



Sun Educational Services

Web Component Development With Java™ Technology

SL-314



Copyright 2001 Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, California 94303, U.S.A. All rights reserved.

This product or document is protected by copyright and distributed under licenses restricting its use, copying, distribution, and decompilation. No part of this product or document may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any.

Third-party software, including font technology, is copyrighted and licensed from Sun suppliers.

Sun, Sun Microsystems, the Sun Logo, the Duke Logo, EJB, Enterprise JavaBeans, JavaBeans, Java virtual machine, Java 2 Platform, Enterprise Edition, Java 2 Platform, Standard Edition, JDBC, J2EE, J2SE, JSP, JVM, and Solaris are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

Netscape Navigator is a trademark or registered trademark of Netscape Communications Corporation.

INTERACTIVE UNIX is a registered trademark of INTERACTIVE Systems Corporation. UNIX is a registered trademark in the United States and other countries, exclusively licensed through X/Open Company, Ltd.

RESTRICTED RIGHTS: Use, duplication, or disclosure by the U.S. Government is subject to restrictions of FAR 52.227-14(g)(2)(6/87) and FAR 52.227-19(6/87), or DFAR 252.227-7015 (b)(6/95) and DFAR 227.7202-3(a).

DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS, AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

Copyright 2001 Sun Microsystems Inc., 901 San Antonio Road, Palo Alto, California 94303, Etats-Unis. Tous droits réservés.

Ce produit ou document est protégé par un copyright et distribué avec des licences qui en restreignent l'utilisation, la copie, la distribution, et la décompilation. Aucune partie de ce produit ou document ne peut être reproduite sous aucune forme, par quelque moyen que ce soit, sans l'autorisation préalable et écrite de Sun et de ses bailleurs de licence, s'il y en a.

Le logiciel détenu par des tiers, et qui comprend la technologie relative aux polices de caractères, est protégé par un copyright et licencié par des fournisseurs de Sun.

Sun, Sun Microsystems, le logo Sun, le logo Duke, EJB, Enterprise JavaBeans, JavaBeans, Java virtual machine, Java 2 Platform, Enterprise Edition, Java 2 Platform, Standard Edition, JDBC, J2EE, J2SE, JSP, JVM, et Solaris sont des marques de fabrique ou des marques déposées de Sun Microsystems, Inc. aux Etats-Unis et dans d'autres pays.

Netscape est une marque de Netscape Communications Corporation aux Etats-Unis et dans d'autres pays. in the United States and other countries.

INTERACTIVE est une marque de INTERACTIVE Systems Corporation. UNIX est une marque enregistrée aux Etats-Unis et dans d'autres pays et licenciée exclusivement par X/Open Company Ltd.

LA DOCUMENTATION EST FOURNIE "EN L'ETAT" ET TOUTES AUTRES CONDITIONS, DECLARATIONS ET GARANTIES EXPRESSES OU TACITES SONT FORMELLEMENT EXCLUES, DANS LA MESURE AUTORISEE PAR LA LOI APPLICABLE, Y COMPRIS NOTAMMENT TOUTE GARANTIE IMPLICITE RELATIVE A LA QUALITE MARCHANDE, A L'APTITUDE A UNE UTILISATION PARTICULIERE OU A L'ABSENCE DE CONTREFAÇON.



About This Course



Course Goal

Upon completion of this course, you should be able to design, develop, and deploy a Web application using Java™ servlet and JavaServer™ Pages (JSP)™ technologies.



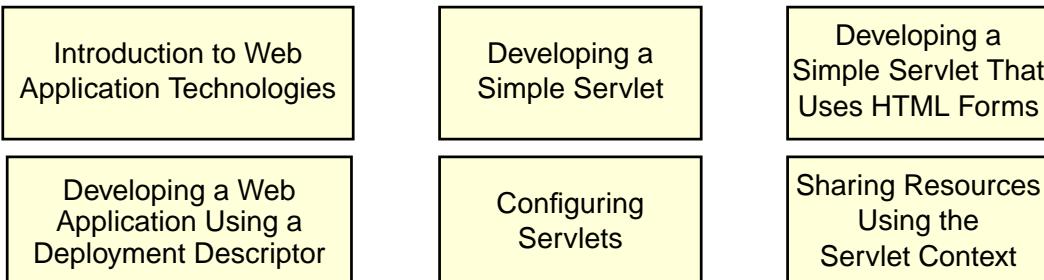
Learning Objectives

- Develop a Web application using Java servlets
- Develop a robust Web application using the Model-View-Controller pattern, session management, exception handling, declarative security, and proper concurrency controls
- Develop a Web application using JavaServer Pages
- Develop a custom tag library
- Develop a Web application that integrates with an n-tiered architecture using either a RDBMS or an EJB™ server

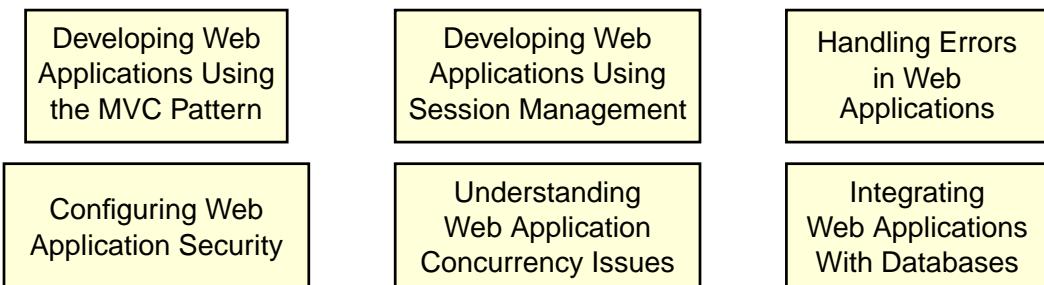


Module-by-Module Overview

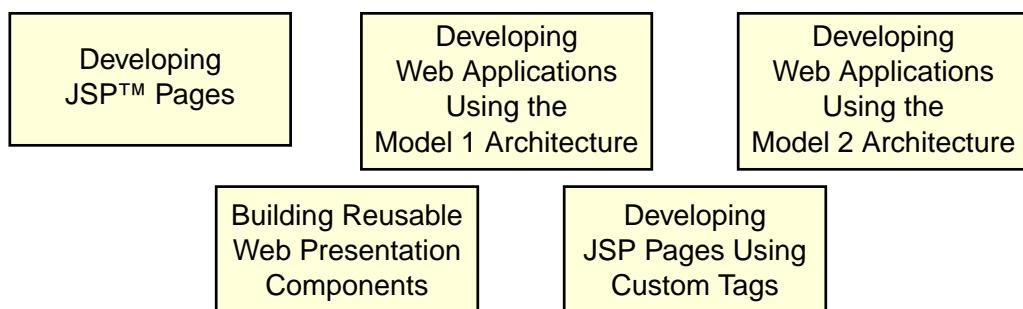
Design and Development of N-tier Web Applications



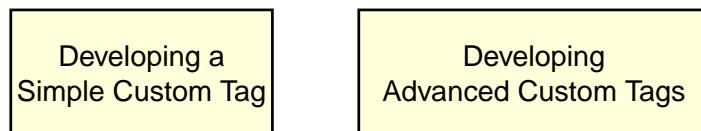
Java Servlet Application Strategies



JSP™ Application Strategies



Developing Custom JSP™ Tag Libraries



Java 2 Platform, Enterprise Edition





Topics Not Covered

- Java technology programming – Covered in SL-275: *The Java™ Programming Language*
- Object-oriented design and analysis – Covered in OO-226: *Object-Oriented Analysis and Design for Java™ Technology (UML)*
- Java 2 Platform, Enterprise Edition – Covered in SEM-SL-345: *Java™ 2 Platform, Enterprise Edition: Technology Overview Seminar*
- Enterprise JavaBeans – Covered in SL-351: *Enterprise JavaBeans™ Programming*



How Prepared Are You?

To be sure you are prepared to take this course, can you answer yes to the following questions?

- Can you create Java technology applications?
- Can you read and use a Java technology API?
- Can you analyze and design a software system using a modeling language like UML?
- Can you develop applications using a component/container framework?



How To Learn From This Course

- Ask questions
- Participate in the discussions and exercises
- Read the code examples
- Use the on-line documentation for the J2SE™, servlet, and JSP API
- Read the servlet and JSP specifications



Introductions

- Name
- Company affiliation
- Title, function, and job responsibility
- Experience developing applications with the Java programming language
- Experience with HTML and Web development
- Experience with Java servlets or JavaServer Pages
- Reasons for enrolling in this course
- Expectations for this course



Icons



Additional resources



Demonstration



Discussion



Note



Caution



Typographical Conventions

- *Courier* is used for the names of commands, files, directories, programming code, programming constructs, and on-screen computer output.
- ***Courier bold*** is used for characters and numbers that you type, and for each line of programming code that is referenced in a textual description.
- *Courier italic* is used for variables and command-line placeholders that are replaced with a real name or value.
- ***Courier italic bold*** is used to represent variables whose values are to be entered by the student as part of an activity.



Typographical Conventions

- *Palatino italic* is used for book titles, new words or terms, or words that are emphasized.



Additional Conventions

Java programming language examples use the following additional conventions:

- Courier is used for the class names, methods, and keywords.
- Methods are not followed by parentheses unless a formal or actual parameter list is shown.
- Line breaks occur where there are separations, conjunctions, or white space in the code.
- If a command on the Solaris™ Operating Environment is different from the Microsoft Windows platform, both commands are shown.



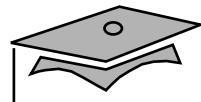
Module 1

Introduction to Web Application Technologies



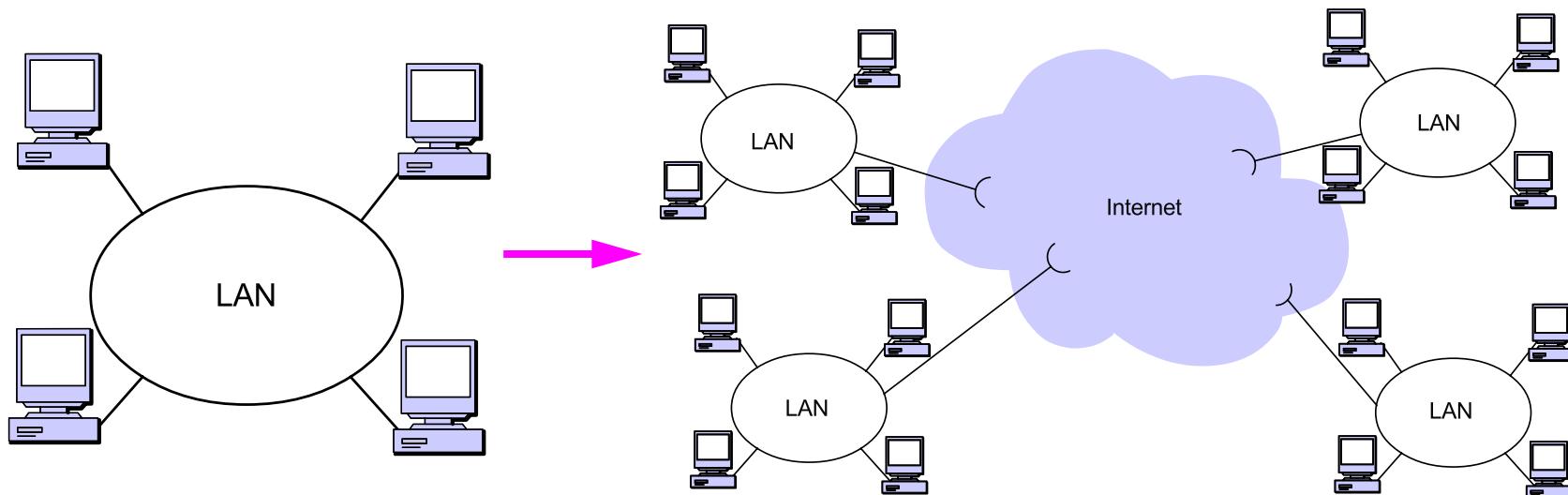
Objectives

- Describe Internet services
- Describe the World Wide Web
- Distinguish between Web applications and Web sites
- Describe Java servlet technology and list three benefits of this technology compared to traditional CGI scripting
- Describe JavaServer Pages technology and list three benefits of JSP technology over rival template page technologies
- Describe the JavaTM 2 Platform, Enterprise Edition (J2EETM)



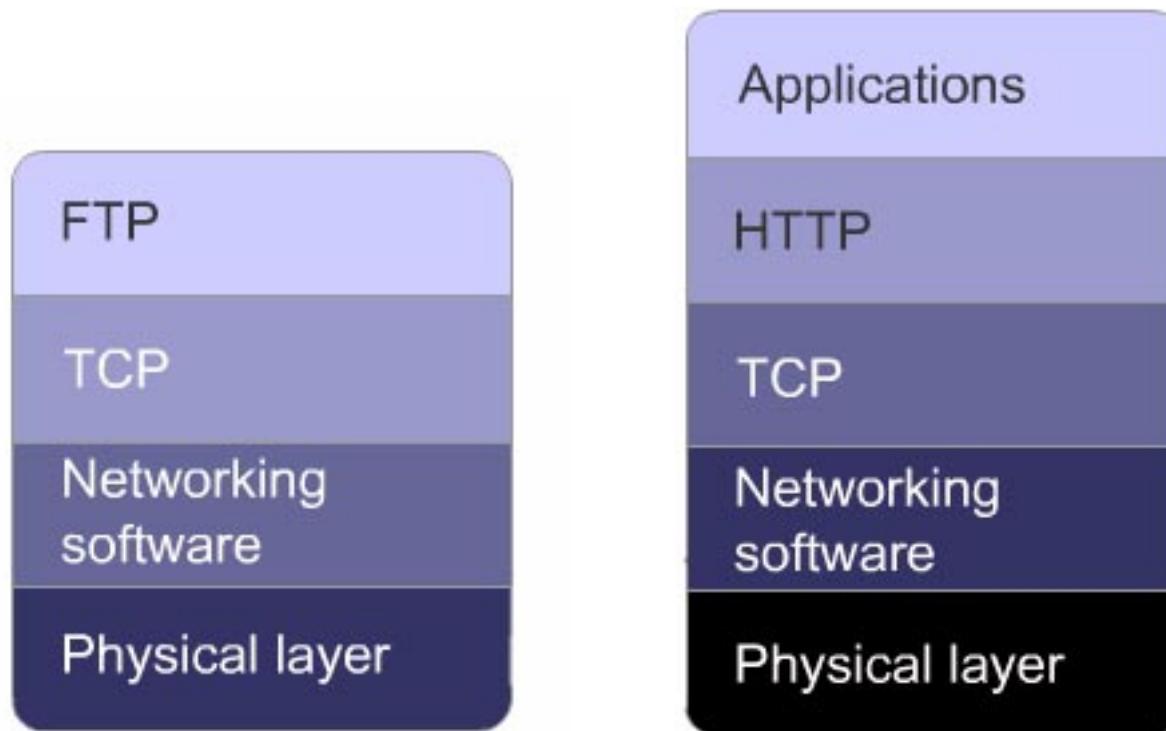
The Internet Is a Network of Networks

Evolution of the Internet:





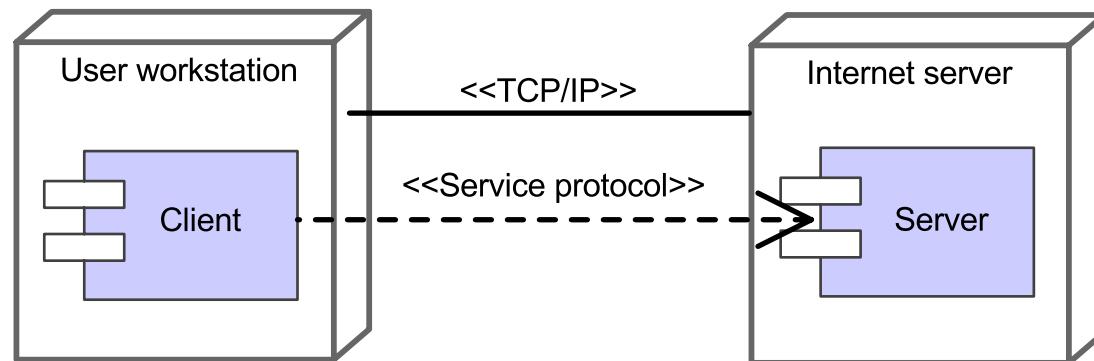
Networking Protocol Stack





Client-Server Architecture

Deployment diagram of a generic, client-server Internet architecture:





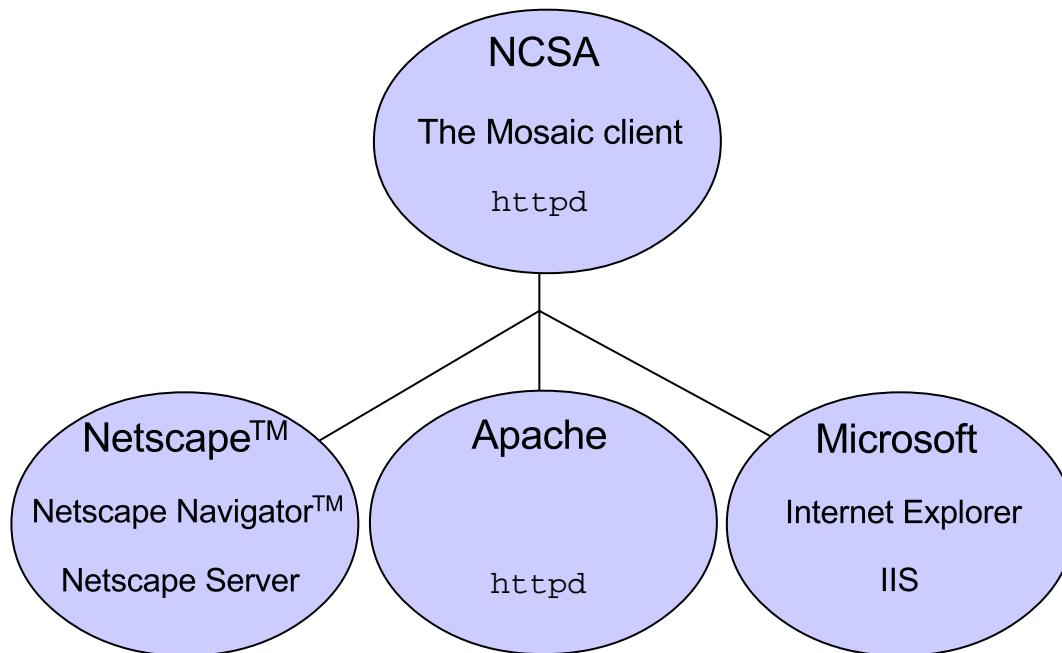
Hypertext Transfer Protocol

- The Hypertext Transfer Protocol (HTTP) supports serving up documents in the Hypertext Markup Language (HTML):
 - HTML documents include links to other Web documents.
 - Web documents may also include forms to pass data from the user to the Web server.
- HTTP can serve any type of document.
- The Multipurpose Internet Mail Extensions (MIME) specification defines a canonical naming convention for documents of various media.



Web Browsers and Web Servers

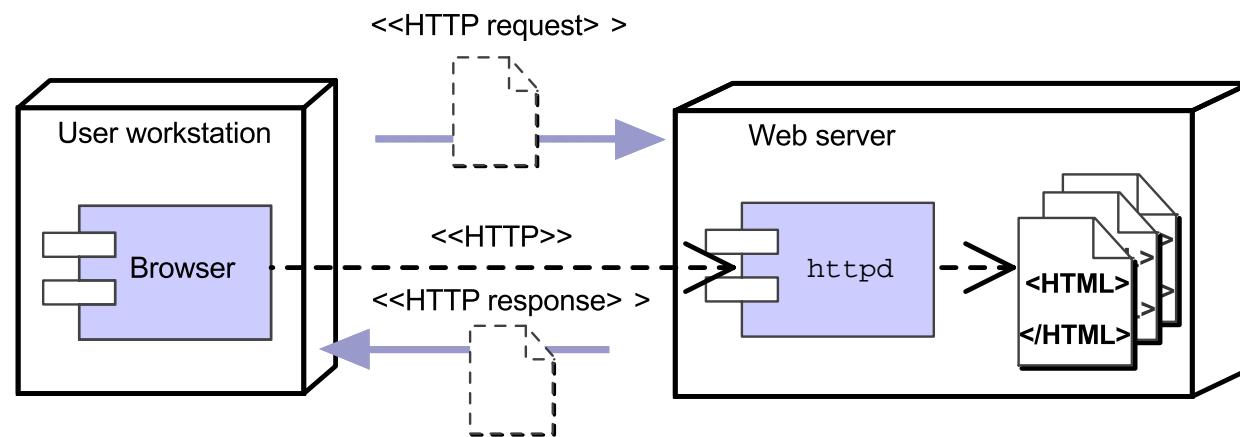
- Two key types of programs:
 - Web browsers
 - Web servers
- Development of browsers and servers:





HTTP Client-Server Architecture

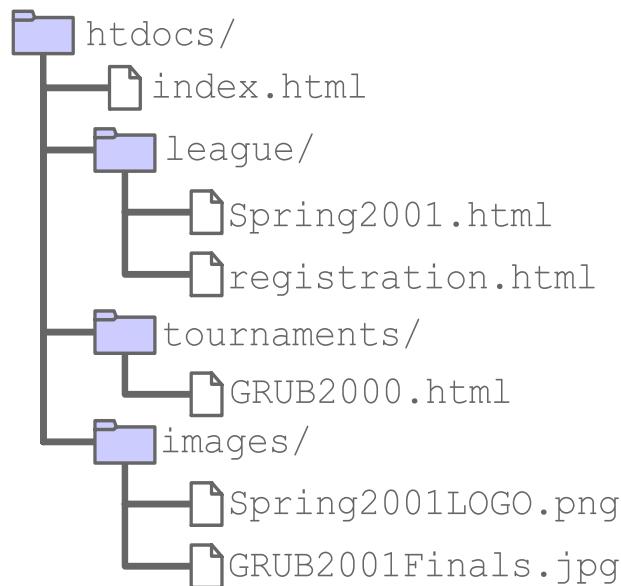
Deployment diagram of a Web server:





The Structure of a Web Site

A Web site is a hierarchy of HTML documents, media files, and the directories that form the structure:



Example of an HTTP URL:

<http://www.soccer.org/league/Spring2001.html>



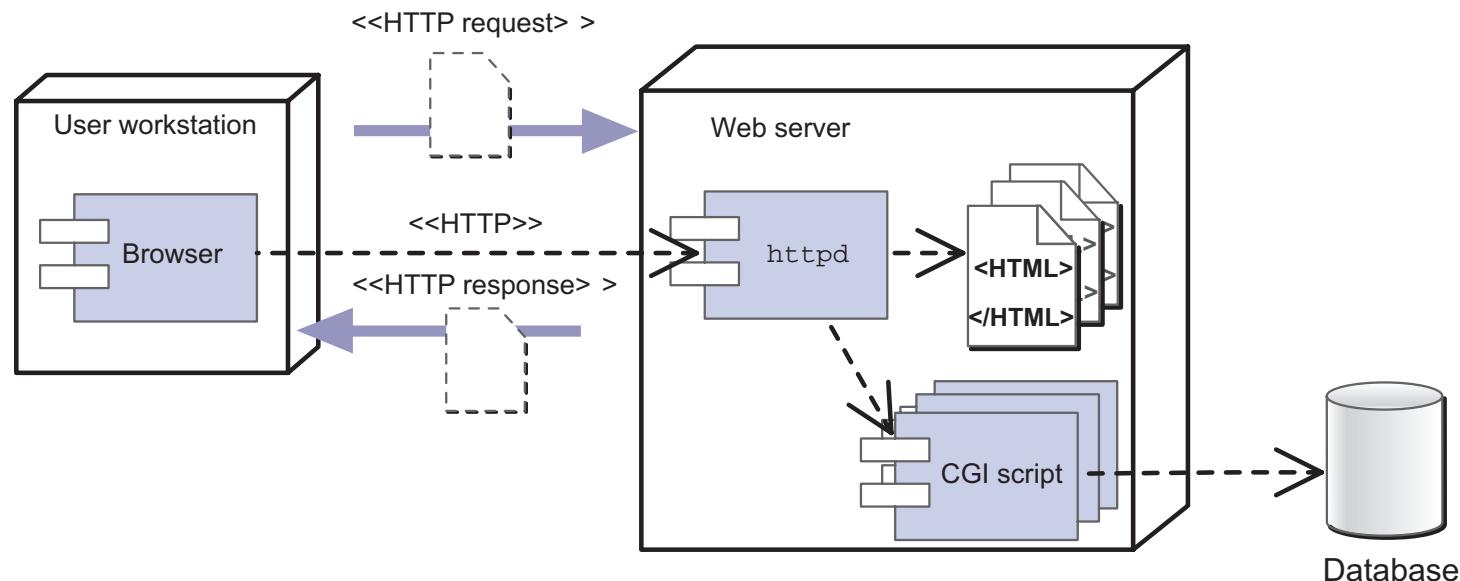
Web Applications

- A Web site is a collection of static HTML pages.
- A Web application is a Web site with dynamic functionality on the server (or sometimes on the client using applets or other interactive elements).
- Web applications use HTML forms as the user interface to code that is running on the server:
 - Data is passed from the HTML form to the server using the Common Gateway Interface (CGI).
 - The CGI data is sent in the HTTP request stream.



CGI Programs on the Web Server

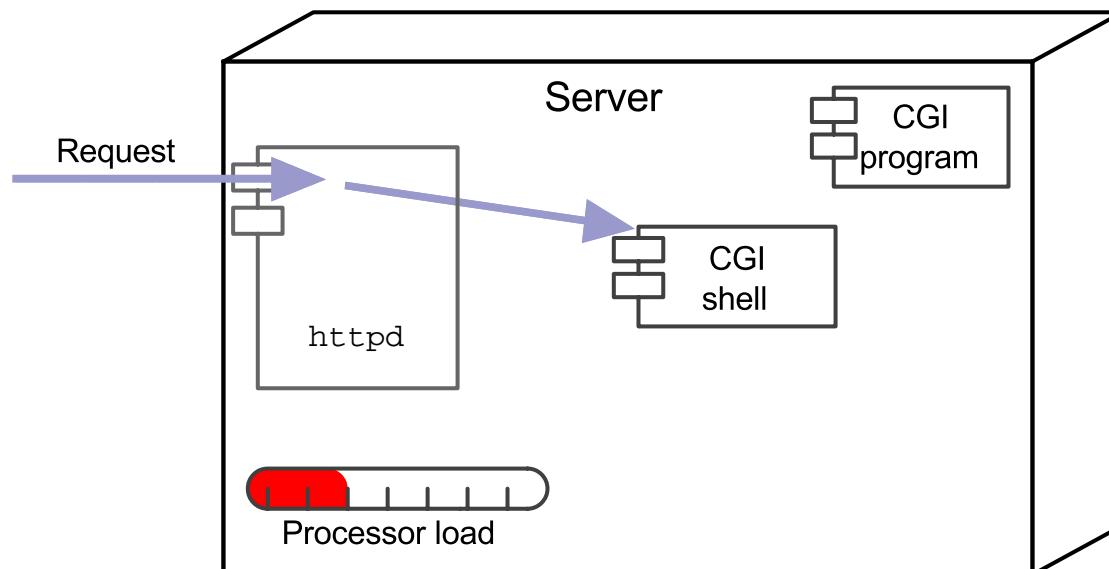
Deployment diagram of a Web server with CGI programs:





Execution of CGI Programs

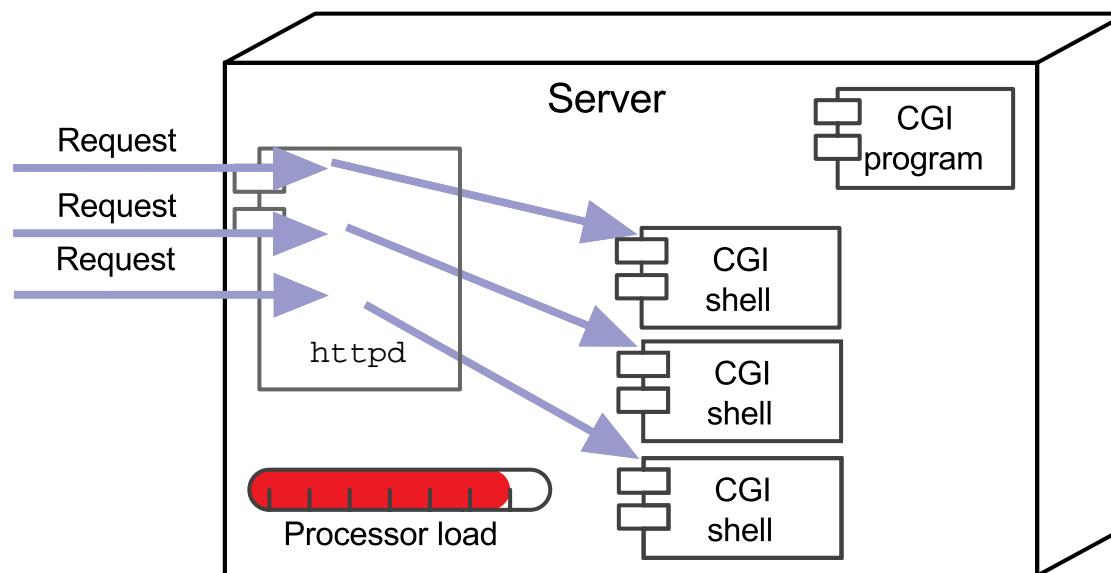
How CGI works with one request:





Execution of CGI Programs

How CGI works with many requests:





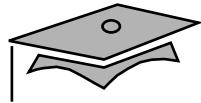
Advantages and Disadvantages of CGI Programs

- CGI program advantages:
 - Written in a variety of languages
 - Relatively easy for a Web designer to implement
- CGI program disadvantages:
 - Each shell is heavyweight.
 - Not scalable.
 - CGI processing code (business logic) is mingled with HTML (presentation logic).
 - Language is not always robust or object oriented.
 - Language is not always platform independent.



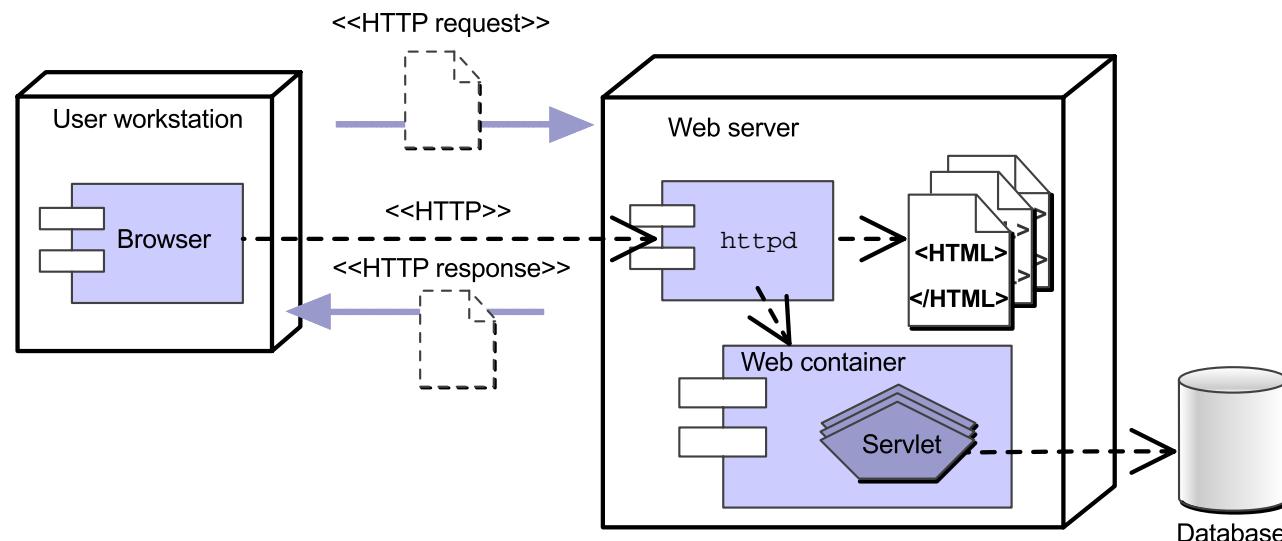
Java Servlets

- A servlet is a Java technology component that executes on the server.
- Servlets perform tasks similar to those performed by CGI programs, but servlets execute in a different environment.
- Servlets perform the following:
 - Process the HTTP request
 - Generate the HTTP response dynamically
- A Web container is a special JVM™ that is responsible for maintaining the life cycle of the servlets as well as issuing threads for each request.



Servlets on the Web Server

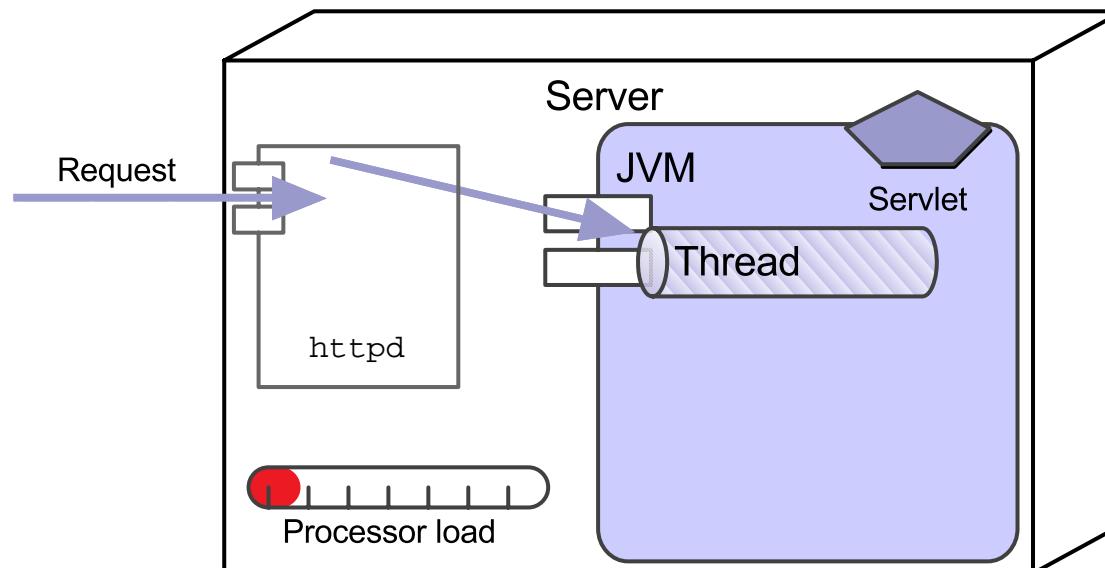
Deployment diagram of a Web server with a Web container:





Execution of Java Servlets

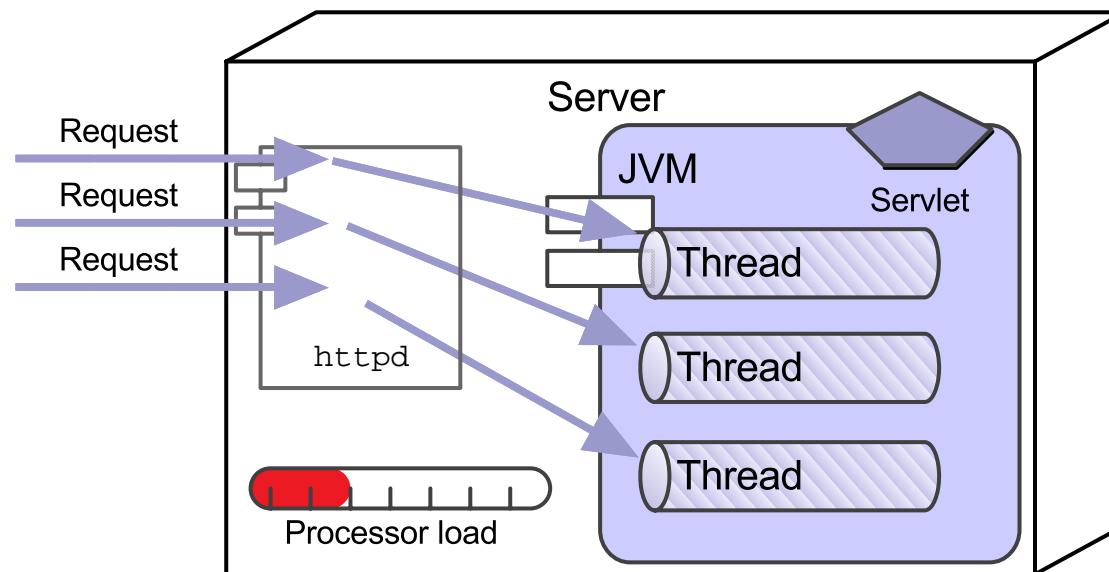
How servlets work with one request:





Execution of Java Servlets

How servlets work with many requests:





Advantages and Disadvantages of Java Servlets

- Servlet advantages:
 - Performance (threads are faster than processes).
 - Scalable.
 - The Java programming language is robust and object oriented.
 - The Java programming language is platform independent.
- The main disadvantage of servlets is that processing code (business logic) is mingled with HTML generation (presentation logic).



Template Pages

- Template pages look like static HTML pages, but with embedded code to perform dynamic generation of data and HTML.
- Example:

```
<TABLE BORDER='1' CELLSPACING='0' CELLPADDING='5'>
<TR><TH>number</TH><TH>squared</TH></TR>
<% for ( int i=0; i<10; i++ ) { %>
<TR><TD><%= i %></TD><TD><%= (i * i) %></TD></TR>
<% } %>
</TABLE>
```

Table of numbers squared:

| number | squared |
|--------|---------|
| 0 | 0 |
| 1 | 1 |
| 2 | 4 |
| 3 | 9 |
| 4 | 16 |
| 5 | 25 |
| 6 | 36 |
| 7 | 49 |
| 8 | 64 |
| 9 | 81 |



Other Template Page Technologies

- PHP (PHP Hypertext Preprocessor)

```
<? for ( $i=0; $i<10; $i++ ) { ?>
<TR><TD><? echo $i ?></TD><TD><? echo ($i * $i) ?></TD></TR>
<? } ?>
```

- ASP (Active Server Pages)

```
<% FOR I = 0 TO 10 %>
<TR><TD><%= I %></TD><TD><%= (I * I) %></TD></TR>
<% NEXT %>
```

- JSP (JavaServer Pages)

```
<% for ( int i=0; i<10; i++ ) { %>
<TR><TD><%= i %></TD><TD><%= (i * i) %></TD></TR>
<% } %>
```



JavaServer Pages Technology

- JSP pages are translated into Java servlet classes, compiled, and are treated just like servlets in the Web container.
- Therefore, unlike ASP and PHP, JSP pages are compiled rather than interpreted.
- If designed well, JSP pages focus on the presentation logic, not on the business logic.
- In a Java technology Web application, JSP pages are often used in conjunction with servlets and business objects in a Model-View-Controller pattern.



Advantages and Disadvantages of JavaServer Pages

- Advantages of JSP technology:

JSP technology has all of the advantages of servlet technology: high performance, high scalability, platform independent, and can use the Java language as its scripting language.

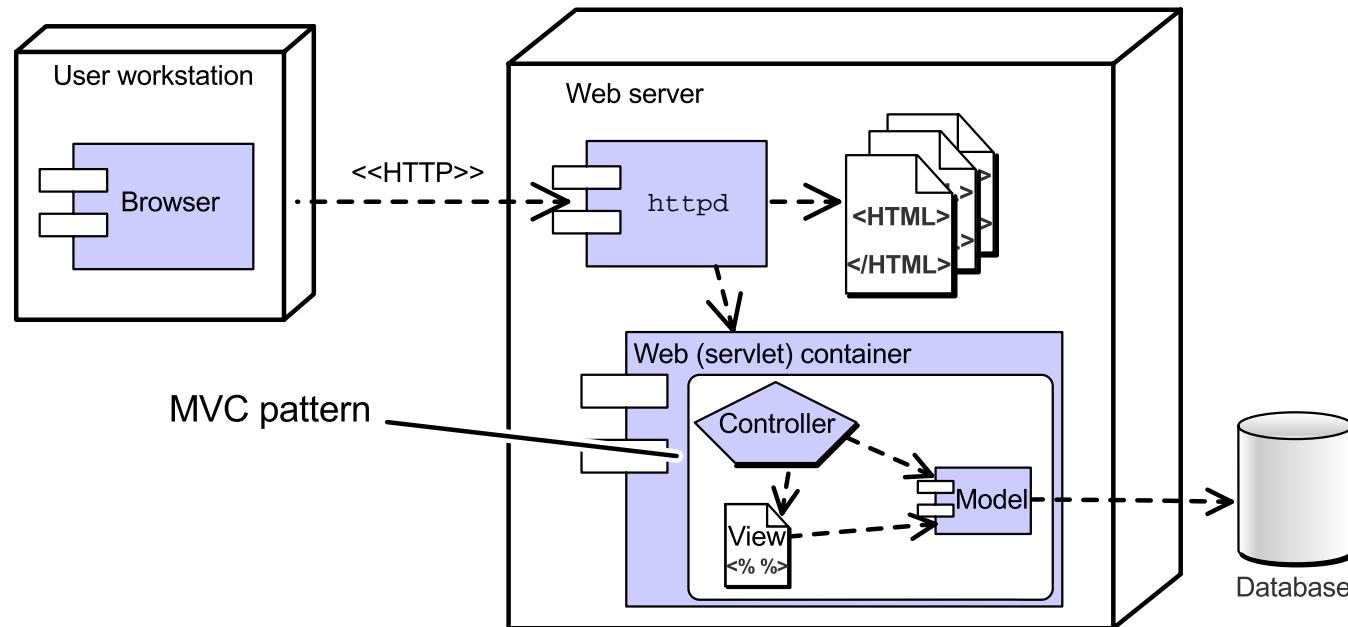
- Disadvantages of JSP technology:

If JSP pages are used in isolation, then the scripting code which performs business and control logic can become cumbersome in the JSP pages.



The Model 2 Architecture

Deployment diagram of a Web container using Model 2 architecture:



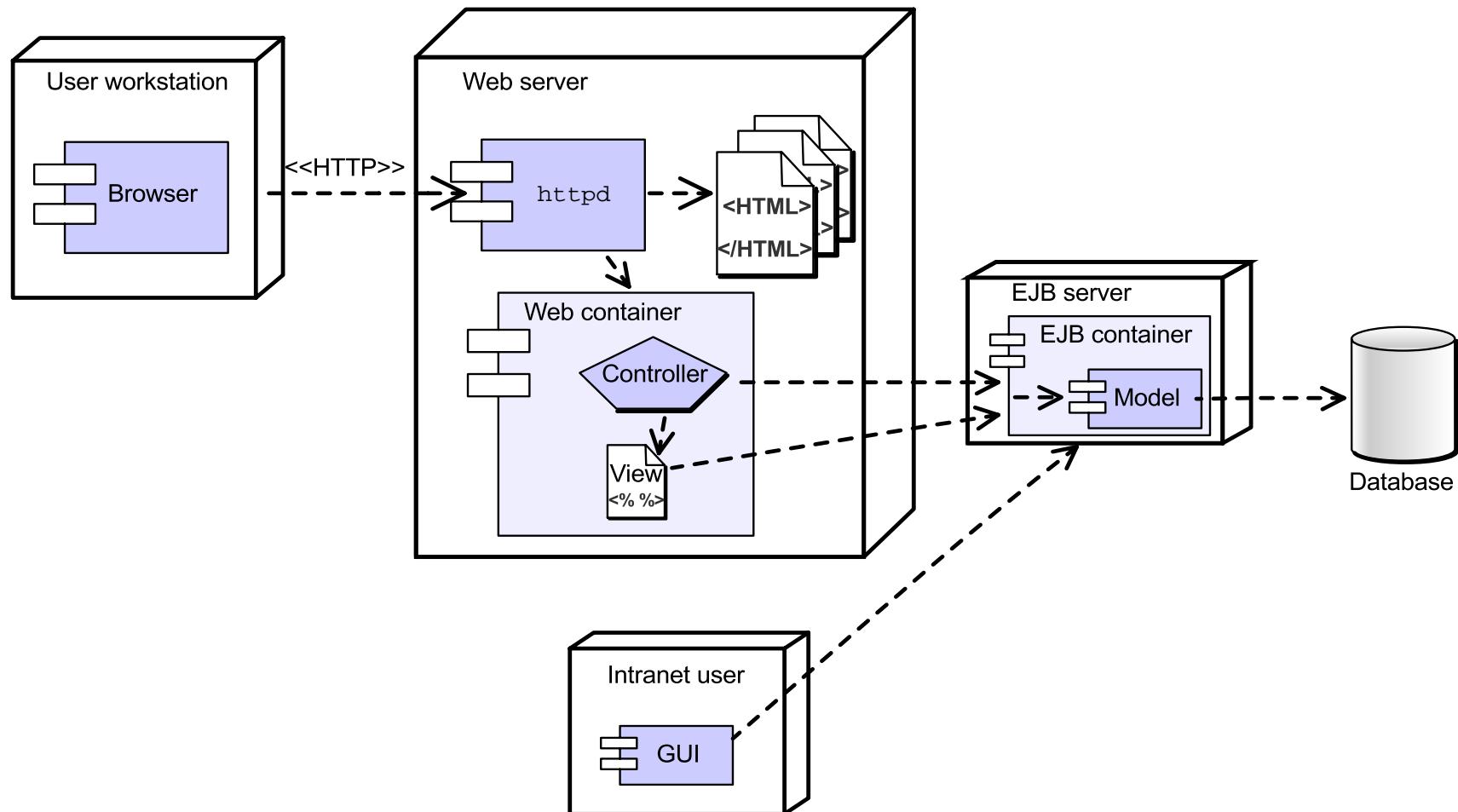


The J2EE Platform

- The J2EE specification recommends an n-tier architecture for Web applications to enhance:
 - Scalability
 - Extensibility
 - Separation of different types of logic
- Modular design allows for easier modification of the business logic.
- Enterprise components can use container-provided services like security, transaction, and persistence management.



An Example of J2EE Architecture





Job Roles

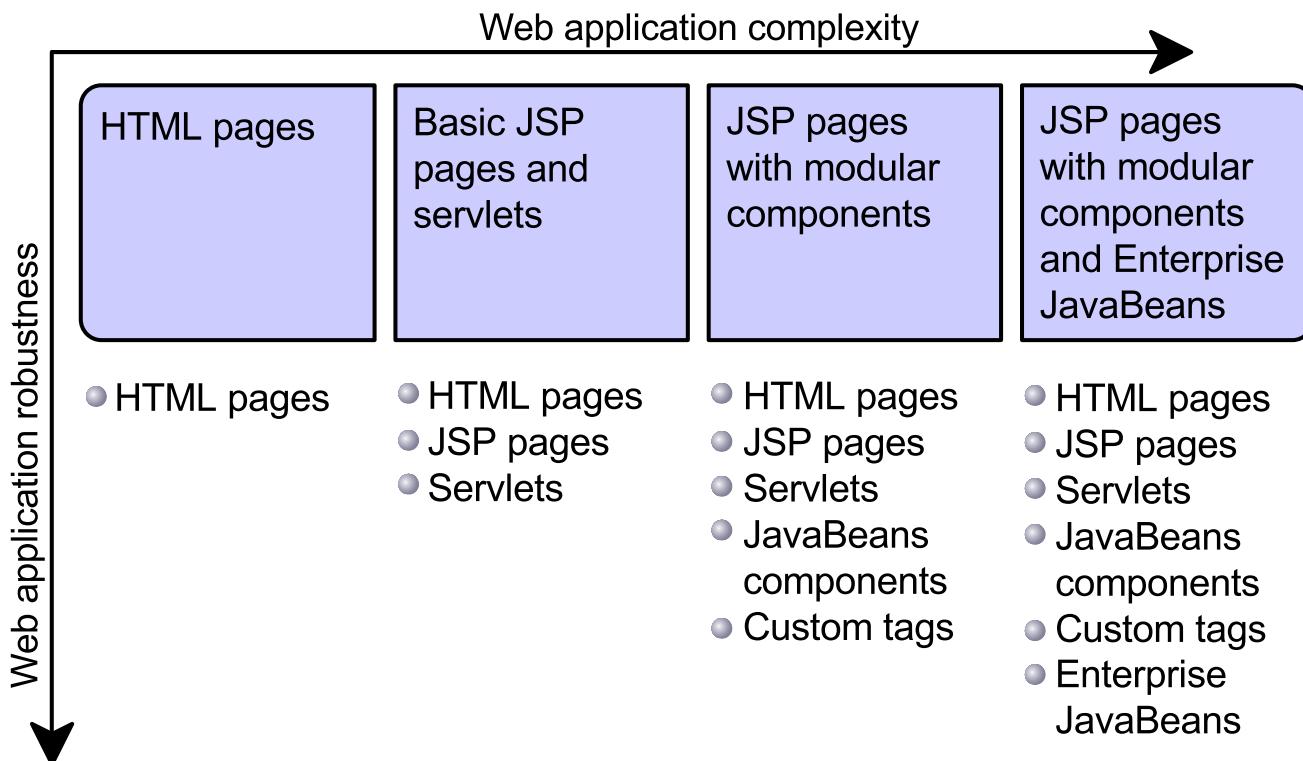
The modularity of a J2EE architecture clearly distinguishes several job roles:

- *Content Creator* – Creates View elements
- *Web Component Developer* – Creates Controller elements
- *Business Component Developer* – Creates Model elements
- *Data Access Developer* – Creates database access elements



Web Application Migration

A matrix showing the relationship between an architecture's complexity and robustness, based on the technologies used:





Summary

- Internet services are typically built on top of TCP/IP.
- HTTP is the Internet service for delivering HTML (and other) documents.
- HTTP servers can receive data from HTML forms through the Common Gateway Interface.
- Servlets are the Java technology for processing HTTP requests.
- JavaServer Pages are a template-based Java technology for handling presentation logic.
- The Java 2 Platform, Enterprise Edition, includes servlets and JSP pages in a broad enterprise development environment.



Module 2

Developing a Simple Servlet

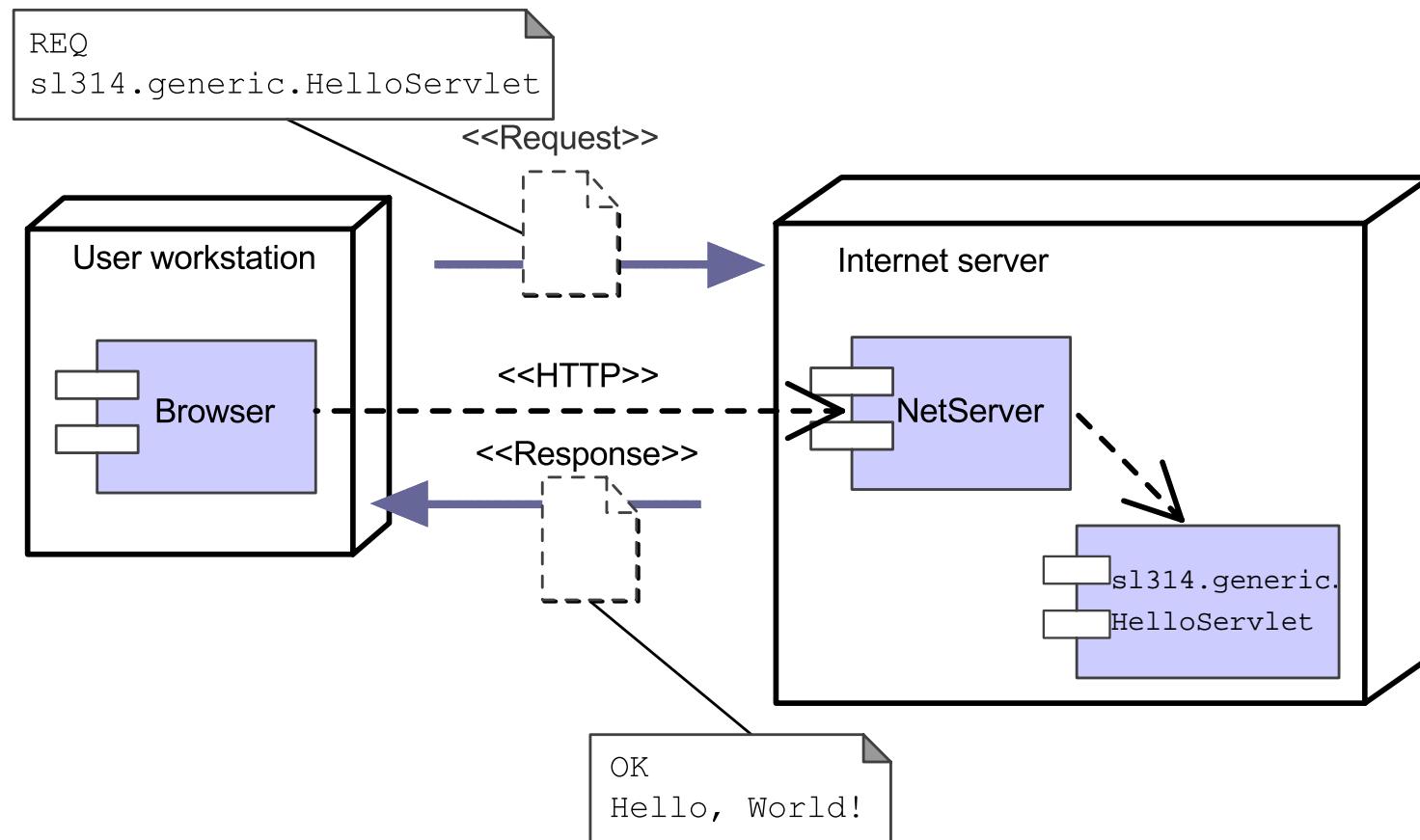


Objectives

- Develop a simple generic servlet
- Describe the Hypertext Transfer Protocol
- Develop a simple HTTP servlet
- Deploy a simple HTTP servlet
- Develop servlets that access request headers
- Develop servlets that manipulate response headers

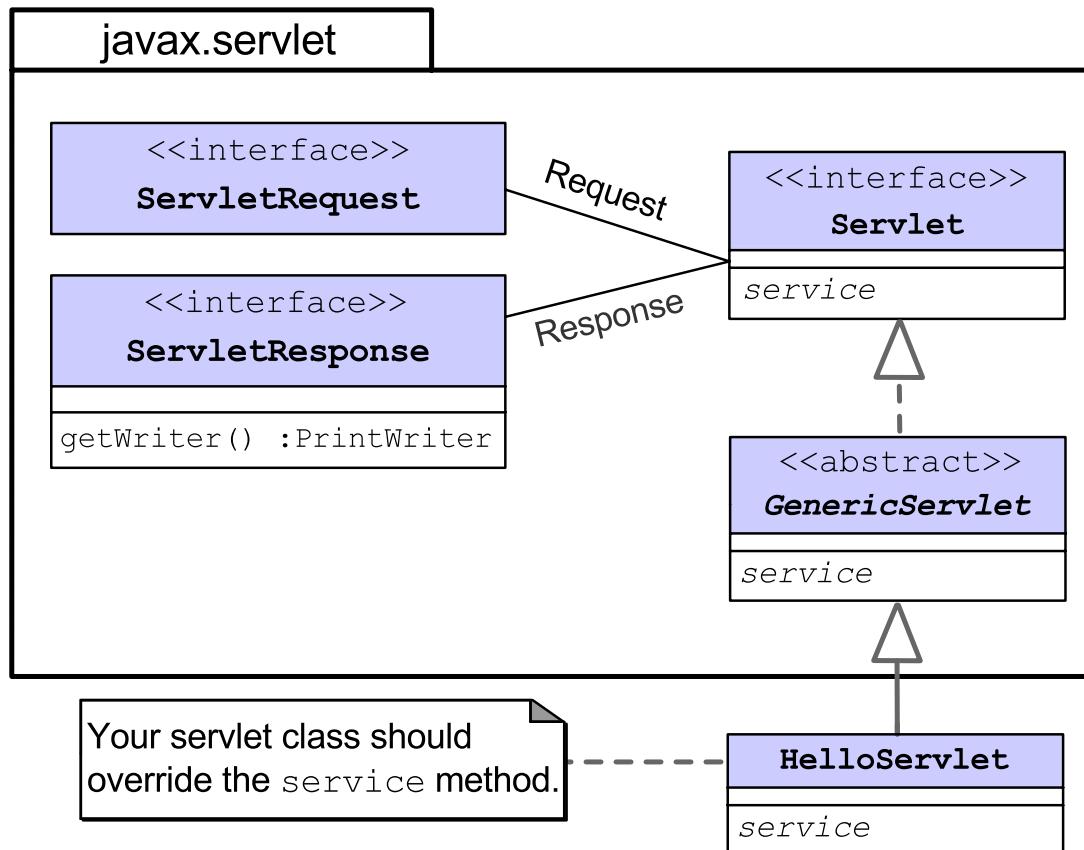


The NetServer Architecture





The Generic Servlets API



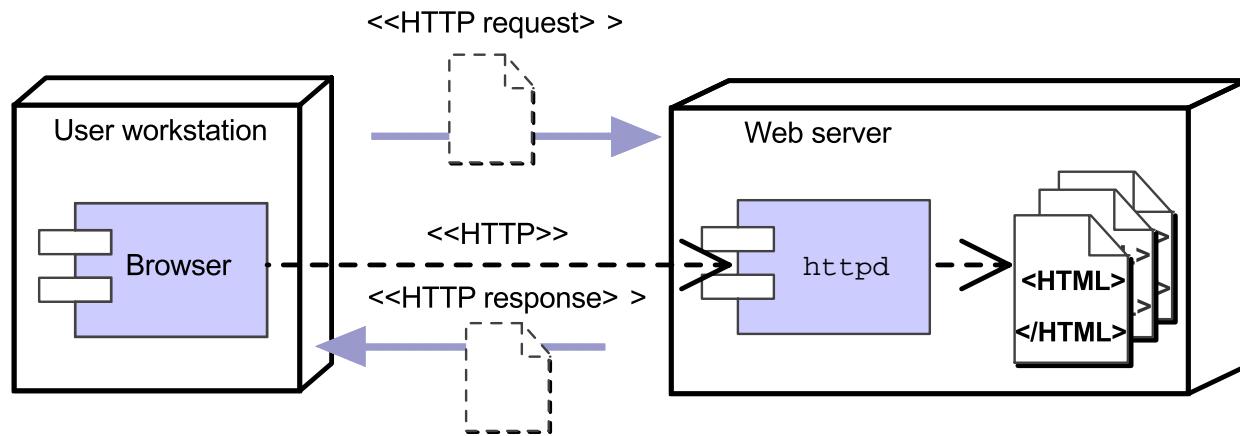


The Generic HelloServlet Class

```
1 package sl314.generic;
2
3 import javax.servlet.GenericServlet;
4 import javax.servlet.ServletRequest;
5 import javax.servlet.ServletResponse;
6 // Support classes
7 import java.io.IOException;
8 import java.io.PrintWriter;
9
10 public class HelloServlet extends GenericServlet {
11
12     public void service(ServletRequest request,
13                         ServletResponse response)
14         throws IOException {
15
16         PrintWriter out = response.getWriter();
17
18         // Generate the response
19         out.println("Hello, World!");
20         out.close();
21     }
22 }
```



Hypertext Transfer Protocol



- The Hypertext Transfer Protocol (HTTP) sends a single request to the HTTP daemon (`httpd`) and responds with the requested HTML document.
- The HTTP GET method is used to retrieve a document.
- Using CGI, HTTP requests can contain data from HTML forms.



