Web Component Development With Servlet and JSP Technologies

Table of Contents

Chapter 1 Introduction to Java Servlets

1.1 Objectives

Upon completion of this module, you should be able to:

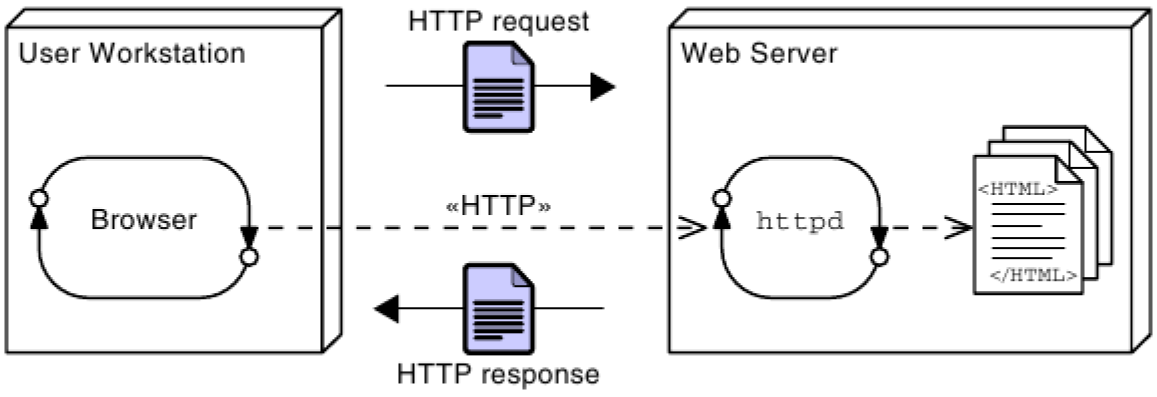
- Outline JavaTM servlet technology

1.2 Web Application Technologies

HTTP is used to transfer instructions and data between machines, while HTML is a document display language that lets users create visually pleasing documents and link from one document to another.

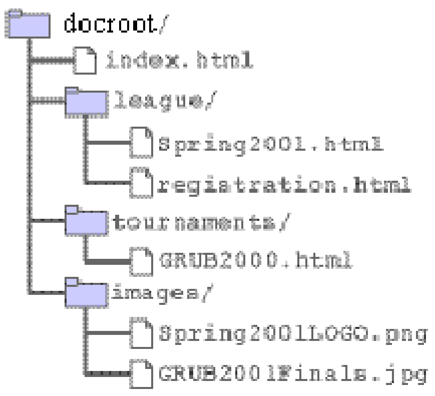
The combination of the HTTP protocol and the HTML page description language are the foundation technologies of the World Wide Web.

1.2.1 HTTP Client-Server Architecture



1.2.2 Web Site Structure

A web site is a collection of HTML pages and other media files that contains all the content that is visible to the user on a given web server.



1.2.2.1 Uniform Resource Locator

A URL locates a specific resource on the Internet. It consists of several parts, and is similar in some ways to a directory and file specification:

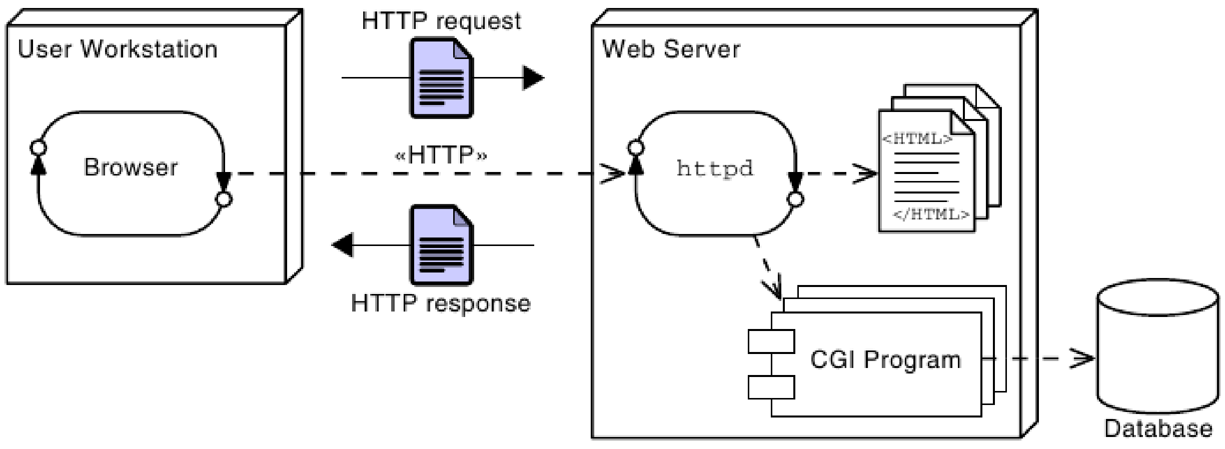
protocol://host:port/path/file

1.3 Web Sites and Web Applications

A web site is a collection of static files, HTML pages, graphics, and various other files. A web application is an element of a web site that provides dynamic functionality on the server. A web application runs a program or programs on the server.

1.4 Execution of CGI Programs

Early in the development of the web, the designers created a mechanism to permit a user to invoke a program on the web server. This mechanism was called the Common Gateway Interface (CGI). When a web site includes CGI processing, this is called a web application.



1.4.1 Advantages and Disadvantages of CGI Programs

1.5 Using Java in the Web

Java gained the ability to service HTTP requests and take its place in the development of dynamic web sites. The earliest technology developed for this was the servlet. Servlets provide a simple framework that allows Java program code to process an HTTP request and create an HTML page in response.

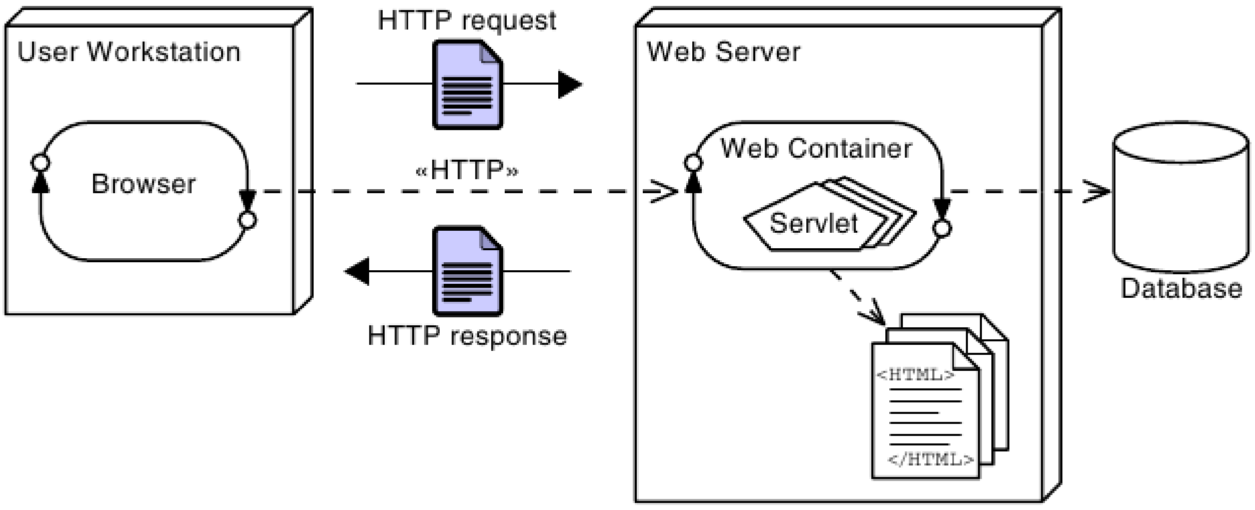
1.5.1 Execution of Java Servlets

The basic processing steps for Java servlets are quite similar to the steps for CGI. However, the servlet runs as a thread in the web container instead of in a separate OS process. The web container itself is an OS process, but it runs as a service and is available continuously. This is opposed to a CGI script in which a new process is created for each request.

1.5.2 Advantages and Disadvantages of Java Servlets

1.6 Java Servlets

Servlets are components that respond to HTTP requests. The web container performs initial processing, and selects the intended servlet to handle the request. The container also controls the life-cycle of the servlet.



1.7 A First Java Servlet

The servlet’s job is two-fold. First, it must perform the required computation; second, it must present the results in a well-formed HTML page.

A simple Servlet code

1.8 HTTP Methods

You might be aware that HTTP provides several ways to send requests, and the GET method is just one of these. Another very common HTTP request is the POST method. As you might guess, the servlet provides a doPost method to handle such a request. The arguments of the doPost method are identical to those of doGet.

Chapter 2 Introduction to Java Server Pages

2.1 Objectives

Upon completion of this module, you should be able to:

- Describe a significant weakness in servlets when used alone
- Write a simple Java Server Page (JSP)
- Describe the translation of a JSP into a servlet
- Understand the basic goals of MVC

2.2 A Weakness in Servlets

Most modern web applications have elegant and complex user interface designs. The designs are often created by dedicated web designers, and even if that is not the case, the HTML pages that implement the designs are frequently created using “What You See Is What You Get” (WYSIWYG) Graphical User Interface (GUI) based design tools. Such design tools work on HTML, not on HTML embedded in Java program code.

Because of this, the pure servlet approach becomes unmanageable—often catastrophically—when maintenance is required on the web pages. Having to re-type all the HTML code back into out.print(...) statements would be a very time consuming and error prone process.

2.2.1 Addressing the Problem With JSPs

Soon after the original introduction of servlets, Java Server Pages, usually called simply JSPs, were introduced to address this problem.

While a servlet is a Java source file containing embedded HTML, a JSP might be considered to be an HTML file with Java embedded in it.

2.3 Key Elements of JSPs

● The bulk of the document is a regular HTML file; indeed, it is entirely legal HTML, and can be edited in a WYSIWYG editor.

● Additional features related to the Java aspects of the JSP are denoted using the <% ... %> pairs.

● Within <% ... %> pairs, three distinct variations are exemplified.

These are:

- ● Comments
- ● A "page directive"
- ● An "expression"

2.4 How a JSP Is Processed

Before a JSP is executed for the first time, it is converted to an equivalent servlet. This is similar to the compilation phase that converts a clean, maintainable source language such as Java into a potentially messy, hard- to-maintain, language such as the Java bytecode or a hardware-specific machine code. In this case, however, the conversion is performed automatically by the web container.

2.5 *Reading Input Parameters From the Browser*

When a browser submits a form, the request commonly includes parameters, that is, data provided by the user. These parameters are available to a servlet through the getParameter method of the request variable.

```
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>JSP Page</title>
</head>
<body>
<h1>Hello
<%= request.getParameter("customerName") %>
</h1>
</body>
</html>
```

2.5.1 Sending Parameters in an HTTP GET Request

The HTTP standard makes it easy to create a GET request that includes a parameter, even without using a form.

2.6 Remaining Problems With the JSP Approach

In effect, the servlet approach results in HTML embedded in a Java source file. The JSP approach results in Java code embedded in an HTML source file. Because of this, programmers and web-designers must share the same file, and changes made by one group might cause bugs in the work of the other group.

2.7 Model, View, and Controller

Based on the historical origin, the code components that handle these three aspects are now called the model, view, and controller. Approaches using these elements are often referred to as MVC or Model-View- Controller designs.

The model is a computational model of the essential problem to be solved. It does not consider how the input is provided, nor the presentation of the output.

The view is responsible for taking the results of the computation from the model and presenting it to the user. Note that in some systems, the “user” might not be human, but might be another computer system. This does not alter the essential design.

The controller is responsible for taking the input, and pre-processing it, so that it can be presented to the model in a generic form that is appropriate to that model. In the web-based system, the controller extracts the essence of the request and argument data from the HTTP request.

Chapter 3 Implementing an MVC Design

3.1 Objectives

Upon completion of this module, you should be able to:

- Implement the controller design element using a servlet
- Implement the model design element using a POJO
- Implement the view design element using a JSP and the Expression Language (EL)
- Connect the model, view, and controller elements to implement a working MVC solution

3.2 Developing the MVC Solution

3.2.1 Responsibilities of the Controller

The controller is the point of contact that the web container knows about. Specifically, the web-container will invoke the controller in response to an HTTP request. The controller must then select a model to process the request, and a view to present the results.

3.2.2 Connecting the Components

For the MVC system to work, communication between the components is necessary. Specifically, the controller must pass request parameter data to the model to support the computation, and the controller must pass results from the model to the view.

3.2.3 Forwarding From a Servlet to a JSP

The mechanism used to invoke the JSP page is called forwarding. Code to perform this is shown below.

```
RequestDispatcher rd = request.getRequestDispatcher("view.jsp");
rd.forward(request, response);
```

Notice that the forwarding is done in two parts. First, a RequestDispatcher object is created by the request. The RequestDispatcher is configured on creation to transfer control to a page called “view.jsp”. Next, the RequestDispatcher is used to pass the original request and response objects into the new page for processing.

3.2.4 Passing Data From the Servlet to the JSP

In fact, there is another map-like feature of the request that is specifically intended to carry information from one page (usually a servlet) so that it will be available to another page (usually a JSP) after a forward operation. This is called the attribute map, and may be read and written using the methods:

ServletRequest.getAttribute(String key)

ServletRequest.setAttribute(String key, Object value)

Using this map, the controller servlet can send data to the view JSP.

3.2.5 Using Data in the JSP

In the JSP, data in the request attribute map are accessible in a number of ways. Perhaps the cleanest and simplest way to access data om the model is using the expression language, usually referred to simply as EL.

So, if a controller servlet writes a data item into an attributed using the following code:

```
request.setAttribute("aValue", 99);
```

Then the value can be read in a JSP page using this syntax:

```
${aValue}
```

3.3 A Complete MVC Example

Chapter 4 The Servlet’s Environment

4.1 Objectives

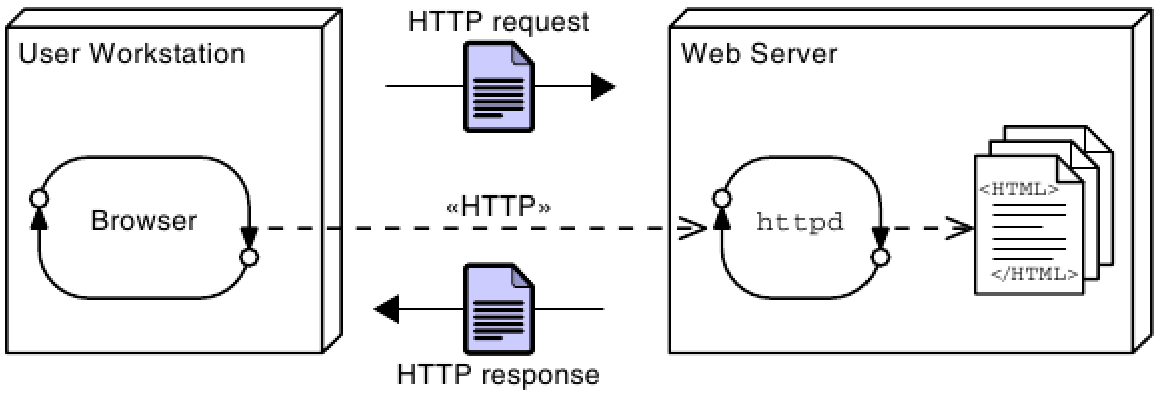
Upon completion of this module, you should be able to:

- Describe the environment in which the servlet runs
- Describe HTTP headers and their function
- Use HTML forms to collect data from users and send it to a servlet
- Understand how the web container transfers a request to the servlet
- Understand and use HttpSession objects

4.2 HTTP Revisited

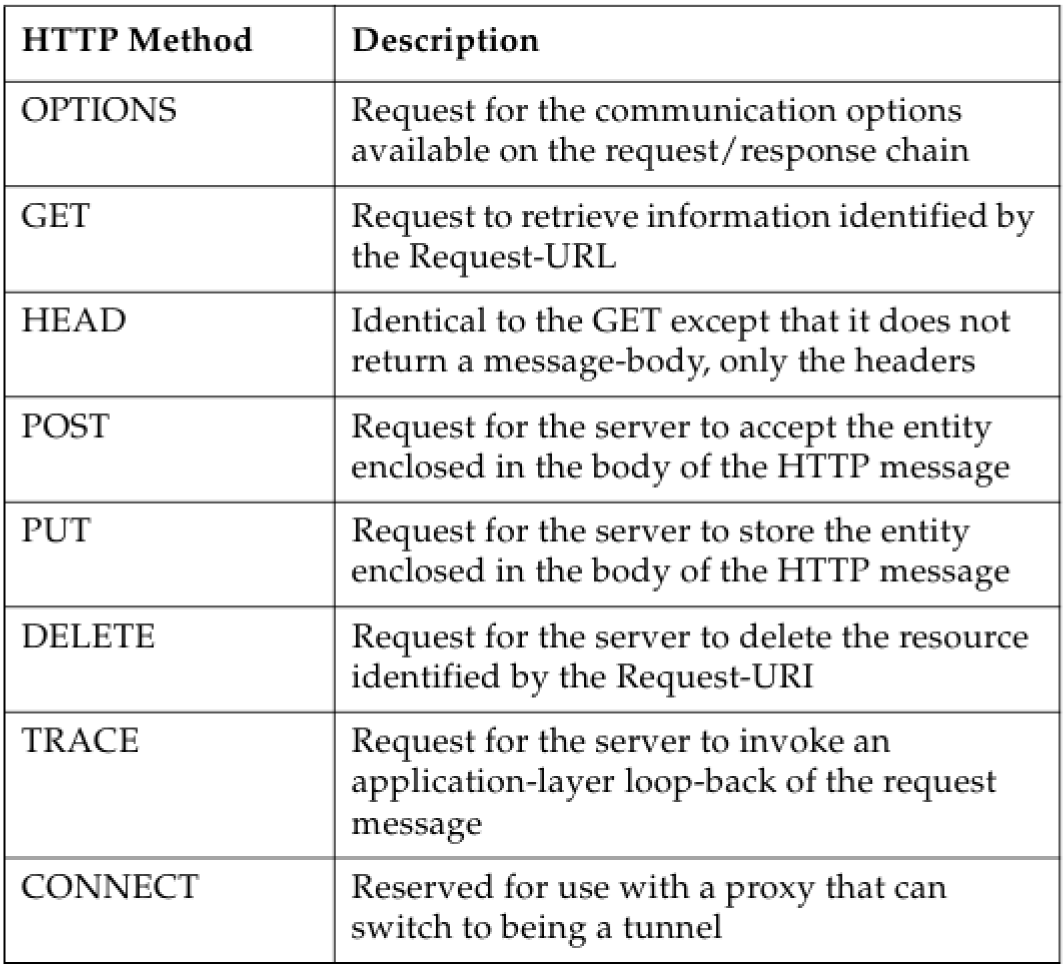
4.2.1 Hypertext Transfer Protocol

In any communication protocol, the client must transmit a request and the server should transmit some meaningful response. In HTTP, the request is some resource that is specified by a URL.



