Sun Certified Web Component Developer Study Guide - v2

Authors Note

The following document was put together as a guide to study for the Sun Certified Web Component Developer Exam. This document is actually a formatted version of the study notes that were used to study for the certification exam.

Most of the following document comes straight from the Servlet 2.3 and JSP 1.2 specifications, ***literally cut and paste***!

The intent of this document is to share resources with other Java developers and should be freely distributed. If this document is used, please feel free to drop the author a message with your comments.

****Please Note*****
With regards to Objective *Section 8.5* about the order of events of a Jsp, the first version of this guide was incorrect. It has been updated thanks to feedback received from www.javaranch.com.

I appreciate the feedback that I have received up to this point. Remember this document was intended to filter the important (potentially tested) parts from the Jsp and Servlet specification and not as an all-encompassing guide on how to pass every possible question on the exam.

Thanks

Please forward comments to:

Ken Zrobok
SCJP2, SCJD2, SCWCD
KZrobok@Setfocus.com

Java Developer / Trainer
http://www.SetFocus.com

Servlets

The Servlet Model

1.1 For each of the HTTP methods, GET, POST and PUT, identify the corresponding method in the HttpServlet class.

HttpServlet is an abstract class that requires sub classing to create an HTTP servlet suitable for a Web site. HttpServlet must override at least one method, usually one of the following:

doGet, if the servlet supports HTTP GET requests
doPost for the HTTP POST requests
doPut for the HTTP put requests

There is no reason to override the class's **service** method since **service** handles HTTP requests by dispatching them to handler methods for each type of HTTP request.

1.2 For each of the HTTP methods, GET, POST and HEAD identify the triggers that might cause a browser to use the method and identify benefits or functionality of the method.

HTTP POST method allows the client to send data of unlimited length to the Web server a single time and is useful when posting information such as credit card numbers. The information is usually posted from a HTML form.

HTTP GET method retrieves whatever information (in the form of an entity) is identified by the Request-URI. If the Request-URI refers to a data-producing process, it is the produced data which shall be returned as the entity in the response and not the source text of the process, unless that text happens to be the output of the process.

HTTP HEAD method is a GET request that returns no body in the response, on the request header fields. The client sends a HEAD request when it wants to see only the headers of a response, such as Content-Type or Content-Length. The HTTP HEAD method counts the number of output bytes in the response to set the Content-Length header accurately. This method can be used for obtaining meta-information about the entity implied by the request without transferring the entity-body itself. This method is often used for testing hypertext links for validity, accessibility, and recent modification.

The servlet container must write the headers before committing the response, because in HTTP the headers must be sent before the response body.

1.3 For each of the following operations, identify the interface and method name that should be used:

Retrieve HTML form parameters from the request

Request parameters are strings sent by the client to a servlet container as part of a request.

When the request is an *HttpServletRequest*, the attributes are populated from the URI query string and posted form data. The parameters are stored by the servlet container as a set of name-value pairs.

Multiple parameter values can exist for any given parameter name. The following methods of the ServletRequest interface are available to access parameters:

- **getParameter**
- **getParameterNames**
- **getParameterValues**

The ***getParameterValues*** method returns an array of String objects containing all the parameter values associated with a parameter name. The value returned from the ***getParameter*** method must be the first value in the array of String objects returned by ***getParameterValues***.

All form data from both the query string and the post body are aggregated into the request parameter set. The order of the aggregation puts the query string data before post body data. For example, if a request is made with a query string of a=hello and a post body of a=goodbye&a=world, the resulting parameter set would be ordered a=(hello, goodbye, world).

Retrieve a servlet initialization parameter

The *ServletConfig* interface has the **getInitParameter(String)** as well as the **getInitParameterNames()** methods.

public String ***getInitParameter***(String)

Returns a String containing the value of the named initialization parameter, or null if the parameter does not exist.

public Enumeration **getInitParameterNames**()

Returns the names of the servlet's initialization parameters as an Enumeration of String objects.

Retrieve HTTP request header information

A servlet can access the headers of an HTTP request through the following methods of the HttpServletRequest interface:

- **getHeader**
- **getHeaders**
- **getHeaderNames**

The ***getHeader*** method returns a header given the name of the header. There can be multiple headers with the same name, e.g. Cache-Control headers, in an HTTP request.

If there are multiple headers with the same name, the **getHeader** method returns the first head in the request. The **getHeaders** method allows access to all the header values associated with a particular header name, returning an Enumeration of String objects.

Headers may contain String representations of int or Date data. The following convenience methods of the *HttpServletRequest* interface provide access to header data in a one of these formats:

- **getIntHeader**
- **getDateHeader**

If the **getIntHeader** method cannot translate the header value to an int, a NumberFormatException is thrown. If the **getDateHeader** method cannot translate the header to a Date object, an IllegalArgumentException is thrown.

Set an HTTP response header

A servlet can set headers of an HTTP response via the following methods of the *HttpServletResponse* interface:

- **setHeader**
- **addHeader**

The **setHeader** method sets a header with a given name and value. A previous header is replaced by the new header. Where a set of header values exist for the name, the values are cleared and replaced with the new value.

The **addHeader** method adds a header value to the set with a given name. If there are no headers already associated with the name, a new set is created.

Headers may contain data that represents an int or a Date object. The following convenience methods of the *HttpServletResponse* interface allow a servlet to set a header using the correct formatting for the appropriate data type:

- **setIntHeader**
- **setDateHeader**

- **addIntHeader**
- **addDateHeader**

To be successfully transmitted back to the client, headers must be set before the response is committed. Headers set after the response is committed will be ignored by the servlet container.

Set the content type of the response

The *ServletResponse* interface has the method **setContentType(String).**

public void **setContentType**(java.lang.String type)

Sets the content type of the response being sent to the client. The content type may include the type of character encoding used,
for example: text/html; charset=ISO-8859-4.

Acquire a text stream for the response

The *ServletResponse* interface has the method **getWriter().** To send character data, use the PrintWriter object returned by **getWriter()**.

public java.io.PrintWriter **getWriter**() throws java.io.IOException

Returns a PrintWriter object that can send character text to the client.

Calling **flush()** on the PrintWriter commits the response.

Acquire a binary stream for the response

The *ServletResponse* interface has the method **getOutputStream().** To send binary data in a MIME body response, use the ServletOutputStream returned by getOutputStream( ) **.**

public ServletOutputStream **getOutputStream**() throws IOException

Returns a ServletOutputStream suitable for writing binary data in the response. The servlet container does not encode the binary data.

Calling **flush()** on the ServletOutputStream commits the response.

Redirect an HTTP request to another URL

The *HttpServletResponse* interface has the method **sendRedirect(String)** method that sends a temporary redirect response to the client using the specified redirect location URL.

public void **sendRedirect**(String location) throws IOException

Sends a temporary redirect response to the client using the specified redirect location URL. This method can accept relative URLs; the servlet container must convert the relative URL to an absolute URL before sending the response to the client. If the location is relative without a leading '/' the container interprets it as relative to the current request URI. If the location is relative with a leading '/' the container interprets it as relative to the servlet container root.

If the response has already been committed, this method throws an IllegalStateException. After using this method, the response should be considered to be committed and should not be written to.

1.4 Identify the interface and method to access values and resources and to set object attributes within the following three Web scopes:

Request

ServletRequest interface has the following methods:

public java.lang.Object **getAttribute**(java.lang.String name)

Returns the value of the named attribute as an Object, or null if no attribute of the given name exists.

public java.util.Enumeration **getAttributeNames**()

Returns an Enumeration containing the names of the attributes available to this request.

public ServletInputStream **getInputStream**() throws java.io.IOException

Retrieves the body of the request as binary data using a ServletInputStream.

public void **setAttribute**(java.lang.String name, java.lang.Object o)

Stores an attribute in this request. Attributes are reset between requests. This method is most often used in conjunction with *RequestDispatcher*.

public void **removeAttribute**(java.lang.String name)

Removes an attribute from this request. This method is not generally needed as attributes only persist as long as the request is being handled.

Session

The *HttpSession* interface has the following methods:

public java.lang.Object **getAttribute**(java.lang.String name)

Returns the object bound with the specified name in this session, or null if no object is bound under the name

public java.util.Enumeration **getAttributeNames**()

Returns an Enumeration of String objects containing the names of all the objects bound to this session.

public void **setAttribute**(java.lang.String name, java.lang.Object value)

Binds an object to this session, using the name specified.

public void **removeAttribute**(java.lang.String name)

Removes the object bound with the specified name from this session. If the session does not have an object bound with the specified name, this method does nothing.

Context

The *ServletContext* has the following methods:

A servlet can bind an object attribute into the context by name. Any attribute bound into a context is available to any other servlet that is part of the same web application. The following methods of *ServletContext* interface allow access to this functionality:

- **setAttribute**
- **getAttribute**
- **getAttributeNames**
- **removeAttribute**

The *ServletContext* interface provides direct access to the hierarchy of static content documents that are part of the web application, including HTML, GIF, and JPEG files, via the following methods of the *ServletContext* interface:

- **getResource**
- **getResourceAsStream**

The **getResource** and **getResourceAsStream** methods take a String with a leading "/" as argument which gives the path of the resource relative to the root of the context. This hierarchy of documents may exist in the server's file system, in a web application archive file, on a remote server, or at some other location. These methods are not used to obtain dynamic content.

For example, in a container supporting the JavaServer Pages specification 1, a method call of the form **getResource**("/index.jsp") would return the JSP source code and not the processed output.

1.5 Given a life-cycle method: init, service or destroy, identify correct statements about its purpose or about how and when it is invoked.
Servlet:

This interface defines methods to initialize a servlet, to service requests, and to remove a servlet from the server. These are known as life-cycle methods and are called in the following sequence:

The servlet is constructed, and then initialized with the **init** method. Any calls from clients to the **service** method are handled. The servlet is taken out of service, then destroyed with the **destroy** method, then garbage collected and finalized.

In addition to the life-cycle methods, this interface provides the **getServletConfig** method, which the servlet can use to get any startup information, and the **getServletInfo method**, which allows the servlet to return basic information about itself, such as author, version, and copyright.

1.6 Use a RequestDispatcher to include or forward to a web resource.

ServletContext

Servlets (and JSP pages also) may be given names via server administration or via a web application deployment descriptor. A servlet instance can determine its name using ServletConfig.**getServletName**().

Returns a RequestDispatcher object that acts as a wrapper for the resource located at the given path. A RequestDispatcher object can be used to forward a request to the resource or to include the resource in a response. The resource can be dynamic or static.

The pathname must begin with a "/" and is interpreted as relative to the current context root. Use getContext to obtain a RequestDispatcher for resources in foreign contexts.

This method returns null if the ServletContext cannot return a RequestDispatcher.

