



Sun Services

---

# Web Component Development With Servlet and JSP™ Technologies

SL-314



Copyright 2005 Sun Microsystems, Inc. 4150 Network Circle, Santa Clara, California 95054, 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, Java, EJB, Enterprise JavaBeans, JavaBeans, Java Naming and Directory Interface, JavaServer Pages, JavaServer, 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.

U.S. Government approval might be required when exporting the product.

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 2005 Sun Microsystems Inc. 4150 Network Circle, Santa Clara, California 95054, 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, Java, EJB, Enterprise JavaBeans, JavaBeans, Java Naming and Directory Interface, JavaServer Pages, JavaServer, 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.

L'accord du gouvernement américain est requis avant l'exportation du produit.

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.

# Course Contents

---

<b>About This Course .....</b>	<b>Preface-xvii</b>
Course Goals .....	Preface-xviii
Course Map .....	Preface-xix
Topics Not Covered .....	Preface-xx
How Prepared Are You? .....	Preface-xxi
How To Learn From This Course .....	Preface-xxii
Introductions .....	Preface-xxiii
Icons .....	Preface-xxiv
Typographical Conventions .....	Preface-xxv
Additional Conventions .....	Preface-xxvii
<b>Introduction to Web Application Technologies .....</b>	<b>1-1</b>
Objectives .....	1-2
Relevance .....	1-3
The Internet Is a Network of Networks .....	1-4
Networking Protocol Stack .....	1-5
Client-Server Architecture .....	1-6
World Wide Web Introduction .....	1-8
Web Browsers and Web Servers .....	1-9
HTTP Client-Server Architecture .....	1-10
Web Site Structure .....	1-11
Web Applications and Web Sites .....	1-13
CGI Programs on the Web Server .....	1-14
Execution of CGI Programs .....	1-15
Advantages and Disadvantages of CGI Programs .....	1-17
Java Servlets .....	1-19



Servlets on the Web Server .....	1-20
Execution of Java Servlets .....	1-21
Advantages and Disadvantages of Java Servlets .....	1-23
JavaServer Pages Technology .....	1-25
Other Template Page Technologies .....	1-26
JavaServer Pages Technology .....	1-27
Advantages and Disadvantages of JavaServer Pages .....	1-28
Model 2 Architecture .....	1-29
Model 2 Frameworks .....	1-30
J2EE™ Platform .....	1-32
J2EE Architecture Example .....	1-33
Job Roles .....	1-34
Web Application Migration .....	1-35
Summary .....	1-36

<b>Developing a View Component .....</b>	<b>2-1</b>
Objectives .....	2-2
Relevance .....	2-3
Types of View Components .....	2-4
Soccer League Case Study .....	2-5
List Leagues Analysis Model .....	2-6
List Leagues Page Flow .....	2-7
Home Page HTML .....	2-8
List Leagues Page HTML .....	2-10
Hypertext Transfer Protocol .....	2-12
HTTP GET Method .....	2-13
HTTP Request .....	2-14
HTTP Request Headers .....	2-15
HTTP Response .....	2-16
HTTP Response Headers .....	2-17



Web Container Architecture .....	2-19
Request and Response Process .....	2-20
Sequence Diagram of an HTTP GET Request .....	2-24
List Leagues Architecture Model .....	2-26
The HttpServlet API .....	2-27
The ListLeaguesServlet Code .....	2-28
The ListLeaguesServlet Code (Part 2) .....	2-29
The ListLeaguesServlet Code (Part 3) .....	2-30
The ListLeaguesServlet Code (Part 4) .....	2-31
Soccer League Web Application Structure .....	2-33
Configuring a Servlet Definition .....	2-34
Configuring a Servlet Mapping .....	2-35
Complete Deployment Descriptor .....	2-36
Web Application Context Root .....	2-38
Tomcat Deployment .....	2-39
WAR Files for Deployment .....	2-40
Activating the Servlet in a Web Browser .....	2-41
Activating the ListLeagues View .....	2-42
Summary .....	2-43
<b>Developing a Controller Component .....</b>	<b>3-1</b>
Objectives .....	3-2
Relevance .....	3-3
Types of Controller Components .....	3-4
Add a New League Analysis Model .....	3-5
Add League Boundary Components .....	3-6
Add a New League Page Flow .....	3-7
Form Verification .....	3-9
Soccer League Web Structure .....	3-10
Creating an HTML Form .....	3-13

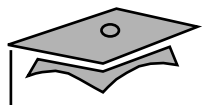


The FORM Tag .....	3-14
Textfield Component .....	3-15
Drop-Down List Component .....	3-16
Submit Button .....	3-17
Complete Add a New League Form .....	3-18
Form Data in the HTTP Request .....	3-20
HTTP GET Method Request .....	3-21
HTTP POST Method Request .....	3-22
HTTP GET and POST Methods .....	3-23
Developing a Controller Servlet .....	3-25
Add League Analysis Model (Stage 1) .....	3-26
Add League Architecture Model (Stage 1) .....	3-27
Servlet API to Retrieve Form Parameters .....	3-28
The AddLeagueServlet Class Declaration .....	3-29
Retrieving Form Parameters and Data Conversion .....	3-30
Performing Form Parameters .....	3-31
Performing the Business Logic .....	3-32
Handling an Exception .....	3-33
Add League Analysis Model (Stage 2) .....	3-35
Add League Architecture Model (Stage 2) .....	3-36
Request Scope .....	3-37
Using a Request Dispatcher .....	3-38
Developing the AddLeagueServlet Code .....	3-39
The SuccessServlet Code .....	3-43
The SuccessServlet Code (Part 2) .....	3-44
The SuccessServlet Code (Part 3) .....	3-45
Summary .....	3-46

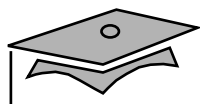


<b>Developing Dynamic Forms .....</b>	<b>4-1</b>
Objectives .....	4-2
Relevance .....	4-3
Servlet Life Cycle Overview .....	4-4
Servlet Class Loading .....	4-5
One Instance Per Servlet Definition .....	4-6
The <code>init</code> Life Cycle Method .....	4-7
The <code>service</code> Life Cycle Method .....	4-8
The <code>destroy</code> Life Cycle Method .....	4-9
Customizing the Add a New League Form .....	4-11
Add League Architecture Model (Step 1) .....	4-12
The <code>AddLeagueFormServlet</code> Code .....	4-13
Configuring Initialization Parameters .....	4-14
The <code>ServletConfig</code> API .....	4-15
The <code>AddLeagueFormServlet</code> Code .....	4-16
Add League Analysis Model (Stage 2) .....	4-18
Error Handling Screen Shots .....	4-19
Add League Architecture Model (Stage 2) .....	4-20
Soccer League Web Application Structure .....	4-21
The <code>AddLeagueServlet</code> Code .....	4-23
The <code>AddLeagueFormServlet</code> Code .....	4-24
Repopulating Web Forms .....	4-28
Repopulating a Text Field .....	4-29
Repopulating a Drop-Down List .....	4-30
Summary .....	4-31
<b>Sharing Application Resources Using the Servlet Context .....</b>	<b>5-1</b>
Objectives .....	5-2
Relevance .....	5-3
Soccer League Demonstration .....	5-4





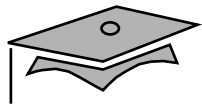
Servlet Context .....	5-5
The ServletContext API .....	5-6
Soccer League Architecture Model .....	5-7
Modified AddLeagueServlet Code .....	5-8
Modified ListLeaguesServlet Code .....	5-9
Modified ListLeaguesServlet Code (Part 2) .....	5-10
League List Initialization Example .....	5-12
Web Application Life Cycle .....	5-13
Soccer League Architecture Model (Revisited) .....	5-14
The ServletContextListener API .....	5-15
The InitializeLeagues Code .....	5-16
The InitializeLeagues Code (Part 2) .....	5-17
The InitializeLeagues Code (Part 3) .....	5-18
Soccer League Deployment Descriptor .....	5-19
Soccer League Physical Hierarchy .....	5-20
Summary .....	5-21
<b>Designing the Business Tier .....</b>	<b>6-1</b>
Objectives .....	6-2
Relevance .....	6-3
Describing the Analysis Model .....	6-4
Registration Use Case Analysis Model .....	6-5
Detailed Analysis Model .....	6-6
Another View: UML Sequence Diagram .....	6-7
Another View: UML Deployment Diagram .....	6-8
Domain Entities .....	6-10
The Player Code .....	6-11
The Player Code (Part 2) .....	6-12
The Player Code (Part 3) .....	6-13
Entity Service .....	6-15



The LeagueService Code .....	6-16
The LeagueService Code (Part 2) .....	6-17
The LeagueService Code (Part 3) .....	6-18
The LeagueService Code (Part 4) .....	6-19
Façade Service .....	6-20
The RegisterService Code .....	6-21
The RegisterService Code (Part 2) .....	6-22
The RegisterService Code (Part 3) .....	6-23
Summary .....	6-24

## **Developing Web Applications Using Struts ..... 7-1**

Objectives .....	7-2
Relevance .....	7-3
Model-View-Controller Pattern .....	7-4
Struts MVC Framework .....	7-5
Front Controller Pattern .....	7-6
Struts MVC Framework .....	7-7
Struts Activity Diagram .....	7-8
Struts Action Class .....	7-10
The AddLeagueAction Code .....	7-11
The AddLeagueAction Code (Part 2) .....	7-12
The AddLeagueAction Code (Part 3) .....	7-13
The AddLeagueAction Code (Part 4) .....	7-14
Configuring the Struts Action Mappings .....	7-16
Configuring the Infrastructure Controller .....	7-17
Front Controller Servlet Mapping .....	7-18
Configuring Action Mappings .....	7-19
Action Mapping Object Representation .....	7-20
Physical Web Hierarchy .....	7-21
Summary .....	7-22



<b>Developing Web Applications Using Session Management .....</b>	<b>8-1</b>
Objectives .....	8-2
Relevance .....	8-3
HTTP and Session Management .....	8-4
Web Container Sessions .....	8-5
Designing Web Applications .....	8-7
Registration Use Case Example .....	8-8
Registration Use Case Analysis Model .....	8-9
Developing Web Applications .....	8-11
Session API .....	8-12
Storing Session Attributes .....	8-13
Accessing Session Attributes .....	8-14
Destroying the Session .....	8-16
Using Cookies for Session Management .....	8-19
Cookie API .....	8-20
Using Cookies Example .....	8-21
Performing Session Management Using Cookies .....	8-22
Using URL-Rewriting for Session Management .....	8-26
URL-Rewriting Implications .....	8-28
Summary .....	8-29
<b>Using Filters in Web Applications .....</b>	<b>9-1</b>
Objectives .....	9-2
Relevance .....	9-3
Web Container Request Cycle .....	9-4
Web Container Request Processing .....	9-5
Applying Filters to an Incoming Request .....	9-6
Filters Applied to a Dispatch .....	9-10
Filter API .....	9-12
The <code>PerformanceFilter</code> Class .....	9-14



The init Method .....	9-15
The doFilter Method .....	9-16
The destroy Method .....	9-18
Configuring the Filter .....	9-20
Summary .....	9-25
<b>Integrating Web Applications With Databases .....</b>	<b>10-1</b>
Objectives .....	10-2
Relevance .....	10-3
Database Overview .....	10-4
JDBC™ API .....	10-5
Designing a Web Application .....	10-7
Domain Objects .....	10-8
Database Tables .....	10-9
Data Access Object Pattern .....	10-11
DAO Pattern Advantages .....	10-13
Traditional Approaches to Database Connections .....	10-15
Using a DataSource and JNDI API .....	10-17
Application DataSource Use .....	10-18
Configuring a DataSource and JNDI .....	10-21
Tomcat DataSource Configuration .....	10-22
Summary .....	10-24
<b>Developing JSP™ Pages .....</b>	<b>11-1</b>
Objectives .....	11-2
Relevance .....	11-3
JavaServer Pages Technology .....	11-4
Hello World Servlet .....	11-5
The hello.jsp Page .....	11-7
Steps of JSP Page Processing .....	11-8