4.2.2 HTTP Methods

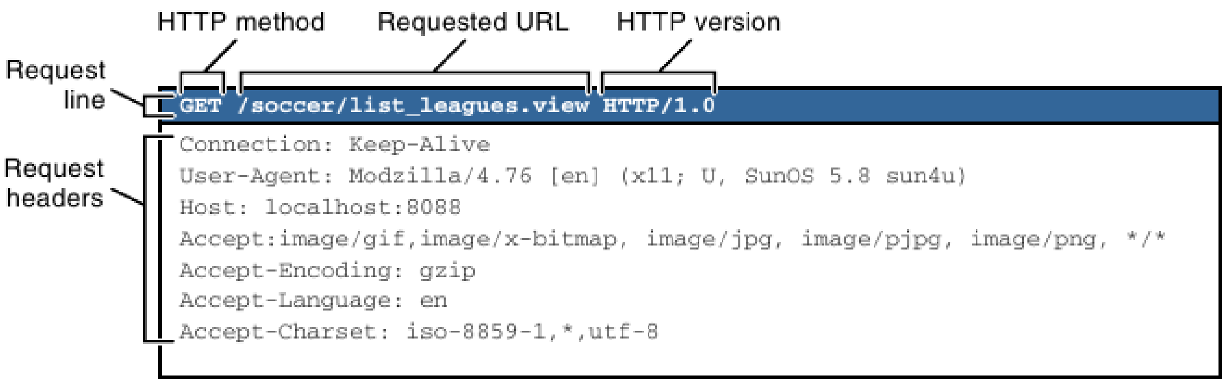
An HTTP request can be made using a variety of methods.



HTTP Method	Description
OPTIONS	Request for the communication options available on the request/response chain
GET	Request to retrieve information identified by the Request-URL
HEAD	Identical to the GET except that it does not return a message-body, only the headers
POST	Request for the server to accept the entity enclosed in the body of the HTTP message
PUT	Request for the server to store the entity enclosed in the body of the HTTP message
DELETE	Request for the server to delete the resource identified by the Request-URI
TRACE	Request for the server to invoke an application-layer loop-back of the request message
CONNECT	Reserved for use with a proxy that can switch to being a tunnel

4.2.3 HTTP Request

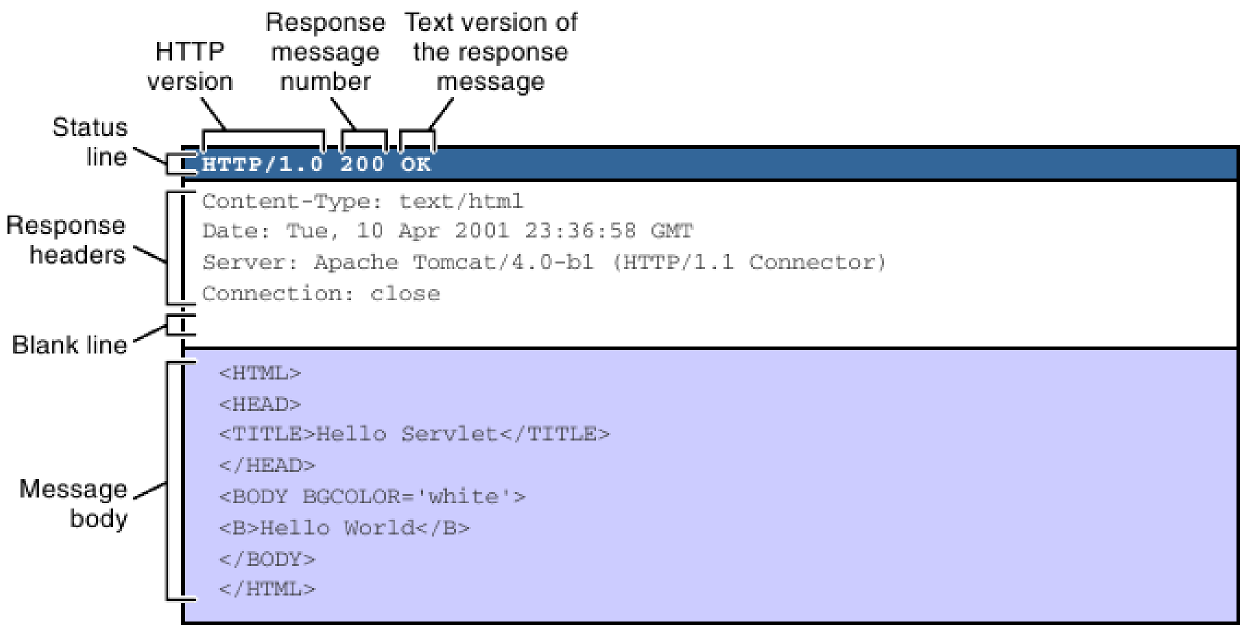
The request stream acts as an envelope to the request URL and message body of the HTTP client request. The first line of the request stream is called the request line. It includes the HTTP method (usually either GET or POST), followed by a space character, followed by the requested URL (usually a path to a static file), followed by a space, and finally followed by the HTTP version number. The request line is followed by any number of request header lines.



```
GET /soccer/list_leagues.view HTTP/1.0
Connection: Keep-Alive
User-Agent: Modzilla/4.76 [en] (x11; U, SunOS 5.8 sun4u)
Host: localhost:8088
Accept:image/gif,image/x-bitmap, image/jpg, image/pjpg, image/png, */*
Accept-Encoding: gzip
Accept-Language: en
Accept-Charset: iso-8859-1,*,utf-8
```

4.2.4 HTTP Response

The response stream acts as an envelope to the message body of the HTTP server response. The first line of the response stream is called the status line. The status line includes the HTTP version number, followed by a space, followed by the numeric status code of the response, followed by a space, and finally followed by a short text message represented by the status code.



```
HTTP/1.0 200 OK
Content-Type: text/html
Date: Tue, 10 Apr 2001 23:36:58 GMT
Server: Apache Tomcat/4.0-b1 (HTTP/1.1 Connector)
Connection: close

  <HTML>
  <HEAD>
  <TITLE>Hello Servlet</TITLE>
  </HEAD>
  <BODY BGCOLOR='white'>
  <B>Hello World</B>
  </BODY>
  </HTML>
```

4.3 Collecting Data From the User

4.3.1 HTML Form Mechanism and Tag

If users are to provide input to a web application, they must be provided with a page that prompts for the desired information, and allows them to enter, or select, their desired values. There must also be a mechanism on the page (usually called simply a form) to allow the user to indicate that the input is complete and should be sent to the server. This indication is usually called “submitting” the form, and a button that triggers this is called a “submit” button.

4.4 Input Types for Use With Forms

Several input types are provided by HTML for use with forms. The following sections introduce three that have very general utility.

4.4.1 Text Input Component

When a user must enter a single line of text, the text input component may be used. The input tag is used to create a textfield component, but you must specify type='text' to get a textfield component. The name attribute of the input tag specifies the field name. The field name and the value entered in the textfield are paired together when the form is submitted to the web container.

4.4.2 Drop-Down List Component

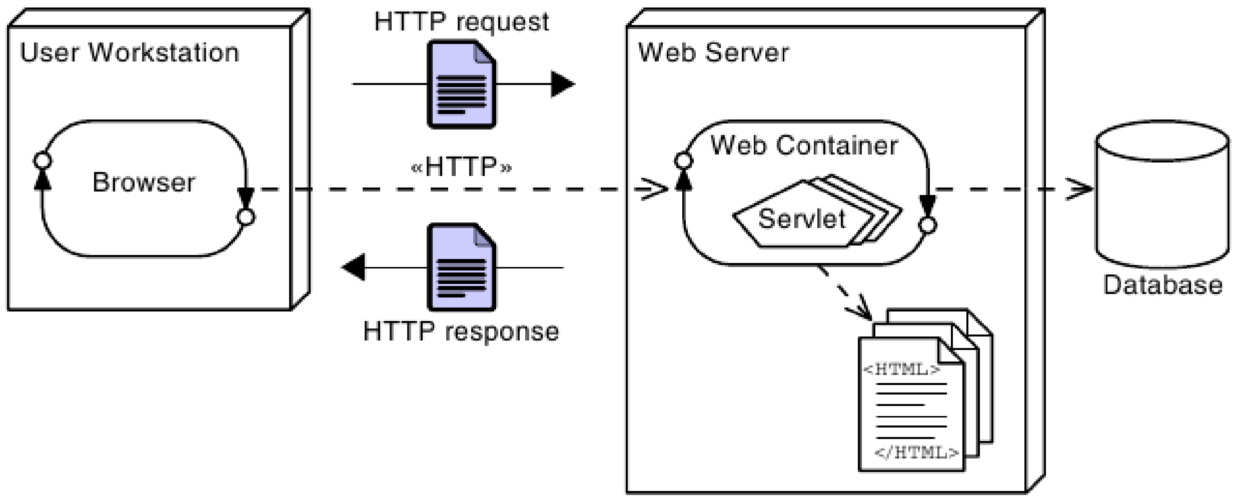
The select tag is used to create a drop-down list component. Similar to the input tag, the select tag uses the name attribute to specify the name of the form field. One or more option tags must be embedded in the select tag. Each option tag provides a single element in the drop-down list. The data sent in the HTTP request on form submission is based on the value attribute of the option tag. The text between the start and end option tags is the content that is displayed in the browser drop-down list.

4.4.3 Submit Button

A submit button can be given a text label at the designer's discretion, though browsers will usually mark the button "Submit" or "Submit Query" if nothing is done to explicitly choose the label.

4.5 Web Container Architecture

Java servlets are components that must exist in a web container. The web container is built on top of the Java Platform, Standard Edition (Java SE) platform and implements the servlet API and all of the services required to process HTTP (and other Transmission Control Protocol/Internet Protocol [TCP/IP]) requests.

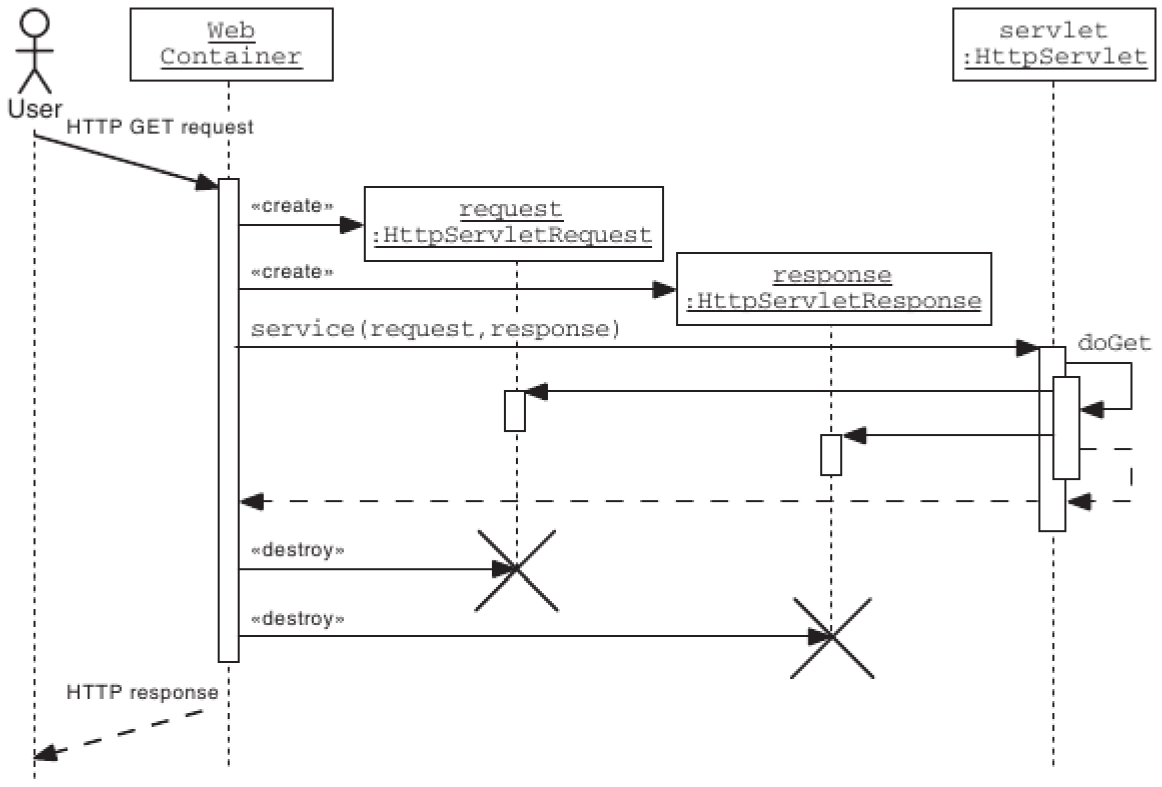


4.5.1 Request and Response Process

1. Browser Connects to the Web Container
2. Web Container Objectifies the Input/Output Streams
3. Web Container Executes the Servlet
4. Servlet Uses the Output Stream to Generate the Response

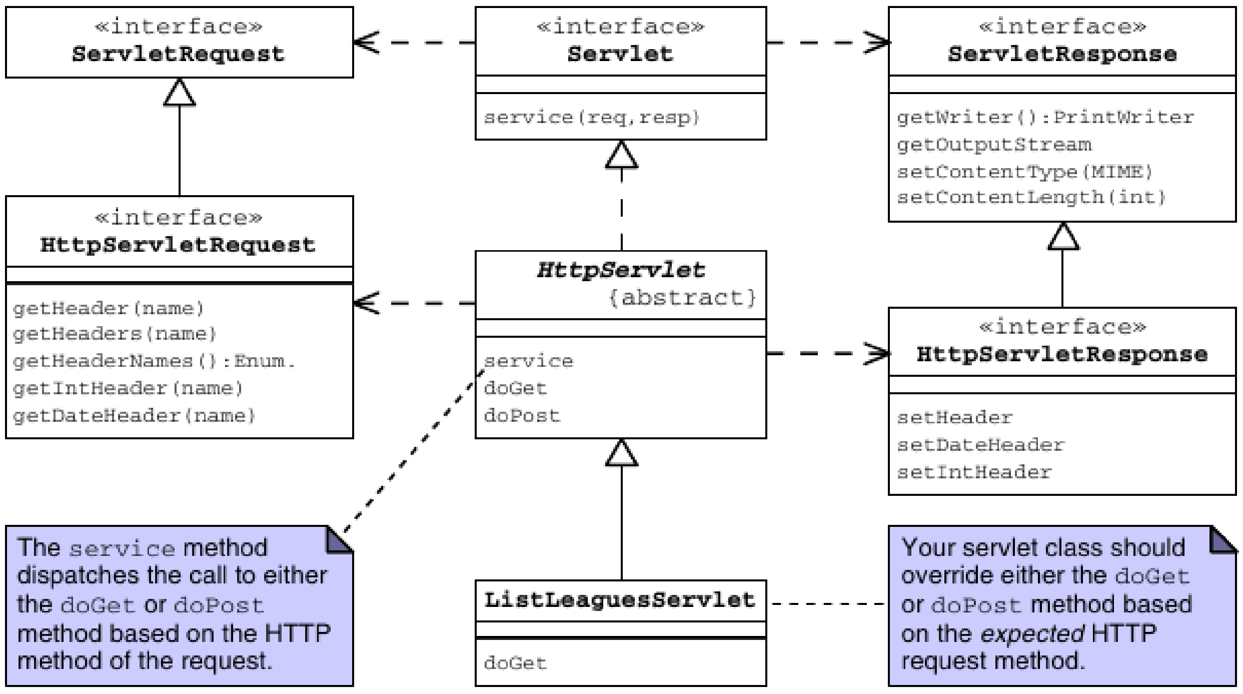
4.5.2 Sequence Diagram of an HTTP GET Request

The web container converts the HTTP request and response streams into runtime objects that implement the HttpServletRequest and HttpServletResponse interfaces. These objects are then passed to the requested servlet as parameters to the service method.