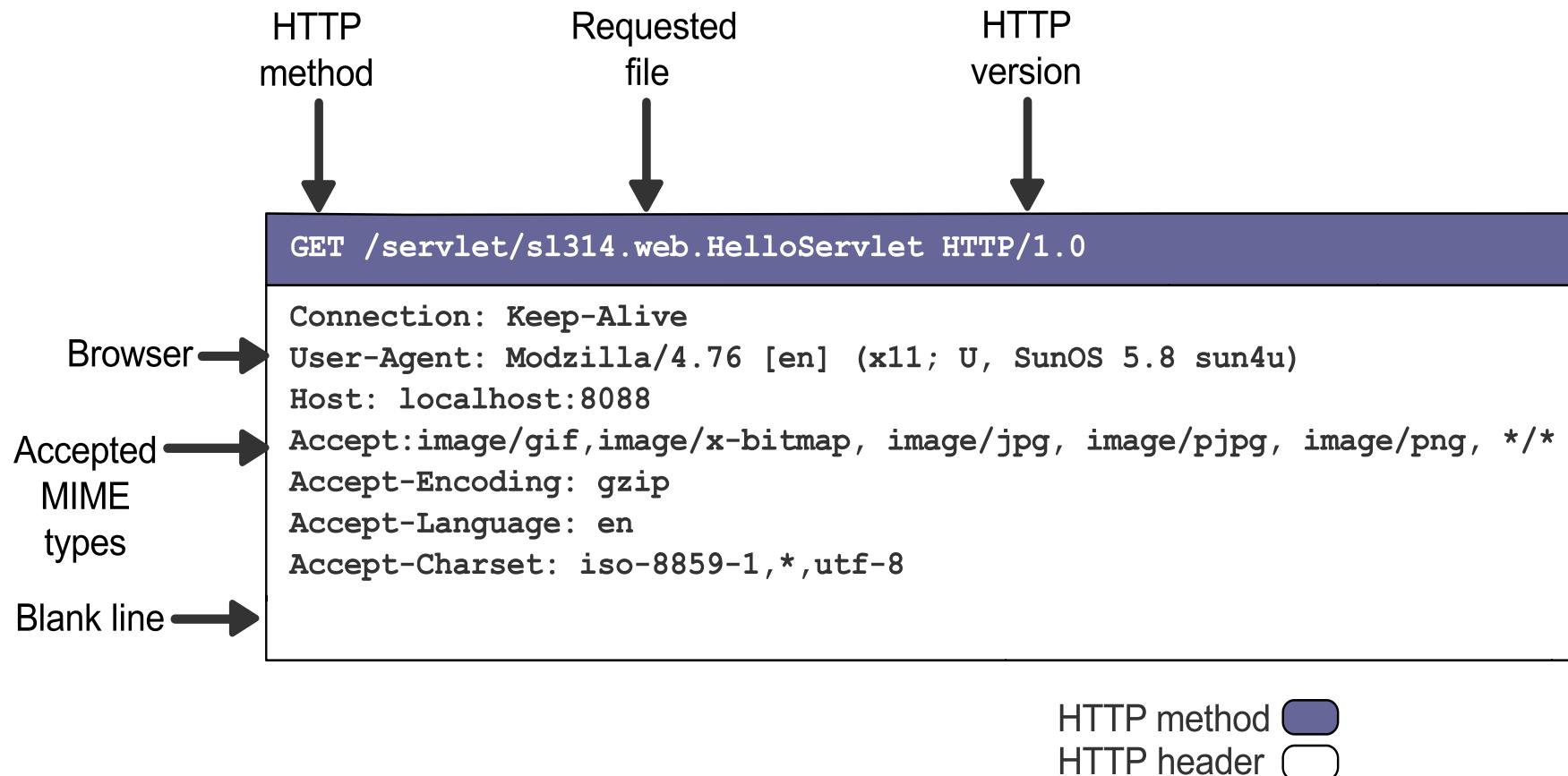
HTTP GET Method

A Web browser issues an HTTP GET request when:

- The user selects a link in the current HTML page
- The user enters a URL in the Location field (Netscape NavigatorTM) or the Address field (Internet Explorer)



HTTP Request





The HttpServletRequest API

Use the following API to retrieve request header data:

<<interface>>

HttpServletRequest

```
getHeader(name) : String  
getHeaderNames() : Enumeration  
getIntHeader(name) : int  
getDateHeader(name) : Date
```



HTTP Request Examples

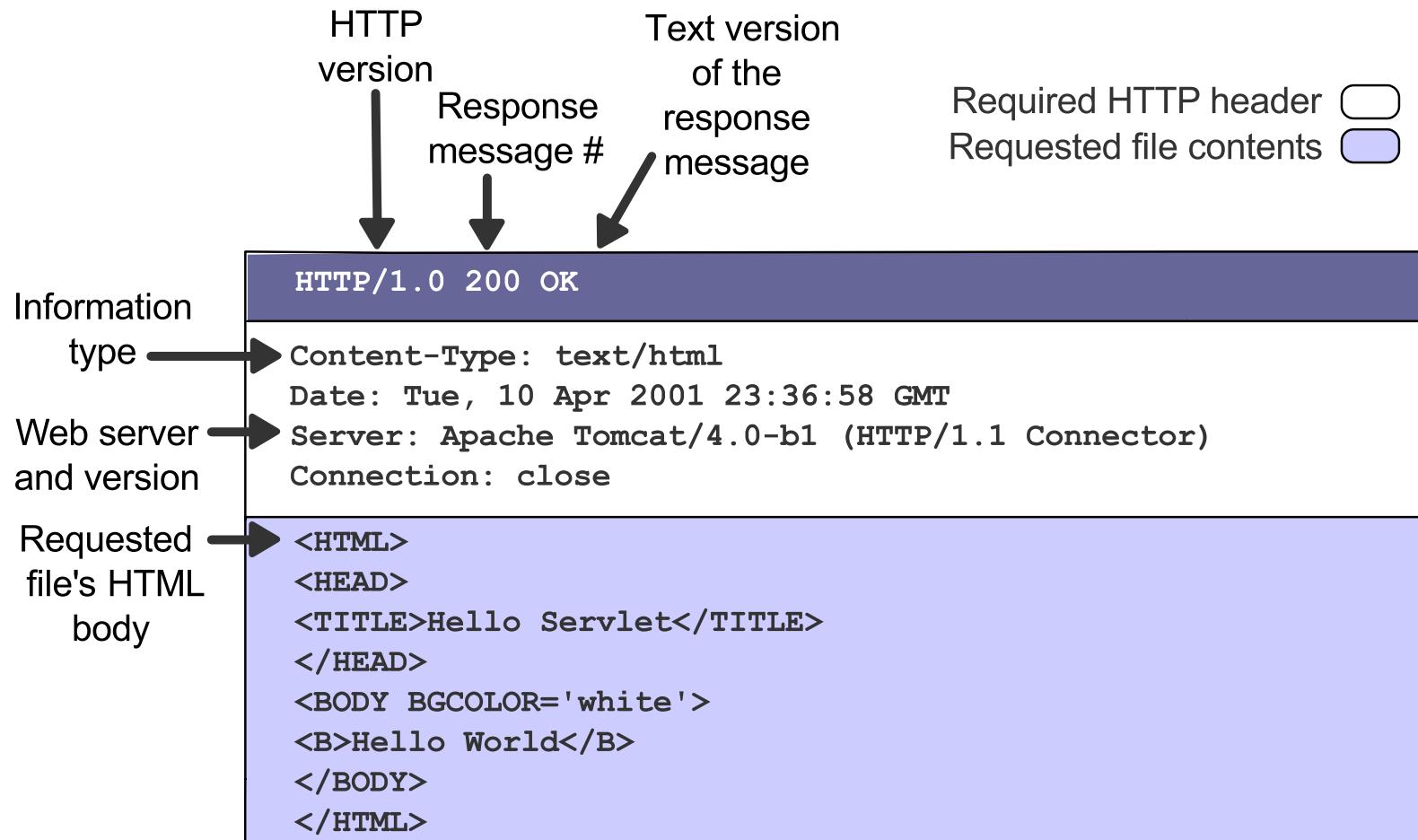
An example GET request:

```
GET /servlet/sl314.web.HelloServlet HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/4.76 [en] (X11; U; SunOS 5.8 sun4u)
Host: localhost:8088
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, image/
Accept-Encoding: gzip
```

```
String userAgent = request.getHeader("User-Agent");
// userAgent == "Mozilla/4.76 [en] (X11; U; SunOS 5.8 sun4u)"
Enumeration mimeTypes = request.getHeaders("Accept");
// first mimeTypes == "image/gif"
// second mimeTypes == "image/x-xbitmap", and so on
Enumeration headerNames = request.getHeaderNames();
// first headerNames == "Connection"
// second headerNames == "User-Agent", and so on
```



HTTP Response





The HttpServletResponse API

Use the following API to modify Response header data:

<<interface>>

HttpServletResponse

```
setContentType(String MIME_type)
getWriter() : PrintWriter
getOutputStream() : ServletOutputStream
addHeader(name:String, value:String)
addIntHeader(name:String, value:int)
addDateHeader(name:String, value:Date)
```



HTTP Response Examples

An example response stream:

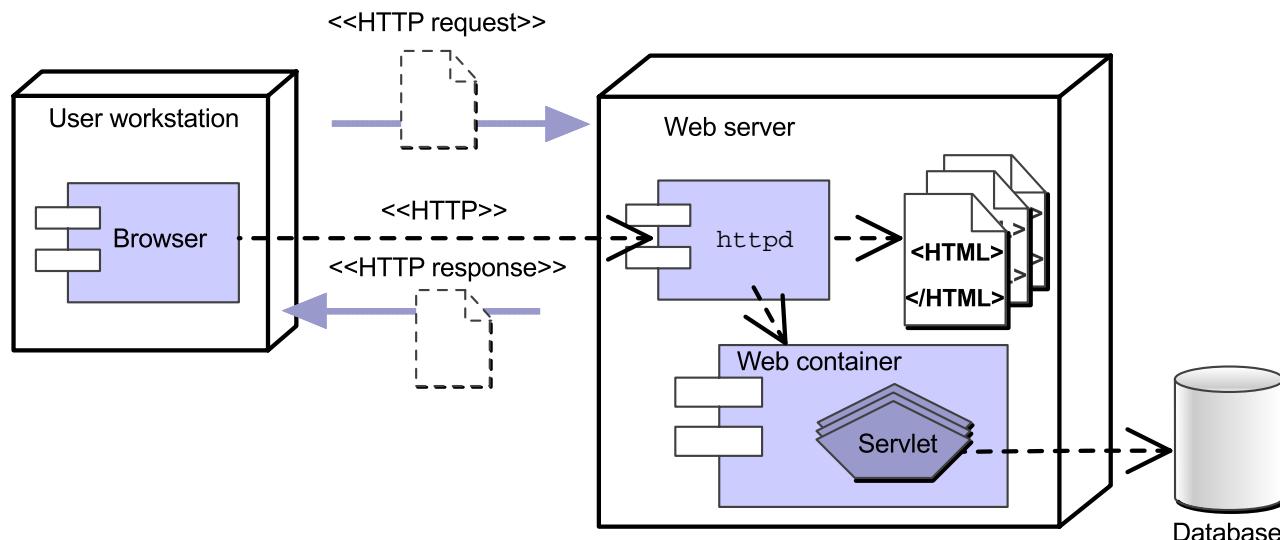
```
HTTP/1.0 200 OK
Content-Type: text/html
Date: Tue, 10 Apr 2001 23:36:58 GMT
Server: Apache Tomcat/4.0-b1 (HTTP/1.1 Connector)
Connection: close

<HTML>
<HEAD>
<TITLE>Hello Servlet</TITLE>
</HEAD>
```

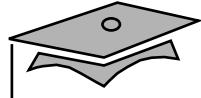
```
response.setContentType("text/html");
response.addDateHeader("Date", new Date());
response.setHeader("Connection", "close");
```



Web Container Architecture



- A Web container may be used to process HTTP requests by executing the service method on an HttpServlet object.



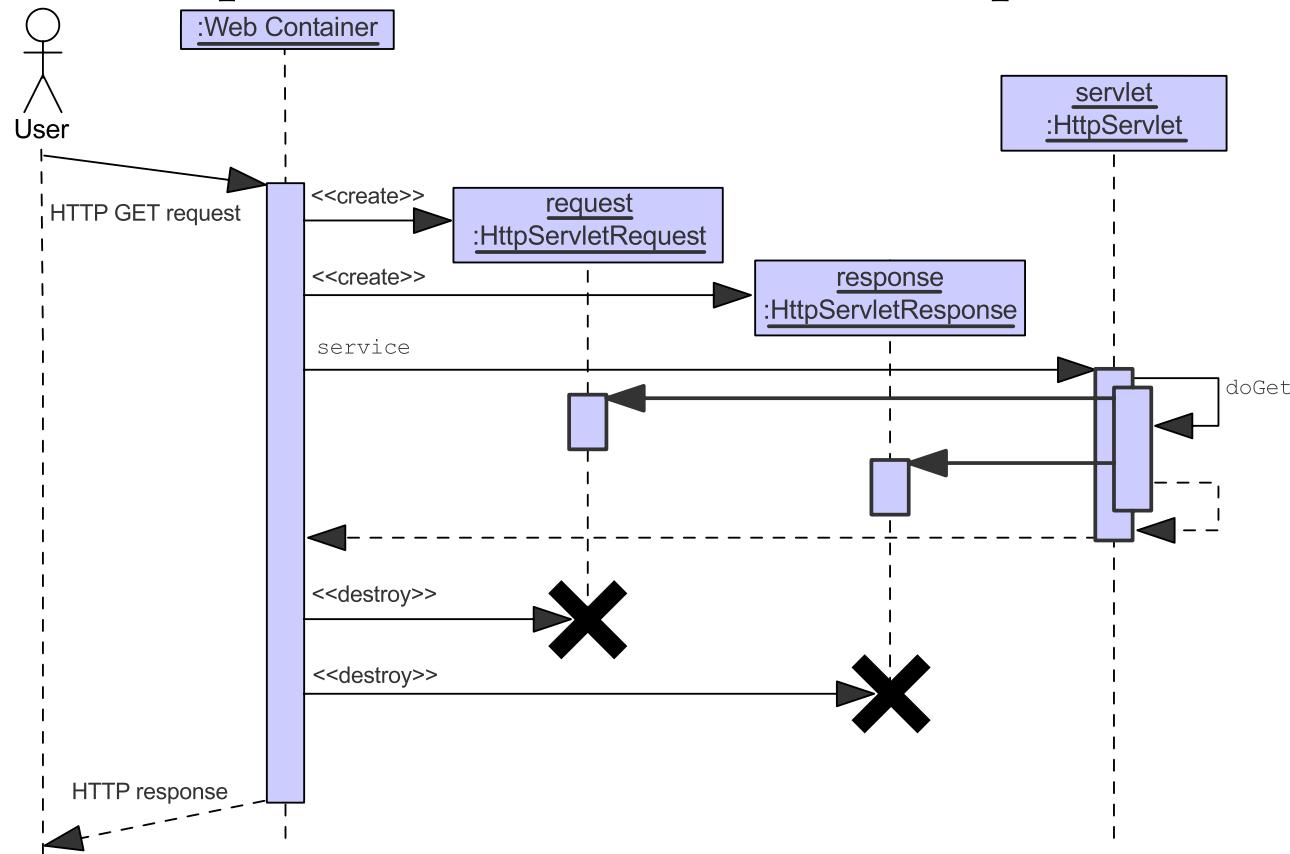
The Web Container

- Controls the communication between the httpd daemon and your servlets
- Is a Java™ virtual machine (JVM™) that includes an implementation of the servlet API



Sequence Diagram of HTTP GET Request

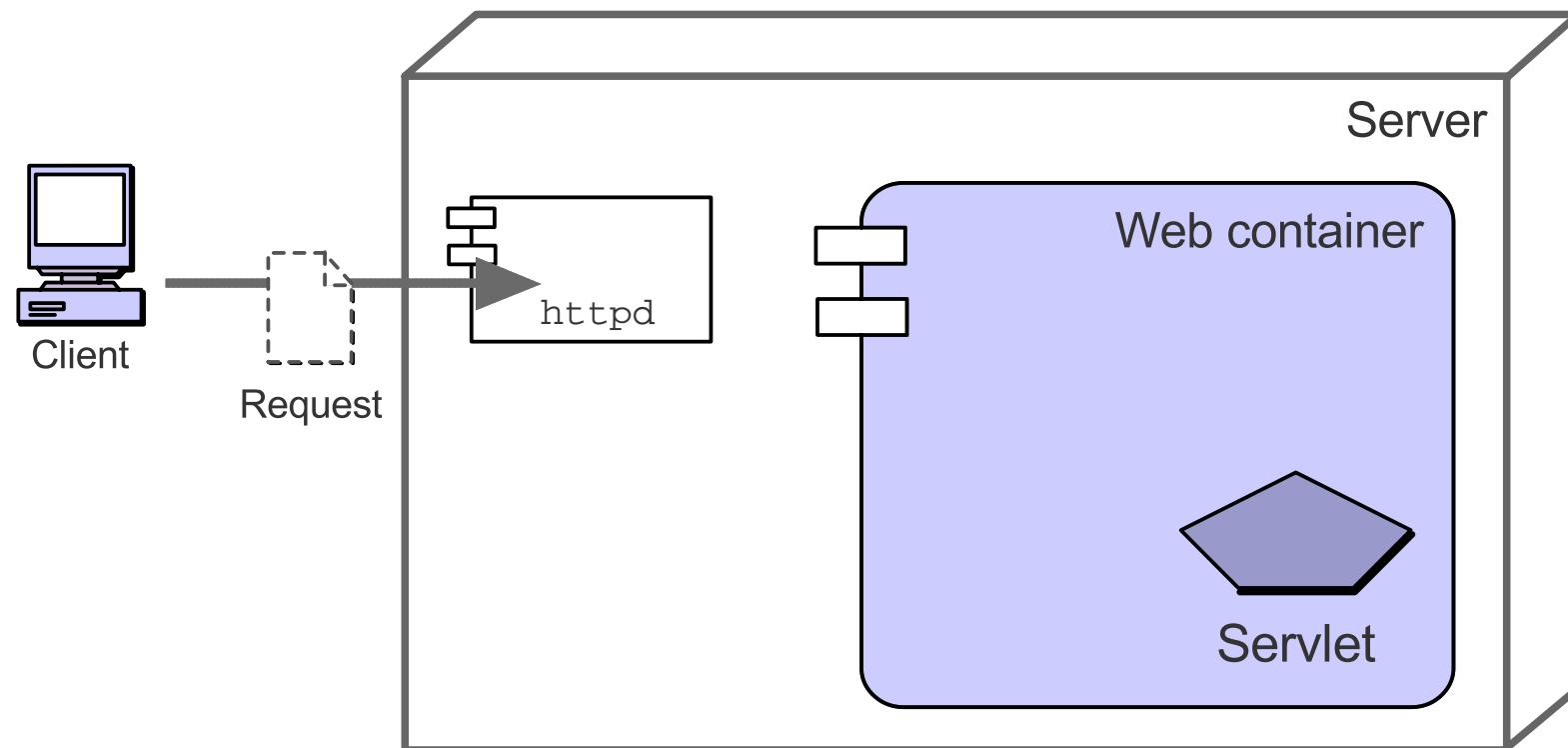
This diagram shows the time-based sequence of events as the Web container processes an HTTP GET request:





Request and Response Process

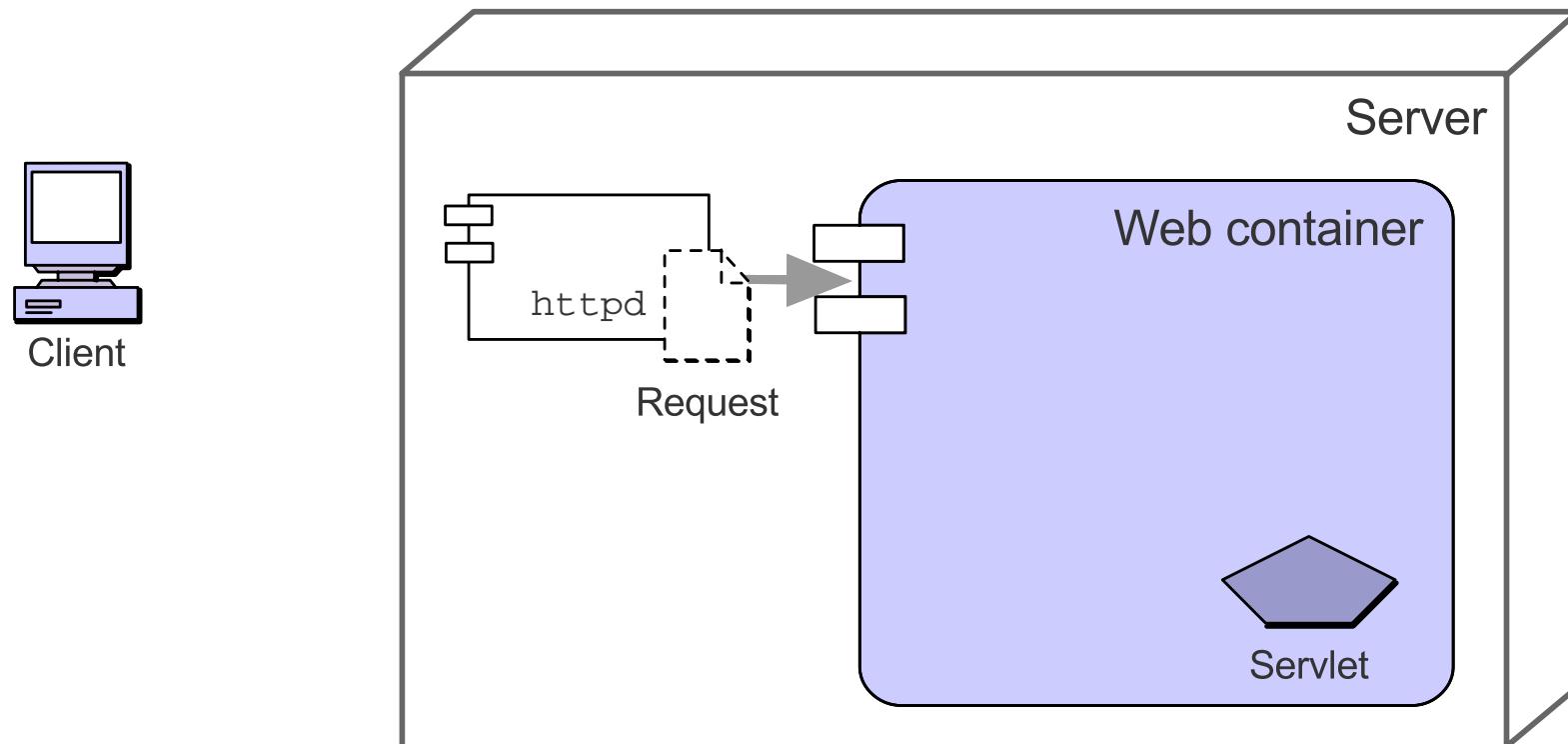
The HTTP request arrives at the HTTP server daemon:





Request and Response Process

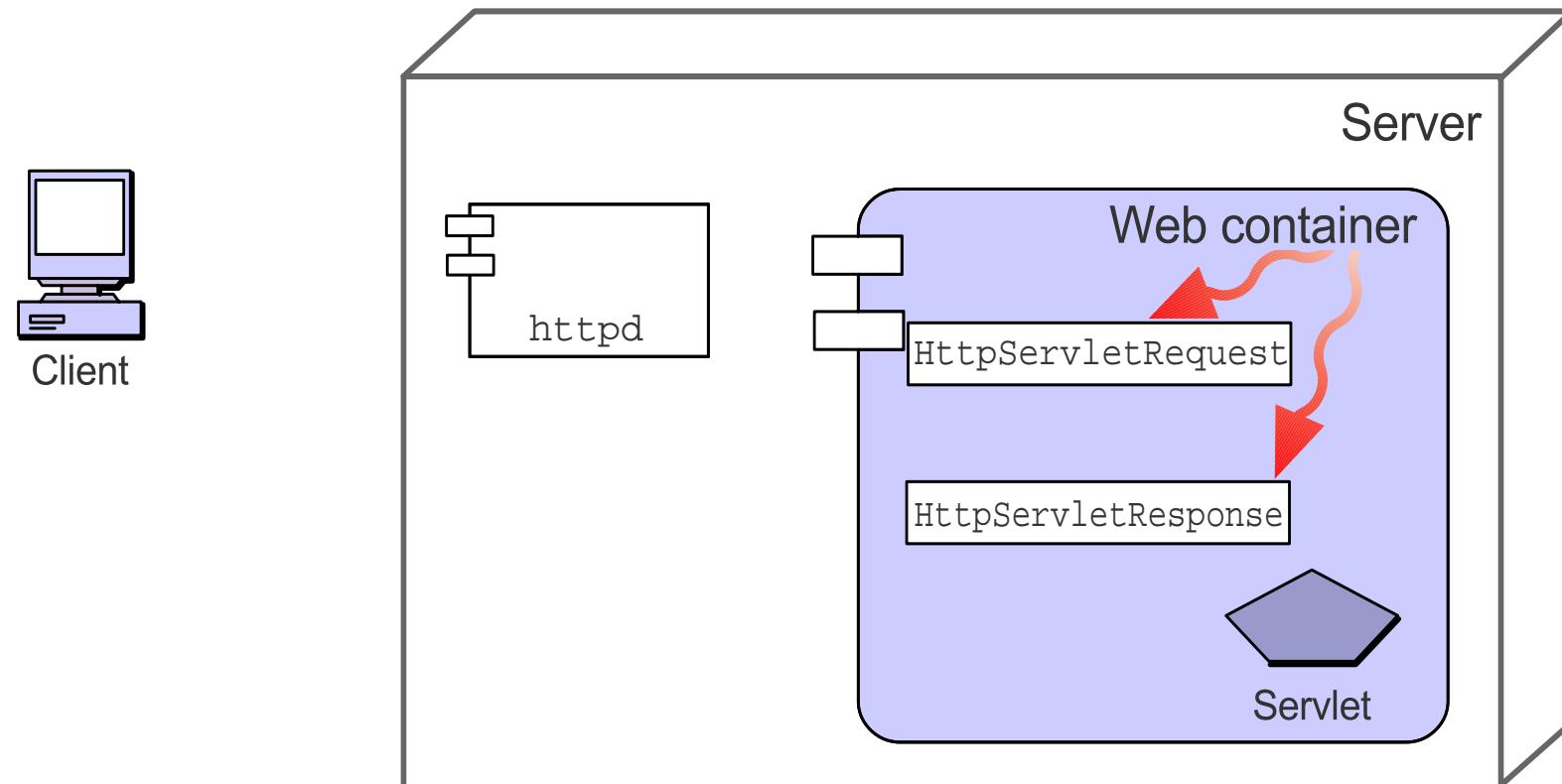
The HTTP request is transferred to the Web container:





Request and Response Process

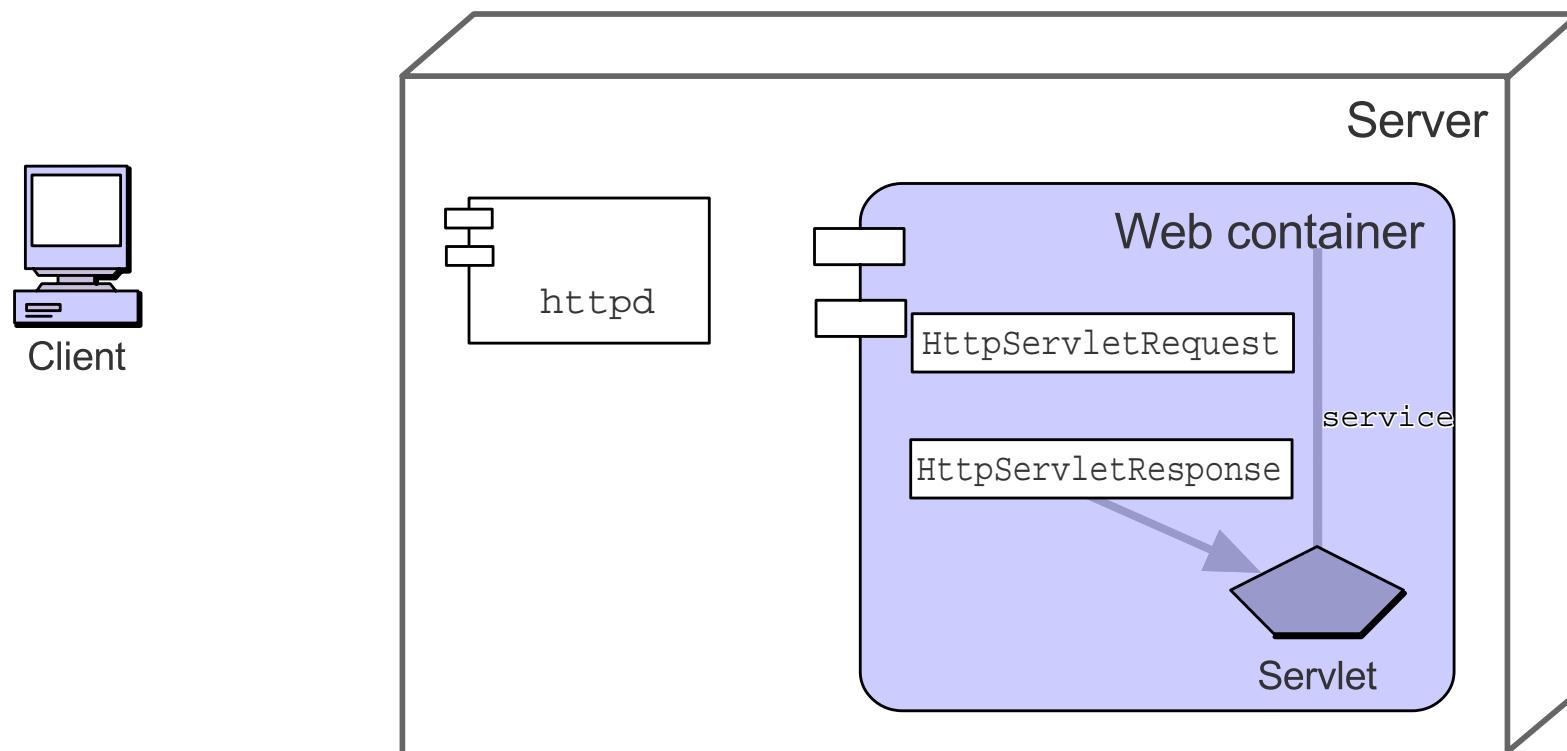
The Web container creates the request and response objects:





Request and Response Process

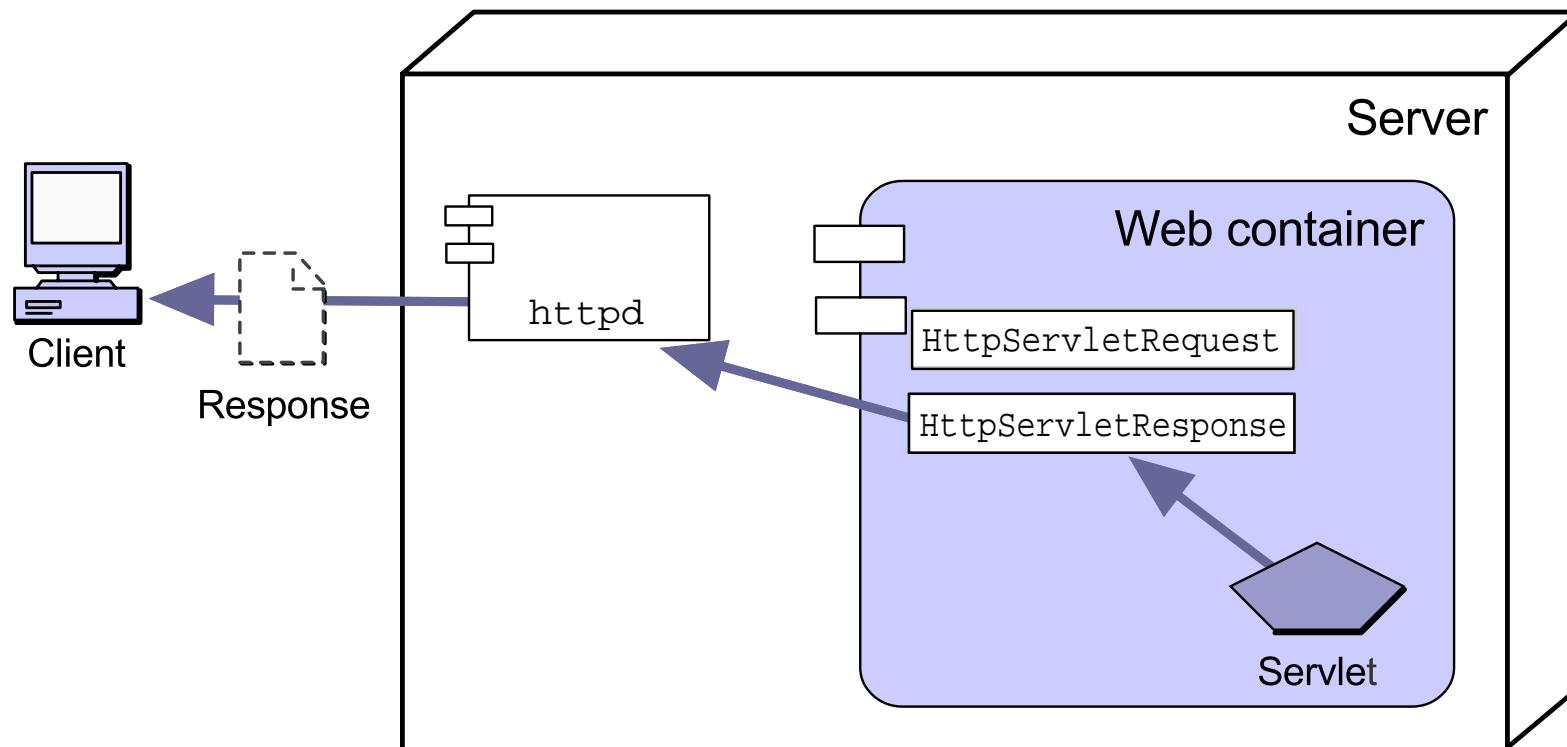
The Web container executes the service method on the requested servlet, passing to it the request and response objects:





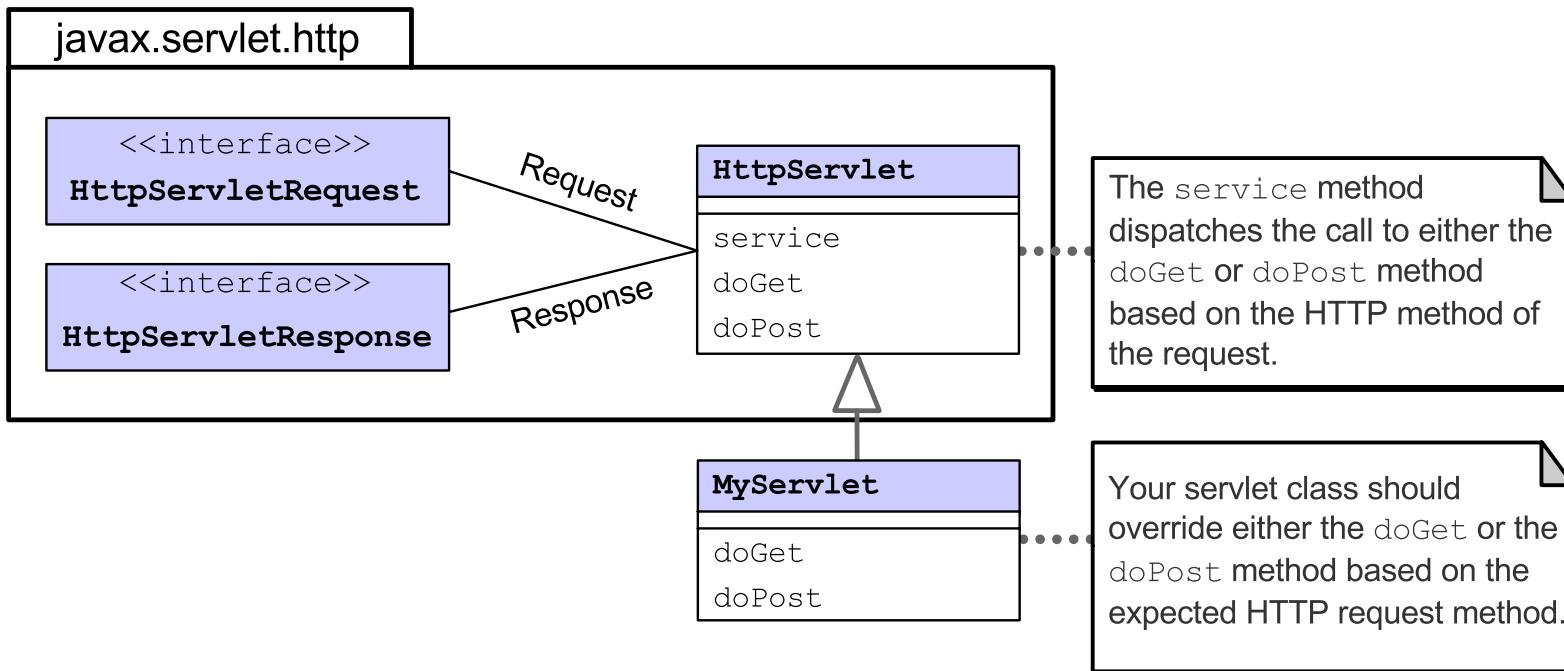
Request and Response Process

The servlet uses the response object to print the HTML response to the client:





The HTTP Servlet API



- In the `HttpServlet` class, the `service` method dispatches based on the HTTP method.
- You can implement an HTTP servlet by extending the `HttpServlet` class and overriding the `doGet` method.



The HTTP HelloServlet Class

```
1 package sl314.web;
2
3 import javax.servlet.http.HttpServlet;
4 import javax.servlet.http.HttpServletRequest;
5 import javax.servlet.http.HttpServletResponse;
6 // Support classes
7 import java.io.PrintWriter;
8 import java.io.IOException;
9
10 public class HelloServlet extends HttpServlet {
11
12     public void doGet(HttpServletRequest request,
13                         HttpServletResponse response)
14         throws IOException {
15
16         // Specify the content type is HTML
17         response.setContentType("text/html");
18         PrintWriter out = response.getWriter();
19
20         // Generate the HTML response
21         out.println("<HTML> ");
22         out.println("<HEAD> ");
23         out.println("<TITLE>Hello Servlet</TITLE> ");
24         out.println("</HEAD> ");
25         out.println("<BODY BGCOLOR='white'> ");
26         out.println("<B>Hello, World</B> ");
27         out.println("</BODY> ");
28         out.println("</HTML> ");
29
30         out.close();
31     }
32 }
```



Deploying a Servlet

Deploying a servlet involves the following steps:

- Ensuring that the Web server is installed, configured, and running correctly
- Putting the servlet in the correct Web server directory
- Providing a means for the servlet to run



Installing, Configuring, and Running the Web Container

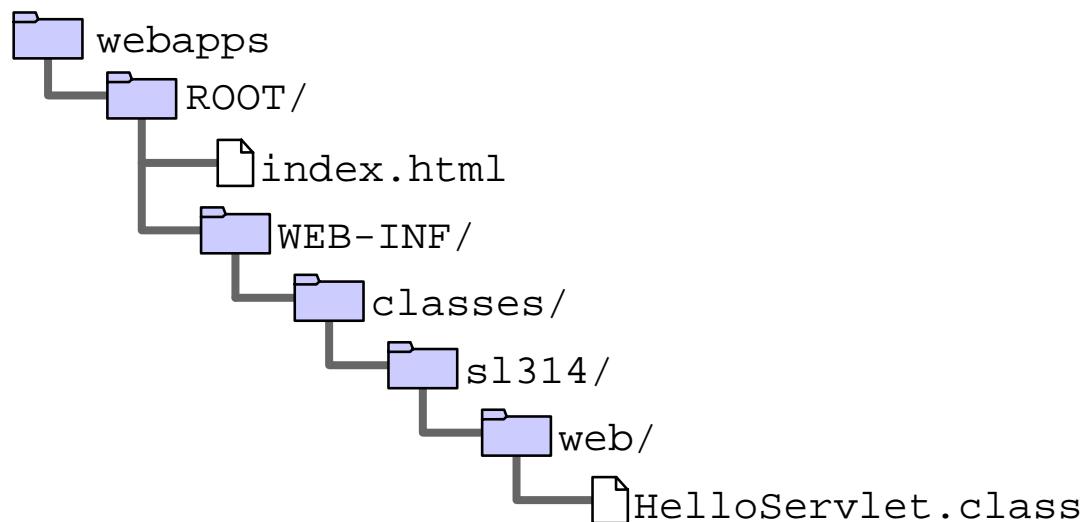
- These steps are vendor dependent.
- This course uses the Tomcat Web server.

In standalone mode, the Tomcat server performs the duties of the Web server (the httpd) as well as the Web container.



Deploying the Servlet to the Web Container

- The details of deploying a simple servlet are specific to the Web server.
- In the Tomcat server, put compiled servlet class files in the webapp/ROOT/WEB-INF/classes/ directory





Activating the Servlet in a Web Browser

Enter the HTTP URL directly into your browser's Location edit field. This activates the servlet using the HTTP GET method.

Syntax:

http://hostname:port/servlet/fully_qualified_class

Example:

`http://localhost:8080/servlet/sl314.web.HelloServlet`



Summary

Guidelines for writing a simple servlet:

- The servlet should be in a package. As Web applications grow the package structure evolves.
- The servlet should generate a response, sending the message back to the browser in HTML. The servlet retrieves the PrintWriter object using the getWriter method on the HttpServletResponse object.
- You must specify the content type of the response message. Use the setContentType method on the HttpServletResponse object. This method takes a String that names a MIME type; such as “text/html.”



Module 3

Developing a Simple Servlet That Uses HTML Forms



Objectives

- Describe the structure of HTML FORM tags
- Describe how HTML forms send data using the Common Gateway Interface
- Develop an HTTP servlet that accesses form data



HTML Forms

Sports League Registration Form

Select League

League Season:

Year:

Enter Player Information

Name:

Address:

City: Province: Postal code:

Select Division

Division:



The FORM Tag

- ACTION – Specifies the destination (as a relative URL) for the form information
- METHOD – Specifies the HTTP method (GET or POST)
- Syntax:

```
<FORM ACTION='servlet-URL' METHOD='{GET|POST}'>  
  {HTML form tags and other HTML content}  
</FORM>
```



The FORM Tag

```
7  <BODY BGCOLOR='white'>
8
9  <B>Say Hello Form</B>
10
11 <FORM ACTION='/servlet/s1314.web.FormBasedHello' METHOD='POST'>
12 Name: <INPUT TYPE='text' NAME='name'> <BR>
13 <BR>
14 <INPUT TYPE='submit'>
15 </FORM>
16
17 </BODY>
```

The HTML generates this form:

The image shows a web page with a pink border. At the top, it says "Say Hello Form". Below that is a text input field with the placeholder "Name:" and the value "Bryan" typed into it. At the bottom is a grey rectangular button with the text "Submit Query" in white.



HTML Form Components

| Form Element | Tag | Description |
|-----------------------|--|---|
| Textfield | <INPUT TYPE='text' ...> | Enter a single line of text. |
| Submit button | <INPUT TYPE='submit'> | The button to submit the form. |
| Reset button | <INPUT TYPE='reset'> | The button to reset the fields in the form. |
| Checkbox | <INPUT TYPE='checkbox' ...> | Choose one or more options. |
| Radio button | <INPUT TYPE='radio' ...> | Choose only one option. |
| Password | <INPUT TYPE='password' ...> | Enter a single line of text, but the text entered cannot be seen. |
| Hidden | <INPUT TYPE='hidden' ...> | A static data field. This does not show up in the HTML form in the browser window, but the data is sent to the server in the CGI. |
| Select drop-down list | <SELECT ...> <OPTION ...> ... </SELECT> | Select one or more options from a list box. |
| Textarea | <TEXTAREA ...> ... </TEXTAREA> | Enter a paragraph of text. |



Input Tags

There are three critical attributes for any INPUT tag:

- TYPE – The type of GUI component
- NAME – The name of the form parameter
- VALUE – The default value of the component

Textfield:

```
<INPUT TYPE='text' NAME='name' VALUE='default value' SIZE='20'>
```

A screenshot of a web browser showing a single-line text input field. The placeholder text "default value" is visible inside the field, which is highlighted with a gray background.



Input Tags

Submit button:

```
<INPUT TYPE='submit'> <BR>
<INPUT TYPE='submit' VALUE='Register'> <BR>
<INPUT TYPE='submit' NAME='operation' VALUE='Send Mail'> <BR>
```



Reset button:

```
<INPUT TYPE='reset'>
```





Input Tags

Checkbox:

```
<INPUT TYPE='checkbox' NAME='fruit' VALUE='apple'> I like apples <BR>
<INPUT TYPE='checkbox' NAME='fruit' VALUE='orange'> I like oranges <BR>
<INPUT TYPE='checkbox' NAME='fruit' VALUE='banana'> I like bananas <BR>
```

I like apples
 I like oranges
 I like bananas

Radio button:

```
<INPUT TYPE='radio' NAME='gender' VALUE='F'> Female <BR>
<INPUT TYPE='radio' NAME='gender' VALUE='M'> Male <BR>
```

Female
 Male



Input Tags

Password:

```
<INPUT TYPE='password' NAME='psword' VALUE='secret' MAXSIZE='16'>
```

A rectangular input field with a gray border. Inside the field, there are five black asterisks ('*****') representing the password.

Hidden:

```
<INPUT TYPE='hidden' NAME='action' VALUE='SelectLeague'>
```

A hidden field does not appear in the browser, but the data is included in the form data of the HTTP request.



The Select Tag

Single selection:

```
<SELECT NAME='favoriteArtist'>
    <OPTION VALUE='Genesis'> Genesis
    <OPTION VALUE='PinkFloyd' SELECTED> Pink Floyd
    <OPTION VALUE='KingCrimson'> King Crimson
</SELECT>
```





The Select Tag

Multiple selection:

```
<SELECT NAME='sports' MULTIPLE>  
  <OPTION VALUE='soccer' SELECTED> Soccer  
  <OPTION VALUE='tennis'> Tennis  
  <OPTION VALUE='ultimate' SELECTED> Ultimate Frisbee  
</SELECT>
```

A screenshot of a web browser showing a multiple-select dropdown menu. The menu contains three items: "Soccer", "Tennis", and "Ultimate Frisbee". The first item, "Soccer", is highlighted with a solid black rectangular background, indicating it is the currently selected option. The other two items, "Tennis" and "Ultimate Frisbee", are shown in white text on a light gray background.



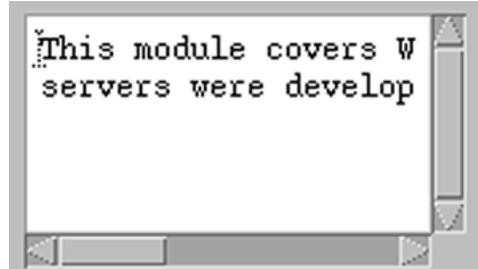
The Textarea Tag

Textarea tag:

```
<TEXTAREA NAME='comment' ROWS='5' COLUMNS='70'>
```

This module covers Web application basics: how browsers and Web servers were developed and what they do. It also...

```
</TEXTAREA>
```





Form Data in the HTTP Request

HTTP includes a specification for data transmission used to send HTML form data from the Web browser to the Web server.

Syntax:

fieldName1=fieldValue1&fieldName2=fieldValue2&... .

Examples:

name=Bryan

season=Spring&year=2001&name=Bryan+Basham&address=4747+Bogus+Drive&city=Boulder&province=Colorado&postalCode=80303&division=Amateur



HTTP GET Method Request

Form data is contained in the URL of the HTTP request:

```
GET /servlet/s1314.web.FormBasedHello?name=Bryan HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/4.76C-CCK-MCD Netscape [en] (X11; U; SunOS 5.8 sun4u)
Host: localhost:8088
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, image/png, */*
Accept-Encoding: gzip
Accept-Language: en
Accept-Charset: iso-8859-1,* ,utf-8
```



HTTP POST Method Request

Form data is contained in the body of the HTTP request:

```
POST /register HTTP/1.0
Referer: http://localhost:8080/model1/formEchoServer.html
Connection: Keep-Alive
User-Agent: Mozilla/4.76C-CCK-MCD Netscape [en] (X11; U; SunOS 5.8 sun4u)
Host: localhost:8088
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, image/png, */*
Accept-Encoding: gzip
Accept-Language: en
Accept-Charset: iso-8859-1,* ,utf-8
Content-type: application/x-www-form-urlencoded
Content-length: 129

season=Spring&year=2001&name=Bryan+Basham&address=4747+Bogus+Drive&city=Bo
ulder&province=Colorado&postalCode=80303&division=Amateur
```

The HTTP POST method can only be activated from a form.



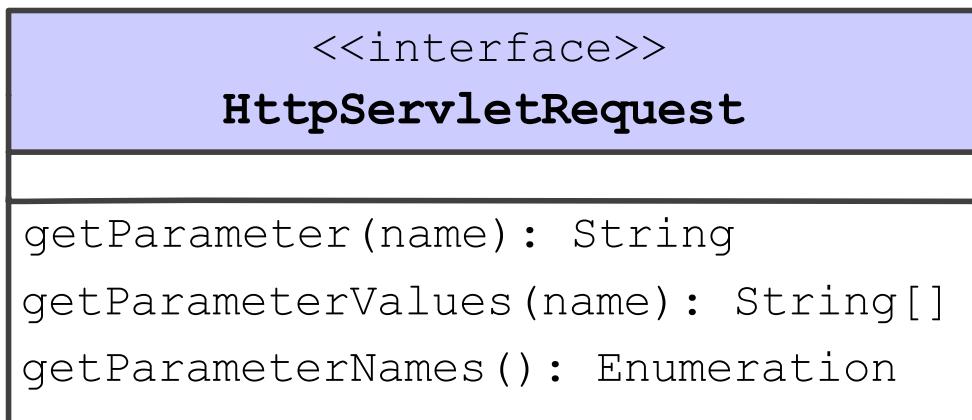
To GET or to POST?

- The HTTP GET method is used when:
 - The processing of the request is idempotent; this means that the request does not have side-effects on the server.
 - The amount of form data is small.
 - You want to allow the request to be bookmarked
- The HTTP POST method is used when:
 - The processing of the request changes the state of the server, such as storing data in a database.
 - The amount of form data is large.
 - The contents of the data should not be visible in the URL (for example, passwords).