ServletRequest

The pathname specified may be relative, although it cannot extend outside the current servlet context. If the path begins with a "/" it is interpreted as relative to the current context root. This method returns null if the servlet container cannot return a RequestDispatcher.

The difference between this method and ServletContext.**getRequestDispatcher**(java.lang.String) is that this method can take a relative path.

Forward

Forwards a request from a servlet to another resource (servlet, JSP file, or HTML file) on the server. This method allows one servlet to do preliminary processing of a request and another resource to generate the response.

For a RequestDispatcher obtained via **getRequestDispatcher(),** the ServletRequest object has its path elements and parameters adjusted to match the path of the target resource.

forward should be called before the response has been committed to the client (before response body output has been flushed). If the response already has been committed, this method throws an IllegalStateException. Uncommitted output in the response buffer is automatically cleared before the forward.

Include

Includes the content of a resource (servlet, JSP page, HTML file) in the response. In essence, this method enables programmatic server-side includes.

The *ServletResponse* object has its path elements and parameters remain unchanged from the caller's. The included servlet cannot change the response status code or set headers; any attempt to make a change is ignored.

The Structure and Deployment of Modern Servlet Web Applications

2.1 Identify the structure of the Web Application and Web Archive file, the name of the WebApp deployment descriptor and the name of the directories where you place the following:

The WebApp deployment descriptor.
The WebApp class files.
Any auxiliary JAR files

A web application exists as a structured hierarchy of directories. The root of this hierarchy serves as a document root files that are part of the application. For example, for a web application with the context path /catalog in a web container, the index.html file the base of the web application hierarchy can be served to satisfy a request from /catalog/index.html.

Since the context path of an application determines the URL namespace of the contents of the web application, web containers must reject web applications defining a context path that is the same, determined case sensitively, as the context path of a web application already deployed that exposes a potentially conflicting URL namespace.

A special directory exists within the application hierarchy named “WEB-INF”. This directory contains all things related to the application that aren’t in the document root of the application. The WEB-INF node is not part of the public document tree of the application. No file contained in the WEB-INF directory may be served directly to a client by the container. However, the contents of the WEB-INF directory are visible to servlet code using the **getResource()** and **getResourceAsStream()** method calls on the ServletContext.

Any application specific configuration information that the developer needs access to from servlet code yet does not wish to be exposed to the web client may be placed under this directory.

Since requests are matched to resource mappings case-sensitively, client requests for ‘/WEB-INF/ foo’, ‘/WEb-iNf/foo’, for example, should not result in contents of the web application located under /WEB-INF being returned, nor any form of directory listing thereof.

The contents of the WEB-INF directory are:

The /WEB-INF/web.xml

Deployment descriptor

The /WEB-INF/classes/*

Directory for servlet and utility classes. The classes in this directory are available to the application class loader.

The /WEB-INF/lib/*.jar

Area for Java ARchive files. These files contain servlets, beans, and other utility classes useful to the web application. The web application class loader can load class from any of these archive files. The web application classloader loads classes from the WEB-INF/classes directory first, and then from library JARs in the WEB-INF/lib directory. Library JARs are loaded from in the same order as they appear in the WAR archive entries.

2.2 Match the name with a description of purpose or functionality, for each of the following deployment descriptor elements:

Servlet instance

The servlet element contains the declarative data of a servlet. If a jsp-file is specified and the load-on-startup element is present, then the JSP should be precompiled and loaded.

<!ELEMENT **servlet** (icon?, **servlet-name**, display-name?, description?, (**servlet-class**|jsp-file), **init-param***, load-on-startup?, run-as?, security-role-ref*) >

Servlet name

The servlet-name element contains the canonical name of the servlet. Each servlet name is unique within the web application.

```
<!ELEMENT servlet-name (#PCDATA)>
```

Servlet class

The servlet-class element contains the fully qualified class name of the servlet.

```
<!ELEMENT servlet-class (#PCDATA)>
```

Initialization parameters

The init-param element contains a name/value pair as an initialization param of the servlet

```
<!ELEMENT init-param (param-name, param-value, description?)>
```

URL to named servlet mapping

The servlet-mapping element defines a mapping between a servlet and a url pattern

```
<!ELEMENT servlet-mapping (servlet-name, url-pattern)>
```

The servlet-name element contains the canonical name of the servlet. Each servlet name is unique within the web application.

```
<!ELEMENT servlet-name (#PCDATA)>
```

The url-pattern element contains the url pattern of the mapping. Must follow the rules specified in Section 11.2 of the Servlet API Specification.

```
<!ELEMENT url-pattern (#PCDATA)>
```

Example

```
<servlet>
        <servlet-name>catalog</servlet-name>
        <servlet-class>com.mycorp.CatalogServlet</servlet-class>
        <init-param>
                <param-name>catalog</param-name>
                <param-value>Spring</param-value>
        </init-param>
</servlet>

<servlet-mapping>
```

```
        <servlet-name>catalog</servlet-name>
        <url-pattern>/catalog/*</url-pattern>
</servlet-mapping>
```

The Servlet Container Model

3.1 Identify the uses for and the interfaces (or classes) and methods to achieve the following features:

Servlet context init parameters

ServletContext

The following methods of the ServletContext interface allow the servlet access to context initialization parameters associated with a web application through its deployment descriptor:

- **getInitParameter**
- **getInitParameterNames**

Initialization parameters are used by an application developer to convey setup information. Typical examples are a webmaster's e-mail address, or the name of a system that holds critical data.

Servlet context listener

ServletContextListener

Implementations of this interface receive notifications about changes to the servlet context of the web application they are part of. To receive notification events, the implementation class must be configured in the deployment descriptor for the web application.

public void **contextInitialized**(ServletContextEvent sce)

Notification that the web application is ready to process requests.

public void **contextDestroyed**(ServletContextEvent sce)

Notification that the servlet context is about to be shut down.

Servlet context attribute listener

ServletContextAttributeListener

Implementations of this interface receive notifications of changes to the attribute list on the servlet context of a web application. To receive notification events, the implementation class must be configured in the deployment descriptor for the web application.

public void **attributeAdded**(ServletContextAttributeEvent scab)

Notification that a new attribute was added to the servlet context. Called after the attribute is added.

public void **attributeRemoved**(ServletContextAttributeEvent scab)

Notification that an existing attribute has been removed from the servlet context. Called after the attribute is removed.

public void **attributeReplaced**(ServletContextAttributeEvent scab)

Notification that an attribute on the servlet context has been replaced. Called after the attribute is replaced.

Session attribute listeners

HttpSessionAttributeListener

This listener interface can be implemented in order to get notifications of changes to the attribute lists of sessions within this web application.

public void **attributeAdded**(HttpSessionBindingEvent se)

Notification that an attribute has been added to a session. Called after the attribute is added.

public void **attributeRemoved**(HttpSessionBindingEvent se)

Notification that an attribute has been removed from a session. Called after the attribute is removed.

public void **attributeReplaced**(HttpSessionBindingEvent se)

Notification that an attribute has been replaced in a session. Called after the attribute is replaced.

Other Listeners

HttpSessionListener

Implementations of this interface may are notified of changes to the list of active sessions in a web application. To receive notification events, the implementation class must be configured in the deployment descriptor for the web application.

public void **sessionCreated**(HttpSessionEvent se)

Notification that a session was created.

public void **sessionDestroyed**(HttpSessionEvent se)

Notification that a session was invalidated.

HttpSessionActivationListener

Objects that are bound to a session may listen to container events notifying them that sessions will be passivated and that session will be activated. A container that migrates session between VMs or persists sessions is required to notify all attributes bound to sessions implementing HttpSessionActivationListener.

HttpSessionBindingListener

Causes an object to be notified when it is bound to or unbound from a session. The object is notified by an HttpSessionBindingEvent object. This may be as a result of a servlet programmer explicitly unbinding an attribute from a session, due to a session being invalidated, or die to a session timing out.

3.2 Identify the WebApp deployment descriptor element name that declares the following features:

Servlet context init parameters

The context-param element contains the declaration of a web application’s servlet context initialization parameters.

<!ELEMENT context-param (param-name, param-value, description?)>

The param-name element contains the name of a parameter. Each parameter name must be unique in the web application.

<!ELEMENT param-name (#PCDATA)>
The param-value element contains the value of a parameter.

<!ELEMENT param-value (#PCDATA)>

Example

```
<web-app>
        <context-param>
                <param-name>Webmaster</param-name>
                <param-value>webmaster@mycorp.com</param-value>
        </context-param>
</web-app>
```

Servlet context listener & Session attribute listeners & Servlet context attribute listeners

The listener element indicates the deployment properties for a web application listener bean.

<!ELEMENT listener (listener-class)>

The listener-class element declares a class in the application must be registered as a web application listener bean. The value is the fully qualified class name of the listener class.

<!ELEMENT listener-class (#PCDATA)>

Example

The following example is the deployment grammar for registering two servlet context lifecycle listeners and an HttpSession listener.

Suppose that com.acme.MyConnectionManager and com.acme.MyLoggingModule both implement javax.servlet.ServletContextListener, and that com.acme.MyLoggingModule additionally implements javax.servlet.HttpSessionListener. Also the developer wants com.acme.MyConnectionManager to be notified of servlet context lifecycle events before com.acme.MyLoggingModule. Here is the deployment descriptor for this application:

```
<web-app>
        <display-name>MyListeningApplication</display-name>

        <listener>
                <listener-class>com.acme.MyConnectionManager</listener-class>
        </listener>

        <listener>
                <listener-class>com.acme.MyLoggingModule</listener-class>
        </listener>
        <servlet>
                <display-name>RegistrationServlet</display-name>
        </servlet>
</web-app>
```

3.3 Distinguish the behavior of the following in a distributable:

- ***Servlet context init parameters***
- ***Servlet context listener***
- ***Servlet context attribute listeners***
- ***Session attribute listeners***

See 3.2

Designing and Developing Servlets to Handle Server-side Expectations

4.1 For each of the following cases, identify correctly constructed code for handling business logic exceptions and match that code with the correct statements about the code's behavior: Return an HTTP error using the sendError response method; Return and HTTP error using the setStatus method.

Error Pages

To allow developers to customize the appearance of content returned to a web client when a servlet generates an error, the deployment descriptor defines a list of error page descriptions. The syntax allows the configuration of resources to be returned by the container either when a servlet sets a status code to indicate an error on the response, or if the servlet generates a Java exception or error that propagates to the container.

If a status code indicating an error is set on the response, the container consults the list of error page declarations for the web application that use the status-code syntax and attempts a match. If there is a match, the container serves back the resource as indicated by the location entry.

A Servlet may throw the following exceptions during processing of a request:

- runtime exceptions or errors
- ServletExceptions or subclasses thereof
- IOExceptions or subclasses thereof

The web application may have declared error pages using the exception-type syntax. In this case the container matches the exception type by comparing the exception thrown with the list of error-page definitions that use the exception-type syntax. A match results in the container serving back the resource indicated in the location entry. The closest match in the class hierarchy wins.