JSP Page Translation .....	11-9
JSP Page Compilation .....	11-10
JSP Page Class Loading .....	11-11
JSP Page Servlet Instance .....	11-12
JSP Page Initialization .....	11-13
JSP Page Service .....	11-14
JSP Page Destroyed .....	11-15
Developing and Deploying JSP Pages .....	11-16
Writing JSP Scripting Elements .....	11-18
Comments .....	11-19
Directive Tag .....	11-20
Declaration Tag .....	11-21
Scriptlet Tag .....	11-22
Expression Tag .....	11-23
Implicit Variables .....	11-24
Using the page Directive .....	11-27
Using Standard Tags .....	11-33
JavaBeans™ Components .....	11-34
The CustomerBean JavaBeans Component .....	11-35
The useBean Tag .....	11-37
The setProperty Tag .....	11-40
The getProperty Tag .....	11-42
Using Expression Language Elements .....	11-45
Bean Access Using EL .....	11-46
EL Implicit Objects .....	11-47
Arithmetic Operators .....	11-50
Comparisons and Logical Operators .....	11-52
Configuring the JSP Environment .....	11-55
Summary .....	11-57



<b>Developing JSP Pages Using Custom Tags .....</b>	<b>12-1</b>
Objectives .....	12-2
Relevance .....	12-3
Job Roles Revisited .....	12-4
Contrasting Custom Tags and Scriptlet Code .....	12-6
Developing JSP Pages Using Custom Tags .....	12-9
Custom Tag Library Overview .....	12-10
Custom Tag Syntax Rules .....	12-12
JSTL Sample Tags .....	12-13
The set Tag .....	12-14
The if Tag .....	12-16
The forEach Tag .....	12-18
The url Tag .....	12-21
The out Tag .....	12-23
Developing JSP Pages Using a Custom Tag Library .....	12-26
Using an Empty Custom Tag .....	12-27
Using a Conditional Custom Tag .....	12-28
Using an Iterative Custom Tag .....	12-29
Summary .....	12-31
<b>Developing Web Applications Using Struts Action Forms .....</b>	<b>13-1</b>
Objectives .....	13-2
Relevance .....	13-3
Struts Application Components .....	13-4
Struts Activity Diagram .....	13-5
Model Elements Review .....	13-6
Control Elements Review .....	13-7
View Elements Review .....	13-9
Developing an ActionForm Class .....	13-11
The Add a New League Form .....	13-12



The AddLeagueForm Class (Parts 1 and 2) .....	13-13
The AddLeagueForm Class (Part 3) .....	13-15
The AddLeagueForm Class (Part 4) .....	13-16
Struts ActionError Class .....	13-17
How the Controller Uses the Form Bean .....	13-18
The AddLeagueAction Class .....	13-19
The AddLeagueAction Class (Part 2) .....	13-20
The AddLeagueAction Class (Part 3) .....	13-21
Developing the JSP Code for a View Form .....	13-23
Struts html Tag Library Overview .....	13-24
The add_league.jsp Page .....	13-25
The add_league.jsp Page (Part 3) .....	13-27
Configuring the View Forms .....	13-29
Configure the View Aspects of the Actions .....	13-30
Summary .....	13-31

<b>Building Reusable Web Presentation Components .....</b>	<b>14-1</b>
Objectives .....	14-2
Relevance .....	14-3
Complex Page Layouts .....	14-4
Presentation Segment Overview .....	14-6
Organizing Presentation Segments .....	14-7
Including JSP Page Segments .....	14-10
Using the include Directive .....	14-11
Using the jsp:include Standard Action .....	14-12
Using the jsp:param Standard Action .....	14-13
Developing Layouts Using Struts Tiles .....	14-17
The layoutPage.jsp Page .....	14-18
The layoutPage.jsp Page (Part 2) .....	14-19
The layoutPage.jsp Page (Part 3) .....	14-20



The layoutPage.jsp Page (Part 4) .....	14-21
Tiles Layout .....	14-22
Content Body .....	14-23
Summary .....	14-24





# Preface

## About This Course



## Course Goals

- Write servlets using the Java™ programming language (Java servlets)
- Create robust web applications using Struts, session management, filters, and database integration
- Write pages created with the JavaServer Pages™ technology (JSP™ pages)
- Create easy to maintain JSP pages using the Expression Language, JSP Standard Tag Library (JSTL), and the Struts Tiles framework
- Create robust web applications that integrate Struts and JSP pages

# Course Map

## Java™ Servlet Application Strategies

Introduction to Web  
Application Technologies

Developing a  
View Component

Developing a  
Controller Component

Developing Dynamic  
Forms

Sharing Application  
Resources Using the  
Servlet Context

Designing the  
Business Tier

Developing Web  
Applications Using  
Struts

Developing Web  
Applications Using  
Session Management

Using Filters in  
Web Applications

Integrating Web  
Applications With  
Databases

## JSP™ Application Strategies

Developing  
JSP™ Pages

Developing  
JSP Pages  
Using Custom Tags

Developing Web  
Applications Using  
Struts Action Forms

Building Reusable  
Web Presentation  
Components



## 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 WJT-310: *Java™ 2 Platform, Enterprise Edition: Technology Overview*
- 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 application programming interface (API)?
- Can you analyze and design a software system using a modeling language like Unified Modeling Language (UML)?
- Can you create a simple web page using Hypertext Markup Language (HTML)?



## 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 Java™ 2 Platform, Standard Edition (J2SE™ platform), 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.



## Typographical Conventions

- ***Courier italic bold*** is used to represent variables whose values are to be entered by the student as part of an activity.
- *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 Common Gateway Interface (CGI) scripting
- Describe JavaServer Pages technology and list three benefits of JSP technology over rival template page technologies
- Describe the Java<sup>TM</sup> 2 Platform, Enterprise Edition (J2EE<sup>TM</sup> platform)



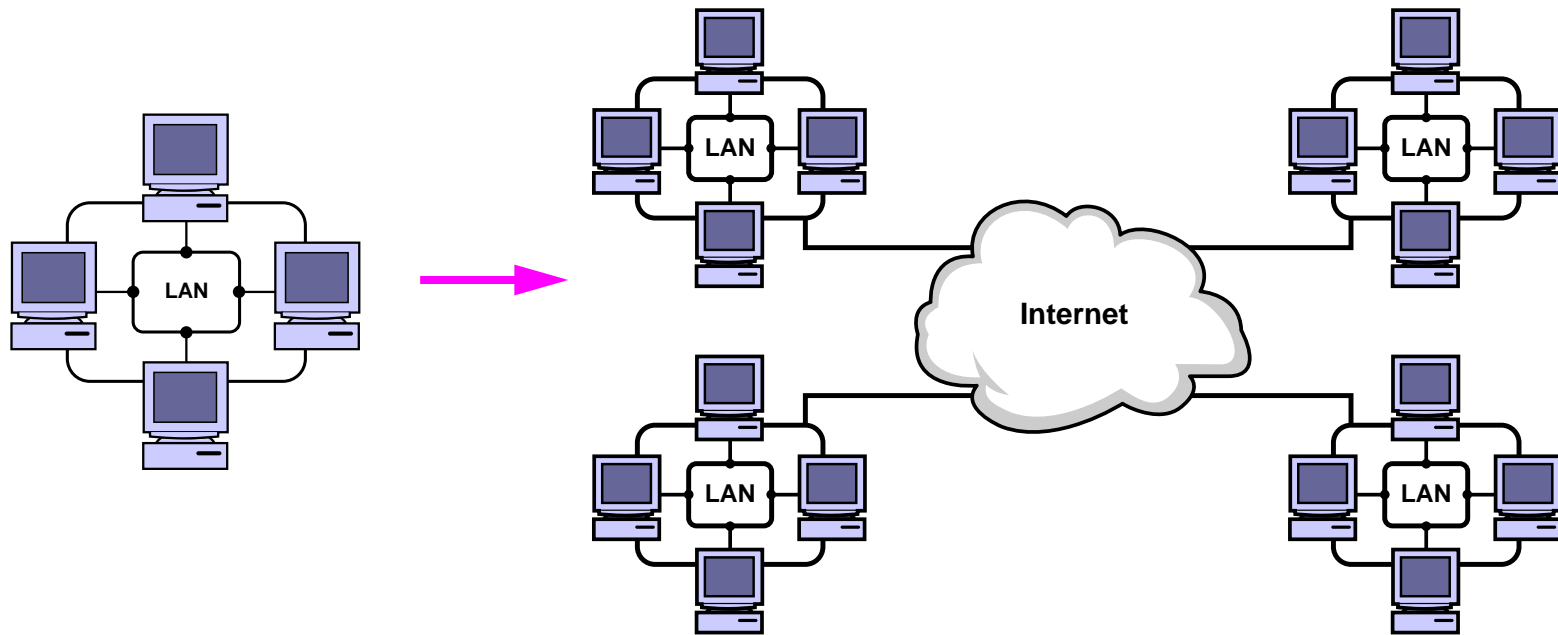
## Relevance

- What web applications have you developed?
- Were there goals you could not achieve because of the technology you used?



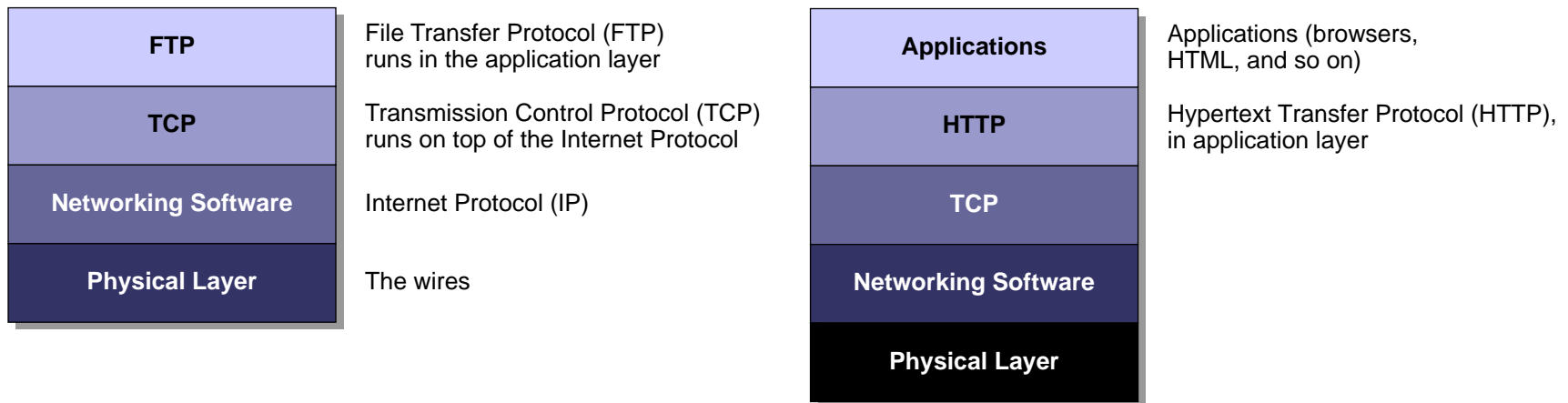
# The Internet Is a Network of Networks

Evolution of the Internet:

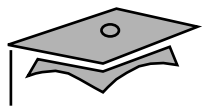




# Networking Protocol Stack

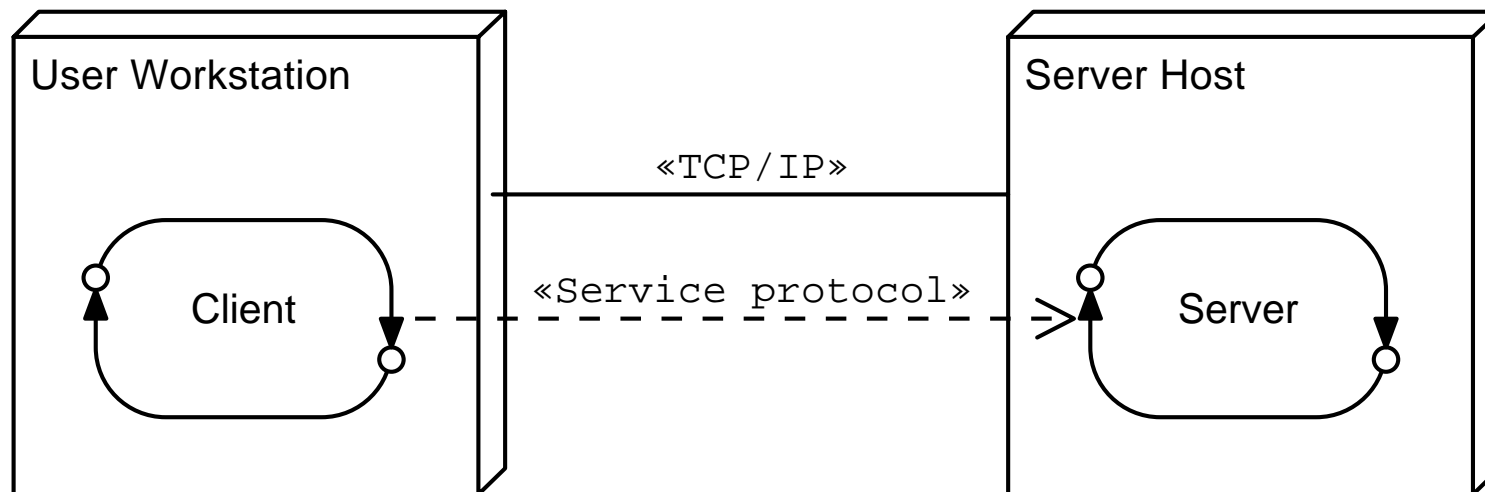






# Client-Server Architecture

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





## 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 Common Gateway Interface (CGI) scripting
- Describe JavaServer Pages technology and list three benefits of JSP technology over rival template page technologies
- Describe the Java 2 Platform, Enterprise Edition (J2EE platform)



# World Wide Web Introduction

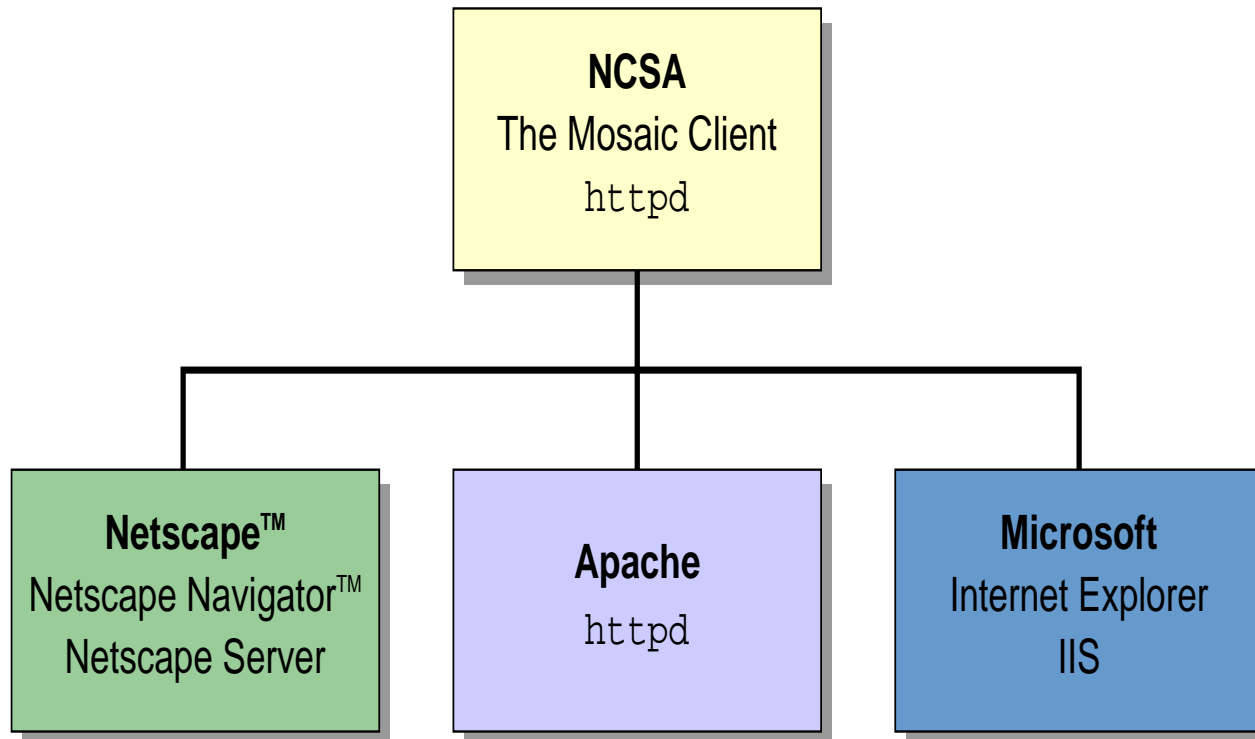
## Hypertext Transfer Protocol:

- 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 can 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

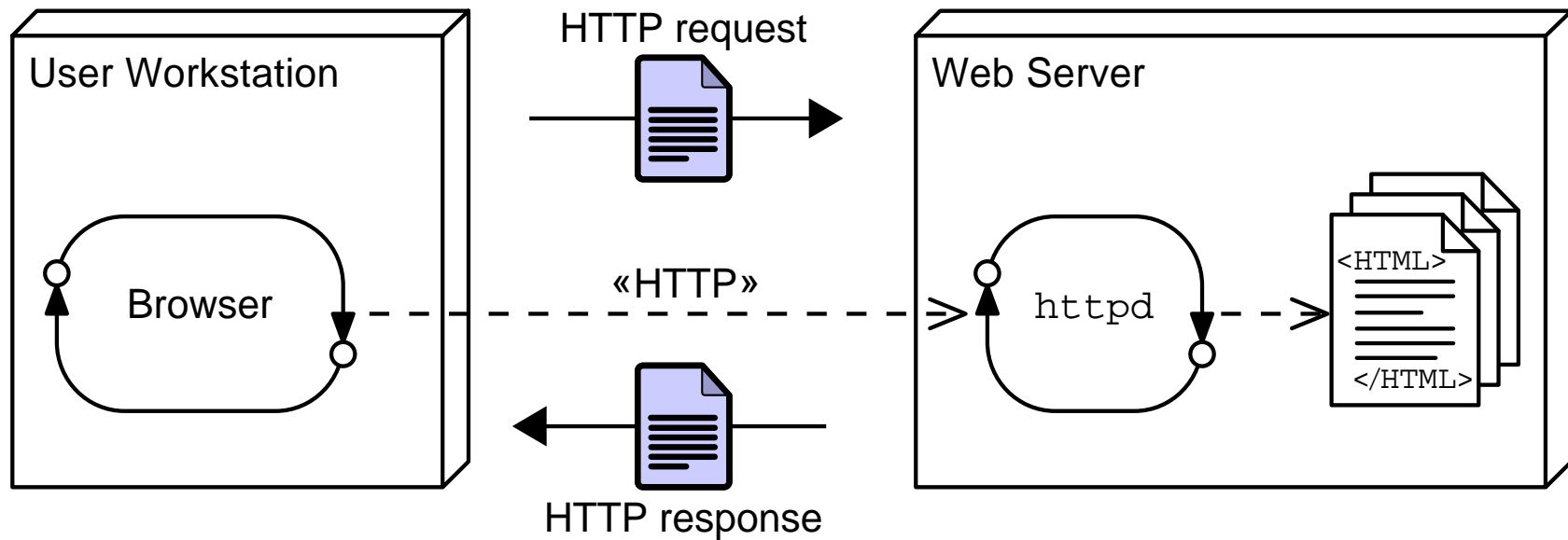
Development of browsers and servers:





# HTTP Client-Server Architecture

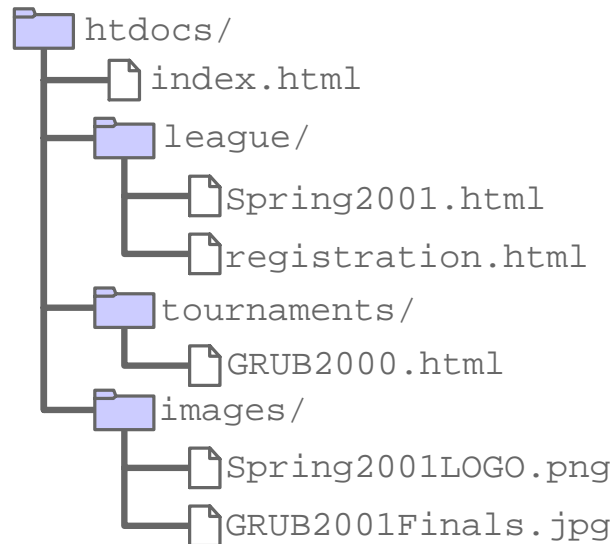
Deployment diagram of a web server:





## Web Site Structure

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`



## 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 Common Gateway Interface (CGI) scripting
- Describe JavaServer Pages technology and list three benefits of JSP technology over rival template page technologies
- Describe the Java 2 Platform, Enterprise Edition (J2EE platform)



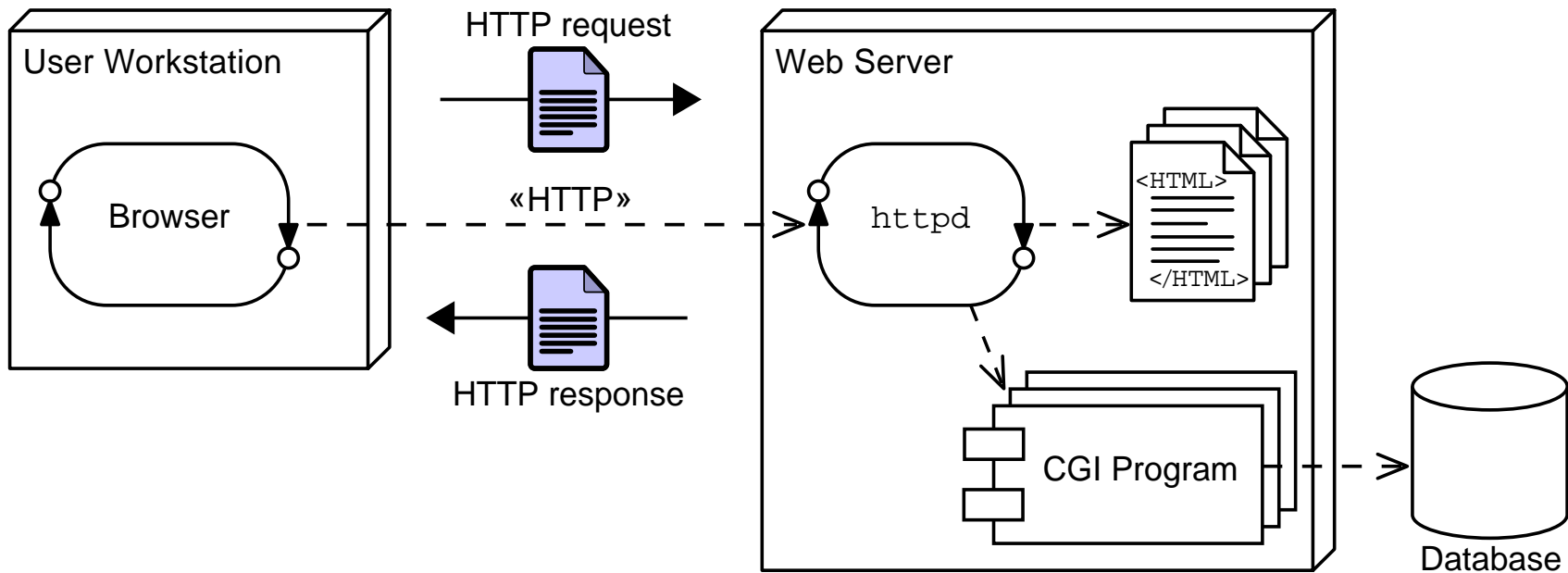
## Web Applications and Web Sites

- 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 CGI.
  - The CGI data is sent in the HTTP request stream.