The Servlet API

You can access the HTML form parameters from the request object:



Examples:

```
String name = request.getParameter("name");
String[] sports = request.getParameterValues("sports");
```



The FormBasedHello Servlet

From examples/forms/say_hello.html:

```
11 <FORM ACTION='/servlet/sl314.web.FormBasedHello' METHOD='POST'>
12 Name: <INPUT TYPE='text' NAME='name'> <BR>
13 <BR>
14 <INPUT TYPE='submit'>
15 </FORM>
```

Say Hello Form

Name:



The FormBasedHello Servlet

```
1 package sl314.web;
2
3 import javax.servlet.http.HttpServlet;
4 import javax.servlet.http.HttpServletRequest;
5 import javax.servlet.http.HttpServletResponse;
6 // Support classes
7 import java.io.IOException;
8 import java.io.PrintWriter;
9
10
11 public class FormBasedHello extends HttpServlet {
12
13     private static final String DEFAULT_NAME = "World";
14
15     public void doGet(HttpServletRequest request,
16                         HttpServletResponse response)
17         throws IOException {
18         generateResponse(request, response);
19     }
20
21     public void doPost(HttpServletRequest request,
22                         HttpServletResponse response)
23         throws IOException {
24         generateResponse(request, response);
25     }
26 }
```



The FormBasedHello Servlet

```
27  public void generateResponse(HttpServletRequest request,
28                      HttpServletResponse response)
29      throws IOException {
30
31     // Determine the specified name (or use default)
32     String name = request.getParameter("name");
33     if ( (name == null) || (name.length() == 0) ) {
34         name = DEFAULT_NAME;
35     }
36
37     // Specify the content type is HTML
38     response.setContentType("text/html");
39     PrintWriter out = response.getWriter();
40
41     // Generate the HTML response
42     out.println("<HTML>");
43     out.println("<HEAD>");
44     out.println("<TITLE>Hello Servlet</TITLE> ");
45     out.println("</HEAD>");
46     out.println("<BODY BGCOLOR='white'> ");
47     out.println("<B>Hello, " + name + "</B> ");
48     out.println("</BODY> ");
49     out.println("</HTML> ");
50
51     out.close();
52 }
53 }
```



Summary

- HTML FORM and INPUT tags are used by the browser to send form data to a servlet.
- The ACTION attribute of the FORM tag tells the browser which servlet to activate. The METHOD attribute tells the browser which HTTP Method to use.
- The NAME attribute in the INPUT tag provides the name in the CGI data.
- The HttpServletRequest object encapsulates the form data.
- The getParameter method is used to extract the value (as a String) of the parameter:

```
String city = request.getParameter("city");
```



Module 4

Developing a Web Application Using a Deployment Descriptor



Objectives

- Describe the requirements of a robust Web application model
- Develop a Web application using a deployment descriptor



Problems With Simple Servlets

- Simple servlets are activated by using the fully qualified class name in the URL.
- Multiple, unrelated servlets are placed into the same directory on the Web server.
- Many common services (such as security authorization) are coded from scratch.

This module introduces you to the concept of a Web application and its deployment descriptor to solve the problems with simple servlets.



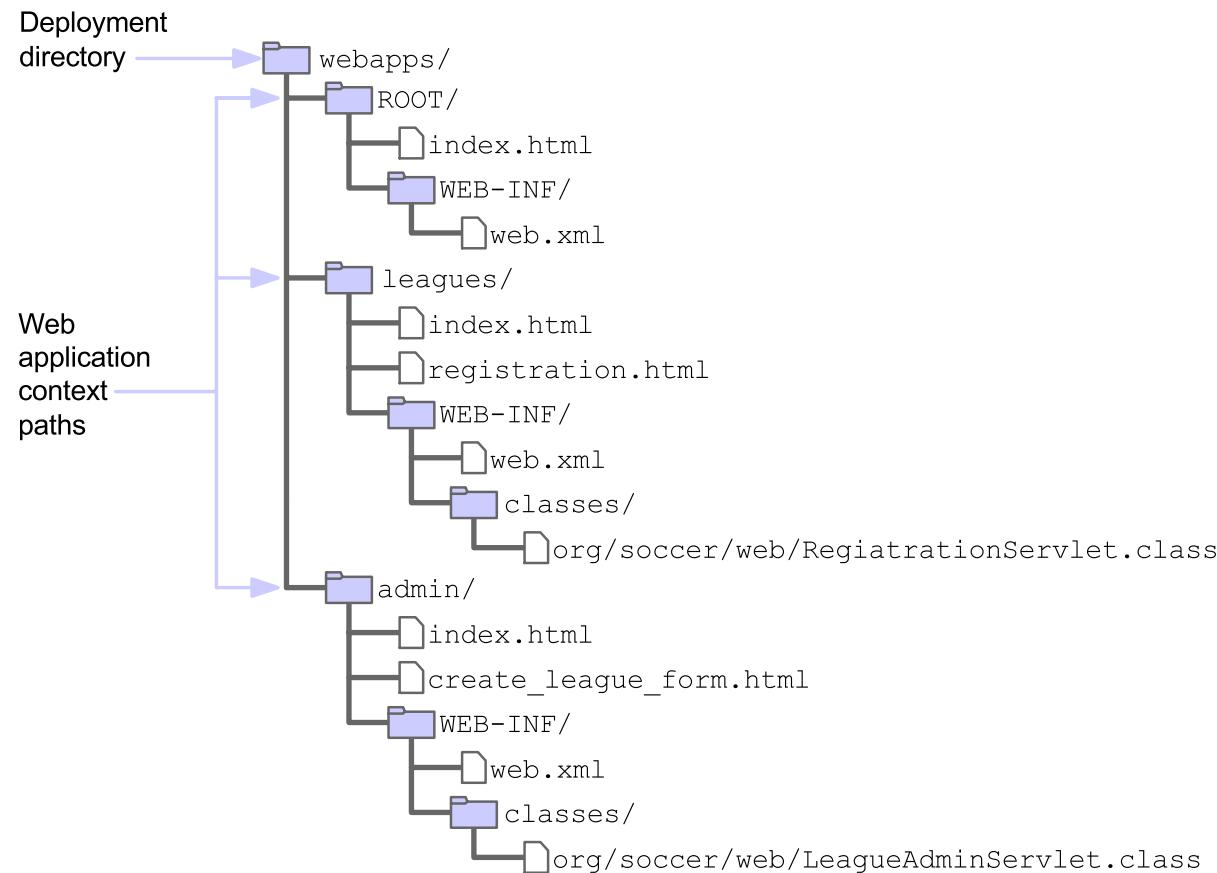
Problems With Deploying in One Place

- The simple servlet deployment model puts all servlet classes in one directory. For large systems, this is a poor organizational choice.
- This deployment model also runs the risk of security breaches if a hacker can “view” the deployment directory, which might contain secure servlets.



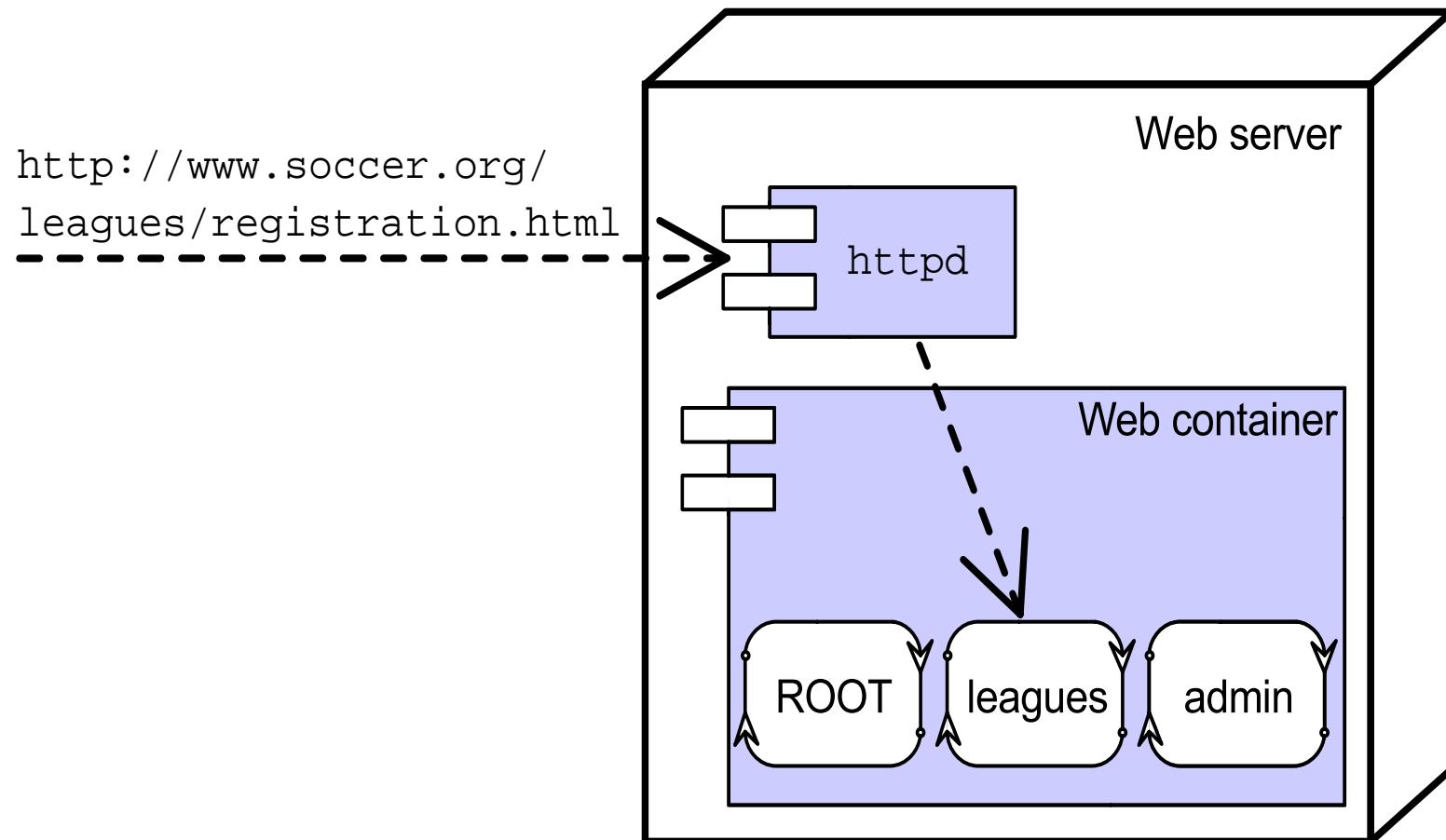
Multiple Web Applications

You can group multiple Web applications in a Web container:





Using Multiple Web Applications





Web Application Context Name

- A Web application is identified in the URL by the first “level of hierarchy.” This is called the context name.
- Syntax:

`http://host:port/context/path/file`

- Examples:

`http://www.soccer.org/leagues/registration.html`

`http://www.soccer.org/admin/create_league_form.html`

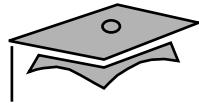
- The ROOT Web application does not have a context name:

`http://www.soccer.org/main.html`



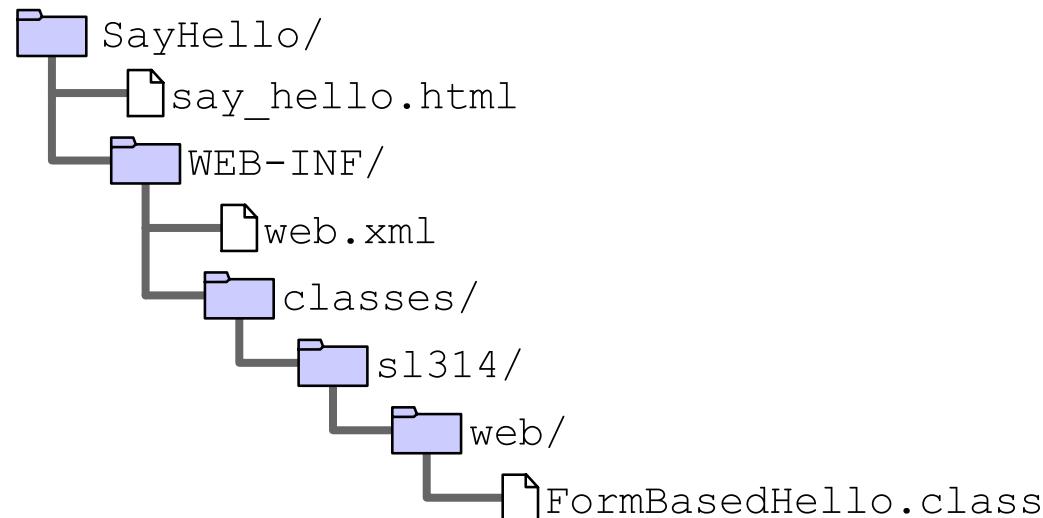
Problems With Servlet Naming

- Using the fully qualified class name in the URL can cause excessive typing for the Web application user as well as for the maintainer of the Web pages and is error prone.
- Using the fully qualified class name in the URL exposes the servlet implementation to the world. This could be a security risk.



Problems With Servlet Naming

Deployment directory structure:



The old-style URL to activate the Hello servlet:

`http://localhost:8080/SayHello/servlet/s1314.web.FormBasedHello`

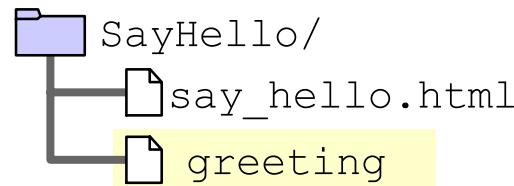


Solution to Servlet Naming Problems

Servlet mapping in the deployment descriptor:

```
12    <servlet>
13        <servlet-name>Hello</servlet-name>
14        <servlet-class>s1314.web.FormBasedHello</servlet-class>
15    </servlet>
16
17    <servlet-mapping>
18        <servlet-name>Hello</servlet-name>
19        <url-pattern>/greeting</url-pattern>
20    </servlet-mapping>
```

Effective Web application structure:



The new-style URL to activate the Hello servlet:

`http://localhost:8080/SayHello/greeting`



Problems Using Common Services

- In past versions of the Servlet specification, services like security, error page handling, declaration of initialization parameters, and so on, were not clearly defined.
- Servlet container vendors implemented these features in a variety of different ways. This lead to vendor-specific deployment strategies which made it difficult to port Web applications to different implementations.

Solution:

- Since version 2.2 of the Servlet specification, the Deployer uses the deployment descriptor to configure the services used by the Web application.



Developing a Web Application Using a Deployment Descriptor

This section describes:

- The structure of the deployment descriptor
- Only one possible structure for your development environment
- The structure of the deployment file hierarchy
- The structure of a Web ARchive (WAR) file



The Deployment Descriptor

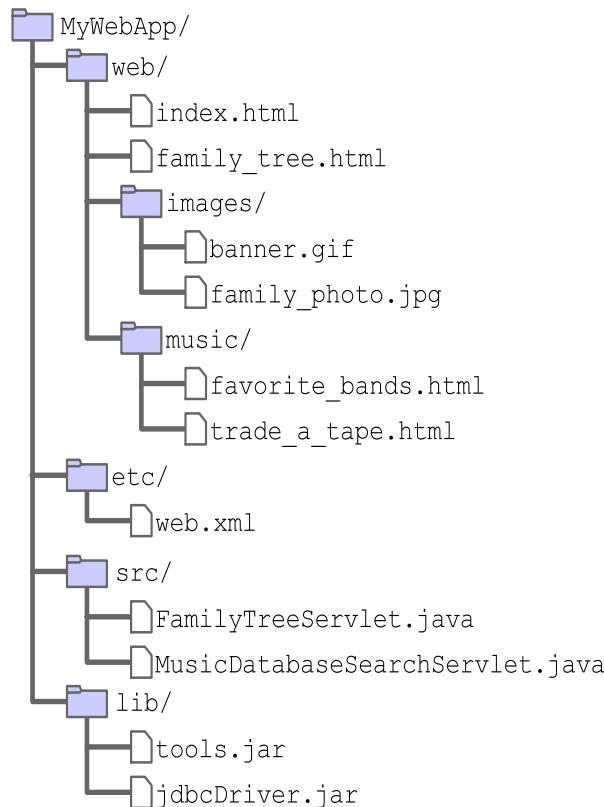
The `web.xml` deployment descriptor file is an XML file using the “web app v2.3” document type definition.

```
1  <?xml version="1.0" encoding="ISO8859_1"?>
2  <!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
   "http://java.sun.com/j2ee/dtds/web-app_2_3.dtd">
3
4  <web-app>
5
6      <display-name>SL-314 WebApp Example</display-name>
7      <description>
8          This Web Application demonstrates a simple deployment descriptor.
9          It also demonstrates a servlet definition and servlet mapping.
10     </description>
11
12     <servlet>
13         <servlet-name>Hello</servlet-name>
14         <servlet-class>sl314.web.FormBasedHello</servlet-class>
15     </servlet>
16
17     <servlet-mapping>
18         <servlet-name>Hello</servlet-name>
19         <url-pattern>/greeting</url-pattern>
20     </servlet-mapping>
21
22 </web-app>
```



A Development Environment

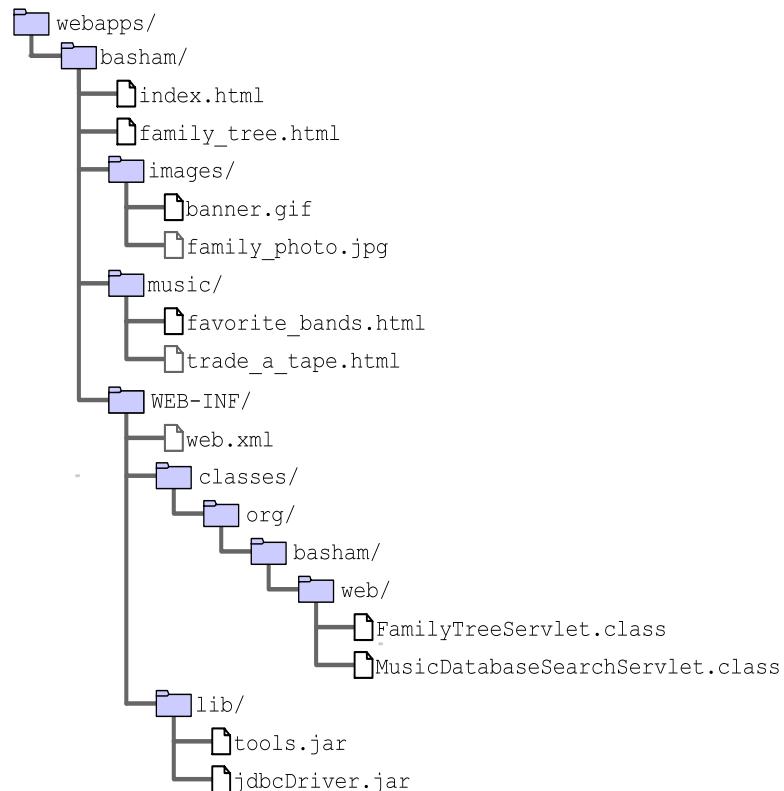
This is one possible directory structure for your development environment:





The Deployment Environment

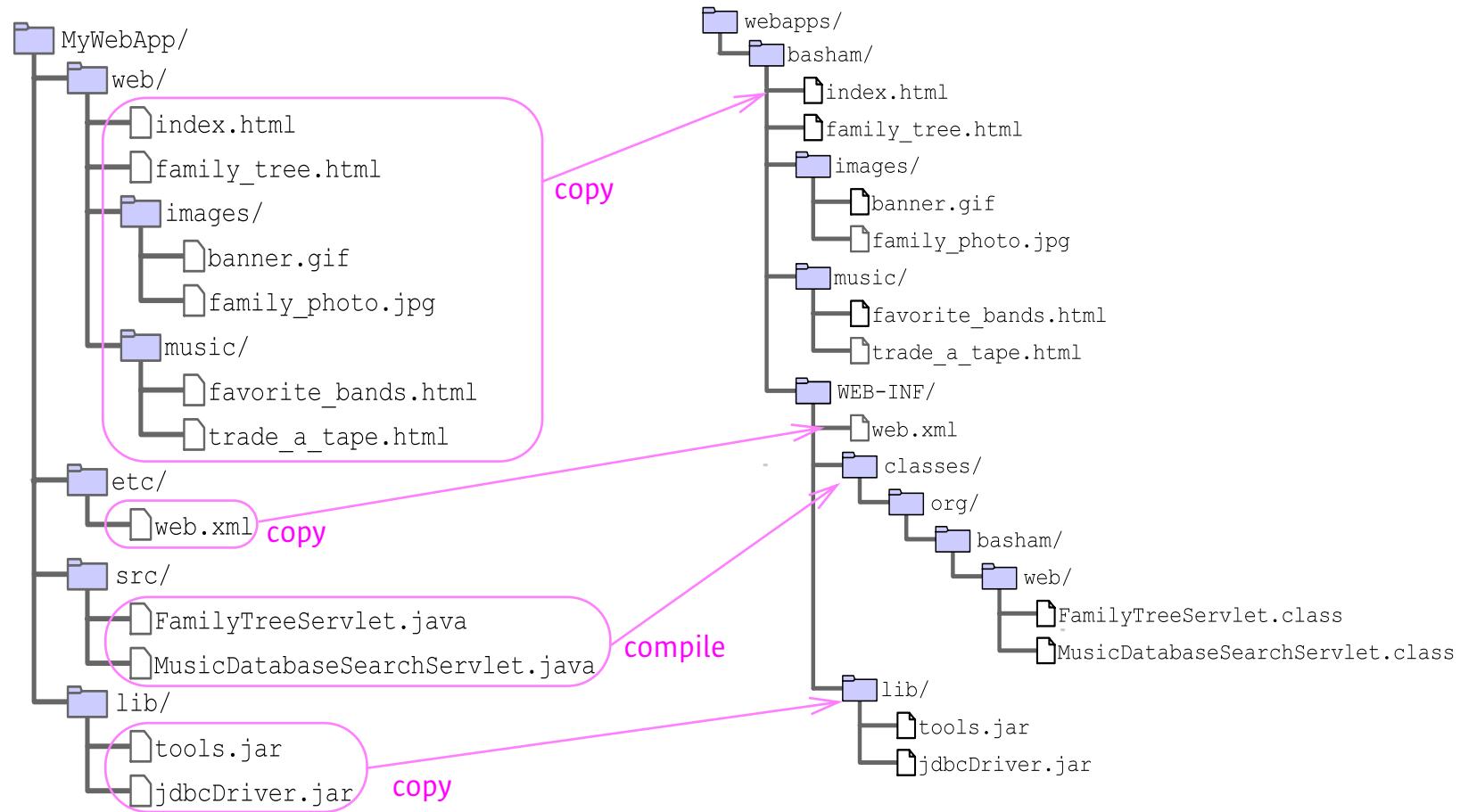
This is the deployment directory structure for a single Web application:





The Deployment Environment

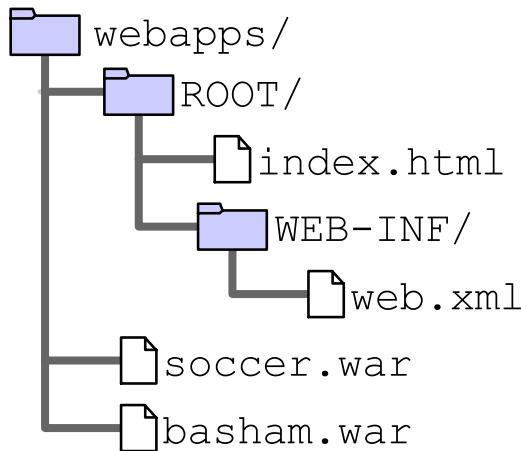
Mapping between the development and deployment:





The Web Archive (WAR) File Format

- A WAR file is a JAR file that contains the same directory structure as the deployed Web application.
- In the Tomcat server, WAR files may be deployed in the webapps / directory.



- Using WAR files makes deployment easier.



Summary

- The Web application deployment descriptor solves several problems:
 - Modularizing Web applications
 - Hiding servlet class names
 - Providing declarative services (such as security)
- The web.xml deployment descriptor is written in XML.
- A development environment usually includes these directories: web, etc, lib, and src.
- The deployment file structure includes: Web files at the top level, WEB-INF, WEB-INF/lib, and WEB-INF/classes.
- Your build tool should copy and compile files from development to deployment.



Module 5

Configuring Servlets

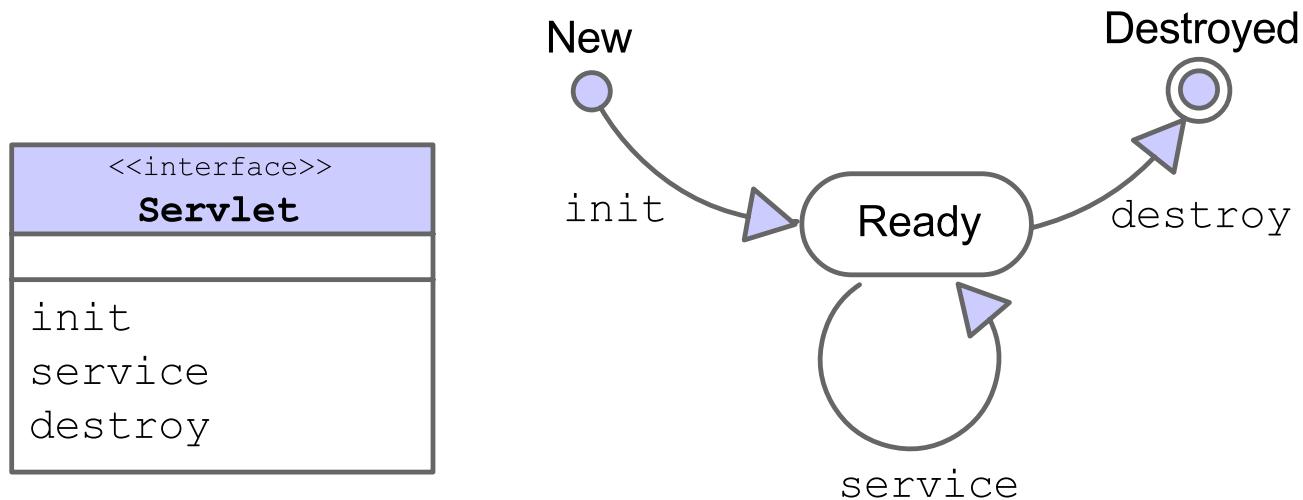


Objectives

- Describe the events in a servlet life cycle and the corresponding servlet API methods
- Describe servlet initialization parameters and their use with individual servlet instances
- Write servlet code to access the configured initialization parameters



Servlet Life Cycle Overview



- The Web container manages the life cycle of a servlet instance. These methods should not be called by your code.
- You can provide an implementation of these methods to manipulate the servlet instance or the resources it depends on.



The init Life Cycle Method

- The `init` method is called by the Web container when the servlet instance is first created.
- The Servlet specification guarantees that no requests will be processed by this servlet until the `init` method has completed.
- Override the `init` method when:
 - You need to create or open any servlet-specific resources that you need for processing user requests
 - You need to initialize the state of the servlet



The service Life Cycle Method

- The service method is called by the Web container to process a user request.
- The HttpServlet class implements the service method by dispatching to doGet, doPost, and so on depending on the HTTP request method (GET, POST, and so on).

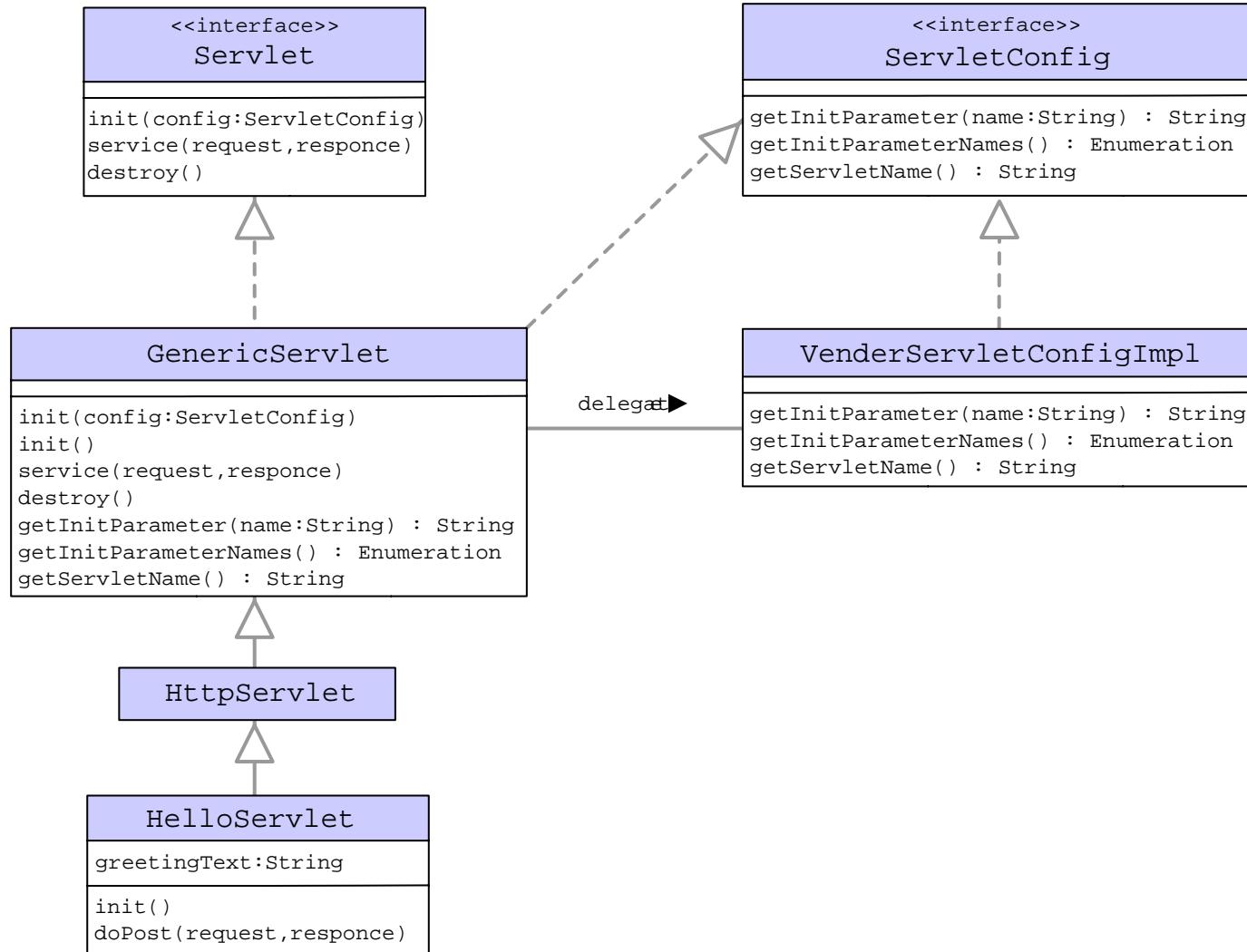


The destroy Life Cycle Method

- The `destroy` method is called by the Web container when the servlet instance is being eliminated.
- The Servlet specification guarantees that all requests will be completely processed before the `destroy` method is called.
- Override the `destroy` method when:
 - You need to release any servlet-specific resources that you had open in the `init` method
 - You need to store the state of the servlet



The ServletConfig API





The ServletConfig API

- The GenericServlet class implements the ServletConfig interface, which provides subclasses with direct access to the config delegate.
- The Web container calls the `init(config)` method, which is handled by the GenericServlet class. This method stores the config object (delegate) and then calls the `init()` method.
- Override the `init()` method in your servlet class. Do not override the `init(config)` method.
- The `getInitParameter` method provides the servlet with access to the initialization parameters for that servlet instance. These parameters are stored in the deployment descriptor.



Initialization Parameters

Deployment Descriptor for a servlet initialization parameter:

```
11   <servlet>
12     <servlet-name>EnglishHello</servlet-name>
13     <servlet-class>sl314.web.HelloServlet</servlet-class>
14     <init-param>
15       <param-name>greetingText</param-name>
16       <param-value>Hello</param-value>
17     </init-param>
18   </servlet>
19
20   <servlet>
21     <servlet-name>FrenchHello</servlet-name>
22     <servlet-class>sl314.web.HelloServlet</servlet-class>
23     <init-param>
24       <param-name>greetingText</param-name>
25       <param-value>Bonjour</param-value>
26     </init-param>
27   </servlet>
```

You can have more than one “servlet definition” (or servlet instance) for a given servlet class.



Initialization Parameters

Servlet accessing an initialization parameter:

```
11 public class HelloServlet extends HttpServlet {  
12  
13     private String greetingText;  
14  
15     public void init() {  
16         greetingText = getInitParameter("greetingText");  
17         // send a message that we have the init param  
18         System.out.println(">> greetingText = '" + greetingText + "'");  
19     }  
}
```

Using the servlet instance variable:

```
37     // Generate the HTML response  
38     out.println("<HTML>");  
39     out.println("<HEAD>");  
40     out.println("<TITLE>Hello Servlet</TITLE>");  
41     out.println("</HEAD>");  
42     out.println("<BODY BGCOLOR='white'>");  
43     out.println("<B>" + greetingText + ", " + name + "</B>");  
44     out.println("</BODY>");  
45     out.println("</HTML>");
```

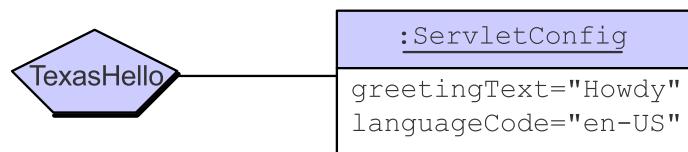


Initialization Parameters

- You can group multiple initialization parameters within a single servlet definition. For example:

```
11 <servlet>
12     <servlet-name>TexasHello</servlet-name>
13     <servlet-class>sl314.web.HelloServlet</servlet-class>
14     <init-param>
15         <param-name>greetingText</param-name>
16         <param-value>Howdy</param-value>
17     </init-param>
18     <init-param>
19         <param-name>languageCode</param-name>
20         <param-value>en-US</param-value>
21     </init-param>
22 </servlet>
```

- The initialization parameters are stored in the `ServletConfig` object that is associated with the servlet instance.





Summary

- The Web container controls the life cycle of a servlet instance:
 - The `init` method is called when the instance is created.
 - The `service` method is called to process all requests to that servlet.
 - The `destroy` method is called when the container wants to remove the servlet from service.
- The `ServletConfig` object stores the initialization parameters that are configured in the deployment descriptor.
- The `getInitParameter` method is used to retrieve initialization parameters.



Module 6

Sharing Resources Using the Servlet Context



Objectives

- Describe the purpose and features of the servlet context
- Develop a context listener that manages a shared Web application resource



The Web Application

- A Web application is a self-contained collection of static and dynamic resources: HTML pages, media files, data and resource files, servlets (and JSP pages), and other auxiliary Java technology classes and objects.
- The Web application deployment descriptor is used to specify the structure and services used by a Web application.
- A ServletContext object is the runtime representation of the Web application.



Duke's Store Web Application

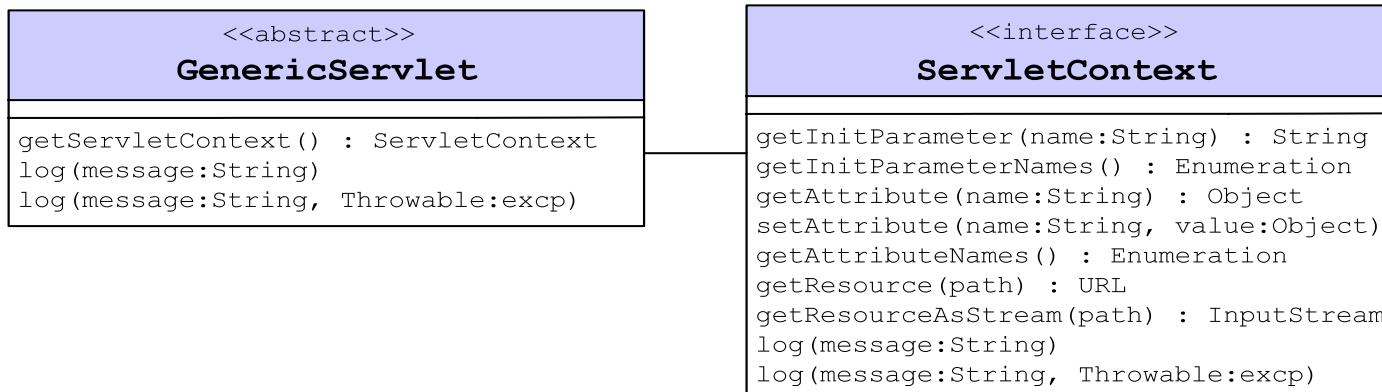
The catalog is a shared resource in the Web application:

The screenshot shows a vintage-style Netscape Communicator window titled "Netscape: Duke's Store Catalog". The menu bar includes File, Edit, View, Go, Communicator, and Help. The location bar shows the URL <http://localhost:8080/context/catalog>. The main content area displays a table titled "Duke's Store Catalog" with three rows of product information. The table has columns for Product Code, Price, Is In Stock?, and Description.

| Product Code | Price | Is In Stock? | Description |
|--------------|-------|--------------|--------------------------------------|
| P-100 | 5.95 | yes | Duke's jeans, extra faded |
| P-101 | 19.95 | yes | Navy blue sweat pants with Duke logo |
| P-102 | 1.95 | yes | "Duke's Wild Ride" bumper sticker |



The ServletContext API



- A servlet has access to the servlet context object through the `getServletContext` method.
- The servlet context object provides:
 - Read-only access to context initialization parameters
 - Read-write access to application-level attributes
 - Read-only access to application-level file resources
 - Write access to the application-level log file



Context Initialization Parameters

Deployment descriptor for a context initialization parameter:

```
3  <web-app>
4
5      <display-name>SL-314 Serlvet Context Example: Duke's Store</display-name>
6      <description>
7          This Web Application demonstrates application-scoped variables (in the
8          servlet context) and WebApp lifecycle listeners.
9      </description>
10
11     <!-- Initialize Catalog -->
12     <context-param>
13         <param-name>catalogFileName</param-name>
14         <param-value>/WEB-INF/catalog.txt</param-value>
15     </context-param>
```

Accessing context initialization parameters in code:

```
18     ServletContext context = sce.getServletContext();
19     String catalogFileName = context.getInitParameter("catalogFileName");
```



Access to File Resources

- The ServletContext object provides read-only access to file resources through the getResourceAsStream method that returns a raw InputStream object.
- Access to text files should be decorated by the appropriate I/O readers.

Accessing resources in code:

```
18 ServletContext context = sce.getServletContext();
19 String catalogFileName = context.getInitParameter("catalogFileName");
20 InputStream is = null;
21 BufferedReader catReader = null;
22
23 try {
24     is = context.getResourceAsStream(catalogFileName);
25     catReader = new BufferedReader(new InputStreamReader(is));
```



Writing to the Web Application Log File

- The `ServletContext` object provides write-only access to the log file:
 - The `log(String)` method writes a message to the log file.
 - The `log(String, Throwable)` method writes a message to the log file and the message and stack trace of the exception or error.
- Web containers must support a separate log file for each Web application.

Writing to the log file in code:

43

```
44     context.log("The ProductList has been initialized.");
```

45



Accessing Shared Runtime Attributes

- The `ServletContext` object provides read-write access to attributes shared across all servlets, through the `getAttribute` and `setAttribute` methods.

Setting attributes in code:

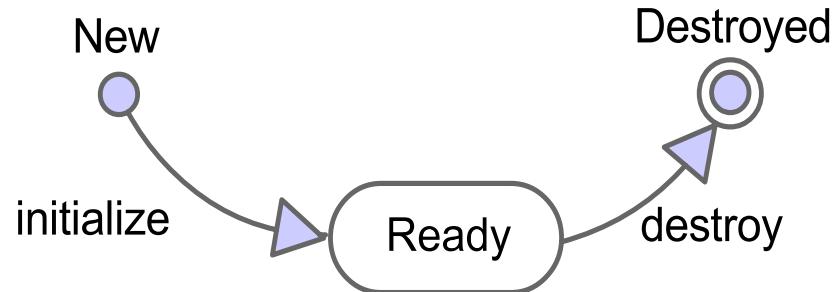
```
41     // Store the catalog as an application-scoped attribute  
42     context.setAttribute("catalog", catalog);
```

Getting attributes in servlet code:

```
19     ProductList catalog = (ProductList) context.getAttribute("catalog");  
20     Iterator items = catalog.getProducts();  
21  
47     while ( items.hasNext() ) {  
48         Product p = (Product) items.next();  
49         out.println("<TR>");  
50         out.println("  <TD>" + p.getProductCode() + "</TD>");  
51         out.println("  <TD>" + p.getPrice() + "</TD>");  
52         out.println("  <TD>" + (p.isInStock() ? "yes" : "no") + "</TD>");  
53         out.println("  <TD>" + p.getDescription() + "</TD>");  
54         out.println("</TR>");  
55     }
```



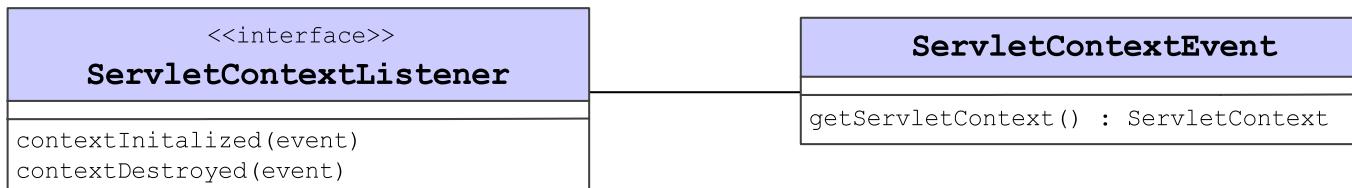
The Web Application Life Cycle



- When the Web container is started, each Web application is initialized.
- When the Web container is shutdown, each Web application is destroyed.
- You can create “listener” objects that are triggered by these events.



The Web Application Lifecycle Listener API



- You can create implementations of the **ServletContextListener** interface to listen to the Web application life cycle events.
- The **ServletContextListener** interface is typically used to initialize heavy-weight, shared resources.



Duke's Store Example

```
15 public class InitializeProductList implements ServletContextListener {  
16  
17     public void contextInitialized(ServletContextEvent sce) {  
18         ServletContext context = sce.getServletContext();  
19         String catalogFileName = context.getInitParameter("catalogFileName");  
20         InputStream is = null;  
21         BufferedReader catReader = null;  
22  
23         try {  
24             is = context.getResourceAsStream(catalogFileName);  
25             catReader = new BufferedReader(new InputStreamReader(is));  
26             String productString;  
27             ProductList catalog = new ProductList();  
28  
29             // Parse the catalog file to construct the product list  
30             while ( (productString = catReader.readLine()) != null ) {  
31                 StringTokenizer tokens = new StringTokenizer(productString, " | ");  
32                 String code = tokens.nextToken();  
33                 String price = tokens.nextToken();  
34                 String quantityStr = tokens.nextToken();  
35                 int quantity = Integer.parseInt(quantityStr);  
36                 String description = tokens.nextToken();  
37                 Product p = new Product(code, price, quantity, description);  
38                 catalog.addProduct(p);  
39             }  
40  
41             // Store the catalog as an application-scoped attribute  
42             context.setAttribute("catalog", catalog);  
43         }  
44     }  
45 }
```



Configuring Servlet Context Listeners

Deployment descriptor for registering a servlet context listener:

```
3   <web-app>
4
5     <display-name>SL-314 Serlvet Context Example: Duke's Store</display-name>
6     <description>
7       This Web Application demonstrates application-scoped variables (in the
8       servlet context) and WebApp lifecycle listeners.
9     </description>
10
11    <!-- Initialize Catalog -->
12    <context-param>
13      <param-name>catalogFileName</param-name>
14      <param-value>/WEB-INF/catalog.txt</param-value>
15    </context-param>
16    <listener>
17      <listener-class>sl314.dukestore.InitializeProductList</listener-class>
18    </listener>
19    <!-- END: Initialize Catalog -->
```



Summary

The `ServletContext` object reflects the Web application as a runtime object:

- It provides access to initialization parameters using the `getInitParameter` method.
- It provides access to file resources using the `getResourceAsStream` method.
- It provides logging using the `log(message)` and `log(message, exception)` methods.
- It provides access to shared attributes using the `getAttribute` and `setAttribute` methods.

The `ServletContextListener` interface provides you with a tool to respond to Web application life cycle events.



Module 7

Developing Web Applications Using the MVC Pattern



Objectives

- List the limitations of a “simple” Web application
- Develop a Web application using a variation on the Model-View-Controller pattern

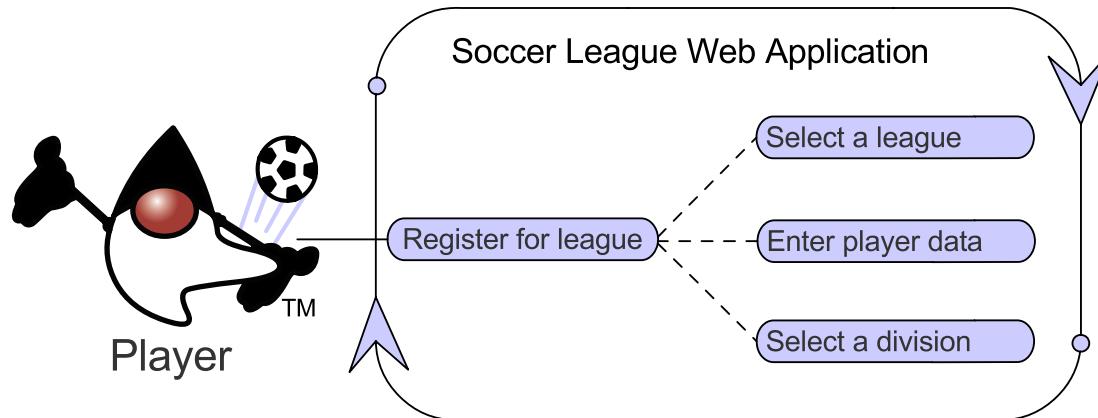


Activities of a Web Application

- Initialize all shared data and resources
- Service each HTTP request:
 - Verify HTML form data (if any)
 - Send an Error page if data fails verification checks
 - Process the data, which may store persistent information
 - Send an Error page if processing fails
 - Send a Response page if processing succeeds



The Soccer League Example

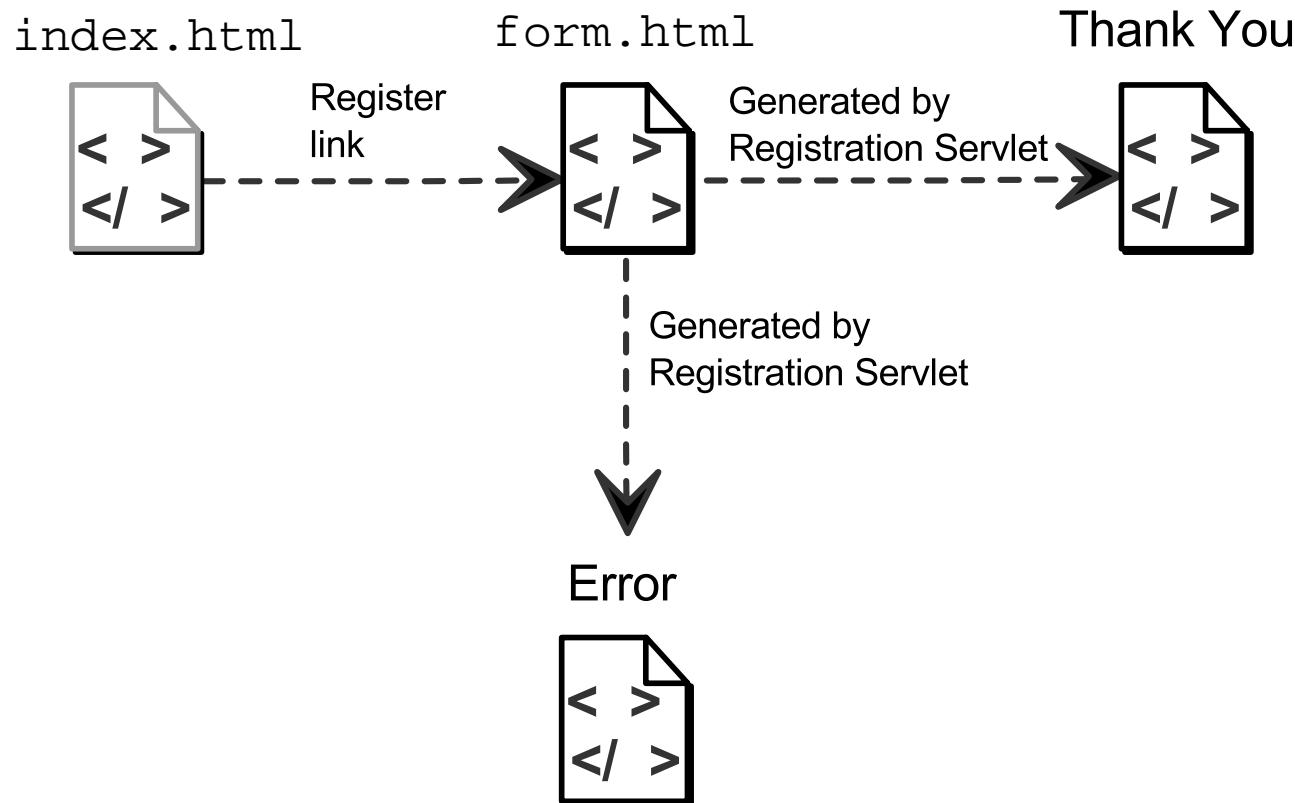


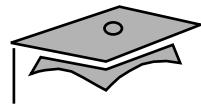
- Initialize the set of existing leagues
- Each request:
 - Verifies the league, player, and division data in the HTML form
 - Stores the player and registration information
 - Sends the appropriate page (Error or Thank You)



The Soccer League Example

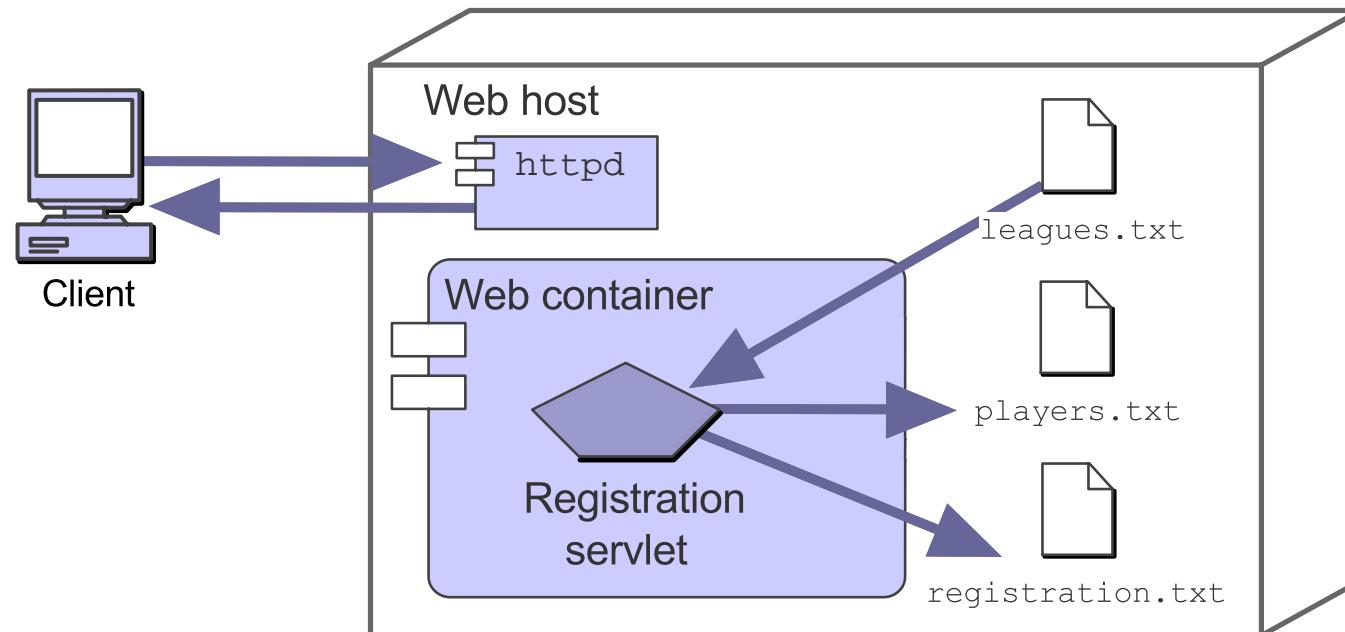
Progression from page to page:





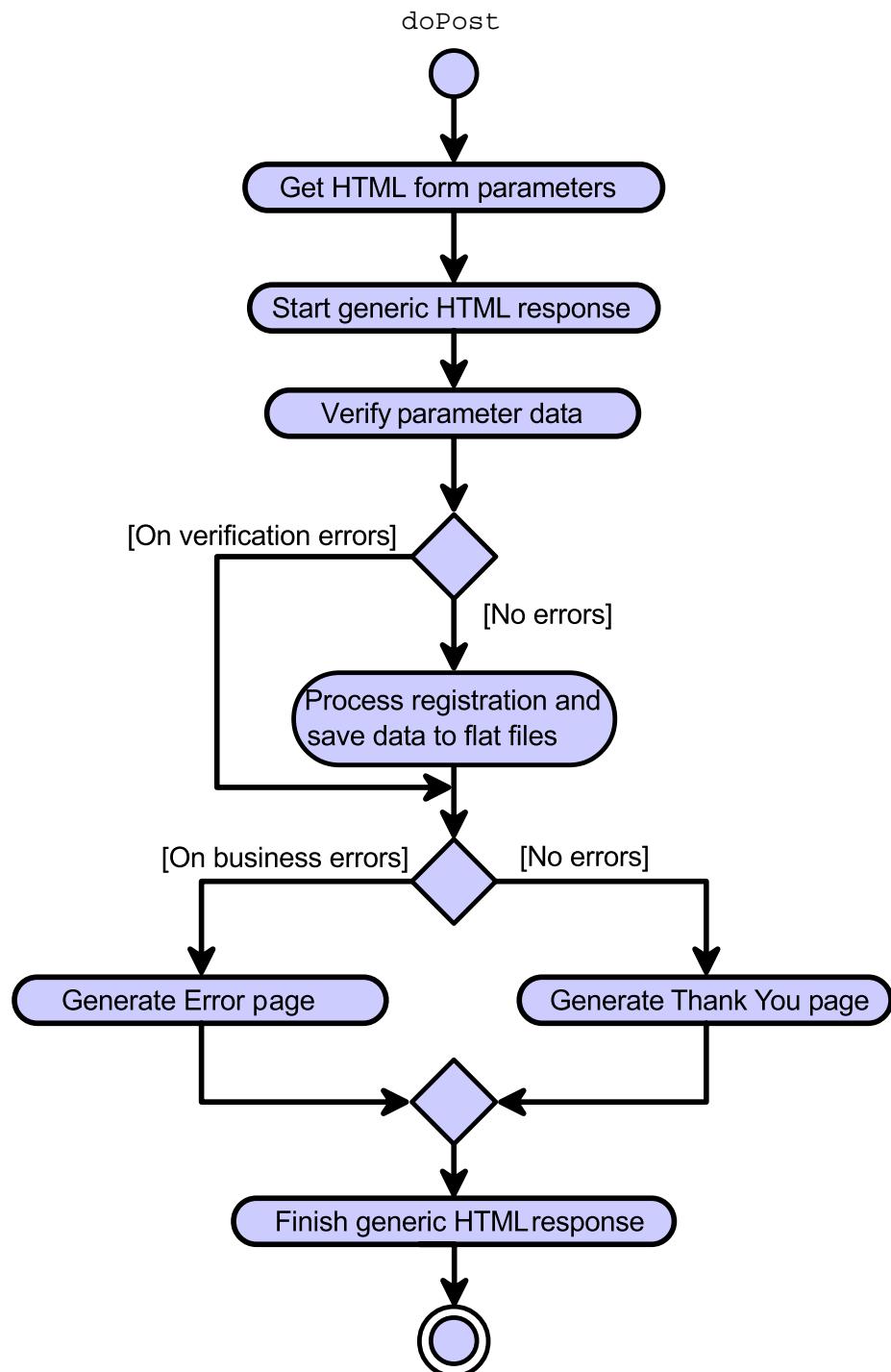
The Soccer League Example

Deployment diagram:



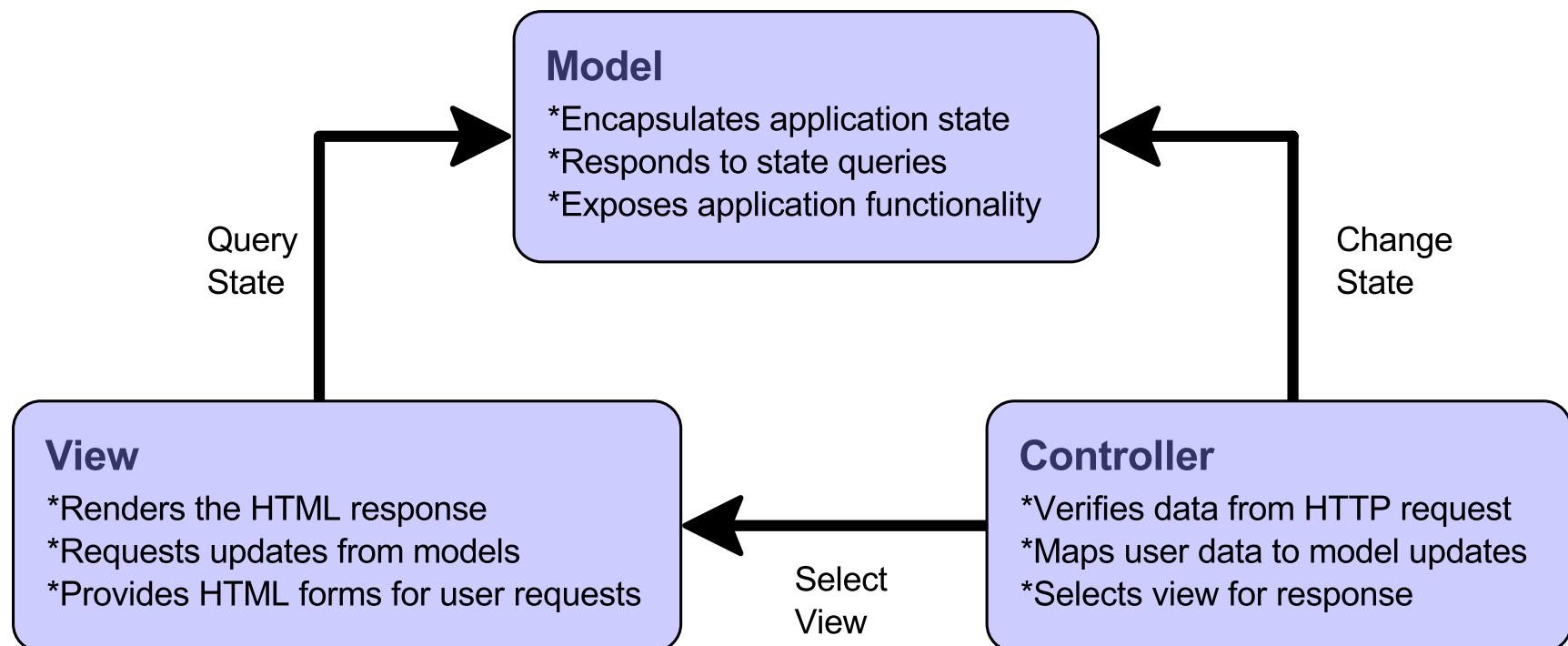


Activity Diagram of the Soccer League Example





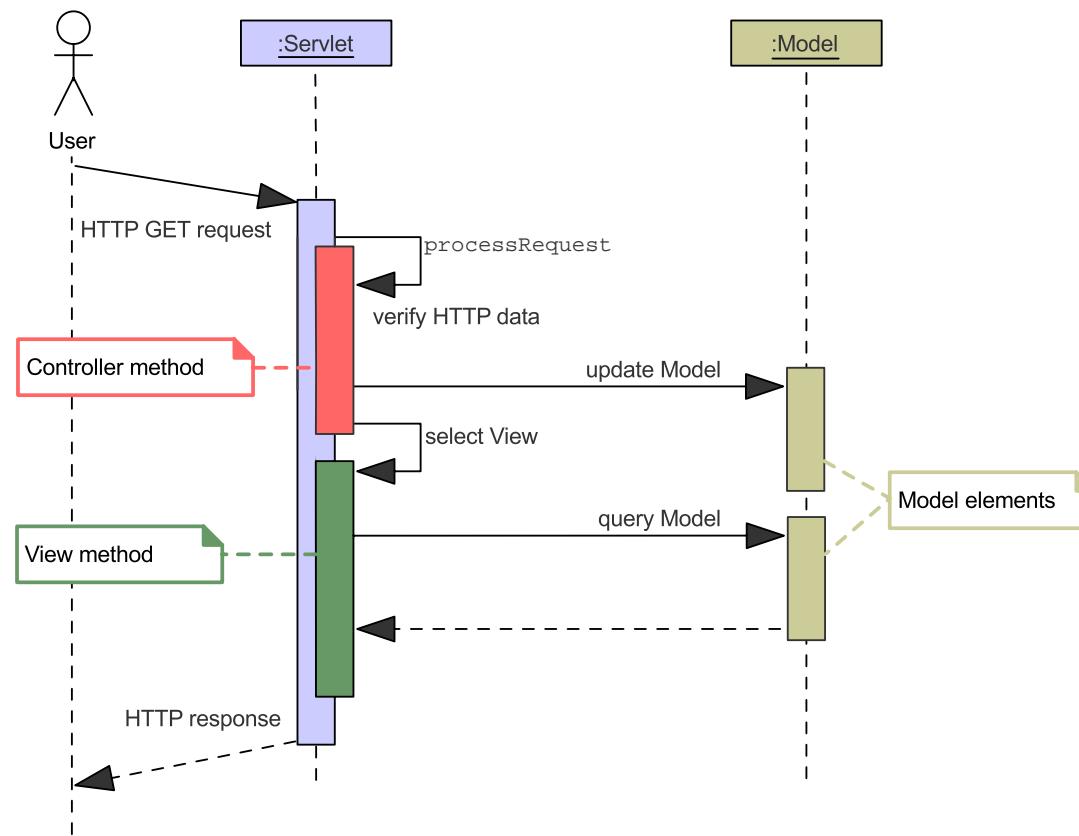
Model-View-Controller for a Web Application





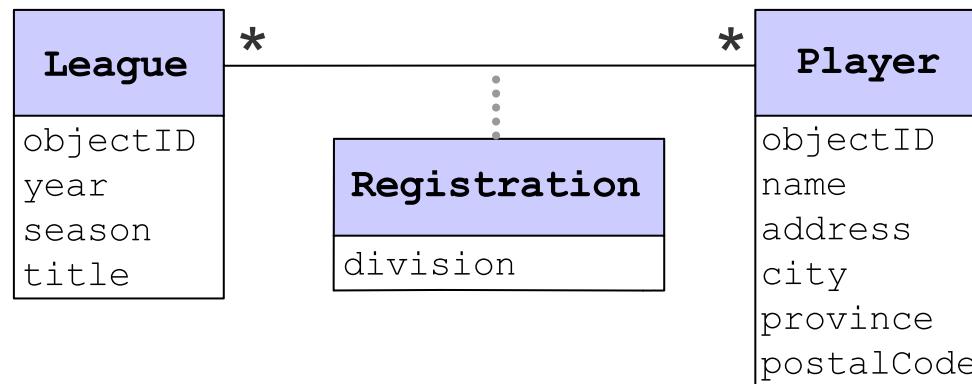
Sequence Diagram of MVC in the Web Tier

This diagram shows the time-based relationships between the servlet Controller/View methods and the Model.





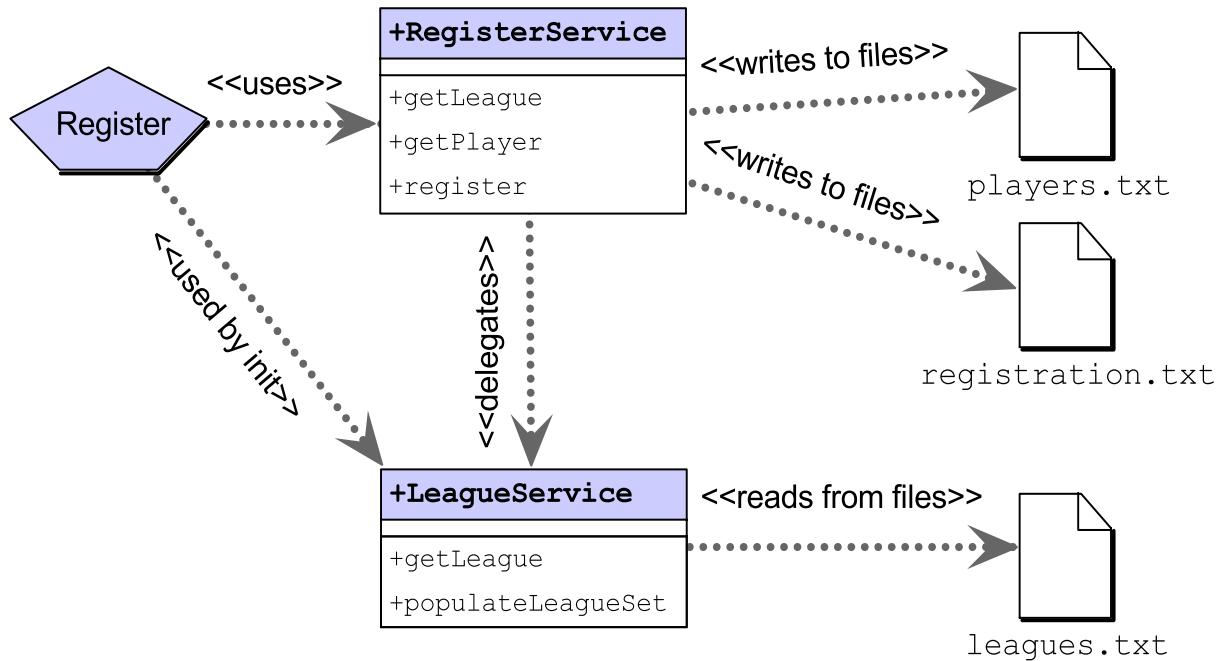
Soccer League Application: The Domain Model



- Domain objects encapsulate entities in the application domain; for example, leagues and players.
- Multiple leagues may exist.
- A player may register for any number of leagues.
- Registration includes the league, the player, and the division for which the player is registering.



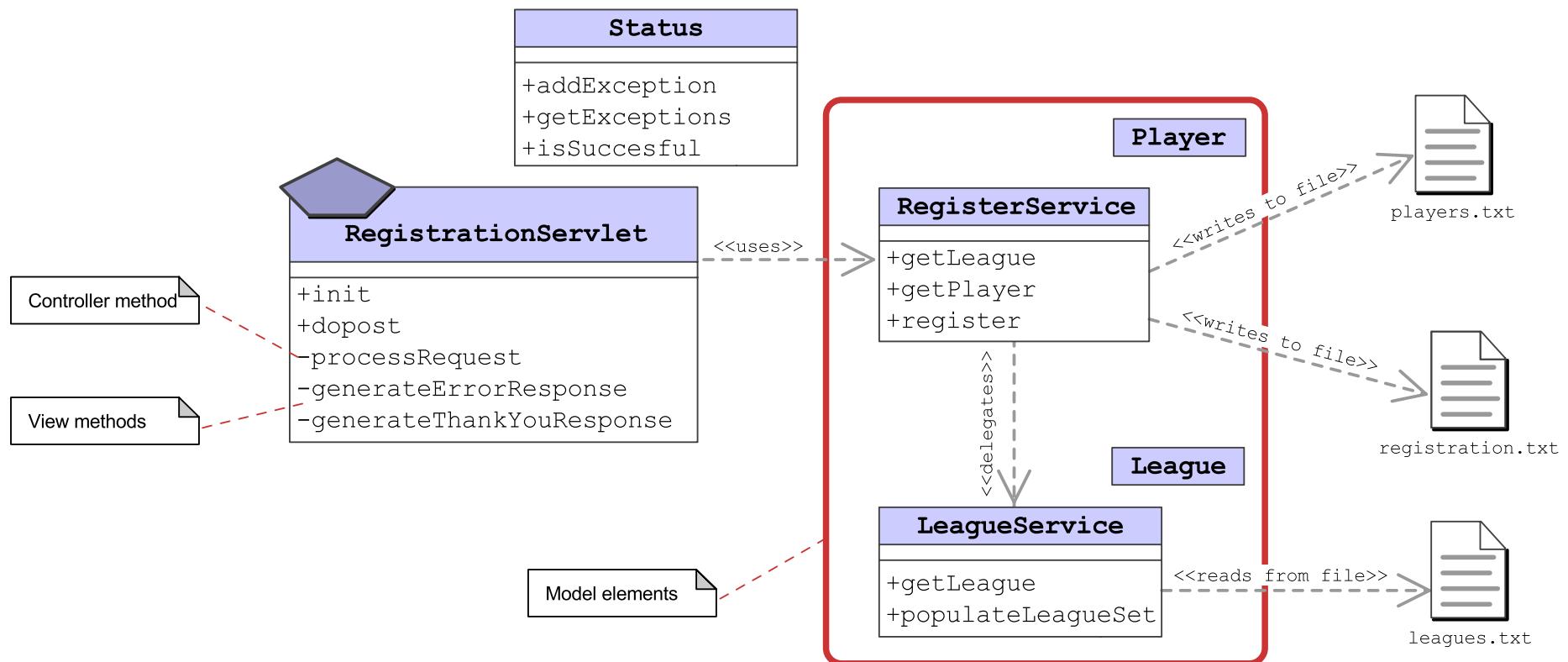
Soccer League Application: The Services Model



Services manipulate domain objects and perform persistence operations.



Soccer League Application: The Big Picture





Soccer League Application: The Controller

- The `processRequest` method of the `RegistrationServlet` acts as the controller.
- The Controller conditionally determines a View. For example, if a verification error occurs the Error page View is generated:

```
77 League league = regService.getLeague(year, season);
78 if ( league == null ) {
79     status.addException(
80         new Exception("The league you selected does not yet exist;
81                         + " please select another."));
82 }
83
84 // If any of the above verification failed, then return the
85 // 'Error Page' View and return without proceeding with the
86 // rest of the business logic
87 if ( ! status.isSuccessful() ) {
88     generateErrorResponse(request, response);
89     return;
90 }
```



Soccer League Application: The Controller

- The Controller passes data to the View method using a request attribute:

```
101    regService.register(league, player, division);
102    request.setAttribute("league", league);
103    request.setAttribute("player", player);
104
105    // The registration process was successful,
106    // Generate the 'Thank You' View
107    generateThankYouResponse(request, response);
```



Soccer League Application: The Views

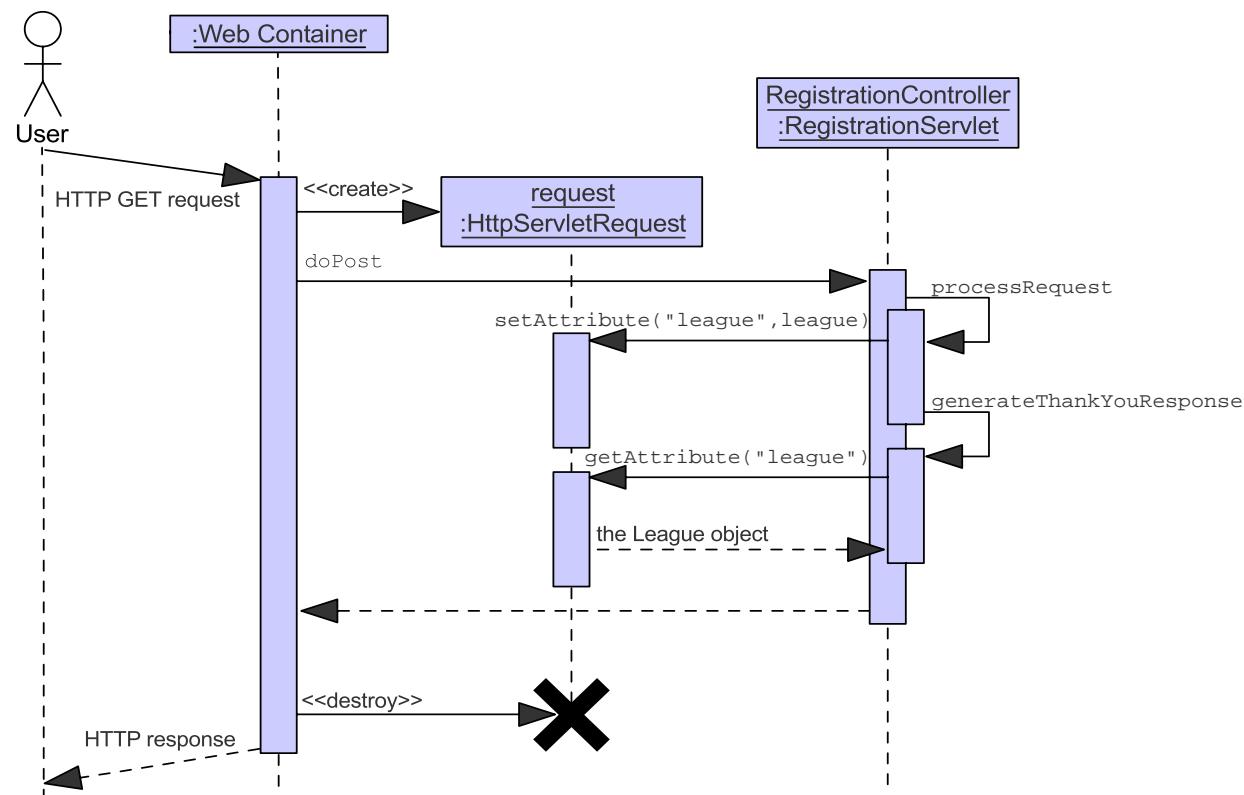
- The generateThankYouResponse method of the RegistrationServlet generates the Thank You View.
- This method retrieves data from the Controller method using the request attributes:

```
120 public void generateThankYouResponse(HttpServletRequest request,
121                                     HttpServletResponse response)
122     throws IOException {
123
124     // Specify the content type is HTML
125     response.setContentType("text/html");
126     PrintWriter out = response.getWriter();
127
128     League league = (League) request.getAttribute("league");
129     Player player = (Player) request.getAttribute("player");
```



The Request Scope

The Controller method of the servlet communicates with the View methods using the request object scope.





Summary

Web applications can use a modified version of the Model-View-Controller design pattern to cleanly separate “roles and responsibilities.”

- The Model represents the business services and domain objects.
- The Controller takes user actions (an HTTP request) and processes the event by manipulating the Model and selecting the next View. The Controller passes domain objects to the View using the `setAttribute` method on the request object.
- The Views of the Web application are a rendering of the user’s state within the application. The View retrieves domain objects using the `getAttribute` method.



Module 8

Developing Web Applications Using Session Management



Objectives

- Describe the purpose of session management
- Design a Web application that uses session management
- Develop servlets using session management
- Describe the cookies implementation of session management
- Describe the URL-rewriting implementation of session management



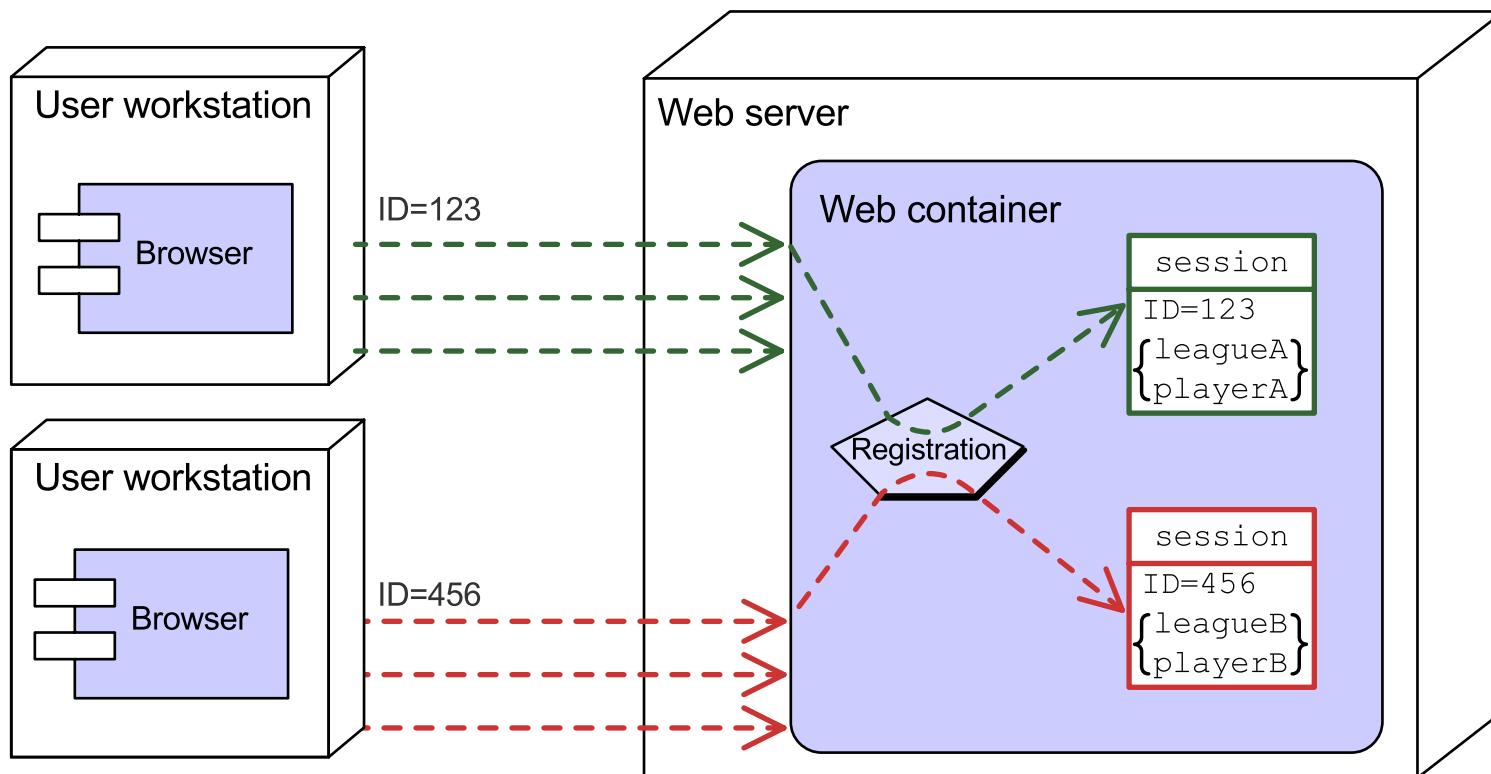
HTTP and Session Management

- HTTP is stateless. Each request and response message connection is independent of all others.
- This is significant because from one request to another (from the same user) the HTTP server forgets the previous request.
- Therefore, the Web container must create a mechanism to store session information for a particular user.



Sessions in a Web Container

The Web container can keep a “session object” for each user:





Web Application Design Using Session Management

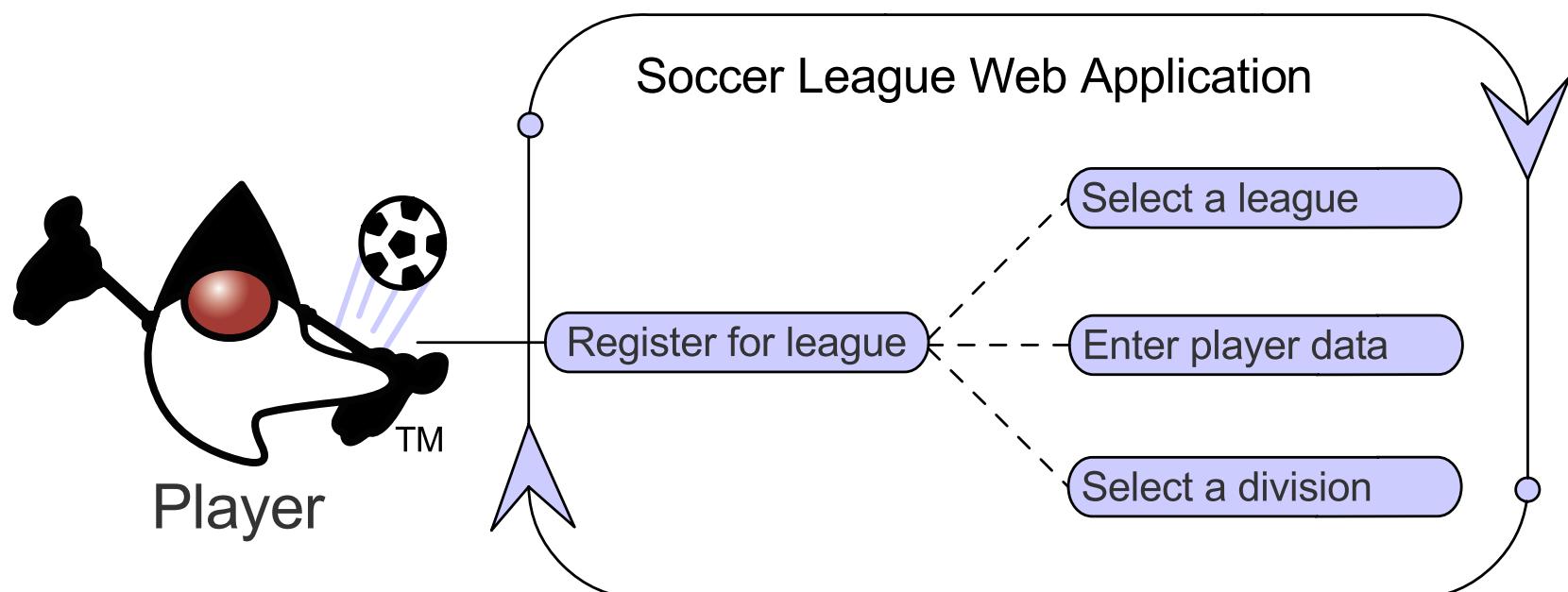
This is *just one* technique for designing Web applications.
There are three steps to this design process:

1. Design multiple, interacting views for a Use Case.
2. Create one servlet controller for every Use Case in your Web application.
3. Use a hidden HTML parameter to indicate which action is being performed at a given stage in the session.



Example: Registration Use Case

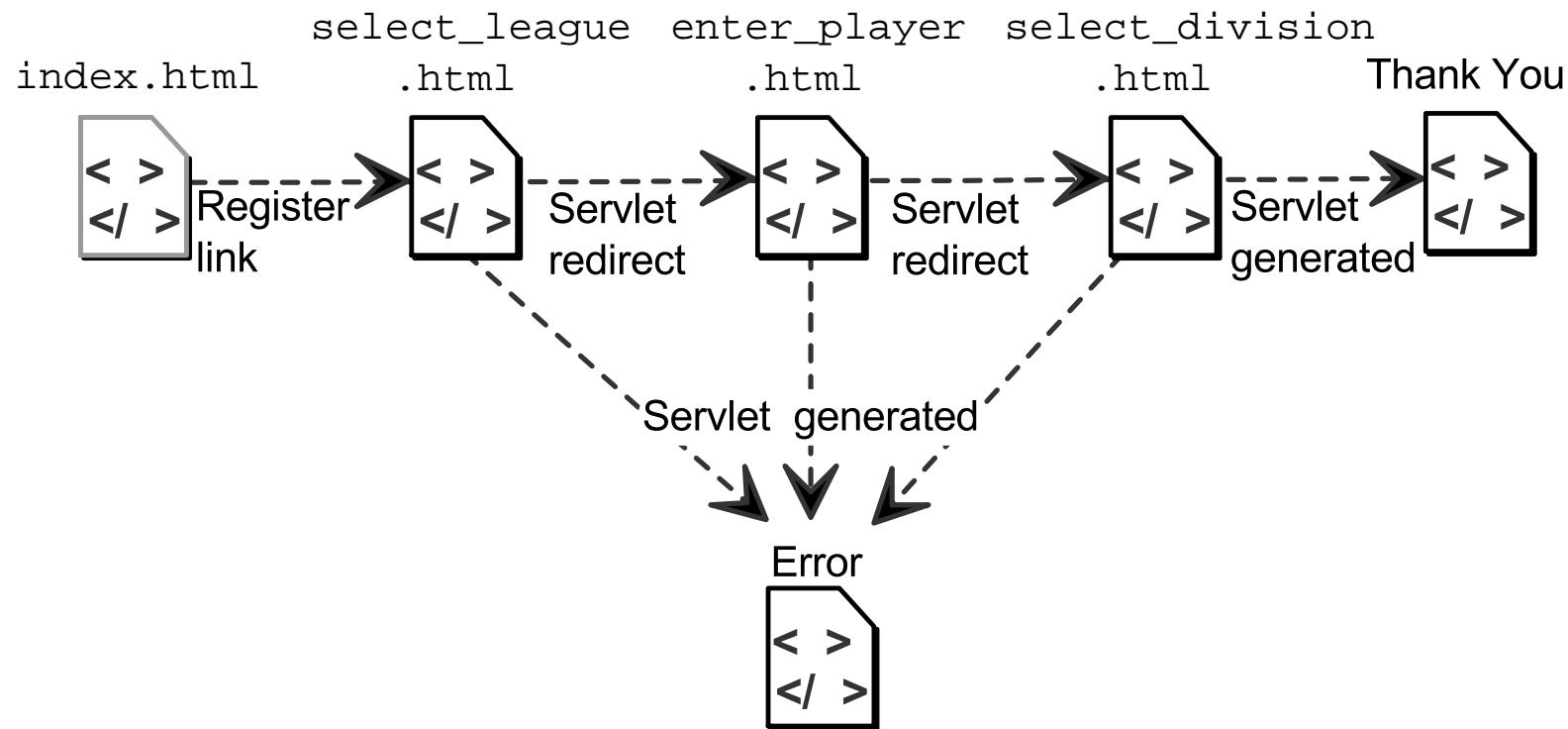
This is the Use Case for on-line league registration:





Example: Multiple Views for Registration

This is the Web page flow for on-line registration:





Example: Enter League Form

This is an excerpt from the select_league.html Web page:

```
23 <FORM ACTION="register" METHOD="POST">  
24  
25 <INPUT TYPE="hidden" NAME="action" VALUE="EnterPlayer">
```

The doPost method dispatches to the control method based on the action parameter:

```
24     public void doPost(HttpServletRequest request,  
25                         HttpServletResponse response)  
26         throws IOException {  
27  
28     String action = request.getParameter("action");  
29  
30     if ( action.equals(ACTION_SELECT_LEAGUE) ) {  
31         processSelectLeague(request, response);  
32  
33     } else if ( action.equals(ACTION_ENTER_PLAYER) ) {  
34         processEnterPlayer(request, response);  
35  
36     } else if ( action.equals(ACTION_SELECT_DIVISION) ) {  
37         processSelectDivision(request, response);  
38     }  
39 }
```

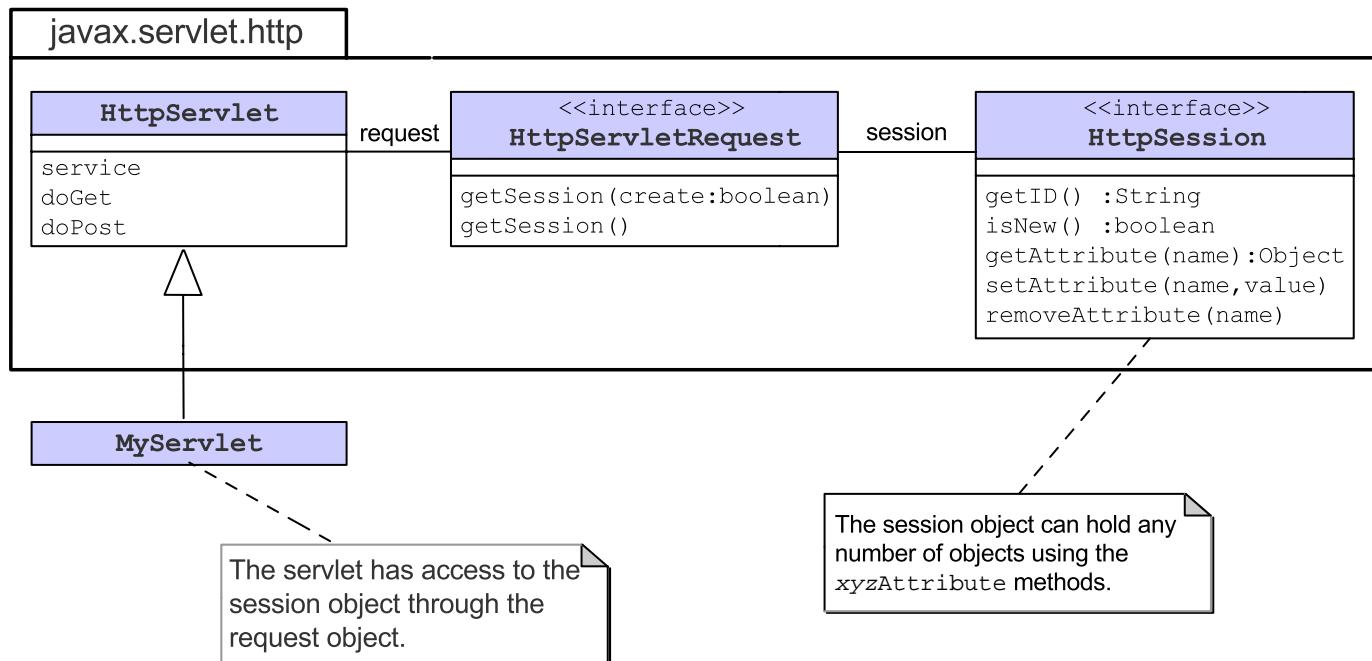


Web Application Development Using Session Management

- Each control method must store attributes (name/object pairs) that are used by other requests within the session. For example, the `processSelectLeague` method must store the league object for later use.
- Any control method can access an attribute that has already been set by processing a previous request. For example, the `processSelectDivision` method must access the league and player objects to complete the registration process.
- At the end of the session, the servlet *might* destroy the session object.



The Session API



- Your servlet accesses the session object through the request object.
- You can store and access any number of objects in the session object.



Retrieving the Session Object

The `processSelectLeague` method retrieves the session object:

```
41  /**
42   * This method performs the "Select League" action
43   */
44  public void processSelectLeague(HttpServletRequest request,
45                                HttpServletResponse response)
46          throws IOException {
47      // Create the HttpSession object
48      HttpSession session = request.getSession();
```



Storing Session Attributes

The processSelectLeague method:

- Looks up the league object:

```
70 // Retrieve the league object
71 League league = registerSvc.getLeague(year, season);
72 if ( league == null ) {
73     status.addException(
74         new Exception("The league you selected does not yet exist;
75                         + " please select another."));
76 }
```

- Stores it in the league attribute in the session object:

```
84 // Store the league object in the session
85 session.setAttribute("league", league);
```

- Selects the next View using the redirect method:

```
87 // Select the next View: "Enter Player" form
88 response.sendRedirect("enter_player.html");
```



Accessing Session Attributes

The `processSelectDivision` method retrieves the league and player objects from the session:

```
149 public void processSelectDivision(HttpServletRequest request,
150                               HttpServletResponse response)
151     throws IOException {
152     // Retrieve the HttpSession object
153     HttpSession session = request.getSession();
154
155     // Retrieve the domain object from the session
156     League league = (League) session.getAttribute("league");
157     Player player = (Player) session.getAttribute("player");
```



Accessing Session Attributes

View methods might also:

- Access session attributes:

```
203 public void generateThankYouResponse(HttpServletRequest request,
204                               HttpServletResponse response)
205     throws IOException {
206     // Retrieve the HttpSession object
207     HttpSession session = request.getSession();
208
209     // Specify the content type is HTML
210     response.setContentType("text/html");
211     PrintWriter out = response.getWriter();
212
213     League league = (League) session.getAttribute("league");
214     Player player = (Player) session.getAttribute("player");
```

- Generate a dynamic response using the attributes:

```
230     out.println("<BR>");
231     out.println("Thank you, " + player.getName() + ", for registering");
232     out.println("in the <B>" + league.getTitle() + "</B> league.");
233     out.println();
234     out.println("</BODY>");
235     out.println("</HTML>");
```

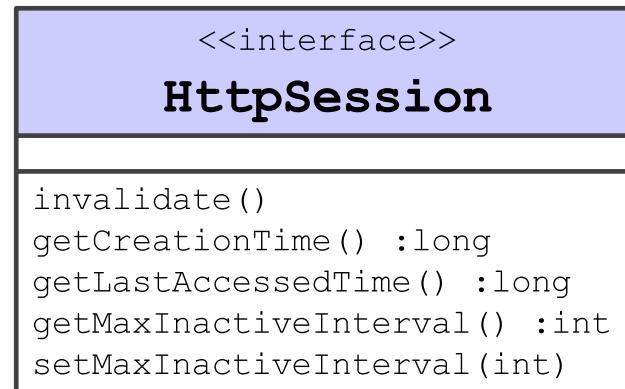


Destroying the Session

- You can control the lifespan of all sessions using the deployment descriptor:

```
35 <session-config>
36     <session-timeout>10</session-timeout>
37 </session-config>
```

- You can control the lifespan of a specific session object using the following APIs:





Destroying the Session

- Session objects can be shared across multiple servlets (for different Use Cases) within the same Web application.
- Session objects are not shared across multiple Web applications within the same Web container.
- Destroying a session using the `invalidate` method might cause disruption to other servlets (or Use Cases).



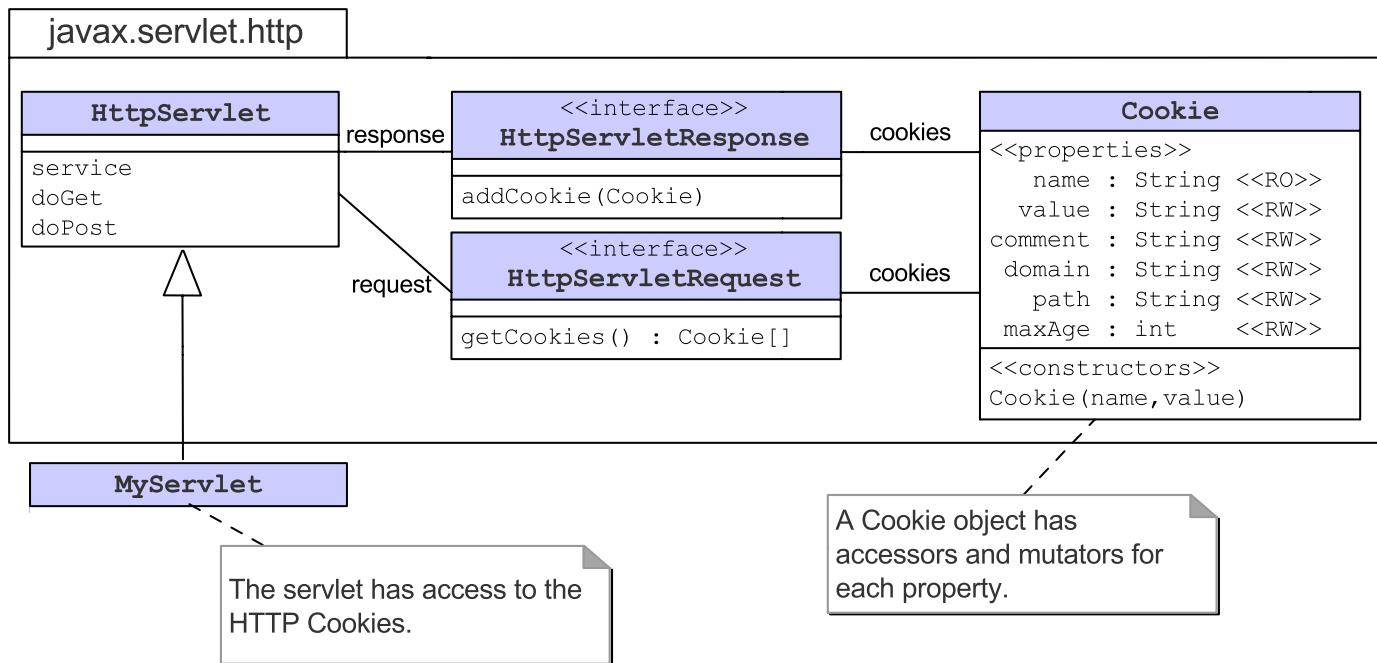
Session Management Using Cookies

IETF RFC 2109 creates an extension to HTTP to allow a Web server to store information on the client machine:

- 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 lifespan and will be flushed by the client browser at the end of that lifespan.



The Cookie API



- Cookies can be stored on the client computer by adding them to the response object.
- Cookies can be retrieved from the request object.



Using Cookies

- The code to store a Cookie in the response:

```
String name = request.getParameter("firstName");  
Cookie c = new Cookie("yourname", name);  
response.addCookie(c);
```

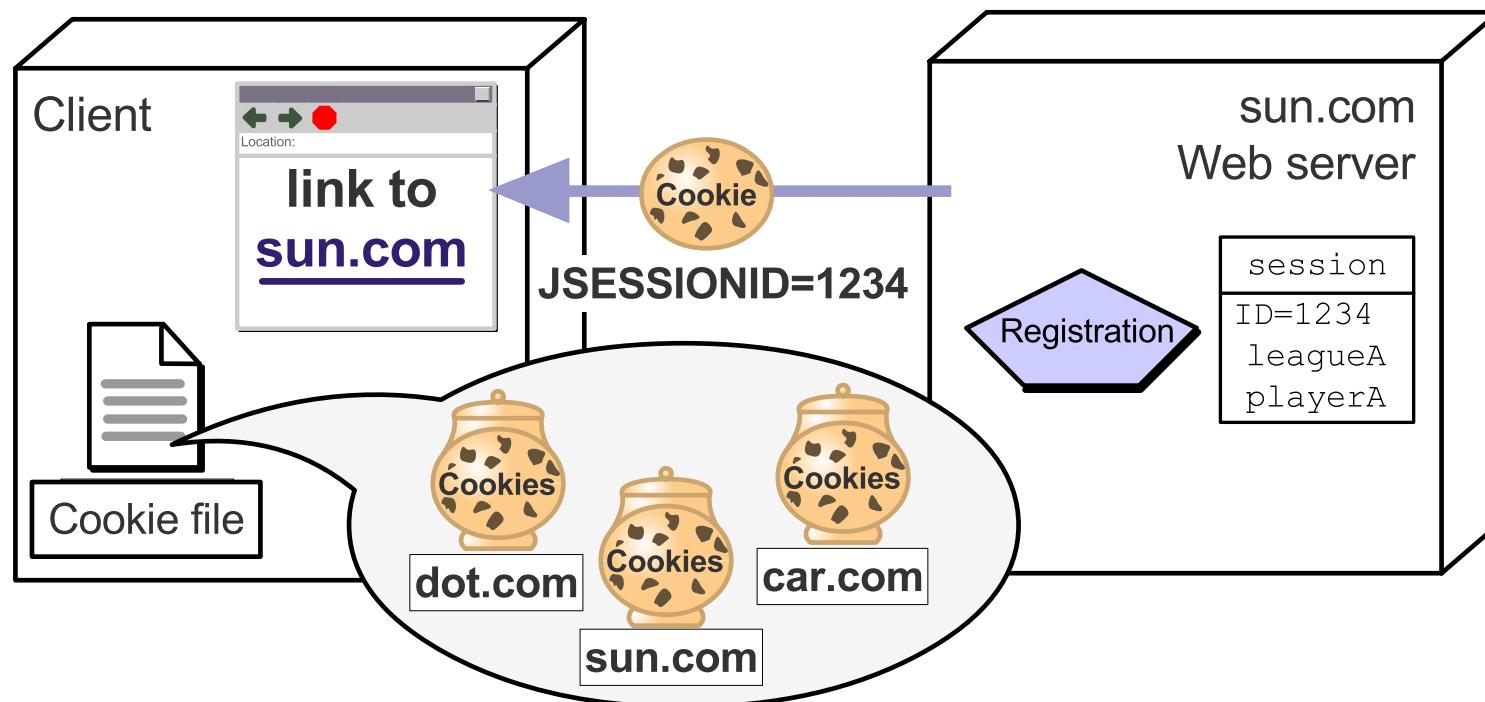
- The code to retrieve a Cookie from the request:

```
Cookie[] allCookies = request.getCookies();  
for ( int i=0; i < allCookies.length; i++ ) {  
    if ( allCookies[i].getName().equals("yourname") ) {  
        name = allCookies[i].getValue();  
    }  
}
```



Using Cookies for Session Management

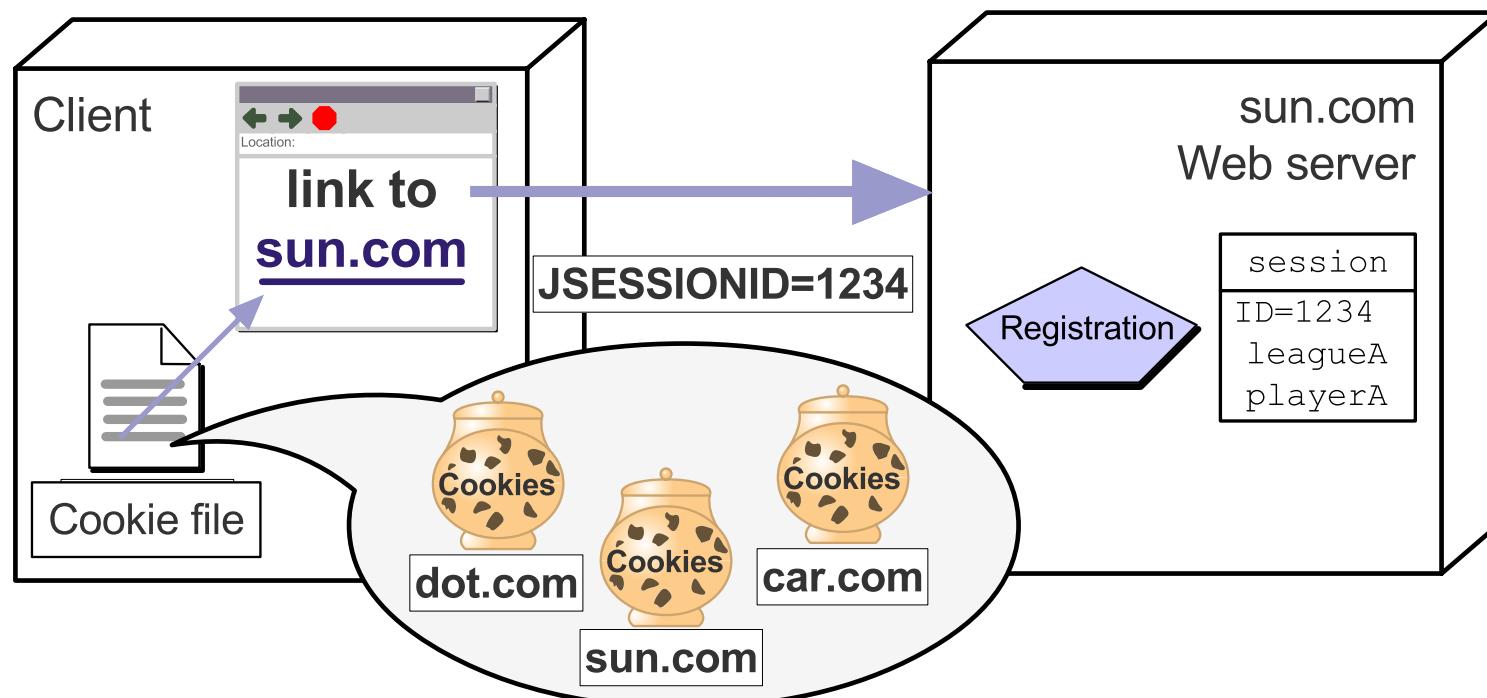
The Web container sends a JSESSIONID Cookie to the client:





Using Cookies for Session Management

The JSESSIONID Cookie is sent in all subsequent requests:





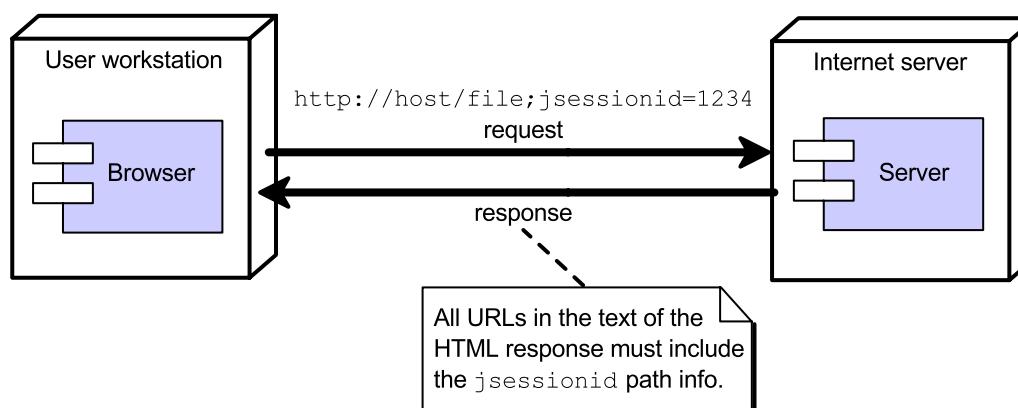
Using Cookies for Session Management

- The Cookie mechanism is the default HttpSession strategy.
- There is nothing special that you code in your servlets to make use of this session strategy.
- Unfortunately, some users turn off Cookies on their browsers.



Session Management Using URL-Rewriting

- URL-rewriting is used when Cookies cannot be used.
- Client appends extra data on the end of each URL.
- Server associates that identifier with data it has stored about that session.
- With this URL:
`http://host/path/file;jsessionid=123`
session information is `jsessionid=123`.





Implications of Using URL-Rewriting

- Every HTML page that participates in a session (using URL-rewriting) must include the session ID in all URLs in those pages; this requires dynamic generation.
- Use the `encodeURL` method on the response object to guarantee that the URLs include the session ID information.
- For example, in the `generateEnterPlayer` method the ACTION attribute on the FORM tag must be encoded:

```
234  out.println("<FORM ACTION=' " + response.encodeURL("register") + "' METHOD='POST'>");  
235  out.println(" ");  
236  out.println("<INPUT TYPE='hidden' NAME='action' VALUE='EnterPlayer'>");
```



Guidelines for Working With Sessions

- A servlet must create a session.
- A servlet can determine if a session already exists:
 - Use `getSession(false)` and check for a null return.
 - Use `getSession(true)` and call `isNew` on the resulting session.
- Any servlet can request that a session be created. A session is shared among all servlets in a Web application.
- Session attributes should be named to avoid ambiguity.
- Sessions can be invalidated and become unusable.
- A session can time out due to browser inactivity.



Summary

- HTTP is a stateless protocol. The servlet API provides a session API.
- Use either of the `getSession` methods on the request object to access (or create) the session object.
- Use the `setAttribute` method on the session object to store one or more name-object pairs.
- Use the `getAttribute` method on the session object to retrieve an attribute.
- Use the `invalidate` method on the session object to destroy a session. You can also use the Web container to destroy the session using timeout declared in the deployment descriptor.



Module 9

Handling Errors in Web Applications



Objectives

- Describe the types of errors that can occur in a Web application
- Declare an HTTP error page using the Web application deployment descriptor
- Declare a Java technology exception error page using the Web application deployment descriptor
- Develop an error handling servlet
- Write servlet code to capture a Java technology exception and forward it to an error handling servlet
- Write servlet code to log exceptions

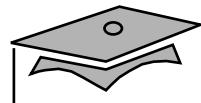


HTTP Error Codes

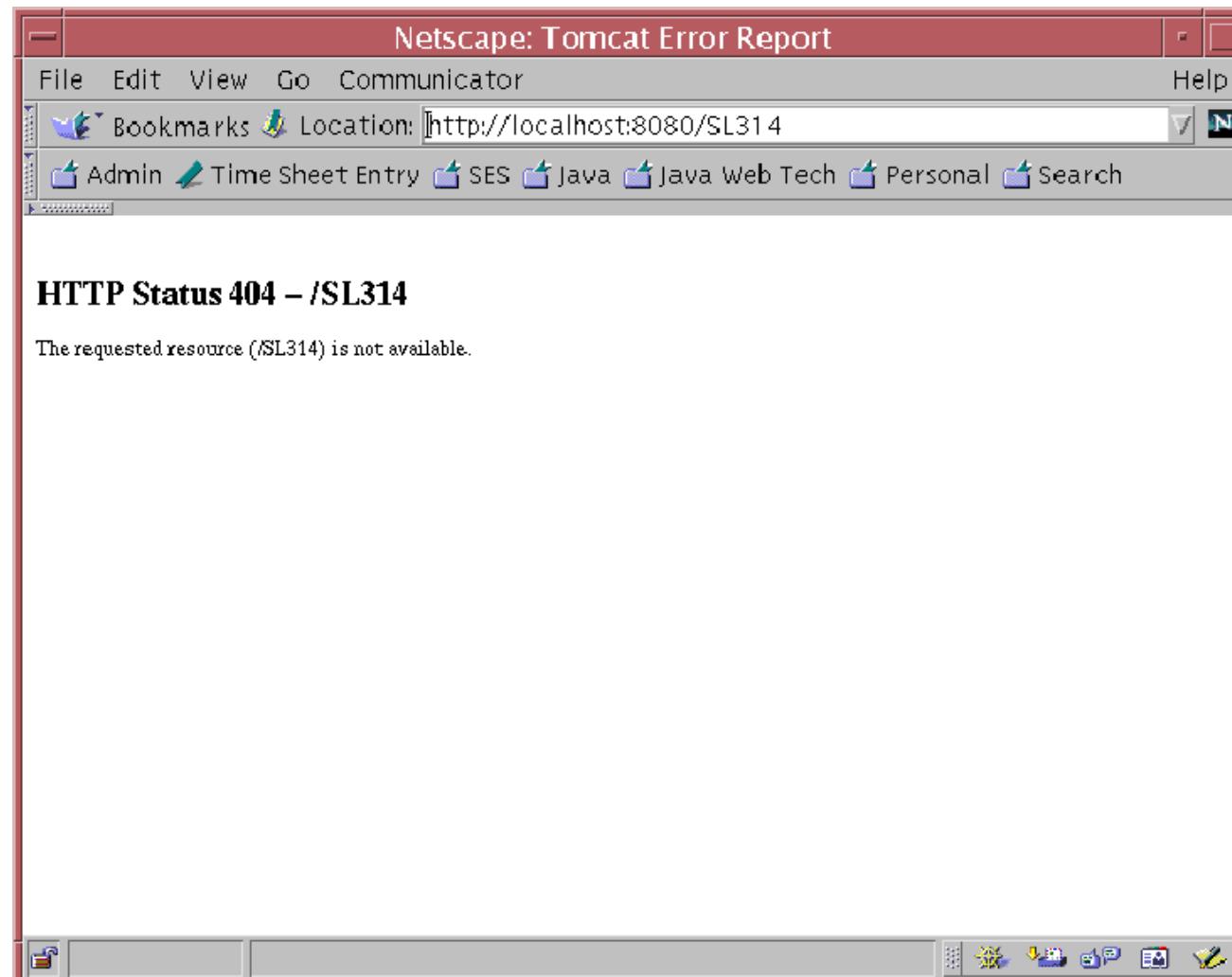
- An HTTP response might indicate a server error using a status code in the 400–500 range. Here are a few examples:

| | |
|-----|------------------------|
| 400 | Bad Request |
| 401 | Unauthorized |
| 404 | Not Found |
| 405 | Method Not Allowed |
| 415 | Unsupported Media Type |
| 500 | Internal Server Error |
| 501 | Not Implemented |
| 503 | Service Unavailable |

- By default the Web browser will display some message to the user. This message is usually generated by the browser not by the Web server.