If no error-page declaration containing an exception-type fits using the class-hierarchy match, and the exception thrown is a ServletException or subclass thereof, the container extracts the wrapped exception, as defined by the ServletException.**getRootCause**() method. It makes a second pass over the error page declarations, again attempting the match against the error page declarations, but using the wrapped exception instead. Error-page declarations using the exception-type syntax in the deployment descriptor must be unique up to the classname of the exception-type. Similarly, error-page declarations using the status-code syntax must be unique in the deployment descriptor up to the status code.

The error page mechanism described does not intervene when errors occur in servlets invoked using the RequestDispatcher. In this way, a servlet using the RequestDispatcher to call another servlets has the opportunity to handle errors generated in the servlet it calls.

If a Servlet generates an error that is unhandled by the error page mechanism as described above, the container must ensure the status code of the response is set to status code 500.

The following convenience methods exist in the *HttpServletResponse* interface:

- **sendRedirect**
- **sendError**

The **sendRedirect** method will set the appropriate headers and content body to redirect the client to a different URL. It is legal to call this method with a relative URL path, however the underlying container must translate the relative path to a fully qualified URL for transmission back to the client.
If a partial URL is given and, for whatever reason, cannot be converted into a valid URL, then this method must throw an IllegalArgumentException.

The **sendError** method will set the appropriate headers and content body for an error message to return to the client. An optional String argument can be provided to the **sendError** method which can be used in the content body of the error.

These methods will have the side effect of committing the response, if it has not already been committed, and terminating it. No further output to the client should be made by the servlet after these methods are called. If data is written to the response after these methods are called, the data is ignored.

If data has been written to the response buffer, but not returned to the client (i.e. the response is not committed), the data in the response buffer must be cleared and replaced with the data set by these methods. If the response is committed, these methods must throw an IllegalStateException.

HttpServletResponse

public void **sendError**(int sc) throws java.io.IOException

Sends an error response to the client using the specified status code and clearing the buffer.

If the response has already been committed, this method throws an IllegalStateException. After using this method, the response should be considered to be committed and should not be written to.

public void **sendError**(int sc, String msg) throws IOException

Sends an error response to the client using the specified status clearing the buffer. The server defaults to creating the response to look like an HTML-formatted server error page containing the specified message, setting the content type to "text/html", leaving cookies and other headers unmodified. If an error-page declaration has been made for the web application corresponding to the status code passed in, it will be served back in preference to the suggested msg parameter.

If the response has already been committed, this method throws an IllegalStateException. After using this method, the response should be considered to be committed and should not be written to.

public void **setStatus**(int sc)

Sets the status code for this response. This method is used to set the return status code when there is no error (for example, for the status codes SC_OK or SC_MOVED_TEMPORARILY). If there is an error, and the caller wishes to invoke an defined in the web application, the sendError method should be used instead.

4.2 Given a set of business logic exceptions, identify the following: The configuration that the deployment descriptor uses to handle each exception; How to use the RequestDispatcher to forward the request to an error page; specify the handling declaratively in the deployment descriptor.

The error-page element contains a mapping between an error code or exception type to the path of a resource in the web application

<!ELEMENT error-page ((error-code | exception-type), location)>

The error-code contains an HTTP error code, ex: 404

<!ELEMENT error-code (#PCDATA)>

The exception type contains a fully qualified class name of a Java exception type.

```
<!ELEMENT exception-type (#PCDATA)>
```

The location element contains the location of the resource in the web application relative to the root of the web application. The value of the location must have a leading '/'.

```
<!ELEMENT location (#PCDATA)>
```

Example 1

```
<web-app>
        <error-page>
                    <error-code>404</error-code>
                    <location>/404.html</location>
        </error-page>
</web-app>
```

Example 2

```
<web-app>
    <error-page>
        <exception-type>java.lang.NumberFormatException</exception-type>
        <location>/MyException.html</location>
   </error-page>
</web-app>
```

4.3 Identify the method used for the following: Write a message to the WebApp log; Write a message and an exception to the WebApp log.

HttpServlet (inherited from GenericServlet*)*

public void **log**(java.lang.String msg)

> Writes the specified message to a servlet log file, prepended by the servlet's name.

public void **log**(java.lang.String message, java.lang.Throwable t)

Writes an explanatory message and a stack trace for a given Throwable exception to the servlet log file, prepended by the servlet's name

ServletContext

public void **log**(java.lang.String msg)

Writes the specified message to a servlet log file, usually an event log.

The name and type of the servlet log file is specific to the servlet container.

public void **log**(java.lang.String message, java.lang.Throwable throwable)

Writes an explanatory message and a stack trace for a given Throwable exception to the servlet log file.

The name and type of the servlet log file is specific to the servlet container, usually an event log.

Designing and Developing Servlets Using Session Management

5.1 Identify the interface and method for each of the following:

Retrieve a session object across multiple requests to the same or different servlets within the same WebApp.

HttpSession

Provides a way to identify a user across more than one page request or visit to a Web site and to store information about that user.

The servlet container uses this interface to create a session between an HTTP client and an HTTP server. The session persists for a specified time period, across more than one connection or page request from the user. A session usually corresponds to one user, who may visit a site many times. The server can maintain a session in many ways such as using cookies or rewriting URLs.

Store objects into a session object.

public void **setAttribute**(String name, Object value)

Binds an object to this session, using the name specified. If an object of the same name is already bound to the session, the object is replaced.

Retrieve objects from a session object.

public Object **getAttribute**(String name)

Returns the object bound with the specified name in this session, or null if no object is bound under the name.

Respond to the event when a particular object is added to a session.

HttpSessionBindingListener

Causes an object to be notified when it is bound to or unbound from a session. The object is notified by an HttpSessionBindingEvent object. This may be as a result of a servlet programmer explicitly unbinding an attribute from a session, due to a session being invalidated, or die to a session timing out.

After **setAttribute(String)** method executes and if the new object implements HttpSessionBindingListener, the container calls HttpSessionBindingListener.**valueBound**. The container then notifies any HttpSessionAttributeListeners in the web application.

If an object was already bound to this session of this name that implements HttpSessionBindingListener, its HttpSessionBindingListener.valueUnbound method is called.

If the value passed in is null, this has the same effect as calling **removeAttribute()**.

Respond to the event when a session is created and destroyed.

HttpSessionListener

Implementations of this interface may are notified of changes to the list of active sessions in a web application. To receive notification events, the implementation class must be configured in the deployment descriptor for the web application.

public void **sessionCreated**(HttpSessionEvent se)

Notification that a session was created.

public void **sessionDestroyed**(HttpSessionEvent se)

Notification that a session was invalidated.

Expunge a session object.

Invalidate

public void **invalidate**()

Invalidates this session then unbinds any objects bound to it.

5.2 Given a scenario, state whether a session object will be invalidated.

In the HTTP protocol, there is no explicit termination signal when a client is no longer active. This means that the only mechanism that can be used to indicate when a client is no longer active is a timeout period. The default timeout period for sessions is defined by the servlet container and can be obtained via the **getMaxInactiveInterval** method of the *HttpSession* interface.

This timeout can be changed by the Developer using the **setMaxInactiveInterval** method of the *HttpSession* interface. The timeout periods used by these methods are defined in seconds. By definition, if the timeout period for a session is set to -1, the session will never expire.

The session-config element defines the session parameters for this web application.

<!ELEMENT session-config (session-timeout?)>

The session-timeout element defines the default session timeout interval for all sessions created in this web application. The specified timeout must be expressed in a whole number of minutes. If the timeout is 0 or less, the container ensures the default behavior of sessions is never to time out.

<!ELEMENT session-timeout (#PCDATA)>

Example

```
<web-app>
                ….
                <session-config>
                        <session-timeout>30</session-timeout>
                </session-config>
                …..
</web-app>
```

5.3 Given that URL-rewriting must be used for session management, identify the design requirement on session-related HTML pages.

Session tracking through HTTP cookies is the most used session tracking mechanism and is required to be supported by all servlet containers.

The container sends a cookie to the client. The client will then return the cookie on each subsequent request to the server, unambiguously associating the request with a session. The name of the session tracking cookie must be **JSESSIONID**.

URL rewriting is the lowest common denominator of session tracking. When a client will not accept a cookie, URL rewriting may be used by the server as the basis for session tracking.

URL rewriting involves adding data, a session id, to the URL path that is interpreted by the container to associate the request with a session. The session id must be encoded as a path parameter in the URL string. The name of the parameter must be jsessionid.

Here is an example of a URL containing encoded path information:

http://www.myserver.com/catalog/index.html;jsessioid=1234

Designing and Developing Secure Web Applications

6.1 Identify correct descriptions or statements about the security issues:

- ***authentication***
- ***authorization***
- ***data integrity***
- ***auditing***
- ***malicious code***
- ***Web site attacks***

A web application contains resources that can be accessed by many users. These resources often traverse unprotected, open networks such as the Internet. In such an environment, a substantial number of web applications will have security requirements.

Although the quality assurances and implementation details may vary, servlet containers have mechanisms and infrastructure for meeting these requirements that share some of the following characteristics:

Authentication: The means by which communicating entities prove to one another that they are acting on behalf of specific identities that are authorized for access.

Access control for resources: The means by which interactions with resources are limited to collections of users or programs for the purpose of enforcing integrity, confidentiality, or availability constraints.

Data Integrity: The means used to prove that information has not been modified by a third party while in transit.

Confidentiality or Data Privacy: The means used to ensure that information is made available only to users who are authorized to access it.

6.2 Identify the deployment descriptor element names, and their structure, that declare the following:

- ***a security constraint***
- ***a Web resource***
- ***the login configuration***
- ***a security role***

Security constraints are a declarative way of annotating the intended protection of web content.

A constraint consists of the following elements:

- **web resource collection**
- **authorization constraint**
- **user data constraint**

A ***web resource collection*** is a set of URL patterns and HTTP methods that describe a set of resources to be protected. All requests that contain a request path that matches a URL pattern described in the web resource collection is subject to the constraint. The container matches URL patterns defined in security constraints using the same algorithm described in this specification for matching client requests to servlets and static resources.

An ***authorization constraint*** is a set of security roles to which users must belong for access to resources described by the web resource collection. If the user is not part of an allowed role, the user must be denied access to the resource requiring it. If the authorization constraint defines no roles, no user is allowed access to the portion of the web application defined by the security constraint.

A ***user data constraint*** describes requirements for the transport layer of the client server: The requirement may be for content integrity (preventing data tampering in the communication process) or for confidentiality (preventing reading while in transit). The container must at least use SSL to respond to requests to resources marked integral or confidential. If the original request was over HTTP, the container must redirect the client to the HTTPS port.