4.6 The HttpServlet API

To create a servlet that responds to an HTTP request, you should create a class that extends the HttpServlet abstract class (provided in the javax.servlet.http package).


«interface» ServletRequest
«interface» Servlet
service(req,resp)
«interface» ServletResponse
getWriter():PrintWriter
getOutputStream
setContentType(MIME)
setContentLength(int)
«interface» HttpServletRequest
getHeader(name)
getHeaders(name)
getHeaderNames():Enum.
getIntHeader(name)
getDateHeader(name)
HttpServlet {abstract}
service
doGet
doPost
«interface» HttpServletResponse
setHeader
setDateHeader
setIntHeader
The service method dispatches the call to either the doGet or doPost method based on the HTTP method of the request.
ListLeaguesServlet
doGet
Your servlet class should override either the doGet or doPost method based on the *expected* HTTP request method.

4.6.1.1 The HttpServletRequest API

The HTTP request information is encapsulated by the HttpServletRequest interface. The getHeaderNames method returns an enumeration of strings composed of the names of each header in the request stream. To retrieve the value of a specific header, you can use the getHeader method. This method returns a String. Some header values might represent integer or date information. There are two convenience methods, getIntHeader and getDateHeader, that perform the conversion for you.

4.6.1.2 The HttpServletResponse API

The HTTP response information is encapsulated by the HttpServletResponse interface. You can set a response header using the setHeader method. If the header value you want to set is either an integer or a date, then you can use the convenience methods setIntHeader or setDateHeader.

4.6.2 Handling Errors in Servlet Processing

One situation in which throwing an exception might be appropriate is when processing page fragments, that is, either servlets or JSPs that are used by inclusion into a larger compound page. In this situation, the fragment is unlikely to be able to decide what to do about the error, but the host page can make a sensible decision. The call to include the page will simply throw the exception that was issued by the included page fragment, allowing the host to use a try/catch block to intercept and handle the problem.

If, for any reason, you decide to throw an exception from a doXxx() servlet method, you should wrap the exception in either a ServletException, or an UnavailableException. The ServletException is used to indicate that this particular request has failed. UnavailableException indicates that the problem is environmental, rather than specific to this request.

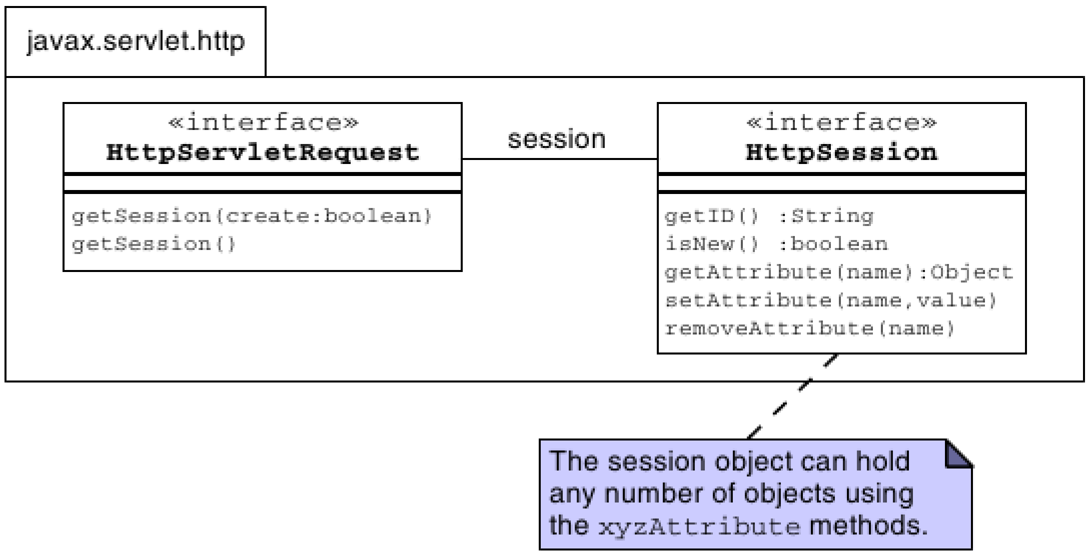
4.7 The HTTP Protocol and Client Sessions

HTTP is a stateless protocol. That means that every request is, from the perspective of HTTP, entirely separate from every other. There is no concept of an ongoing conversation. This provides great potential for scalability and redundancy in servers, because any request can go to any one of a collection of identically equipped servers.

Web servers in a Java EE environment are required to provide mechanisms for identifying clients and associating each client with a map that can be used to store client specific data. That map is implemented using the HttpSession class.

4.8 The HttpSession API

The Servlet specification provides an HttpSession interface that lets you store session data, referred to as attributes. You can store, retrieve, and remove attributes from the session object. The servlet has access to the session object through two getSession methods in the HttpServletRequest object.


javax.servlet.http
«interface»
HttpServletRequest
getSession(create:boolean)
getSession()
session
«interface»
HttpSession
getID() :String
isNew() :boolean
getAttribute(name):Object
setAttribute(name,value)
removeAttribute(name)
The session object can hold any number of objects using the xyzAttribute methods.

4.8.1 Storing Session Attributes

To store a value in a session object, use the setAttribute method.

4.8.2 Retrieving Session Attributes

You can use the getAttribute method to retrieve data previously stored in the session object.

4.8.3 Closing the Session

When the web application has finished with a session, such as if the user logs out, you should call the invalidate method on the HTTPSession object.

4.8.4 Additional Session Methods

4.9 Using Cookies for Client-Specific Storage

Cookies allow a web server to store information on the client machine. Data are stored as key-value pairs in the browser and are specific to the individual client.

● Cookies are sent in a response from the web server.

● Cookies are stored on the client’s computer.

● Cookies are stored in a partition assigned to the web server’s domain name. Cookies can be further partitioned by a path within the domain.

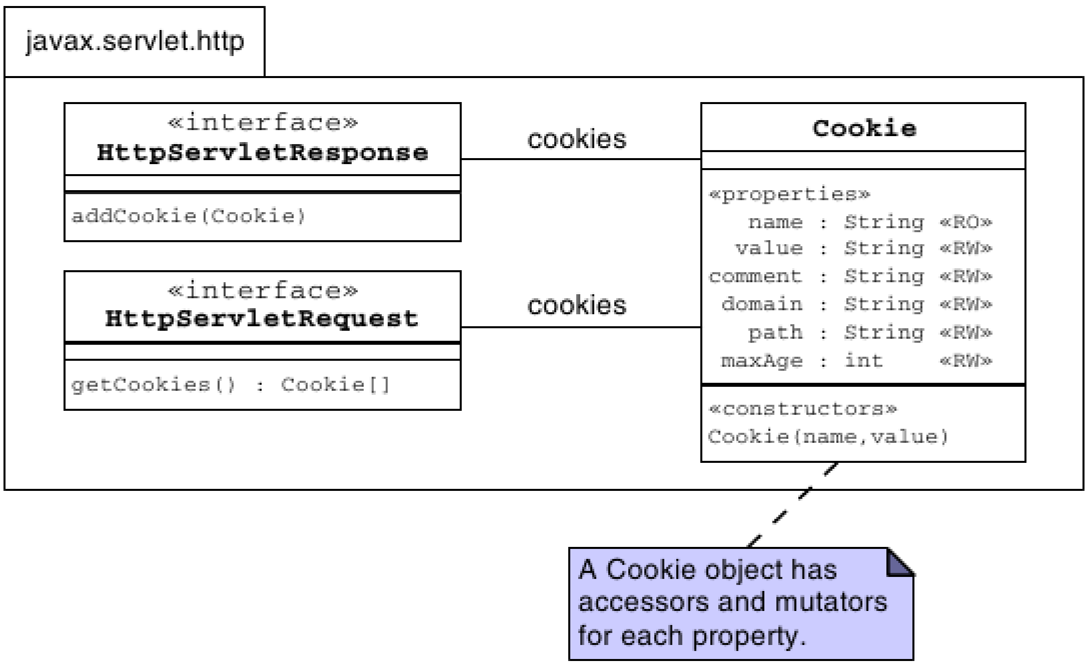
● All cookies for that domain (and path) are sent in every request to that web server.

● Cookies have a life span and are flushed by the client browser at the end of that life span.

● Cookies can be labelled “HTTP-Only” which means that they are not available to scripting code such as JavaScript. This enhances security.

4.9.1 Cookie API

You can create cookies using the Cookie class. You can add cookies to the response object using the addCookie method. This sends the cookie information over the HTTP response stream. The information is stored on the user’s computer (through the web browser). You can access cookies sent back from the user’s computer in the HTTP request stream using the getCookies method.

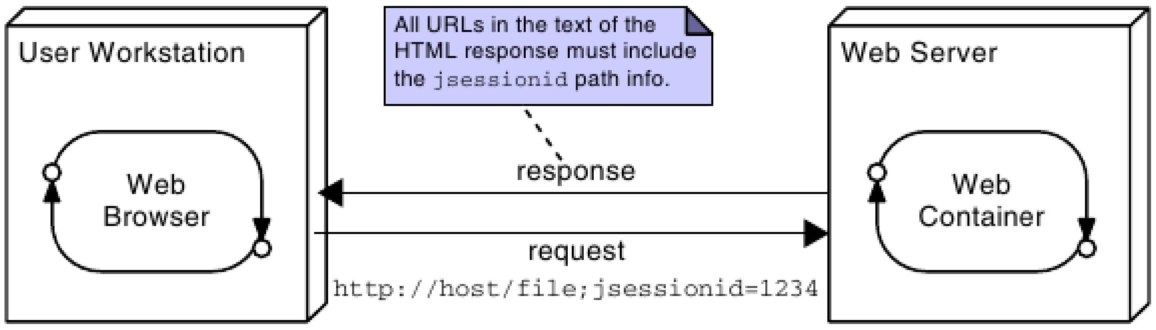

javax.servlet.http
«interface»
HttpServletResponse
addCookie(Cookie)
«interface»
HttpServletRequest
getCookies() : Cookie[]
cookies
cookies
Cookie
«properties»
name : String «RO»
value : String «RW»
comment : String «RW»
domain : String «RW»
path : String «RW»
maxAge : int «RW»
«constructors»
Cookie(name,value)
A Cookie object has accessors and mutators for each property.

4.10 Using Cookies Example

4.11 Using URL-Rewriting for Session Management

URL-rewriting is an alternative session management strategy that the web container must support. Typically, a web container attempts to use cookies to store the session ID. If that fails (that is, the user has cookies turned off in the browser), then the web container tries to use URL-rewriting.

URL-rewriting is implemented by attaching additional information (the session ID) onto the URL that is sent in the HTTP request stream .


User Workstation
Web Browser
All URLs in the text of the HTML response must include the jsessionid path info.
response
request
http://host/file;jsessionid=1234
Web Server
Web Container

Chapter 5 Container Facilities for Servlets and JSPs

5.1 Objectives

Upon completion of this module, you should be able to:

- Describe the purpose of deployment descriptors
- Determine the context root for a web application
- Create servlet mappings to allow invocation of a servlet
- Create and access context and init parameters
- Use the @WebServlet and @WebInitParam annotations to configure your servlets

5.2 Packaging Web Applications

The JAR file that is used to package a web application has a specific set of structural and content rules that make it a little different from other, more general, archives. Because of this, the file is given the extension .war, indicating that it is a web archive, rather than .jar, which would not distinguish it from archives created for any other purpose. The file is often referred to as a WAR file, or simply a WAR.

5.2.1 The Default Context Root

The context root is the base URL that is used as a prefix when locating pages and servlets within the application. The most basic default for the context root is that it uses the name of the war file from which the application was deployed.

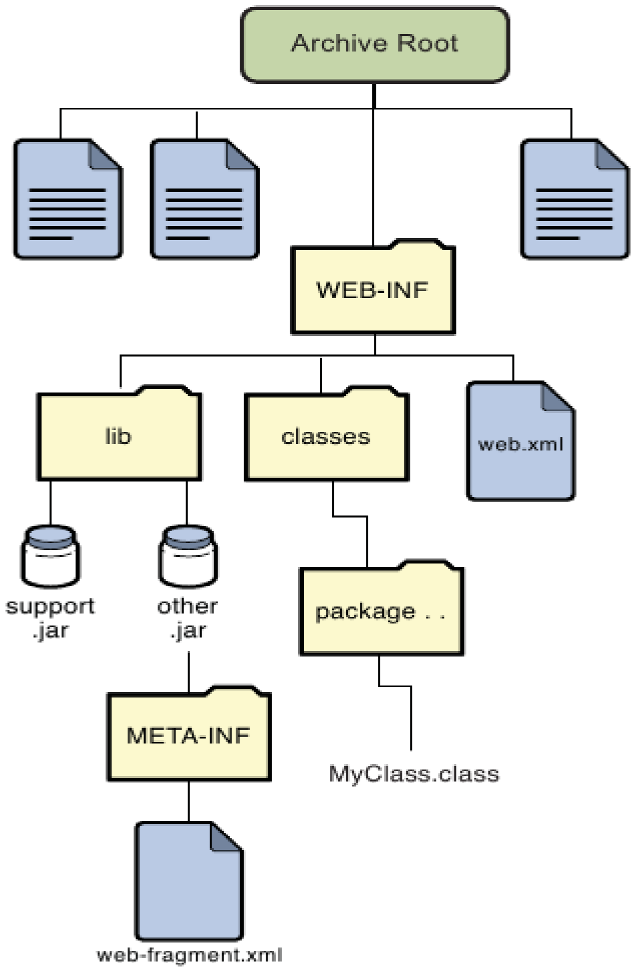
For example, if a war file called AnApplication.war contains a file startingPoint.html in the root of the archive, then when deployed to a local server, the URL used to reach the startingPoint.html file might be:

http://localhost:8080/AnApplication/startingPoint.html

In this way, the context root is /AnApplication.

5.2.2 Essential Structure of a WAR file

Figure below shows the essential structure of a WAR file. Notice that the root of the archive, and arbitrary directories below the root are all proper places to place HTML files, JSPs, and images and other resources. Content in these locations will be directly accessible from the application's context root (this is not the same as the root of the server). A special directory, called WEB-INF is also placed in the root of the archive. If resources are placed there, the container must be given special instructions to make those resources available to a client.



5.3 Deployment Descriptors

In versions of Java EE prior to EE 6, deployment configuration was supported using the web.xml file. This file was required and is generally known as the deployment descriptor (DD). This type of external deployment descriptor facilitates reuse of code in differing environments without problems arising from hard-coded values needing to change, and supports the notion of keeping separate things that change separately.

In addition to allowing configuration to be done using annotations, another method of specifying deployment information is also supported. This is the use of web-fragment.xml files in libraries.

5.3.1 The web-fragment.xml Files

The web-fragment.xml file provides a mechanism to include deployment information that is specific to a library, without having to edit the main deployment descriptor for the whole web application. This reflects the OO principle of keeping unrelated things apart.

5.3.2 Controlling Configuration

With three elements capable of providing configuration information, web.xml, web-fragment.xml, and annotations, it is possible for these to provide conflicting information. To address this, some rules and controls are provided. Three are particularly significant, and these are:

- Precedence
- <metadata-complete> tag
- Ordering

5.3.3 Dynamic Configuration of Web Applications

Java EE 6 adds features that allow the programmatic configuration of servlets, filters, and related components. This feature is only available as the web application is being loaded, specifically during the time that the servlet context is being initialized. It is perhaps most useful for framework creators.

5.4 Servlet Mapping

A mapping is information that specifies that a particular URL, or group of URLs, refer to a particular named servlet. The name is not the class name, but is an internal reference within the web container to a particular servlet class configured in a particular way.

```
<web-app [...] >
  <servlet>
    <servlet-name>MyServletName</servlet-name>
    <servlet-class>SL314.package.MyServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>MyServletName</servlet-name>
    <url-pattern>/MyServlet</url-pattern>
  </servlet-mapping>
</web-app>
```

5.4.1 Multiple and Wildcard URL Patterns

It is quite common to find a single servlet being used to respond to a variety of URLs. This is supported in two ways: First, the <url-pattern> XML element may be repeated. Second, the pattern specified by a URL may be a wildcard, ending with an asterisk.

```
<servlet-mapping>
        <servlet-name>MyServlet</servlet-name>
        <url-pattern>/MyServlet</url-pattern>
        <url-pattern>/YourServlet</url-pattern>
        <url-pattern>/HisServlet/*</url-pattern>
</servlet-mapping>
```

5.4.2 Mapping Using Annotations

Since Java EE 6, servlet mappings can be achieved using annotations. A servlet should be given the annotation:

javax.servlet.annotation.WebServlet

This annotation supports various optional elements, but a common form is that which has been used in examples in previous modules:

```
@WebServlet(name="MyServletName",
urlPatterns={"/MyServlet", “/YourServlet”, “/HisServlet/*”})
```

5.4.3 Invoking a Servlet by Name

A servlet can be invoked directly by its name through a RequestDispatcher, in a forward or include call. The necessary dispatcher is obtained using the servlet context which offers a method getNamedDispatcher(String).

5.5 Servlet Context Information

The deployment descriptor can include configuration information for the servlet. Two configuration elements of frequent utility are context parameters and init parameters.