Generic HTTP Error Page



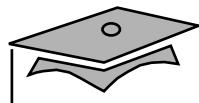


Servlet Exceptions

- A servlet can throw a `ServletException` to indicate an exception has occurred.

```
12  public void doGet(HttpServletRequest request,
13          HttpServletResponse response)
14      throws ServletException {
15      int x = 0;
16      int y = 0;
17
18      try {
19          int z = x / y;
20      } catch (ArithmaticException ae) {
21          throw new ServletException(ae);
22      }
23 }
```

- The Web container will catch these exceptions and send an HTTP response with a 500 status code and an HTML response with the backtrace of the exception.
- Errors and runtime exceptions are also handled directly by the Web container.



Generic Servlet Error Page

The screenshot shows a vintage-style Netscape Communicator browser window with a red title bar. The title bar reads "Netscape: Tomcat Exception Report". The menu bar includes "File", "Edit", "View", "Go", "Communicator", and "Help". The toolbar below the menu bar has icons for "Bookmarks", "Location", "Admin", "Time Sheet Entry", "SES", "Java", "Java Web Tech", "Personal", and "Search". The location bar shows the URL "http://localhost:8080/SL314_Mod07_Decl/ErrorServlet". The main content area displays the following text:

A Servlet Exception Has Occurred

Exception Report:

```
javax.servlet.ServletException: / by zero
    at sl314.mod07.examples.ErrorProneServlet.doGet(ErrorProneServlet.java:23)
    at javax.servlet.http.HttpServlet.service(HttpServlet.java:740)
    at javax.servlet.http.HttpServlet.service(HttpServlet.java:853)
    at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:24)
    at org.apache.catalina.core.StandardWrapperValve.invoke(StandardWrapperValve.java:251)
    at org.apache.catalina.core.ContainerBase.invoke(ContainerBase.java:977)
    at org.apache.catalina.core.StandardContextValve.invoke(StandardContextValve.java:196)
    at org.apache.catalina.core.ContainerBase.invoke(ContainerBase.java:977)
    at org.apache.catalina.core.StandardContext.invoke(StandardContext.java:2041)
    at org.apache.catalina.core.StandardHostValve.invoke(StandardHostValve.java:161)
    at org.apache.catalina.valves.ValveBase.invokeNext(ValveBase.java:242)
    at org.apache.catalina.valves.AccessLogValve.invoke(AccessLogValve.java:414)
    at org.apache.catalina.core.ContainerBase.invoke(ContainerBase.java:975)
    at org.apache.catalina.core.StandardEngineValve.invoke(StandardEngineValve.java:159)
    at org.apache.catalina.core.ContainerBase.invoke(ContainerBase.java:977)
    at org.apache.catalina.connector.http.HttpProcessor.process(HttpProcessor.java:818)
    at org.apache.catalina.connector.http.HttpProcessor.run(HttpProcessor.java:897)
    at java.lang.Thread.run(Thread.java:484)
```

Root Cause:

```
java.lang.ArithmetricException: / by zero
```



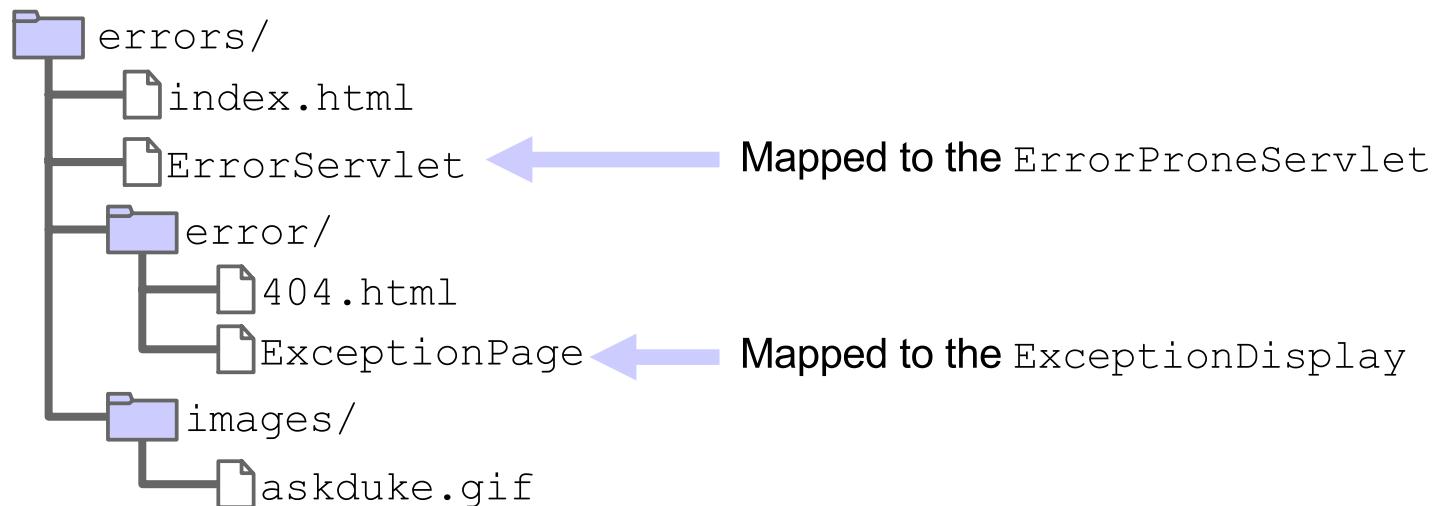
Using Custom Error Pages

- Declarative – Use the deployment descriptor to declare error pages for specific situations (HTTP errors or Java technology exceptions) and let the Web container handle the forwarding to these pages.
- Programmatic – Handle the Java technology exceptions directly in your servlet code and forward the HTTP request to the error page of your choice.



Creating Error Pages

An error page can be a static HTML page or a servlet:





Declaring HTTP Error Pages

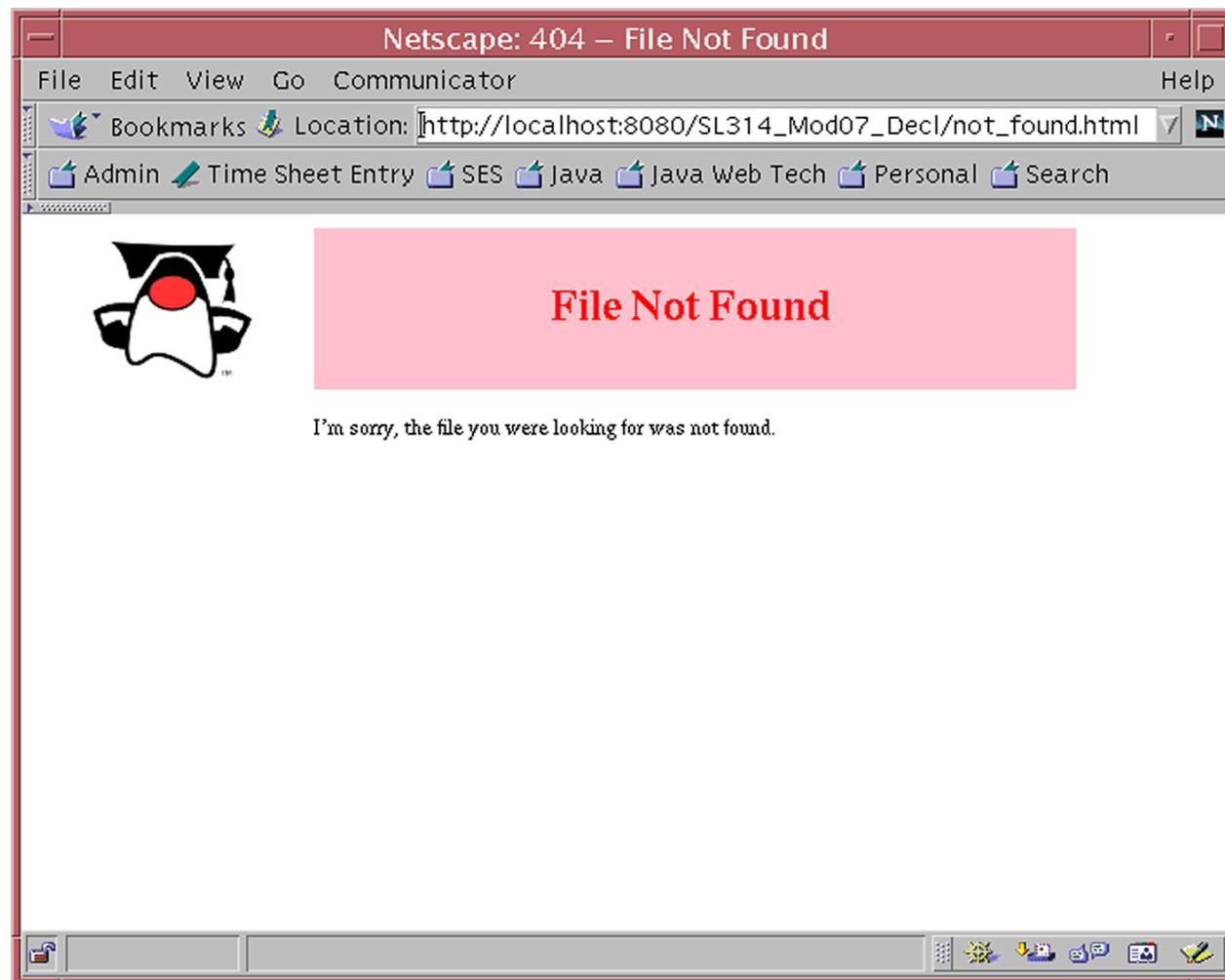
- Use the `error-page` element to declare a handler for a given HTTP status code:
 - `error-code` – The error code, like 404, that you want to show an error page for
 - `location` – The URL of the HTML page or servlet to display as an error page

```
44    <error-page>
45        <error-code>404</error-code>
46        <location>/error/404.html</location>
47    </error-page>
```

- You may declare any number of error pages, but only one for a given status code.



Example HTTP Error Page





Declaring Servlet Exception Error Pages

- Use the exception-type element to declare a handler for a given Java technology exception.

```
49 <error-page>
50     <exception-type>java.lang.ArithmeticException</exception-type>
51     <location>/error/ExceptionPage</location>
52 </error-page>
```

- You may declare any number of error pages, but only one for a given exception type.
- Furthermore, you *cannot* capture multiple exception types using a superclass, such as `java.lang.Exception`.

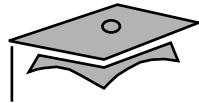


Throwing Servlet Exceptions

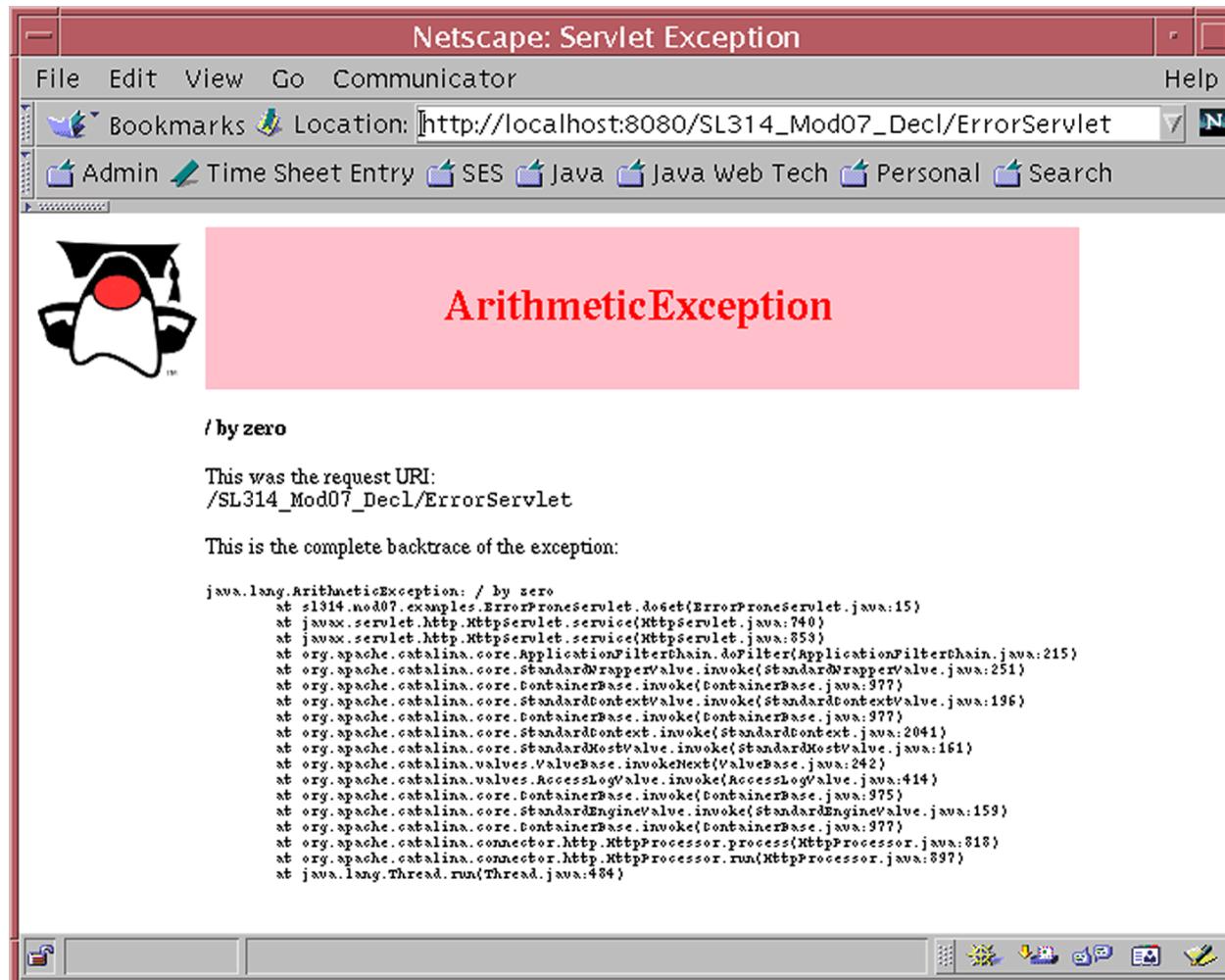
- A servlet can throw a `ServletException` to indicate that an exception has occurred.

```
12  public void doGet(HttpServletRequest request,
13          HttpServletResponse response)
14      throws ServletException {
15      int x = 0;
16      int y = 0;
17
18      try {
19          int z = x / y;
20      } catch (ArithmaticException ae) {
21          throw new ServletException(ae);
22      }
23 }
```

- All checked exceptions must be caught and wrapped in a `ServletException` because you cannot declare new exceptions in the service (`doGet`, and so on) method signature.



Example Servlet Error Page





Developing an Error Handling Servlet

Override both the doGet and doPost methods:

```
12 public final class ExceptionDisplay extends HttpServlet {  
13  
14     public void doGet(HttpServletRequest request,  
15                         HttpServletResponse response)  
16         throws IOException {  
17         generateResponse(request, response);  
18     }  
19  
20     public void doPost(HttpServletRequest request,  
21                          HttpServletResponse response)  
22         throws IOException {  
23         generateResponse(request, response);  
24     }  
25  
26     public void generateResponse(HttpServletRequest request,  
27                                 HttpServletResponse response)  
28         throws IOException {  
29 }
```

Do this to handle exceptions thrown from the original servlet's doGet or doPost method.



Developing an Error Handling Servlet

Two predefined request attributes:

```
26  public void generateResponse(HttpServletRequest request,
27                      HttpServletResponse response)
28      throws IOException {
29
30     response.setContentType("text/html");
31     PrintWriter out = response.getWriter();
32
33     Throwable exception
34         = (Throwable) request.getAttribute("javax.servlet.error.exception");
35     String expTypeFullName
36         = exception.getClass().getName();
37     String expTypeName
38         = expTypeFullName.substring(expTypeFullName.lastIndexOf(".") + 1);
39     String request_uri
40         = (String) request.getAttribute("javax.servlet.error.request_uri");
```

- `javax.servlet.error.exception` contains the original (wrapped) exception.
- `javax.servlet.error.request_uri` contains the URI of the original client request.



Programmatic Exception Handling

- You can write your servlet code to catch all exceptions and handle them yourself, as opposed to letting the Web container do this for you.
- You can handle only Java technology exceptions using this technique, but not HTTP errors.



Programmatic Exception Handling

- Use a try-catch block for all code in which an exception could occur.
- Use a RequestDispatcher (Lines 28–29) in the catch block to forward (Line 33) the exception to a servlet exception handler.
- (Optional) You can also set specific request attributes (Lines 30–32).

```
25 // Catch any exceptions and forward to the Exception Handler servlet
26 } catch (Exception e) {
27     ServletContext context = getServletContext();
28     RequestDispatcher errorPage
29         = context.getNamedDispatcher("ExceptionHandler");
30     request.setAttribute("javax.servlet.error.exception", e);
31     request.setAttribute("javax.servlet.error.request_uri",
32                         request.getRequestURI());
33     errorPage.forward(request, response);
34 }
35 }
```

--



Programmatic Exception Handling

- Declare the servlet exception handler, but do not provide a URL mapping.

```
22 <servlet>
23     <servlet-name>ExceptionHandler</servlet-name>
24     <description>
25         The servlet that handles displaying any exception
26         thrown from another servlet.
27     </description>
28     <servlet-class>s1314.web.ExceptionDisplay</servlet-class>
29 </servlet>
```

- Use servlet name to acquire the RequestDispatcher.

```
25 // Catch any exceptions and forward to the Exception Handler servlet
26 } catch (Exception e) {
27     ServletContext context = getServletContext();
28     RequestDispatcher errorPage
29         = context.getNamedDispatcher("ExceptionHandler");
30     request.setAttribute("javax.servlet.error.exception", e);
31     request.setAttribute("javax.servlet.error.request_uri",
32                         request.getRequestURI());
33     errorPage.forward(request, response);
34 }
35 }
```



Trade-offs for Declarative Exception Handling

The advantage of declarative exception handling is that it is easy to implement.

The disadvantages are:

- You must create a URL mapping for every exception handling servlet.
- You cannot do many-to-one mappings.
- It is often too generic.



Trade-offs for Programmatic Exception Handling

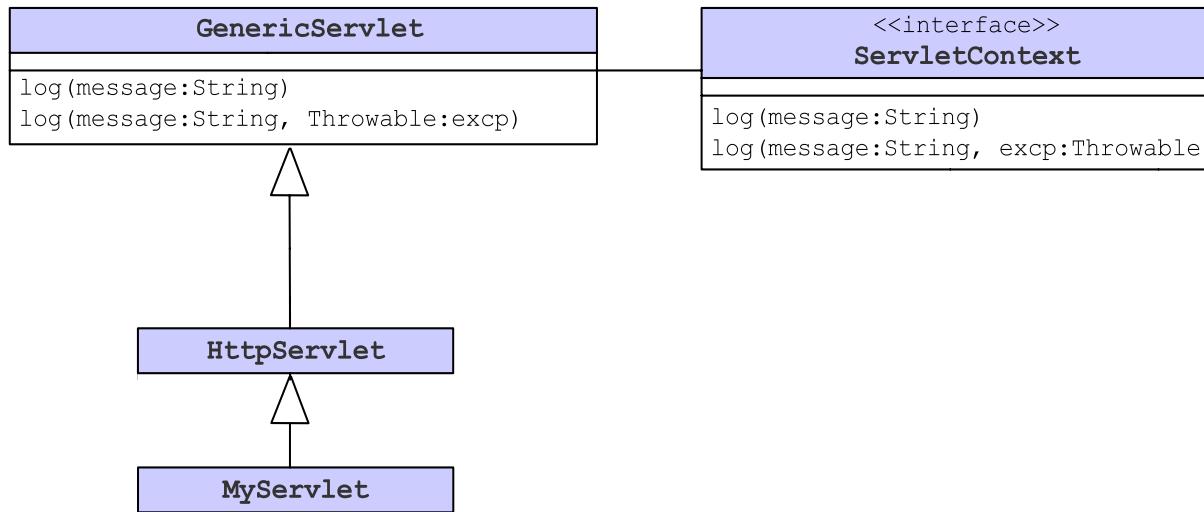
The advantages of programmatic exception handling are that:

- It keeps the handler code close to the Controller.
- It makes dealing with exceptions explicit.
- The handler can be customized to the situation.

The disadvantage is that it requires more code to implement.



Logging Exceptions



Log exceptions using the two log methods:

- `log(String)` – Prints the message to a log file
- `log(String, Throwable)` – Prints the message and the stack trace of the exception to a log file



Summary

- There are two types of errors in a Web application: HTTP errors and Java technology exceptions.
- You can use the `error-page` deployment descriptor element to declare error pages for both types.
- You must wrap checked exceptions in a `ServletException` object in your `doXYZ` methods.
- An error handler servlet has access to two request attributes (`javax.servlet.error.exception` and `javax.servlet.error.request_uri`).
- For programmatic error handling, you can use a `RequestDispatcher` object to forward the HTTP request directly to the error handler servlet.



Module 10

Configuring Web Application Security



Objectives

- State the importance of Web security
- Use the deployment descriptor to configure authorization for a Web application resource
- Use the deployment descriptor to configure authentication of users for a Web application



Web Security Issues

- Security issues are generally not Web-application-specific. The Web container provides a group of generic services.
- This module presents a general overview of:
 - Authentication
 - Authorization
 - Maintaining data integrity
 - Tracking access (auditing)
 - Dealing with malicious code (within the Web application)
 - Dealing with Web attacks (such as “denial of service”)



Authentication

Authentication is the process of verifying the user's identity. This is usually done with a username and password.

You can configure four authentication techniques:

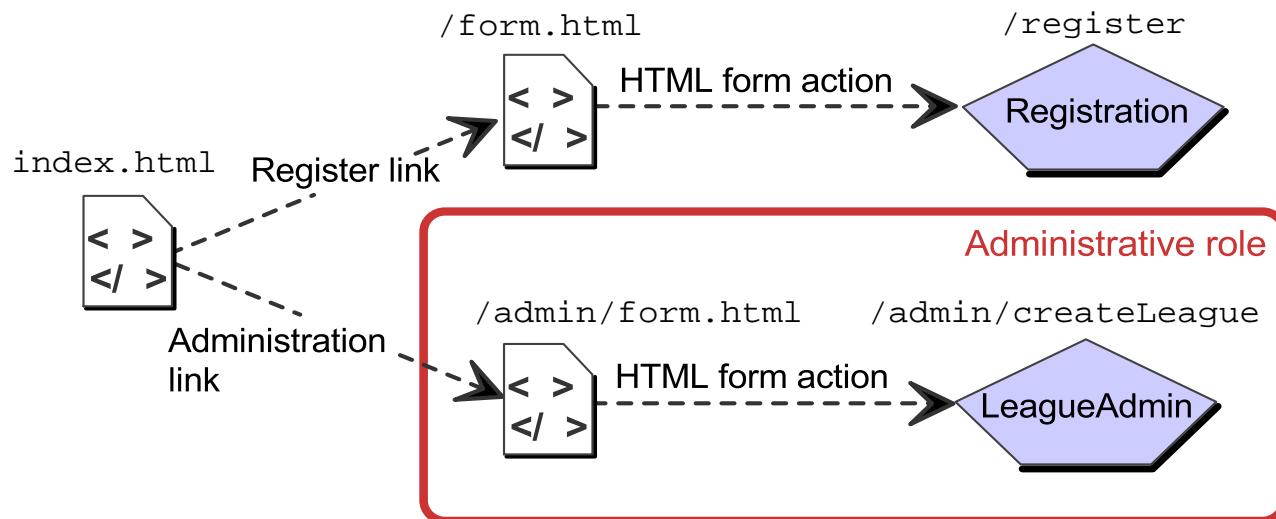
- **BASIC** – The Web browser solicits the username and password and sends it to the Web server “in the open.”
- **DIGEST** – The Web browser solicits the username and password and sends this data to the Web server, which has been encoded using an algorithm (such as MD5).
- **FORM** – The Web application supplies an HTML form that is sent to the Web browser.
- **CLIENT-CERT** – The Web container uses Secure Sockets Layer (SSL) to verify the user.



Authorization

Authorization is the process of partitioning Web resources based on user roles.

Identify *security domains* in the Web application.



The Web container uses a vendor-specific *security realm* mechanism to verify user roles.



Maintaining Data Integrity

Data integrity is the process of securing the transmission of your confidential data using encryption.

- Data sent across the network is vulnerable in two ways:
 - Data can be observed or intercepted
 - Data can be corrupted during transmission
- You can protect data against both risks using HTTPS.
- HTTPS (Secure Hypertext Transfer Protocol) is HTTP using the Secure Sockets Layer protocol.



Access Tracking

Access tracking (also called *auditing*) is the process of keeping records of every access to your Web application.

- You can audit the actions that occur in your Web application by writing them to log files.
- Use the appropriate logging method:
 - `log(String)` – Prints the message to a log file
 - `log(String, Throwable)` – Prints the message and the stack trace of the exception to a log file
 - These methods exist in both the `ServletContext` interface and the `GenericServlet` class.



Dealing With Malicious Code

Malicious code is code designed with the intent of damaging the system on which it is running.

- In general, malicious code includes viruses, worms, Trojan horses, and so on.
- In Web applications, malicious code is code on the server that has the hidden intent of damaging the server.
- Internet service providers need to partition Web applications to prevent malicious code attacks.
- A Web container can isolate Web applications using a Java SecurityManager. This is not required in the Servlet specification; therefore, is a “value added” feature for vendors.



Dealing With Web Attacks

Web attacks are attempts to compromise a server by an outside individual or group.

- Web attacks include “denial of service,” cracking server passwords, downloading confidential files, and so on.
- There is nothing in the Servlet specification to safeguard against Web attacks.
- You can prevent some Web attacks by using a firewall. For example, you can prevent all external TCP connections that are not HTTP or HTTPS requests.



Declarative Authorization

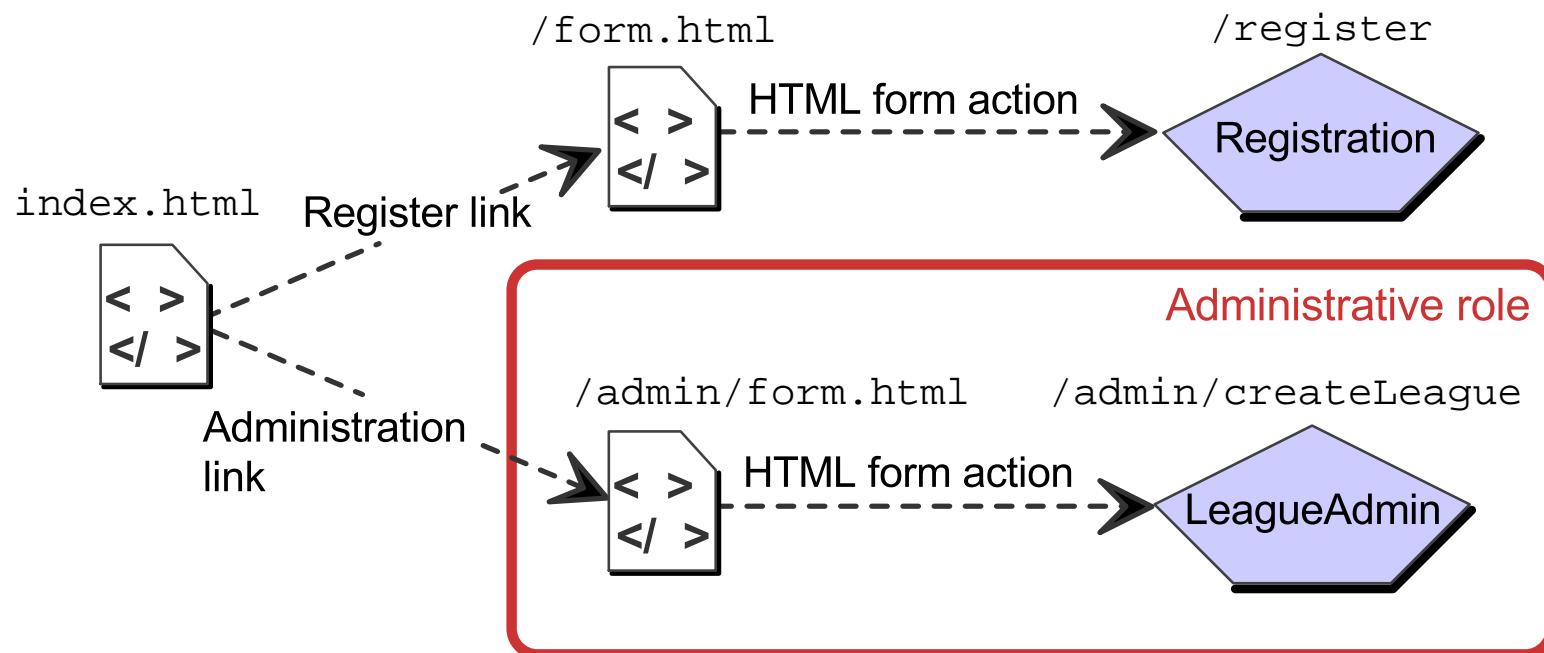
To implement declarative authorization, you must:

- Identify the Web resource collections
- Identify the roles
- Map the Web resource collection to the roles
- Identify the users in each of those roles



Web Resource Collection

Here is an example Web resource collection: an administration hierarchy in the Soccer League Web application.





Web Resource Collection

A Web resource collection defines:

- The set of URLs that define the collection (Line 53).
- The HTTP methods that are used to access the resources in the URL patterns (Lines 54 and 55).

```
48 <web-resource-collection>
49   <web-resource-name>League Administration</web-resource-name>
50   <description>
51     Resources accessible only to administrators.
52   </description>
53   <url-pattern>/admin/*</url-pattern>
54   <http-method>POST</http-method>
55   <http-method>GET</http-method>
56 </web-resource-collection>
```



Declaring Security Roles

Use the deployment descriptor to declare an authorization constraint:

```
47 <security-constraint>
48   <web-resource-collection>
49     <web-resource-name>League Administration</web-resource-name>
50     <description>
51       Resources accessible only to administrators.
52     </description>
53     <url-pattern>/admin/*</url-pattern>
54     <http-method>POST</http-method>
55     <http-method>GET</http-method>
56   </web-resource-collection>
57   <auth-constraint>
58     <role-name>administrator</role-name>
59   </auth-constraint>
60 </security-constraint>
```

In addition, you must define every security role:

```
66 <security-role>
67   <description>A simple restricted-access user role.</description>
68   <role-name>administrator</role-name>
69 </security-role>
```



Security Realms

- A *security realm* is a software component for matching users to roles. It also verifies the user's password.
- Every Web container must include a security realm, but this mechanism is *vendor-specific*.
- There are many possible mechanisms:
 - Flat-file (the MemoryRealm class in the Tomcat server)
 - Database tables (the JDBCRealm class in the Tomcat server)
 - Lightweight Directory Access Protocol (LDAP)
 - Network Information System (NIS)



Declarative Authentication

Use the deployment descriptor to declare the Web application's authentication technique:

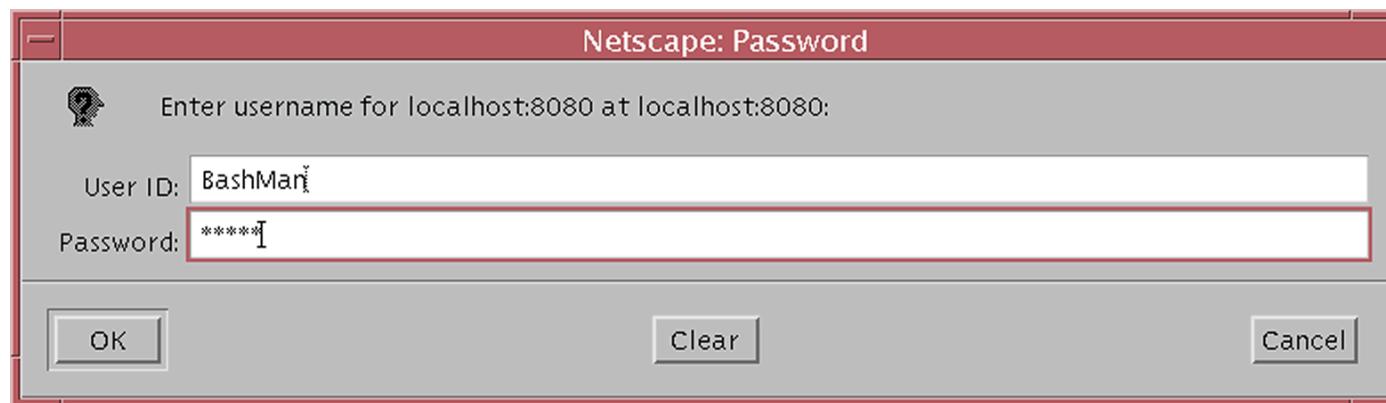
```
62 <login-config>
63   <auth-method>BASIC</auth-method>
64 </login-config>
```

- The auth-method element can take one of four values: BASIC, DIGEST, FORM, or CLIENT-CERT.
- CLIENT-CERT uses SSL and is the most secure of the four techniques, but requires the user to have an X-509 certificate.
- The other techniques can be combined with server-side SSL to become as secure as CLIENT-CERT.



BASIC Authentication

Example Netscape authentication dialog box:



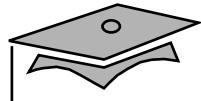


Form-based Authentication

Use the deployment descriptor to declare the Web application's login form (and error page):

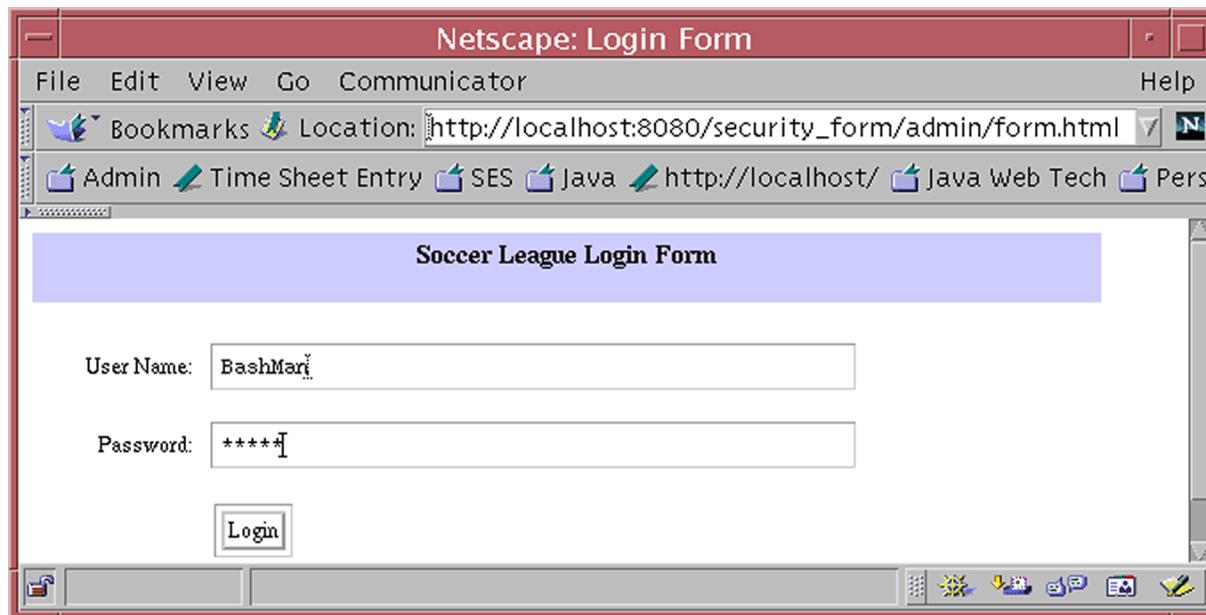
```
62    <login-config>
63        <auth-method>FORM</auth-method>
64        <form-login-config>
65            <form-login-page>/login/form.html</form-login-page>
66            <form-error-page>/login/error.html</form-error-page>
67        </form-login-config>
68    </login-config>
```

You need to develop the form and error pages.



Form-based Authentication

Example login form:





Form-based Authentication

The HTML form:

```
17 <FORM ACTION='j_security_check' METHOD='POST'>
18
19 <TABLE BORDER='0' CELLSPACING='0' CELLPADDING='5' WIDTH='600'>
20 <TR>
21   <TD ALIGN='right'>User Name:</TD>
22   <TD><INPUT TYPE='text' NAME='j_username' SIZE='50'></TD>
23 </TR>
24 <TR>
25   <TD ALIGN='right'>Password:</TD>
26   <TD><INPUT TYPE='password' NAME='j_password' SIZE='50'></TD>
27 </TR>
28 <TR>
29   <TD></TD>
30   <TD><INPUT TYPE='submit' VALUE='Login'></TD>
31 </TR>
32 </TABLE>
33
34 </FORM>
```

The Web container intercepts the j_security_check action and handles the authentication. There is nothing that you need to code.



Summary

- There are six main security issues: authorization, authentication, data integrity, tracking access, dealing with malicious code, and dealing with Web attacks.
- You can use the deployment descriptor to configure the authorization security domains in your Web application.
- You can use the deployment descriptor to configure the authentication technique for your Web application.



Module 11

Understanding Web Application Concurrency Issues



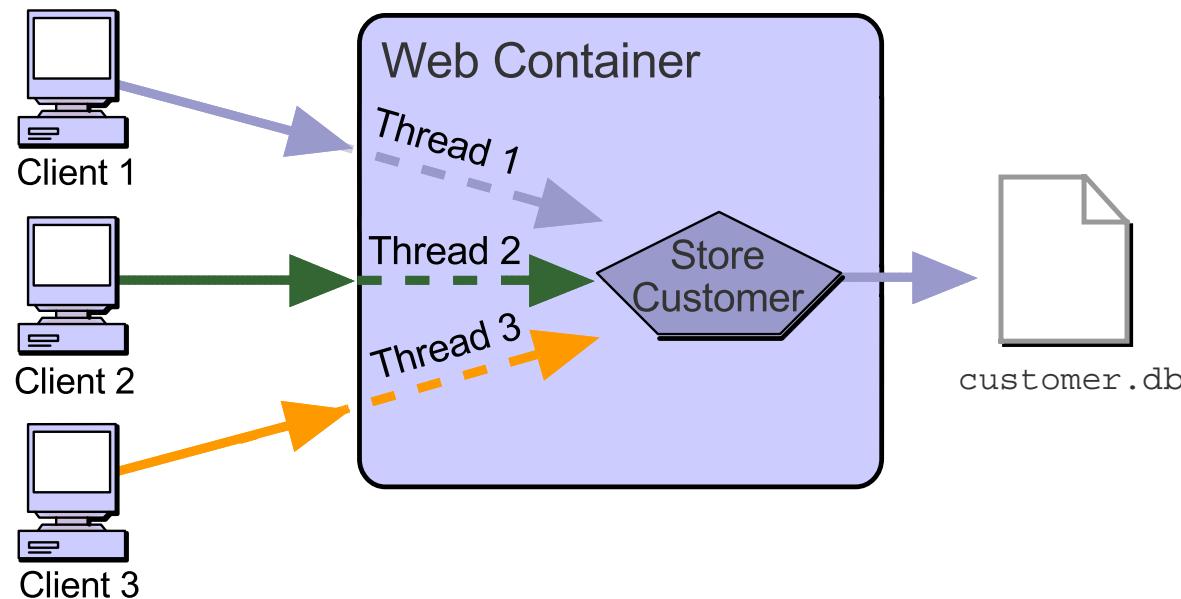
Objectives

- Describe why servlets need to be thread-safe
- Describe the “attribute” scope rules and the corresponding currency issues
- Describe the “single thread model” for servlets and the issues with this concurrency strategy
- Design a Web application for concurrency



The Need for Servlet Concurrency Management

- Multiple simultaneous threads run; they all share the same servlet instance:



- This can cause data corruption if shared data and resources are not properly synchronized.



Concurrency Management Example

The example servlet stores customer data to a common file:

```
45 // Extract all of the request parameters
46 String name = request.getParameter("name");
47 String address = request.getParameter("address");
48 String city = request.getParameter("city");
49 String province = request.getParameter("province");
50 String postalCode = request.getParameter("postalCode");
51
52 // Store the customer data to the flat-file
53 customerDataWriter.write(name, 0, name.length());
54 customerDataWriter.write('|');
55 customerDataWriter.write(address, 0, address.length());
56 customerDataWriter.write('|');
57 customerDataWriter.write(city, 0, city.length());
58 customerDataWriter.write('|');
59 // Simulate a suspension of the current thread.
60 Thread.yield();
61 customerDataWriter.write(province, 0, province.length());
62 customerDataWriter.write('|');
63 customerDataWriter.write(postalCode, 0, postalCode.length());
64 customerDataWriter.write('\n');
65 customerDataWriter.flush();
```



Concurrency Management Example

You can eliminate the concurrency problems by synchronizing on the file writer object:

```
45 // Extract all of the request parameters
46 String name = request.getParameter("name");
47 String address = request.getParameter("address");
48 String city = request.getParameter("city");
49 String province = request.getParameter("province");
50 String postalCode = request.getParameter("postalCode");
51
52 // Store the customer data to the flat-file
53 synchronized (customerDataWriter) {
54     customerDataWriter.write(name, 0, name.length());
55     customerDataWriter.write('|');
56     customerDataWriter.write(address, 0, address.length());
57     customerDataWriter.write('|');
58     customerDataWriter.write(city, 0, city.length());
59     customerDataWriter.write('|');
60 // Simulate a suspension of the current thread.
61 Thread.yield();
62     customerDataWriter.write(province, 0, province.length());
63     customerDataWriter.write('|');
64     customerDataWriter.write(postalCode, 0, postalCode.length());
65     customerDataWriter.write('\n');
66     customerDataWriter.flush();
67 }
```



Attributes and Scope

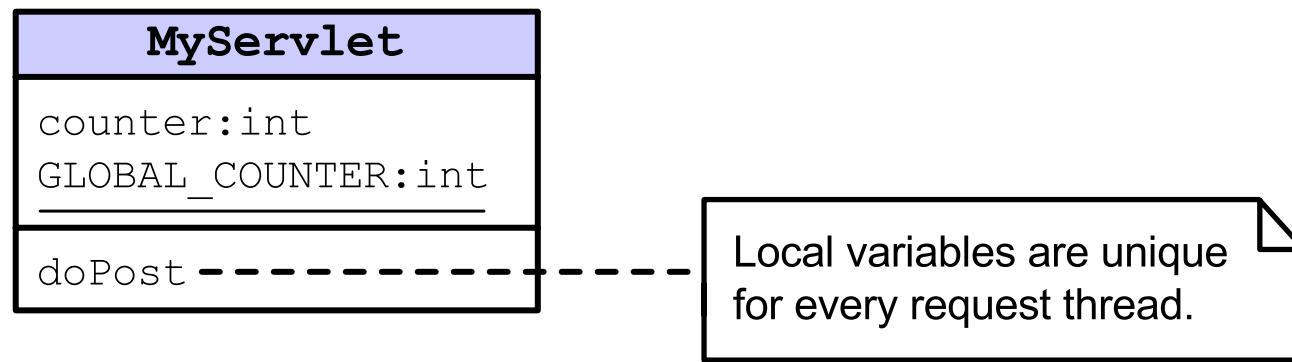
There are six scopes for attributes in a Web application:

- Local variables (also called the page scope)
- Instance variables
- Class variables
- Request attributes (also called the request scope)
- Session attributes (also called the session scope)
- Context attributes (also called the application scope)



Local Variables

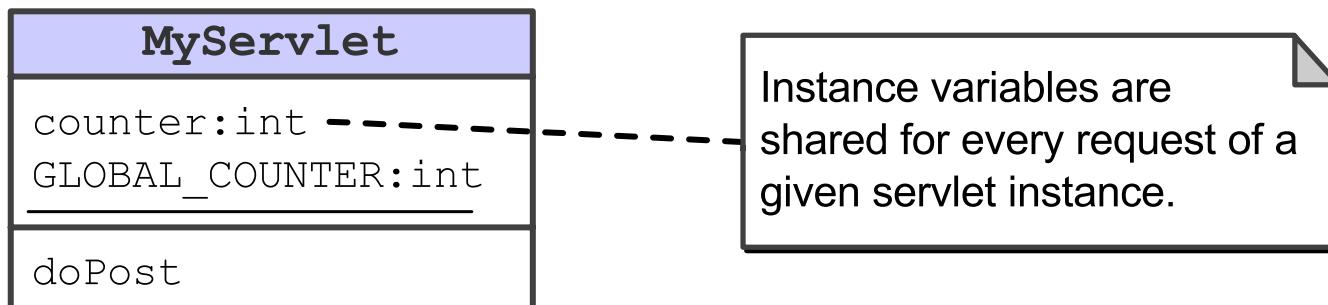
- Thread-safe? Yes
- Uses: Any data processing local to the servlet method
- API:





Instance Variables

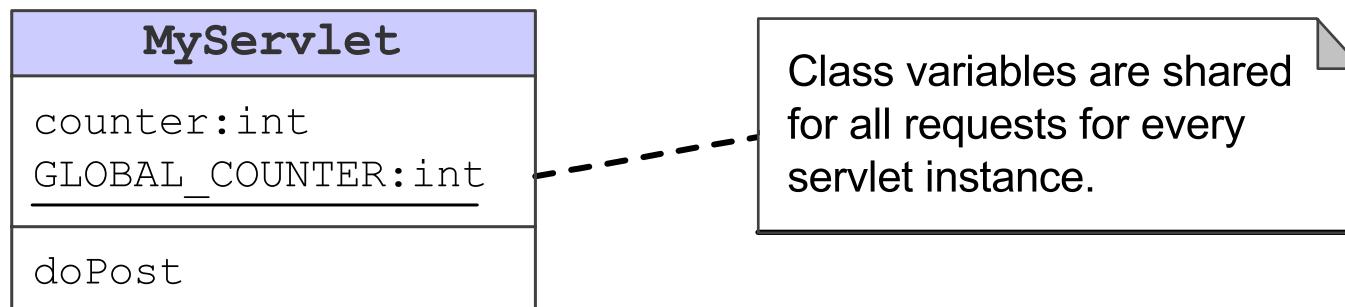
- Thread-safe? No
- Uses:
 - Storing data across multiple requests for a given servlet definition
 - Storing initialization parameters (usually read-only)
- API:





Class Variables

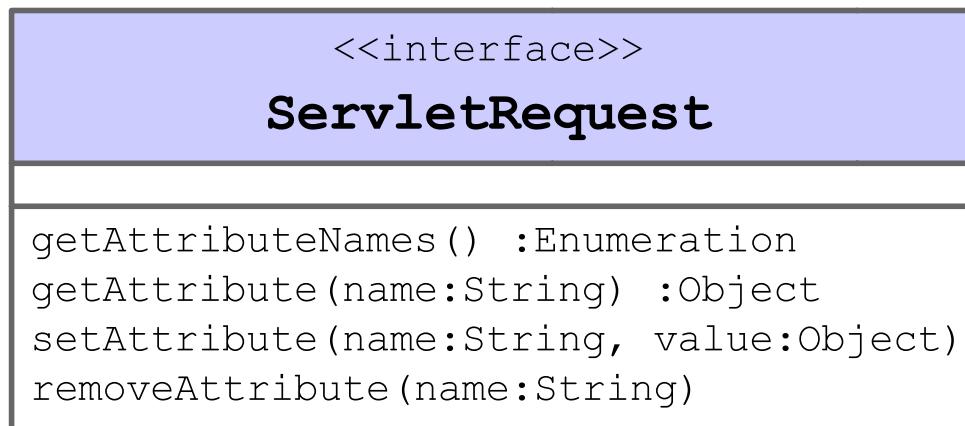
- Thread-safe? No
- Uses: Storing data across multiple servlet definitions
- API:





Request Scope

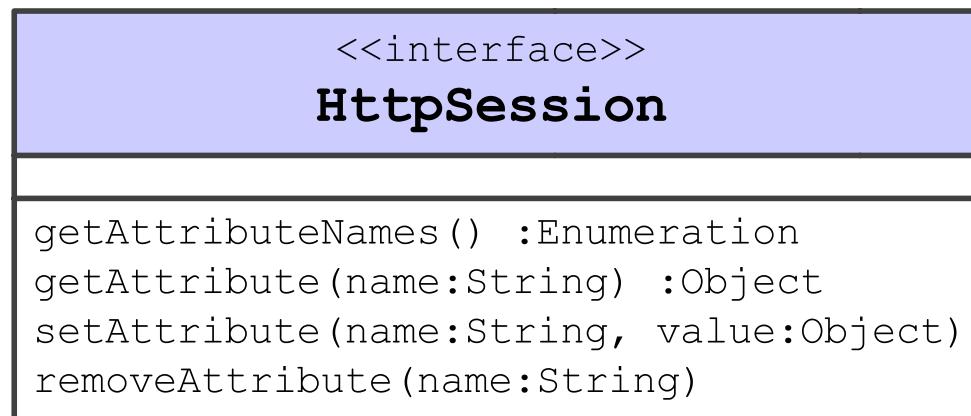
- Thread-safe? Yes
- Uses: Moving data from the Controller to the View for presentation
- API:





Session Scope

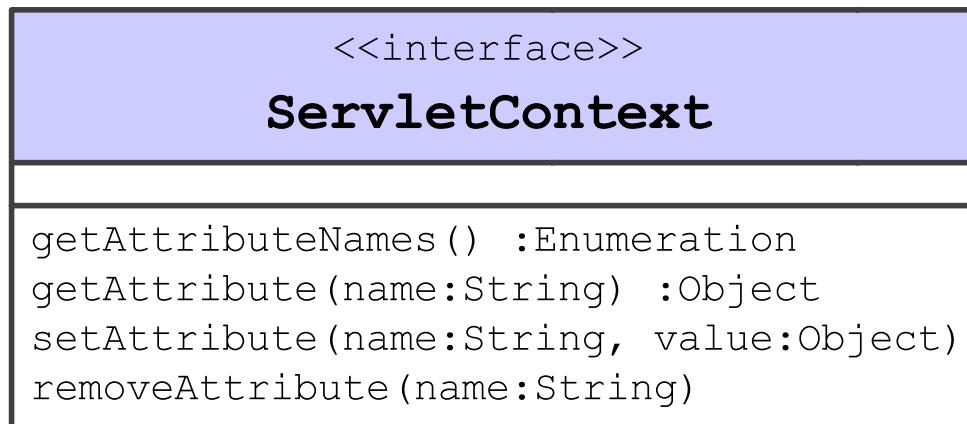
- Thread-safe? No
A user might have multiple browsers active and access the same Web application.
- Uses: Storing data that is common within a Web session
- API:





Application Scope

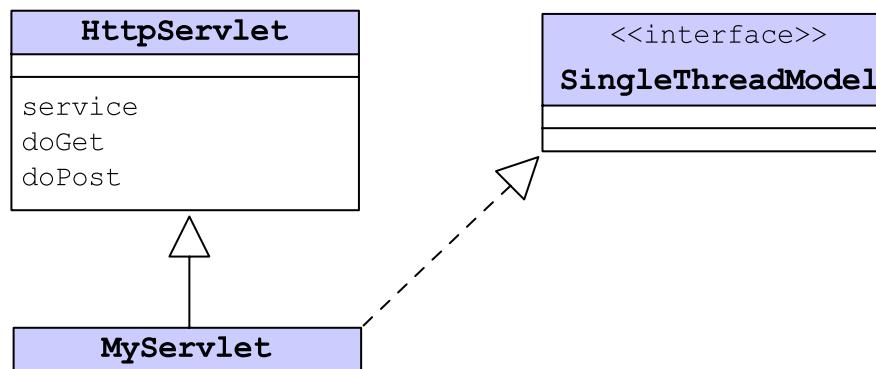
- Thread-safe? No
- Uses: Resources common to all servlets in the Web application
- API:





The SingleThreadModel Interface

- The servlet specification ensures that if your servlet class implements the SingleThreadModel (STM) interface, then only one request thread will execute the service method at a time.



- The SingleThreadModel interface has no methods to implement.
- You can use STM to signal to the Web container that the servlet class must be handled specially.



How the Web Container Might Implement the Single Threaded Model

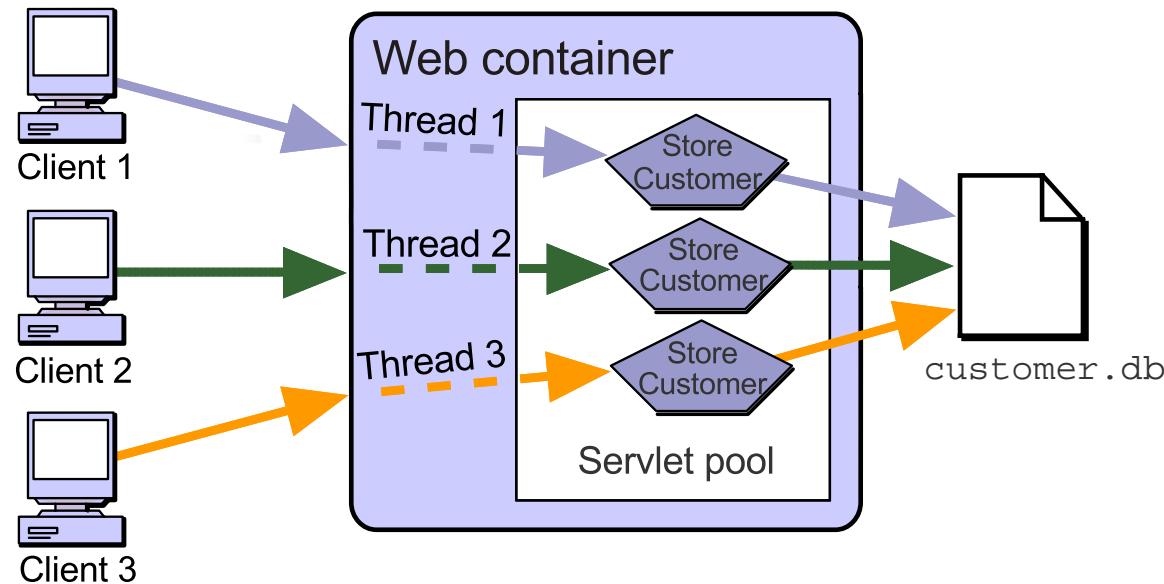
There are several ways that a Web container can implement this:

- Queueing up all of the requests and passing them one at a time to a single servlet instance.
- Creating an infinitely large pool of servlet instances, each of which handles one request at a time.
- Creating a fixed-size pool of servlet instances, each of which handles one request at a time. If more requests arrive than the size of the pool, then some requests are postponed until some other request has finished.