Where there are multiple security constraints specified, the container matches authentication methods and authorizations to security constraints on a ‘first match wins’ basis.

A ***security role*** is a logical grouping of users defined by the Application Developer or Assembler. When the application is deployed, roles are mapped by a Deployer to principals or groups in the runtime environment.

A servlet container enforces declarative or programmatic security for the principal associated with an incoming request based on the security attributes of the principal. This may happen in either of the following ways:

A deployer has mapped a security role to a user group in the operational environment.

The user group to which the calling principal belongs is retrieved from its security attributes. The principal is in the security role only if the principal’s user group matches the user group to which the security role has been mapped by the deployer.

A deployer has mapped a security role to a principal name in a security policy domain. In this case, the principal name of the calling principal is retrieved from its security attributes. The principal is in the security role only if the principal name is the same as a principal name to which the security role was mapped.

The security-constraint element is used to associate security constraints with one or more web resource collections.

```
 <!ELEMENT security-constraint (display-name?, web-resource-collection+,
                                 auth-constraint?, user-data-constraint?)>
```

The web-resource-collection element is used to identify a subset of the resources and HTTP methods on those resources within a web application to which a security constraint applies. If no HTTP methods are specified, then the security constraint applies to all HTTP methods.

```
<!ELEMENT web-resource-collection (web-resource-name, description?,
                                    url-pattern*, http-method*)>
```

The web-resource-name contains the name of this web resource collection

```
<!ELEMENT web-resource-name (#PCDATA)>
```

The http-method contains an HTTP method (GET | POST |...)

```
<!ELEMENT http-method (#PCDATA)>
```

The user-data-constraint element is used to indicate how data communicated between the client and container should be protected

<!ELEMENT user-data-constraint (description?, transport-guarantee)>

The transport-guarantee element specifies that the communication between client and server should be ***NONE***, ***INTEGRAL***, or ***CONFIDENTIAL***.

NONE means that the application does not require any transport guarantees.

A value of INTEGRAL means that the application requires that the data sent between the client and server be sent in such a way that it can't be changed in transit.

CONFIDENTIAL means that the application requires that the data be transmitted in a fashion that prevents other entities from observing the contents of the transmission.

In most cases, the presence of the INTEGRAL or CONFIDENTIAL flag will indicate that the use of SSL is required.

<!ELEMENT transport-guarantee (#PCDATA)>

The auth-constraint element indicates the user roles that should be permitted access to this resource collection. The role-name used here must either correspond to the role-name of one of the security-role elements defined for this web application, or be the specially reserved role-name "*" that is a compact syntax for indicating all roles in the web application. If both "*" and rolenames appear, the container interprets this as all roles.

If no roles are defined, no user is allowed access to the portion of the web application described by the containing security-constraint.

The container matches role names case sensitively when determining access.

<!ELEMENT auth-constraint (description?, role-name*)>

The role-name element contains the name of a security role.

<!ELEMENT role-name (#PCDATA)>

The login-config element is used to configure the authentication method that should be used, the realm name that should be used for this application, and the attributes that are needed by the form login mechanism.

<!ELEMENT login-config (auth-method?, realm-name?, form-login-config?)>

The realm name element specifies the realm name to use in HTTP Basic authorization

<!ELEMENT realm-name (#PCDATA)>

The security-role element contains the declaration of a security role which is used in the security-constraints placed on the web application.

```
<!ELEMENT security-role (description?, role-name)>
```

The security-role-ref element defines a mapping between the name of role called from a Servlet using ***isUserInRole(String name)*** and the name of a security role defined for the web application.

For example, to map the security role reference "FOO" to the security role with role-name "manager" the syntax would be:

```
<security-role-ref>
        <role-name>FOO</role-name>
        <role-link>manager</manager>
</security-role-ref>
```

In this case if the servlet called by a user belonging to the "manager" security role made the API call ***isUserInRole("FOO")*** the result would be true.

Since the role-name "*" has a special meaning for authorization constraints, its value is not permitted here.

```
<!ELEMENT security-role-ref (description?, role-name, role-link)>
```

The role-link element is used to link a security role reference to a defined security role. The role-link element must contain the name of one of the security roles defined in the security-role elements.

```
<!ELEMENT role-link (#PCDATA)>
```

Example

```
<web-app>
        <display-name>A Secure Application</display-name>
        <security-role>
                <role-name>manager</role-name>
        </security-role>
        <servlet>
                <servlet-name>catalog</servlet-name>
                <servlet-class>com.mycorp.CatalogServlet</servlet-class>
                <init-param>
                        <param-name>catalog</param-name>
                        <param-value>Spring</param-value>
                </init-param>
                <security-role-ref>
                        <role-name>MGR</role-name>
                        <!-- role name used in code -->
                        <role-link>manager</role-link>
                </security-role-ref>
        </servlet>
```

```
<servlet-mapping>
        <servlet-name>catalog</servlet-name>
        <url-pattern>/catalog/*</url-pattern>
</servlet-mapping>

<security-constraint>
        <web-resource-collection>
                <web-resource-name>SalesInfo</web-resource-name>
                <url-pattern>/salesinfo/*</url-pattern>
                <http-method>GET</http-method>
                <http-method>POST</http-method>
        </web-resource-collection>
        <auth-constraint>
                <role-name>manager</role-name>
        </auth-constraint>
        <user-data-constraint>
            <transport-guarantee>CONFIDENTIAL</transport-guarantee>
        </user-data-constraint>
</security-constraint>
</web-app>
```

6.3 Given an authentication type: BASIC, DIGEST, FORM and CLIENT-CERT, identify the correct definition of its mechanism.

A web client can authenticate a user to a web server using one of the following mechanisms:

- **HTTP Basic Authentication**
- **HTTP Digest Authentication**
- **HTTPS Client Authentication**
- **Form Based Authentication**

HTTP Basic Authentication

HTTP Basic Authentication, which is based on a username and password, is the authentication mechanism defined in the HTTP/1.0 specification. A web server requests a web client to authenticate the user.

As part of the request, the web server passes the *realm* (a string) in which the user is to be authenticated. The realm string of Basic Authentication does not have to reflect any particular security policy domain (confusingly also referred to as a realm). The web client obtains the username and the password from the user and transmits them to the web server. The web server then authenticates the user in the specified realm.

Basic Authentication is not a secure authentication protocol. User passwords are sent in simple base64 encoding, and the target server is not authenticated.

HTTP Digest Authentication

Like HTTP Basic Authentication, HTTP Digest Authentication authenticates a user based on a username and a password. However the authentication is performed by transmitting the password in an encrypted form which is much more secure than the simple base64 encoding used by Basic Authentication, e.g. HTTPS Client Authentication.

As Digest Authentication is not currently in wide-spread use, servlet containers are encouraged but not required to support it.

HTTPS Client Authentication

End user authentication using HTTPS (HTTP over SSL) is a strong authentication mechanism. This mechanism requires the user to possess a Public Key Certificate (PKC). Currently, PKCs are useful in e-commerce applications and also for a single-signon from within the browser. Servlet containers that are not J2EE compliant are not required to support the HTTPS protocol.

Form Based Authentication

The look and feel of the “login screen” cannot be varied using the web browser’s built-in authentication mechanisms. This specification introduces a required form based authentication mechanism which allows a Developer to control the look and feel of the login screens.

The web application deployment descriptor contains entries for a login form and error page. The login form must contain fields for entering a username and a password. These fields must be named ’j_username’ and ’j_password’, respectively.

When a user attempts to access a protected web resource, the container checks the user’s authentication. If the user is authenticated and possesses authority to access the resource, the requested web resource is activated and a reference to it is returned. If the user is not authenticated, all of the following steps occur:

The login form associated with the security constraint is sent to the client and the URL path triggering the authentication is stored by the container.

The user is asked to fill out the form, including the username and password fields.

The client posts the form back to the server.

The container attempts to authenticate the user using the information from the form.

If authentication fails, the error page is returned using either a forward or a re-direct, and the status code of the response is set to 401.

If authentication succeeds, the authenticated user's principal is checked to see if it is in an authorized role for accessing the resource.

If the user is authorized, client is redirected to the resource using the stored URL path. The error page sent to a user that is not authenticated contains information about the failure.

Form Based Authentication has the same lack of security as Basic Authentication since the user password is transmitted as plain text and the target server is not authenticated.

Form based login and URL based session tracking can be problematic to implement. Form based login should be used only when sessions are being maintained by cookies or by SSL session information.

In order for the authentication to proceed appropriately, the action of the login form must always be j_security_check. This restriction is made so that the login form will work no matter which resource it is for, and to avoid requiring the server to specify the action field of the outbound form.

Here is an example showing how the form should be coded into the HTML page:

```
<form method="POST" action="j_security_check">
        <input type="text" name="j_username">
        <input type="password" name="j_password">
</form>
```

If the form based login is invoked because of an HTTP request, the original request parameters must be preserved by the container for use if, on successful authentication, it redirects the call to the requested resource.

The form-login-config element specifies the login and error pages that should be used in form based login. If form based authentication is not used, these elements are ignored.

```
<!ELEMENT form-login-config (form-login-page, form-error-page)>
```

The form-login-page element defines the location in the web app where the page that can be used for login can be found. The path begins with a leading / and is interpreted relative to the root of the WAR.

```
<!ELEMENT form-login-page (#PCDATA)>
```

The form-error-page element defines the location in the web app where the error page that is displayed when login is not successful can be found. The path begins with a leading / and is interpreted relative to the root of the WAR.

<!ELEMENT form-error-page (#PCDATA)>

The auth-method element is used to configure the authentication mechanism for the web application. As a prerequisite to gaining access to any web resources which are protected by an authorization constraint, a user must have authenticated using the configured mechanism.

Legal values for this element are "***BASIC***", "***DIGEST***", "***FORM***", or "***CLIENT-CERT***".

<!ELEMENT auth-method (#PCDATA)>

Designing and Developing Thread-safe Servlets

7.1 Identify which attribute scopes are thread-safe:

- ***local variables***
- ***instance variables***
- ***class variables***
- ***request attributes***
- ***session attributes***
- ***context attributes***

Number of Instances

For a servlet not implementing *SingleThreadModel* and not hosted in a distributed environment (the default), the servlet container must use only one instance per servlet declaration.

For a servlet implementing the *SingleThreadModel* interface, the servlet container may instantiate multiple instances to handle a heavy request load and serialize requests to a particular instance.

In the case where a servlet was deployed as part of an application marked in the deployment descriptor as *distributabl*e, a container may have only one instance per servlet declaration per VM. If the servlet in a distributable application implements the *SingleThreadModel* interface, the container may instantiate multiple instances of that servlet in each VM of the container.

Note about SingleThreadModel

The use of the *SingleThreadModel* interface guarantees that only one thread at a time will execute in a given servlet instance’s **service** method. It is important to note that this guarantee only applies to each servlet instance, since the container may choose to pool such objects.