# CGI Programs on the Web Server

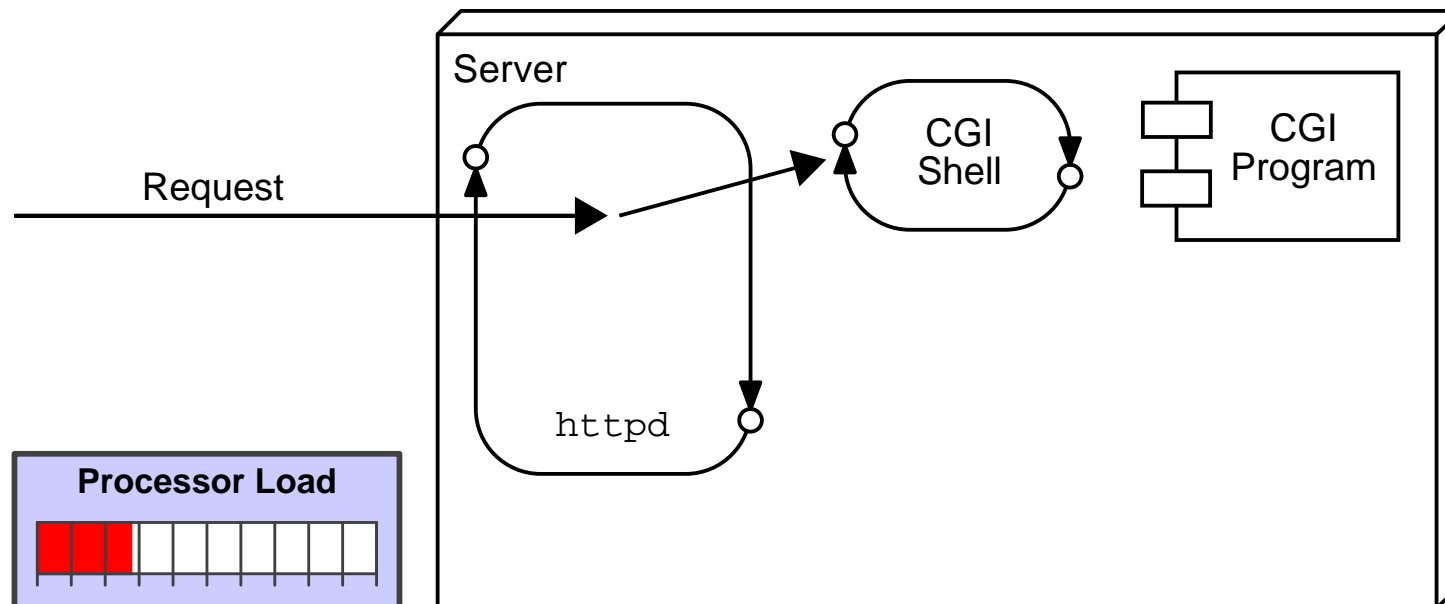
Deployment diagram of a web server with CGI programs:





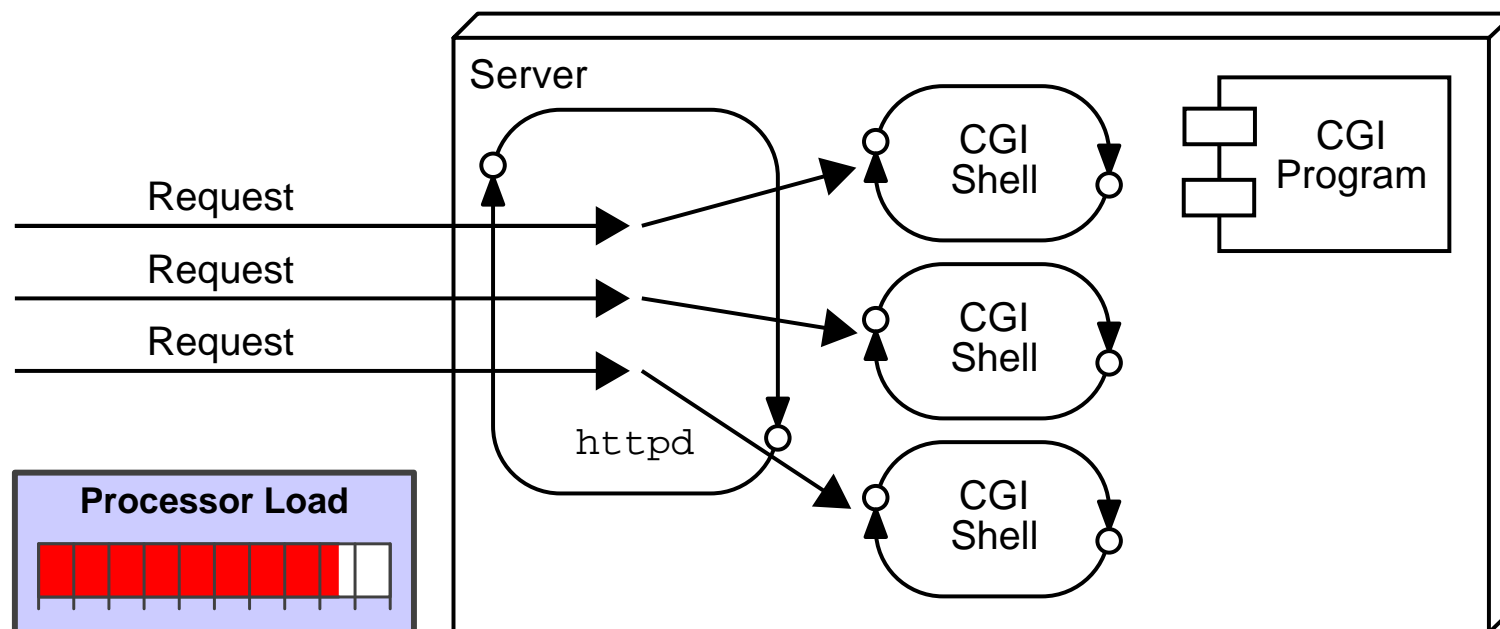
# Execution of CGI Programs

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 reference
- CGI program disadvantages:
  - Each shell is heavyweight
  - Not scalable
  - CGI processing code (business logic) is mingled with HTML (presentation logic)
  - Language is not always secure or object-oriented
  - Language is not always platform independent



## 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 Common Gateway Interface (CGI) scripting
- Describe JavaServer Pages technology and list three benefits of JSP technology over rival template page technologies
- Describe the Java 2 Platform, Enterprise Edition (J2EE platform)



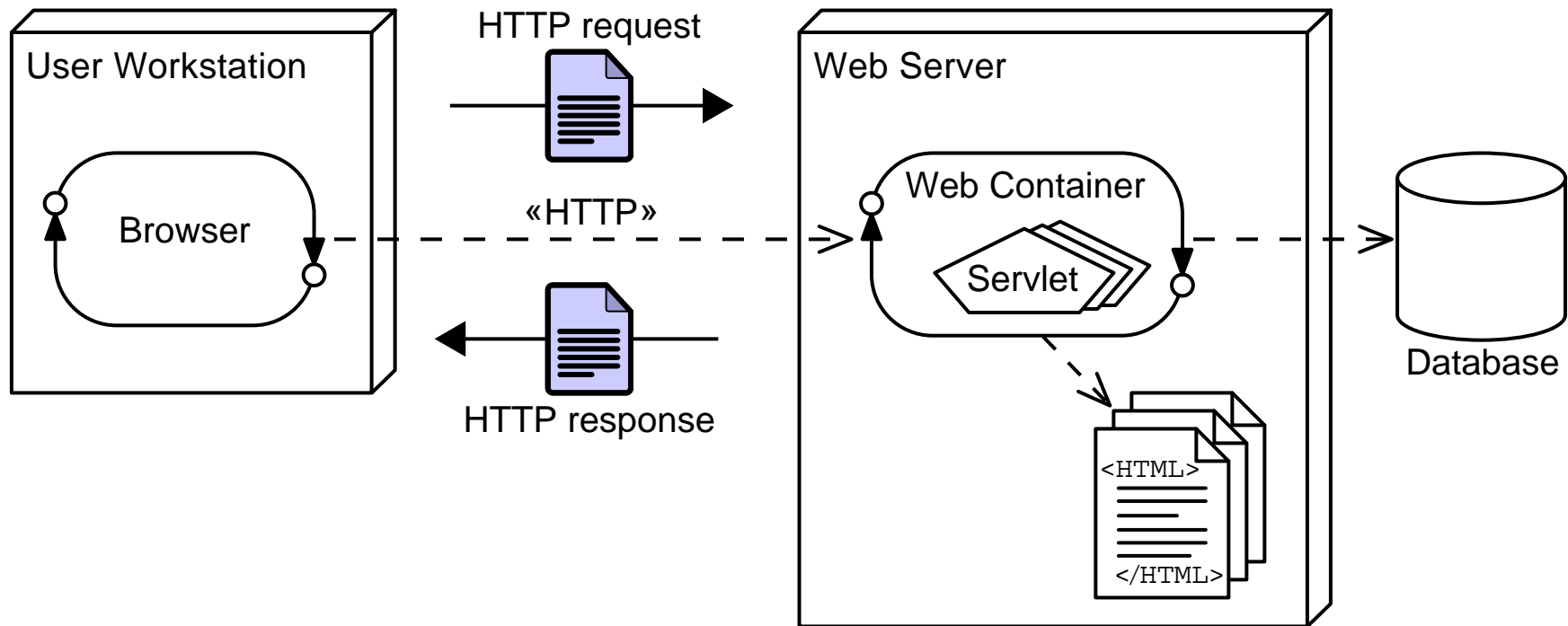
## Java Servlets

- 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 Java™ Virtual Machine (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

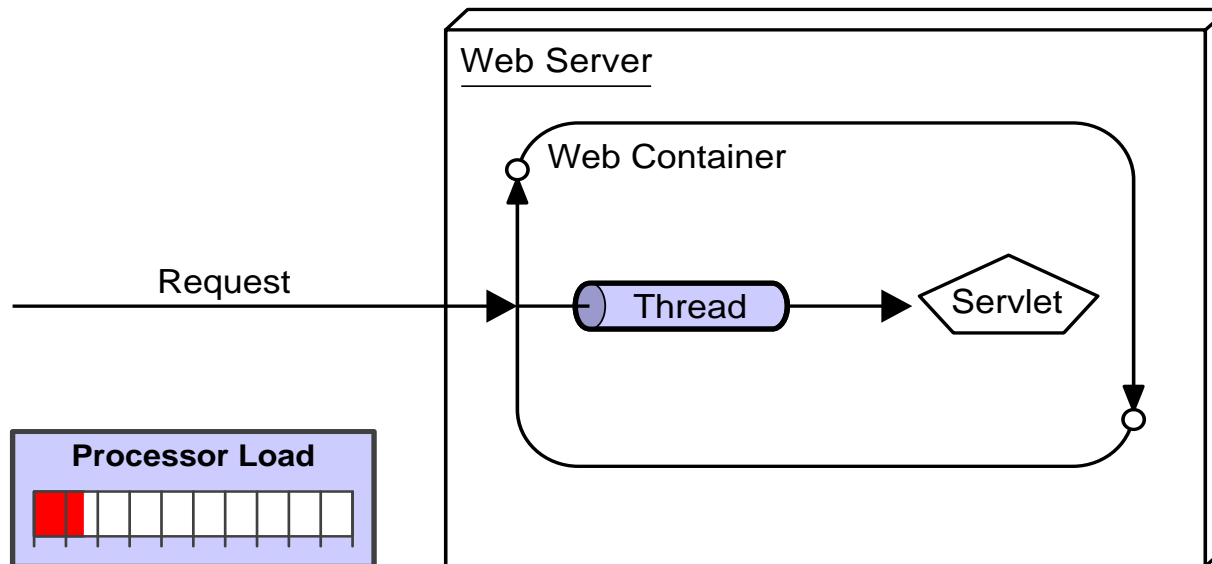
Deployment diagram of a web server with a web container:



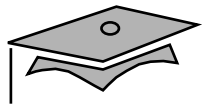


# Execution of Java Servlets

How servlets work with one request:

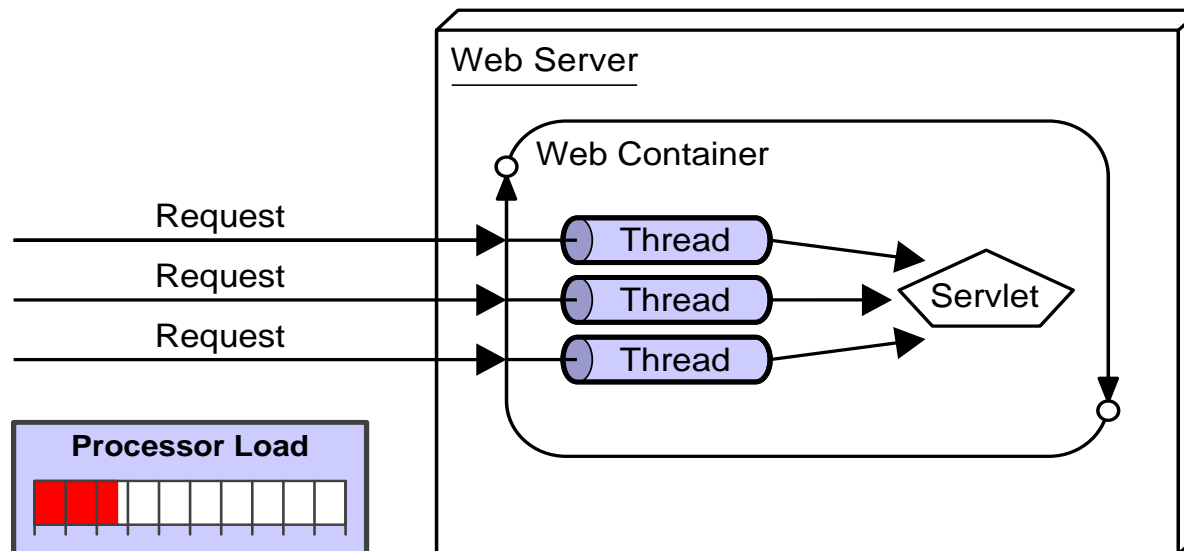






# Execution of Java Servlets

How servlets work with many requests:





# Advantages and Disadvantages of Java Servlets

- Java 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
- Java servlet disadvantages:
  - Separation of concerns: business and presentation logic
  - Concurrency issues



## 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 Common Gateway Interface (CGI) scripting
- Describe JavaServer Pages technology and list three benefits of JSP technology over rival template page technologies
- Describe the Java 2 Platform, Enterprise Edition (J2EE platform)



# JavaServer Pages Technology

- 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, are compiled, and are treated just like servlets in the web container.
- If designed well, JSP pages focus on the presentation logic, not on the business logic.
- In JSP pages, custom tags and JSP Expression Language provide for reusable code and separation of concerns.
- In a Java technology web application, JSP pages are often used in conjunction with servlets and business objects in a Model-View-Controller pattern.



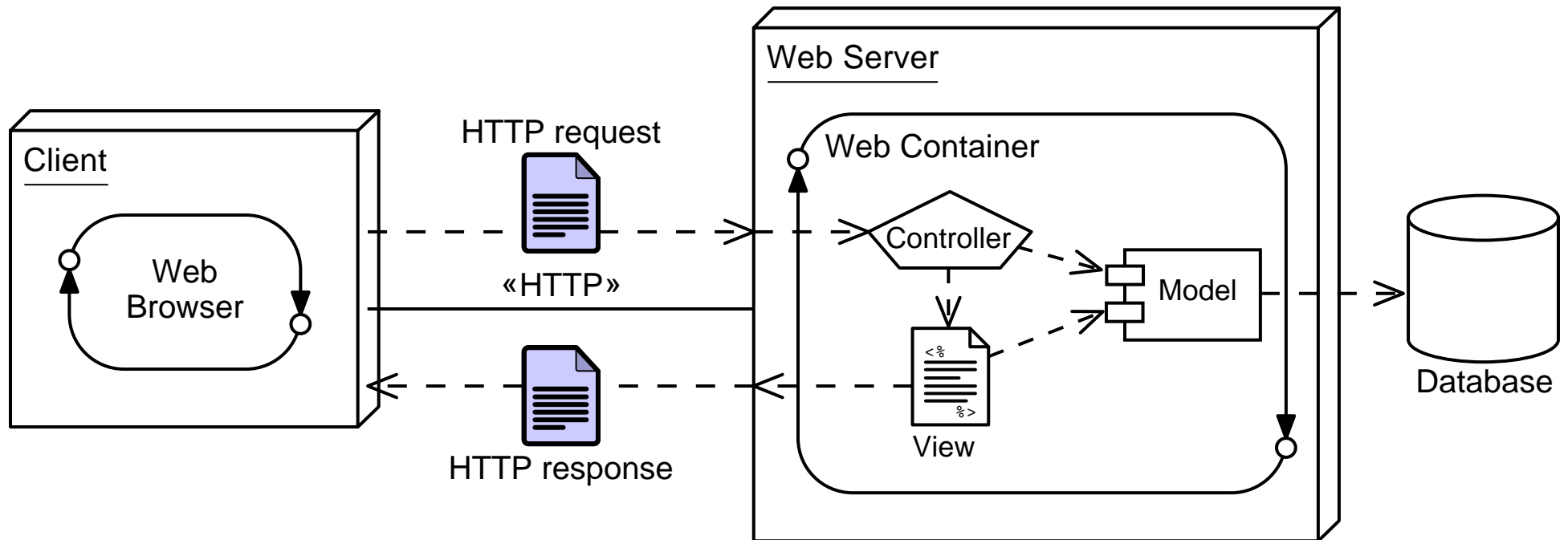
# Advantages and Disadvantages of JavaServer Pages

- 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. JSP pages are also difficult to debug.



# Model 2 Architecture

Deployment diagram of a web container using Model 2 architecture:







## Model 2 Frameworks

- Frameworks are partial implementations on which you can build your components.
- There are several Model 2 frameworks available:
  - Struts from the Jakarta group
  - JavaServer™ Faces technology from Sun
  - Velocity from Apache



## 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 Common Gateway Interface (CGI) scripting
- Describe JavaServer Pages technology and list three benefits of JSP technology over rival template page technologies
- Describe the Java 2 Platform, Enterprise Edition (J2EE platform)

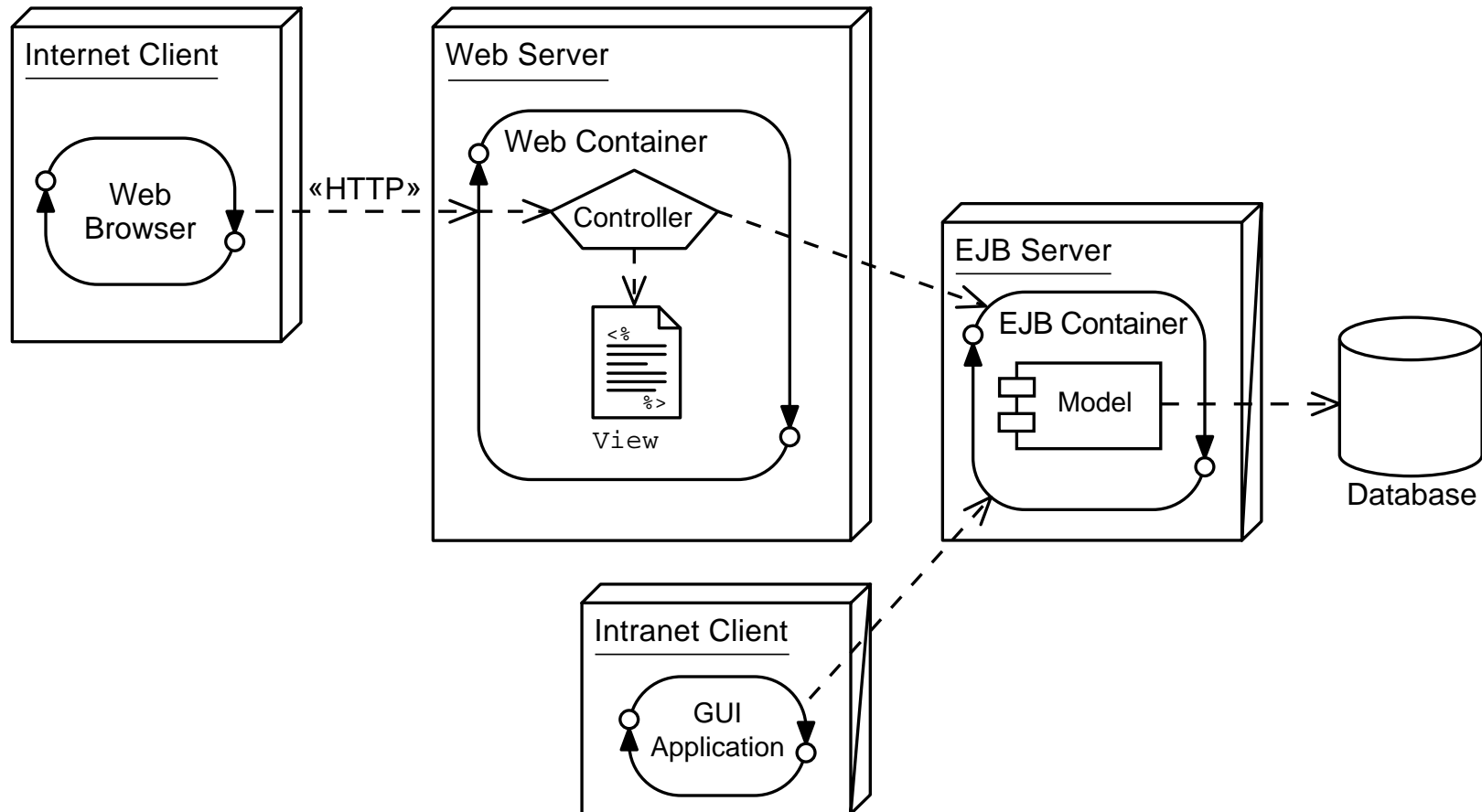


## J2EE™ Platform

- Modular design allows for easier modification of the business logic.
- Enterprise components can use container-provided services such as security, transaction, persistence, and life cycle management.