5.5.1 Context Parameters

The term “servlet context” essentially refers to the web application and the container in which the servlet runs. An interface javax.servlet.ServletContext provides access to several aspects of this environment, but in the context of this discussion, the methods of interest are two methods that allow read-only access to a map of context parameters. These methods are:

String getInitParameter(String name)

Enumeration<String> getInitParameterNames()

5.5.2 Supplying Context Parameters

To specify a context parameter using a web.xml, or web-fragment.xml file, the specification is quite simple. Here is an example.

```
<web-fragment [...]>
        <context-param>
                <param-name>fragmentContext</param-name>
                <param-value>fragment Context value</param-value>
        </context-param>
</web-fragment>
```

5.6 Servlet Initialization Parameters

In addition to configuration parameters for the entire application, initialization parameters can be associated with the servlet individually. These are typically known simply as init parameters. This association may be achieved using the deployment descriptor file like this:

```
<servlet>
        <servlet-name>MyServlet</servlet-name>
        <servlet-class>SL314.package.MyServlet</servlet-class>
        <init-param>
                <param-name>name</param-name>
                <param-value>Fred Jones</param-value>
        </init-param>

</servlet>
```

5.6.1 Using Annotations

Because init parameters are associated directly with the servlet, they can be specified using the annotations mechanism. The annotation that would be equivalent to the preceeding declaration would be:

```
@WebServlet(
name=”MyServlet”,
urlPatterns=({“/MyServ”}),
initParams =
{@WebInitParam(name="name", value="Fred Jones")}
)
```

5.6.2 Reading Servlet Initialization Parameters

The servlet itself provides direct access to servlet initialization parameters using the method getInitParameter(String), and another method of the same signature is provided in ServletConfig object. So, these two lines of code, executed in the servlet, are equivalent:

```
name = this.getInitParameter("name");
name = getServletConfig().getInitParameter("name");
```

5.7 Other Configuration Elements

The deployment descriptor can contain elements of many other types beyond those discussed in this module. Similarly, several other annotations exist for configuration purposes.

While these are topics for later modules, you should know that the deployment descriptor is used to configure security, and connections to other aspects of a Java EE system, such as Enterprise JavaBeansTM (EJBTM), message queues, and databases.

Chapter 6 More View Facilities

6.1 Objectives

Upon completion of this module, you should be able to:

- Understand and use the four scopes
- Understand and use the EL
- Recognize and use the EL implicit objects

6.2 Scopes

To review, the locations discussed so far are:

The request object – Contains a map of attributes and is shared across forward calls. Its scope is bounded by a request from a single client.

The session object – Contains a map of attributes shared across all calls from a single browser within a single session. Session duration may be bounded by time limits or by a login/logout type process.

The servlet's initialization parameters – Is a map of read-only configuration data available to this servlet from the deployment descriptor.

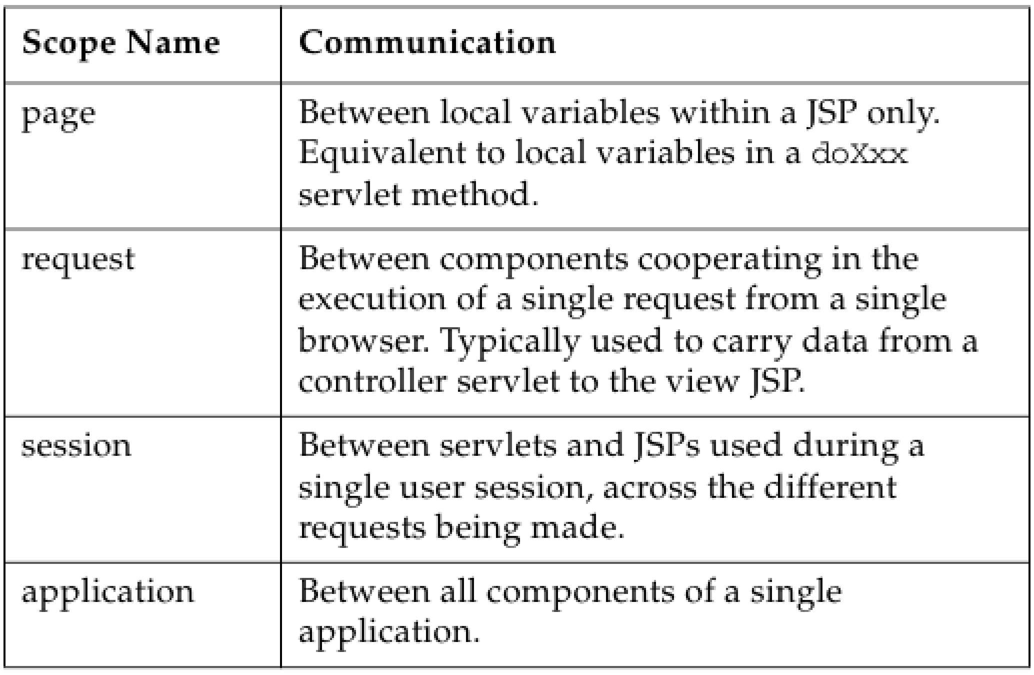
The application's initialization parameters - Is a map of read-only configuration data from the deployment descriptor and available to any servlet in the application.

The browser's cookies - Is a map of cookie values set in the browser by the applicaion.

There is another place that data can be stored for sharing between different parts of the application. The servlet context object itself provides a read-write map of attributes that are shared throughout the entire application

Three of these locations are referred to as scopes. These are the request scope, the session scope, and the application scope. The application scope is the name given to the map in the servlet context.

EL Implict Objects



Scope Name	**Communication**
page	Between local variables within a JSP only. Equivalent to local variables in a `doXxx` servlet method.
request	Between components cooperating in the execution of a single request from a single browser. Typically used to carry data from a controller servlet to the view JSP.
session	Between servlets and JSPs used during a single user session, across the different requests being made.
application	Between all components of a single application.

6.3 More Details About the Expression Language (EL)

EL was introduced in “Using Data in the JSP” . As you might expect, EL is considerably more powerful than was indicated at that point.

Syntax Overview

The syntax of EL in a JSP page is as follows:

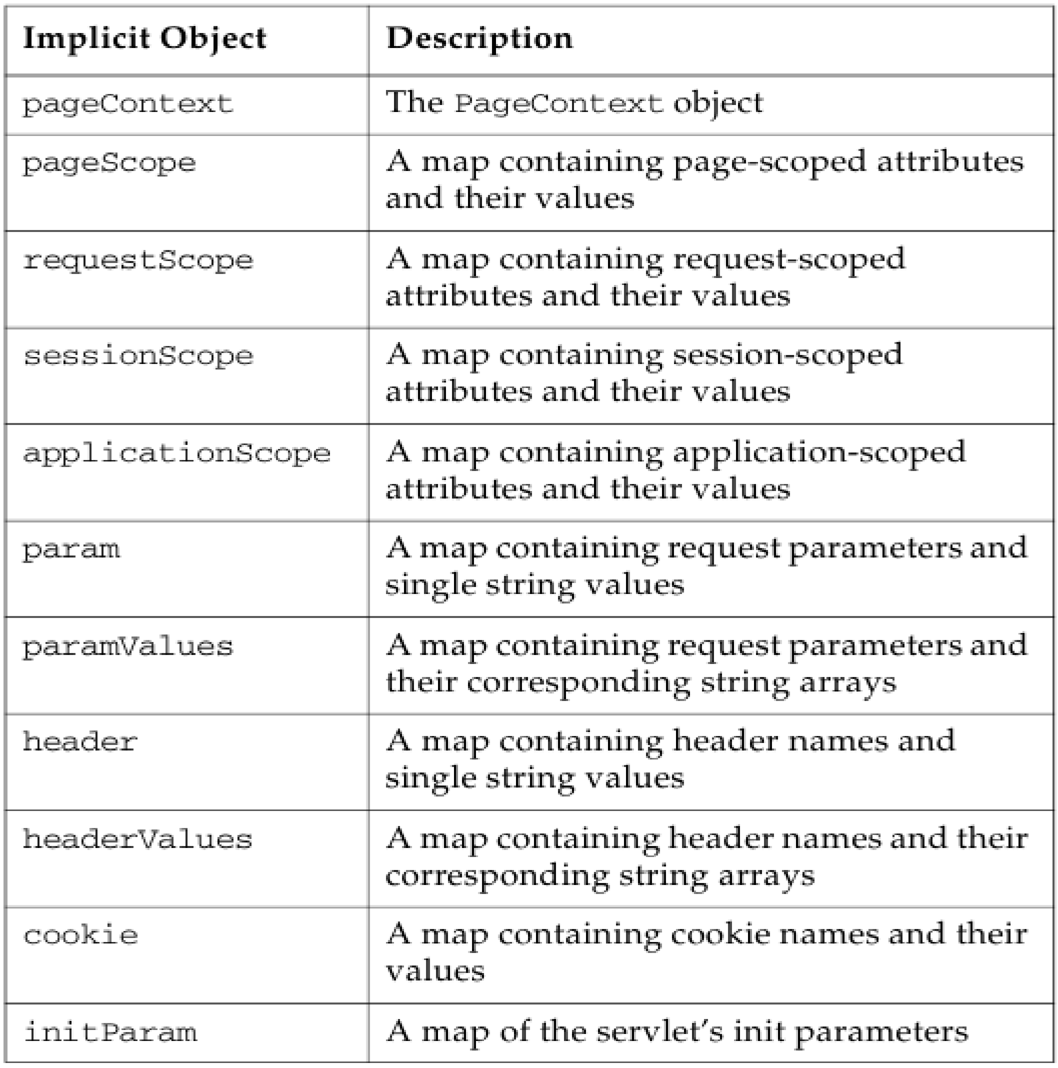
${expr}

6.3.1 EL and Scopes

In fact, EL scans all four scopes looking for a bean with the name given in the expression. Clearly if two beans with the same name exist in different scopes, confusion might arise. However, the scopes are scanned in a fixed order: page, request, session, and application.

6.3.2 EL Implicit Objects

The variables used to explicitly name a scope containing a bean are called implicit objects. Implicit



Implicit Object	Description
pageContext	The PageContext object
pageScope	A map containing page-scoped attributes and their values
requestScope	A map containing request-scoped attributes and their values
sessionScope	A map containing session-scoped attributes and their values
applicationScope	A map containing application-scoped attributes and their values
param	A map containing request parameters and single string values
paramValues	A map containing request parameters and their corresponding string arrays
header	A map containing header names and single string values
headerValues	A map containing header names and their corresponding string arrays
cookie	A map containing cookie names and their values
initParam	A map of the servlet's init parameters

objects are pre-defined variable names known to EL that provide access to bean-like data.

6.3.3 The Dot Operator In EL

It is probably obvious by now that the EL syntax allows cascading of the dot operator to select elements from within elements. This is exactly equivalent to calling methods on the return values of methods in Java itself. The name elements used with the dot operator must follow the naming conventions for Java identifiers.

6.3.4 Array Access Syntax With EL

If a bean returns an array, an element in the array can be accessed by providing its index:

${paramValues.fruit[2]}

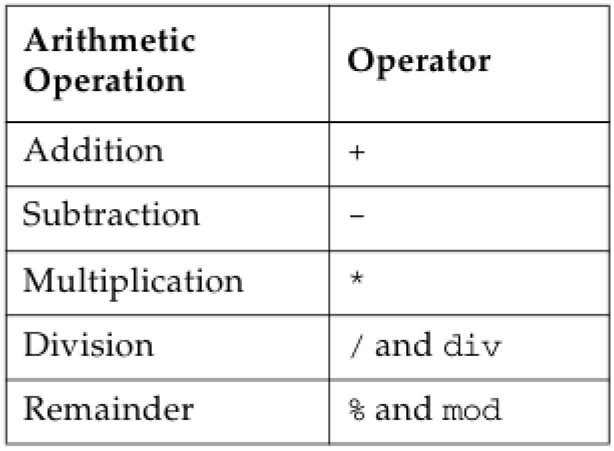
As with Java, the dot operator may be used to access elements of an object after it has been selected from an array.

6.3.5 EL and Errors

EL silently hides errors that arise from null references or not-found variables. That is, such expressions translate to empty strings. This is deliberate, as it ensures that errors do not mess up the pages that the user sees. Of course, the information on the page might be incomplete.

6.3.6 EL Arithmetic Operators

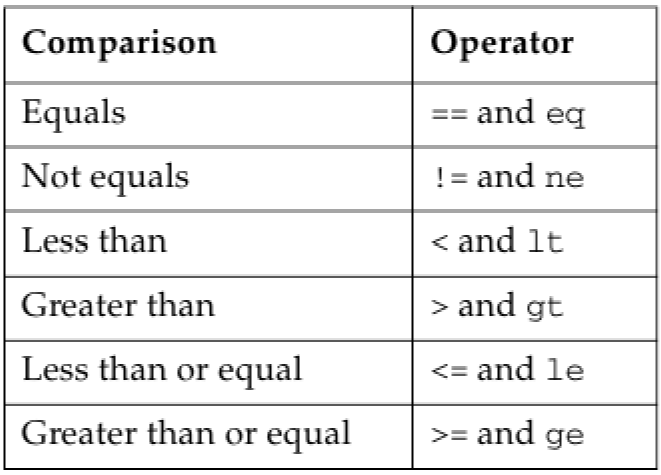
The five arithmetic operators that are available in EL.



Arithmetic Operation	Operator
Addition	+
Subtraction	-
Multiplication	*
Division	/ and `div`
Remainder	% and `mod`

6.3.7 Comparisons and Logical Operators

The six comparison operators available in EL.



Comparison	Operator
Equals	== and `eq`
Not equals	!= and `ne`
Less than	< and `lt`
Greater than	> and `gt`
Less than or equal	<= and `le`
Greater than or equal	>= and `ge`

6.4 Configuring the JSP Environment

You can modify the behavior of JSP pages in the application using the jsp-config tag in the web.xml deployment descriptor.

```
<jsp-config>
        <jsp-property-group>
                <url-pattern>/scripting_off/*</url-pattern>
                <scripting-invalid>true</scripting-invalid>
        </jsp-property-group>
        <jsp-property-group>
                <url-pattern>/EL_off/*</url-pattern>
                <el-ignored>true</el-ignored>
        </jsp-property-group>
</jsp-config>
</web-app>
```

6.5 Presentation Programming

6.5.1 Standard Custom Tags

When the tag library mechanism was introduced, tags were called “custom tags,” as each developer was expected to create their own. However, a popular set of tags was rapidly adopted as the “standard tags” (or JSTL). For this reason, you’ll often here tags referred to as both custom, and standard, sometimes in the same sentence. Strictly speaking, there is now a set of standard tags, and others are custom tags. Both use the same custom tags mechanism for their implementation and execution.

6.5.2 Tag Example

6.6 Key View Programming Tags

Two tags in particular provide great value for view programming. Good design suggests you should take care to use them to perform the programmatic operations that relate specifically to the presentation.

6.6.1 JSTL if Tag

The if tag is a conditional tag in the core library. You use this tag to conditionally evaluate what is contained in the tag’s body or to set a variable based on the evaluation of a test. When a body is supplied, the body is only evaluated if the test expression is true:

```
<c:if test=”expression” var=”varName”
[scope=”{page|request|session|application}”] >
body evaluated if expression evaluates to true
</c:if>
```

The var attribute specifies the variable name into which the results of the test (true or false) will be stored. The scope attribute specifies where the variable will be stored.

6.6.2 JSTL forEach Tag

The forEach tag provides a mechanism for iteration over the body of the tag within a JSP page. The attributes of the tag are items, var, varStatus, begin, end, and step. You use this tag to iterate over an existing collection or for a specified number of iterations. To iterate over a collection, the items attribute is used to supply the name of a collection.

```
<c:forEach items=”collection” [var=”varName”]
[varStatus=”varStatusName”]
[begin=”begin”] [end=”end”] [step=”step”]>
body content
</c:forEach>
```

JSTL Example:

Chapter 7 Developing JSP Pages

7.1 Objectives

Upon completion of this module, you should be able to:

- Describe JSP page technology
- Write JSP code using scripting elements
- Write JSP code using the page directive
- Write JSP code using standard tags
- Write JSP code using the EL
- Configure the JSP page environment in the web.xml file

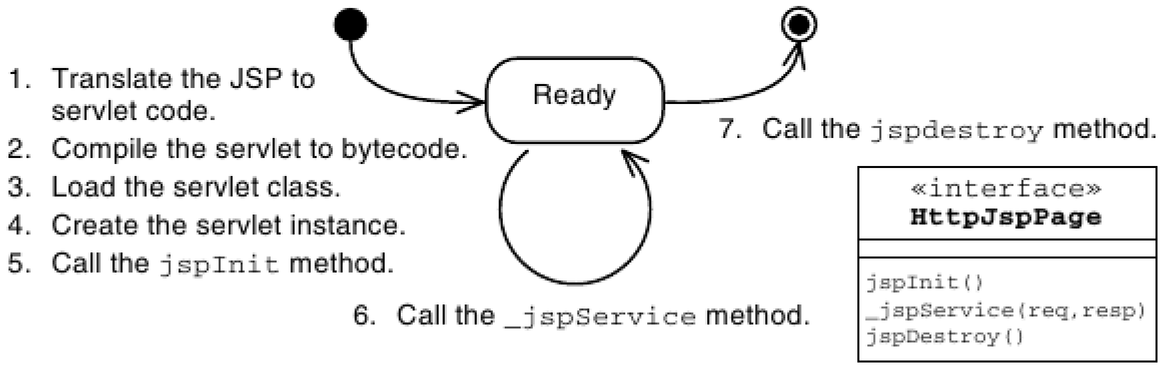
7.2 JavaServer Pages Technology

This module introduces the tools and techniques of those original JSPs, so that you will be equipped to work on such pages should you be called upon to do so.