Objects that are accessible to more than one servlet instance at a time, such as instances of HttpSession, may be available at any particular time to multiple servlets, including those that implement *SingleThreadModel*.

Multithreading Issues

A servlet container may send concurrent requests through the **service** method of the servlet. To handle the requests the developer of the servlet must make adequate provisions for concurrent processing with multiple threads in the service method.

An alternative for the developer is to implement the *SingleThreadModel* interface which requires the container to guarantee that there is only one request thread at a time in the service method. A servlet container may satisfy this requirement by serializing requests on a servlet, or by maintaining a pool of servlet instances.

If the servlet is part of a web application that has been marked as distributable, the container may maintain a pool of servlet instances in each VM that the application is distributed across.

For servlets not implementing the *SingleThreadModel* interface, if the **service** method (or methods such as **doGet** or **doPost** which are dispatched to the **service** method of the HttpServlet abstract class) has been defined with the synchronized keyword, the servlet container cannot use the instance pool approach, but must serialize requests through it.

It is strongly recommended that developers ***not synchronize*** the **service** method (or methods despatched to it) in these circumstances because of detrimental effects on performance.

Thread Safety

Implementations of the request and response objects are not guaranteed to be thread-safe. This means that they should only be used within the scope of the request handling thread.

References to the request and response objects must not be given to objects executing in other threads as the resulting behavior may be nondeterministic.

Session Semantics and Threading Issues

Multiple servlets executing request threads may have active access to a single session object at the same time. The Developer has the responsibility for synchronizing access to session resources as appropriate.

Listener Instances and Threading

The container is required to complete instantiation of the listener classes in a web application prior to the start of execution of the first request into the application. The container must reference each listener instance until the last request is serviced for the web application.

Attribute changes to *ServletContext* and *HttpSession* objects may occur concurrently.

The container is not required to synchronize the resulting notifications to attribute listener classes. Listener classes that maintain state are responsible for the integrity of the data and should handle this case explicitly.

7.2 Identify correct statements about differences between multi-threaded and single-threaded servlet models.

See 7.1

7.3 Identify the interface used to declare that a servlet must use the single thread model.

SingleThreadModel

Ensures that servlets handle only one request at a time. This interface has no methods.

If a servlet implements this interface, you are ***guaranteed*** that no two threads will execute concurrently in the servlet's **service** method. The servlet container can make this guarantee by synchronizing access to a single instance of the servlet, or by maintaining a pool of servlet instances and dispatching each new request to a free servlet.

This interface does not prevent synchronization problems that result from servlets accessing shared resources such as static class variables or classes outside the scope of the servlet.

JSPs

The JSP Model

8.1 Write the opening and closing tags for the following JSP tag types:

directive

Directive elements have a syntax of the form **<%@ directive...%>**
Directives have this syntax: <%@directive {attr=”value ” }* %>.

declaration

Declarations follow the syntax ***<%! ... %>***;

<%! declaration(s) %>

scriptlet

Scriptlets follow the syntax ***<% .... %>***;

<%scriptlet %>

expression

Expressions follow the syntax ***<%= ... %>.***

<%=expression %>

8.2 Given a type of JSP tag identify correct statements about its purpose or use.

Directives provide global information that is conceptually valid independent of any specific request received by the JSP page. They provide information for the translation phase. Directives are messages to the JSP container.

- The *page* directive defines a number of page dependent properties and communicates these to the JSP container.
- The *taglib* directive in a JSP page declares that the page uses a tag library, uniquely identifies the tag library using a URI and associates a tag prefix that will distinguish usage of the actions in the library.
- The *include* directive is used to substitute text and/or code at JSP page translation-time.

Scripting elements provide *glue* around template text and actions. There are three types of scripting elements: *declaration*s, *scriptlets* and *expressions.*

Declarations are used to declare variables and methods in the scripting language used in a JSP page. A declaration should be a complete declarative statement, or sequence thereof, according to the syntax of the scripting language specified. Declarations do not produce any output into the current out stream. Declarations are initialized when the JSP page is initialized and are made available to other declarations, scriptlets, and expressions.

Scriptlets can contain any code fragments that are valid for the scripting language specified in the language directive. Whether the code fragment is legal depends on the details of the scripting language. Scriptlets are executed at request-processing time. Whether or not they produce any output into the out stream depends on the code in the scriptlet.

An *expression* element in a JSP page is a scripting language expression that is evaluated and the result is coerced to a String. The result is subsequently emitted into the current out JspWriter object.

8.3 Given a JSP tag type identify the equivalent XML-based tags.

Directive: page

<%@page info=”my latest JSP Example” %>

or

<jsp:directive.page *info=”my latest JSP Example”* />

Directive: include

<%@include file=”copyright.html” %>

or

<jsp:directive.include file=”copyright.html” />

Scripting Element: Declaration

<%! int i =0; %>

or

```
<jsp:declaration>int i=0; </jsp:declaration>
```

Scripting Element: Scriptlet

```
<%

if (Calendar.getInstance().get(Calendar.AM_PM)==Calendar.AM) {
Good Morning
}else {
Good Afternoon
}

%>
```

or

```
<jsp:scriptlet>

if (Calendar.getInstance().get(Calendar.AM_PM)==Calendar.AM) {
Good Morning
}else {
Good Afternoon
}

</jsp:scriptlet>
```

Scritping Element: Expression

```
<%= (new java.util.Date()).toLocaleString() %>
```

or

```
<jsp:expression>(new java.util.Date()).toLocaleString() </jsp:expression>
```

8.4 Identify the page directive attributes and its values, that:

imports a Java class into the JSP page

```
<%@page  import=”java.util.Enumeration” %>
```

declares that a JSP page exists within a session

```
<%@page session=”true” %>
```

declares that a JSP page uses an error page

```
<%@page errorPage=”myErrorPage.jsp” %>
```

declares that a JSP page is an error page

```
<%@page isErrorPage=”true” %>
```

8.5 Identify and put in sequence the following elements of the JSP life-cycle:

- ***page translation***
- ***JSP compilation***
- ***load class***
- ***create instance***
- ***call jspInit***
- ***call _jspService***
- ***call _jspDestroy***

JSP Life Cycle Information

JSP pages are textual components. They go through two phases: a translation phase, and a request phase. Translation is done once per page. The request phase is done once per request.

The JSP page is translated to create a servlet class, the JSP page implementation class that is instantiated at request time. The instantiated JSP page object handles requests and creates responses.

JSP pages may be translated prior to their use, providing the web application, with a servlet class that can serve as the textual representation of the JSP page.

The translation may also be done by the JSP container at deployment time or on-demand as the requests reach an untranslated JSP page.

In the execution phase the container manages one or more instances of this class in response to requests and other events.

During the *translation phase* the container locates or creates the JSP page implementation class that corresponds to a given JSP page. This process is determined by the semantics of the JSP page. The container interprets the standard directives and actions, and the custom actions referencing tag libraries used in the page. A tag library may optionally provide a validation method to validate that a JSP page is correctly using the library.

A JSP container has flexibility in the details of the JSP page implementation class that can be used to address quality-of-service, most notably performance-issues.

During the *execution phase* the JSP container delivers events to the JSP page implementation object. The container is responsible for instantiating request and response objects and invoking the appropriate JSP page implementation object.

Upon completion of processing, the response object is received by the container for communication to the client.

The translation of a JSP source page into its implementation class can occur at any time between initial deployment of the JSP page into the JSP container and the receipt and processing of a client request for the target JSP page.

When the first request is delivered to a JSP page, a **jspInit()**method, if present, will be called to prepare the page. Similarly, a JSP container may invoke a JSP's **_jspDestroy()** method to reclaim the resources used by the JSP page at any time when a request is not being serviced. This is the same life-cycle as for *servlets.*

A JSP page may be *compiled* into its implementation class plus deployment information during development. (A JSP page can also be compiled at deployment time.)

Compilation of a JSP page in the context of a web application provides resolution of relative URL specifications in include directives (and elsewhere), taglib references, and translation-time actions used in custom actions.

A JSP page can also be compiled at deployment time.

Order of Events

The translation of a JSP source page into its implementation class can occur at any time between initial deployment of the JSP page into the JSP container and the receipt and processing of a client request for the target JSP page.

A JSP page can be compiled at development time or wait until deployment time to compile. During the compilation the JSP page is converted to the page implementation class (in other words a Servlet).

page translation ✍JSP compilation

Once the page is requested the container loads the class and creates and instance of that class.

load class✍create instance

The first time the JSP page is requested the **jspInit()** method is called, and the request is handled by the **_jspService()** method.

✍call jspInit✍call _jspService

When the container is done with the page, the **_jspDestroy()** method is called. This is where any cleanup (database connection) may be handled.

_jspDestroy

8.6 Match correct descriptions about purpose, function, or use with any of the following implicit objects:

request

The request object represents a sub class of: *javax.servlet.ServletRequest* e.g: javax.servlet.HttpServletRequest

The request object can be used like the HttpServletRequest parameter of the **service()** method of a Servlet.

response

The response object represents a sub class of: *javax.servlet.ServletResponse* e.g: servlet.HttpServletResponse

The response object can be used like the HttpServletResponse parameter of the **service()** method of a Servlet.

out

The out object represents a JspWriter that will write to the JSP's output stream.

session

The session object represents a user's session (*HttpSession)* created for the requesting client.

config

The config object represents the servlet config (*ServletConfig)* that represents the servlet's configuration.

application

The **application** object represents the servlet context (*ServletContext)* obtained from the servlet config.

page

The instance of this page's implementation class processing the current request.

pageContext

The page context for this JSP page.

exception

The uncaught Throwable that resulted in the error page being invoked.

8.7 Distinguish correct and incorrect scriptlet code for:

a conditional statement

```
<% if (Calendar.getInstance().get(Calendar.AM_PM)==Calendar.AM) {
 %>
      Good Morning
<%
} else {
%>
      Good Afternoon
<%
}
%>
```

an iteration statement

```
<% for ( int i=0; i < 10; i++) {
%>

<p>HelloWorld Counter <%= i %></p>

<%
}
%>
```

Designing and Developing Reusable Web Components

9.1 Given a description of required functionality, identify the JSP directive or standard tag in the correct format with the correct attributes required to specify the inclusion of a Web Component into the JSP page.

The <%@include file=”relativeURLspec ” %>directive inserts the text of the specified resource into the .jsp file. The included file is subject to the access control available to the JSP container.

Examples

The following example requests the inclusion, at translation time, of a copy-right file. The file may have elements which will be processed too.

```
<%@include file=”copyright.html ” %>
```

Syntax

<%@include file="*relativeURLspec* "%>

A <jsp:include .../>element provides for the inclusion of static and dynamic resources in the same context as the current page.

