><%=userID%></tsx:userid><tsx:passwd><%=passWord%></tsx:passwd></tsx:dbconnect><FORM
NAME="EmployeeForm" ACTION="employeeRepeatResults.jsp" METHOD="post"
ENCODE="application/x-www-form-urlencoded"><h2>Add Employee Record</h2><P>To add a new employee
record to the database, submit the following data:</P><TABLE><TR VALIGN="TOP"
ALIGN="LEFT"><TD><B><I>Employee Number:<br>(1 to 6 characters)</I></B></TD><TD> <INPUT TYPE="text"
NAME="EMPNO"> </TD></TR><TR VALIGN="TOP" ALIGN="LEFT"><TD><B><I>First name:</I></B></TD><TD><INPUT
TYPE="text" NAME="FIRSTNME" VALUE="First Name"><BR></TD></TR><TR VALIGN="TOP"
ALIGN="LEFT"><TD><B><I>Middle Initial:</I></B></TD><TD><INPUT TYPE="text" NAME="MIDINIT"
VALUE="M"><BR></TD></TR><TR VALIGN="TOP" ALIGN="LEFT"><TD><B><I>Last Name: </I></B></TD><TD><INPUT
TYPE="text" NAME="LASTNAME" VALUE="Last Name"><BR></TD></TR><TR VALIGN="TOP" ALIGN="LEFT"><TD><%--
Query the database to get the list of departments --%><tsx:dbquery id="qs" connection="conn" >
select * from DEPARTMENT  </tsx:dbquery><B><I>Department:</I></B></TD><TD><SELECT NAME="WORKDEPT"
><tsx:repeat> <OPTION VALUE= "<tsx:getProperty name="qs" property="DEPTNO" />" ><tsx:getProperty
name="qs" property="DEPTNAME" /></tsx:repeat></SELECT></TD></TR><TR VALIGN="TOP"
ALIGN="LEFT"><TD><B><I>Education:</I></B></TD><TD><SELECT NAME="EDLEVEL"><OPTION VALUE="1"
SELECTED>BS<OPTION VALUE="2">MS<OPTION VALUE="3">PhD</SELECT></TD></TR></TABLE><INPUT TYPE="submit"
NAME="Submit" VALUE="Update"><INPUT TYPE="hidden" NAME="USERID" VALUE="<%=userID%>"><INPUT
TYPE="hidden" NAME="PASSWD" VALUE="<%=passWord%>"></FORM><HR><FORM NAME="EmployeeForm"
ACTION="employeeRepeatResults.jsp" METHOD="post" ENCODE="application/x-www-form-urlencoded"><h2>View
Employees by Department</h2><P>To view records for employees by department, select the departmentand
submit the query:</P><TABLE><TR VALIGN="TOP" ALIGN="LEFT"><TD><B><I>Department:</I></B></TD><TD><%--
Use the bean generated by earlier QUERY tag --%><SELECT NAME="WORKDEPT" ><tsx:repeat> <OPTION VALUE=
"<tsx:getProperty name="qs" property="DEPTNO" />" ><tsx:getProperty name="qs" property="DEPTNAME"
/></tsx:repeat></SELECT></TD></TR></TABLE><INPUT TYPE="submit" NAME="Submit" VALUE="Query"><INPUT
TYPE="hidden" NAME="USERID" VALUE="<%=userID%>"><INPUT TYPE="hidden" NAME="PASSWD"
VALUE="<%=passWord%>"></FORM><HR></BODY></HTML>
```

JSP code example - EmployeeRepeatResults

```
<HTML><HEAD><TITLE>JSP Employee Results</TITLE></HEAD><H1><CENTER>EMPLOYEE RESULTS</CENTER></H1><BODY><% String userID      =
request.getParameter("USERID"); %><% String passWord    = request.getParameter("PASSWD"); %><% String empno       =
request.getParameter("EMPNO"); %><% String firstnme    = request.getParameter("FIRSTNME"); %><% String midinit     =
request.getParameter("MIDINIT"); %><% String lastname    = request.getParameter("LASTNAME"); %><% String workdept    =
request.getParameter("WORKDEPT"); %><% String edlevel     = request.getParameter("EDLEVEL"); %><!-- Get a connection to the local
DB2 database using parameters from login.jsp --><tsx:dbconnect id="conn" url="jdbc:db2:sample"
driver="COM.ibm.db2.jdbc.app.DB2Driver"><tsx:userid><%=userID%></tsx:userid><tsx:passwd><%=passWord%></tsx:passwd></tsx:dbconnect><%
if  ( ( request.getParameter("Submit")).equals("Update") ) { %><tsx:dbmodify connection="conn" >   INSERT INTO  EMPLOYEE
(EMPNO,FIRSTNME,MIDINIT,LASTNAME,WORKDEPT,EDLEVEL)   VALUES    ( '<%=empno%>',     '<%=firstnme%>',      '<%=midinit%>',
'<%=lastname%>',     '<%=workdept%>',     <%=edlevel%>)    </tsx:dbmodify> <B><UL>UPDATE SUCCESSFUL</UL></B> <BR><BR><tsx:dbquery
id="qs" connection="conn" >   select * from Employee  where  WORKDEPT= '<%=workdept%>'</tsx:dbquery><B><CENTER><U>EMPLOYEE
LIST</U></CENTER></B><BR><BR><HR><TABLE><TR
VALIGN=BOTTOM><TD><B>EMPLOYEE<BR><U>NUMBER</U></B></TD><TD><B><U>NAME</U></B></TD><TD><B><U>DEPARTMENT</U></B></TD>
<TD><B><U>EDUCATION</U></B></TD></TR><tsx:repeat><TR><TD><B><I><tsx:getProperty name="qs" property="EMPNO"
/></I></B></TD><TD><B><I><tsx:getProperty name="qs" property="FIRSTNME" /></I></B></TD><TD><B><I><tsx:getProperty name="qs"
property="WORKDEPT" /></I></B></TD><TD><B><I><tsx:getProperty name="qs" property="EDLEVEL" /></I></B></TD></TR></tsx:repeat>
</TABLE><HR><BR><% } %><% if  ( ( request.getParameter("Submit")).equals("Query") ) { %><tsx:dbquery id="qs2" connection="conn" >
select * from Employee  where  WORKDEPT= '<%=workdept%>'</tsx:dbquery><B><CENTER><U>EMPLOYEE
LIST</U></CENTER></B><BR><BR><HR><TABLE><TR><TR
VALIGN=BOTTOM><TD><B>EMPLOYEE<BR><U>NUMBER</U></B></TD><TD><B><U>NAME</U></B></TD><TD><B><U>DEPARTMENT</U></B></TD><TD><B><U>EDUCATION</U></B></TD></TR><tsx:repeat><TR><TD><B><I><tsx:getProperty
name="qs2" property="EMPNO" /></I></B></TD><TD><B><I><tsx:getProperty name="qs2" property="FIRSTNME"
/></I></B></TD><TD><B><I><tsx:getProperty name="qs2" property="WORKDEPT" /></I></B></TD><TD><B><I><tsx:getProperty name="qs2"
property="EDLEVEL" /></I></B></TD></TR></tsx:repeat> </TABLE><HR><BR><% } %></BODY></HTML>
```

4.2.2.3b: JSP .91 examples

The example JSP application accesses the Sample database that you can install with IBM DB2. The example application includes:

(Login.jsp)	An interface for logging in to the application
(Employee.jsp)	A dialog for querying and updating database records
(EmployeeRepeatResults.jsp)	A dialog for displaying update confirmations and query results

4.2.3: Incorporating XML

IBM WebSphere Application Server provides XML Document Structure Services, which consist of a document parser, a document validator, and a document generator for server-side XML processing.

See article 4.1.1.2 for all of the details about XML supportin the product.If you are just becoming familiar with XML, start with article 0.33, a primer on XML concepts, vocabulary, and uses.

Other related information provides guidance on the following topics:

- Structure -- defining and obeying the syntax for an XML tag set
- Content -- determining the mechanism for filling XML tags with data
- Presentation -- determining the mechanism for formatting and displaying XML content

In addition, some special topics are covered, including DOM objects andmanipulation of Channel Definition Format (CDF) files as illustrated bythe SiteOutliner example.

When you install IBM WebSphere Application Server, the core XML APIs are automatically added to the appropriate class path, enabling you to serve static XML documents as soon as the product is installed.

To serve XML documents that are dynamically generated, use the core APIs to develop servlets or Web applications that generate XML documents (for example, the applications might read the document content from a database) and then deploy those components on your application server.

4.2.3.2: Specifying XML document structure

The structure of an XML document is governed by syntax rules for its tag set. Those tags are defined formally in an XML-based grammar, such as a Document Type Definition (DTD). At the time of this publication, DTD is the most widely-implemented grammar. Therefore, this article discusses options for using DTDs.

Options for XML document structure include:

Do not use a DTD. Not using a DTD enables maximum flexibility in evolving XML document structure, but this flexibility limits the ability to share the documents among users and applications. An XML document can be parsed without a DTD. If the parser does not find an inline DTD or a reference to an external DTD, the parser proceeds using the actual structure of the tags within the document as an implied DTD. The processor evaluates the document to determine whether it meets the rules for well-formedness.

Use a public DTD. Various industry and other interest groups are developing DTDs for categories of documents, such as chemical data and archival documents. Many of these DTDs are in the public domain and are available over the Internet. Using an industry standard DTD maximizes sharing documents among applications that act on the grammar. If the standard DTD does not accomodate the schema the applications need, flexibility is limited.

Several industry and interest groups have developed and proposed DTD grammars for the types of documents they produce and exchange. To make it easier for you to use those grammars, local copies are installed with the product. Use the grammars as examples in developing your own grammars as well as for creating and validating XML documents of those types. The library is located atproduct_installation_root\web\xml\grammar\

Develop a DTD. If none of the public DTDs meet an enterprise's needs and enforcing document validity is a requirement, the XML implementers can develop a DTD. Developing a DTD requires careful analysis of the information (data) that the documents will contain.

For DTD updates,visit the XML Industry Portal.For details about the DTD specifications and sample DTDs, refer toIBM's developerWorkssite for education and other DTD resources.

4.2.3.3: Providing XML document content

The content of an XML document is the actual data that appears within the document tags. XML implementers must determine the source and the mechanism for putting the data into the document tags. The options include:

Static content. XML document content is created and stored on the Web server as static files. The XML document author composes the document to include valid XML tags and data in a manner similar to how HTML authors compose static HTML files. This approach works well for data that is not expected to change or that will change infrequently. Examples are journal articles, glossaries, and literature.

Dynamically generated content. XML document content can be dynamically generated from a database and user input. In this scenario, XML-capable servlets, Java beans, and even inline Java code within a JavaServer Page (JSP) file can be used to generate the XML document content.

A hybrid of static and dynamically generated content. This scenario involves a prudent combination of static and dynamically generated content.

You can also use XSL to add to or remove information from existing XML content.For details, see the Related information.

4.2.3.4: Rendering XML documents

Options for presenting XML documents include:

Present the XML document in an XML-enabled browser. An XML-enabled browser can parse a document, apply its XSL stylesheet, and present the document to the user. Searching and enabling users to modify an XML document are other possible functions of XML-enabled browsers.

Present the XML document to a browser that converts XML to HTML. Until XML-enabled browsers are readily available, presenting XML documents to users will involve converting the XML document to HTML. That conversion can be handled by conversion-capable browsers. Another option is to use JavaScript or ActiveX controls embedded within the XML document. Microsoft Internet Explorer Version 5 is an XML-to-HTML converter. HTML is not the only format to which XML documents can be converted. It's just the easiest to implement given the commerically available browsers and user agents.

Send an HTML file to the browser. If the users do not have XML-capable browsers, the XML document must be converted at the server before being transmitted to the browser. The server-side XML application that handles the conversion could also determine the capability of the browser before converting the document to HTML, to avoid unnecessary processing if the browser is XML-capable. The XSL processor included with this product supports such server-side functions.

Using XSL to convert XML documents to other formats

IBM WebSphere Application Server includes the Lotus XSL processor and its open-sourceversion, Xalan, for formatting and converting XML documents. Processing can be done at the server or at the browser, to HTML or to other XML-compliant markup languages.For sample code, see the Xalan documentation.

Use of the XSL processor with the Xerces XML parserrequires a liaison object, as follows:

```
XSLTProcessor processor =    XSLTProcessorFactory.getProcessor(new
com.lotus.xml.xml4j2dom.XML4JLiaison4dom());
```

Converting XML documents at the server

One option for presenting an XML document is for the server to convert the XML document to HTML and return the HTML document to the client. On the server side, this typically requires the creation of a servletto handle the processing of one data stream (the XML document)with another (the XSL document).The output of that process is then forwarded back to the browser.

Server-side processing often requires the passing in of parameters through theXSL processor to customize the output.For an example, see the Xalan documentation.

4.2.3.6: Using DOM to incorporate XML documents into applications

The Document Object Model (DOM) is an API for representing XML and HTML documentsas objects that can be accessed by object-oriented programs, such as business logic, for the purposes of creating, navigating, manipulating, and modifying the documents.

Article 0.33.3 introduces DOM concepts and vocabulary. Article 4.1.1.2 tells youwhere to find the DOM specification and org.w3c.dom package.

Article 4.2.3.6.1 providesa quick reference so that you can jump right into DOM development, referring to thepackage and specification as needed.

4.2.3.6.1: Quick reference to DOM object interfaces

This section highlights a few of the object interfaces. Refer to theDOM Specification for details (see article 4.1.1.2).

Node methods

Node methods include:

Method	Description
hasChildNodes	Returns a boolean to indicate whether a node has children
appendNode	Appends a new child node to the end of the list of children for a parent node
insertBefore	Inserts a child node before the existing child node
removeChild	Removes the specified child node from the node list and returns the node
replaceChild	Replaces the specified child node with the specified new node and returns the new node

Document methods

The Document object represents the entire XML document. Document methods include:

Method	Description
createElement	Creates and returns an Element (tag) of the type specified. If the document will be validated against a DTD, that DTD must contain an Element declaration for the created element.
createTextNode	Creates a Text node that contains the specified string
createComment	Creates a Comment node with the specified content (enclosed within <! - - and - -> tags)
createAttribute	Creates an Attribute node of the specified name. Use the setAttribute method of Element to set the value of the Attribute. If the document will be validated against a DTD, that DTD must contain an Attribute declaration for the created attribute.
createProcessingInstruction	Creates a Processing Instruction with the specified name and data (enclosed within <? and ?> tags). A processing instruction is an instruction to the application (such as an XML document formatter) that receives the XML document.

Element methods

Element node methods include:

Method	Description
getAttribute	Returns the value of the specified attribute or empty string
setAttribute	Adds a new attribute-value pair to the element

removeAttribute	Removes the specified attribute from the element
getElementsByTagName	Returns a list of the element descendants that have the specified tag name

A Text node can be a child of an Element or Attribute node and contains the textual content (character data) for the parent node. If the content does not include markup, all of the content is placed within a single Text node. If the content includes markup, that markup is placed in one or more Text nodes that are siblings of the Text node that contains the non-markup content.

The Text node extends the CharacterData interface, which has methods for setting, getting, replacing, inserting, and making other modifications to a Text node. In addition to those methods, the Text node adds a method:

Method	**Description**
splitText	Splits the Text node at the specified offset. Returns a new Text node, which contains the original content starting at the offset. The original Text node contains the content from the beginning to the offset.

4.2.3.6.2: Manually generating an XML element node

You can manually create any XML element node by using the PseudoNode construct,as follows:

```
new PseudoNode("literal");
```

If a DOM tree contains PseudoNode instances, you can use the tree for printing only. PseudoNode prevents validation against a DTD.

4.2.3.7: SiteOutliner sample

The SiteOutliner servlet illustrates how to use the XML Document Structure Services to generate and view a Channel Definition Format (CDF) file for a target directory on the servlet's Web server. Use Lotus Notes 5 (the Headlines page), Microsoft Internet Explorer 4 Channel Bar, PointCast, Netscape Navigator 4.06, orother CDF-capable viewers to view and manipulate the CDF file.

SiteOutliner is part of the WebSphere Samples Gallery. When you open the gallery,follow the links to SiteOutliner to run it on your local machine.

4.2.4: Putting it all together (Web applications)

This section discusses some Web application features, such as data access and security,that can implemented in a variety of ways.

4.2.4.2: Obtaining and using database connections

IBM WebSphere Application Server Version 3.5 provides two options for accessing database connections:

- Connection pooling (model based on JDBC 2.0)
- Connection manager (now deprecated, model based on JDBC 1.0)

Because connection pooling is the most efficient model for Web applications, it is recommended that you use connection pooling for all new applications requiring database access. You should consider migrating existing applications to connection pooling if your applications use connection manager or the standard JDBC 1.0 methods for getting database connections.

IBM WebSphere Application Server also provides data access beans, which offer a rich set of features for working with relational database queries and result sets.

For a comprehensive treatment of WebSphere connection pooling and data access,be sure to read the IBM whitepaper to be published on the Webduring the summer of 2001.

4.2.4.2.1: Accessing data with the JDBC 2.0 Optional Package APIs

In JDBC 1.0 and the JDBC 2.0 Core API, the DriverManager class is used exclusively for obtaining a connection to a database. The database URL, user ID, and password are used in the getConnection() call. In the JDBC 2.0 Optional Package API, the DataSource object provides a means for obtaining connections to a database. The benefit of using datasourcesis that the creation and management of the connection factory is centralized. Applications do not need to have specific information like the database name, user ID, or password in order to obtain a connection to the database.

The steps for obtaining and using a connection with the JDBC 2.0 Optional Package API differ slightly from those in the JDBC 2.0 Core API example. Using the extensions, you access a relational database as follows:

1. Retrieve a DataSource object from the JNDI naming service
2. Obtain a Connection object from the datasource
3. Send SQL queries or updates to the database management system
4. Process the results

The connection obtained from the datasource is a pooled connection. This means that the Connection object is obtained from a pool of connections managed by IBM WebSphere Application Server.The following code fragment shows how to obtain and use a connection with the JDBC 2.0 Optional Package API:

```
try {// Retrieve a DataSource through the JNDI Naming Service       java.util.Properties parms = new
java.util.Properties();   parms.setProperty(Context.INITIAL_CONTEXT_FACTORY,
"com.ibm.ejs.ns.jndi.CNInitialContextFactory");      // Create the Initial Naming Context
javax.naming.Context ctx = new javax.naming.InitialContext(parms);      // Lookup through the naming
service to retrieve a DataSource object   javax.sql.DataSource ds =
(javax.sql.DataSource)ctx.lookup("jdbc/SampleDB");     // Obtain a Connection from the DataSource
java.sql.Connection conn = ds.getConnection(); // query the database   java.sql.Statement stmt =
conn.createStatement();   java.sql.ResultSet rs =       stmt.executeQuery("SELECT EMPNO, FIRSTNME,
LASTNAME FROM SAMPLE");      // process the results   while (rs.next()) {      String empno =
rs.getString("EMPNO");      String firstnme = rs.getString("FIRSTNME");      String lastname =
rs.getString("LASTNAME");   // work with results   }} catch (java.sql.SQLException sqle) {// handle
SQLException} finally {   try {      if (rs != null) rs.close();   }   catch (java.sql.SQLException
sqle) {   // can ignore   }   try {      if (stmt != null) stmt.close();   }    catch
(java.sql.SQLException sqle) {   // can ignore   }   try {      if (conn != null) conn.close();   }
catch (SQLException sqle) {   // can ignore   }} // end finally
```

In the previous example, the first action is to retrieve a DataSource object from the JNDI namespace. This is done by creating a Properties object of parameters used to set up an InitialContext object. After a context is obtained, a lookup on the context is performed to find the specific datasource necessary, in this case, SampleDB.

(In this example, it is assumed the datasource has already been created and bound into JNDI by the WebSphere administrator. For information about doing thisin application code, see the Related information.)

After a DataSource object is obtained, the application code calls getConnection()on the datasource to get a Connection object. After the connection is acquired, the querying and processing steps are the same as for theJDBC 1.0 example.

4.2.4.2.1.1: Creating datasources with the WebSphere connection pooling API

IBM WebSphere Application Serverprovides a public API to enable you to configure a WebSphere datasource in application code.This is necessary only when the application must create a datasource on demand.Otherwise, the datasource is configured by the administrator in the administrative console.

The complete API specification can be found in javadoc for the class com.ibm.websphere.advanced.cm.factory.DataSourceFactory. See the Related information.

To create a datasource on demand in an application, the application must do the following:

1. Create a Properties object with datasource properties
2. Obtain a datasource from the factory
3. Bind the datasource into JNDI

The following code fragment shows how an application would create a datasource and bind it into JNDI:

```
import com.ibm.websphere.advanced.cm.factory.DataSourceFactory;...try {   // Create a properties file for the DataSource   java.util.Properties prop = new java.util.Properties(); prop.put(DataSourceFactory.NAME, "SampleDB");   prop.put(DataSourceFactory.DATASOURCE_CLASS_NAME, "COM.ibm.db2.jdbc.DB2ConnectionPoolDataSource");   prop.put(DataSourceFactory.DESCRIPTION, "My sample                datasource");   prop.put("databaseName", "sample");// Obtain a DataSource from the factory   DataSource ds = DataSourceFactory.getDataSource(prop);// Bind the DataSource into JNDI DataSourceFactory.bindDataSource(ds);} catch (ClassNotFoundException cnfe) {// check the class path for all necessary classes} catch (CMFactoryException cmfe) {// Example of exception: incorrect properties} catch (NamingException ne) {// Example of exception:  datasource by this name may already exist}
```

To create a datasource for binding into JNDI, the application must firstcreate a Properties object to hold the DataSource configuration properties.The only properties required for the datasource from a WebSphere perspective are:

- NAME -The name of the datasource. This is used to identify the datasource when it is bound into JNDI.
- DATASOURCE_CLASS_NAME - The complete name of the DataSource class that can be found in the JDBC resource archive file(often referred to as the JDBC driver package). This DataSource class will be used to create connections to the database. The class specified here must implement javax.sql.ConnectionPoolDataSource or javax.sql.XADataSource.

However, depending on the DataSource class specified in the DATASOURCE_CLASS_NAME property, there may be other vendor-specific properties required. In this example, the databaseName property is also required,because DB2ConnectionPoolDataSource is being used. For more information on these vendor-specific properties, see the vendor's documentation for the complete list of properties supported for a datasource.

After a properties object is created, the application can create a new DataSource object by calling getDataSource() on the factory, passing in the Properties object as a parameter. This creates an object of type DataSource, but it is not yet bound into JNDI. To bind a datasource into JNDI,call bindDataSource() on the factory.At this point, other applications can share the datasource by retrieving it from JNDI with the name property specified on creation.

All other APIs specific to connection pooling are not public APIs. Applications that use a WebSphere datasource should followthe JDBC 2.0 Core and JDBC 2.0 Optional Package APIs.

4.2.4.2.1.2: Tips for using connection pooling

Most best practices have been documented elsewhere in Related information.The following are additional items that have not been explicitly called out:

Obtain and close connection in the same method. An application should obtain and close its connection in the method that requires the connection. This keeps the application from holding resources not being used and leaves more available connections in the pool for other applications. In addition, this practice removes the temptation to use the same connection in multiple transactions, which, by default, is not allowed. This practice does not cost the application much in performance,because the Connection object is from a pool of connections, where the overhead for establishing the connection has already been incurred.Lastly, make sure to declare the Connection object in the same method as the getConnection() call in a servlet;otherwise, the Connection object works as if it is a static variable(see "Worst Practices" later in this article for problems with this).

If you opened it, close it. All JDBC resources that have been obtained by an application should be explicitly closed by that application. The product tries to clean up JDBC resources on a connection after the connection has been closed. However, this behavior should not be relied upon, especially if the application might be migrated to another platform in the future.

For servlets, obtain DataSource in the init() method. For performance reasons, it is usually a good idea to put the JNDI lookup for the datasource into the init() method of the servlet. Because the datasource is simply a factory for connections that does not typically change,retrieving it in this method ensures that the lookup happens only once.

Worst practices

The following are some very common problems with applications that should be avoided, because they most often result in unexpected failures:

Do not close connections in a finalize() method. If an application waits to close a connection or other JDBC resource until the finalize() method, the connection is not closed until the object that obtained it is garbage-collected. This leads to problems if the application is not very thorough about closing its JDBC resources, such as ResultSet objects. Databases can quickly run out of the memory required to store the information about all of the JDBC resources it currently has open. In addition, the pool can quickly run out of connections to service other requests.

Do not declare connections as static objects. It is never recommended that connections be declared as static objects. If a connection is declared as static, the same connection might get used on different threads at the same time. This causes a great deal of difficulty, within both the product and the database.

In servlets, do not declare Connection objects as instance variables. In a servlet, all variables declared as instance variables act as if they are class variables. For example, in a servlet with an instance variable

```
Connection conn = null;
```

this variable acts as if it were static. In this case, all instances of the servlet use the same Connection object. This is because a single servlet instance can be used to serve multiple Web requests in different threads.

In CMP beans, do not manage data access. If a Container Managed Persistence (CMP) bean is writtenso that it manages its own data access, this data access may be part of a global transaction. Generally, if specialized data access is required,use a BMP session bean.

4.2.4.2.1.3: Handling data access exceptions

For data access, the standard Java exception class to catch is java.sql.SQLException.IBM WebSphere Application Servermonitors for specific SQL exceptions thrown from the database. Several of these exceptions are mapped to WebSphere-specific exceptions. The product provides WebSphere-specific exceptions to ease development by not requiring you to know all of the database-specific SQL exceptions that could be thrown in typical situations. In addition, monitoring SQL exceptions enables the product and application to recover from common problems like intermittent network or database outages.

ConnectionWaitTimeoutException

This exception (com.ibm.ejs.cm.pool.ConnectionWaitTimeoutException) indicates that the application has waited for the connectionTimeout (CONN_TIMEOUT) number of seconds and has not been returned a connection. This can occur when the pool is at its maximum size and all of the connections are in use by other applications for the duration of the wait. In addition, there are no connections currently in use that the application can share, because either the user ID and password are differentor it is in a different transaction.The following code fragment shows how to use this exception:

```
java.sql.Connection conn = null;javax.sql.DataSource ds = null;...try {// Retrieve a DataSource through the JNDI Naming Service       java.util.Properties parms = new java.util.Properties(); setProperty.put(Context.INITIAL_CONTEXT_FACTORY, "com.ibm.websphere.naming.WsnInitialContextFactory");      // Create the Initial Naming Context javax.naming.Context ctx = new       javax.naming.InitialContext(parms);      // Lookup through the naming service to retrieve a DataSource object   javax.sql.DataSource ds = (javax.sql.DataSource)ctx.lookup("jdbc/SampleDB");   conn = ds.getConnection();    // work on connection} catch (com.ibm.ejs.cm.pool.ConnectionWaitTimeoutException cw) {// notify the user that the system could not provide a // connection to the database}  catch (java.sql.SQLException sqle) {// deal with exception}
```

In all cases in which the ConnectionWaitTimeoutException is caught, there is very little to do in terms of recovery. It usually doesn't make sense to retry the getConnection() method, because if a longer wait time is required, the connectionTimeout datasource property should be set higher. Therefore, if this exception is caught by the application, the administrator should review the expected usage of the application and tune the connection pool and the database accordingly.

StaleConnectionException

This exception (com.ibm.ejs.cm.portability.StaleConnectionException)indicates that the connection currently being held is no longer valid. This can occur for numerous reasons, including:

- The application fails to get a connectionbecause of a problem such as the database not being started.
- A connection is no longer usable because of a database failure. When an application tries to use a connection it previously obtained, the connection is no longer valid. In this case, all connections currently in use by the application may prompt this exception.
- The application using the connection has already called close() and then tries to use the connection again.
- The connection has been orphaned, andthe application tries to use the orphaned connection.
- The application tries to use a JDBC resource, such as Statement, obtained on a now-stale connection.

When application code catches StaleConnectionException, it should take explicit steps to handle the exception. StaleConnectionException extends SQLException, so it can be thrown from any method that is declared to throw SQLException. The most common occasion for a StaleConnectionException to be thrown is the first time a connection is used, just after it has been retrieved. Because connections are pooled, a database failure is not detected until the operation immediately following its retrieval from the pool, which is the first time communication with the database is attempted. It is only when a failure is detected thatthe connection is marked stale. StaleConnectionException occurs less often if each method that accesses the database gets a new connection from the pool. Typically, this occurs because all connections currently allocatedto an application are marked stale; the more connections the application has, the more connections can be stale.

Generally,when a StaleConnectionException is caught, the transaction in which the connection was involvedneeds to be rolled back and a new transaction begun with a new connection.

For more information and detailed code samples,be sure to read the IBM whitepaper to be published on the Webduring the summer of 2001.

4.2.4.2.2: Accessing data with the JDBC 1.0 reference model

The reference model that uses the JDBC 1.0 APIs, which still work under JDBC 2.0 and Application Server Version 3.x, is based on the code fragments shown in the following steps:

1. Load the driver for a specific relational database product. The specific driver class should be available from the WebSphere administrator.

   This step is typically performed once, during the init() method of the servlet.

   ```
   Class.forName("COM.ibm.db2.jdbc.app.DB2Driver");
   ```

2. Use the static getConnection() method of the DriverManager class to get a JDBC connection to the relational database product, again using parameters for the specific database product. The WebSphere administrator can provide the subprotocol, database, user ID, and password information.

   This step is performed for each client request made to the servlet, typically in the doGet() or doPost() method. (The subprotocol and database information are combined into what is called the database URL, shown as "jdbc:subprotocol:database" in the following code.)

   ```
   Connection conn =  DriverManager.getConnection("jdbc:subprotocol:database",     // database URL
   "userid",                                "password");
   ```

3. Given the connection, do the necessary data server interactions for each client request. This step is typically performed in the doGet() or doPost() method.
4. At the end of each client request, free the connection resource. This step is typically performed at the end of the doGet() or doPost() method.

   ```
   conn.close();
   ```

4.2.4.2.3: Accessing relational databases with the IBM data access beans

Java programs that access JDBC-compliant relational databases typically use the classes and methods in the java.sql package to access data. Instead of using the java.sql package, you can use the classes and methods in the package com.ibm.db, the IBM data access beans. This gives you additional features for data access beyond those available in the java.sql package.

The Related information discusses what the data access beans are, their advantages, and how to use them. A data access bean uses a connection that you provide to it, such as a connection from a connection pool that you get through a DataSource object.

4.2.4.2.3.1: Example: Servlet using data access beans

The sample servlet uses the data access beans and is based on the sample servlet discussed in Article 4.2.4.2.1.1. The connection pooling sample servlet uses classes such as Connection, Statement, and ResultSet from the java.sql package to interact with the database. In contrast, this sample servlet uses the data access beans, instead of the classes in the java.sql package, to interact with the database. For convenience, call this sample servlet the DA (for data access beans) and call the sample servlet on which it is based the CP (for connection pooling).

The CP and DA sample servlets benefit from the performance and resource management enhancements made possible by connection pooling. The programmer coding the DA sample servlet benefits from the additional features and functions provided by the data access beans.

The DA sample servlet differs slightly from the CP sample servlet. This discussion covers only the changes. See Article 4.2.4.2.1.1 for the discussion of the CP sample servlet. The DA sample servlet shows the basics of connection pooling and the data access beans, but keeps other code to a minimum. Therefore, the servlet is not entirely realistic. You are expected to be familiar with basic servlet and JDBC coding.

The changes

This section describes how the DA sample servlet differs from the CP sample servlet. To view the coding in one or both of the samples while you read this section, click these links:

- DA sample
- CP sample

Steps 1 through 6 of the CP sample servlet are mostly unchanged in the DA sample servlet. The main changes to the DA sample servlet are:

- New package

  The com.ibm.db package (containing the data access beans classes) must be imported. The classes are in the databeans.jar file, found in the lib directory under the Application Server root install directory. You will need this jar file in your CLASSPATH in order to compile a servlet using the data access beans.

- The metaData variable

  This variable is declared in the Variables section at the start of the code, outside of all methods. This allows a single instance to be used by all incoming user requests. The full specification of the variable is completed in the init() method.

- The init() method

  New code has been appended to the init() method to do a one-time initialization on the metaData object when the servlet is first loaded. The new code begins by creating the base query object sqlQuery as a String object. Note the two "?" parameter placeholders. The sqlQuery object specifies the base query within the metaData object. Finally, the metaData object is provided higher levels of data (metadata), in addition to the base query, that will help with running the query and working with the results. The code sample shows:

  - The addParameter() method notes that when running the query, the parameter idParm is supplied as a Java Integer datatype, for the convenience of the servlet, but that idParm should be converted (through the metaData object) to do a query on the SMALLINT relational datatype of the underlying relational data when running the query.

    A similar use of the addParameter() method for the deptParm parameter notes that for the same underlying SMALLINT relational datatype, the second parameter will exist as a different Java

datatype within the servlet - as a String rather than as an Integer. Thus parameters can be Java datatypes convenient for the Java application and can automatically be converted by the metaData object to be consistent with the required relational datatype when the query is run.

Note that the "?" parameter placeholders in the sqlQuery object and the addParameter() methods are related. The first addParameter() attaches idParm to the first "?", and so on. Later, a setParameter() will use idParm as an argument to replace the first "?" in the sqlQuery object with an actual value.

- The addColumn() method performs a function somewhat similar to the addParameter() method. For each column of data to be retrieved from the relational table, the addColumn() method maps a relational datatype to the Java datatype most convenient for use within the Java application. The mapping is used when reading data out of the result cache and when making changes to the cache (and then to the underlying relational table).
- The addTable() method explicitly specifies the underlying relational table. This information is needed if changes to the result cache are to be propagated to the underlying relational table.

- Step 5

Step 5 has been rewritten to use the data access beans to do the SQL query instead of the classes in the java.sql package. The query is run using the selectStatement object, which is a SelectStatement data access bean.

Step 5 is part of the process of responding to the user request. When steps 1 through 4 have run, the conn Connection object from the connection pool is available for use. The code shows:

1. The dataAccessConn object (a DatabaseConnection bean) is created to establish the link between the data access beans and the database connection - the conn object.
2. The selectStatement object (a SelectStatement bean) is created, pointing to the database connection through the dataAccessConn object, and pointing to the query through the metaData object.
3. The query is "completed" by specifying the parameters using the setParameter() method. The "?" placeholders in the sqlQuery string are replaced with the parameter values specified.
4. The query is executed using the execute() method.
5. The result object (a SelectResult bean) is a cache containing the results of the query, created using the getResult() method.
6. The data access beans offer a rich set of features for working with the result cache - at this point the code shows how the first row of the result cache (and the underlying relational table) can be updated using standard Java coding, without the need for SQL syntax.
7. The close() method on the result cache breaks the link between the result cache and the underlying relational table, but the data in the result cache is still available for local access within the servlet. After the close(), the database connection is unnecessary. Step 6 (which is unchanged from the CP sample servlet) closes the database connection (in reality, the connection remains open but is returned to the connection pool for use by another servlet request).

- Step 7

Step 7 has been entirely rewritten (with respect to the CP sample servlet) to use the query result cache retrieved in Step 5 to prepare a response to the user. The query result cache is a SelectResult data access bean.

Although the result cache is no longer linked to the underlying relational table, the cache can still be accessed for local processing. In this step, the response is prepared and sent back to the user. The code shows the following:

- The nextRow() and previousRow() methods are used to navigate through the result cache.

Additional navigation methods are available.

- The getColumnValue() method is used to retrieve data from the result cache. Because of properties set earlier in creating the metaData object, the data can be easily cast to formats convenient for the needs of the servlet.

A possible simplification

If you do not need to updatethe relational table, you can simplify the sample servlet:

- At the end of the init() method, you can drop the lines with the addColumn() and addTable() methods, since the metaData object does not need to know as much if there are no relational table updates.
- You will also need to drop the lines with the setColumnValue() and updateRow() methods at the end of step 5, because you are no longer updating the relational table.
- Finally, you can remove most of the type casts associated with the getColumnValue() methods in step 7. You will, however, need to change the type cast to (Short) for the "ID" and "DEPT" use of the getColumnValue() method.

4.2.4.2.4: Database access by servlets and JSP files

Servlets using getConnection() to access a data source

When used without parameters, getConnection() assumes the default user ID and password for a data source. The WebSphere administrative clients do not offer a way to configure a default user ID and password for a data source to be used by a servlet.

Therefore, servlets using getConnection() to access a data source should specify a user ID and password:

```
getConnection(userid,password);
```

4.2.4.4: Providing ways for clients to invoke applications

A Web application is of little use if users cannot access it. Users access a Webapplication by invoking a component that provides an entry point into the Web application, such as a JSP or servlet. The entry point is usually accessible froma Web page or the like. See article Installing applications and setting classpathsfor more information.

4.2.4.4.1: Providing Web clients access to JSP files

Suppose an application contains one or more JSP files -- how does the applicationdeveloper allow a user at a Web client (browser) to invoke the JSP files? The tablesummarizes the available approaches. Click an approach for details.

Programming approach	How user accesses JSP file
Provide the JSP file URL to users for direct access, or include an HREFlink to the JSP file on the Web site	Type the JSP URL in a browser, or follow a link to it
Call JSP file from an HTML form	Fill out an HTML form and submit it to the JSP file for processing
Call JSP file from another JSP file	Open a JSP file that invokes the JSP file

4.2.4.4.1.1: Invoking servlets and JSP files by URLs

Users can invoke a servlet or JSP file by its URL, using a browser to open:

`http://your.server.name/application_Web_path/servlet_or_JSP_Web_path`

Users must be provided with the URL to use in order to invoke the servlet.See the Related information to learn how to determine the URL.


Appending `/$/foo` to the URL allows you to access the servlet URL, butthe URL is then misunderstood by the runtime environment.This type of URL may create a security exposure.The best practice for securing servlets is to follow the Java security specifications documented in the Securing applications section.

Note that in order for servlets to be invoked by their class names,the administrator must have manually enabled the option while configuring theWeb application to which the servlet belongs.

4.2.4.4.1.2: Invoking servlets and JSP files within HTML forms

A Web page can be designed so that users can invoke a servlet or JSP file from an HTML form. An HTML form enables a user to enter data on a Web page (from abrowser) and submit the data to a servlet, or a servlet generated by a JSP file.

The HTML FORM tag has attributes for specifying how to invoke the servlet or JSP file:

FORM attribute	Description
METHOD	Indicates how user information is to be submitted.
ACTION	Indicates the URL used to invoke the servlet or JSP file

If the information entered by the user is to be submitted to a *servlet* by a GET or POST method, the servlet must override the doGet() method or doPost() method. For JSP files, the override is notnecessary. The same service method that is called whether the form is submitted using GET or POST.

Examples

Using GET:

```
<FORM METHOD="GET" ACTION="/application_Web path/servlet_Web_path"><!-- HTML tags for text entry areas, buttons, and other prompts go here --></FORM>
```

Using POST:

```
<FORM METHOD="POST" ACTION="application_Web_path/servlet_Web_path"><!-- HTML tags for text entry areas, buttons, and other prompts go here --></FORM>
```

4.2.4.4.1.2.1: Example: Invoking servlets within HTML forms

Suppose the application programmer uses an HTML form toprovide users access to a servlet. Assuming the METHOD attributeon the FORM tag is "GET," the flow is as follows:

1. The user views the form in a browser. The user providesinformation requested by the form and specifies to submit theform (usually by clicking a Submit button or other button visibleon the form).
2. The form encodes the user-supplied information into a URL-encodedquery string. It appends the query string to the servlet URL andsubmits the entire URL.
3. The servlet processes the information. The getParameterNames(), getParameter(), andgetParameterValues() methods of the HttpServletRequest object provide accessto the form parameter names and values in the client request.The extraction process also decodes the names and values.
4. Often, the final action of the servlet is to dynamically create an HTMLresponse (based on parameter input from the form) and pass it back to the userthrough the server. Methods of the HttpServletResponse object are usedto send the response, which is sent back to the client as a complete HTMLpage.

4.2.4.4.1.3: Invoking JSP files within other JSP files

An application developer can enable users to invoke a JSP file fromwithin another JSP file. Within the first JSP file, the developer should use one of the following methods for invoking the second JSP file:

- Specify the URL of the second JSP file on the FORM ACTION attribute.
- Specify the URL of the second JSP file on the anchor tag HREF attribute(<a href="*JSP_URL*">).
- Use the javax.servlet.http.RequestDispatcher.forward() method to invoke the second JSP file.
- Use the jsp:forward and jsp:include elements.

4.2.4.4.2: Providing Web clients access to servlets

Suppose an application contains one or more servlets -- how doesthe application developer allow a user at a Web client (browser) toinvoke the servlets? The table summarizes the available approaches. Clickan approach for details.

Programming approach	How user accesses servlet
Provide the servlet URL to users for direct access,or include an HREF link to the servlet URL on the Web site	Type the servlet URL in a browser, or follow a link to it
Call servlet from an HTML form	Fill out an HTML form and submit it to the servlet for processing
Call servlet from a JSP file	Open a JSP page that invokes the servlet

4.2.4.4.2.1: Invoking servlets within SERVLET tags

The user can invoke a servlet from an HTML page containinga SERVLET tag.

⚠ This method is **not** recommended becauseit only works with JSP .91, and withdrawal of JSP .91 supportis imminent.

The application developer includes a SERVLET tag in an HTML page tocause a *server-side include* in which everything between andincluding the two SERVLET tags is overlaid withthe output from the servlet called within the tags. As the name suggests,all processing occurs on the server. The resulting HTML page is sent to the user.

The following HTML fragment shows how to use the SERVLET tag:

```
<SERVLET NAME="myservlet" CODE="myservlet.class" CODEBASE="url" initparm1="value"><PARAM
NAME="parm1" VALUE="value"></SERVLET>
```

Servlet attributes

The loaded servlet will assume a servlet name matching the name specifiedin the NAME attribute. Subsequent requests for that servlet name will invokethe same servlet instance.

SERVLET attribute	Description
NAME	It specifies a servlet name, or the servlet class name without the .class extension
CODE	It specifies the servlet class name
CODEBASE	It specifies a URL on a remote system from which theservlet is to be loaded

Using the NAME and CODE attributes provides flexibility. Either or bothcan be used. CODEBASE is optional. With the product, it is recommended thatyou specify both NAME and CODE, or only NAME if the NAME specifies the servletname. If only CODE is specified, an instance of the servlet with NAME=CODE iscreated.

If both NAME and CODE are present, and NAME specifies an existing servlet,the servlet specified in NAME is always used. Because the servletcreates part of an HTML file, the application developer will probably usea subclass of the HttpServlet classwhen creating the servlet and override the doGet() method, because GET is thedefault method for providing information to the servlet. Another optionis to override the service() method.

Parameter attributes

In the previous tagging example, *parm1* is thename of an initialization parameter and *value* is the value of theparameter.

PARAM attribute	Description
NAME	It specifies the name of the parameter for use with this particularinvocation of the servlet
VALUE	It specifies the value of the parameter for use with this particularinvocation of the servlet

You can specify more than one set of name-valuepairs. Use the getInitParameterNames() and getInitParameter() methodsof the ServletConfig object (which the servlet engine passes to the servlet's init()method) to find a string array of parameter names and values.

In the example, *parm1* is the name of a parameter that is set to*value* after the servlet is initialized. Because the parametersset using the <PARAM> tag are available only through methods of aRequest object, the server must have invoked the servlet service() method,passing a request from a user.

To get information about the user'srequest, the application developer should usethe getParameterNames(), getParameter(), and getParameterValues()methods.

The parameters set within the <PARAM> attribute are fora specific invocation of the servlet. If a second JSP file invokesthe same servlet with no <PARAM> parameters, the <PARAM>parameters set by the first invocation of the servlet are not available to thesecond invocation of the servlet.

In contrast, servlet initialization parameters are persistent. Suppose a clientinvokes the servlet by invoking an JSP file containing some initializationparameters. Assume that a second client invokes the same servletthrough a second JSP file, which does not specify any initializationparameters. The initialization parameters set by the first invocationof the servlet remain available and unchanged through all successiveinvocations of the servlet through any other JSP file. The initializationparameters cannot be reset until after the destroy() method has beencalled on the servlet.

For example, if another JSP file specifies adifferent value for an initialization parameter but the servlet is alreadyloaded, the value is ignored.

4.2.4.4.2.2: Invoking servlets within JSP files

Users can invoke servlets from within JavaServer Page (JSP)files. Application developers should consult the JavaServer Pages (JSP)reference for a complete description of the JSP syntax.

To invoke a JSP file, a user can either:

- Use a Web browser to open the JSP file
- Use a Web browser to invoke a servlet that invokes the JSP file

4.2.5: Using the Bean Scripting Framework

Most Web developers are familiar with using scripting languages to generateuser-cued HTML pages or to create new browser windows.

The *Bean Scripting Framework* (BSF) enables developers to usescripting language functions in their Java, server-side applications. It also extends scripting languages so that existing Java classes and Java beanscan now be invoked from that language.

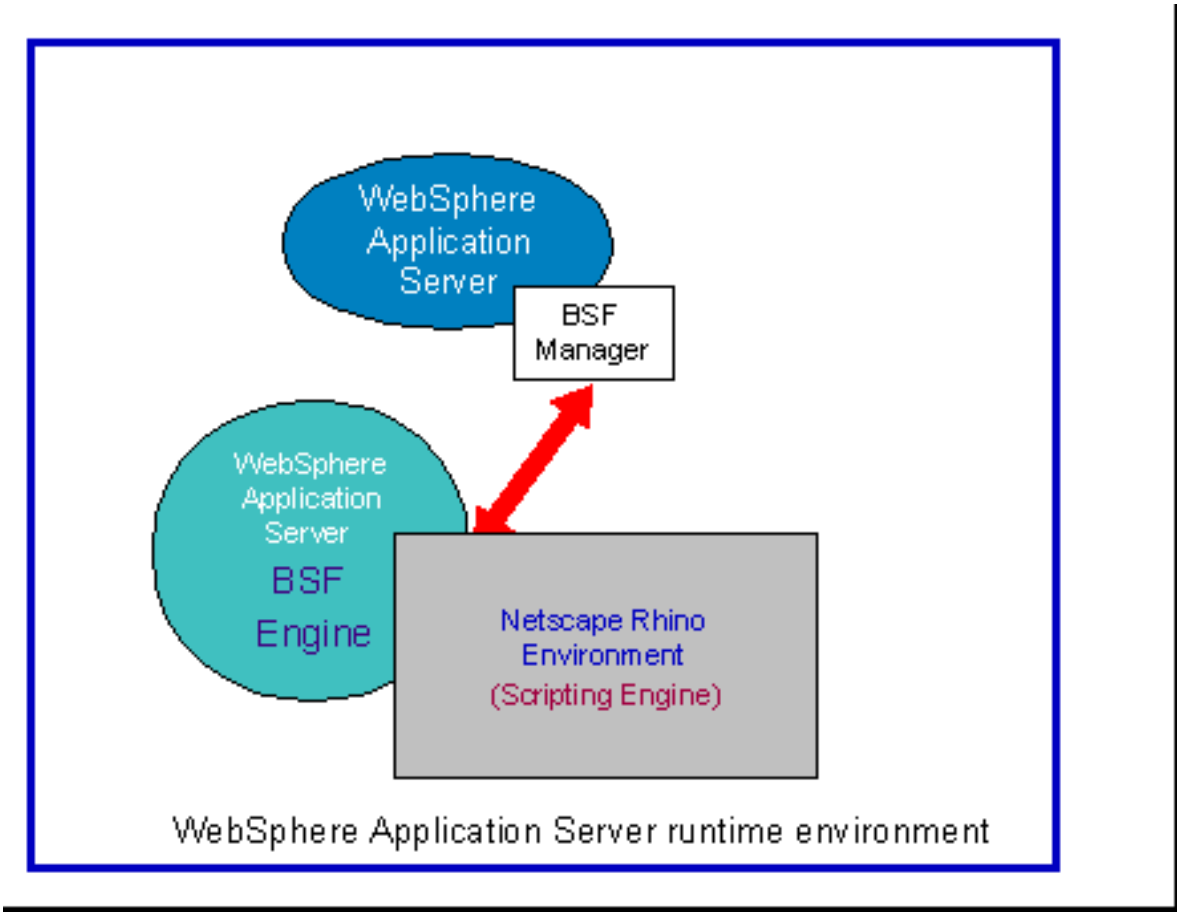
With BSF, scripts can now create, manipulate and access values from Java objects and, conversely, Java programs can now evaluate and accessresults from scripts.

BSF components:

WebSphere Application Server provides the *Bean Scripting Framework* (BSF),which consists of a BSF Manager and a BSF Engine, and a scripting engine which is the *Rhino* version 1.5 environment from Netscape.

JavaScript from Netscape is the only language supported by WebSphere Application Server's implementation of BSF.

The relationship of the BSF components is illustrated in the following graphic:



Features of BSF:

The *BSF Manager* is a bean that provides scripting services for the applicationand support services for the scripting engine to enable it to interact with the JVM.

The *BSF Engine* is an interface that allows a specific scripting language,in this case Netscape's JavaScript, to become part of the bean scripting framework.

Visit the BSF project Web sitefor news on the latest updates to BSF functionality.

See article "BSF examples and samples" when you are ready to delve into programming examples.

4.2.5.1: BSF examples and samples

There are no WebSphere Application Server implementation restrictions on using BSF.Invoke BSF as you would any other Web application, using the instructions in thearticle Installing application files to administer your application.

To test these code samples, from a Browser window,copy the code samples and paste them into your own file. You can use any file name, but the file extension must be **.jsp**. To see the results, the file must be servedfrom a server with a JSP engine, such as WebSphere Application Server.

The following steps and code samples describe how to implement BSF:

1. Create a JSP file
2. Change the Java code to JavaScript
3. Add the required BSF tag as illustrated in the View 2 sample
4. Add the file to the Web application document root directory
5. Invoke the code.

See the file JSP access models for more JSP information.

1. Create a JSP file that looks like this next example:

```
<html>  <head>     <title> Temperature Table using Java >/title>  </head>  <body>  <h1>Temperature
Table using Java</h1>  <p>American tourists visiting Canada can use this handy temperature  table
which converts from Fahrenheit to Celsius:  <br> <br>  <table BORDER COLS=2 WIDTH="20%" >  <tr
BGCOLOR="#FFFF00">  <th>Fahrenheit</th>  <th>Celsius</th>  </tr> <%   for (int i=0; i<101; i+=10) {
out.println ("<tr ALIGN=RIGHT BGCOLOR=\"#CCCCCC\">");    out.println ("<td>" +  i + "</td>");
out.println ("<td>" + ((i - 32)*5/9) + "</td>");    out.println ("</tr>");   } %> </table>  <p><i>
<%= new java.util.Date () %> </i></p>  </body>  </html>
```

2. Change the Java code in the previous file to JavaScriptso the file now looks like the following example:

```
<%@ page language="javascript" %>
<html>  <head>     <title> Temperature Table using JavaScript >/title>  </head>  <body>
<h1>Temperature Table using JavaScript</h1>  <p>American tourists visiting Canada can use this handy
temperature  table which converts from Fahrenheit to Celsius:  <br> <br>  <table BORDER COLS=2
WIDTH="20%" >  <tr BGCOLOR="#FFFF00">  <th>Fahrenheit</th>  <th>Celsius</th>  </tr> <%   for (var
i=0; i<101; i+=10) {     out.println ("<tr ALIGN=RIGHT BGCOLOR=\"#CCCCCC\">");    out.println
("<td>" +  i + "</td>");    out.println ("<td>" + Math.round((i - 32)*5/9) + "</td>");
out.println ("</tr>");     } %> </table>  <p><i> <%= new java.util.Date () %> </i></p>  </body>
</html>
```

3. The only BSF-specific tag that is required in your file is
   <%@ page language="javascript" %>
   This tag identifies the language to BSF. View 2 illustrates where thistag is located in the file.

4.2.8: Programming high performance Web applications

This article offers tips and guidelines for creating Web applicationsthat perform well in the WebSphere Application Server environment. It alsoincludes enterprise beans tips as appropriate.

Best Practices White Paper

You are encouraged to refer to the IBM White Papers site for a White Paper entitled "WebSphere Application Server Development Best Practices for Performance and Scalability."

Use calls to ServletContext.log() sparingly

Each calls to the ServletContext.log() method is recorded in the WebSphereadministrative database. Overusing calls to this method will seriously degrade performance. Limit calls to only those events that should beconsidered SeriousEvents.

4.2.9: Setting language encoding in Web applications

This article provides tips and guidelines for using various lanaguageencodings in WebSphere applications.

Viewing encoded output streams

The correct encoding must be used for sending characters from a servlet or JSP file to a Web browser. If Double Byte Character Set (DBCS) output streams are notbeing displayed correctly in Web browser clients accessing applicationsin the WebSphere Application Server environment, consider the following solutions.

Configure application servers

To configure an entire application serverto use a particular encoding, add the following command line argument to the application server:

`-Ddefault.client.encoding=`*`encoding`*

where *encoding* is the encoding of your choice.The default value of default.client.encoding is `"UTF-8"`.

Specify encodings for particular JSP files or servlets

To specify the character encoding of the resulting stream, insert the encoding statement in the JSP file:

`<%@ page contentType="`*`encoding`* `%>`

where *encoding* is the encoding of your choice.

For a servlet, add the statement:

```
HttpServletResponse.setContentType("text/html;charset=Big5");
```

The above example assumes that you want to use the Big5 character set to encode your servlet output. You can substitute a different encoding.

Run the entire product in a locale that supports the encoding

WebSphere Application Server can be run in the locale which supports that encoding. For instance, it can be run in the zh-TW (Traditional Chinese Locale) which supports the Big5 encoding.

Specify the locale when you install IBM WebSphere Application Server.

Check the browser and operating system support

To display encoded characters in your browser, a user must install the support for that language on his or her operating system.

For Windows NT, accomplish this by opening Internet Explorer and clicking Tools -> Windows Update. This will take you to the Windows Update site, which provides a list of languages for which you can install support on your machine. From the list, select the languages that you want your system to support.

Writing to and from databases

When writing data to a database, a servlet (or other applicationcomponent) must use the same encoding as that data stored in the database. Similarly,the database and a servlet obtaining data from the database must usethe same encoding.

For example, a servlet writing data in a Korean encoding cannot writethe data into a database configured with an English encoding, unless theservlet first converts the data to an English encoding. The same is trueof any two encodings.

4.2.10: Converting WAR files to Web applications (wartowebapp script)

This script converts a WAR file into a format that can be used by a standalone Servlet engine runtime outside of the full WebSphere administration system. It will not affect the operation of the WebSphere Application Server by configuring new web applications on the server. This script is intended to provide a means for developers to execute a web application from XML on a Servlet Engine inside a development runtime environment (such as VisualAge for Java). The wartowebapp script converts the web.xml file into a .webapp format file used by the Servlet engine.

The command line syntax is as follows:

```
wartowebapp [war filename] [webapp destination]   [virtual host name] [webapp path] [webapp name]
```

where:

- **war filename** - full path to the WAR file
- **webapp destination** - directory where web application will be rooted (a subdirectory will be created that matches the specified web app name)
- **virtual host** - name of the virtual host you wish this application to be accessible from
- **webapp path** - the context path of the web application
- **webapp name** - the name of the web application you wish to use

Example:

```
wartowebapp c:\temp\servlet-tests.war c:\websphere\appserver\hosts\default_host   default_host
/servlet-tests servlet_tests
```

If security is enabled, modify the sas.client.props file toauthenticate using the properties file, instead of by prompting. See article 6.6.18 for details.

4.4: Personalizing applications

Personalization describes a range of features that enable applications to treat visitors as particular individuals. For a really simple example, consider a site that issues the message "Hello, John Smith" when the customer John Smith logs onto the site.

Personalized service can give your Web site a competitive edge, much like a good customer service team can add value to human-to-human interactions at your physical site and keep customerscoming back. Personalization can also increase the chance that your Web site presents a user with content that is of particular interest to that person.

For an e-business site, personalization can be fairly necessary, even if it does not go so far as to call customers by name. For example,suppose several Web site visitors are performing various transactions concurrently. Applications need some way to group each user's transactions into a unit that is separate from the transactions of other users. *Session tracking* provides such functionality.

See articles 0.11 and 0.12 to learn about two complementary personalizationapproaches supported by IBM WebSphere Application Server -- trackinguser sessions and maintaining user profiles.

If you are already familiar withthe concepts, skip ahead to 4.4.1 and 4.4.2 for programming details. See 6.6.11 and 6.6.12 to take a look at the administrative aspects.

For additional capability offered by the IBM WebSphere Personalization product,visit the following Web site:

`http://www.ibm.com/software/webservers/personalization/`

4.4.1: Tracking sessions

IBM WebSphere Application Server provides a service for trackinguser sessions -- the Session Manager. The service is provided in the form of IBM classes and packages.

The key activities for session tracking are summarized.

1. Become familiar with the programming model for accessing session support from servlets. See article 4.4.1.1 for an overviewwith links to details about security, clustering, limitations,and other topics.
2. Create or modify your own servlets to use session support to maintain sessions onbehalf of Web applications.

   Follow the model outlined in the previous step.
3. Ensure the administrator appropriately configures Session Managers in theadministrative domain. See article 6.6.11.
4. Adjust configuration settings and perform other tuning activities foroptimal use of sessions in your environment. See article 4.4.1.1.7.

4.4.1.1: Session programming model and environment

The session lifecycle, from creation to completion, is as follows:

1. Get the HttpSession object
2. Store and retrieve user-defined data in the session
3. (Optional) Output an HTML response page containing data from the HttpSession object
4. (Optional) Notify Listeners
5. End the session

The steps are described in detail below. This information, combined with thecoding example SessionSample.java, provides a programming model for implementing sessions in your own servlets.

It is also recommended that you read the topics listed in therelated information. They can influence how you implement sessions in your own servlets.

Lifecycle in detail

1. **Get the HttpSession object.**

   To obtain a session, use the getSession() method of thejavax.servlet.http.HttpServletRequest object in the Java Servlet 2.1 API.

   When you first obtain the HttpSession object, the Session Manager uses one of two ways to establish tracking of the session: cookies or URL rewriting. See section 4.4.1.1.1 for a discussion to help you decide which is more appropriate for your situation.

   Assume the Session Manager uses cookies. In such a case, the Session Manager creates a unique session ID and typically sends it back to the browser as a *cookie*. Each subsequent request from this user (at the same browser) passes the cookie containing the session ID, and the Session Manager uses this to find the user's existing HttpSession object.

   In Step 1 of the code sample, the Boolean(create) is set to `true` so that the HttpSession is created if it does not already exist. (With the Servlet 2.1 API, the javax.servlet.http.HttpServletRequest.getSession() method with no boolean defaults to `true` and creates a session if one does not already exist for this user.)
2. **Store and retrieve user-defined data in the session.**

   After a session is established, you can add and/or retrieve user-defined data to the session. The HttpSession object has methods similar to those in java.util.Dictionary for adding, retrieving, and removing arbitrary Java objects.

   If Java objects will be added to a session, be sure to place the class files for those objects in the application server classpath or in the directory containing other servlets used in WebSphere Application Server. In the case of session clustering, this applies to every node in the cluster.

   In Step 2 of the code sample, the servlet reads an integer object from the HttpSession, increments it, and writes it back. You can use any name to identify values in the HttpSession object. The code sample uses the name sessiontest.counter.

   Because the HttpSession object is shared among servlets that the user might access, consider adopting a site-wide naming convention to avoid conflicts.

3. **(Optional) Output an HTML response page containing data from the HttpSession object.**

   In order to provide feedback to the user that an action has taken place during the session, you may wish to pass HTML code to the client browser that indicates that an action has occurred.

   For example, in step 3 of the code sample the servlet generates a Web page that is returned to the user and displays the value of the sessiontest.counter each time the user visits that Web page during the session.

4. **(Optional) Notify Listeners.**

   Objects stored in a session that implement the javax.servlet.http.HttpSessionBindingListener interface are notified when the session is preparing to end, that is, about to be invalidated. This notice enables you to perform post-session processing, including permanently saving to a database data changes made during the session.

5. **End the session.**

   You can end a session:

   - Automatically with the Session Manager, if a session has been inactive for a specified time. The administrative clients provide a way to specify the amount of time after which to invalidate a session.
   - By coding the Servlet to call the invalidate() method on the session object.

4.4.1.1.1: Deciding between session tracking approaches

Suppose a servlet implementing sessions is receiving requests from three differentusers. For each user request, the servlet must be able to figure out the session towhich the user request pertains. Each user request belongs to just one of the threeuser sessions being tracked by the servlet. Currently, the product offers two ways to address the problem.

Cookies provide a fairly simple approach to tracking sessions. Because cookies donot work in all situations, URL rewriting provides an alternative. If the administratorenables URL rewriting, it will be used, even in situations in which cookies are feasible.When deciding whether to use URL rewriting, carefully review the codingrequirements it imposes on applications that require session support.

Cookies

When session management is enabled and a client makes a request, the HttpSession object is created and the session ID is sent to the browser as a cookie. On subsequent requests, the browser sends the session ID back as a cookie and the Session Manager uses the cookie to find the HttpSession associated with the user.

URL rewriting

There are situations in which cookies will not work. Some browsers do not support cookies. Other browsers allow the user to disable cookie support. In such cases, the Session Manager must resort to a second method, URL rewriting, to manage the user session.

With URL rewriting, links returned to the browser or redirect have the session ID appended to them. For example, the following link in a Web page:

```
<a href="/store/catalog">
```

is rewritten as:

```
<a href="store/catalog;jsessionid=DA32242SSGE2">
```

When the user clicks the link, the rewritten form of the URL is sent to the server as part of the client's request. The servlet engine recognizes

```
;jsessionid=DA32242SSGE2
```

as the session ID and saves it for obtaining the proper HttpSession object for this user.

Note: Do not make assumptions aboutthe length or exact content of the ID that follows the equals sign (=).In fact, the IDs are longer than what this example shows.

To use URL rewriting, applications must follow certain coding guidelines. Also,special preparation is required. See the related information for details.

4.4.1.1.1.1: Using cookies to track sessions

No special programming is required to track sessions with cookies. Followthe programming model and example described in section 4.4.1.1.

4.4.1.1.1.2: Using URL rewriting to track sessions

An application that uses URL rewriting to track sessions must adhere to certain programming guidelines. The application developer needs to:

- Program session servlets to encode URLs
- Supply a servlet or JSP file as an entry point to theapplication
- Avoid using plain HTML files in the application

Program session servlets to encode URLs

Depending on whether the servlet is returning URLs to the browser orredirecting them, include either encodeURL( ) or encodeRedirectURL( ) inthe servlet code. Here are examples demonstrating what toreplace in your current servlet code.

Rewrite URLs to return to the browser

Suppose you currently have this statement:

```
out.println("<a href=\"/store/catalog\">catalog<a>");
```

Change the servlet to call the encodeURL method before sending the URL to the output stream:

```
out.println("<a href=\"");out.println(response.encodeURL
("/store/catalog"));out.println("\">catalog</a>");
```

Rewrite URLs to redirect

Suppose you currently have the following statement:

```
response.sendRedirect ("http://myhost/store/catalog");
```

Change the servlet to call the encodeRedirectURL method before sending the URL to the output stream:

```
response.sendRedirect (response.encodeRedirectURL ("http://myhost/store/catalog"));
```

The encodeURL() and encodeRedirectURL() methods are part of the HttpServletResponse object. These calls check to see if URL rewriting is configuredbefore encoding the URL. If it is not configured, they return the original URL.

With Version 3.x, if URL encoding is configured and response.encodeURL() orencodeRedirectURL() is called, the URL is encoded, even if thebrowser making the HTTP request processed cookies. This differs from thebehavior in previous releases, which checked for the condition and haltedURL encoding in such a case.

You can also configure session support to enable protocol switch rewriting.When this option is enabled, the product encodes the URL with the session IDfor switching between HTTP and HTTPS protocols.For details, see the Related information.

Supply a servlet or JSP file as an entry point

The entry point to an application (such as the initial screen presented) may not require the use of sessions. However, if the application in general requires session support (meaning some part of it, such as a servlet, requires session support) then after a session is created, all URLs must encoded in order to perpetuate the session ID for the servlet (or other application component) requiring the session support.

The following example shows how Java code can be embedded within a JSP file:

```
<%response.encodeURL ("/store/catalog");%>
```

Avoid using plain HTML files in the application

Note that to use URL rewriting to maintain session state, do notlink to parts of your applications from plain HTML files (files with .html or .htm extensions).

The restriction is necessary because URL encoding cannot be used in plain HTML files. To maintain state using URL rewriting, every page that the user requests during the session must have code that can be understood by the Java interpreter.

If you have such plain HTML files in your application (or Web application) and portions of the site that the user might access during the session, convert them to JSP files.

This impacts the application writer because maintaining sessions with URL rewriting requires that each servlet in the application must use URL encoding for every HREF attribute on <A> tags, as described previously.

Sessions will be lost if one or more servlets in an application do not call theencodeURL(String url) orencodeRedirectURL(String url) methods.

4.4.1.1.2: Controlling write operations to persistent store

You can manually control when modified session data can be persisted to the datastore by using the sync() method in the interface com.ibm.websphere.servlet.session.IBMSession, which extends the javax.servlet.http.HttpSession interface.

By calling sync() from the service() method of a servlet, you send any changes in the session to the database.

If neither the manual updatenor the time-based write option is enabled,the sync() call performs no updates.It merely returns.

Ideally, call sync() after all updates have been made to the session andthe session will not be accessed any more. In other words, wait until theend of the servlet service() method to call sync().

4.4.1.1.3: Securing sessions

HTTP sessions and security are integrated in IBM WebSphere Application Server. When WebSphere security is enabled,all resources fromwhich sessions are created or accessed must be eithersecured or unsecured. You cannot mix secured and unsecured resources.

Security integration rules for HTTP sessions

- Sessions in unsecured pages are treated as accesses by "anonymous" users.
- Sessions created in unsecured pages are created under the identity of that "anonymous" user.
- Sessions in secured pages are treated as accesses by the authenticated user.
- Sessions created in secured pages are created under the identity of the authenticated user. They can only be accessed in other secured pages by the same user. To protect these sessions from use by unauthorized users, they cannot be accessed from an insecure page.

Programmatic details and scenarios

IBM WebSphere Application Server maintains the security ofindividual sessions.

An identity or user name, readable by thecom.ibm.websphere.servlet.session.IBMSession interface, is associated witha session. An unauthenticated identity is denoted by the user name "anonymous." IBM WebSphere Application Server includes the com.ibm.websphere.servlet.session.UnauthorizedSessionRequestExceptioninterface,which is used when a session is requested without the necessary credentials.

The Session Manager uses the WebSphere security infrastructure todetermine the authenticated identity associated with a client HTTP requestthat either retrieves or creates a session. WebSphere security determinesidentity using certificates, LPTA, and other methods.

After obtaining the identity of the current request, the Session Managerdetermines whether the session requested using a getSession() call should be returned.

To turn off the association of user identity with a session,set the following system property:

```
HttpSessionSecurity=false
```

The following table lists possible scenarios whose outcomes depend on whetherthe HTTP request was authenticated and whether a valid session IDand user name was passed to the Session Manager.

	Unauthenticated HTTP request is used to retrieve a session	**HTTP request is authenticated, with an identity of "FRED" used to retrieve a session**
No session ID was passed in for this request, or the ID is for a session that is no longer valid	A new session is created. The user name is "anonymous"	A new session is created. The user name is "FRED"

A session ID for a valid session is passed in. The current session user name is "anonymous"	The session is returned.	The session is returned. TheSession Manager changes the user name to "FRED"
A session ID for a valid session is passed in. The current session user name is "FRED"	The session is not returned. UnauthorizedSessionRequest Exception is thrown*	The session is returned.
A session ID for a valid session is passed in. The current session user name is "BOB"	The session is not returned. UnauthorizedSessionRequestException is thrown*	The session is not returned. UnauthorizedSessionRequestException is thrown*

* UnauthorizedSessionRequestException is sent to the application server error log.If getSession(true) was specified in the servlet, a new session is created by using the current authenticated name.

4.4.1.1.4: Deciding between single-row and multirow schema for sessions

Using the single-row schema, each user session maps to a single database row. Using the multirow schema, each user session maps to multiple database rows.(In a multirow schema, each session attribute maps to a database row.)

In addition to allowing larger session records, using multirow schemacan yield performance benefits, as discussed in article 4.4.1.1.7.3. However, itrequires a little work to switch to from single-row to multirow schema, as shown in the instructions below.

Switching from single-row to multirow schema

To switch from single-row to multirow schema for sessions:

1. Modify the Session Manager properties to switch from single to multirowschema.
2. Manually drop the database table or delete all the rows in the databasetable that the product uses to maintain HttpSession objects.

   To drop the table:

   1. Determine which data source configuration the Session Manager is using.
   2. In the data source configuration, look up the database name.
   3. Use the database facilities to connect to the database.
   4. Drop the SESSIONS table.
3. Restart the Session Manager.

Coding considerations and test environment

Consider configuring direct single-row usage to one database and multirow usageto another database while you verify which option suits your application's specific needs.(Do this in code by switching the datasource used;then monitor performance.)

Programming issue	**Application scenario**
Reasons to use single-row	• You can read or write all values with just one record read/write. • This takes up less space in a database, because you are guaranteed that each session is only one record long.
Reasons **not** to use single-row	2-megabyte limit of stored data per session.
Reasons to use multirow	• The application can store an unlimited amount of data; that is, you are limited only by the size of the database and a 2-megabyte-per-record limit. • The application can read individual fields instead of the whole record. When large amounts of data are stored in the session but only small amounts are specifically accessed during a given servlet's processing of an HTTP request, multirow sessions can improve performance by avoiding unneeded Java object serialization.
Reasons **not** to use multirow	If data is small in size, you probably do not want the extra overhead of multiple row reads when everything could be stored in one row.

In the case of multirow usage, design your application data objects not to have references to each other, to

prevent circular references. For example, suppose you are storing two objects A and B in the session using HttpSession.put(..) , and A contains a reference to B. In the multirow case, because objects are stored in different rows of the database, when objects A and B are retreived later, the object graph between A and B is different than stored. A and B behave as independent objects.

4.4.1.1.6: Limitations in session support

- The product does not provide non-JDBC, native access to a database version of session persistence.
- JTA-enabled datasources are not supported for session persistence.
- For now, the administrator should use only the direct-to-database persistence type. The EJB persistence type is intended for securely and reliably accessing a HttpSession outside the scope of a servlet; however, it is not fully functional at this time.

4.4.1.1.7: Tuning session support

IBM WebSphere Application Server session support has features for tuning session performance and operating characteristics, particularly whensessions are persisted in a database. These options allow the administrator flexibility in determining the performance and failovercharacteristics for their environment.

The table summarizesthe features, including whether they apply to sessions tracked in memory, in a database, or either.Click a feature for details about the feature. Some features are easily manipulatedusing administrative settings; others require code or database changes.

Feature or option	Goal	Applies to sessions in memory or database?
Session caching	Minimize database read operations.	Database
Multirow schema	Fully utilize database capacities.	Database
Base in-memory session pool size	Fully utilize system capacity withoutoverburdening system.	Either

4.4.1.1.7.1: Tuning session support: Session persistence

IBM WebSphere Application Server avoids using the database to read in or access the session when it is determined that the entry in the session cache is still the most recently updated copy.To tune the cache,set the base in-memory session pool sizeand allow overflow.

In addition to the cache table itself,the product maintains a list of the most recently used sessions in memory,ordered from least to most recently used.Whenever a session is accessed, it is added to the most-recently-used end of the list.When the cache table becomes full and a session that is not in the cache is accessed, the least recently used session is removed from the cache (but not from the database; the session is still valid until explicitly invalidated or timed out)to make room for the new entry.

This removal occurs whether or not overflow is enabled.However, under heavy-concurrent-access scenarios, multiple new sessions might competefor the space vacated by the single, least recently used entry.

- When overflow is disabled, only one new session is placed in the cache;the others must be reread from the database.To optimize performance,the product does not retry to add the next new sessionby removing the next least recently used entry.
- When overflow is enabled, one new session is added to the base table, and the rest reside in memory in the overflow table. Analysis and customer experience show that the size of this table remains relatively small comparedto the base in-memory session pool size.

It is also important to establish sessionaffinity so that the caching can be most effective. See the Related informationfor details.

4.4.1.1.7.3: Tuning session support: Multirow schema

By default, a single session maps to a single row in the database table used tohold sessions. With this setup, there are hard limits to the amount of user-defined,application-specific data that WebSphere Application Server can access.

IBM WebSphere Application Server supports the use of a multirow schema option in which each piece of application specific data is stored in a separate row of the database. With this setup, the total amount that can be placed in a session is now bound only by the database capacities. The only practical limit that remains is the size of a session attribute object itself.

The multirow schema potentially has performance benefits in certain usage scenarios, such as when larger amounts of data are stored in the session but only small amounts are specifically accessed during a given servlet's processing of a http request. In such a scenario, avoiding unneeded Java object serialization is beneficial to performance.

It should be stressed that switching between multirow and single row is not a trivial proposition. See the Related information for details.

4.4.1.1.7.4: Tuning session support: Write frequency

In the Session Manager, you can configure the frequency for writing session data to the database. This flexibility enables you to weigh session performance gains against varying degrees of failover support. The following options are available in Session Manager for tuning write frequency:

- End of service method (the default) - Write session data at the end of the servlet's service() method call.
- Manual update - Write session data when the servlet calls the IBMSession.sync() method.

When a session is first created,session information is always written to the databaseat the end of the service() call.

End of service method

By default, IBM WebSphere Application Server updates the database with any changes made to the session during the servlet processing of an HTTP request (for example, during the execution of the service() method). These updates minimally include the last access time of the session and typically also include changes affected by the servlet, such as updating or removing application data.

Manual update

With manual updates, the servlet using a session determines when to write session information to the database. Switching to manual updates improves performance when the number of times an HTTP request's processing leads to changing a session (typically its application data) is typically less than the number of times the session is accessed or read in.

When manual update is set, the product session support no longer automatically updates the database at the end of a servlet's service() method.(However, when an HttpSession object is first created, session information is written to the database as part of postprocessing for the servlet request in which the session was created.)The last update times are cached and updated asynchronously prior to checks for session invalidation.

For any permanent changes to the session as part of servlet processing, the servlet code must specifically call the sync() method of the com.ibm.websphere.servlet.session.IBMSession interface.

Programming issue	**Application scenario**
Reasons to use manual update	• You want direct control over when session information is persisted to the database. • The servlets of the application typically read in the session data but do not write it back as much. • The servlets of the application take a long time to finish processing, thereby holding locks on the database records for a long time.
Reasons **not** to use manual update	• You do not want to control persistence of session information by using the IBMSession object, or you prefer that WebSphere explicitly control persistence to the database. • The servlets of the application are writing session information frequently. • Your code must comply completely with the Servlet 2.1 specification. The sync() method is not part of the Servlet specification; it is an IBM extension.

4.4.1.1.7.5: Tuning session support: Base in-memory session pool size

The base in-memory session pool size number has different meanings, dependingon session support configuration:

- When sessions are being stored in memory, sessionaccess is optimized for up to this number of sessions.
- When sessions are being stored in a database, it also specifies the cache size andthe number of last access time updates that are saved in manual update mode.

For persistent sessions,when the session cache has reached its maximum sizeand a new session is requested, Session Manager removes the least recently used sessionfrom the cache to make room for the new one.

General memory requirements for the hardware system, and the usagecharacteristics of the e-business site, will determine the optimum value.

Note that increasing the base in-memory session pools size can necessitateincreasing the heap sizes of the Java processes for the correspondingWebSphere application servers.

Overflow in non-persistent sessions

By default, the number of sessions maintained in memory is specified byBase in-memory session pool size. If you do notwish to place a limit on the number of sessions maintained in memory and allowoverflow, set overflow to *true*.

Allowing an unlimited amount of sessions can potentially exhaust systemmemory and even allow for system sabotage. Someone could write a maliciousprogram that continually hits your site and creates sessions, but ignoresany cookies or encoded URLs and never utilizes the same session from oneHTTP request to the next.

When overflow is disallowed, the Session Manager still returns a session with the HttpServletRequest's getSession(true) method if the memory limit has currently been reached, but it would be an invalid session that is not saved in any fashion.

With the WebSphere extension to HttpSession, com.ibm.websphere.servlet.session.IBMSession, an isOverflow() method returns *true* if the session is such an invalid session. An application could check this and react accordingly.

4.4.1.1.8: Best practices for session programming

When developing new objects to be stored in the HTTP session, make sure to implement the Serializable interface. This enables the object to properly persist session information to the database. An example of this is:

```
public class MyObject implements java.io.Serializable {...}
```

Without this extension, the object will not persist correctly and will throw an error.

When adding Java objects to a session, make sure they are in the correct class path. If Java objects will be added to a session, be sure to place the class files for those objects in the application server class path or in the web application path.In the case of session clustering, this applies to every node in the cluster. Because the HttpSession object is shared among servlets that the user might access, consider adopting a site-wide naming convention to avoid conflicts.Also, if objects are only in the web application class path and more than one web application is sharing sessions, the following restrictions apply:

- You cannot use single-row session persistence, because the applications that do not have the objects in the class path cannot read the session data.
- You cannot have two web applications reading in the same session concurrently (that is, through a multiframed JSP).

Do not store large Object graphs in HttpSession. In most applications, each servlet requires only a fraction of the total session data. However, by storing the data in HttpSession as one large object, an application forces WebSphere to process all of it each time.

Release HttpSession objects when you are finished. HttpSession objects live inside the WebSphere servlet engine until:

- The application explicitly and programmatically releases it using javax.servlet.http.HttpSession.invalidate(); quite often, programmatic invalidation is part of an application logout function.
- The application server destroys the allocated HttpSession object when it expires (default is 1800 seconds or 30 minutes).When session persistence is used, the application server can maintain only a certain number of HttpSession objects in memory. When this limit is reached,the application server simply does not cache any new sessions; session updates are automatically sent back to the databasewithout checking for their presence in the cache.

Do not try to save and reuse the HttpSession object outside of each servlet or JSP. The HttpSession object is a function of the HttpRequest (you can get it only through req.getSession()), and a copy of it is valid only for the life of the service() method of the servlet or JSP. You cannot cache the HttpSession object and refer to it outside the scope of a servlet or JSP.

Use session affinity to help achieve higher cache hits. The plug-in reads the cookie data (or encoded URL) from the browser and helps direct the request to the appropriate application based on the assigned session key. This helps to achieve a greater use of the in-memory cache and reduces hits to the session database.

You can improve performance by not breaking session affinity. Some suggestions to help avoid breaking session affinity are:

- Do not use multiframed JSPs in which the frames point to different web applications. This breaks affinity and will cause separate JVMs to process a session concurrently. When this happens, consistent state cannot be guaranteed.
- When using multiframe JSPs, create the session for the frame page but do not create sessions for the pages within the frame. (See discussion later in this topic.)

When applying security to servlets or JSPs that use sessions, secure all of the pages (not just some). When it comes to security and sessions, it's all or nothing. It does not make sense to protect access to session state only part of the time.When WebSphere security is enabled, all resources from which a session is created or accessed must be either secured or unsecured. You cannot mix secured and unsecured resources.

The problem with securing only a couple of pages is that sessions created in secured pages are created under the identity of the authenticated user. They can be accessed in other secured pages only by the same user. To protect these sessions from use by unauthorized users, they cannot be accessed from an unsecure page. When a request from an unsecure page occurs, access is denied and an UnauthorizedSessionRequestException is thrown. (UnauthorizedSessionRequestException is a run-time exception; itis logged for you.)

Use manual update and sync() in applications that mostly read session data but update infrequently. When an application is using a session, the LastAccess time field is updatedany time data is read from or written to that session. If persistent sessions are being used, this produces a new write to the database. This performance hit can be avoided by using manual update and having the record written back to the database only when data values are updated, not on every read or write of the record.To use manual update, you first need to turn it on in the Session Manager. In addition, the application

code must use com.ibm.websphere.servlet.session.IBMSession instead of the generic HttpSession class. Within IBMSession, the sync() methodtells the application server that the data in the session object should be written out to the database. This enables the developer to improve overall performance by having the session information persist only when necessary.

When using multiframe Java Server Pages (JSP), create the session for the frame page (JSP) but do not create it for the pages (JSPs) within the frame. By default, JSPs create HTTPSession objects by calling the `request.getSession(true)` method. By doing this, each page in the browser is requesting a new session, but only one session is used per browser instance. You can use

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

to turn off the automatic session creation. Then if the page needs to access session information, use

```
<% HttpSession session = javax.servlet.http.HttpServletRequest.getSession(false); %>
```

to get the already existing session that was created by the frame JSP. This enables you to not break session affinity on the initial loading of the frame pages.

Implement the following suggestions to achieve high performance:

- Use IBM WebSphere Edge Server, taking advantage of its affinity options.
- If your applications do not change the session data frequently, use manual update and the sync() function to efficiently persist session information.
- Keep the amount of data stored in the session as small as possible. With the ease of using sessions to hold data, sometimes too much data is stored in the session objects. A proper balance of data storage and performance must be determined to effectively use sessions.
- Use a dedicated database for the session database. Do not use the WebSphere repository database or another application's database. This helps to avoid contention for JDBC connections and enables better database performance.

For more information, see the following IBM documents on the Web:

- "WebSphere Application Server: Best Practices using HTTP Sessions,"by David Draegar and Jay Toogood.This article is available from the DeveloperWorks site.
- "WebSphere Application Server Development Best Practices for Performance and Scalability," by Harvey W. Gunther. This IBM white paper is available from the Library section of theWebSphere Application Server product site.

4.4.2: Keeping user profiles

IBM WebSphere Application Server provides a service for processing user profiles, called the *User Profile Manager*.

The key activities for implementing user profilesare summarized. For more information about each point, consult the Related information below.

1. Customize the user profile support as necessary. Options include:
    - Using the data representation class with exactly the name/value pairs it currently allows (no action required)
    - Extending the data representation class to allow additional, arbitrary name/value pairs
    - Adding columns to the base user profile representation

    Basically, you need to evaluate whether the user profilerepresentation provided by IBM represents the kind of data youwould like to keep about your users. You might find it desirableto customize the IBM user profile support in one or more of theabove ways.
2. Create or modify servlets to use the User Profile Manager and related user profile support classes to maintain user profiles on behalf of Web applications.
3. Ensure the administrator appropriately configures User Profile Managers in the administrative domain.

    If the programmer andadministrator are not the same person, the programmer might needto provide settings information to the administrator, based onhow the programmer implemented user profiles.

4.4.2.1: Data represented in the base user profile

WebSphere Application Server provides a base implementation for data representation in user profiles through the interfacecom.ibm.websphere.userprofile.UserProfile.

The interface includes these columns corresponding to fields for demographic data on individual users:

- Address (first line)
- Address (second line)
- First Name
- Surname
- Day phone number
- Night phone number
- City
- Nation
- Employer
- Fax number
- Language
- Email address
- State/Province
- Postal code

4.4.2.2: Customizing the base user profile support

The application developer has a few options for customizing the user profile supportprovided by IBM WebSphere Application Server. The Related information provides instructionsand additional details about each option.

Extend the data represented in user profiles

As discussed in section 4.4.2.1, the base implementation allows Web applications to maintain several pieces of data about users. The data representation canbe extended to allow the collection of arbitrary name/value pairs.

Adding columns to the base user profile implementation

Application developers can customize userprofiles by adding columns to the base user profile implementation. Adding new columns is accomplished by implementing the interface:

`com.ibm.websphere.userprofile.UserProfileExtender`

and extending the base class:

`com.ibm.servlet.personalization.userprofile.UserProfile`

4.4.2.2.1: Extending data represented in user profiles

Use following interface withcom.ibm.websphere.userprofile.UserProfileExtender to extend a user profile hash table:

`com.ibm.websphere.userprofile.UserProfileProperties`

This enables you to place arbitrary name/value pairs in theuser profile.Extending the hash table is similar to using the java.util.Dictionaryclass in the base JDK 1.x or any of the classes that extend it.

4.4.2.2.2: Adding columns to the base user profile implementation

The base implementation of the user profile is contained in the class:

`com.ibm.servlet.personalization.userprofile.UserProfile`

It contains the columns discussed in section 4.4.2.1. The application developer can add columns to the base implementation,but cannot delete columns from it.

Adding columns is a two-step process, as follows:

1. Extend the UserProfile class.
2. Modify your existing servlets to use the new columns.

Several examples are available to demonstrate how to extend thebase user profile implementation and utilize the extension with aservlet.

Example	Description
UPServletExample.java	Demonstrates how a servlet opens a user profile and prints the fields contained within it
UserProfileExtendedSample.java	Shows how to extend the UserProfile class to add a column to the user profile for a cellular phone number. The WebSphere administrator needs to configure the User Profile Manager to point tothe extended class.
UPServletExampleExtended.java	Shows how to modify the UPServletExample servlet to include the cellular phone number in the output
UserProfileExtended.java	Shows how to extend a hash table to place arbitraryname/value pairs into the user profile
UPServletExtended.java	Shows how to extend the servlet. When any of the newly added columnsare removed or replaced, look for the table named "USERPROFILE" in the database to which the user profile is configured and drop that table.

The examples are encoded in HTML for viewing in a browser.The documentation directory also contains non-HTML versions (.javafiles) that are ready for use.

4.4.2.3: Accessing user profiles from a servlet

Servlets and other application building blocks requiring user profile support should make calls to the class: `com.ibm.websphere.userprofile.UserProfileManager`

The class supports the following functions:

- Creating and deleting user profiles
- Getting and updating (cached and immediate) to and from the database
- Getting user profiles for read-only tasks
- Performing queries on database columns

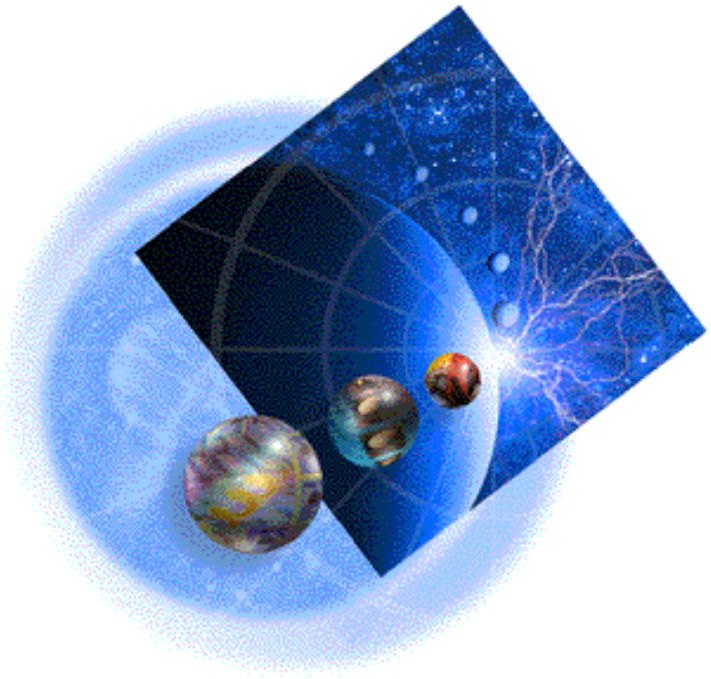
4.5: Employing pervasive computing

The IBM WebSphere Everyplace Suite Solutions Web siteprovides a set of pervasive computing services.

The IBM WebSphere Everyplace Suite is a comprehensive, integrated software platform for extending the reach of e-business applications,enterprise data, and Internet content into the realm of pervasive computing.

See the WebSphere Everyplace Suite: Getting Started manualfor information on implementing WebSphere Everyplace Suite Solutions in WebSphere Application Server.

WebSphere Application Server Samples



The Samples gallery offers a set of small generic samples that show you how to perform common Web application tasks, provide reusable components and demonstrate handy techniques.

The gallery also includes the YourCo intranet Web site, which brings many of the small samples together in common applications.

Once installed on your local machine, the Samples are located at:

`http://localhost/WSsamples/index.html` if your database is DB2
or
`http://localhost/WSsamplesIDB/index.html` if your database is InstantDB

Open the above URL in your Web browser, follow the database configuration instructions, and try the Samples.

The above links will not work if:

- The Samples are not installed on the machine onwhich you are viewing this documentation ("localhost").
- Your Web server is not running.
- You are viewing this documentation from the IBM Web site insteadof viewing locally installed documentation.

If you don't find the Samples on your localhost, confirm their installation. The Samples are an option in the product installation. See the installation documentation for a varietyof case-specific installation steps.