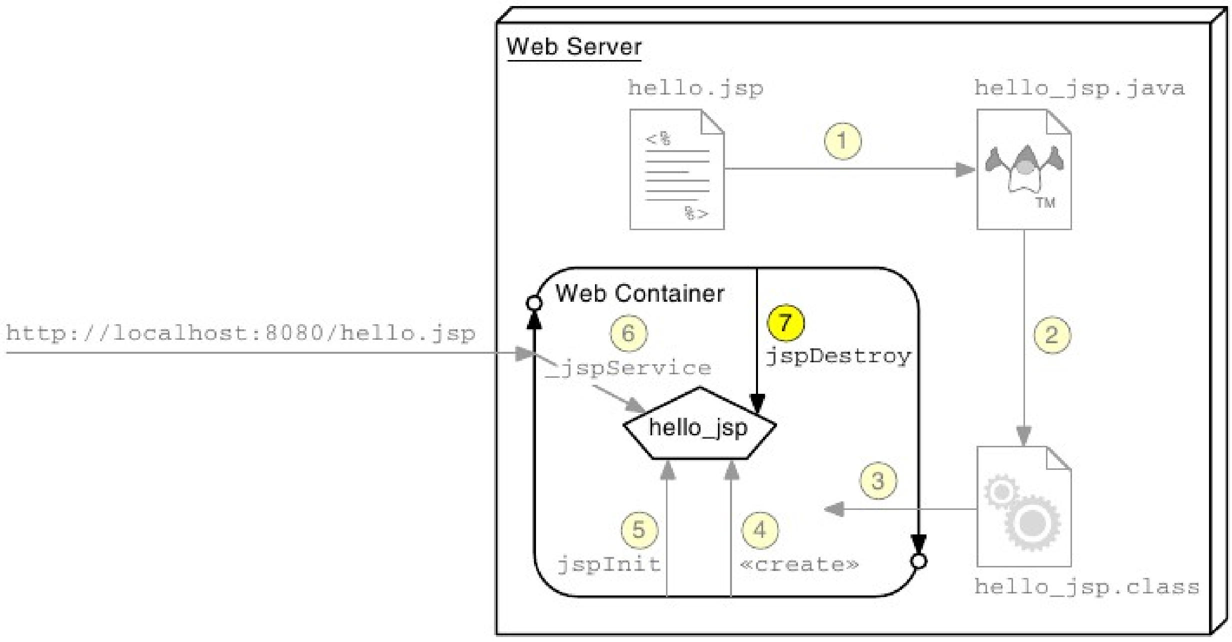
7.2.1 How a JSP Page Is Processed

A JSP page is essentially source code, and must be converted into a servlet before it can service requests. Figure shows the steps of JSP page processing.

1. Translate the JSP to servlet code.
2. Compile the servlet to bytecode.
3. Load the servlet class.
4. Create the servlet instance.
5. Call the jspInit method.
6. Call the _jspService method.


1. Translate the JSP to servlet code.
2. Compile the servlet to bytecode.
3. Load the servlet class.
4. Create the servlet instance.
5. Call the jspInit method.
Ready
6. Call the _jspService method.
7. Call the jspdestroy method.
«interface»
HttpJspPage
jspInit()
_jspService(req,resp)
jspDestroy()

1) **JSP Page Translation**
2) **JSP Page Compilation**
3) **JSP Page Class Loading**
4) **JSP Page Servlet Instance**
5) **JSP Page Initialization**
6) **JSP Page Service**
7) **JSP Page Destroyed**

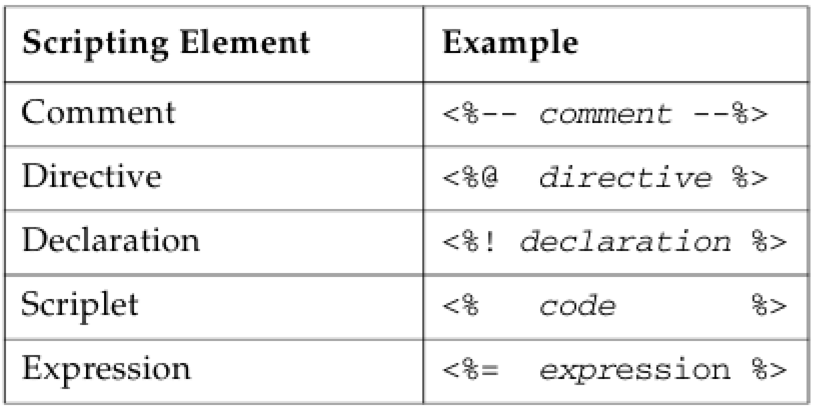

Web Server
hello.jsp
hello_jsp.java
1
2
3
hello_jsp.class
Web Container
http://localhost:8080/hello.jsp
6
_jspService
7
jspDestroy
hello_jsp
5
jspInit
4
«create»

7.3 Writing JSP Scripting Elements

JSP scripting elements are embedded with the <% %> tags and are processed by the JSP engine during translation of the JSP page. Any other text in the JSP page is considered part of the response and is copied verbatim to the HTTP response stream that is sent to the web browser.

```
<html>
<%-- scripting element --%>
</html>
```

Lists the five types of scripting elements.



Scripting Element	**Example**
Comment	`<%-- comment --%>`
Directive	`<%@ directive %>`
Declaration	`<%! declaration %>`
Scriplet	`<% code %>`
Expression	`<%= expression %>`

7.3.1 Comments

There are three types of comments permitted in a JSP page:

- **HTML comments**
- **JSP page comments**
- **Java technology comments**

7.3.2 Directive Tag

A directive tag provides information that affects the overall translation of the JSP page. The syntax for a directive tag is as follows:

<%@ DirectiveName [attr=”value”]* %>

Three types of directives are used in JSP pages: page, include, and taglib.

7.3.3 Declaration Tag

A declaration tag lets you include members in the JSP servlet class, either attributes or methods. The syntax for a declaration tag is as follows:

<%! JavaClassDeclaration %>

7.3.4 Scriptlet Tag

A scriptlet tag lets the JSP page developer include arbitrary Java technology code in the _jspService method. The syntax for a scriptlet tag is as follows:

<% JavaCode %>

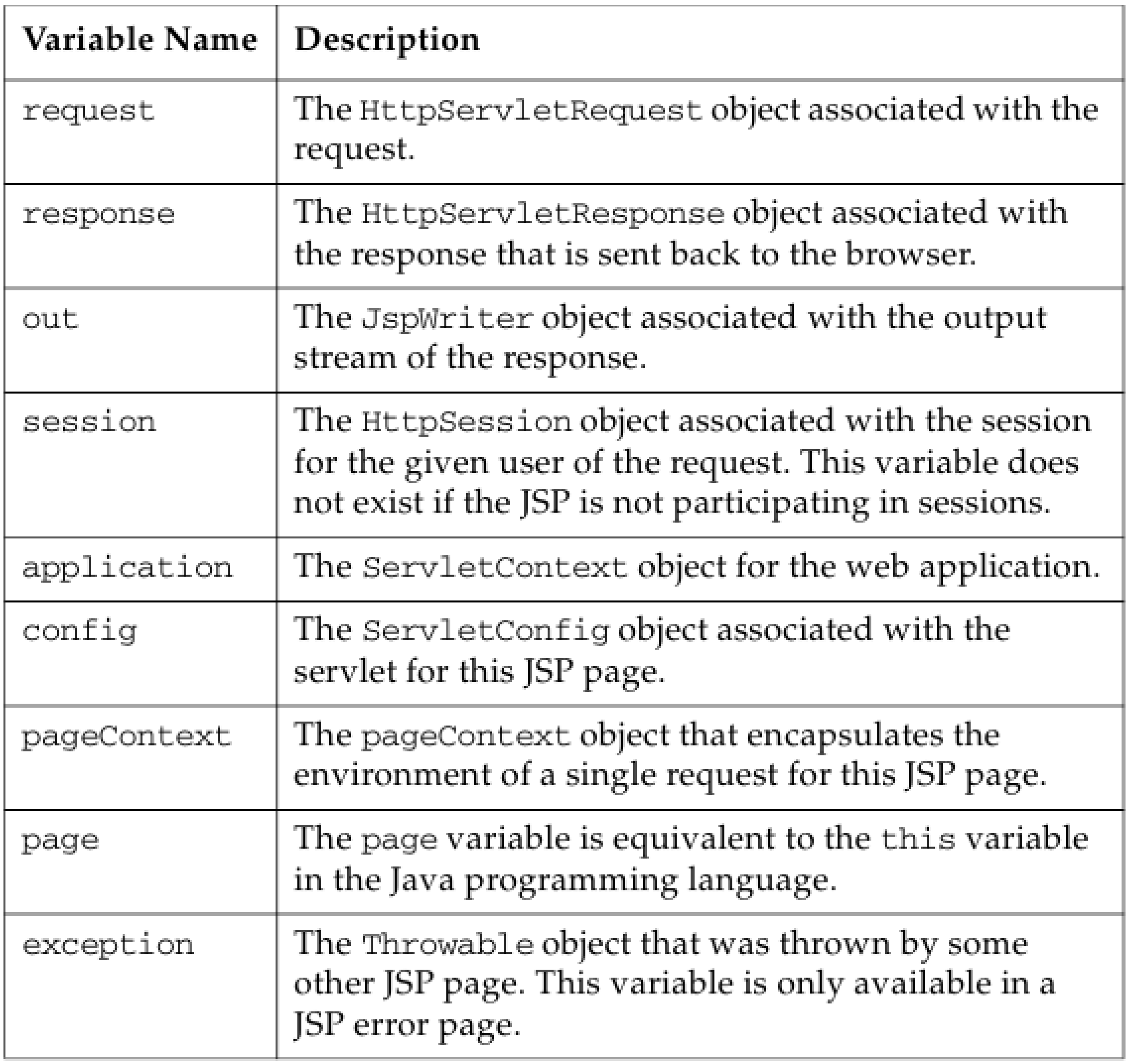
7.3.5 Expression Tag

An expression tag holds a Java language expression that is evaluated during an HTTP request. The result of the expression is included in the HTTP response stream. The syntax for an expression tag is as follows:

```
<%= JavaExpression %>
```

7.3.6 Implicit Variables

The JSP engine gives you access to the following implicit variables, also called implicit objects, in scriptlet and expression tags.



Variable Name	Description
`request`	The `HttpServletRequest` object associated with the request.
`response`	The `HttpServletResponse` object associated with the response that is sent back to the browser.
`out`	The `JspWriter` object associated with the output stream of the response.
`session`	The `HttpSession` object associated with the session for the given user of the request. This variable does not exist if the JSP is not participating in sessions.
`application`	The `ServletContext` object for the web application.
`config`	The `ServletConfig` object associated with the servlet for this JSP page.
`pageContext`	The `pageContext` object that encapsulates the environment of a single request for this JSP page.
`page`	The `page` variable is equivalent to the `this` variable in the Java programming language.
`exception`	The `Throwable` object that was thrown by some other JSP page. This variable is only available in a JSP error page.

7.4 Using the page Directive

You use the page directive to modify the overall translation of the JSP page. For example, you can declare that the servlet code generated from a JSP page requires the use of the Date class:

<%@ page import=”java.util.Date” %>

You can have more than one page directive, but can only declare any given attribute once per page. This applies to all attributes except for the import attribute. You can place a page directive anywhere in the JSP file but it is a good practice to place the page directive at the beginning.

7.5 Including JSP Page Segments

There are two standard approaches to including presentation segments in your JSP pages:

● The include directive

● The jsp:include standard action

7.5.1 Using the include Directive

The include directive lets you include a segment in the text of the main JSP page at translation time. The syntax of this JSP technology directive is as follows:

<%@ include file=”segmentURL” %>

7.6 Using Standard Tags

The JSP specification provides standard tags for use within your JSP pages. Standard tags begin with the jsp: prefix. You use these tags to reduce or eliminate scriptlet code within your JSP page. However, the standard tags only provide limited functionality. Using the JSTL and EL reduces the need for the standard tags.

JavaBeans Components

A JavaBeans component is a Java technology class with at minimum the following features:

● Properties defined with accessors and mutators (get and set methods).

● A no-argument constructor.

● No public instance variables.

● The class implements the java.io.Serializable interface.

7.6.1 The useBean Tag

If you want to interact with a JavaBeans component using the standard tags in a JSP page, you must first declare the bean.

The syntax of the useBean tag is as follows:

```
<jsp:useBean id=”beanName”
scope=”page | request | session | application”
class=”className” />
```

7.6.2 The setProperty Tag

You use the setProperty tag to store data in the JavaBeans instance. The syntax of the setProperty tag is as follows:

```
<jsp:setProperty name="beanName" property_expression />
```

7.6.3 The getProperty Tag

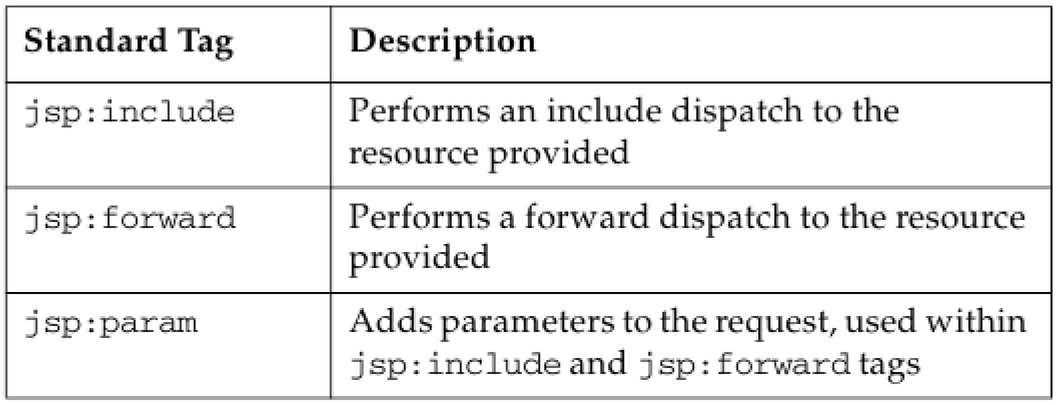
The getProperty tag is used to retrieve a property from a JavaBeans instance and display it in the output stream. The syntax of the getProperty tag is as follows:

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

7.7 Other Standard Tags

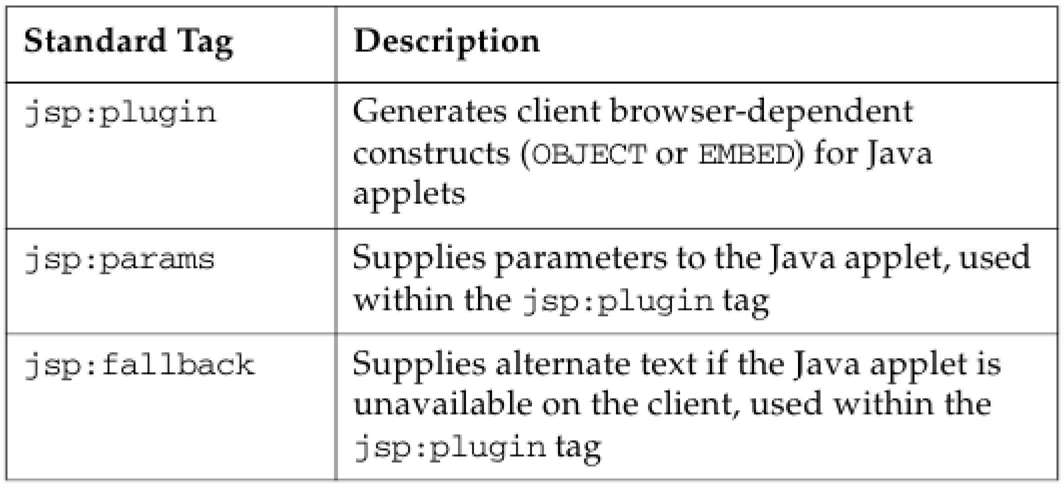
The following tables outline the standard tags defined in the JSP specification.

Include and Forward Standard Actions



Standard Tag	Description
`jsp:include`	Performs an include dispatch to the resource provided
`jsp:forward`	Performs a forward dispatch to the resource provided
`jsp:param`	Adds parameters to the request, used within `jsp:include` and `jsp:forward` tags

Standard Actions for Plug-ins



Standard Tag	Description
`jsp:plugin`	Generates client browser-dependent constructs (`OBJECT` or `EMBED`) for Java applets
`jsp:params`	Supplies parameters to the Java applet, used within the `jsp:plugin` tag
`jsp:fallback`	Supplies alternate text if the Java applet is unavailable on the client, used within the `jsp:plugin` tag

Chapter 8 Developing JSP Pages Using Custom Tags

8.1 Objectives

Upon completion of this module, you should be able to:

- Describe the Java EE job roles involved in web application development
- Design a web application using custom tags
- Use JSTL tags in a JSP page

8.2 The JSTL

This module looks in more detail at the tags that are available in the JSTL.

8.2.1 The Java EE Job Roles Involved in Web Application Development

One reason behind the development of tags, and the JSTL, was to support the separate job roles often found in organizations developing large web applications. These roles often include:

● **Web Designers** – Responsible for creating the views of the application, which are primarily composed of HTML pages

● **Web Component Developers** – Responsible for creating the control elements of the application, which are almost exclusively Java technology code

● **Business Component Developers** – Responsible for creating the model elements of the application, which might reside on the web server or on a remote server (such as an EJB technology server)

8.3 Designing JSP Pages With Custom Tag Libraries

A custom tag is an XML tag used in a JSP page to represent some dynamic action or the generation of content within the page during runtime. Custom tags are used to eliminate scripting elements in a JSP page.