Examples

<jsp:include page="/templates/copyright.html "/>

Syntax

<jsp:include page="*urlSpec* " flush=="true|false"/>

and

```
<jsp:include page="urlSpec " flush=="true|false">
        {<jsp:param ..../>}*
</jsp:include>
```

An include directive regards a resource like a JSP page as a static object; i.e. the bytes in the JSP page are included. An include action regards a resource like a JSP page as a dynamic object; i.e. the request is sent to that object and the result of processing it is included.

Designing and Developing JSPs Using JavaBeans

10.1 For any of the following tag functions, match the correctly constructed tag, with attributes and values as appropriate, with the corresponding description of the tag's functionality.

Declare the use of a JavaBean within the page.

A jsp:useBean action associates an instance of a Java programming language object defined within a given scope and available with a given id with a newly declared scripting variable of the same id.

Examples

In the following example, a Bean with name "connection" of type myco.myapp.Connection " is available after actions on this element, either because it was already created and found, or because it is newly created.

<jsp:useBean id="connection " class="com.myco.myapp.Connection " />

Specify, for jsp:useBean or jsp:getProperty tags, the name of an attribute.
Specify, for a jsp:useBean tag, the class of the attribute.
Specify, for a jsp:useBean tag, the scope of the attribute.

<jsp:useBean id="*name* "scope="page|request|session|application" *typeSpec* />
typeSpec ::=class="className " ||
class="*className* " type="*typeName* " ||
type="*typeName* " class="*className* " ||
beanName="*beanName* " type="*typeName* " ||
type="*typeName* " beanName="*beanName* " ||
type="typeName "

If the action has a body, it is of the form:

<jsp:useBean id="*name* "scope="page|request|session|application" *typeSpec* >
    body
</jsp:useBean>

In this case, the body will be invoked if the Bean denoted by the action is created. Typically, the *body* will contain either scriptlets or jsp:setProperty tags that will be used to modify the newly created object, but the contents of the body are not restricted.

The <jsp:useBean>tag has the following attributes:

id

The name used to identify the object instance in the specified scope's namespace, *and* also the scripting variable name declared and initialized with that object reference.

The name specified is case sensitive and shall conform to the current scripting language variable-naming conventions.

scope

The scope within which the reference is available. The default value is page . The fully qualified name of the class that defines the implementation of the object. The class name is case sensitive.

If the class and beanName attributes are not specified the object must be present in the given scope.

beanName

The name of a Bean, as expected by the instantiate() method of the java.beans.Beans class.

This attribute can accept a request-time attribute expression as a value.

type

If specified, it defines the type of the scripting variable defined. This allows the type of the scripting variable to be distinct from, but related to, the type of the implementation class specified.
The type is required to be either the class itself, a superclass of the class, or an interface implemented by the class specified.

The object referenced is required to be of this type, otherwise a java.lang.ClassCastException shall occur at request time when the assignment of the object referenced to the scripting variable is attempted.

If unspecified, the value is the same as the value of the class attribute.

Access or mutate a property from a declared JavaBean.
Specify, for a jsp:getProperty tag, the property of the attribute.
Specify, for a jsp:setProperty tag, the property of the attribute to mutate and the new value.

Examples

The following two elements set a value from the request parameter values.

```
<jsp:setProperty name=”request ” property=”*” />

<jsp:setProperty name=”user ” property=”user ” param=”username ” />
```

The following element sets a property from a value

```
<jsp:setProperty name=”results ”property=”row ” value=”<%=i+1 %>”/>
```

Syntax

```
<jsp:setProperty name="beanName"prop_expr />
        prop_expr ::=property="*"|
        property=”propertyName ”|
        property=”propertyName ” param=="parameterName"|
        property=”propertyName ” value==”propertyValue ”
        propertyValue ::=string
```

The value propertyValue can also be a request-time attribute value.

Examples

<jsp:getProperty name=”user ” property=”name ” />

Syntax

<jsp:getProperty name=”name ” property=”propertyName ” />

10.2 Given JSP attribute scopes: request, session, application, identify the equivalent servlet code.

Bean Scope:	***Servlet Scope:***
Request	HttpServletRequest
Session	HttpSession
Application	ServletContext

10.3 Identify techniques that access a declared JavaBean.

A jsp:useBean action associates an instance of a Java programming language object defined within a given scope and available with a given id with a newly declared scripting variable of the same id.

The jsp:useBean action is quite flexible; its exact semantics depends on the attributes given. The basic semantic tries to find an existing object using id and scope. If the object is not found it will attempt to create the object using the other attributes.

It is also possible to use this action to give a local name to an object defined elsewhere, as in another JSP page or in a Servlet. This can be done by using the type attribute and not providing class or beanName attributes.

At least one of type and class must be present, and it is not valid to provide both class and beanName. If type and class are present; class must be assignable to type (in the Java platform sense). For it not to be assignable is a translation-time error.

Designing and Developing JSPs Using Custom Tags

Identify properly formatted taglib directive in a JSP page.

The web.xml file can include a map between URIs and TLD resource paths. The map is described using the taglib element of the Web Application Deployment descriptor in WEB-INF/web.xml, as described in the Servlet 2.3 spec and in “http://java.sun.com/j2ee/dtds/web-app_2_3.dtd”.

A taglib element has two sub-elements: taglib-uri and taglib-location.

<taglib>

A taglib is a sub-element of web-app:

The taglib element provides information on a tag library that is used by a JSP page within the Web Application.

A taglib element has two sub-elements and one attribute:

<taglib-uri>

A taglib-uri element describes a URI identifying a tag library used in the web application.

It may be either an absolute URI specification, or a relative URI.

<taglib-location>

A taglib-location contains the location (as a resource) of the Tag Library Description File for the tag library.

Where to find the Tag Library Descriptor file.

Example

The use of relative URI specifications enables very short names in the taglib directive. For example:

```
<%@taglib uri="/myPRlibrary " prefix="x " %>
```

```
<taglib>
        <taglib-uri>/myPRlibrary</taglib-uri>
        <taglib-location>/WEB-INF/tlds/PRlibrary_1_4.tld</taglib-uri>
</taglib>
```

11.2 Identify properly formatted taglib directive in a JSP page.

```
<%@taglib uri="/myPRlibrary" prefix="x" %>
```

11.3 Given a custom tag library identify properly formatted custom tag usage in a JSP page. Uses include:

Assuming prefix 'x' and custom tag 'sample'

an empty custom tag

```
<x:sample />
<x:sample></x:sample>
```

a custom tag with attributes

```
<x:sample rows="3"  name="someValue" />
```

a custom tag that surrounds other JSP code

```
<x:sample>some JSP code as the body</x:sample>
```

nested custom tags

```
<x:sample>
        <x:if>some condition</x:if>
</x:sample>
```

Designing and Developing a Custom Tag Library

12.1 Identify the tag library descriptor element names that declare the following:

the name of the tag
the class of the tag handler
the type of content that the tag accepts
any attributes of the tag

The element syntax is as follows:

<!ELEMENT **tag** (name, tag-class, tei-class?, body-content?, display-name?, small-icon?, large-icon?, description?, variable*,attribute*,example?)>

<tag-class>

Defines the tag handler class implementing the javax.serlvet.jsp.tagext.Tag interface. This element is required.

The fully qualified Java class name.

<tei-class>

Defines the subclass of javax.servlet.jsp.tagext.TagExtraInfo for this tag. This element is optional.

The fully qualified Java class name.

<body-content>

Provides a hint as to the content of the body of this action. Primarily intended for use by page composition tools.

There are currently three values specified:

tagdependent

The body of the action is passed verbatim to be interpreted by the tag handler itself, and is most likely in a different "language", e.g. embedded SQL statements. The body of the action may be empty.

JSP

The body of the action contains elements using the JSP syntax. The body of the action may be empty.

empty

The body must be empty. The default value is "JSP".

12.2 Identify the tag library descriptor element names that declare the following:

the name of a tag attribute.
whether a tag attribute is required.
whether or not the attribute's value can be dynamically specified.

<!ELEMENT **attribute** (name,required?,rtexprvalue?, type?, description?)>

<name>

Defines the canonical name of a tag or attribute being defined

<required>

Defines if the nesting attribute is required or optional.

Values are: true | false | yes | no
If not present then the default is "false", i.e. the attribute is optional.

<rtexprvalue>

Defines if the nesting attribute can have scriptlet expressions as a value, i.e the value of the attribute may be dynamically calculated at request time, as opposed to a static value determined at translation time.

Values are: true | false | yes | no
If not present then the default is "false", i.e the attribute has a static value.

<type>

Defines the Java type of the attribute's value. For static values (those determined at translation time) the type is always java.lang.String. If the rtexprvalue element is true, that is the value of the nesting attribute may be calculated from an expression scriptlet during request processing then the type defines the return type expected from any scriptlet expression specified as the value of this attribute.

The fully qualified Java class name of result type

An example is:
<type>java.lang.Object </type>

12.3 Given a custom tag, identify the necessary value for the bodycontent TLD element for any of the following tag types:

empty-tag
custom tag that surrounds other JSP code
custom tag that surrounds content that is used only by the tag handler

<body-content>

Provides a hint as to the content of the body of this action. Primarily intended for use by page composition tools.

There are currently three values specified:

tagdependent

The body of the action is passed verbatim to be interpreted by the tag handler itself, and is most likely in a different "language", e.g. embedded SQL statements. The body of the action may be empty.

JSP

The body of the action contains elements using the JSP syntax. The body of the action may be empty.

empty

The body must be empty. The default value is "JSP".

12.4 Given a tag event method (doStartTag, doAfterBody and doEndTag), identify the correct description of the methods trigger.

When a tag is first encountered the **doStartTag()** method is invoked.
When the custom tag implements *IterationTag* interface, when the body of the custom tag has completed, the **doAfterBody()** method is invoked.
When the custom tag reached the end tag the **doEndTag()** method is invoked.

12.5 Identify valid return values for the following methods:

doStartTag()

Empty and Non-Empty Action

If the action is an empty action, the **doStartTag()** method must return SKIP_BODY.

If the action is a non-empty action, the **doStartTag()** method may return SKIP_BODY or EVAL_BODY_INCLUDE.

If SKIP_BODY is returned the body is not evaluated.
If EVAL_BODY_INCLUDE is returned, the body is evaluated and "passed through" to the current out.

doAfterBody()

If **doAfterBody()** returns EVAL_BODY_AGAIN, a new evaluation of the body will happen (followed by another invocation of **doAfterBody()**).

If **doAfterBody()** returns SKIP_BODY no more body evaluations will occur, the value of out will be restored using the popBody method in pageContext, and then doEndTag will be invoked.

doEndTag()

If this method returns EVAL_PAGE, the rest of the page continues to be evaluated.