STM and Concurrency Management

A servlet using STM might use multiple servlet instances, and all of these instances would share a common resource:





STM and Concurrency Management

Using STM with your servlets can have significant drawbacks:

- The implementation of the STM mechanism is vendor-specific. You cannot code your servlets with knowledge of a specific mechanism and know that your servlets will be able to port to another Web container vendor.
- The use of STM can significantly slow performance when using the “one at a time” approach.
- The use of STM does not control access to static variables when using a “servlet pool” approach.
- None of the approaches solve concurrency issues for session and application scope attributes.

For these reasons, you should not use the `SingleThreadModel`.



Recommended Approaches to Concurrency Management

- Whenever possible, use only local and request attributes.
- Use the synchronized syntax to control concurrency issues when accessing or updating frequently changing attributes and when using common resources.

```
HttpSession session = request.getSession();
synchronized (session) {
    session.setAttribute("count", new Integer(count+1));
}

HttpSession session = request.getSession();
synchronized (session) {
    countObj = (Integer) session.getAttribute("count");
}
count = countObj.intValue();
```



Recommended Approaches to Concurrency Management

- Minimize the use of synchronization blocks and methods in your servlet class code. Never synchronize the whole doGet or doPost method.
- Use resource classes that have been properly designed for thread-safety. For example, you can assume that your JDBC technology-based driver vendor has designed a thread-safe DataSource class.



Summary

- Using shared resources and multiple, concurrent requests can corrupt your data.
- Only local variables and request attributes are thread-safe; all other scopes are not thread-safe.
- Do not use the `SingleThreadModel` interface.
- Use the `synchronized` syntax to control concurrency issues when accessing/changing thread-unsafe attributes.
- Minimize the use of synchronization blocks and methods in your servlet class code.
- Use resource classes that have been properly designed for thread-safety.



Module 12

Integrating Web Applications With Databases



Objectives

- Understand what a database management system (DBMS) does
- Design a Web application to integrate with a DBMS
- Develop a Web application using a connection pool
- Develop a Web application using a data source and the Java Naming and Directory Interface (JNDI)



Database Overview

- A database is a collection of logically related data.
- Tables contain multiple columns (one for each data attribute) and multiple rows.
- Tables can manage the persistence of domain objects (one row for every object) or of relationships between objects.
- A DBMS includes four fundamental operations:
 - Create a row in table
 - Retrieve one or more rows in a table
 - Update some data in a table
 - Delete one or more rows in a table



The JDBC API

- JDBC™ is the Java technology API for interacting with a relational 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.
- Techniques for designing and developing a Web application in which the JDBC technology code is encapsulated using the Data Access Object design pattern will be described.
- A naive technique is to create a connection object for each request, but this approach is extremely slow and does not scale well.



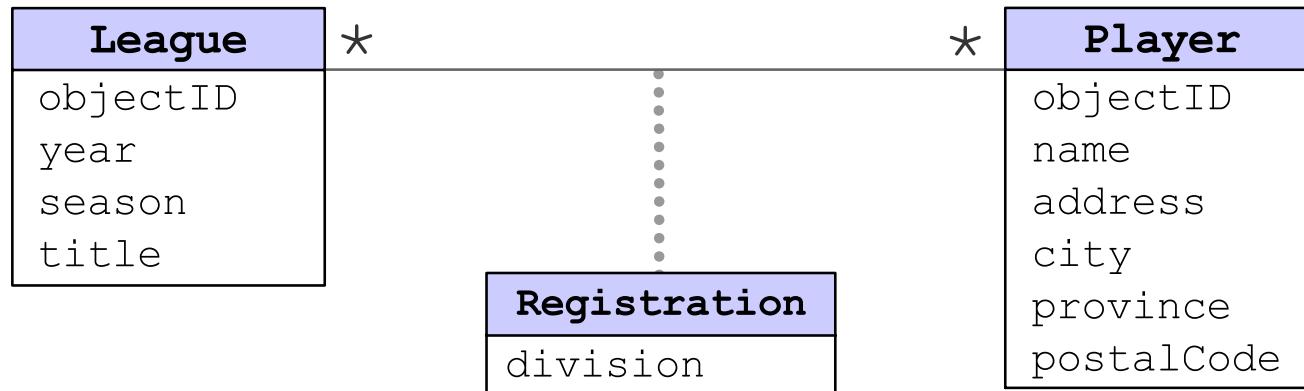
Designing a Web Application That Integrates With a Database

- Design the domain objects of your application
- Design the database tables that map to the domain objects
- Design the business services (the Model) to separate the database code into classes using the DAO pattern



Domain Objects

These are the domain objects in Soccer League Web application:

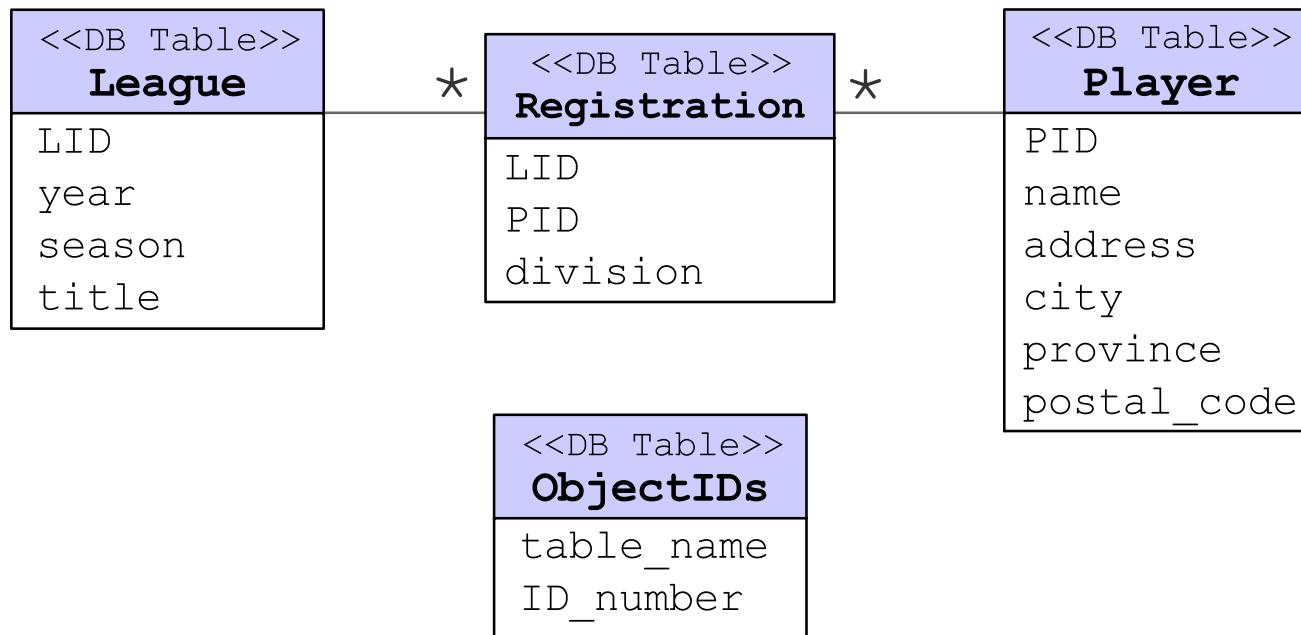


The objectID has been added to the classes to provide a unique ID in the database (DB) table for each of these entities.



Database Tables

This is one possible DB design for the domain objects:



The object ID in the Java technology object corresponds to the ID in the database table. For example, the object ID in the League objects corresponds to the LID in the League table.



Database Tables

Example data:

League

| LID | year | season | title |
|-----|------|--------|----------------------------|
| 001 | 2001 | Spring | Soccer League (Spring '01) |
| 002 | 2001 | Summer | Summer Soccer Fest 2001 |
| 003 | 2001 | Fall | Fall Soccer League 2001 |
| 004 | 2004 | Summer | The Summer of Soccer Love |

Registration

| LID | PID | division |
|-----|-----|--------------|
| 001 | 047 | Amateur |
| 001 | 048 | Amateur |
| 002 | 048 | Semi-Pro |
| 002 | 049 | Professional |
| 003 | 048 | Professional |

Player

| PID | name | address | city | province | postal_code |
|-----|------------------|--------------------|------------|------------|-------------|
| 047 | Steve Sterling | 12 Grove Park Road | Manchester | Manchester | M4 6NF |
| 048 | Alice Hornblower | 62 Woodside Lane | Reading | Berks | RG31 9TT |
| 049 | Wally Winkle | 17 Chippenham Road | London | London | SW19 4FT |

ObjectIDs

| table_name | ID_number |
|------------|-----------|
| League | 005 |
| Player | 050 |



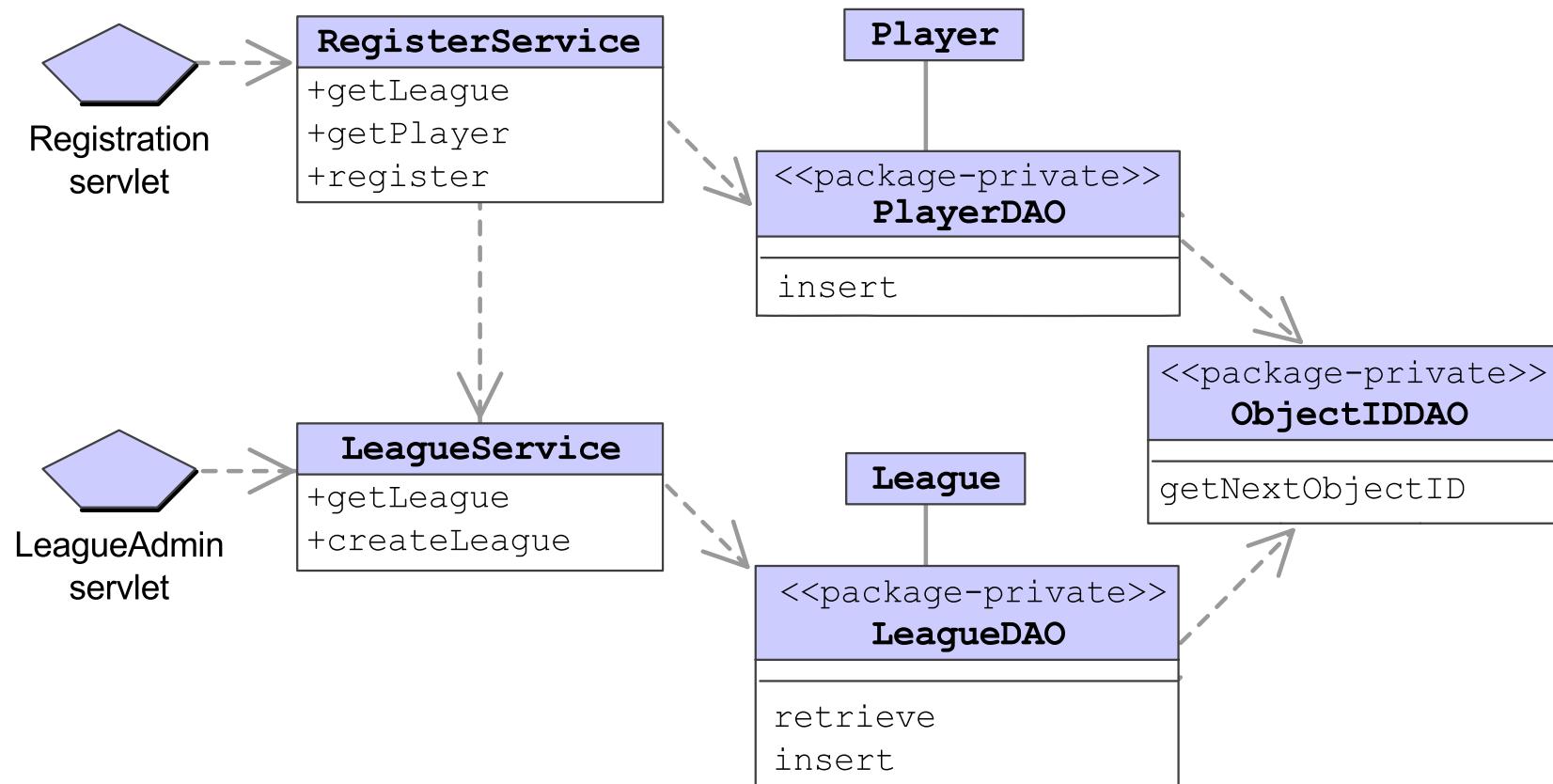
Data Access Object Pattern

- The Data Access Object (DAO) pattern separates the business logic from the data access (data storage) logic.
- The data access implementation (usually JDBC technology calls) is encapsulated in DAO classes.
- The DAO pattern permits the business logic and the data access logic to change independently.

For example, if the DB schema changes, then you would only need to change the DAO methods and not the business services or the domain objects.



Data Access Object Pattern





Advantages of the DAO Pattern

- Business logic and data access logic are now separate.
- The data access objects promote reuse and flexibility in changing the system.
- Developers writing other servlets can reuse the same data access code.
- This design makes it easy to change front-end technologies.
- This design makes it easy to change back-end technologies.



Developing a Web Application That Uses a Connection Pool

- Build or buy a connection pool subsystem.
- Store the connection pool object in a global “name space.”
- Design your DAOs to use the connection pool, retrieving it from the name space.
- Develop a servlet context listener to initialize the connection pool and store it in the name space.

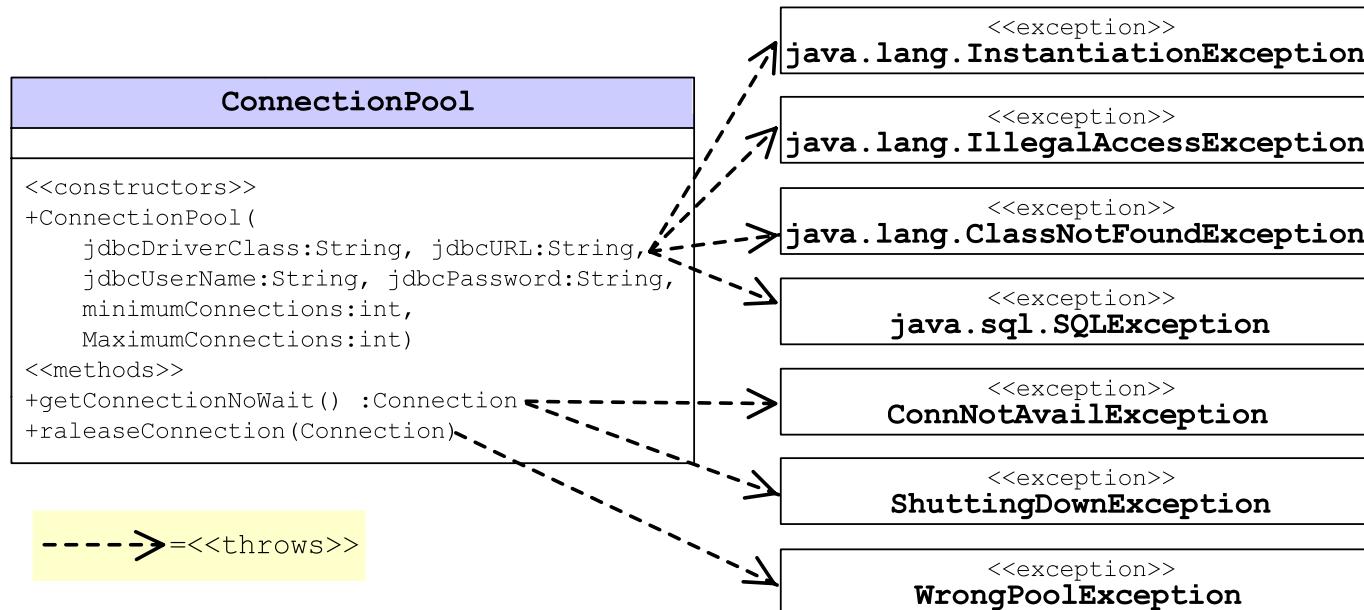


Connection Pool

- A connection pool is a subsystem that manages a collection of JDBC connection objects.
- There must be a method to retrieve a connection object and there must be a method to release the connection back into the pool.
- The goal of the connection pool is to allow the DAO subsystem to retrieve a connection object only when it is needed. This reduces contention for the connection resources.



Connection Pool



- Use the `getConnectionNoWait` method to acquire a connection
- Use the `releaseConnection` method to release a connection



Storing the Connection Pool in a Global Name Space

- The goal is to allow the DAO classes to get a connection object globally. There are many strategies for creating a global name space.
- The Soccer League example uses a simple hand-coded NamingService class that implements the Singleton pattern.





Accessing the Connection Pool

The DAO classes can now retrieve the connection pool from the NamingService. For example, in the LeagueDAO class:

```
27  /**
28   * This method retrieves a League object from the database.
29  */
30 League retrieve(String year, String season)
31     throws ObjectNotFoundException {
32
33     // Retrieve the connection pool from the global Naming Service
34     NamingService nameSvc = NamingService.getInstance();
35     ConnectionPool connectionPool
36         = (ConnectionPool) nameSvc.getAttribute("connectionPool");
37
38     // Database variables
39     Connection connection = null;
40     PreparedStatement stmt = null;
41     ResultSet results = null;
42     int num_of_rows = 0;
43
44     // Domain variables
45     League league = null;
46
```



Initializing the Connection Pool

The `InitializeConnectionPool` listener class creates the connection pool in the `contextInitialized` method:

```
68     // Intialize the connection pool object
69     try {
70         NamingService nameSvc = NamingService.getInstance();
71         connectionPool = new ConnectionPool(jdbcDriver, jdbcURL,
72                                             jdbcUserName, jdbcPassword,
73                                             minimumConnections,
74                                             maximumConnections);
75         nameSvc.setAttribute("connectionPool", connectionPool);
76         context.log("Connection pool created for URL=" + jdbcURL);
```

The `contextDestroyed` method shuts down the pool:

```
101    NamingService nameSvc = NamingService.getInstance();
102    ConnectionPool connectionPool
103        = (ConnectionPool) nameSvc.getAttribute("connectionPool");
104
105    // Shutdown the connection pool
106    connectionPool.shutdown();
107    context.log("Connection pool shutdown.");
```



Developing a Web Application That Uses a Data Source

If you are developing in a full J2EE environment, then you should use a data source object instead of a connection pool.

- Buy a data source subsystem from your DBMS vendor or your J2EE platform vendor.
- Store the DataSource object in JNDI. This is handled by the J2EE platform deployment environment.
- Design your DAOs to use the data source, retrieving it from JNDI.



Summary

- A DBMS is often used to implement a robust persistence mechanism for large scale applications.
- The JDBC API is used for interacting with a DBMS.
- The DAO pattern permits the separation of the business logic (and domain objects) from the persistence mechanism.
- A connection pool is used to manage a pool of reusable Connection objects.
- A data source might also be used to manage a pool of reusable Connection objects. Data sources are part of the J2EE platform.



Module 13

Developing JSP™ Pages



Objectives

- Describe JavaServer Page (JSP) technology
- Write JSP code using scripting elements
- Write JSP code using the page directive
- Create and use JSP error pages
- Describe what the Web container is doing behind the scenes in processing a JSP page



JavaServer Page Technology

- JavaServer Pages enable you to write standard HTML pages containing tags that run powerful programs based on Java technology.
- The goal of JSP technology is to support separation of presentation and business logic:
 - Web designers can design and update pages without learning the Java programming language.
 - Programmers for Java platform can write code without dealing with Web page design.



JavaServer Page Technology

To compare the HelloServlet with an equivalent JSP page:

```
27  public void generateResponse(HttpServletRequest request,
28                      HttpServletResponse response)
29      throws IOException {
30
31     // Determine the specified name (or use default)
32     String name = request.getParameter("name");
33     if ( (name == null) || (name.length() == 0) ) {
34         name = DEFAULT_NAME;
35     }
36
37     // Specify the content type is HTML
38     response.setContentType("text/html");
39     PrintWriter out = response.getWriter();
40
41     // Generate the HTML response
42     out.println("<HTML>");
43     out.println("<HEAD>");
44     out.println("<TITLE>Hello Servlet</TITLE>");
45     out.println("</HEAD>");
46     out.println("<BODY BGCOLOR='white'>");
47     out.println("<B>Hello, " + name + "</B>");
48     out.println("</BODY>");
49     out.println("</HTML>");
50
```



JavaServer Page Technology

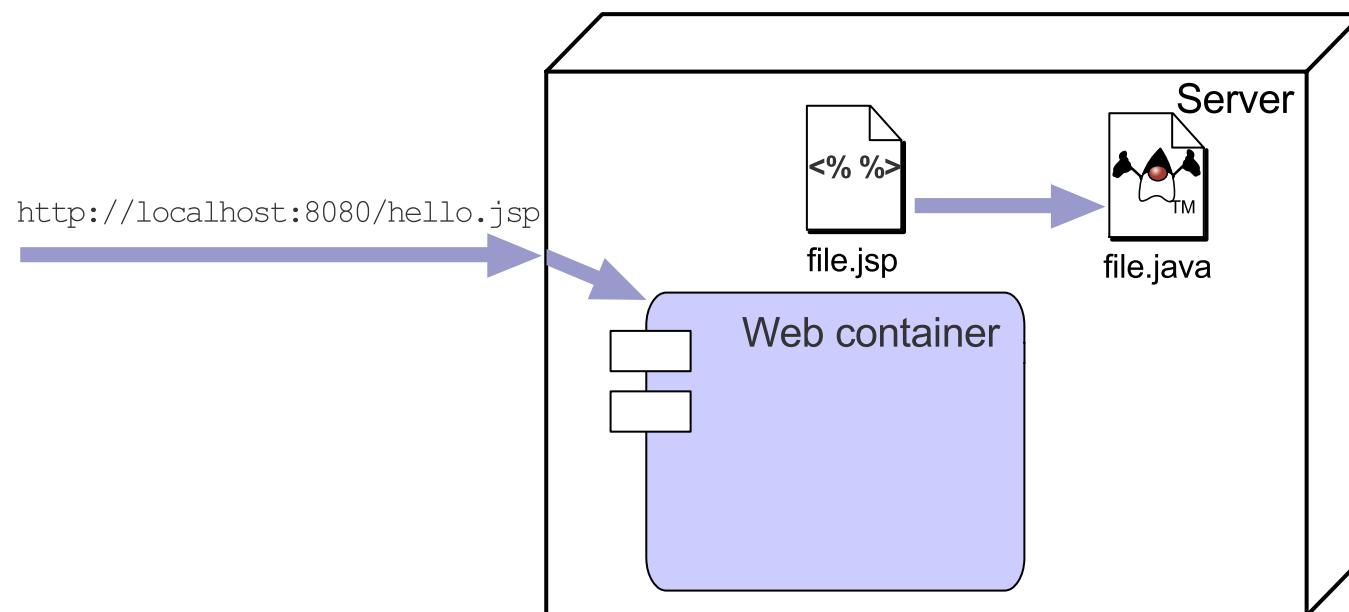
The Hello servlet could be written as the hello.jsp page:

```
1  <%! private static final String DEFAULT_NAME = "World"; %>
2
3  <HTML>
4
5  <HEAD>
6  <TITLE>Hello JavaServer Page</TITLE>
7  </HEAD>
8
9  <%-- Determine the specified name (or use default) --%>
10 <%
11     String name = request.getParameter("name");
12     if ( (name == null) || (name.length() == 0) ) {
13         name = DEFAULT_NAME;
14     }
15 >
16
17 <BODY BGCOLOR='white'>
18
19 <B>Hello, <%= name %></B>
20
21 </BODY>
22
23 </HTML>
```



How a JSP Page Is Processed

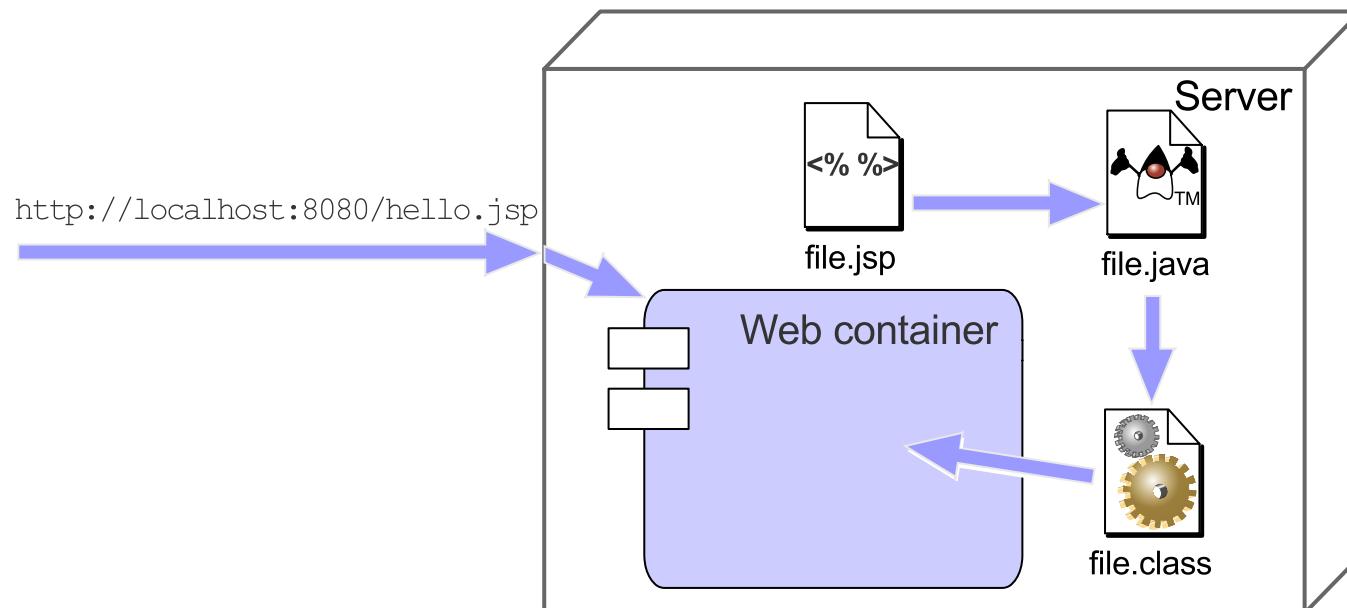
Step 1: The request comes in to the Web container. The JSP page is translated into servlet code.





How a JSP Page Is Processed

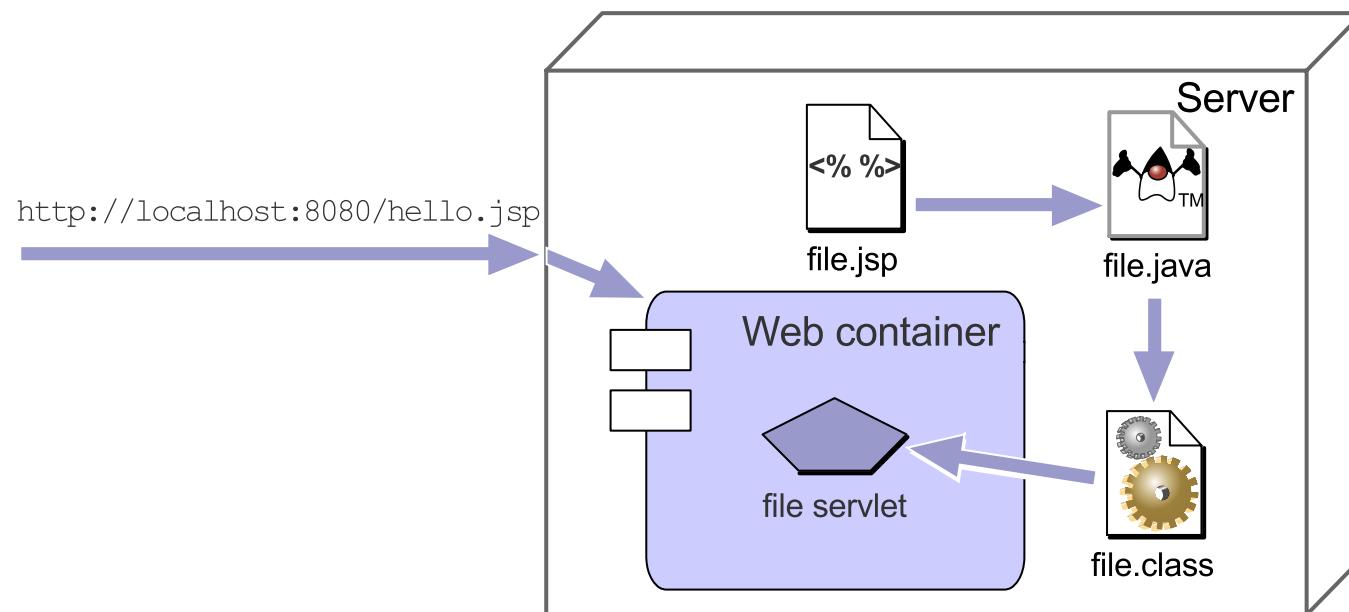
Step 2: The servlet code is compiled and this compiled class file is loaded into the Web container (JVM) environment.





How a JSP Page Is Processed

Step 3: The Web container creates an instance of the servlet class for the JSP page and executes the `jspInit` method.

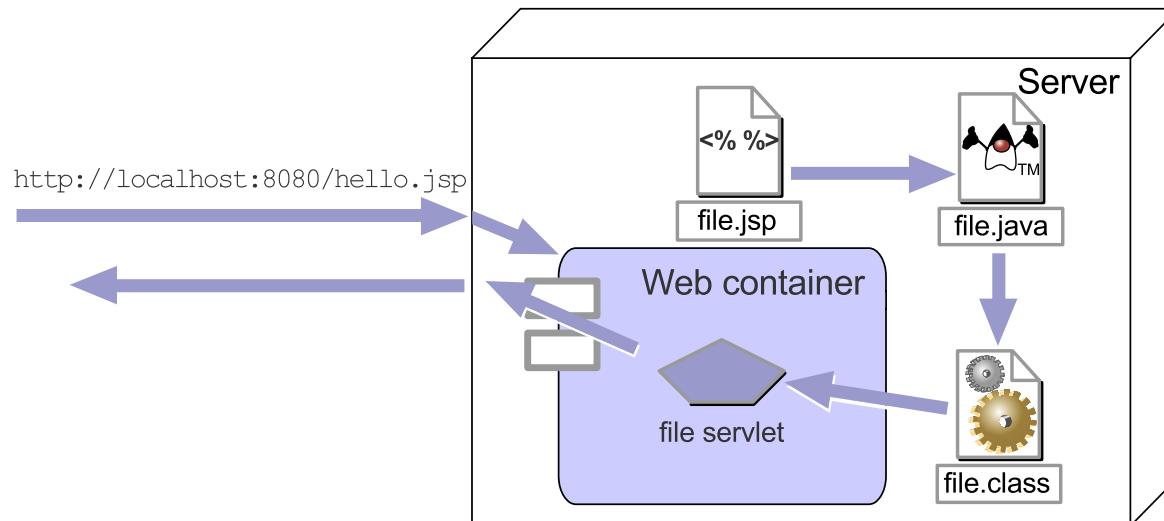


Note: The JSP servlet class might also have a `jspDestroy` method that is called by the Web container at shutdown.



How a JSP Page Is Processed

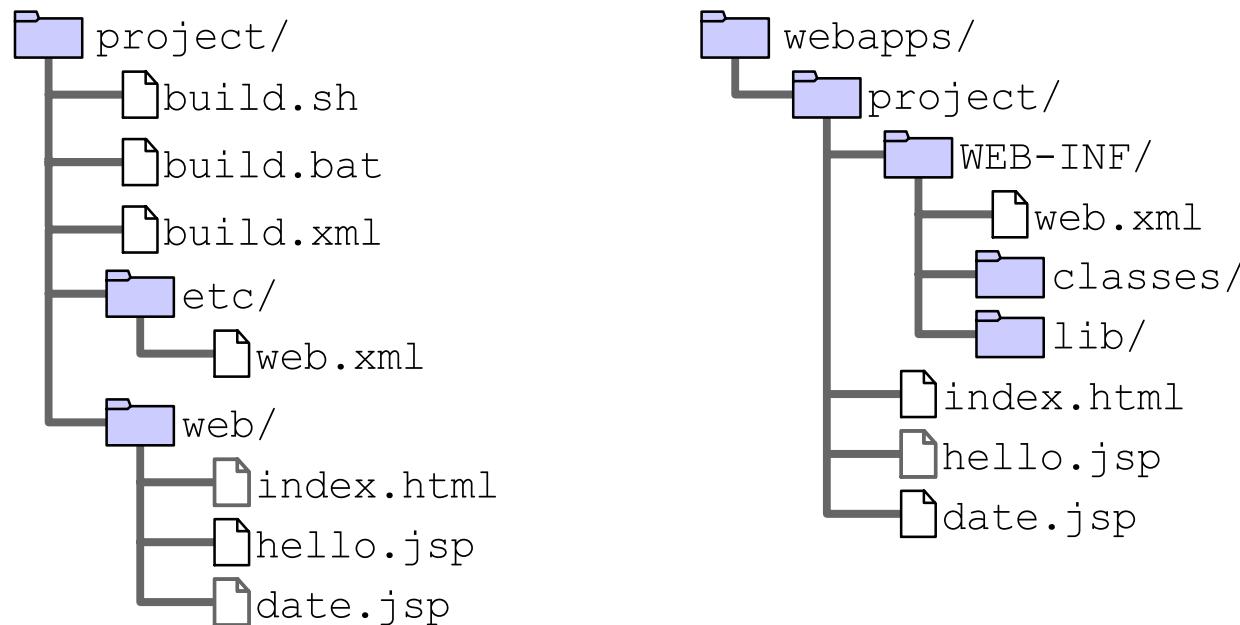
Step 4: The Web container calls the `_jspService` method on the servlet instance for that JSP page; the result is sent back to the user.





Developing and Deploying JSP Pages

Place your JSP files in the web directory during development. They are copied to the main HTML hierarchy at deployment:





Objectives

- Describe JavaServer Page (JSP) technology
- Write JSP code using scripting elements
- Write JSP code using the page directive
- Create and use JSP error pages
- Describe what the Web container is doing behind the scenes in processing a JSP page



JSP Scripting Elements

- JSP scripting elements <% %> are processed by the JSP engine. All other text, outside the scripting elements, is considered part of the response, which is usually HTML.

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

- There are five types of scripting elements:
 - Comments <%-- *comment* --%>
 - Directive tag <%@ *directive* %>
 - Declaration tag <%! *decl* %>
 - Scriptlet tag <% *code* %>
 - Expression tag <%= *expr* %>



Comments

- HTML comments

```
<!-- This is an HTML comment. It will show up in the response. -->
```

- JSP page comments

```
<%-- This is a JSP comment. It will only be seen in the JSP code.  
     It will not show up in either the servlet code or the response.  
--%>
```

- Java technology comments

```
<%  
/* This is a Java comment. It will show up in the servlet code.  
     It will not show up in the response. */  
%>
```



Directive Tag

- Purpose: A directive tag affects the JSP page translation phase.
- Syntax:

```
<%@ DirectiveName [attr="value"]* %>
```

- Examples:

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

```
<%@ include file="incl/copyright.html" %>
```



Declaration Tag

- Purpose: A declaration tag allows the JSP page developer to include declarations at the class-level.
- Syntax:

```
<%! JavaClassDeclaration %>
```

- Examples:

```
<%! public static final String DEFAULT_NAME = "World"; %>
```

```
<%! public String getName(HttpServletRequest request) {  
    return request.getParameter("name");  
}  
%>
```

```
<%! int counter = 0; %>
```



Scriptlet Tag

- Purpose: A scriptlet tag allows the JSP page developer to include arbitrary Java technology code in the `_jspService` method.
- Syntax:

```
<% JavaCode %>
```

- Examples:

```
<% int i = 0; %>
```

```
<% if ( i > 10 ) { %>
```

I is a big number.

```
<% } else { %>
```

I is a small number

```
<% } %>
```



Expression Tag

- Purpose: An expression tag encapsulates a Java technology runtime expression, the value of which is sent to the HTTP response stream.
- Syntax:

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

- Examples:

```
<B>Ten is <%= ( 2 * 5 ) %></B>
```

Thank you, <I><%= name %></I>, for registering for the soccer league.

The current day and time is: <%= new java.util.Date() %>



Implicit Variables

These variables are predefined in the `_jspService` method.

| | |
|-------------|--|
| request | The <code>HttpServletRequest</code> object associated with the request. |
| response | The <code>HttpServletResponse</code> object associated with the response that is sent back to the browser. |
| out | The <code>JspWriter</code> object associated with the output stream of the response. |
| session | The <code>HttpSession</code> object associated with the session for the given user of the request. This variable is only meaningful if the JSP page is participating in an HTTP session. |
| application | The <code>ServletContext</code> object for the Web application. |
| config | The <code>ServletConfig</code> object associated with the servlet for this JSP page. |
| pageContext | This object encapsulates the environment of a single request for this JSP page. |
| page | This variable is equivalent to the <code>this</code> variable in the Java programming language. |
| exception | The <code>Throwable</code> object that was thrown by some other JSP page. This variable is only available in a "JSP error page." |



Objectives

- Describe JavaServer Page (JSP) technology
- Write JSP code using scripting elements
- **Write JSP code using the page directive**
- Create and use JSP error pages
- Describe what the Web container is doing behind the scenes in processing a JSP page



The page Directive

- The page directive is used 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.
- You can place a page directive anywhere in the JSP file. It is a good practice to make the page directive the first statement in the JSP file.



The page Directive

The page directive defines a number of page-dependent properties and communicates these to the Web container at translation time.

| | |
|----------|--|
| language | This attribute defines the scripting language to be used in the page. The value "java" is the only value currently defined and is the default. |
| extends | This defines the (fully-qualified) class name of the superclass of the servlet class that is generated from this JSP page. <i>Do not</i> use this attribute. |
| import | This defines the set of classes and packages that must be imported in the servlet class definition. The value of this attribute is a comma-delimited list of fully-qualified class names or packages. For example: <code>import="java.sql.Date, java.util.* , java.text.*"</code> |
| session | This defines whether the JSP page is participating in an HTTP session. The value may be either true (the default) or false. |
| buffer | This defines the size of the buffer used in the output stream (a <code>JspWriter</code> object). The value is either none or <i>N</i> kb. The default buffer size is 8 KB or greater. For example: <code>buffer="8kb"</code> or <code>buffer="none"</code> |



The page Directive

| | |
|--------------|---|
| autoFlush | This defines whether the buffer output is flushed automatically when the buffer is filled or whether an exception is thrown. The value is either <code>true</code> (automatically flush) or <code>false</code> (throw an exception). The default is <code>true</code> . |
| isThreadSafe | The <code>isThreadSafe</code> attribute allows the JSP page developer to declare that the JSP page is thread-safe or not. |
| info | This defines an informational string about the JSP page. |
| errorPage | This indicates another JSP page that will handle all runtime exceptions thrown by this JSP page. The value is a URL that is either relative to the current Web hierarchy or relative to the Web application's context root. For example, <code>errorPage="error.jsp"</code> (this is relative to the current hierarchy) or <code>errorPage="/error/formErrors.jsp"</code> (this is relative to the context root) |
| isErrorPage | This defines that the JSP page has been designed to be the target of another JSP page's <code>errorPage</code> attribute. The value is either <code>true</code> or <code>false</code> (default). All JSP pages that "are an error page" automatically have access to the <code>exception</code> implicit variable. |
| contentType | This defines the MIME type of the output stream. The default is <code>text/html</code> . |
| pageEncoding | This defines the character encoding of the output stream. The default is <code>ISO-8859-1</code> . |



Objectives

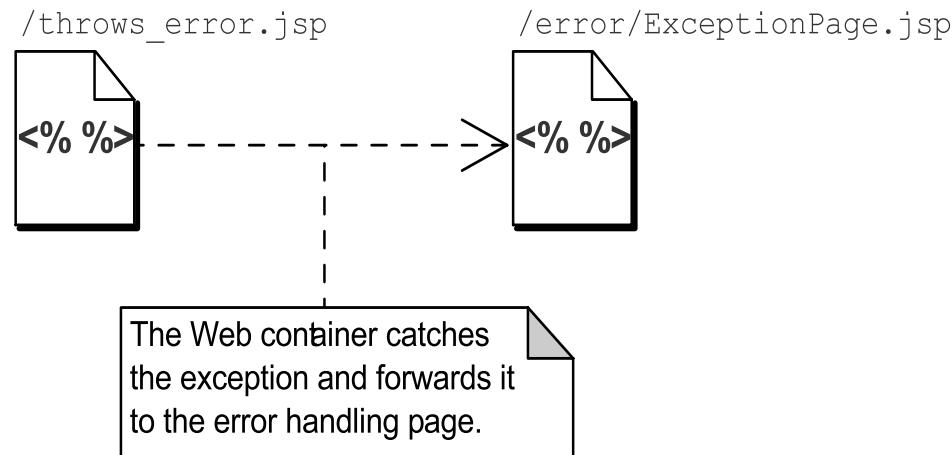
- Describe JavaServer Page (JSP) technology
- Write JSP code using scripting elements
- Write JSP code using the page directive
- **Create and use JSP error pages**
- Describe what the Web container is doing behind the scenes in processing a JSP page



JSP Exception Handling

Because you cannot “wrap” a try-catch block around your JSP page, the JSP specification provides a different mechanism. You can specify an error page in your main pages.

For example:





Declaring an Error Page

In the example, throws_error.jsp declares an error page:

```
1  <%@ page session="false" errorPage="error/ExceptionPage.jsp" %>
2  <%-- This page will cause an "divide by zero" exception --%>
3
4  <HTML>
5
6  <HEAD>
7  <TITLE>Demonstrate Error Pages</TITLE>
8  </HEAD>
9
10 <BODY BGCOLOR='white'>
11
12 <OL>
13 <%
14     for ( int i=10; i > -10; i-- ) {
15     %>
16     <LI><%= 100/i %>
17     <%
18     }
19     %>
20 </OL>
21
22 </BODY>
23
24 </HTML>
```



Developing an Error Page

In the example, `ExceptionPage.jsp` is an error page:

```
1  <%@ page session="false" isErrorPage="true"
2      import="java.io.PrintWriter" %>
3
4  <%-- Using the implicit variable EXCEPTION
5      extract the name of the exception --%>
6 <%
7     String expTypeFullName
8     = exception.getClass().getName();
9     String expTypeName
10    = expTypeFullName.substring(expTypeFullName.lastIndexOf( ".") +1 );
11     String request_uri
12     = (String) request.getAttribute("javax.servlet.error.request_uri");
13 %>
14
15 <HTML>
16
17 <HEAD>
18 <TITLE>JSP Exception Page</TITLE>
19 </HEAD>
20
```



Objectives

- Describe JavaServer Page (JSP) technology
- Write JSP code using scripting elements
- Write JSP code using the page directive
- Create and use JSP error pages
- Describe what the Web container is doing behind the scenes in processing a JSP page



Behind the Scenes

Translated servlet generated from the hello.jsp page:

```
10 public class hello_jsp extends HttpJspBase {  
11  
12     // begin [file="/usr/local/jakarta/tomcat/webapps/jsp/hel-  
13     // lo.jsp";from=(1,3);to=(1,56)]  
14         private static final String DEFAULT_NAME = "World";  
15     // end  
16     static {  
17     }  
18     public hello_jsp( ) {  
19     }  
20  
21     private static boolean _jspx_initited = false;  
22  
23     public final void _jspx_init() throws org.apache.jasper.JasperException {  
24     }  
25  
26     public void _jspService(HttpServletRequest request, HttpServletResponse response)  
27         throws java.io.IOException, ServletException {  
28  
29         JspFactory _jspxFactory = null;  
30         PageContext pageContext = null;  
31         ServletContext application = null;  
32         ServletConfig config = null;  
33         JspWriter out = null;
```



Debugging a JSP Page

There are basically three opportunities for errors in developing JSP pages:

- Translation time (parsing errors)
- Compilation time (errors in the servlet code)
- Runtime (logic errors)



Summary

- A JSP page is like an HTML page with dynamic code in scripting elements. A JSP page is translated into a servlet class, which is then compiled and loaded as a servlet instance to process requests.
- There are five scripting elements:
 - Comments <%-- *comment* --%>
 - Directive tag <%@ *directive* %>
 - Declaration tag <%! *decl* %>
 - Scriptlet tag <% *code* %>
 - Expression tag <%= *expr* %>
- The page directive can be used to modify the translation of the JSP page.



Module 14

Developing Web Applications Using the Model 1 Architecture



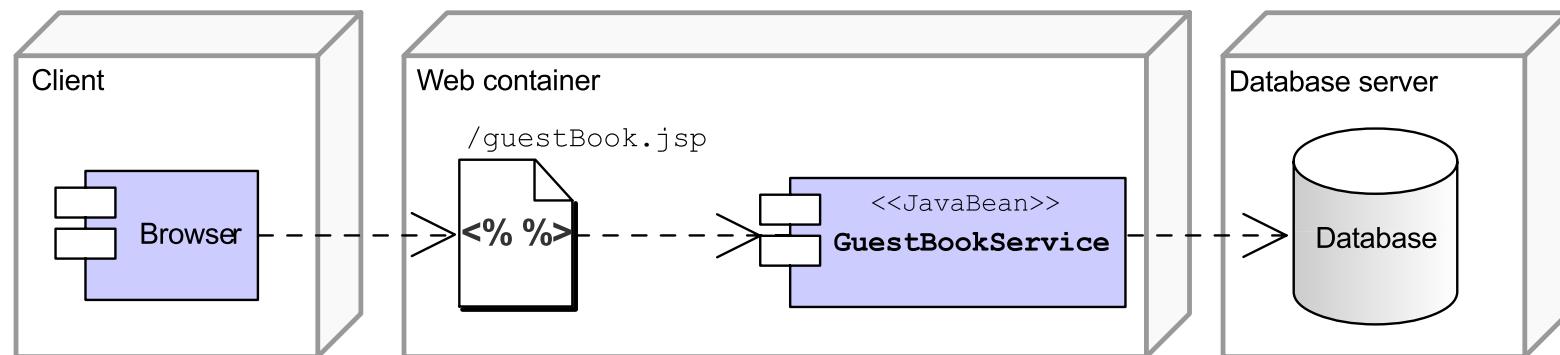
Objectives

- Design a Web application using a Model 1 architecture
- Develop a Web application using a Model 1 architecture



Designing With Model 1 Architecture

The Model 1 architecture uses a JSP page to handle the View and Control aspects of the Web application.

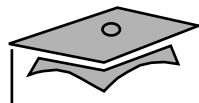


Servlets are not used in this architecture.

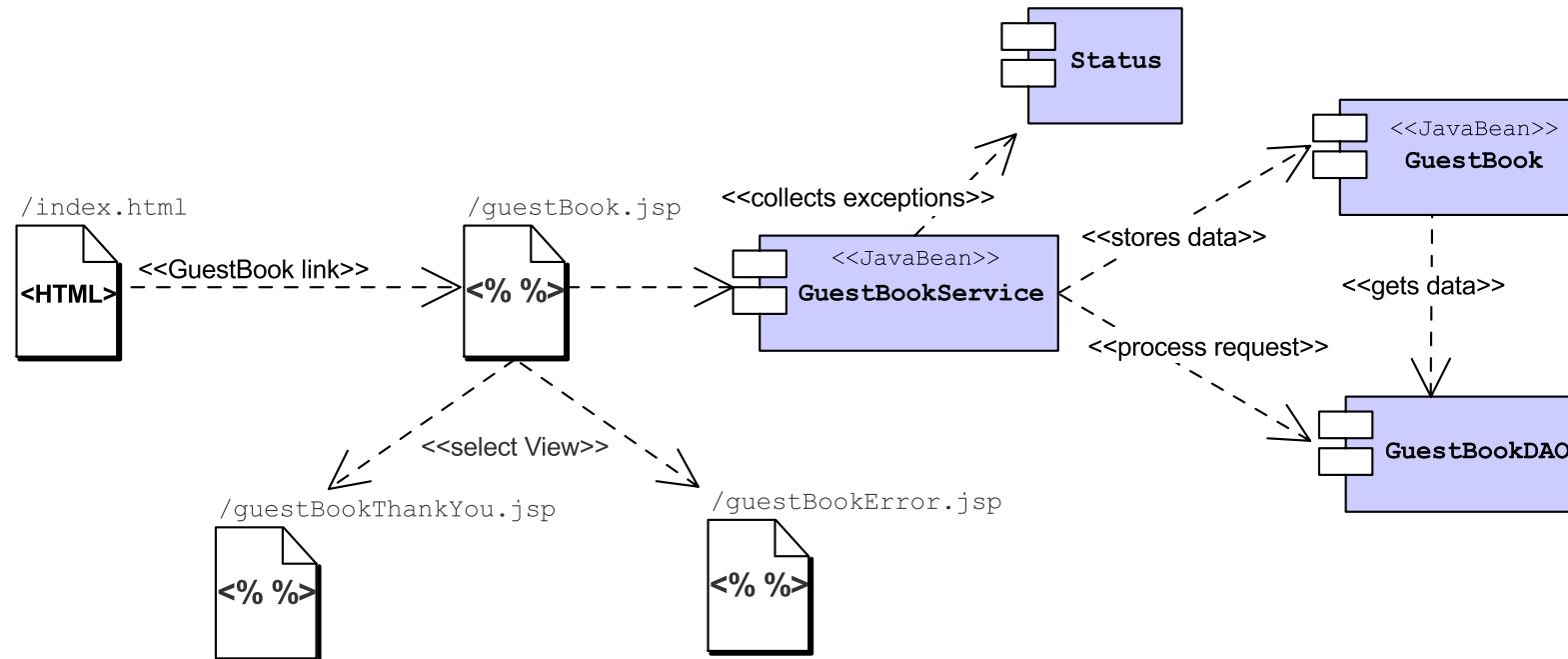


Guest Book Form

The screenshot shows a classic Netscape Communicator browser window. The title bar reads "Netscape: Guest Book". The menu bar includes "File", "Edit", "View", "Go", "Communicator", and "Help". The toolbar has icons for "Bookmarks", "Location" (set to "http://localhost:8080/model1/guestBook.jsp"), and "Java Jacket". Below the toolbar, there's a toolbar with icons for "Admin", "Time Sheet Entry", "SES", "Java", "Tomcat", and "Java Web". The main content area has a purple header bar with the text "Soccer League Guest Book". The form itself contains fields for "Name" (with an input field containing "I"), "Email Address" (with an input field containing ":"), and a large "Comments" text area. At the bottom is a "Submit" button. The status bar at the bottom of the browser window shows "100%" and various icons.



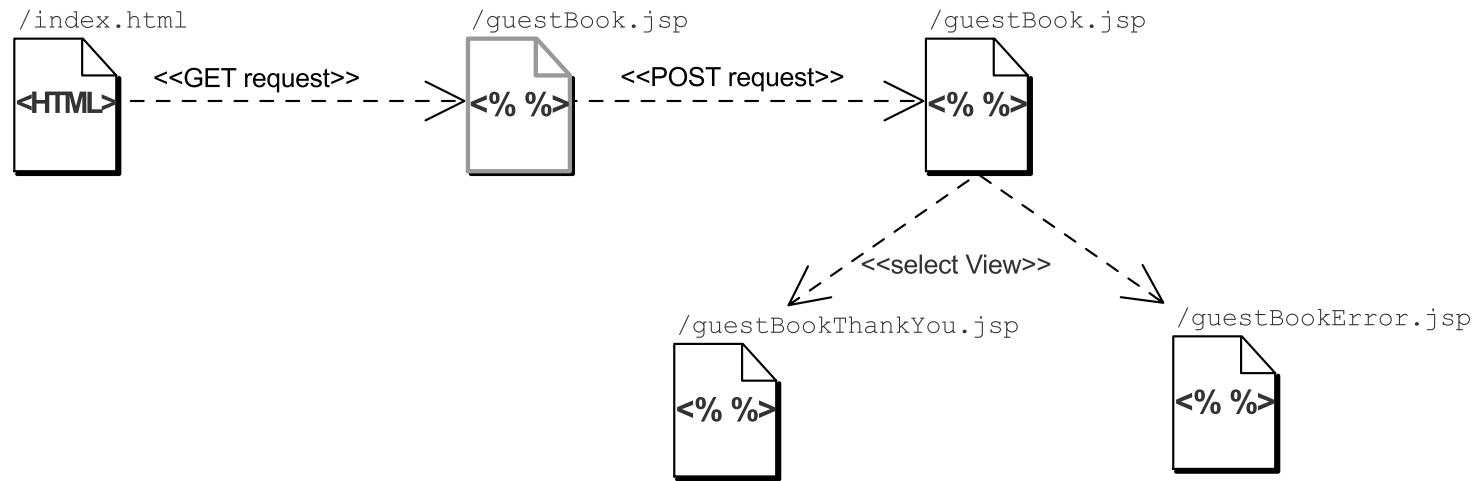
Guest Book Components





Guest Book Page Flow

The guestBook.jsp page acts as both form and processor:



1. The first call to the guestBook.jsp page uses a GET request.
2. The HTML form activates the same guestBook.jsp page now using a POST request.

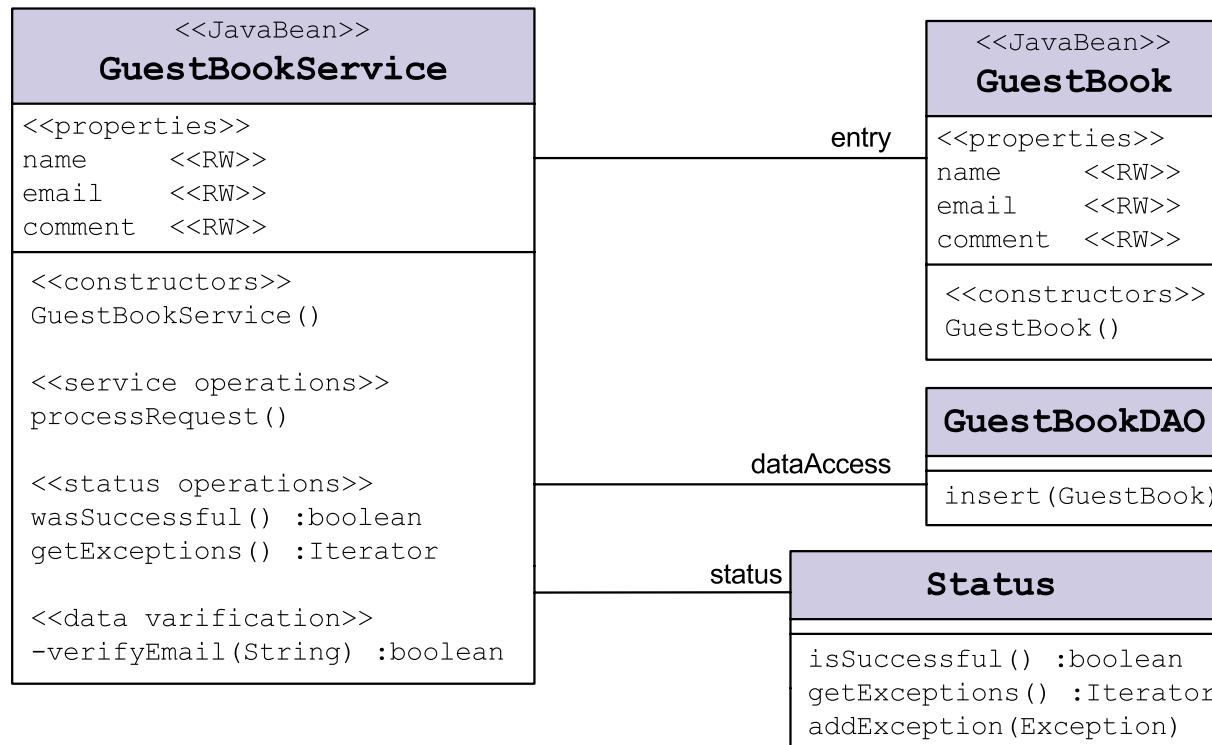


What Is a JavaBeans Component?