Compared with coding these conditions and loops in Java scriptlets, JSTL tags offer these advantages:

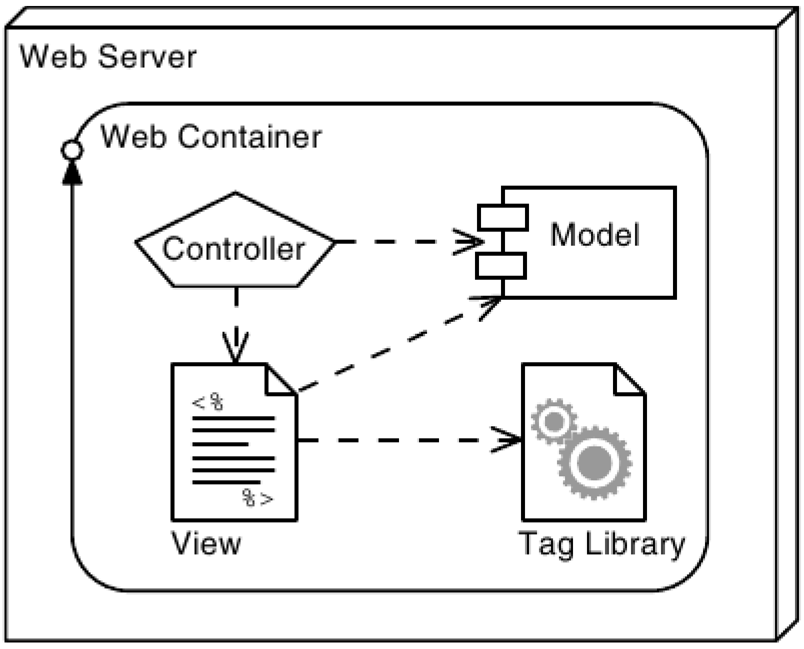
● Java technology code is removed from the JSP page.

● Custom tags are reusable components.

● Standard job roles are supported.

8.3.1 Custom Tag Library Overview

A custom tag library is a collection of custom tag handlers and the tag library descriptor file. A custom tag library is a web component that is part of the web application. Figure 8-1 shows a tag library as a web component.



8.3.2 Custom Tag Syntax Rules

JSP technology custom tags use XML syntax. There are four fundamental XML rules that all custom tags must follow:

- Standard tag syntax must conform to the following structure:

<prefix:name {attribute={”value”|’value’}}*>
body
</prefix:name>

- Empty tag syntax must conform to the following structure:

<prefix:name {attribute={”value”|’value’}}* />

- Tag names, attributes, and prefixes are case sensitive.

- Tags must follow nesting rules:

8.3.3 JSTL Sample Tags

The JSTL core library contains several tags that reduce the scripting necessary in a JSP page.

JSTL set Tag

JSTL url Tag

JSTL out Tag

more tags

8.4 Using a Custom Tag Library in JSP Pages

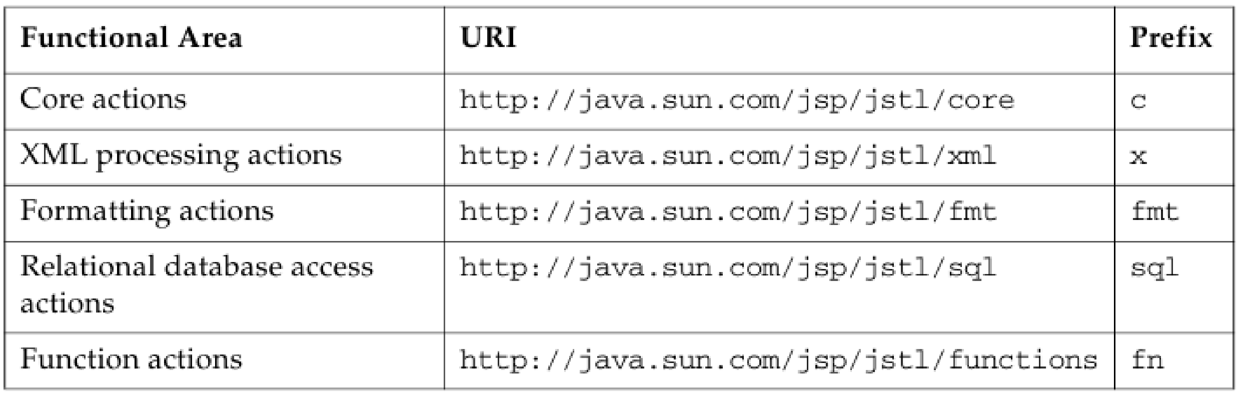
A custom tag library is made up of two parts: the JAR file of tag handler classes and the tag library descriptor (TLD). The TLD is an XML file that names and declares the structure of each custom tag in the library.

A JSP page can use a tag library by directing the JSP technology translator to access the TLD. This is specified using the JSP technology taglib directive. This directive includes the TLD URI and a custom tag prefix.

<%@ taglib prefix=”c” uri=”http://java.sun.com/jsp/jstl/core” %>

8.5 JSTL Tags

Lists the five functional categories of tags in JSTL, their URI values (to be used in JSP taglib directives), and the typical prefix used for each.



Functional Area	URI	Prefix
Core actions	`http://java.sun.com/jsp/jstl/core`	`c`
XML processing actions	`http://java.sun.com/jsp/jstl/xml`	`x`
Formatting actions	`http://java.sun.com/jsp/jstl/fmt`	`fmt`
Relational database access actions	`http://java.sun.com/jsp/jstl/sql`	`sql`
Function actions	`http://java.sun.com/jsp/jstl/functions`	`fn`

8.5.1 Core Actions

Tags in the core functional area include actions to output to the response stream, perform conditional operations, and perform iterations.

8.5.2 XML Processing

JSTL includes tags for processing XML documents.

8.5.3 Formatting Actions

JSTL includes tags for internationalization and formatting.

8.5.4 Relational Database Access

JSTL contains tags for database access.

8.5.5 Functions

JSTL contains tags for functions, many of them from java.lang.String.

Chapter 9 More Controller Facilities

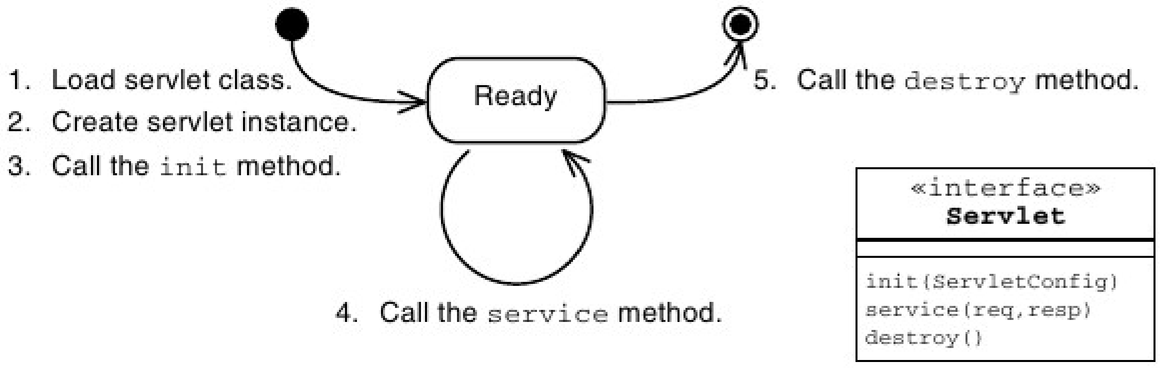
9.1 Objectives

Upon completion of this module, you should be able to:

- Understand the lifecycle of a servlet
- Understand the threading model of a servlet
- Write filters and apply them to groups of servlets or JSPs
- Handle multipart form data

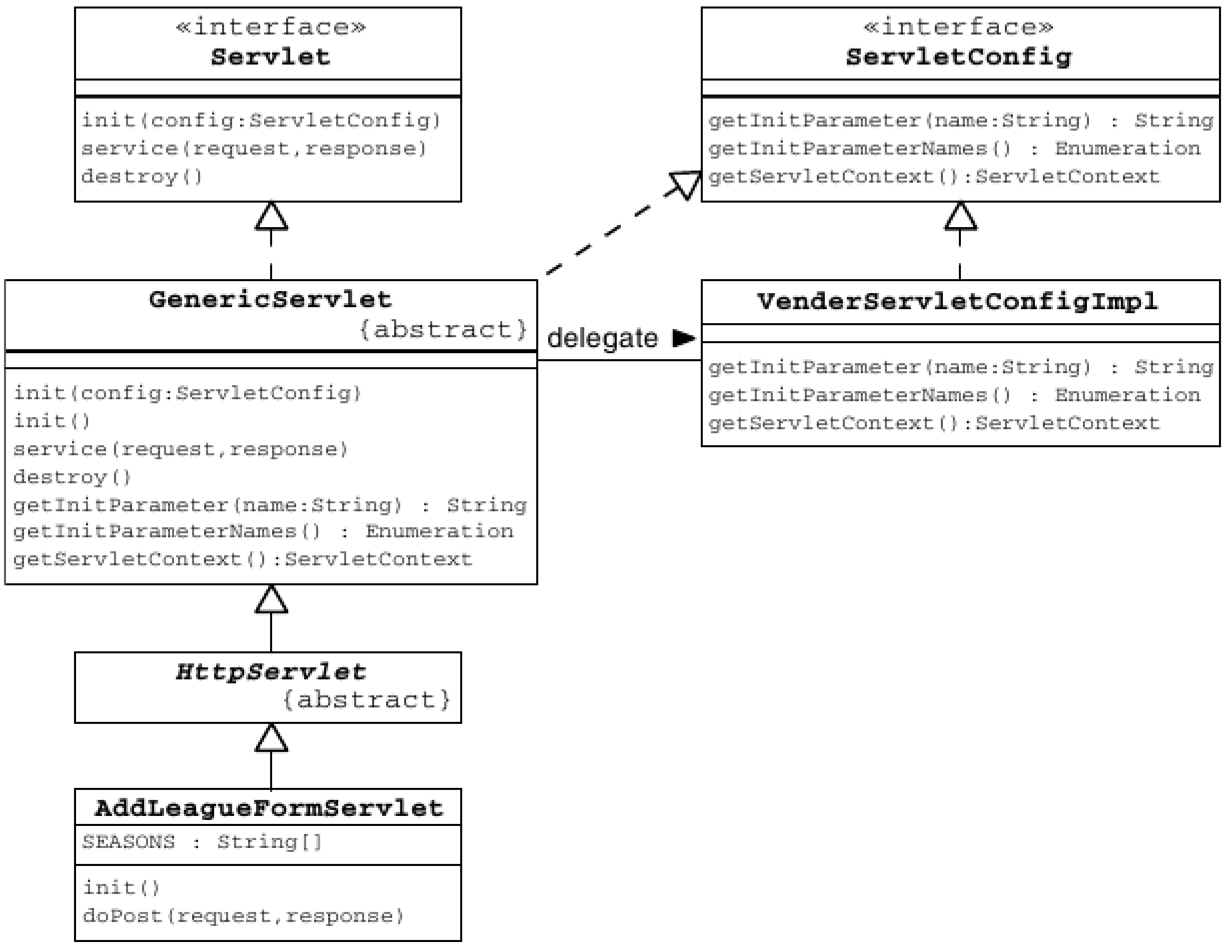
9.2 Servlet Life Cycle Overview

A graphical view of the servlet life cycle is provided by an annotated UML Statechart diagram.



9.2.1 The ServletConfig API

Every web container vendor must implement the ServletConfig interface. Instances of this class are passed into the init(ServletConfig) method defined in the Servlet interface.


«interface»
Servlet
init(config:ServletConfig)
service(request,response)
destroy()
«interface»
ServletConfig
getInitParameter(name:String) : String
getInitParameterNames() : Enumeration
getServletContext():ServletContext
GenericServlet
{abstract}
init(config:ServletConfig)
init()
service(request,response)
destroy()
getInitParameter(name:String) : String
getInitParameterNames() : Enumeration
getServletContext():ServletContext
delegate
VenderServletConfigImpl
getInitParameter(name:String) : String
getInitParameterNames() : Enumeration
getServletContext():ServletContext
HttpServlet
{abstract}
AddLeagueFormServlet
SEASONS : String[]
init()
doPost(request,response)

9.3 Servlet Lifecycle and Annotations

Java EE 5 introduced some key annotations for container managed objects. In the web container, these objects are Servlets, Filters, and many listeners.

A variety of annotations provide dependency injection for different resource types. Some of these are listed below:

- @EJB
- @Resource
- @PersistenceContext
- @PersistenceUnit
- @WebServiceRef

In addition, several plural versions of these annotations are defined so that multiple resources may be injected.

9.3.1 Lifecycle Method Annotations

The Java EE specification requires that objects that qualifies dependency injection must also be supported with two lifecycle annotations. These are @PostConstruct and @PreDestroy. These methods are very similar in function to the init() and destroy() methods of a servlet.

If an @PostConstruct method throws any exceptions the container must abandon the object. No methods may be called on the object and the object must not be put into service.

9.4 Servlets and Threading

Specifically, this means that if multiple clients invoke behavior that calls the same method, in the same servlet instance, at the same time, then that one servlet method (in a single servlet instance), might be concurrently executing many times under the control of many different threads—one thread for each client.

9.4.1 Handling Concurrency

9.4.2 Data Shared Between Invocations by a Single Client

Data that are shared between invocations by a single client are essentially session data. At first glance, this should simply involve using the HttpSession object to ensure that the lifetime and visibility of the data are appropriate. Unfortunately, however, it is possible for a single client to invoke servlets concurrently. This can arise if multiple browser windows or tabs are open.

9.4.3 Sharing Data Between Multiple Clients

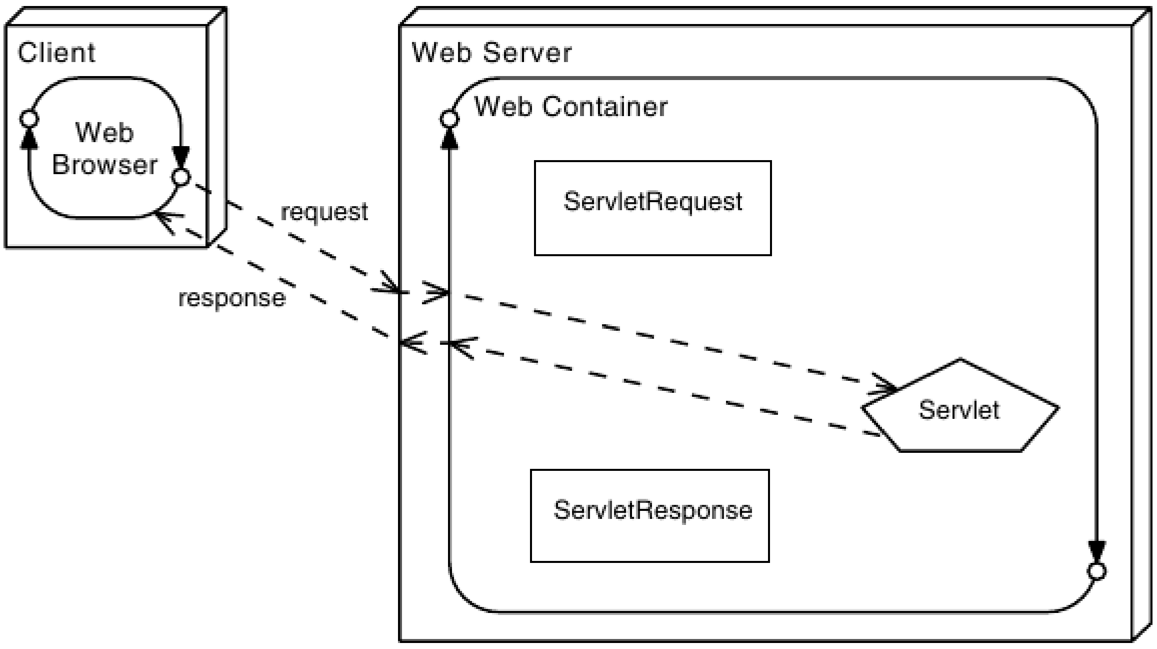
The sharing of data between clients can be achieved by storing those items in instance variables in the servlet object itself. However, if you are using older web containers prior to 2.4 of the servlet specification, be aware that many used to create multiple instances of each servlet class. If this happens, writing data to the instance variable in one servlet will not reliably result in all other threads seeing that data, because some threads will execute in other servlet instances. Instead, such data should be stored in the application context.

9.5 Web Container Request Cycle

One of the benefits to the servlet framework is that the web container intercepts incoming requests before they get to your code, preprocessing the request and additional functionality (such as security). The container can also post-process the response that your servlet generates.

9.5.1 Web Container Request Processing

Shows the creation of a request and a response for each request.



9.5.2 Applying Filters to an Incoming Request

Filters are components that you can write and configure to perform some additional pre-processing and post-processing tasks. When a request is received by the web container, several operations occur:

1. The web container performs its pre-processing of the request. What happens during this step is the responsibility of the container provider.