If this method returns SKIP_PAGE, the rest of the page is not evaluated and the request is completed. If this request was forwarded or included from another page (or Servlet), only the current page evaluation is completed.

pageContext.getOut()

Returns the current JspWriter stream being used for client response.

12.6 Given a "BODY" or "PAGE" constant, identify a correct description of the constant's use in the following methods:

doStartTag()
doAfterBody()
doEndTag()

SKIP_BODY

Skip body evaluation.
Valid return value for doStartTag and doAfterBody.

EVAL_BODY_INCLUDE

Evaluate body into existing out stream.
Valid return value for doStartTag.

SKIP_PAGE

Skip the rest of the page.
Valid return value for doEndTag.

EVAL_PAGE

Continue evaluating the page.
Valid return value for doEndTag().

EVAL_BODY_AGAIN

Request the reevaluation of some body.
Returned from doAfterBody. For compatibility with JSP 1.1, the value is carefully selected to be the same as the, now deprecated, BodyTag.EVAL_BODY_TAG,

EVAL_BODY_BUFFERED

Request the creation of new buffer, a BodyContent on which to evaluate the body of this tag.
Returned from doStartTag when it implements BodyTag.
This is an illegal return value for doStartTag when the class does not implement BodyTag.

12.7 Identify the method is the custom tag handler that accesses:

a given JSP implicit variable

A PageContext instance provides access to all the namespaces associated with a JSP page, provides access to several page attributes, as well as a layer above the implementation details.

The following methods provide **convenient access** to implicit objects: getOut(), getException(), getPage() getRequest(), getResponse(), getSession(), getServletConfig() and getServletContext().

The following methods provide support for **forwarding, inclusion and error handling**: forward(), include(), and handlePageException().

the JSP page's attributes.

PageContext.findAttribute()

public abstract java.lang.Object **findAttribute**(java.lang.String name)

Searches for the named attribute in page, request, session (if valid), and application scope(s) in order and returns the value associated or null.

PageContext.getAttributeNamesInScope()

public abstract java.util.Enumeration **getAttributeNamesInScope**(int scope)

Enumerate all the attributes in a given scope

12.8 Identify methods that return an outer tag from within an inner tag handler.

findAncestorWithClass

public static final Tag **findAncestorWithClass**(Tag from, java.lang.Class klass)

Find the instance of a given class type that is closest to a given instance. This method uses the getParent method from the Tag interface. This method is used for coordination among cooperating tags.

getParent

public Tag **getParent**()

Get the parent (closest enclosing tag handler) for this tag handler. This method is used by the findAncestorWithClass() method in TagSupport.

J2EE Design Patterns

13.1 Given a scenario description with a list of issues, select the design patters (Value Objects, MVC, Data Access Object or Business Delegate) that would best solve those issues.

13.2 Match design patterns with statements describing potential benefits that accrue from the use of the pattern, for any of the following patterns:
ValueObjects
MVC
Data Access Object
Business Delegate

See Below

Value Objects

Context

Application clients need to exchange data with enterprise beans.

Problem

J2EE applications implement server-side business components as session beans and entity beans. Session beans represent the business services and maintain a one-to-one relationship with the client. Entity beans on the other hand are multi-user, transactional objects representing persistent data. A session bean provides coarse-grained service methods when implemented per the Session Façade pattern. Some of the service methods may return data to the client that invoked the methods. In such cases, the client must invoke the session bean's get methods multiple times until the client obtains values for all the attribute values. Every such method call made to the session bean is potentially remote.

An entity bean implements persistent business components. An entity bean exposes the values of its attributes by providing an accessor method (also referred to as a getter or get method) for each attribute. When a client needs the data values from an entity bean, it may invoke the entity bean's get methods multiple times until the client obtains data for every attribute that it needs. Again, like the session bean, every such method call made to the entity bean is potentially remote.

Thus, in an EJB application each invocation on a session bean or an entity bean is potentially a remote method invocation that utilizes the network layer regardless of the proximity of the client to the bean. Such invocations on the enterprise beans create an overhead on the network. As the usage of these remote methods increases, application performance can significantly degrade. Therefore, multiple calls to get methods that return single attribute values is inefficient for obtaining data values from an enterprise bean.

Enterprise bean method calls may permeate the network layers of the system even if the client and the EJB container holding the entity bean are both running in the same JVM, OS, or physical machine. Some vendors may implement mechanisms to reduce this overhead by using a more direct access approach.

Forces

J2EE applications implement business components as enterprise bean components. All access to an enterprise bean is performed via remote interfaces to the bean. Every call to an enterprise bean is potentially a remote method call with network overhead.

Typically, applications have a greater frequency of read transactions than update transactions. The client requires the data from the business tier for presentation, display, and other read-only types of processing. The client updates the data in the business tier much less frequently than it reads the data.

The client usually requires more than one attribute value of an enterprise bean. The client may also need values for the dependent objects of the enterprise bean. Thus, the client may invoke multiple remote calls to obtain the required data.

The number of calls made by the client to the enterprise bean impacts network performance. Chattier applications—those with increased traffic between client and server tiers—often degrade network performance.

Solution

Use a Value Object to encapsulate the business data. A single method call is used to send and retrieve the Value Object. When the client requests the enterprise bean for the business data, the enterprise bean can construct the Value Object, populate it with its attribute values, and pass it by value to the client.

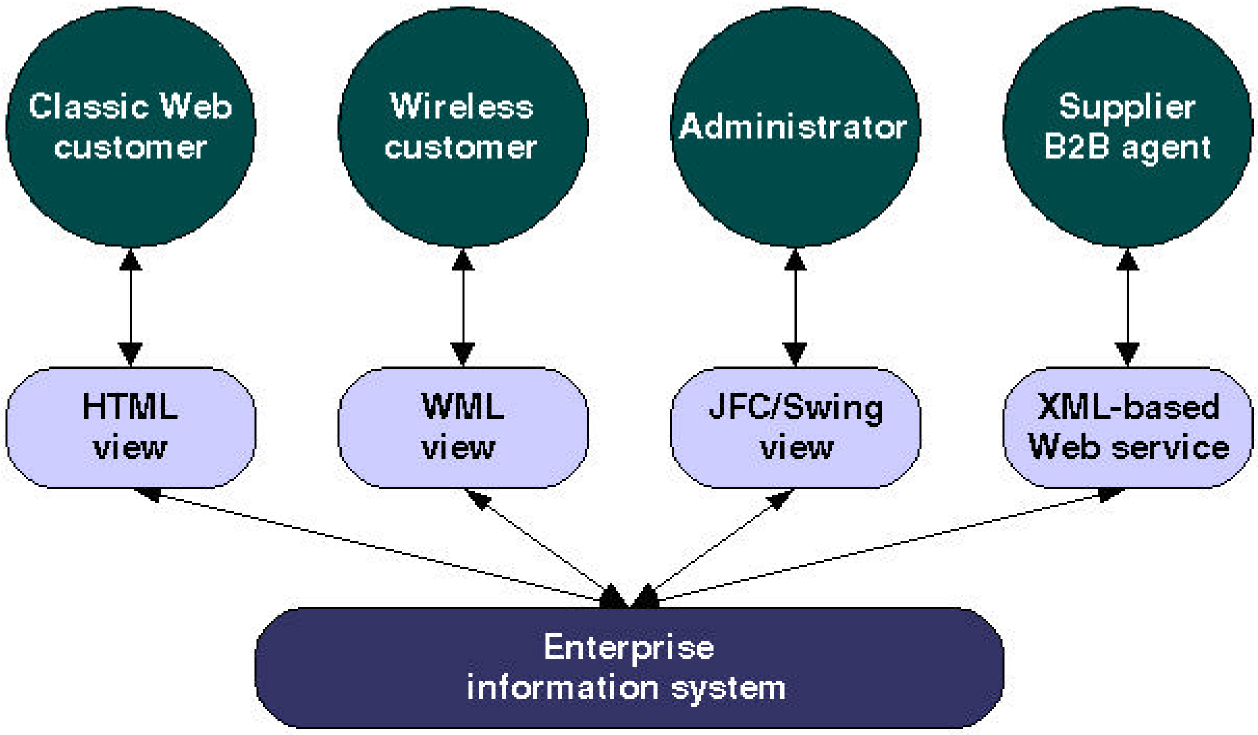
Clients usually require more than one value from an enterprise bean. To reduce the number of remote calls and to avoid the associated overhead, it is best to use value objects to transport the data from the enterprise bean to its client.

When an enterprise bean uses a value object, the client makes a single remote method invocation to the enterprise bean to request the value object instead of numerous remote method calls to get individual attribute values. The enterprise bean then constructs a new value object instance and copies the attribute values from its attributes into the newly created value object. It then returns the value object to the client. The client receives the value object and can then invoke its accessor (or getter) methods to get the individual attribute values from the value object. Or, the implementation of the Value Object may be such that makes all attributes public. Because the value object has been passed by value to the client, all calls to the value object instance are local calls instead of remote method invocations.

Model View and Control (MVC)

The Problem:
Supporting Multiple Enterprise Clients

Now, more than ever, enterprise applications need to support multiple types of users with multiple types of interfaces. For example, an online store may require an HTML front for Web customers, a WML front for wireless customers, a JFC/Swing interface for administrators, and an XML-based Web service for suppliers.



When developing an application to support a single type of client, it is sometimes beneficial to interweave data access logic with interface-specific logic for presentation and control. Such an approach, however, is inadequate when applied to enterprise systems that need to support multiple types of clients: Different applications need to be developed, one to support each type of client interface.

Non-interface-specific code is duplicated in each application, resulting in duplicate efforts in implementation (often of the copy-and-paste variety), as well as testing and maintenance.

The task of determining what to duplicate is expensive in itself, since interface-specific and non-interface-specific code are intertwined.