# J2EE Architecture Example





## Job Roles

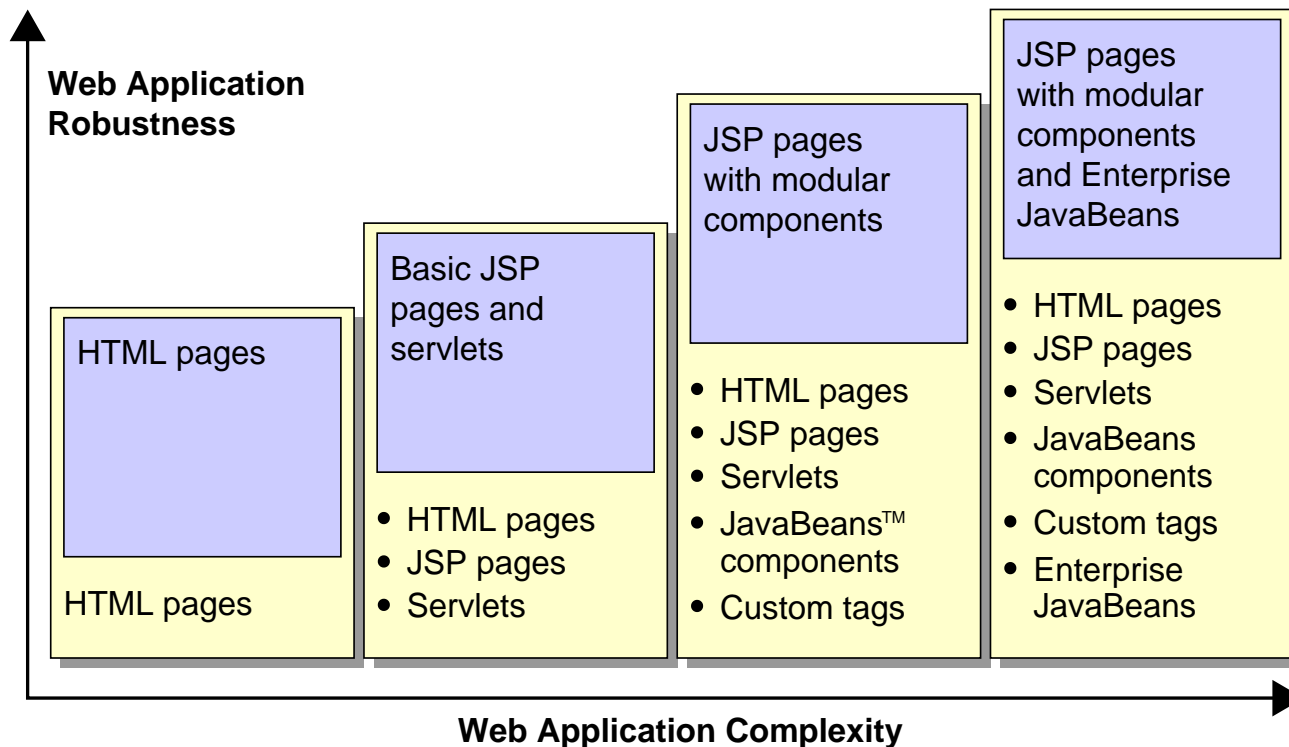
The modularity of a J2EE architecture clearly distinguishes several job roles:

- Web Designer – 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

- The HTTP protocol is built on top of the standard Transmission Control Protocol/Internet Protocol (TCP/IP) protocol stack.
- CGI provided hooks for web servers to execute application programs.
- Java servlets are like CGI, but they execute in a JVM using threading.
- JSP pages are like servlets, but they are better suited for generating HTML content.
- The Model 2 architecture uses servlets in conjunction with JSP pages to build web applications.
- Well designed web applications, using Model 2 can be easily migrated to more complex J2EE architectures.



# Module 2

## Developing a View Component





# Objectives

- Design a view component
- Describe the Hypertext Transfer Protocol
- Describe the web container behavior
- Develop a view servlet
- Configure and deploy a servlet



## Relevance

- What is a view component?
- What types of view components are you familiar with?

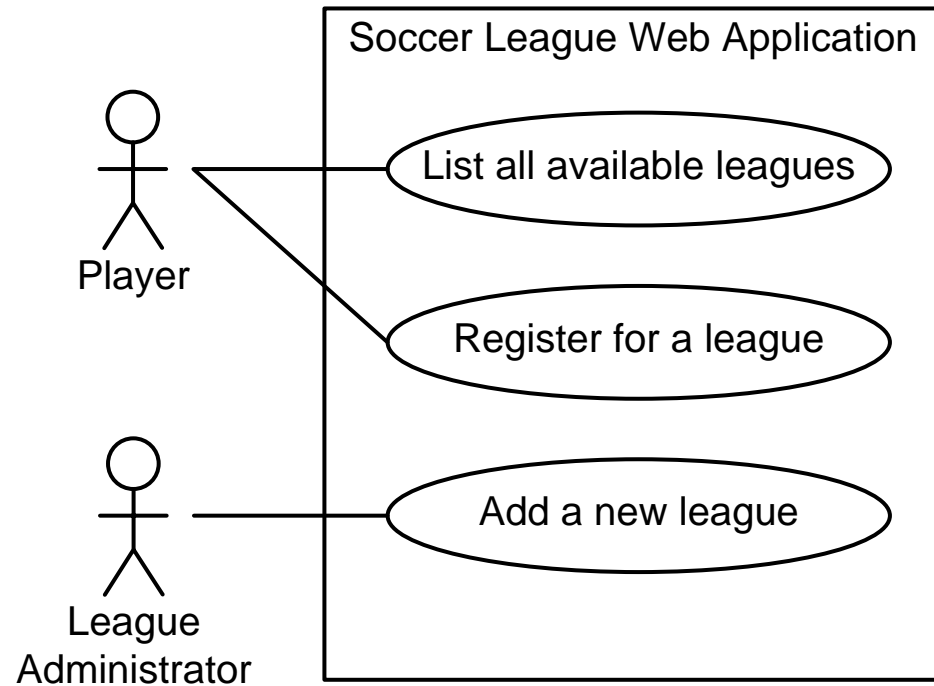


## Types of View Components

- Data presentation
- Data forms
- Navigational aids
- Informational screens or pop-ups

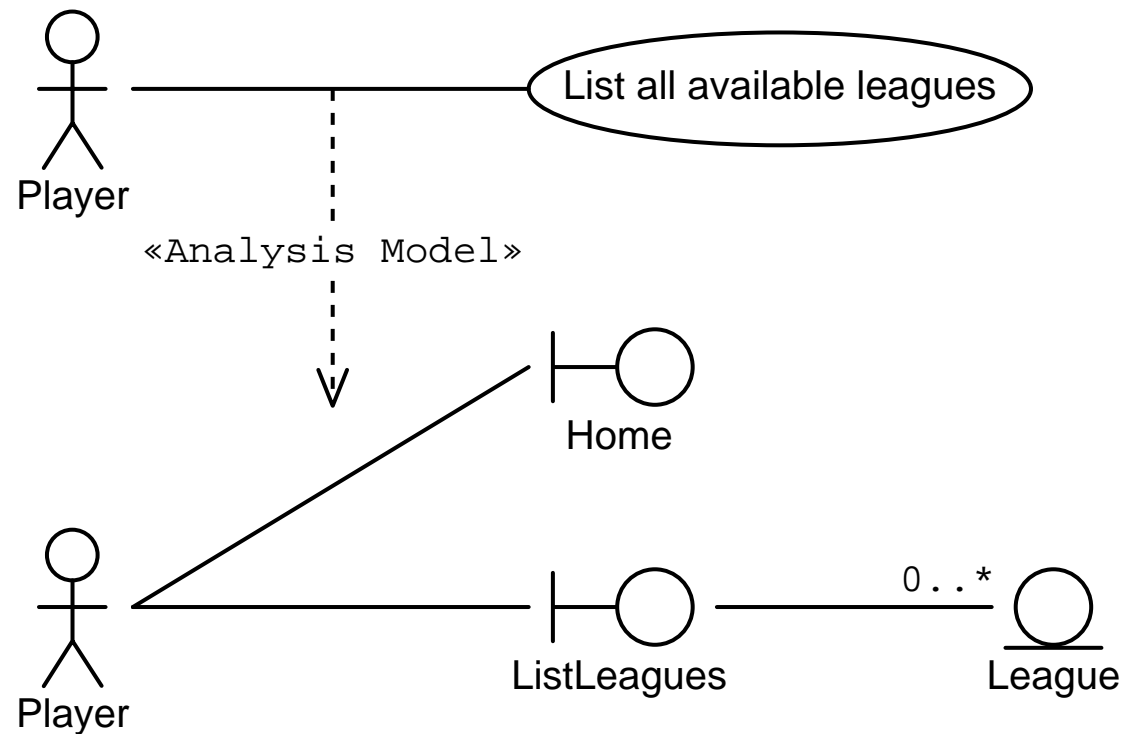


# Soccer League Case Study





# List Leagues Analysis Model

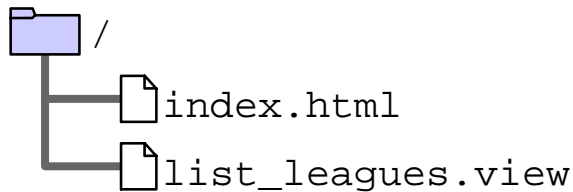




# List Leagues Page Flow



The logical web application hierarchy:





# Home Page HTML

```
1  <html>
2
3  <head>
4  <title>Duke's Soccer League: Home</title>
5  </head>
6
7  <body bgcolor='white'>
8
9  <!-- Page Heading -->
10 <table border='1' cellpadding='5' cellspacing='0' width='400'>
11 <tr bgcolor='#CCCCFF' align='center' valign='center' height='20'>
12   <td><h3>Duke's Soccer League: Home</h3></td>
13 </tr>
14 </table>
15
```



## Home Page HTML

```
16 <p>
17 This is the Home page for Duke's Soccer League.
18 </p>
19
20 <h3>Players</h3>
21
22 <ul>
23   <li><a href='list_leagues.view'>List all leagues</a></li>
24   <li>Register for a league (TBA)</li>
25 </ul>
26
27 <h3>League Administrator</h3>
28
29 <ul>
30   <li>Add a new league (TBA)</li>
31 </ul>
32
33 </body>
34
35 </html>
```





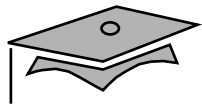
## List Leagues Page HTML

```
10 <table border='1' cellpadding='5' cellspacing='0' width='400'>
11 <tr bgcolor='#CCCCFF' align='center' valign='center' height='20'>
12   <td><h3>Duke's Soccer League: List Leagues</h3></td>
13 </tr>
14 </table>
15
16 <p>
17 The set of soccer leagues are:
18 </p>
19
20 <ul>
21   <li>The Summer of Soccer Love 2004</li>
22   <li>Fall Soccer League (2003)</li>
23   <li>Fall Soccer League (2004)</li>
24   <li>Soccer League (Spring '03)</li>
25   <li>Summer Soccer Fest 2003</li>
26   <li>Soccer League (Spring '04)</li>
27 </ul>
28
29 </body>
30 </html>
```



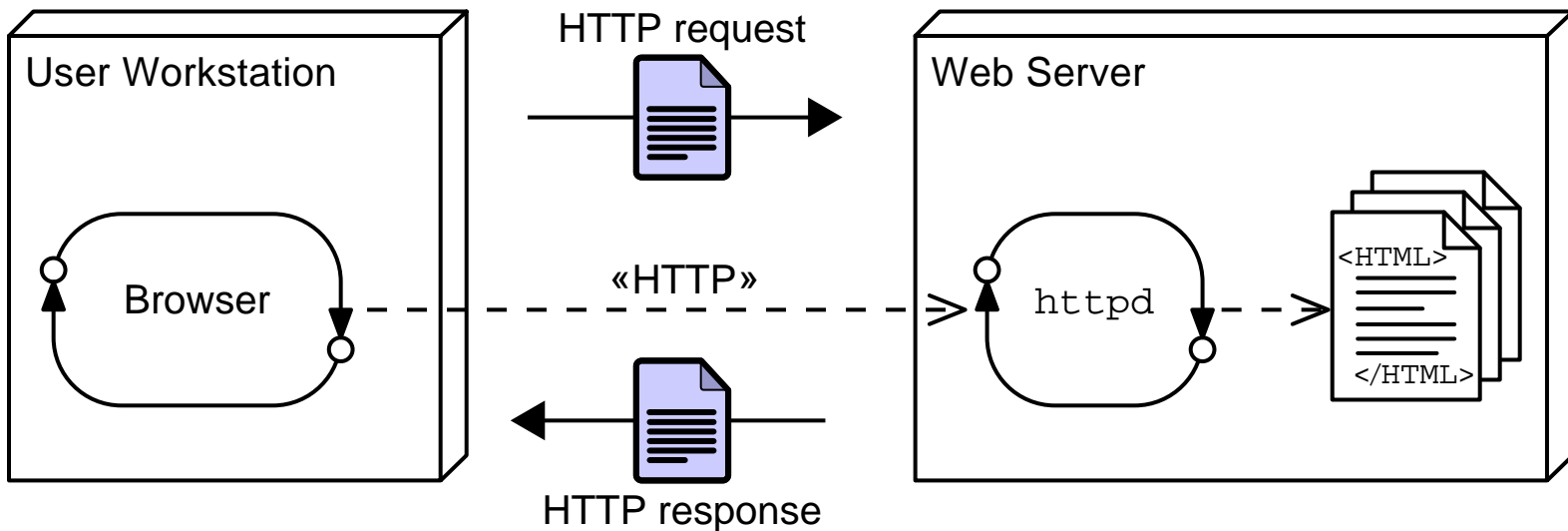
## Objectives

- Design a view component
- Describe the Hypertext Transfer Protocol
- Describe the web container behavior
- Develop a view servlet
- Configure and deploy a servlet



# Hypertext Transfer Protocol

The HTTP sends a single request to the HTTP daemon (httpd) and responds with the requested resource.