- A JavaBeans™ component is a Java technology class with at least the following features:
 - Properties defined with accessors and mutators (get and set methods); for example, a read-write property `firstName` would have `getFirstName` and `setFirstName` methods
 - A no-argument constructor
 - No public instance variables
- JavaBeans components *are not* Enterprise JavaBeans (EJB) components.



The GuestBookService JavaBeans Component





The Guest Book HTML Form

The guestBook.jsp page is activated using a POST request.

```
47
48 <FORM ACTION='guestBook.jsp' METHOD='POST'>
49
50 <TABLE BORDER='0' CELLSPACING='0' CELLPADDING='5' WIDTH='600'>
51 <TR>
52   <TD ALIGN='right'>Name:</TD>
53   <TD><INPUT TYPE='text' NAME='name' SIZE='50'></TD>
54 </TR>
55 <TR>
56   <TD ALIGN='right'>Email Address:</TD>
57   <TD><INPUT TYPE='text' NAME='email' SIZE='50'></TD>
58 </TR>
59 <TR>
60   <TD ALIGN='right'>Comments:</TD>
61   <TD><TEXTAREA NAME='comment' ROWS='5' COLUMNS='70'></TEXTAREA></TD>
62 </TR>
63 <TR HEIGHT='10'><TD HEIGHT='10' COLSPAN='2'><!-- vertical space --></TD></TR>
64 <TR>
65   <TD></TD>
66   <TD><INPUT TYPE='submit' VALUE='Submit'></TD>
67 </TR>
68 </TABLE>
69
70 </FORM>
```



JSP Standard Actions

- XML-like tags are used in JSP pages to perform actions at runtime.
- Syntax:

```
<jsp:action [attr="value"]* />
```

- Examples:

```
<jsp:useBean id="beanName" class="BeanClass" />
<jsp:setProperty name="beanName" property="prop1" value="val" />
<jsp:getProperty name="beanName" property="prop1" />
```

- Standard actions reduce scripting elements in JSP pages. Standard action tag names always begin with the `jsp` prefix.



Creating a JavaBeans Component in a JSP Page

- You can use the `jsp:useBean` standard action to create a JavaBeans component:

```
<jsp:useBean id="guestBookSvc" class="sl314.domain.GuestBookService" />
```

- You can use scriptlet code to initialize the bean properties:

```
3 <jsp:useBean id="guestBookSvc" class="sl314.domain.GuestBookService" scope="request">
4 <%
5   guestBookSvc.setName(request.getParameter("name"));
6   guestBookSvc.setEmail(request.getParameter("email"));
7   guestBookSvc.setComment(request.getParameter("comment"));
8 <%
9 </jsp:useBean>
--
```



Using the jsp:setProperty Action to Initialize a JavaBeans Component

- You can use the `jsp:setProperty` action to initialize the bean properties:

```
3 <jsp:useBean id="guestBookSvc" class="sl314.domain.GuestBookService" scope="request">
4   <jsp:setProperty name="guestBookSvc" property="name" />
5   <jsp:setProperty name="guestBookSvc" property="email" />
6   <jsp:setProperty name="guestBookSvc" property="comment" />
7 </jsp:useBean>
```

- You can use `property="*"` to initialize all of the bean properties from the request parameters:

```
3 <jsp:useBean id="guestBookSvc" class="sl314.domain.GuestBookService" scope="request">
4   <jsp:setProperty name="guestBookSvc" property="*" />
5 </jsp:useBean>
```

- You can also use `jsp:setProperty` outside of the `jsp:useBean` action.



Control Logic in the GuestBook JSP Page

Use scriptlet code to process the request and use the `jsp:forward` standard action to generate a different View.

```
11  <%
12  if ( request.getMethod().equals("POST") ) {
13      guestBookSvc.processRequest();
14
15      if ( guestBookSvc.wasSuccessful() ) {
16          %>
17          <%-- Proceed to the Thank You page. --%>
18          <jsp:forward page="guestBookThankYou.jsp" />
19      <%
20      } else {
21          %>
22          <%-- There was a failure so print the error messages. --%>
23          <jsp:forward page="guestBookError.jsp" />
24      <%
25      } // end of IF guestBookSvc.wasSuccessful()
26  %>
27  <%
28 } // end of IF HTTP Method was POST
29 %>
```



Accessing a JavaBeans Component in a JSP Page

- You can use the `jsp:useBean` standard action to access a JavaBeans component:

```
18 <%-- Retrieve the GuestBookService bean from the request scope. --%>
19 <jsp:useBean id="guestBookSvc" class="s1314.domain.GuestBookService" scope="request" /
20
21 <BR>
22 Thank you, <jsp:getProperty name="guestBookSvc" property="name"/>, for
23 signing our guest book.
```

- The `guestBookSvc` bean had already been created by the `guestBook.jsp` page and placed in the request scope.
- You can use the `jsp:getProperty` standard action to retrieve properties of a JavaBeans component.



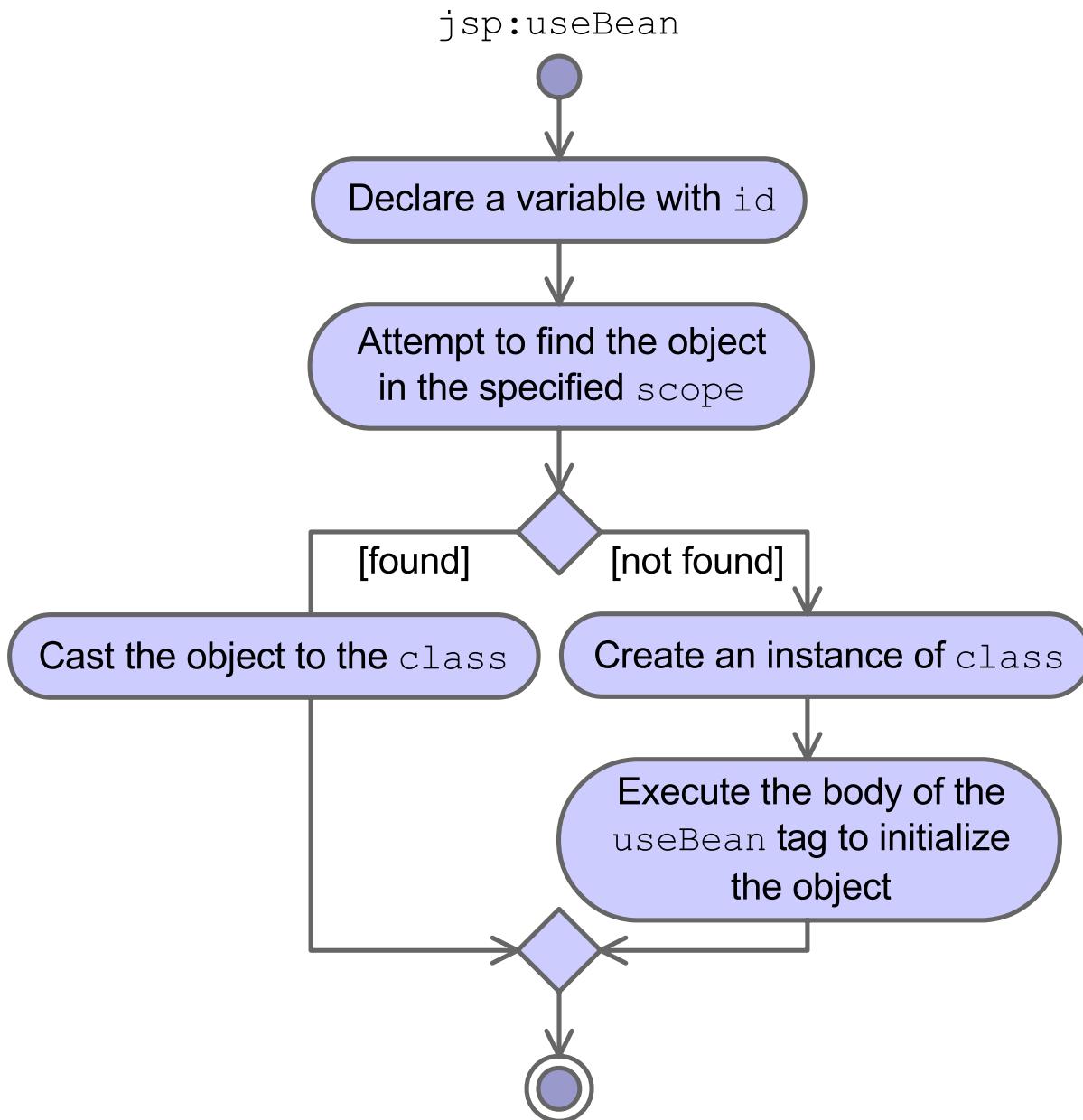
Beans and Scope

A bean can exist in one of four scopes:





Review of the `jsp:useBean` Action





Summary

- The Model 1 architecture uses a JSP page to handle both Control and View aspects. The Model is handled by a service JavaBeans component.
- You create a bean in a JSP page using the `jsp:useBean` standard action.
- You can set properties in the bean using either scriptlet code or the `jsp:setProperty` standard action.
- One JSP page can forward to another JSP page using the `jsp:forward` standard action.
- You can access a bean property using the `jsp:getProperty` standard action.



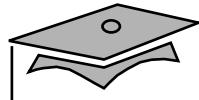
Module 15

Developing Web Applications Using the Model 2 Architecture



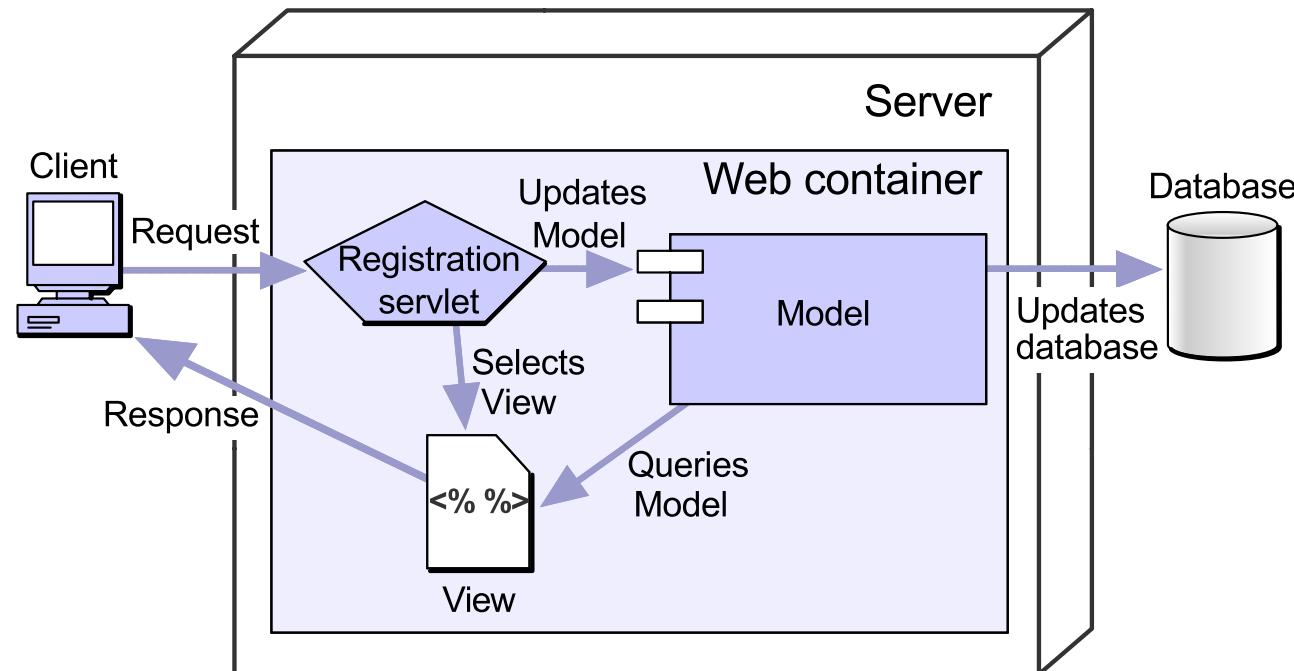
Objectives

- Design a Web application using a Model 2 architecture
- Develop a Web application using a Model 2 architecture



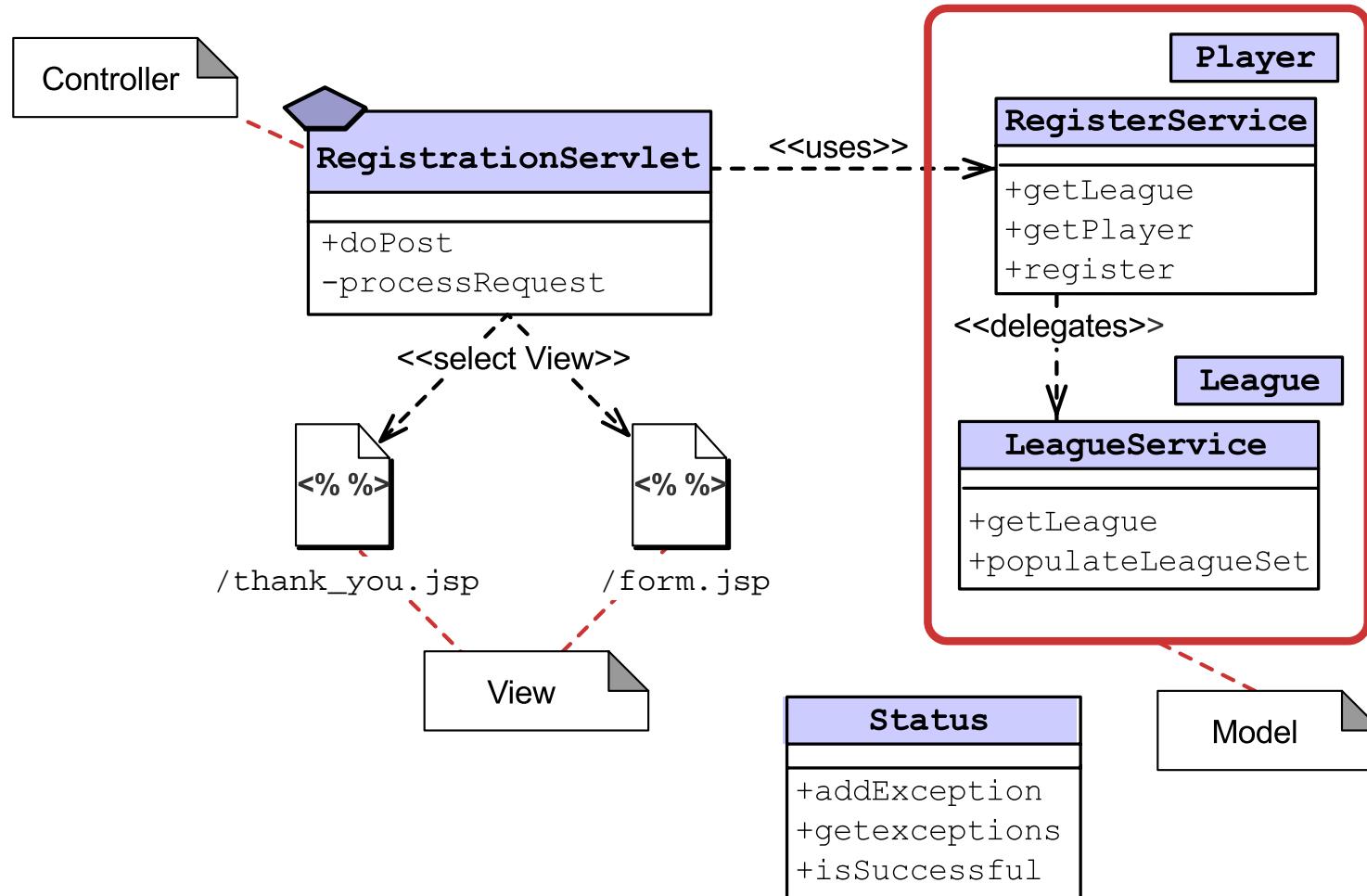
Designing With Model 2 Architecture

The Model 2 architecture uses MVC with JSP pages acting as the Views and a servlet acting as the Controller.



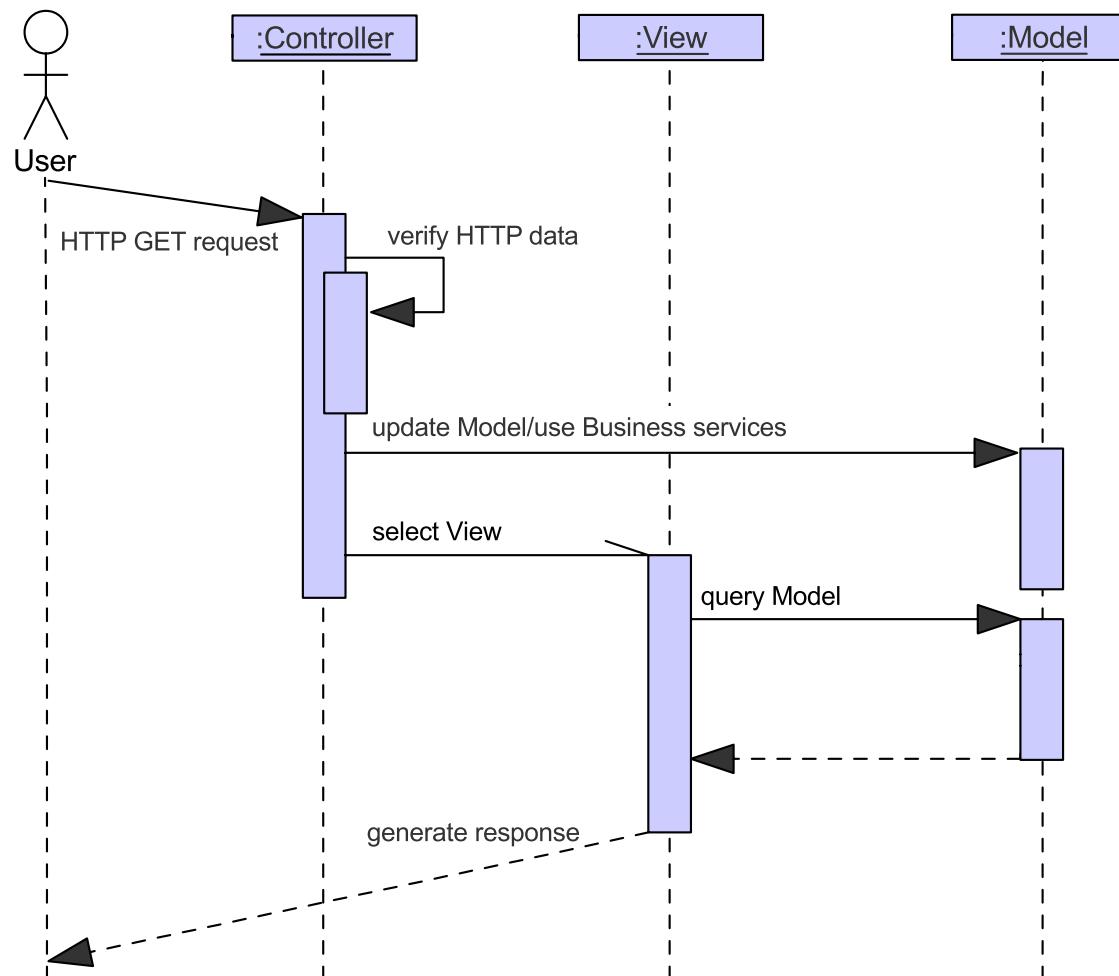


The Soccer League Example Using Model 2 Architecture





Sequence Diagram of Model 2 Architecture





Developing With Model 2 Architecture

- The servlet Controller must:
 - Verify the HTML form data
 - Call the business services in the Model
 - Store domain objects in the request (or session) scope
 - Select the next user View
- The JSP page Views must:
 - Render the user interface (in HTML)
 - Access the domain objects



Controller Details

- The Controller stores domain objects in the request:

```
105     // Now delegate the real work to the RegisterService object  
106     regService.register(league, player, division);  
107     request.setAttribute("league", league);  
108     request.setAttribute("player", player);
```

- The Controller uses a RequestDispatcher to forward to the JSP page View:

```
80      // If any of the above verification failed, then return the  
81      // 'Registration Form' View and return without proceeding with the  
82      // rest of the business logic  
83      if ( ! status.isSuccessful() ) {  
84          view = request.getRequestDispatcher("form.jsp");  
85          view.forward(request, response);  
86          return;  
87      }
```



Request Dispatchers

- Dispatchers from the context object:

```
ServletContext context = getServletContext();
RequestDispatcher servlet = context.getNamedDispatcher("MyServlet");
servlet.forward(request, response);
```

- Dispatchers from the request object:

```
RequestDispatcher view = request.getRequestDispatcher("tools/nails.jsp");
view.forward(request, response);
```



View Details

- The View retrieves the domain objects using a `jsp:useBean` action.

```
18 <%-- Retrieve the LEAGUE and PLAYER beans from the REQUEST scope. --%>
19 <jsp:useBean id="league" scope="request" class="s1314.domain.League"/>
20 <jsp:useBean id="player" scope="request" class="s1314.domain.Player"/>
```

- The View renders the dynamic element of the response using the `jsp:getProperty` action.

```
22 <BR>
23 Thank you, <jsp:getProperty name="player" property="name"/>, for registering
24 in the <B><jsp:getProperty name="league" property="title"/></B> league.
```



Summary

- Model 2 architecture uses the Web-MVC pattern.
- The MVC pattern clearly distinguishes the role of each technology:
 - A servlet is used as the Controller.
 - JSP pages are used as the Views.
 - Java technology classes are used as the Model.
- The servlet Controller passes the domain objects to the View through request (or session) attributes; it selects the next View by using the forward method on a RequestDispatcher object.
- The JSP page View accesses the domain objects using the `jsp:useBean` action.



Module 16

Building Reusable Web Presentation Components



Objectives

- Describe how to build Web page layouts from reusable presentation components
- Write JSP technology code using the `include` directive
- Write JSP technology code using the `jsp:include` standard action



Complex Page Layouts





Complex Page Layouts

Use a hidden table to construct your layout:

```
<BODY>  
<TABLE BORDER='0' CELLPADDING='0' CELLSPACING='0' WIDTH='640'>  
<TR>  
  <TD WIDTH='160'> <!-- logo here --> </TD>  
  <TD WIDTH='480'> <!-- banner here --> </TD>  
</TR>  
<TR>  
  <TD WIDTH='160'> <!-- side-bar menu here --> </TD>  
  <TD WIDTH='480'> <!-- main content here --> </TD>  
</TR>  
<TR>  
  <TD WIDTH='160'> <!-- nothing here --> </TD>  
  <TD WIDTH='480'> <!-- copyright notice here --> </TD>  
</TR>  
</TABLE>  
</BODY>
```



What Does a Fragment Look Like?

A fragment can be any text file that contains static HTML or dynamic JSP technology code:

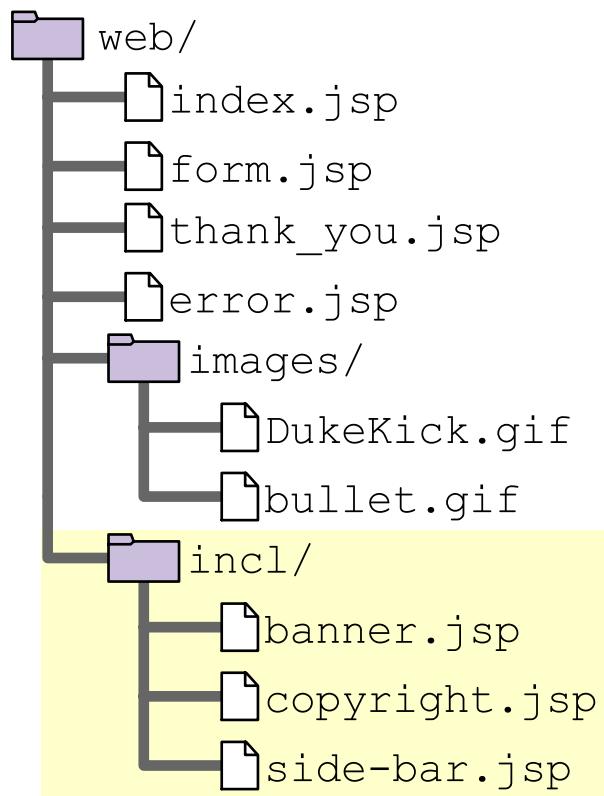
```
1  <%@ page import="java.util.Calendar" %>
2
3  <%-- get today's year --%>
4  <%
5      Calendar today = Calendar.getInstance();
6      int year = today.get(Calendar.YEAR);
7  %>
8
9      <SPACER HEIGHT='15'>
10     <HR WIDTH='50%' SIZE='1' NOSHADE COLOR='blue'>
11     <FONT SIZE='2' FACE='Helvetica, san-serif'>
12         &copy; Duke's Soccer League, 2000-<%= year %>
13     </FONT>
```

Note: No fragments should start with <HTML> or even <BODY> tags.



Organizing Your Presentation Fragments

You should isolate your reusable fragments:





Including JSP Page Fragments

There are two techniques for including presentation fragments in your main JSP pages:

- The `include` directive
- The `jsp:include` standard action



Using the include Directive

- Purpose: The `include` directive allows you to include a fragment into the text of the main JSP page at translation time.
- Syntax:

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

- Example:

```
30  <!-- START of side-bar -->
31  <TD BGCOLOR='#CCCCFF' WIDTH='160' ALIGN='left'>
32      <%@ include file="/incl/side-bar.jsp" %>
33  </TD>
34  <!-- END of side-bar -->
```



Using the jsp:include Standard Action

- Purpose: The jsp:include action allows you to include a fragment into the text of the HTTP response at runtime.
- Syntax:

```
<jsp:include page="fragmentURL" />
```

- Example:

```
20    <!-- START of banner -->
21    <jsp:include page="/incl/banner.jsp" />
22    <!-- END of banner -->
```



Using the jsp:param Standard Action

The screenshot shows a web page for the "Soccer League (Spring '01)". At the top left is a cartoon illustration of a soccer player kicking a ball. To the right of the banner, the text "Soccer League (Spring '01)" and "Thank You!" is displayed. Below the banner, a purple sidebar contains links for "Members" (with options to "Register" or view rosters and schedule) and "Administrators" (with an option to "Create" a new league). The main content area contains the message "Thank you, Bryan, for registering in the Soccer League (Spring '01) league." At the bottom right of the page, there is a copyright notice: "© Duke's Soccer League, 2000-2001".

The `jsp:include` action can take dynamically specified parameters using the `jsp.param` standard action:

```
22    <!-- START of banner -->
23    <jsp:include page="/incl/banner.jsp">
24        <jsp:param name="subTitle" value="Thank You!"/>
25    </jsp:include>
26    <!-- END of banner -->
```



Using the jsp:param Standard Action

The subTitle parameter is attached to the request object:

```
1  <%@ page import="sl314.domain.League" %>
2
3  <%-- Determine the page title and sub-title. --%>
4  <%
5      String bannerTitle = "Duke's Soccer League";
6      String subTitle = request.getParameter("subTitle");
7
8      // Use the title of the selected league (if known)
9      League league = (League) request.getAttribute("league");
10     if ( league != null ) {
11         bannerTitle = league.getTitle();
12     }
13 %>
14
15     <FONT SIZE='5' FACE='Helvetica, san-serif'>
16     <%= bannerTitle %>
17     </FONT>
18
19     <% if ( subTitle != null ) { %>
20     <BR><BR>
21     <FONT SIZE='4' FACE='Helvetica, san-serif'>
22     <%= subTitle %>
23     </FONT>
24     <% } %>
```



Summary

- Complex layouts reuse fragments of presentation code throughout a large Web application. For ease of maintenance, isolate these fragments into separate files.
- The `include` directive is used to include fragments at translation time.
- The `jsp:include` action is used to include fragments at runtime.



Module 17

Developing JSP Pages Using Custom Tags



Objectives

- Describe the problem with JSP technology scripting tags
- Given an existing custom tag library, develop a JSP page using this tag library



Job Roles Revisited

Job roles for a large Web application might include:

- *Content Creators* – 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 is almost exclusively Java technology code
- *Business Component Developers* – Responsible for creating the Model elements of the application, which may reside on the Web server or on a remote server (such as an EJB technology server)



Contrasting Custom Tags and Scriptlet Code

Example scriptlet code in the registration form:

```
151  <TD ALIGN='right'>Address:</TD>
152  <TD>
153      <% String addressValue = request.getParameter("address");
154          if ( addressValue == null ) addressValue = ""; %>
155      <INPUT TYPE='text' NAME='address'
156          VALUE='<%= addressValue %>' SIZE='50'>
157  </TD>
```

Equivalent custom tag in the registration form:

```
88  <TD ALIGN='right'>Address:</TD>
89  <TD>
90      <INPUT TYPE='text' NAME='address'
91          VALUE='<soccer:getReqParam name="address"/>' SIZE='50'>
92  </TD>
```



Contrasting Custom Tags and Scriptlet Code

Advantages of custom tags compared to scriptlet code:

- Java technology code is removed from the JSP page.
- Custom tags are reusable components.
- Standard job roles are supported.



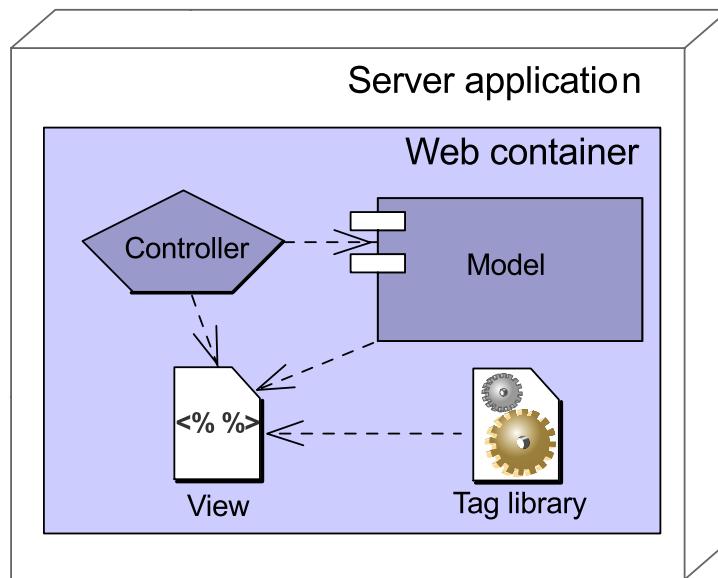
Developing JSP Pages Using Custom Tags

- Use a custom tag library description
- Understand that custom tags follow the XML tag rules
- Declare the tag library in the JSP page and in the Web application deployment descriptor
- Use an empty custom tag in a JSP page
- Use a custom tag in a JSP page to make a section of the HTML response conditional
- Use a custom tag in a JSP page to iterate over a section of the HTML response



What Is a Custom Tag Library?

A custom tag library is a Web component that contains a tag library descriptor file and all associated tag handler classes:



Custom tag handlers used in a JSP page can access any object that is also accessible to the JSP page, such as attributes in the request and session scopes.



Custom Tag Syntax Rules

Custom tags use XML syntax.

- Standard tags (containing a body):

```
<prefix:name {attribute={"value" | 'value'} }*>  
    body  
</prefix:name>
```

- Empty tags:

```
<prefix:name {attribute={"value" | 'value'} }*> />
```

- Tag names, attributes, and prefixes are case sensitive.
- Tags must follow nesting rules:

```
<tag1>  
    <tag2>  
    </tag2>  
</tag1>
```



Example Tag Library: Soccer League

The `getReqParam` tag inserts the value of the named request parameter into the output. If the parameter does not exist, then either the default is used (if provided) or the empty string is used.

- **Body content:** This is an empty tag.
- **Attribute: name**

This mandatory attribute is the name of the request parameter.

- **Attribute: default**

This optional attribute can provide a default if the parameter does not exist.

- **Example:**

```
<soccer:getReqParam name="countryCode" default="JP" />
```



Example Tag Library: Soccer League

The heading tag generates a hidden HTML table that creates a colorful, and well-formatted page heading.

- **Body content:** This tag contains JSP technology code.
- **Attribute: alignment**

This mandatory attribute specifies the text alignment of the heading. The acceptable values are: left, center, and right.

- **Attribute: color**

This mandatory attribute specifies the color of the box around the heading. The acceptable values include HTML color names or an RGB code.

- **Example:**

```
<soccer:heading alignment="center" color="#CCCCFF">  
    Soccer League Registration Form  
</soccer:heading>
```



Example Tag Library: Soccer League

The checkStatus tag is used to make the body conditional based upon whether the “status” attribute has been set and if the Status object indicates “was unsuccessful.” This is used when a forms page is revisited (after a form validation error occurred).

- **Body content:** This tag contains JSP technology code.
- **Example:**

```
<soccer:checkStatus>
    <%-- JSP code to show error messages --%>
</soccer:checkStatus>
```



Example Tag Library: Soccer League

The `iterateOverErrors` tag iterates over all of the exception objects in the `Status` object. This tag *must* be used within the `checkStatus` tag.

- **Body content:** This tag contains JSP technology code.
- **Example:**

```
<soccer:checkStatus>
    <soccer:iterateOverErrors>
        <%-- JSP code showing a single error message --%>
    </soccer:iterateOverErrors>
</soccer:checkStatus>
```



Example Tag Library: Soccer League

The `getCurrentMessage` tag is an empty tag that prints the current error message. This tag *must* be used within the `iteratorOverErrors` tag.

- **Body content:** This is an empty tag.
- **Example:**

```
<soccer:checkStatus>
    <soccer:iterateOverErrors>
        <soccer:getCurrentMessage/>
    </soccer:iterateOverErrors>
</soccer:checkStatus>
```



Developing JSP Pages Using a Custom Tag Library

Use the `taglib` element in the deployment descriptor to declare that the Web application makes use of a tag library.

```
61   <taglib>
62     <taglib-uri>http://www.soccer.org/taglib</taglib-uri>
63     <taglib-location>/WEB-INF/taglib.tld</taglib-location>
64   </taglib>
```

Use the `taglib` directive in the JSP page to identify which tag library is being used and which prefix to use for those custom tags.

```
1  <%@ page session="false" %>
2  <%@ taglib uri="http://www.soccer.org/taglib" prefix="soccer" %>
```

Any number of tag libraries may be included in a JSP page, but each must have a unique prefix.



Using an Empty Custom Tag

An empty tag is often used to embed simple dynamic content:

```
81  <TD ALIGN='right'>Name:</TD>
82  <TD>
83      <INPUT TYPE='text' NAME='name'
84          VALUE='<soccer:getReqParam name="name"/>' SIZE='50'>
85  </TD>
```

Note that the slash (/) is at the end of the tag.



Using a Conditional Custom Tag

Partial scriptlet code in the registration form:

```
<%
  if ( (status != null) && !status.isSuccessful() ) {
%>
<%-- JSP code --%>
<%
  } // end of IF status
%>
```

Equivalent custom tag in the registration form:

```
<soccer:checkStatus>
<%-- JSP code --%>
</soccer:checkStatus>
```



Using an Iterative Custom Tag

Example scriptlet code in the registration form:

```
26 There were problems processing your request:  
27 <UL>  
28 <%  
29     Iterator errors = status.getExceptions();  
30     while ( errors.hasNext() ) {  
31         Exception ex = (Exception) errors.next();  
32     %>  
33     <LI><%= ex.getMessage() %>  
34     <%  
35     } // end of WHILE loop  
36 %>  
37 </UL>
```

Equivalent custom tag in the registration form:

```
19 There were problems processing your request:  
20 <UL>  
21     <soccer:iteratorOverErrors>  
22         <LI><soccer:getErrorMessage/>  
23     </soccer:iteratorOverErrors>  
24 </UL>
```



Summary

- Custom tag libraries provide a mechanism to completely remove scriptlet code from your JSP pages.
- Follow these steps to use a tag library in your Web application:
 - Use the `taglib` element in the deployment descriptor
 - Use the `taglib` directive in the JSP page to identify which tag library is being used and by which prefix
 - Use XML syntax (plus the prefix) when using custom tags



Module 18

Developing a Simple Custom Tag



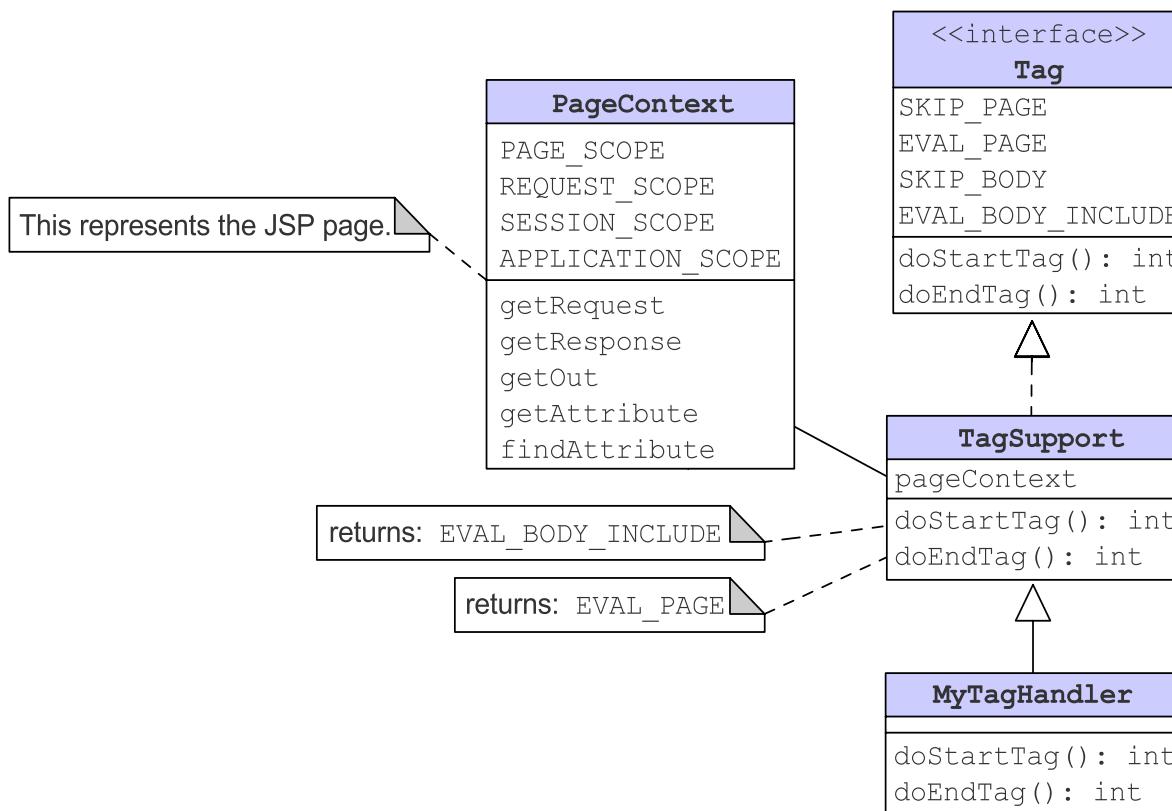
Objectives

- Describe the structure and execution of a custom tag in a JSP page
- Develop the tag handler class for a simple, empty custom tag
- Write the tag library description for a simple, empty custom tag
- Develop a custom tag that includes its body in the content of the HTTP response



Fundamental Tag Handler API

The tag handler API allows you to create your own tag handler classes:





Tag Handler Life Cycle

Three elements in the JSP page:

```
<tag attr="value"> ← Start tag: create tag handler object, initialize attributes, doStartTag  
    body ← If EVAL_BODY_INCLUDE then process body  
</tag> ← End tag: doEndTag, release tag handler object
```

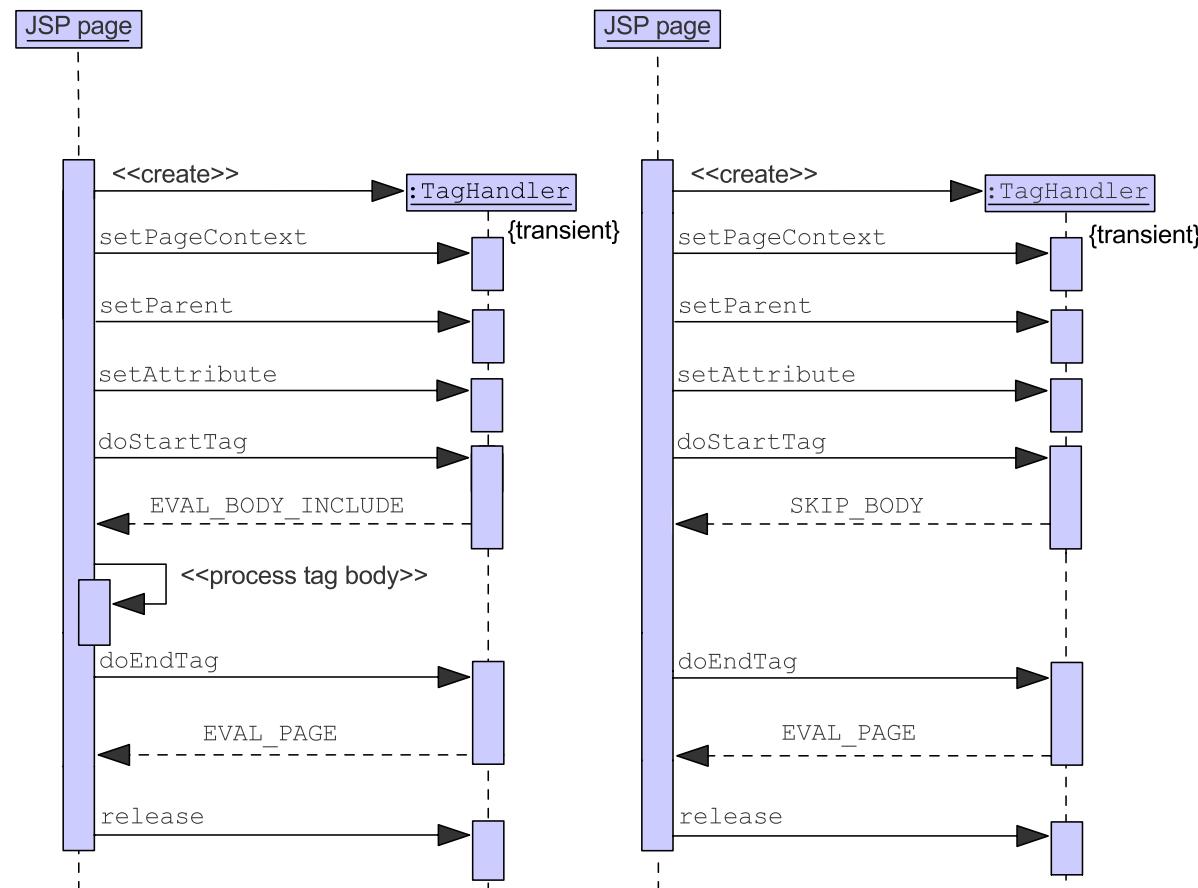
Pseudo-code for tag handler execution in the JSP servlet code:

```
1  TagHandler tagObj = new TagHandler();  
2  tagObj.setPageContext(pageContext);  
3  tagObj.setAttr("value");  
4  try {  
5      int startTagResult = tagObj.doStartTag();  
6      if ( startTagResult != Tag.SKIP_BODY ) {  
7          out.write("body"); // process the BODY of the tag  
8      }  
9      if ( tagObj.doEndTag() == Tag.SKIP_PAGE ) {  
10         return; // do not continue processing the JSP page  
11     }  
12 } finally {  
13     tagObj.release();  
14 }
```



Tag Handler Life Cycle

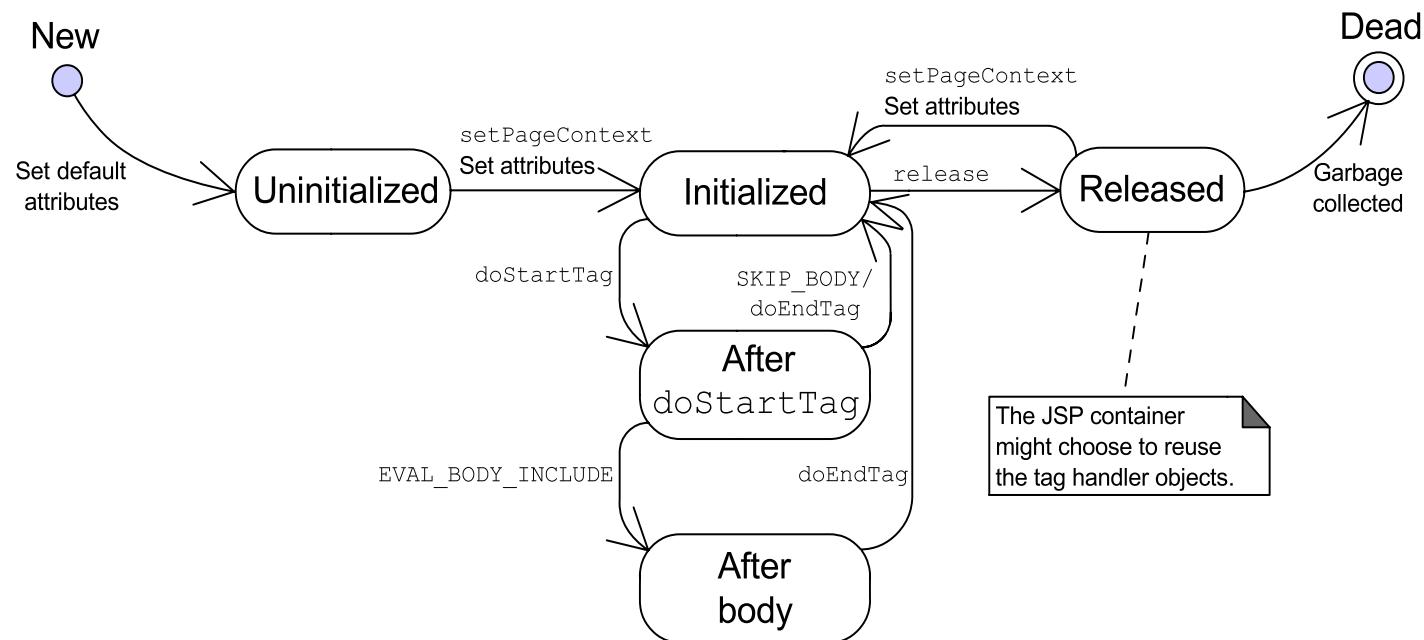
Sequence diagrams of a tag handler life cycle:





Tag Handler Life Cycle

State diagram of a tag handler object:





Tag Library Relationships

JSP page

```
<%@ taglib prefix="soccer" uri="http://www.soccer.org/taglib" %>  
  
<soccer:getReqParam name="countryCode" default="JP"/>
```

Deployment descriptor

```
<taglib>  
  <taglib-uri>  
    http://www.soccer.org/taglib  
  </taglib-uri>  
  <taglib-location>  
    WEB-INF/taglib.tld  
  </taglib-location>  
</taglib>
```

Tag library descriptor

```
<tag>  
  <name>getReqParam</name>  
  <tag-class>sl314.web.taglib.GetRequestParamHandler</tag-class>  
  <body-content>empty</body-content>  
  <attribute>  
    <name>name</name>  
    <required>true</required>  
  </attribute>  
  <attribute>  
    <name>default</name>  
    <required>false</required>  
  </attribute>  
</tag>
```

Tag handler class

| |
|--|
| GetRequestParamHandler {from sl314.web.taglib} |
| setName(String) setDefault(String) doStartTag():int release() |



Developing a Tag Handler Class

- Extend the TagSupport class.
- Provide private instance variables for each tag attribute. Provide an explicit, default value for all attributes that are not required in the tag definition.
- Create mutator methods, `setXyz(String)`, for each tag attribute.
- Override the `doStartTag` method to handle the start tag processing.
- Override the `release` method to reset all attributes instance variables back to their default values.



The getReqParam Tag

The getReqParam inserts the value of the named request parameter into the output. If the parameter does not exist, then either the default is used (if provided) or the empty string is used.

- **Body content:** This is an empty tag.
- **Attribute: name**

This mandatory attribute is the name of the request parameter.

- **Attribute: default**

This optional attribute can provide a default if the parameter does not exist.

- **Example:**

```
<soccer:getReqParam name="countryCode" default="JP"/>
```



The getReqParam Tag Handler Class

Place the tag handler class in an appropriate package. Import the necessary classes. Extend the TagSupport class.

```
1 package sl314.web.taglib;
2
3 // Servlet imports
4 import javax.servlet.jsp.tagext.TagSupport;
5 import javax.servlet.jsp.JspWriter;
6 import javax.servlet.jsp.JspException;
7 import javax.servlet.ServletRequest;
8 import java.io.IOException;
9
10
11 /**
12  * This class handles the "get request parameter" tag.
13  * This is an empty tag.
14 */
15 public class GetRequestParamHandler extends TagSupport {
```



The getReqParam Tag Handler Class

Create instance variables for each tag attribute. Create mutator methods for each tag attribute.

```
15 public class GetRequestParamHandler extends TagSupport {  
16  
17     private String name;  
18     private String defaultValue = "";  
19  
20     public void setName(String name) {  
21         this.name = name;  
22     }  
23  
24     public void setDefault(String defaultValue) {  
25         this.defaultValue = defaultValue;  
26     }
```

The JSP technology container uses introspection to determine the proper mutator method to use for a given tag attribute. If the attribute is called xyz, then the mutator method must be called setXyz.



The getReqParam Tag Handler Class

Override the doStartTag method to process the tag.

```
28  public int doStartTag() throws JspException {
29      ServletRequest request = pageContext.getRequest();
30      String paramValue = request.getParameter(name);
31      JspWriter out = pageContext.getOut();
32
33      try {
34          if ( paramValue == null ) {
35              out.print(defaultValue);
36          } else {
37              out.print(paramValue);
38          }
39      } catch (IOException ioe) {
40          throw new JspException(ioe);
41      }
42
43      // This is an empty tag, skip any body
44      return SKIP_BODY;
```



The getReqParam Tag Handler Class

Override the release method to reset the default attribute values.

```
47  public void release() {  
48      defaultValue = "";  
49  }  
50 } // end of GetRequestParamHandler class
```



Configuring the Tag Library Descriptor

- Basic structure of the TLD file:

```
1  <?xml version="1.0" encoding="ISO-8859-1" ?>
2  <!DOCTYPE taglib
3      PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
4      "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_2.dtd">
5
6  <taglib>
7
8      <tlib-version>1.2</tlib-version>
9      <jsp-version>1.2</jsp-version>
10     <short-name>Sports League Web Taglib</short-name>
11     <uri>http://www.soccer.org/taglib</uri>
12     <description>
13         An example tag library for the Soccer League Web Application.
14     </description>
15
16     <tag>
17         <!-- tag declaration -->
18     </tag>
19
20 </taglib>
```

- Create any number of tag declaration elements.



Tag Declaration Element

- Basic structure of a tag declaration element:

```
1  <tag>
2      <name>getReqParam</name>
3      <tag-class>s1314.web.taglib.GetRequestParamHandler</tag-class>
4      <body-content>empty</body-content>
5      <description>
6          This tag inserts into the output the value of the named
7          request parameter. If the parameter does not exist, then
8          either the default is used (if provided) or the empty string.
9      </description>
10     <attribute>
11         <name>name</name>
12         <required>true</required>
13         <rteprvalue>false</rteprvalue>
14     </attribute>
15     <attribute>
16         <name>default</name>
17         <required>false</required>
18         <rteprvalue>false</rteprvalue>
19     </attribute>
20 </tag>
```



Custom Tag Body Content

The body-content element must contain one of these three values:

- empty – This value tells the JSP technology engine that the tag does not accept any body content.
- JSP – This value tells the JSP technology engine that the tag can accept arbitrary JSP technology code in its body.
- tag-dependent – This value tells the JSP technology engine that the tag can accept arbitrary content in the tag body. The JSP technology engine will not process the body, but will pass it directly to the tag handler.



Custom Tag Attributes

The attribute element contains three sub-elements:

- name – The name of the attribute (case-sensitive).
- required – Whether the attribute must be used in every tag use in a JSP page.
- rtxexprvalue – Whether the attribute value might be generated from JSP technology code at runtime. For example, if you had a tag that generated HTML headings, then you might want to include an attribute, level, which takes a number.

```
<prefix:heading level="<% currentHeading %>">  
    Act IV - Romeo Awakes  
</prefix:heading>
```



Custom Tag That Includes the Body

- Here is the desired look and feel for a page heading:



- Here is the HTML code the tag must generate:

```
11 <TABLE BORDER='0' CELLSPACING='0' CELLPADDING='0' WIDTH='600'>
12 <TR>
13   <TD BGCOLOR='#CCCCFF' ALIGN='center'>
14     <H3>Soccer League Registration Form</H3>
15   </TD>
16 </TR>
17 </TABLE>
```

- Here is the JSP page custom tag used:

```
12 <soccer:heading alignment="center" color="#CCCCFF">
13   Soccer League Registration Form
14 </soccer:heading>
```



The heading Tag

The heading tag generates a hidden HTML table that creates a colorful, and well-formatted page heading.

- **Body content:** This tag contains JSP technology code.
- **Attribute: alignment**

This mandatory attribute specifies the text alignment of the heading. The acceptable values are: left, center, and right.

- **Attribute: color**

This optional attribute specifies the color of the box around the heading. The acceptable values include HTML color names or an RGB code.