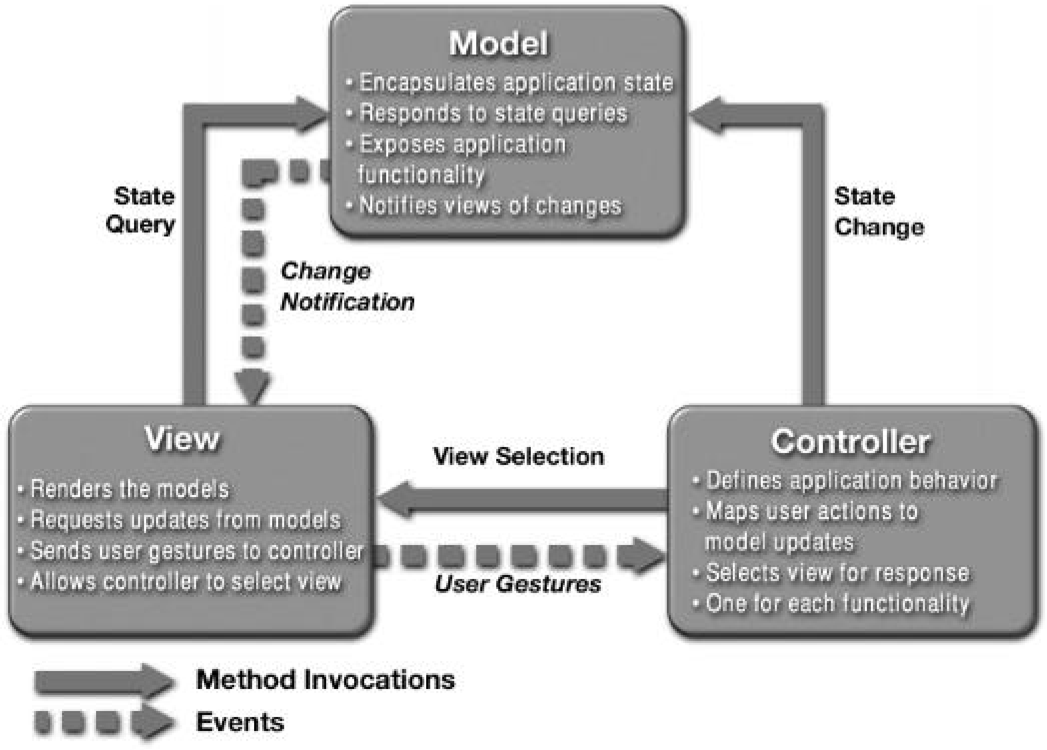
The duplication efforts are inevitably imperfect. Slowly, but surely, applications that are supposed to provide the same core functionality evolve into different systems.

How can an enterprise support multiple types of clients and at the same time avoid these costs? The following forces influence the solution:
The same enterprise data needs to be accessed when presented in different views: *e.g.* HTML, WML, JFC/Swing, XML.

The same enterprise data needs to be updated through different interactions: *e.g.* link selections on an HTML page or WML card, button clicks on a JFC/Swing GUI, SOAP messages written in XML.

Supporting multiple types of views and interactions should not impact the components that provide the core functionality of the enterprise application.
A Solution: Use the Model-View-Controller Architecture

By applying the Model-View-Controller (MVC) architecture to a J2EE application, you separate core data access functionality from the presentation and control logic that uses this functionality. Such separation allows multiple views to share the same enterprise data model, which makes supporting multiple clients easier to implement, test, and maintain.


Model
• Encapsulates application state
• Responds to state queries
• Exposes application functionality
• Notifies views of changes
State Query
Change Notification
State Change
View
• Renders the models
• Requests updates from models
• Sends user gestures to controller
• Allows controller to select view
View Selection
User Gestures
Controller
• Defines application behavior
• Maps user actions to model updates
• Selects view for response
• One for each functionality
Method Invocations
Events

The MVC architecture has its roots in Smalltalk, where it was originally applied to map the traditional input, processing, and output tasks to the graphical user interaction model. However, it is straightforward to map these concepts into the domain of multi-tier enterprise applications:

The **model** represents enterprise data and the business rules that govern access to and updates of this data. Often the model serves as a software approximation to a real-world process, so simple real-world modeling techniques apply when defining the model.

A **view** renders the contents of a model. It accesses enterprise data through the model and specifies how that data should be presented. It is the view's responsibility to maintain consistency in its presentation when the model changes. This can be achieved by using a *push* model, where the view registers itself with the model for change notifications, or a *pull* model, where the view is responsible for calling the model when it needs to retrieve the most current data.

A **controller** translates interactions with the view into actions to be performed by the model. In a stand-alone GUI client, user interactions could be button clicks or menu selections, whereas in a Web application, they appear as GET and POST HTTP requests. The actions performed by the model include activating business processes or changing the state of the model. Based on the user interactions and the outcome of the model actions, the controller responds by selecting an appropriate view.

The MVC architecture has the following **benefits**:

Multiple views using the same model. The separation of model and view allows multiple views to use the same enterprise model. Consequently, an enterprise application's model components are easier to implement, test, and maintain, since all access to the model goes through these components.

Easier support for new types of clients. To support a new type of client, you simply write a view and controller for it and wire them into the existing enterprise model.

Data Access Object (DAO)

Context

Access to data varies depending on the source of the data. Access to persistent storage, such as to a database, varies greatly depending on the type of storage (RDBMS, OODBMS, flat files, and so forth) and the vendor implementation.

Problem

Many real-world J2EE^{TM} applications need to use persistent data at some point. For many applications, the persistent storage is implemented with different mechanisms, and there are marked differences in the APIs used to access these different persistent storage mechanisms. Other applications may need to access data that resides on a different system. For example, the data may reside on a mainframe, an LDAP repository, a B2B service, a credit card bureau, and so forth.

Typically, applications use persistent shared distributed components such as entity beans to represent persistent data. An application is considered to employ bean-managed persistence for its entity beans when these entity beans explicitly access the persistent storage—that is, the entity bean includes code to directly access the persistent storage. An application with simpler requirements may forego using entity beans and instead, use session beans or servlets to directly access the persistent storage to retrieve and modify the data.

Applications can use the JDBC^{TM} API to access data residing in an RDBMS. The JDBC API enables standard access and manipulation of data in a persistent storage, such as a relational database. JDBC enables J2EE applications to use SQL statements, which are the standard means for accessing RDBMS tables. However, even within an RDBMS environment, the actual syntax and format of the SQL statements may vary depending on the particular database product.

There is even greater variation with different types of persistent storage. Access mechanisms, supported APIs, and features vary drastically when comparing a relational type of persistent storage such as RDBMS to other types of persistent stores, such as Object Oriented Databases (OODBMS), file system-based ISAM databases, or simply flat files. Applications that need to access data from a legacy or disparate system (for example, a mainframe, CORBA service or B2B service) are often required to use APIs that may be proprietary. Such disparate data sources offer challenges to the application and can potentially create a tight integration between application code and integration code. When business components—entity beans, session beans and even presentation components like servlets and JSP^{TM})—need to access a data source, they can use the appropriate API to achieve connectivity and manipulate the data source. But, including the connectivity and data access code within these components introduces a tight coupling between the components and the data source implementation. Such code dependencies in the components makes it difficult and tedious to migrate the application from one type of data source to another. When the data source changes, the components need to be changed to handle the new type of data source.

Forces

Components such as bean-managed entity beans, session beans and servlets/JSP need to retrieve and store information from persistent stores and other data sources like legacy systems, B2B, LDAP, and so forth.
Persistent storage APIs vary depending on the product vendor. Other data sources may have APIs that are non-standard and/or proprietary. These APIs and their capabilities also vary depending on the storage type (RDBMS, OODBMS, XML documents, flat files, and so forth). There is a lack of uniform APIs to address the requirements to access such disparate systems.
Components accessing legacy systems to retrieve and store data is typically done using proprietary APIs.

Portability of the components is directly affected when specific access mechanisms and APIs are included in the components.
Components need to be transparent to the actual persistent store or data source implementation to provide easy migration to different vendor products, different storage types, and different data source types.

Solution

Use a Data Access Object to abstract and encapsulate all access to the data source. The Data Access Object manages the connection with the data source to obtain and store data.

The Data Access Object (DAO) is the primary object of this pattern. The DAO implements the access mechanism required to work with the data source. The data source could be a persistent store like an RDBMS, an external service like a B2B exchange, a repository like an LDAP database or a business service accessed via CORBA IIOP or low-level sockets. The business component that relies on the DAO object uses the simpler interface exposed by the DAO for its clients. The DAO completely hides the data source implementation details from its clients. Because the interface exposed by the DAO to clients does not change when the underlying data source implementation changes, this pattern allows the

DAO to adapt to different storage schemes without affecting its clients or business components. Essentially, the DAO acts as an adapter between the component and the data source.

Business Delegate

Context

The system exposes the entire business service API to its clients, often across a network.

Problem

Presentation-tier components interact directly with business services. This direct interaction exposes the underlying implementation details of the business service API to the presentation tier. As a result, the presentation-tier components are vulnerable to changes in the implementation of the business services: when the implementation of the business services change, the exposed implementation code in the presentation tier must change too.

Additionally, there may be a detrimental impact on network-performance because presentation-tier components that use the business service API make too many invocations over the network. This happens when presentation-tier components use the underlying API directly with no client-side caching mechanism or aggregating service.

Lastly, exposing the service APIs directly to the client forces the client to deal with the networking issues associated with the distributed nature of EJB^{TM} technology.

Forces

Presentation-tier clients need access to business services.
Device clients (including rich clients) need access to the business service.
Business service API may change as business requirements evolve.
Goal should be to minimize coupling between presentation-tier clients and the business service API, thus hiding the underlying implementation details of the service, such as lookup and access.

Desirable to implement caching mechanism for business service information.
Desirable to reduce network traffic between client and business services.
Business Delegate may be seen as adding an unnecessary layer between the client and the service thus introducing added complexity and decreasing flexibility.

Solution

Use a Business Delegate to reduce coupling between presentation-tier clients and business services. The Business Delegate hides the underlying implementation details of the business service, such as lookup and access details of the EJB architecture.

The Business Delegate acts as a client-side business abstraction; it provides an abstraction for, and thus hides the implementation of the business services. Using a Business Delegate reduces the coupling between presentation-tier clients and the system's business services. Depending on the implementation strategy, the Business Delegate may shield clients, from possible volatility in the implementation of the business service API. Potentially, this reduces the number of changes that must be made to the presentation-tier client code when the business service API or its underlying implementation changes. However, the Business Delegate may still require modification if the underlying business service API changes.

Often, developers are skeptical when a design goal such as abstracting the business layer causes additional upfront work in return for future gains. However, using this pattern or its strategies results in only a small amount of additional up front work and provides considerable benefits. The main benefit is hiding the details of the underlying service. For example, the client can become transparent to naming and lookup services. The Business Delegate also handles the exceptions from the business services such as EJB exceptions, JMS exceptions and so on. The Business Delegate may intercept such service level exceptions and generate application level exceptions instead. Application level exceptions are easier to handle by the clients, and may be user friendly. These gains present a compelling reason to use the pattern.

Another benefit is that the delegate may cache results. Caching results can significantly improve performance, because it limits unnecessary and potentially costly round trips over the network.

A Business Delegate uses a component called the Lookup Service. The Lookup Service is responsible for hiding the underlying implementation details of the business service lookup code. The Lookup Service may be written as part of the delegate, but we recommend that it be implemented as a separate component, as outlined in the Service Locator [SJC] pattern.

When the Business Delegate is used with a Session Façade, typically there is a 1-1 relationship between the two. This 1-1 relationship exists because logic that might have been encapsulated in a Business Delegate relating to its interaction with multiple business services (creating a 1-many relationship) will often be factored back into a Session Façade.

Finally, it should be noted that this pattern could be used to reduce coupling between other tiers, not simply the presentation and the business tiers.