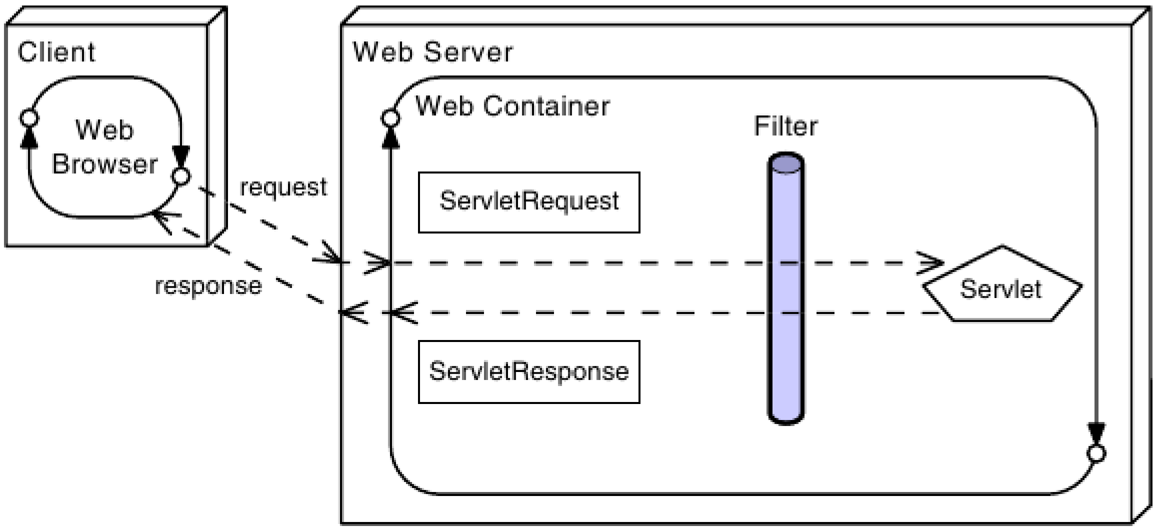
2. The web container checks if any filter has a URL pattern that matches the URL requested.

3. The web container locates the first filter with a matching URL pattern. The filter's code is executed.

4. If another filter has a matching URL pattern, its code is then executed. This continues until there are no more filters with matching URL patterns.

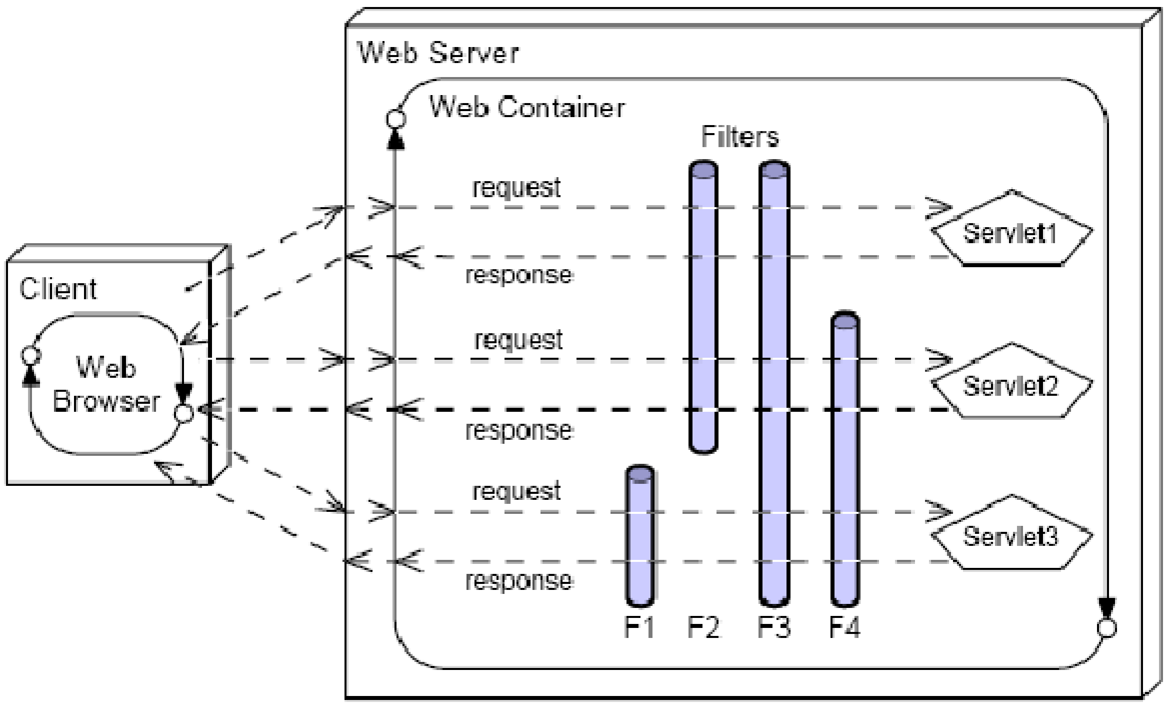
5. If no errors occur, the request passes to the target servlet. 6. The servlet passes the response back to its caller. The last filter that was applied to the request is the first filter applied to the response.

7. The first filter originally applied to the request passes the response to the web container. The web container might perform post-processing tasks on the response.



You should design filters to perform one task in the application, which helps ensure that they remain modular and reusable. You do not need to apply the same filters to every request.

Shows modular use of filters.

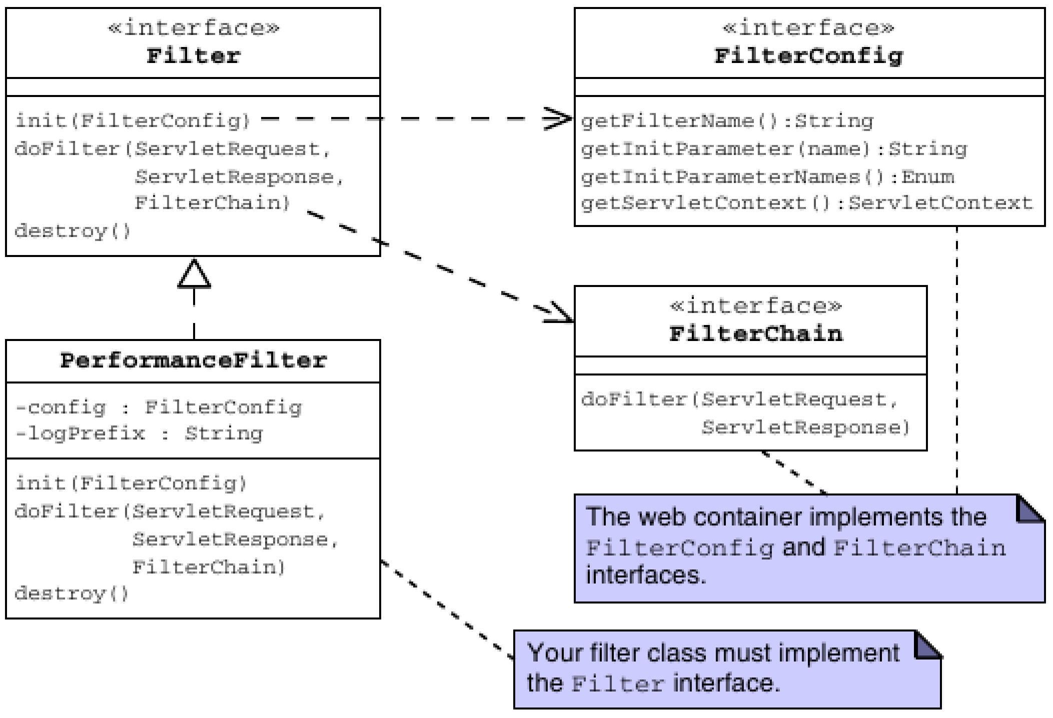


Filters can be used for operations such as:

● Blocking access to a resource based on user identity or role membership
● Auditing incoming requests
● Compressing the response data stream
● Transforming the response
● Measuring and logging servlet performance

9.6 Filter API

The Filter API is part of the base servlet API. The interfaces can be found in the javax.servlet package.


«interface»
Filter
init(FilterConfig)
doFilter(ServletRequest, ServletResponse, FilterChain)
destroy()
«interface»
FilterConfig
getFilterName():String
getInitParameter(name):String
getInitParameterNames():Enum
getServletContext():ServletContext
«interface»
FilterChain
doFilter(ServletRequest, ServletResponse)
PerformanceFilter
-config : FilterConfig
-logPrefix : String
init(FilterConfig)
doFilter(ServletRequest, ServletResponse, FilterChain)
destroy()
The web container implements the FilterConfig and FilterChain interfaces.
Your filter class must implement the Filter interface.

9.7 Developing a Filter Class

Sample filter code:

9.7.1 The init Method

The init method of a filter is called once when the container instantiates a filter. This method is passed a FilterConfig, which is typically stored as a member variable for later use .

9.7.2 The doFilter Method

The doFilter method is called for every request intercepted by the filter. It is the filter equivalent of a servlet's service method. This method is passed three arguments:ServletRequest,ServletResponse, and FilterChain.

9.7.3 The destroy Method

Before the web container removes a filter instance from service, the destroy method is called. You use this method to perform any operations that need to occur at the end of the filter's life.

9.8 Configuring the Filter

Because the web container is responsible for the life cycles of filters, filters must be configured in the web application's deployment descriptor.

9.8.1 Configuring a Filter Using Annotations

Java EE 6 provides an annotation that allows declaration and mapping of a filter. The annotation is @WebFilter, and this allows specification of the filter name, the url patterns to which it responds, and the types of invocation for which it should be invoked.

9.8.2 Declaring a Filter in the web.xml File

For older Java EE versions, of for version independence, you can define and map filters using the web.xml deployment descriptor file.

Shows a filter declared within filter elements in the deployment descriptor.

```
<filter>
        <filter-name>perfFilter</filter-name>
        <filter-class>sl314.web.PerformanceFilter</filter-class>
        <init-param>
                <param-name>Log Entry Prefix</param-name>
                <param-value>Performance: </param-value>
         </init-param>
</filter>
```

9.8.3 Declaring a Filter Mapping in the web.xml File

Filters are executed when the resource to which they are mapped is requested.

```
<filter-mapping>
        <filter-name>perfFilter</filter-name>
        <url-pattern>*.do</url-pattern>
</filter-mapping>
```

9.9 Handling Multipart Forms

Java EE 6 introduced a mechanism to simplify handling of multipart form data. Three key elements make this up, which are: additional methods in the javax.servlet.http.HttpServletRequest object, an annotation

javax.servlet.annotation.MultipartConfig, and an interface

javax.servlet.http.Part.

Chapter 10 More Options for the Model

10.1 Objectives

Upon completion of this module, you should be able to:

- Understand the nature of the model as a macro-pattern
- Implement persistent storage for web applications using JDBC or Java Persistence API

10.2 The Model as a Macro-Pattern

In the MVC architecture pattern, the model component embodies a great deal of functionality. Here are some of the responsibilities that fall to the model as it has been discussed so far:

- Present an interface to the controller
- Implement the business logic
- Present a JavaBeans compliant view of the data to the view component, in a form that is convenient for the view to use.

10.3 Database and Resource Access

Virtually all practical web applications will require access to a database or other external support system. Good OO requires that the elements of the model that handle this should be separated from the elements of the model that represent the domain objects and the elements that perform business operations on those objects.

10.3.1 Data Access Object (DAO) Pattern

The DAO pattern eases maintenance of applications that use databases by separating the business logic from the data access (data storage) logic. The data access implementation (perhaps JDBC technology calls) is encapsulated in DAO classes. The DAO pattern permits the business logic and the data access logic to change independently, increasing the flexibility of your application. For example, if the database schema changes, you only need to change the DAO methods, and not the business services or the domain objects.

10.3.2 JDBC API

The JDBC API is the Java technology API for interacting with a database management system (DBMS). The JDBC API includes interfaces that manage connections to the DBMS, statements to perform operations, and result sets that encapsulate the result of retrieval operations.

10.4 Developing a Web Application Using a Database

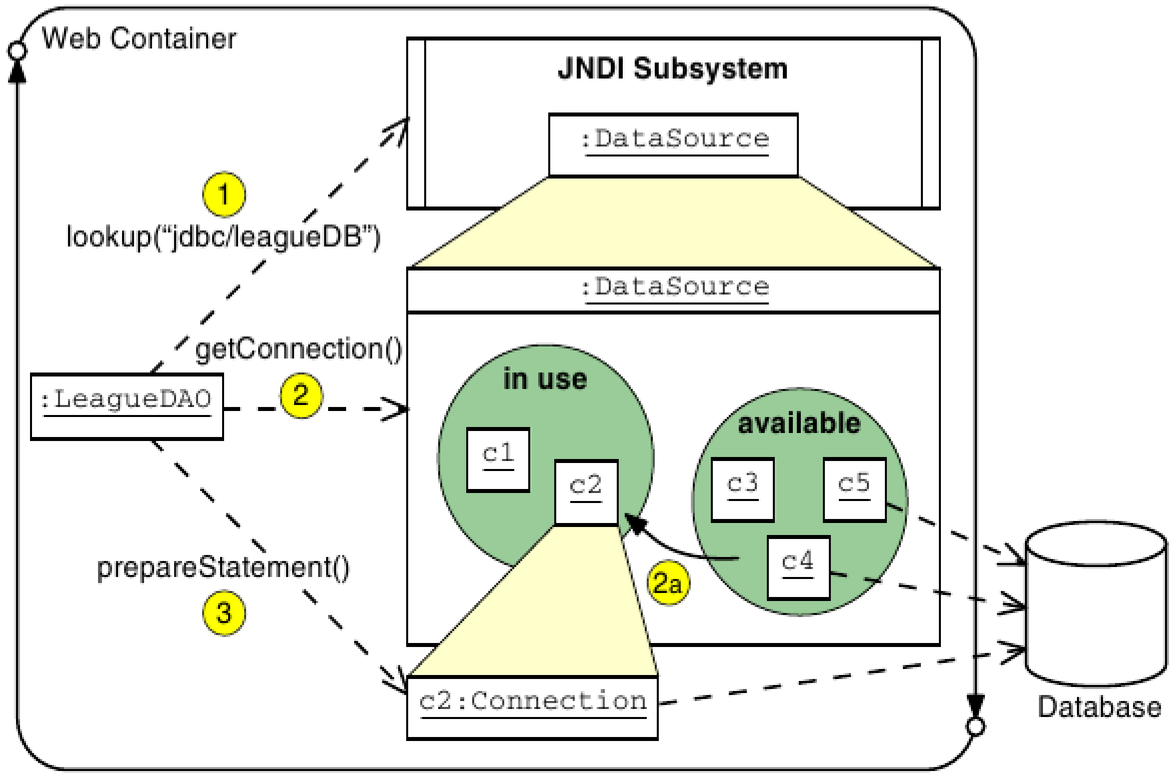
10.4.1 Traditional Approaches to Database Connections

In Java technology, the most basic mechanism for obtaining a database connection is to use the static method java.sql.DriverManager.getConnection. Obtaining an exclusive connection is expensive, and the total number of concurrent connections is usually limited. Because connections are needed in many places in a Java EE application, using this approach would create performance problems in a real application.

10.4.2 Using a DataSource and the Java Naming and Directory Interface API

A DataSource is a resource that contains the information needed to connect to an underlying database (typically the database URL, driver, user name, and password). All Java EE application servers are required to provide a naming service into which resources such as DataSource can be stored. The naming service can be queried using the Java Naming and Directory InterfaceTM (JNDI) APIs. Behind the scenes, the server will maintain a pool of connections. You can use the DataSource to retrieve a connection from this pool.

Shows how DataSource is used in an application.



10.4.3 Configuring a DataSource and the JNDI API

The DataSource reference must be created in the application server, and published by name in the JNDI lookup service. In principle, three things must be configured:

● The database and the tables/schema for the application

● A connection pool

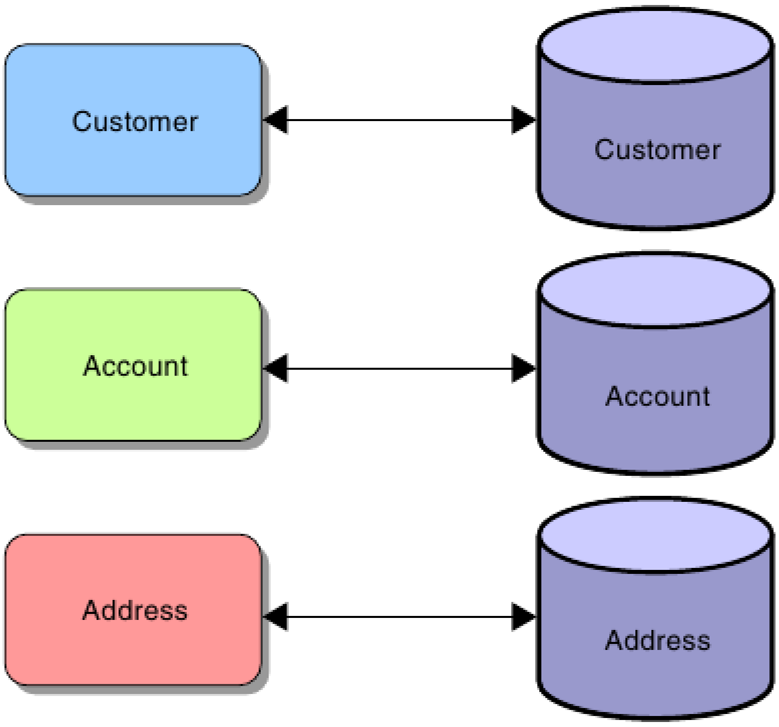
● An entry in the JNDI service that references the connection pool

The method of achieving these steps varies between application servers, and the lab exercise for this module will guide you through that process for the GlassFishTM/NetBeans combination.