- **Example:**

```
<soccer:heading alignment="center" color="#CCCCFF">  
    Soccer League Registration Form  
</soccer:heading>
```



The heading Tag Handler Class

The tag handler attributes:

```
11  /**
12   * This class handles the "heading" tag.
13   * This tag takes a body, which is the text of the heading.
14   */
15  public class HeadingHandler extends TagSupport {
16
17      private static String DEFAULT_COLOR = "white";
18
19      private String alignment;
20      private String color = DEFAULT_COLOR;
21
22      public void setAlignment(String alignment) {
23          this.alignment = alignment;
24      }
25
26      public void setColor(String color) {
27          this.color = color;
28      }
29
```



The heading Tag Handler Class

```
29
30     public int doStartTag() throws JspException {
31         JspWriter out = pageContext.getOut();
32         try {
33             out.println("<TABLE BORDER='0' CELLSPACING='0' "
34                         + "CELLPADDING='0' WIDTH='600'>");
35             out.println("<TR ALIGN='" + alignment
36                         + "' BGCOLOR='" + color + "'>");
37             out.println("  <TD><H3>");
38         } catch (IOException ioe) {
39             throw new JspException(ioe);
40         }
41         // Tell the JSP page to include the tag body
42         return EVAL_BODY_INCLUDE;
43     }
44     public int doEndTag() throws JspException {
45         JspWriter out = pageContext.getOut();
46         try {
47             out.println("  </H3></TD>");
48             out.println("</TR>");
49             out.println("</TABLE>");
50         } catch (IOException ioe) {
51             throw new JspException(ioe);
52         }
53         // Continue processing the JSP page
54         return EVAL_PAGE;
55     }
```



The heading Tag Descriptor

The tag declaration:

```
37   <tag>
38     <name>heading</name>
39     <tag-class>sl314.web.taglib.HeadingHandler</tag-class>
40     <body-content>JSP</body-content>
41     <description>
42       This tag creates a customizable page heading.
43     </description>
44     <attribute>
45       <name>alignment</name>
46       <required>true</required>
47       <rteprvalue>false</rteprvalue>
48     </attribute>
49     <attribute>
50       <name>color</name>
51       <required>false</required>
52       <rteprvalue>false</rteprvalue>
53     </attribute>
54   </tag>
```



Summary

A tag library requires the coordination of four components: the JSP page, the Web application deployment descriptor, the tag library descriptor, and the tag handler classes.

The tag handler class:

- Should extend the TagSupport class
- Must contain mutator methods, `setXyz(String)`, for every tag attribute
- Must override the `doStartTag` method
- Should override the `release` method to reset tag attributes to their default values



Summary

The tag library descriptor:

- Must include the URI of the tag library
- Must include an entry for each tag in the library
- Each tag entry must include:
 - The name element
 - The tag-class element
 - The body-content element
 - An attribute entry for every tag attribute



Module 19

Developing Advanced Custom Tags



Objectives

- Develop a custom tag in which the body is conditionally included
- Develop a custom tag in which the body is iteratively included



Writing a Conditional Custom Tag

- A conditional tag tells the JSP page to include the body of the tag, if a certain condition is met.
- The conditional aspect of the tag is implemented through the `doStartTag` method:
 - If the condition is true, the `doStartTag` method should return `EVAL_BODY_INCLUDE`.
 - If it is false, it should return `SKIP_BODY`.



Example: The checkStatus Tag

The checkStatus tag is used to make the body conditional based upon whether the “status” attribute has been set and if the Status object indicates “was unsuccessful.” This is used when a forms page is revisited (after a form validation error occurred).

- **Body content:** This tag contains JSP technology code.
- **Example:**

```
<soccer:checkStatus>
    <%-- JSP code to show error messages --%>
</soccer:checkStatus>
```



Example: The checkStatus Tag

Here is how the checkStatus tag is used in the League Registration Form:

```
16 <%-- BEGIN: error status presentation --%>
17 <soccer:checkStatus>
18 <FONT COLOR='red'>
19 There were problems processing your request:
20 <UL>
21   <soccer:iteratorOverErrors>
22     <LI><soccer:getErrorMessage/>
23   </soccer:iteratorOverErrors>
24 </UL>
25 </FONT>
26 </soccer:checkStatus>
27 <%-- END: error status presentation --%>
```



The checkStatus Tag Handler

The tag handler class:

```
18 public class CheckStatusHandler extends TagSupport {  
19  
20     public int doStartTag() {  
21         Status status  
22             = (Status) pageContext.getAttribute("status",  
23                           PageContext.REQUEST_SCOPE);  
24  
25         if ( (status != null) && !status.isSuccessful() ) {  
26             return EVAL_BODY_INCLUDE;  
27         } else {  
28             return SKIP_BODY;  
29         }  
30     }  
31 }
```

The body of the tag is only included if the status was unsuccessful.

The `getAttribute` method on the `pageContext` object is used to retrieve the `Status` object from the request scope.



The checkStatus Tag Life Cycle

Pseudo-code for tag handler execution in the JSP servlet code:

```
1  CheckStatusHandler tagObj = new CheckStatusHandler();
2  tagObj.setPageContext(pageContext);
3  try {
4      int startTagResult = tagObj.doStartTag();
5      if ( startTagResult != SKIP_BODY ) {
6          // process the BODY of the tag
7      }
8      if ( tagObj.doEndTag() == SKIP_PAGE ) {
9          return; // do not continue processing the JSP page
10     }
11 } finally {
12     tagObj.release();
13 }
```



Writing an Iterator Custom Tag

- An iterator custom tag tells the JSP page to process the body of the tag multiple times, until the iteration is complete.
- An iteration (in general) requires five basic steps:

```
1 Iterator elements = getIterator(); // INITIALIZATION
2 while ( elements.hasNext() ) {           // ITERATION TEST
3     Object obj = elements.next();        // GET NEXT OBJECT
4     process(obj);                      // PROCESS THE OBJECT
5 }
```



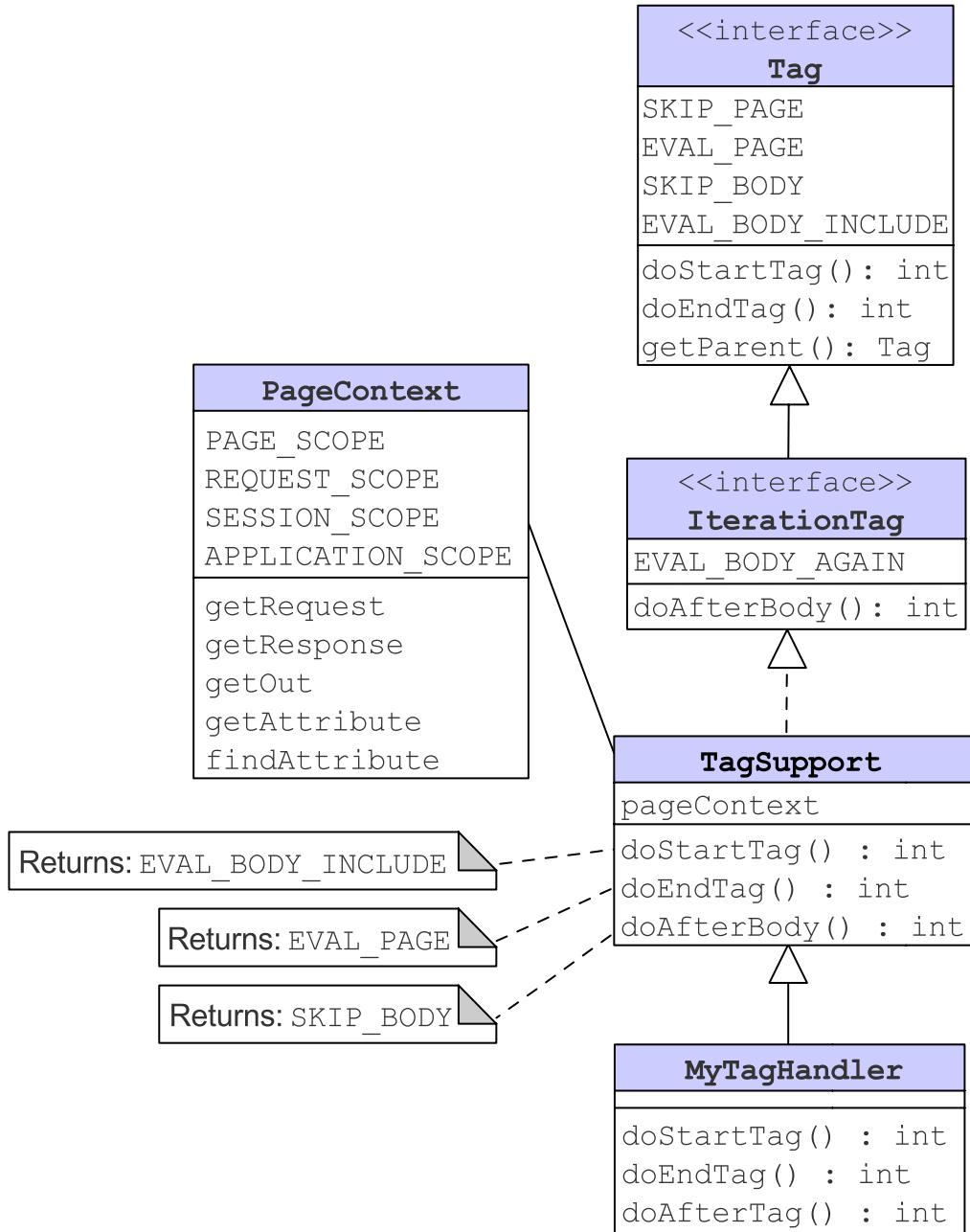
Writing an Iterator Custom Tag

- The `doStartTag` method is used to initialize the iteration, perform the first test, and (if the test passes) get the first object in the iteration.
- The JSP technology code of the body performs the “body processing.”
- A new method, `doAfterBody`, is used to perform subsequent iteration tests.



Iteration Tag API

The `IteratorTag` interface adds the `doAfterBody` method:

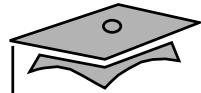




Iteration Tag Life Cycle

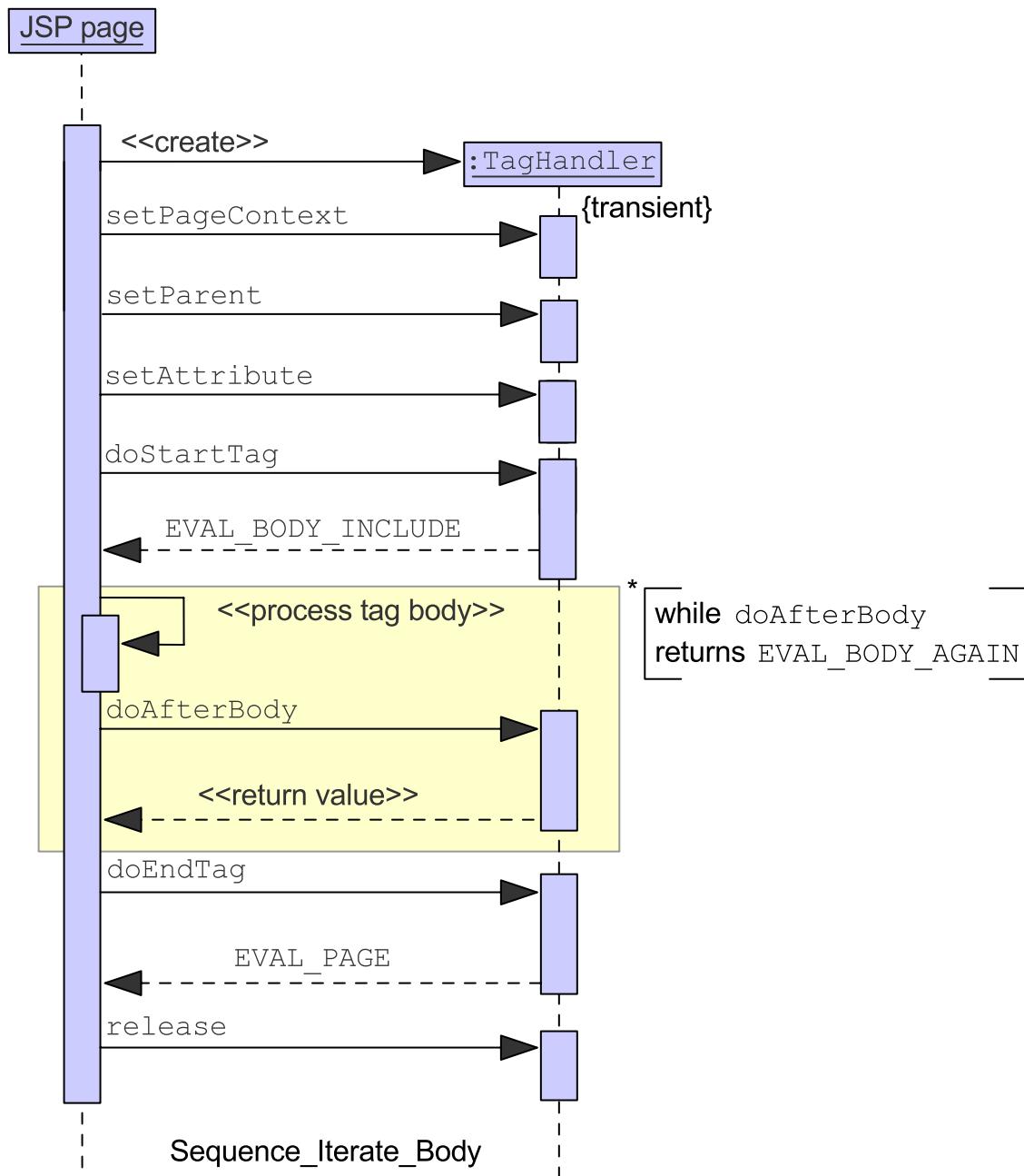
Pseudo-code for tag handler execution in the JSP servlet code:

```
1  TagHandler tagObj = new TagHandler();
2  tagObj.setPageContext(pageContext);
3  tagObj.setAttr("value");
4  try {
5      int startTagResult = tagObj.doStartTag();
6      if ( startTagResult != SKIP_BODY ) {
7          do {
8              out.write("body"); // process the BODY of the tag
9          } while ( tagObj.doAfterBody() == EVAL_BODY_AGAIN );
10     }
11     if ( tagObj.doEndTag() == SKIP_PAGE ) {
12         return; // do not continue processing the JSP page
13     }
14 } finally {
15     tagObj.release();
16 }
```



Iteration Tag Life Cycle

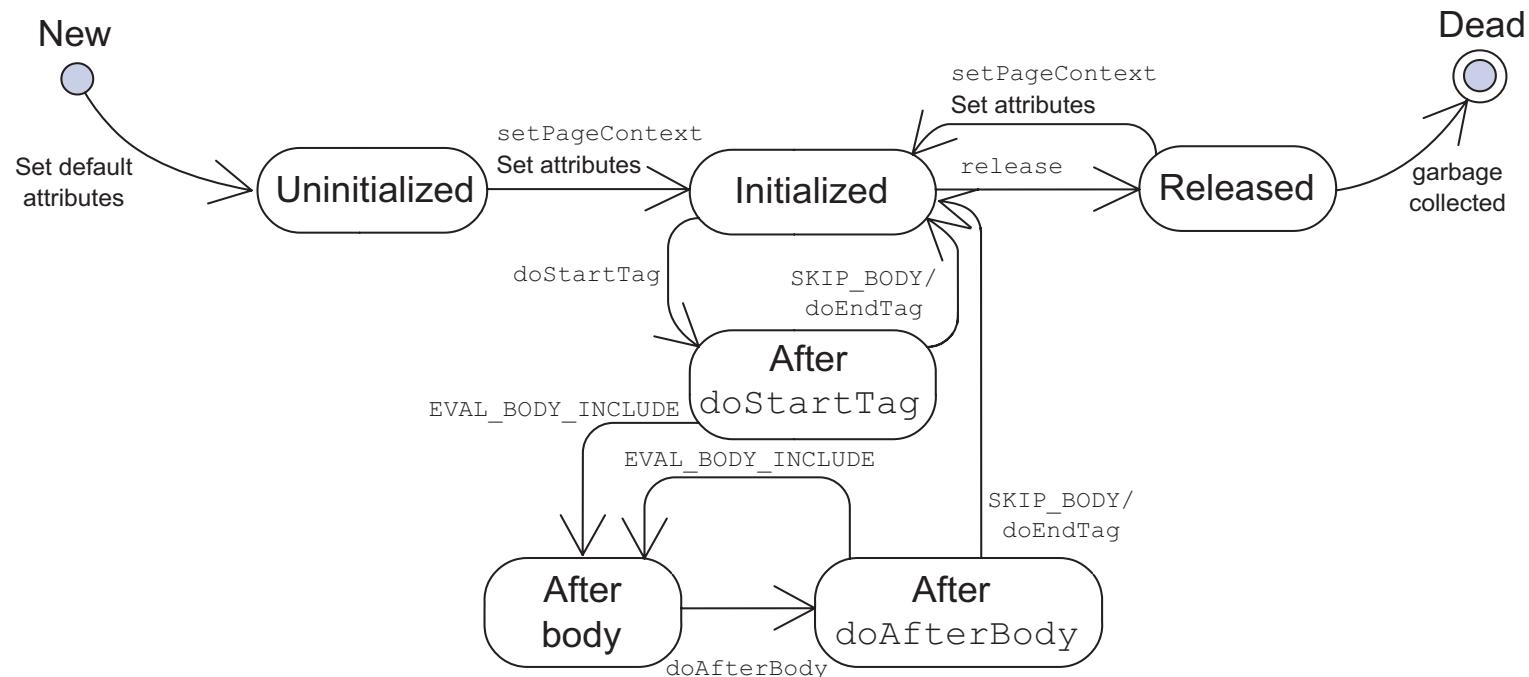
Sequence diagram of an iteration tag life cycle:





Iteration Tag Life Cycle

State diagram of an iteration tag handler object:





Example: The iterateOverErrors Tag

The iterateOverErrors tag iterates over all of the exception objects in the Status object. This tag *must* be used within the checkStatus tag.

- **Body content:** This tag contains JSP technology code.
- **Example:**

```
<soccer:checkStatus>
    <soccer:iterateOverErrors>
        <%-- JSP code showing a single error message --%>
    </soccer:iterateOverErrors>
</soccer:checkStatus>
```



Example: The iterateOverErrors Tag

Here is how the `iterateOverErrors` tag is used in the League Registration Form:

```
16 <%-- BEGIN: error status presentation --%>
17 <soccer:checkStatus>
18 <FONT COLOR='red'>
19 There were problems processing your request:
20 <UL>
21   <soccer:iteratorOverErrors>
22     <LI><soccer:getErrorMessage/>
23   </soccer:iteratorOverErrors>
24 </UL>
25 </FONT>
26 </soccer:checkStatus>
27 <%-- END: error status presentation --%>
```



The iterateOverErrors Tag Handler

The tag handler class:

```
11  /**
12   * This class handles the "iterate over all status exceptions" tag.
13   * This tag handler assumes that it is the child of the "check status"
14   * tag handler. This assumption allows this tag handler to assume
15   * that there is at least one exception in the set of errors in the
16   * Status object. On each iteration, the "current exception object"
17   * is stored within the PAGE scope for use by the "get current error
18   * message" tag.
19 */
20 public class IterateOverMessagesHandler extends TagSupport {
21
22     private Iterator errors;
23 }
```

The errors instance variable holds the Iterator object for the set of exceptions in the Status object.



The iterateOverErrors Tag Handler

The doStartTag method:

```
24  public int doStartTag() {
25      Status status
26      = (Status) pageContext.getAttribute("status",
27          PageContext.REQUEST_SCOPE);
28
29      // Initialize the iterator
30      errors = status.getExceptions();
31
32      // Store the first error in the PAGE scope
33      pageContext.setAttribute("iteratorOverErrors.currentError",
34          errors.next(),
35          PageContext.PAGE_SCOPE);
36
37      return EVAL_BODY_INCLUDE;
38  }
```



The iterateOverErrors Tag Handler

The doAfterBody method:

```
40  public int doAfterBody() {
41
42      // Is there another element to iterate over?
43      if ( errors.hasNext() ) {
44
45          // If yes, then store the next error in the PAGE scope,
46          pageContext.setAttribute("iteratorOverErrors.currentError",
47                                  errors.next(),
48                                  PageContext.PAGE_SCOPE);
49
50          // and tell the JSP interpreter to evaluate the body again
51          return EVAL_BODY_AGAIN;
52
53      } else {
54          // If no, tell the JSP run-time that we are done iterating
55          return SKIP_BODY;
56      }
57  }
```



Using the Page Scope to Communicate

The `getErrorMessage` tag needs to get access to the “current error” in the iteration:

```
20 <UL>
21   <soccer:iteratorOverErrors>
22     <LI><soccer:getErrorMessage/>
23   </soccer:iteratorOverErrors>
24 </UL>
```

The `GetCurrentErrorMsgHandler` tag handler:

```
21 public class GetCurrentErrorMsgHandler extends TagSupport {
22
23   public int doStartTag() throws JspException {
24     JspWriter out = pageContext.getOut();
25
26     // Retrieve the current exception object from the PAGE scope.
27     Exception exc
28       = (Exception)
29         pageContext.getAttribute("iteratorOverErrors.currentError",
30                               PageContext.PAGE_SCOPE);
31
32     try {
33       // Print out the message of the exception object.
34       out.print(exc.getMessage());
```



Summary

- You can create a conditional tag by making the decision to return either SKIP_BODY or EVAL_BODY_INCLUDE from the doStartTag method.
- An iteration tag requires coordination between the doStartTag and doAfterBody methods:
 - The doStartTag method initializes the iteration, retrieves the first element (if any), and determines whether to process the body if there is an element.
 - The doAfterBody method determines whether to process the body again (returning EVAL_BODY AGAIN) or to skip the body if the iteration is done.
- Tag handlers can communicate using the page scope.



Module 20

Integrating Web Applications With Enterprise JavaBeans Components

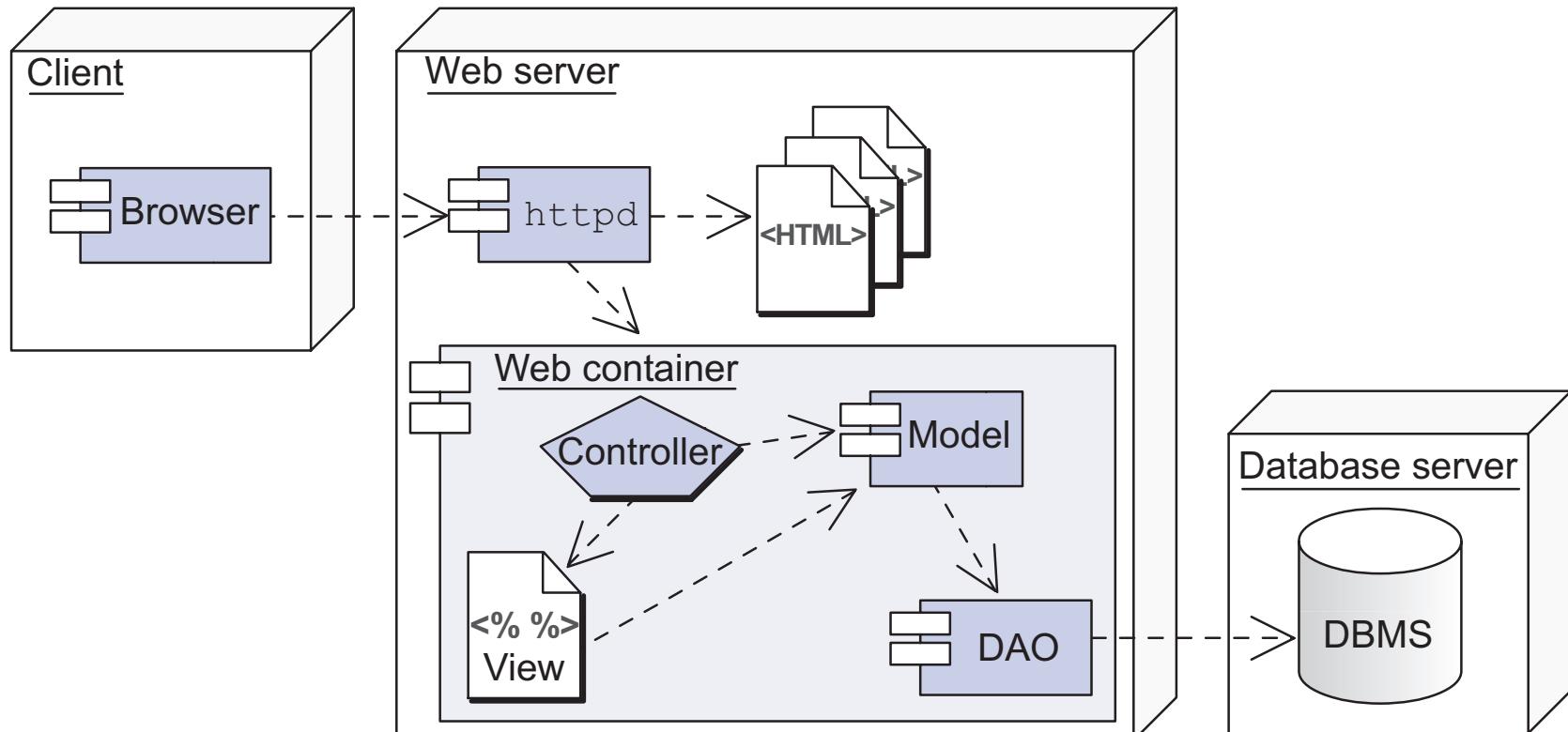


Objectives

- Understand the Java 2 Platform, Enterprise Edition, (J2EE) at a high-level
- Develop a Web application that integrates with an Enterprise JavaBeans (EJB) component using the Business Delegate pattern

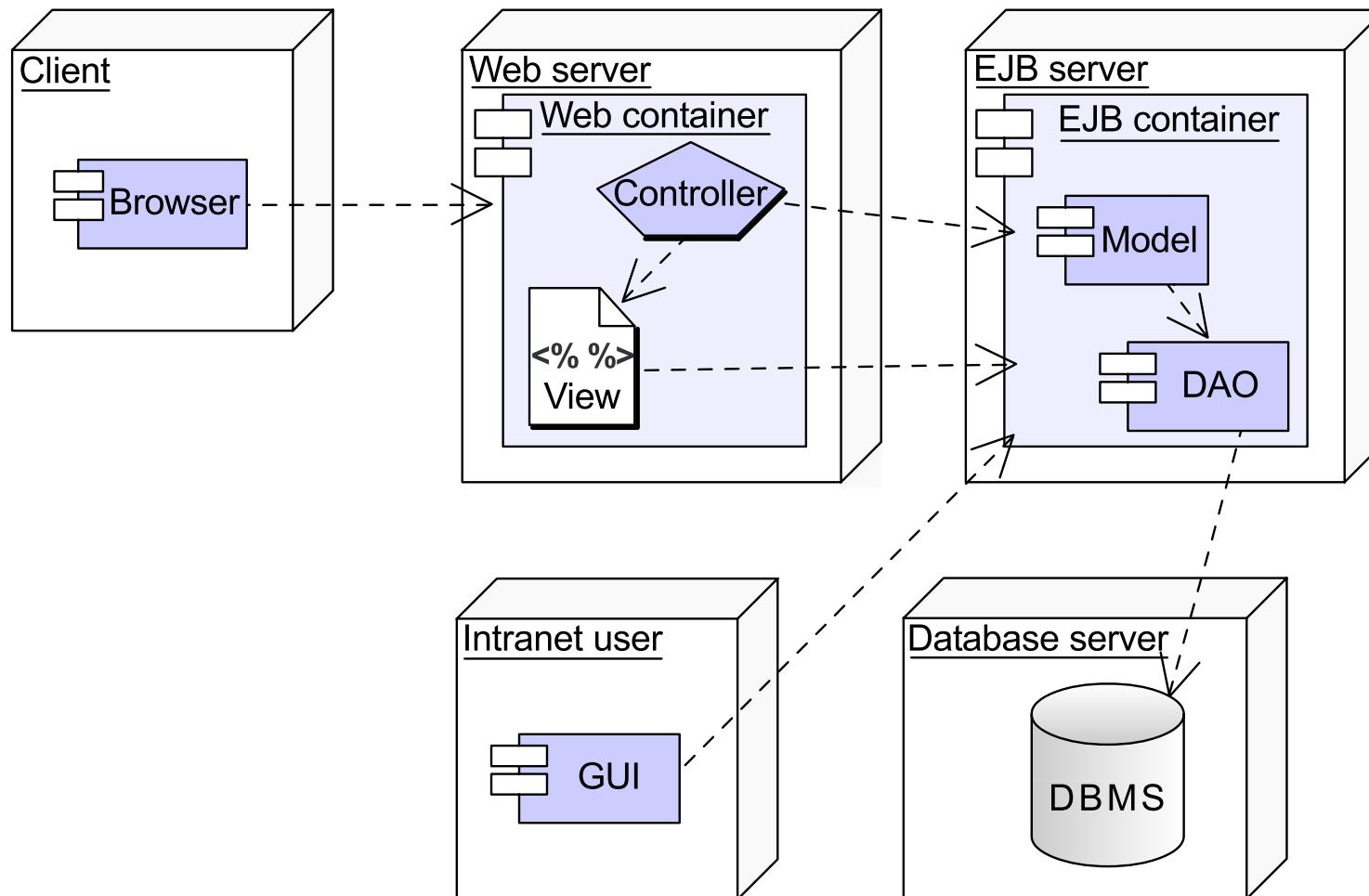


3-Tier Web Architecture





Java 2 Platform, Enterprise Edition





Enterprise JavaBeans Technology

- Enterprise JavaBeans are components that model business logic.
- Session beans are used to encapsulate business services.
- Entity beans are used to encapsulate business domain objects.
- Note: Enterprise JavaBeans components *are not* the same as JavaBeans components.



Integrating the Web Tier With the EJB Tier

The three steps for integration are:

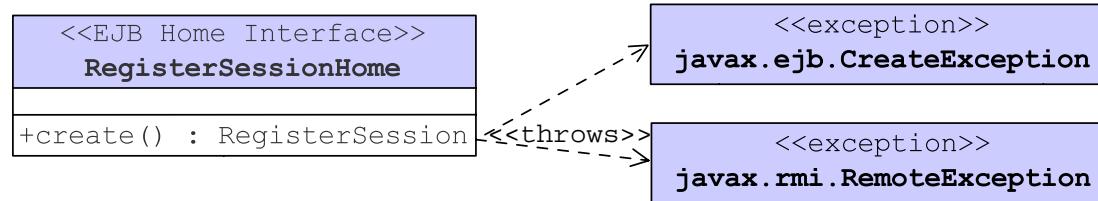
1. The Business Component Developers publish the EJB component interfaces to the application Model.
2. At deployment, names are given to the Enterprise beans and the Java Naming and Directory Interface (JNDI) host is identified.
3. The Web Component Developers create servlets that access the Enterprise beans by using a delegate to hide the complexity of dealing with the EJB tier.



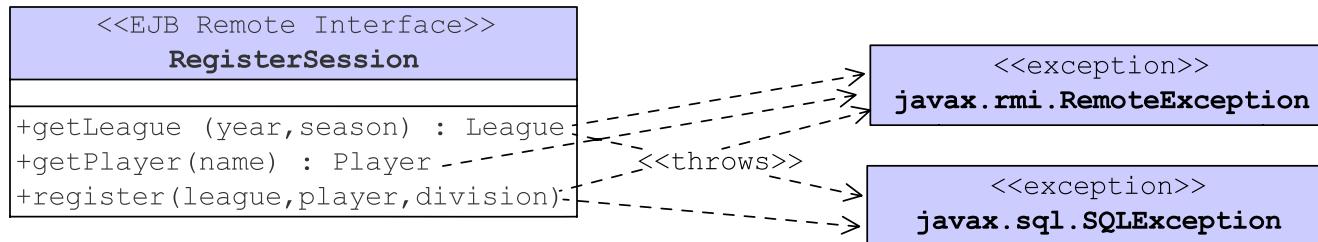
Enterprise JavaBeans Interfaces

An EJB component consists of two interfaces:

- The Home interface allows you to create an EJB bean:



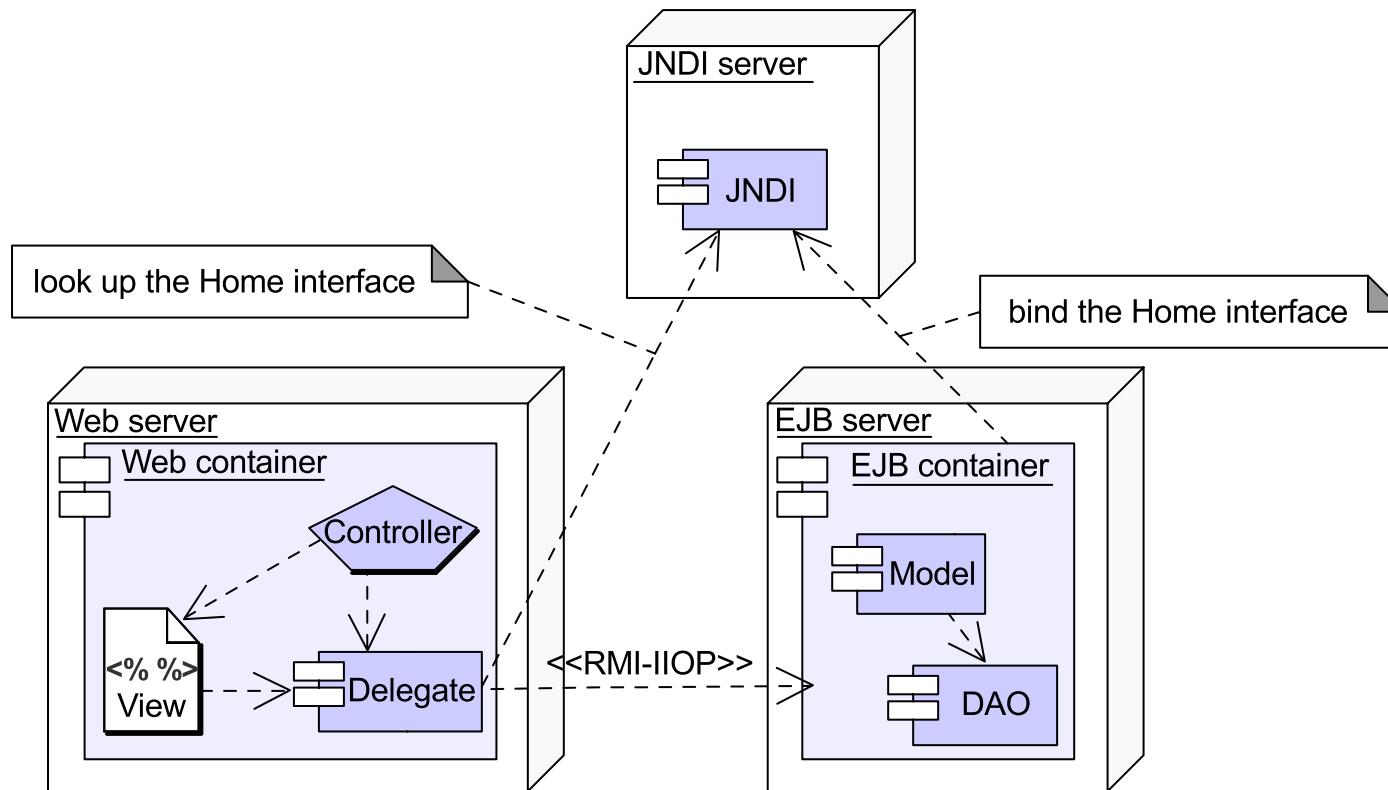
- The Remote interface allows you to run the business methods on the EJB bean:





Java Naming and Directory Interface

JNDI can be used to access remote objects.





Creating the Business Delegate

You create a delegate to the EJB tier. The servlet interacts with the delegate like it did with the business service objects.

The delegate must:

- Use JNDI to look up the Home interface
- Use the Home interface to create the Enterprise bean object
- Call business methods on the Enterprise bean object
- Remove the Enterprise bean when the business method has completed its work



Creating the Business Delegate

Declare the package, imports, and class:

```
1  package sl314.web;
2
3  // Domain imports
4  import sl314.ejb.League;
5  import sl314.ejb.Player;
6  import sl314.ejb.RegisterSession;
7  import sl314.ejb.RegisterSessionHome;
8  // Remote service imports
9  import javax.naming.InitialContext;
10 import javax.naming.Context;
11 import javax.rmi.PortableRemoteObject;
12 import java.rmi.RemoteException;
13 import javax.ejb.CreateException;
14 import java.sql.SQLException;
15
16 /**
17  * This object performs a variety of league registration services.
18  * It acts a Delegate to the RegisterSession Enterprise bean.
19  */
20 public class RegisterDelegate {
21
```



Creating the Business Delegate

Declare an instance variable to hold the home object and perform the JNDI lookup in the delegate constructor:

```
21
22     /**
23      * The EJB remote object for the RegisterSession service.
24      */
25     private RegisterSessionHome registerSvcHome;
26
27     /**
28      * This constructor creates a Registration Delegate object.
29      */
30     public RegisterDelegate() {
31         try {
32             Context c = new InitialContext();
33             Object result = c.lookup("ejb/registerService");
34             registerSvcHome
35                 = (RegisterSessionHome)
36                     PortableRemoteObject.narrow(result, RegisterSessionHome.class);
37
38         } catch (Exception e) {
39             e.printStackTrace(System.err);
40         }
41     }
42 }
```



Creating the Business Delegate

In the business methods, use the home object to create a remote instance of the Enterprise bean, delegate the business method call to that remote bean, and remove it:

```
46  public League getLeague(String year, String season)
47      throws CreateException, RemoteException, SQLException {
48  League league = null;
49  RegisterSession registerSvc = null;
50
51  // Delegate to the EJB session bean
52  try {
53      registerSvc = registerSvcHome.create();
54      league = registerSvc.getLeague(year, season);
55
56  } finally {
57      try {
58          if ( registerSvc != null ) registerSvc.remove();
59      } catch (Exception e) {
60          e.printStackTrace(System.err);
61      }
62  }
63
64  return league;
65 }
```



Using a Delegate in a Servlet

- Create the delegate just as you did for the service object
- Execute business methods on the delegate just as you did on the service object
- Handle any exceptions dealing with remote access to EJB components in the try-catch block



Using a Delegate in a Servlet

In the try-catch block, create and use the delegate:

```
89     // Now delegate the real work to the RegisterDelegate object
90     try {
91         // Create a "registration delegate" object
92         registerDlg = new RegisterDelegate();
93
94         // Retrieve the league object
95         league = registerDlg.getLeague(year, season);
96         // and verify that it exists
97         if ( league == null ) {
98             // If not, then forward to the Error page View
99             status.addException(
100                 new Exception("The league you selected does not yet exist;" +
101                     + " please select another."));
102             responsePage = request.getRequestDispatcher("/form.jsp");
103             responsePage.forward(request, response);
104             return;
105         }
```



Using a Delegate in a Servlet

In the try-catch block, catch any exception dealing with remote access to EJB components and store it in the Status object:

```
122
123     // Handle any remote exceptions
124     } catch (CreateException ce) {
125         status.addException(
126             new Exception("Could not create the EJB Session object.<BR>" +
127                         + ce.getMessage()));
128
129     // Handle any remote exceptions
130     } catch (RemoteException re) {
131         status.addException(
132             new Exception("An EJB error occurred.<BR>" +
133                         + re.getMessage()));
134
135     // Handle any SQL exceptions
136     } catch (SQLException se) {
137         status.addException(
138             new Exception("A remote SQL exception occurred.<BR>" +
139                         + se.getMessage()));
```



Summary

- On the EJB tier, the Business Component Developer publishes the Home and Remote interfaces.
- On the Web tier, the Web Component Developer creates a Business Delegate to the EJB tier and modifies the servlets to interact with that delegate.
- The Business Delegate must:
 - Use JNDI to look up the Home interface
 - Use the Home interface to create the Enterprise bean object
 - Call business methods on the Enterprise bean object
 - Remove the Enterprise bean when the business method has completed its work

Course Contents

| | |
|---|------------------|
| About This Course | Preface-i |
| Course Goal | Preface-ii |
| Learning Objectives | Preface-iii |
| Module-by-Module Overview | Preface-iv |
| Topics Not Covered | Preface-v |
| How Prepared Are You? | Preface-vi |
| How To Learn From This Course | Preface-vii |
| Introductions | Preface-viii |
| Icons | Preface-ix |
| Typographical Conventions | Preface-x |
| Introduction to Web Application Technologies | 1-1 |
| Objectives | 1-2 |
| The Internet Is a Network of Networks | 1-3 |
| Networking Protocol Stack | 1-4 |
| Client-Server Architecture | 1-5 |
| Hypertext Transfer Protocol | 1-6 |
| Web Browsers and Web Servers | 1-7 |
| HTTP Client-Server Architecture | 1-8 |
| The Structure of a Web Site | 1-9 |
| Web Applications | 1-10 |
| CGI Programs on the Web Server | 1-11 |
| Execution of CGI Programs | 1-12 |
| Advantages and Disadvantages of CGI Programs | 1-14 |
| Java Servlets | 1-15 |
| Servlets on the Web Server | 1-16 |



| | |
|--|------|
| Execution of Java Servlets | 1-17 |
| Advantages and Disadvantages of Java Servlets | 1-19 |
| Template Pages | 1-20 |
| Other Template Page Technologies | 1-21 |
| JavaServer Pages Technology | 1-22 |
| Advantages and Disadvantages of JavaServer Pages | 1-23 |
| The Model 2 Architecture | 1-24 |
| The J2EE Platform | 1-25 |
| An Example of J2EE Architecture | 1-26 |
| Job Roles | 1-27 |
| Web Application Migration | 1-28 |
| Summary | 1-29 |

Developing a Simple Servlet 2-1

| | |
|--|------|
| Objectives | 2-2 |
| The NetServer Architecture | 2-3 |
| The Generic Servlets API | 2-4 |
| The Generic HelloServlet Class | 2-5 |
| Hypertext Transfer Protocol | 2-6 |
| HTTP GET Method | 2-7 |
| HTTP Request | 2-8 |
| The HttpServletRequest API | 2-9 |
| HTTP Request Examples | 2-10 |
| HTTP Response | 2-11 |
| The HttpServletResponse API | 2-12 |
| HTTP Response Examples | 2-13 |
| Web Container Architecture | 2-14 |
| The Web Container | 2-15 |
| Sequence Diagram of HTTP GET Request | 2-16 |
| Request and Response Process | 2-17 |



| | |
|--|------|
| The HTTP Servlet API | 2-22 |
| The HTTP HelloServlet Class | 2-23 |
| Deploying a Servlet | 2-24 |
| Installing, Configuring, and Running the Web Container | 2-25 |
| Deploying the Servlet to the Web Container | 2-26 |
| Activating the Servlet in a Web Browser | 2-27 |
| Summary | 2-28 |

Developing a Simple Servlet That Uses HTML Forms 3-1

| | |
|-------------------------------------|------|
| Objectives | 3-2 |
| HTML Forms | 3-3 |
| The FORM Tag | 3-4 |
| HTML Form Components | 3-6 |
| Input Tags | 3-7 |
| The Select Tag | 3-11 |
| The Textarea Tag | 3-13 |
| Form Data in the HTTP Request | 3-14 |
| HTTP GET Method Request | 3-15 |
| HTTP POST Method Request | 3-16 |
| To GET or to POST? | 3-17 |
| The Servlet API | 3-18 |
| The FormBasedHello Servlet | 3-19 |
| Summary | 3-22 |

Developing a Web Application Using a Deployment Descriptor 4-1

| | |
|--|-----|
| Objectives | 4-2 |
| Problems With Simple Servlets | 4-3 |
| Problems With Deploying in One Place | 4-4 |
| Multiple Web Applications | 4-5 |
| Using Multiple Web Applications | 4-6 |



| | |
|--|------|
| Web Application Context Name | 4-7 |
| Problems With Servlet Naming | 4-8 |
| Solution to Servlet Naming Problems | 4-10 |
| Problems Using Common Services | 4-11 |
| Developing a Web Application Using a Deployment Descriptor | 4-12 |
| The Deployment Descriptor | 4-13 |
| A Development Environment | 4-14 |
| The Deployment Environment | 4-15 |
| The Web Archive (WAR) File Format | 4-17 |
| Summary | 4-18 |

Configuring Servlets 5-1

| | |
|-------------------------------------|------|
| Objectives | 5-2 |
| Servlet Life Cycle Overview | 5-3 |
| The init Life Cycle Method | 5-4 |
| The service Life Cycle Method | 5-5 |
| The destroy Life Cycle Method | 5-6 |
| The ServletConfig API | 5-7 |
| Initialization Parameters | 5-9 |
| Summary | 5-12 |

Sharing Resources Using the Servlet Context 6-1

| | |
|---|-----|
| Objectives | 6-2 |
| The Web Application | 6-3 |
| Duke's Store Web Application | 6-4 |
| The ServletContext API | 6-5 |
| Context Initialization Parameters | 6-6 |
| Access to File Resources | 6-7 |
| Writing to the Web Application Log File | 6-8 |
| Accessing Shared Runtime Attributes | 6-9 |



| | |
|--|------|
| The Web Application Life Cycle | 6-10 |
| The Web Application Lifecycle Listener API | 6-11 |
| Duke's Store Example | 6-12 |
| Configuring Servlet Context Listeners | 6-13 |
| Summary | 6-14 |

Developing Web Applications Using the MVC Pattern 7-1

| | |
|---|------|
| Objectives | 7-2 |
| Activities of a Web Application | 7-3 |
| The Soccer League Example | 7-4 |
| Activity Diagram of the Soccer League Example | 7-7 |
| Model-View-Controller for a Web Application | 7-8 |
| Sequence Diagram of MVC in the Web Tier | 7-9 |
| Soccer League Application: The Domain Model | 7-10 |
| Soccer League Application: The Services Model | 7-11 |
| Soccer League Application: The Big Picture | 7-12 |
| Soccer League Application: The Controller | 7-13 |
| Soccer League Application: The Views | 7-15 |
| The Request Scope | 7-16 |
| Summary | 7-17 |

Developing Web Applications Using Session Management 8-1

| | |
|--|-----|
| Objectives | 8-2 |
| HTTP and Session Management | 8-3 |
| Sessions in a Web Container | 8-4 |
| Web Application Design Using Session Management | 8-5 |
| Example: Registration Use Case | 8-6 |
| Example: Multiple Views for Registration | 8-7 |
| Example: Enter League Form | 8-8 |
| Web Application Development Using Session Management | 8-9 |



| | |
|--|------|
| The Session API | 8-10 |
| Retrieving the Session Object | 8-11 |
| Storing Session Attributes | 8-12 |
| Accessing Session Attributes | 8-13 |
| Destroying the Session | 8-15 |
| Session Management Using Cookies | 8-17 |
| The Cookie API | 8-18 |
| Using Cookies | 8-19 |
| Using Cookies for Session Management | 8-20 |
| Session Management Using URL-Rewriting | 8-23 |
| Implications of Using URL-Rewriting | 8-24 |
| Guidelines for Working With Sessions | 8-25 |
| Summary | 8-26 |

| | |
|---|------------|
| Handling Errors in Web Applications | 9-1 |
| Objectives | 9-2 |
| HTTP Error Codes | 9-3 |
| Generic HTTP Error Page | 9-4 |
| Servlet Exceptions | 9-5 |
| Generic Servlet Error Page | 9-6 |
| Using Custom Error Pages | 9-7 |
| Creating Error Pages | 9-8 |
| Declaring HTTP Error Pages | 9-9 |
| Example HTTP Error Page | 9-10 |
| Declaring Servlet Exception Error Pages | 9-11 |
| Throwing Servlet Exceptions | 9-12 |
| Example Servlet Error Page | 9-13 |
| Developing an Error Handling Servlet | 9-14 |
| Programmatic Exception Handling | 9-16 |
| Trade-offs for Declarative Exception Handling | 9-19 |



| | |
|--|------|
| Trade-offs for Programmatic Exception Handling | 9-20 |
| Logging Exceptions | 9-21 |
| Summary | 9-22 |

Configuring Web Application Security 10-1

| | |
|-----------------------------------|-------|
| Objectives | 10-2 |
| Web Security Issues | 10-3 |
| Authentication | 10-4 |
| Authorization | 10-5 |
| Maintaining Data Integrity | 10-6 |
| Access Tracking | 10-7 |
| Dealing With Malicious Code | 10-8 |
| Dealing With Web Attacks | 10-9 |
| Declarative Authorization | 10-10 |
| Web Resource Collection | 10-11 |
| Declaring Security Roles | 10-13 |
| Security Realms | 10-14 |
| Declarative Authentication | 10-15 |
| BASIC Authentication | 10-16 |
| Form-based Authentication | 10-17 |
| Summary | 10-20 |

Understanding Web Application Concurrency Issues 11-1

| | |
|---|-------|
| Objectives | 11-2 |
| The Need for Servlet Concurrency Management | 11-3 |
| Attributes and Scope | 11-6 |
| Local Variables | 11-7 |
| Instance Variables | 11-8 |
| Class Variables | 11-9 |
| Request Scope | 11-10 |



| | |
|---|-------|
| Session Scope | 11-11 |
| Application Scope | 11-12 |
| The SingleThreadModel Interface | 11-13 |
| How the Web Container Might Implement the Single Threaded Model | 11-14 |
| STM and Concurrency Management | 11-15 |
| Recommended Approaches to Concurrency Management | 11-17 |
| Summary | 11-19 |

Integrating Web Applications With Databases 12-1

| | |
|---|-------|
| Objectives | 12-2 |
| Database Overview | 12-3 |
| The JDBC API | 12-4 |
| Designing a Web Application That Integrates With a Database | 12-5 |
| Domain Objects | 12-6 |
| Database Tables | 12-7 |
| Data Access Object Pattern | 12-9 |
| Advantages of the DAO Pattern | 12-11 |
| Developing a Web Application That Uses a Connection Pool | 12-12 |
| Connection Pool | 12-13 |
| Storing the Connection Pool in a Global Name Space | 12-15 |
| Accessing the Connection Pool | 12-16 |
| Initializing the Connection Pool | 12-17 |
| Developing a Web Application That Uses a Data Source | 12-18 |
| Summary | 12-19 |

Developing JSP™ Pages 13-1

| | |
|--|-------|
| Objectives | 13-2 |
| JavaServer Page Technology | 13-3 |
| How a JSP Page Is Processed | 13-6 |
| Developing and Deploying JSP Pages | 13-10 |



| | |
|--------------------------------|-------|
| Objectives | 13-11 |
| JSP Scripting Elements | 13-12 |
| Comments | 13-13 |
| Directive Tag | 13-14 |
| Declaration Tag | 13-15 |
| Scriptlet Tag | 13-16 |
| Expression Tag | 13-17 |
| Implicit Variables | 13-18 |
| Objectives | 13-19 |
| The page Directive | 13-20 |
| Objectives | 13-23 |
| Declaring an Error Page | 13-25 |
| Developing an Error Page | 13-26 |
| Objectives | 13-27 |
| Behind the Scenes | 13-28 |
| Debugging a JSP Page | 13-29 |
| Summary | 13-30 |

| | |
|---|-------------|
| Developing Web Applications Using the Model 1 Architecture | 14-1 |
| Objectives | 14-2 |
| Designing With Model 1 Architecture | 14-3 |
| Guest Book Form | 14-4 |
| Guest Book Components | 14-5 |
| Guest Book Page Flow | 14-6 |
| What Is a JavaBeans Component? | 14-7 |
| The GuestBookService JavaBeans Component | 14-8 |
| The Guest Book HTML Form | 14-9 |
| JSP Standard Actions | 14-10 |
| Creating a JavaBeans Component in a JSP Page | 14-11 |
| Using the <code>jsp:setProperty</code> Action to Initialize a JavaBeans Component | 14-12 |



| | |
|---|-------|
| Control Logic in the GuestBook JSP Page | 14-13 |
| Accessing a JavaBeans Component in a JSP Page | 14-14 |
| Beans and Scope | 14-15 |
| Review of the <code>jsp:useBean</code> Action | 14-16 |
| Summary | 14-17 |

Developing Web Applications Using the Model 2 Architecture 15-1

| | |
|--|-------|
| Objectives | 15-2 |
| Designing With Model 2 Architecture | 15-3 |
| The Soccer League Example Using Model 2 Architecture | 15-4 |
| Sequence Diagram of Model 2 Architecture | 15-5 |
| Developing With Model 2 Architecture | 15-6 |
| Controller Details | 15-7 |
| Request Dispatchers | 15-8 |
| View Details | 15-9 |
| Summary | 15-10 |

Building Reusable Web Presentation Components 16-1

| | |
|--|-------|
| Objectives | 16-2 |
| Complex Page Layouts | 16-3 |
| What Does a Fragment Look Like? | 16-5 |
| Organizing Your Presentation Fragments | 16-6 |
| Including JSP Page Fragments | 16-7 |
| Using the <code>include</code> Directive | 16-8 |
| Using the <code>jsp:include</code> Standard Action | 16-9 |
| Using the <code>jsp:param</code> Standard Action | 16-10 |
| Summary | 16-12 |



| | |
|---|-----------------|
| Developing JSP Pages Using Custom Tags | 17-1 |
| Objectives | 17-2 |
| Job Roles Revisited | 17-3 |
| Contrasting Custom Tags and Scriptlet Code | 17-4 |
| Developing JSP Pages Using Custom Tags | 17-6 |
| What Is a Custom Tag Library? | 17-7 |
| Custom Tag Syntax Rules | 17-8 |
| Example Tag Library: Soccer League | 17-9 |
| Developing JSP Pages Using a Custom Tag Library | 17-14 |
| Using an Empty Custom Tag | 17-15 |
| Using a Conditional Custom Tag | 17-16 |
| Using an Iterative Custom Tag | 17-17 |
| Summary | 17-18 |
| Developing a Simple Custom Tag | 18-1 |
| Objectives | 18-2 |
| Fundamental Tag Handler API | 18-3 |
| Tag Handler Life Cycle | 18-4 |
| Tag Library Relationships | 18-7 |
| Developing a Tag Handler Class | 18-8 |
| The getReqParam Tag | 18-9 |
| The getReqParam Tag Handler Class | 18-10 |
| Configuring the Tag Library Descriptor | 18-14 |
| Tag Declaration Element | 18-15 |
| Custom Tag Body Content | 18-16 |
| Custom Tag Attributes | 18-17 |
| Custom Tag That Includes the Body | 18-18 |
| The heading Tag Handler Class | 18-20 |
| The heading Tag Descriptor | 18-22 |
| Summary | 18-23 |



| | |
|--|-----------------|
| Developing Advanced Custom Tags | 19-1 |
| Objectives | 19-2 |
| Writing a Conditional Custom Tag | 19-3 |
| Example: The checkStatus Tag | 19-4 |
| The checkStatus Tag Handler | 19-6 |
| The checkStatus Tag Life Cycle | 19-7 |
| Writing an Iterator Custom Tag | 19-8 |
| Iteration Tag API | 19-10 |
| Iteration Tag Life Cycle | 19-11 |
| Example: The iterateOverErrors Tag | 19-14 |
| The iterateOverErrors Tag Handler | 19-16 |
| Using the Page Scope to Communicate | 19-19 |
| Summary | 19-20 |
| Integrating Web Applications With Enterprise JavaBeans Components | 20-1 |
| Objectives | 20-2 |
| Java 2 Platform, Enterprise Edition | 20-4 |
| Enterprise JavaBeans Technology | 20-5 |
| Integrating the Web Tier With the EJB Tier | 20-6 |
| Enterprise JavaBeans Interfaces | 20-7 |
| Java Naming and Directory Interface | 20-8 |
| Creating the Business Delegate | 20-9 |
| Using a Delegate in a Servlet | 20-13 |
| Summary | 20-16 |