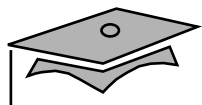




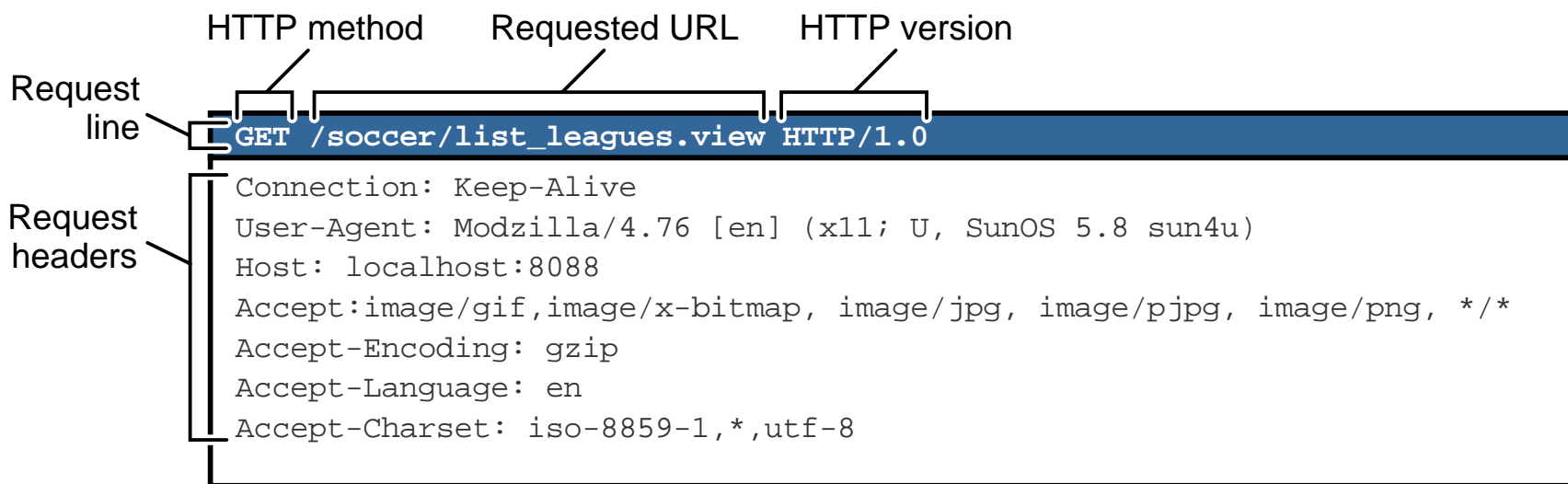
## HTTP GET Method

A web browser issues an HTTP GET request when:

- The user selects a link in the current HTML page
- The user enters a Universal Resource Locator (URL) in the Location field (Netscape Navigator™) or the Address field (Microsoft Internet Explorer)



# HTTP Request





# HTTP Request Headers

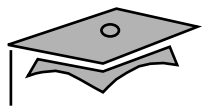
Headers are provided in the request by the client and can modify how the request is processed on the server.

Example headers:

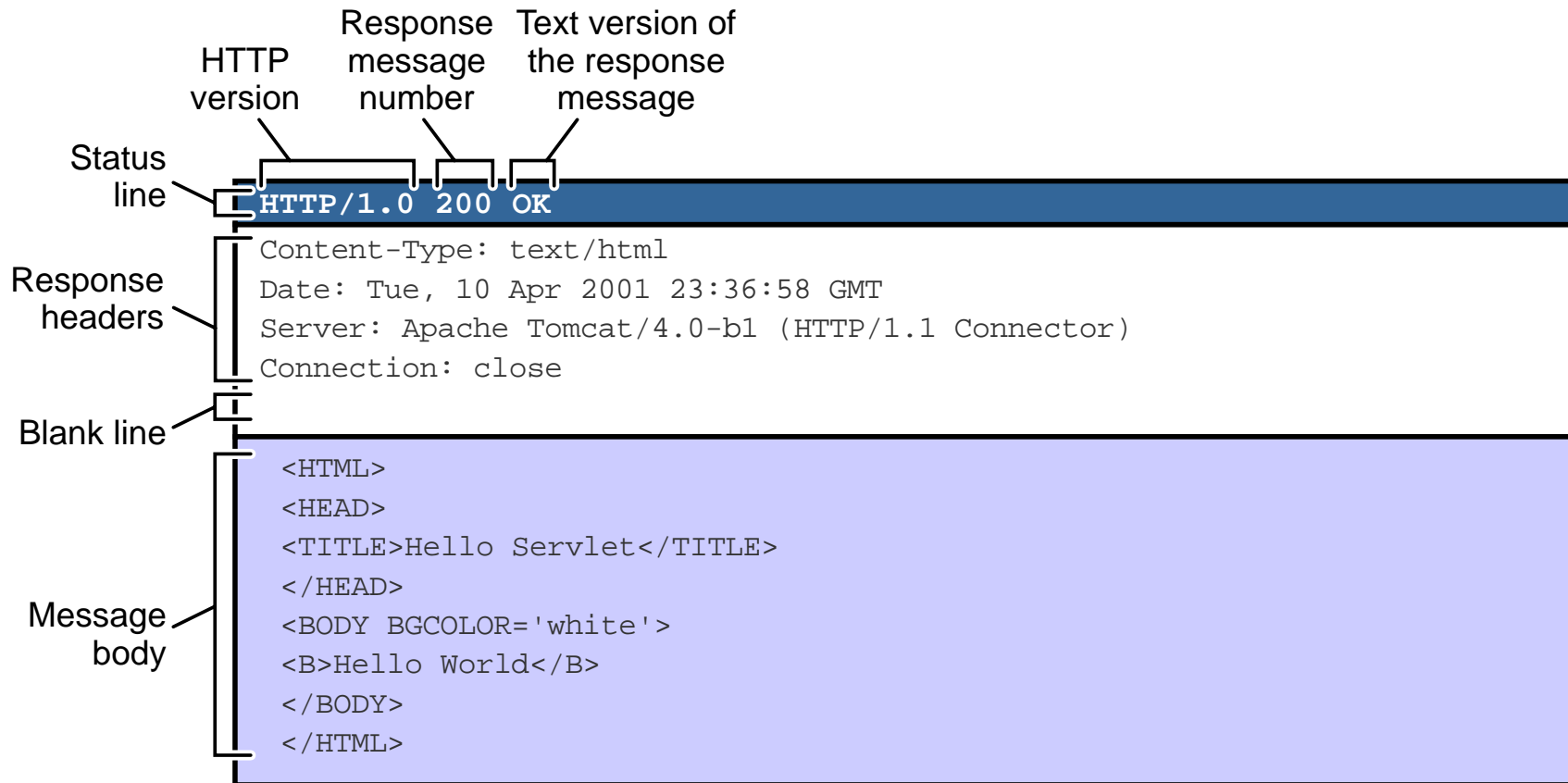
---

<b>Header</b>	<b>Use</b>
Accept	The MIME types the client can receive
Host	The internet host and port number of the resource being requested
Referer	The address from which the Request-Universal Resource Identifier (URI) was obtained
User-Agent	The information about the client originating the request

---



# HTTP Response





# HTTP Response Headers

Headers are provided in the response by the server and can modify how the response is processed on the client.

Example headers:

---

<b>Header</b>	<b>Use</b>
Content-Type	A MIME type (such as <code>text/html</code> ) which classifies the type of data in the response
Content-Length	The length (in bytes) of the payload of the response
Server	An informational string about the server that responded to this HTTP request
Cache-Control	A directive for the web browser (or proxies) to indicate whether or not the content of the response should be cached

---



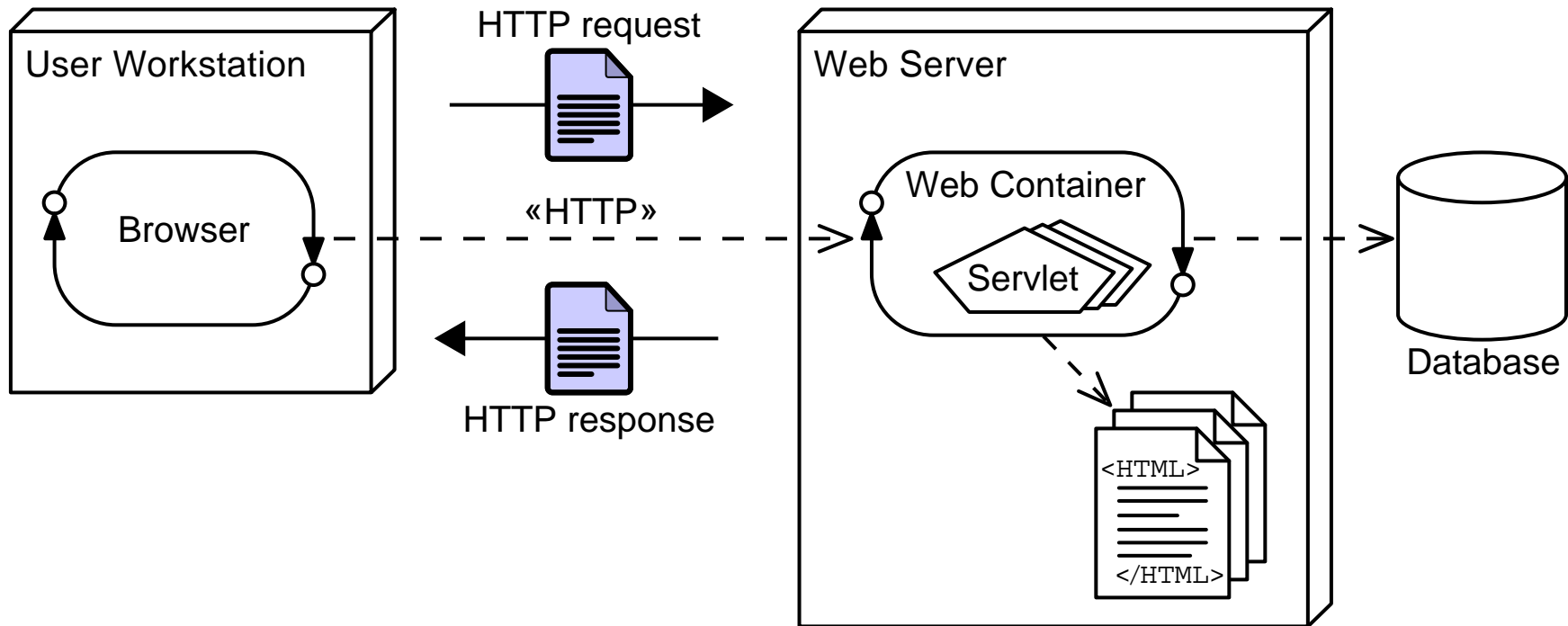


## Objectives

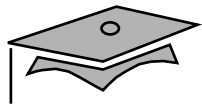
- Design a view component
- Describe the Hypertext Transfer Protocol
- Describe the web container behavior
- Develop a view servlet
- Configure and deploy a servlet