10.5 Object Relational Mapping Software

Writing code to translate from an object-oriented domain scheme to a relational database scheme can be a time consuming endeavor. Object Relational Mapping (ORM) software attempts to provide this mapping to OO software developers without requiring much or any coding. Often just configuration information in the form of code annotations or XML configuration files are supplied to ORM software. Examples of existing ORM software are EclipseLink and Hibernate.

One-to-One Mapping of Java Objects to Relational Tables



10.6 Java Persistence API Structure

The Java Persistence API splits the problem of persistence into a number of key elements, each of which requires configuration if the system is to work. These elements are:

- An entity class, defined by the programmer, that presents the object- oriented view of the business entity.
- A mapping between the fields of the entity class and the schema of the persistent storage. This can be provided using annotations on fields within the entity class along with default mapping rules, and may be modified where necessary by additional annotations.
- Mapping that specifies the persistence implementation (recall that Java Persistence API is simply a specification), the JNDI DataSource, and other configuration elements of the implementation. This is provided using a file called persistence.xml.
- A JNDI lookup service entry that maps the published name of the DataSource to an actual DataSource. This is provided in a container-specific manner as discussed earlier.
- A DataSource that is connected to a connection pool. This has also been discussed earlier, and is also provided in a container-specific way.
- A connection pool that is connected to the database. As before, this is container specific and was introduced earlier.
- The actual database, along with the necessary tables and schema to support the application. Again, this is container specific.

10.7 Entity Class Requirements

To define an entity class, you are required to perform the following tasks:

● Declare the entity class.

● Verify and override the default mapping.

10.7.1 Declaring the Entity Class

The following steps outline a process you can use to declare an entity class.

1. Collect information required to declare the entity class. This step involves identifying the application domain object (identified in the object oriented analysis and design phase of application development) that you want to persist.

● Use the domain object name as the class name.

● Use the domain object field names and data types as the field names and types of entity classes.

2. Declare a public Java technology class. The class must not be final, and no methods of the entity class can be final. The class can be concrete or abstract. The class can not be an inner class.

3. If an entity instance is to be passed by value as a detached object through a remote interface, then ensure the entity class implements the Serializable interface.

4. Annotate the class with the javax.persistence.Entity annotation.

5. Declare the attributes of the entity class. Attributes must not have public visibility. They can have private, protected, or package visibility. Clients must not attempt to access an entity classes' attributes directly.

6. You can optionally declare a set of public getter and setter methods for every attribute declared in the previous step.

7. Annotate the primary key field or the getter method that corresponds to the primary key column with the Id annotation.

8. Declare a public or protected no-arg constructor that takes no parameters. The container uses this constructor to create instances of the entity class. The class can have additional constructors.

10.8 Life Cycle and Operational Characteristics of Entity Components

Besides entity classes, there are several key concepts that must be understood to begin leveraging the Java Persistence API. They are:

● Persistence Unit

● Entity manager

● Persistence context

● Persistent identity

10.8.1 Persistence Units

A persistence unit is a collection of entity classes stored in a EJB-JAR, WAR, or JAR archive along with a persistence.xml file. The key concept to understand at this point is that a persistence unit defines what entity classes are controlled by an entity manager. A persistence unit is limited to a single DataSource.

10.8.2 The persistence.xml file

Configuration of a persistent unit is controlled by an XML configuration file named persistence.xml which does the following:

● Configures which classes make up a persistence unit

● Defines the base of a persistence unit by its location

● Specifies the DataSource used

10.8.3 The EntityManager

An entity manager provides methods to control events of a persistence context and the life cycle of entity instances in a persistence context. An entity manager has the following characteristics:

● Provides operations, such as flush(), find(), and createQuery(), to control a persistence context

● Replaces some of the functionality of home interfaces in EJB 2.1 entity beans An entity manager can be obtained using dependency injection in managed classes.

@PersistenceContext private EntityManager em;

10.8.4 Entity Instance Management

When a reference to an entity manager has been obtained, you use its methods to marshall entity instance data into and out of the database.

EntityManger Injection

EntityManager Methods

10.8.5 User Transactions

A UserTransaction object can be used to demarcate the boundaries of a transaction in your code. The UserTransaction object is created by injection as was the EntityManager. It can be created as a field in a class like this:

import javax.transaction.UserTransaction;

[...]

// Declaring a field in the class:

@Resource UserTransaction utx;

Then the UserTransaction methods begin, commit, rollback, and setRollbackOnly can be used to provide control over the transaction and its scope.

10.9 Java Persistence API Example

Chapter 11 Asynchronous Servlets and Clients

11.1 Objectives

Upon completion of this module, you should be able to:

- Use the Asynchronous Servlet mechanism
- Use JavaScript to send an HTTP request from a client
- Process an HTTP response entirely in JavaScript
- Combine these techniques to create the effect of server-push

11.2 Asynchronous Servlets

Java EE 6 adds the option to perform request processing asynchronously. If this is done, then the servlet execution thread can be freed up to service requests from other clients, and the generation of the response may be performed in another thread, for example the thread that creates the trigger condition that allows the response to be prepared.

11.2.1 Separating Request Receipt from Response Generation

To allow for separation of request from response generation, the servlet API provides a class called AsyncContext. A servlet that wishes to hand off response generation to another thread can obtain an AsyncContext object and pass this to another thread, perhaps using a queue. The AsyncContext object provides access to the original HttpServletRequest and HttpServletResponse objects. This allows processing of the request and generation of the response. Additionally, the AsyncContext allows for a forwarding mechanism. This does not use the

RequestDispatcher, is achieved using one of serveral dispatch methods

in the AsyncContext itself.

11.2.2 Asynchronous Servlet Example

11.3 Asynchronous Listeners

The AsyncListener interface must be implemented by the listener that will be notified when these situations arise. The interface defines four listener methods:

● onComplete(AsyncEvent)

● onError(AsyncEvent)

● onTimeout(AsyncEvent)

● onStartAsync(AsyncEvent)

11.4 Asynchronous JavaScript Clients

In these Asyncrhonous JavaScript approaches, JavaScript code on the client is used to modify elements of the existing page based on a combination of the user's interactions, local client-side processing, and information retrieved from the server. Remember that in a normal HTTP request, an entire page is fetched, but in this approach, JavaScript code fetches a small amount of data which is then integrated into the existing page. Usually the server's response is really very small, for example, it might be just a fragment of HTML .

Simple Asynchronous Client Example

11.5 Combining Asynchronous Servlets With Asynchronous JavaScript

Although each of these techniques serves a purpose on its own, the two can be used together to create something akin to a server-push environment.

Consider a page containing JavaScript code that makes an asynchronous request for update from the server, and a server-side implementation that uses the asynchronous servlet techniques. Now, if the server does not choose to respond for several minutes, the user is not inconvenienced, as the body of the page is operating normally. Similarly, because of the use of asynchronous servlet execution, the resource load on the server is minimized. Consequently, the response may be sent at a time convenient to the server, and the effect of server-push is achieved in an architecturally manageable way.

Chapter 12 Implementing Security

12.1 Objectives

Upon completion of this module, you should be able to:

- Describe a common failure mode in security
- Require that a user log in before accessing specific pages in your web application
- Describe the Java EE security model
- Require SSL encrypted communication for certain URLs or servlets

12.2 Security Considerations

Every application that is accessible over the web must consider security. Your site must be protected from attack, the private data of your site’s users must be kept confidential, and your site must also protect the browsers and computers used to access your site.

12.3 Confusion of Code and Data

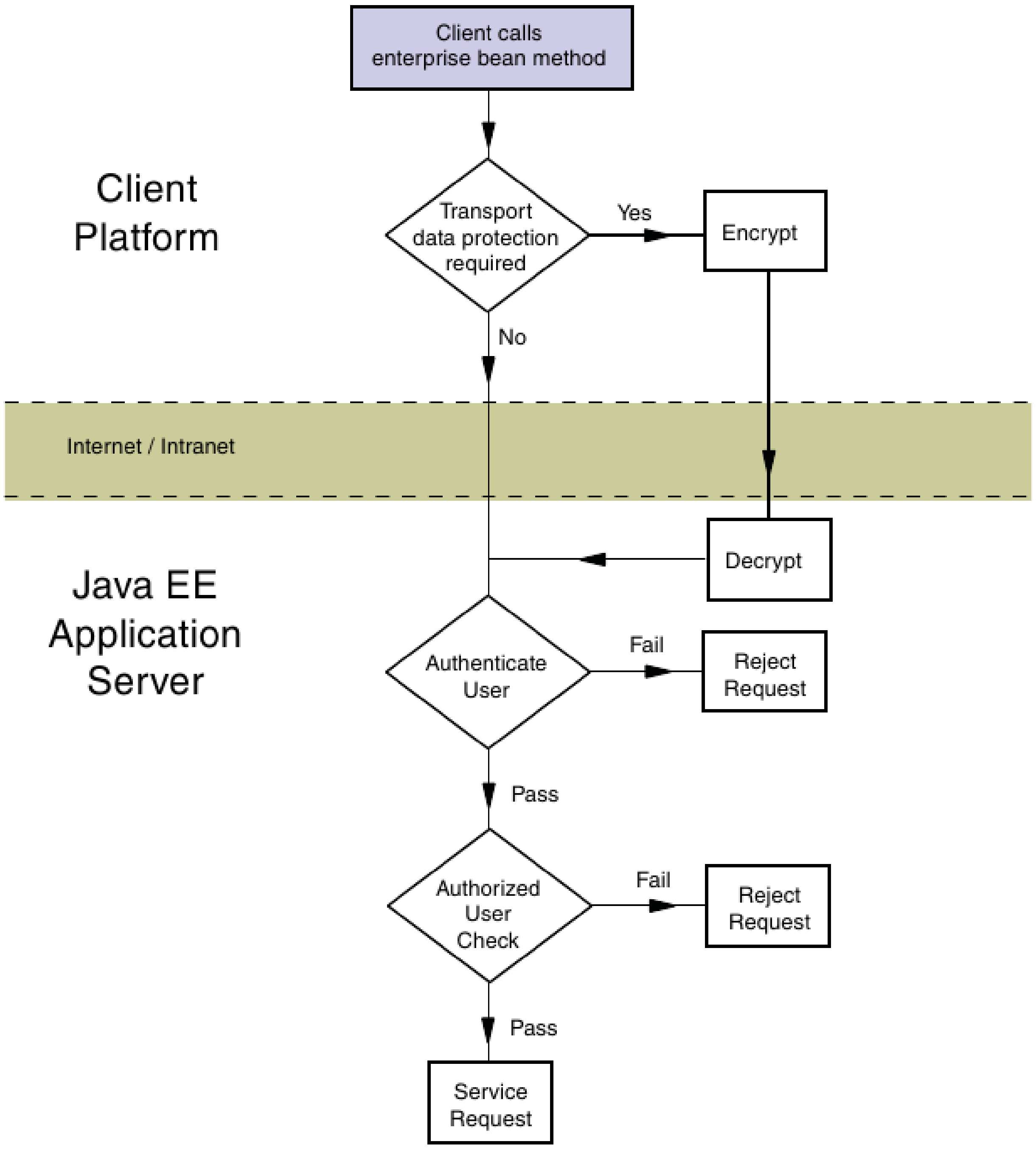
A very substantial proportion of attacks against computer systems are variations on the seemingly simple idea of tricking the system into executing as code something that was passed into the system in the guise of data.

Consider that data do not control the operation of the machine, and further that the machine must allow data to be provided from outside— from the user. You can see that systems do not need to prevent data from entering the system. However, if the system’s willingness to allow data in can be tricked, it might be possible to provide code in the guise of data. This will bypass the security restrictions of the system and might be catastrophic.

12.3.1 Preventing Code as Data Attacks

12.4 Authentication and Authorization

The application usually needs to be able to identify the user, decide what operations the user is allowed to perform, and maintain the confidentiality and the integrity of the data that is in transit.



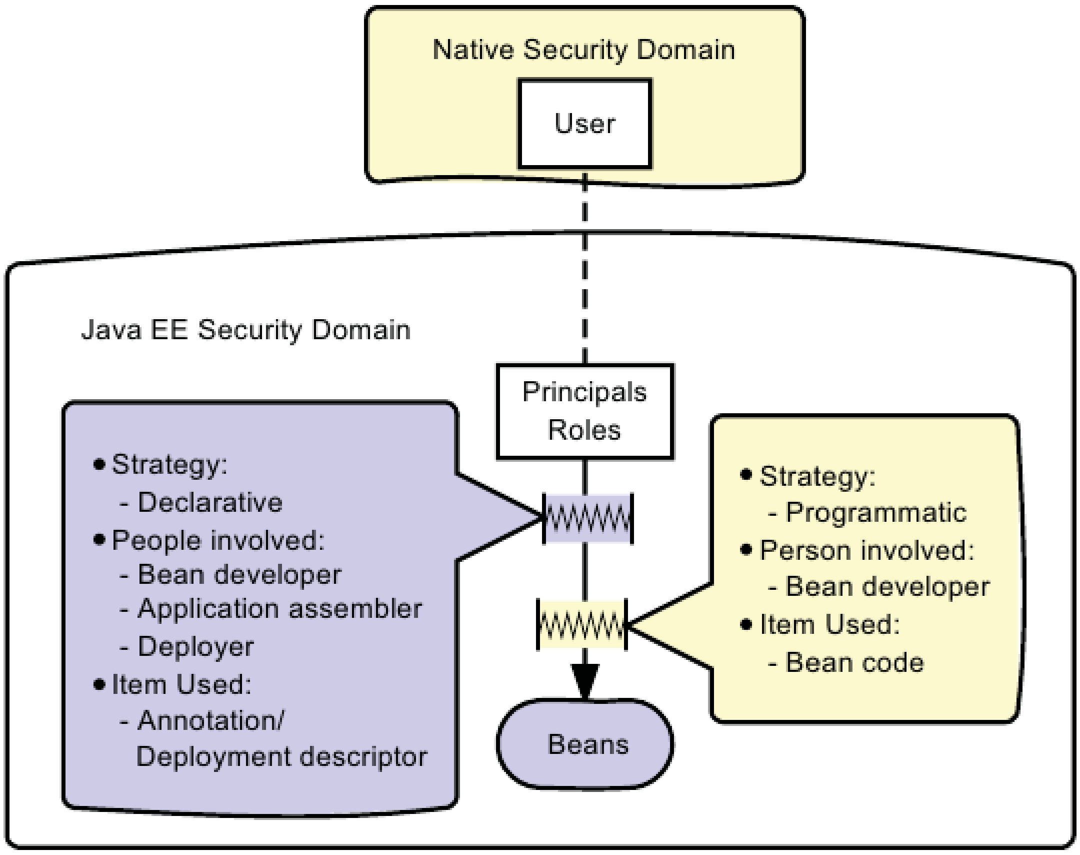
12.5 Authenticating the Caller

Caller authentication is the process of verifying what the user’s identity is, and consists of the following two steps.

1. Determine the identity claimed by the user
2. Verify that the user is who they claim to be (Authenticate the user)

12.6 Examining the Java EE Authorization Strategies

The primary purpose of the Java EE security model is to control access to business services and resources in the application. The Java EE security model provides two complementary strategies for access control: programmatic access control and declarative access control. Both strategies assume that the user has been authenticated by the application server, and the roles of which the user is a member can therefore be determined by the web container.



12.7 Using Declarative Authorization

Declarative authorization can be configured by the programmer, assembler, or deployer. Usually the programmer sets up some initial security properties when developing or testing the web application or its components. The assembler and deployer can then modify these properties according to the needs of the application as a whole. The declarative security policy can be defined in the servlets using annotations or the DD. The use of vendor-supplied packaging and deployment tools can be helpful here, because they can provide a basic, graphical representation of the security policy. Declarative authorization for web applications involves the following tasks:

- Collection of user credentials into a credentials database
- Declaration of roles
- Mapping users to roles
- Specification of role requirements for access to URLs

The following sections address these requirements.

12.7.1 Creating a Credentials Database

12.7.2 Declaring Security Roles

12.7.3 Mapping Users to Roles

12.7.4 Declaring Permission Requirements

12.8 Using Programmatic Authorization

Programmatic authorization is the responsibility of the bean developer. The following methods in the HttpServletRequest support programmatic authorization:

- boolean isUserInRole(String role)
- Principal getUserPrincipal()

Programmatic authorization is more expressive than the declarative approach, but is more cumbersome to maintain, and because of the additional complexity, more error prone. In particular, declarative authorization controls access at a role level, while programmatic authorization can selectively permit or block access to principals belonging to a role.

12.8.1 Using the isUserInRole Method

A servlet can use the isCallerInRole method to verify the role of the caller. The isUserInRole method allows recognition of roles without identifying the specific principal.

12.9 Enforcing Encrypted Transport

Provided the server has been configured with a public key certificate, you can require that communication between client and server be encrypted. If you allow some pages to be browsed over an unencrypted connection while others are encrypted, you might end up exposing cookies, particularly those that identify a session, in such a way that allows an attacker to pretend to be the user that logged in properly earlier.

12.9.1 Using an Annotation to Mandate Encrypted Transport

An annotation can be used to require encrypted transport. If used, the annotation applies to the servlet, rather than to a collection of URLs. The format of the annotation is:

```
@ServletSecurity(@HttpConstraint(transportGuarantee = TransportGuarantee.CONFIDENTIAL))
```