# Web Container Architecture

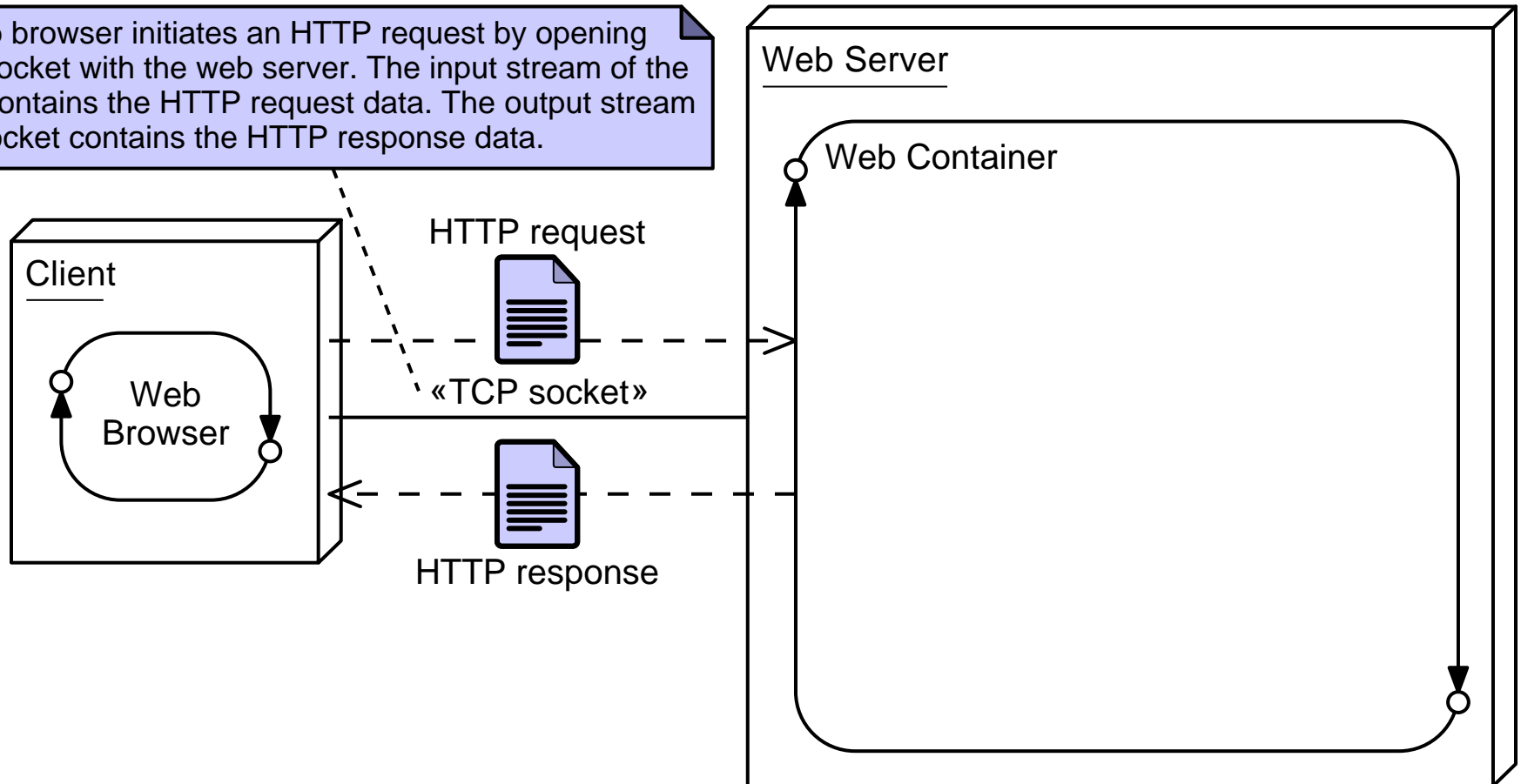


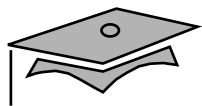
A web container can be used to process HTTP requests by executing the service method on an `HttpServlet` object.



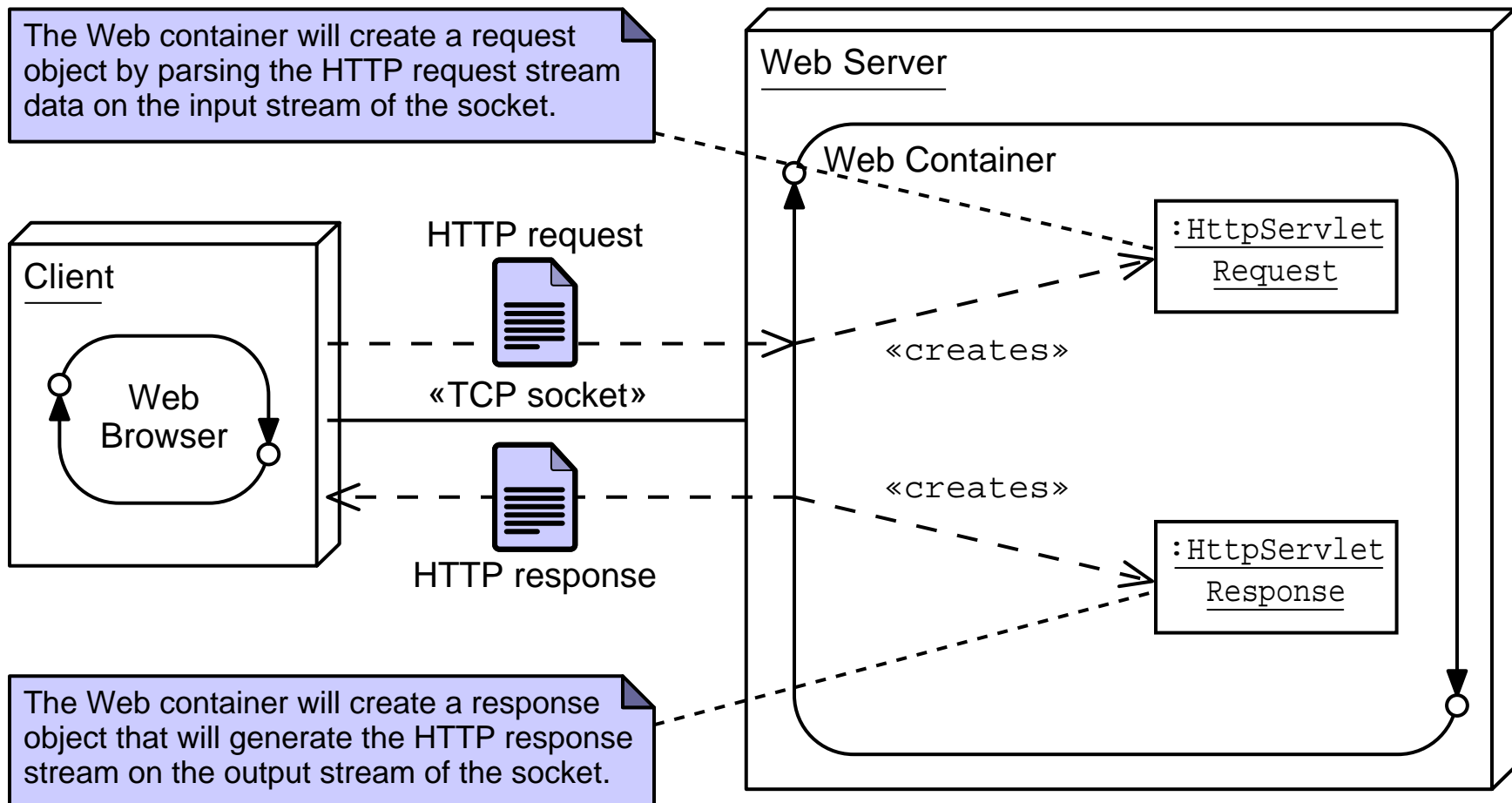
# Request and Response Process

The web browser initiates an HTTP request by opening a TCP socket with the web server. The input stream of the socket contains the HTTP request data. The output stream of the socket contains the HTTP response data.



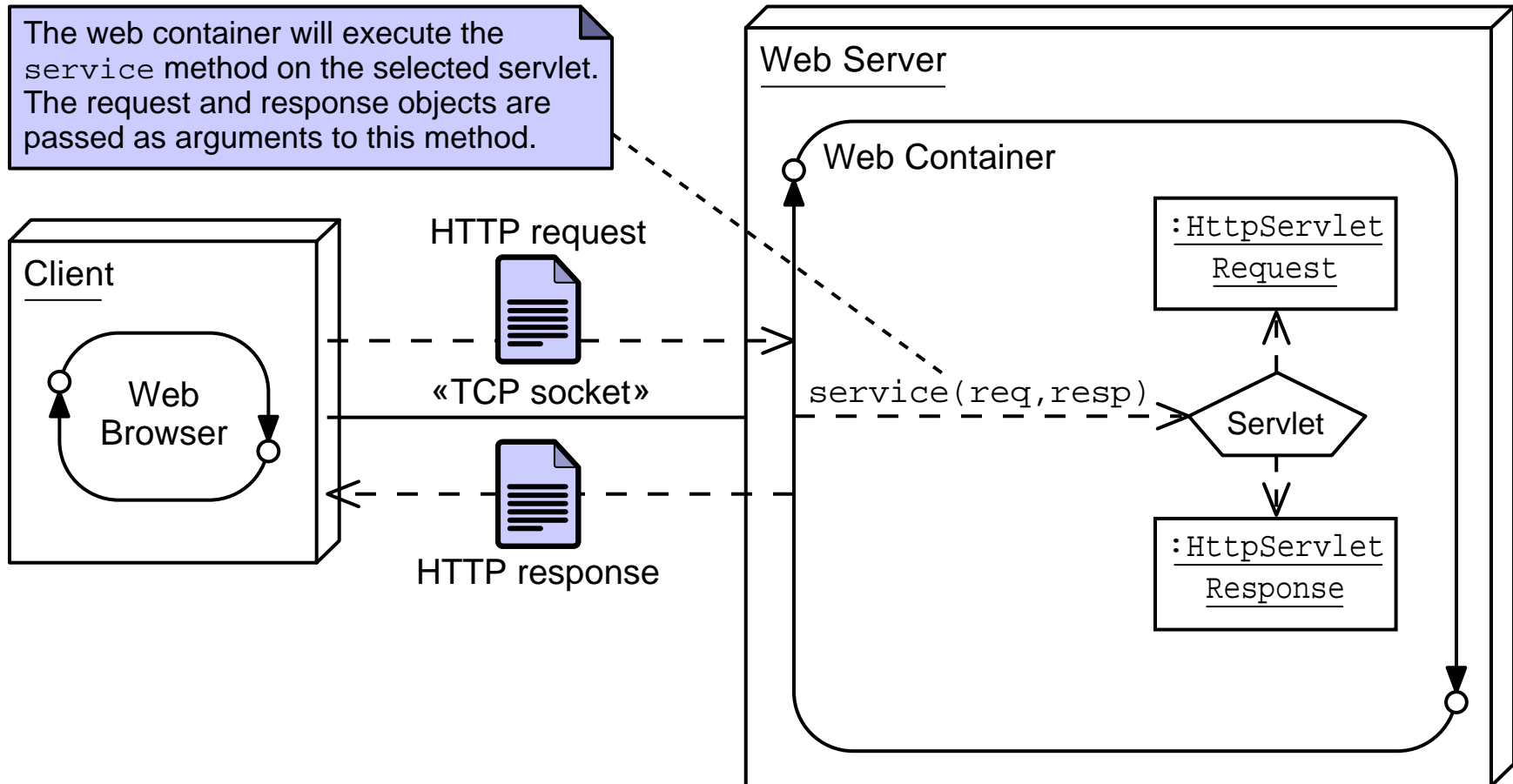


# Request and Response Process



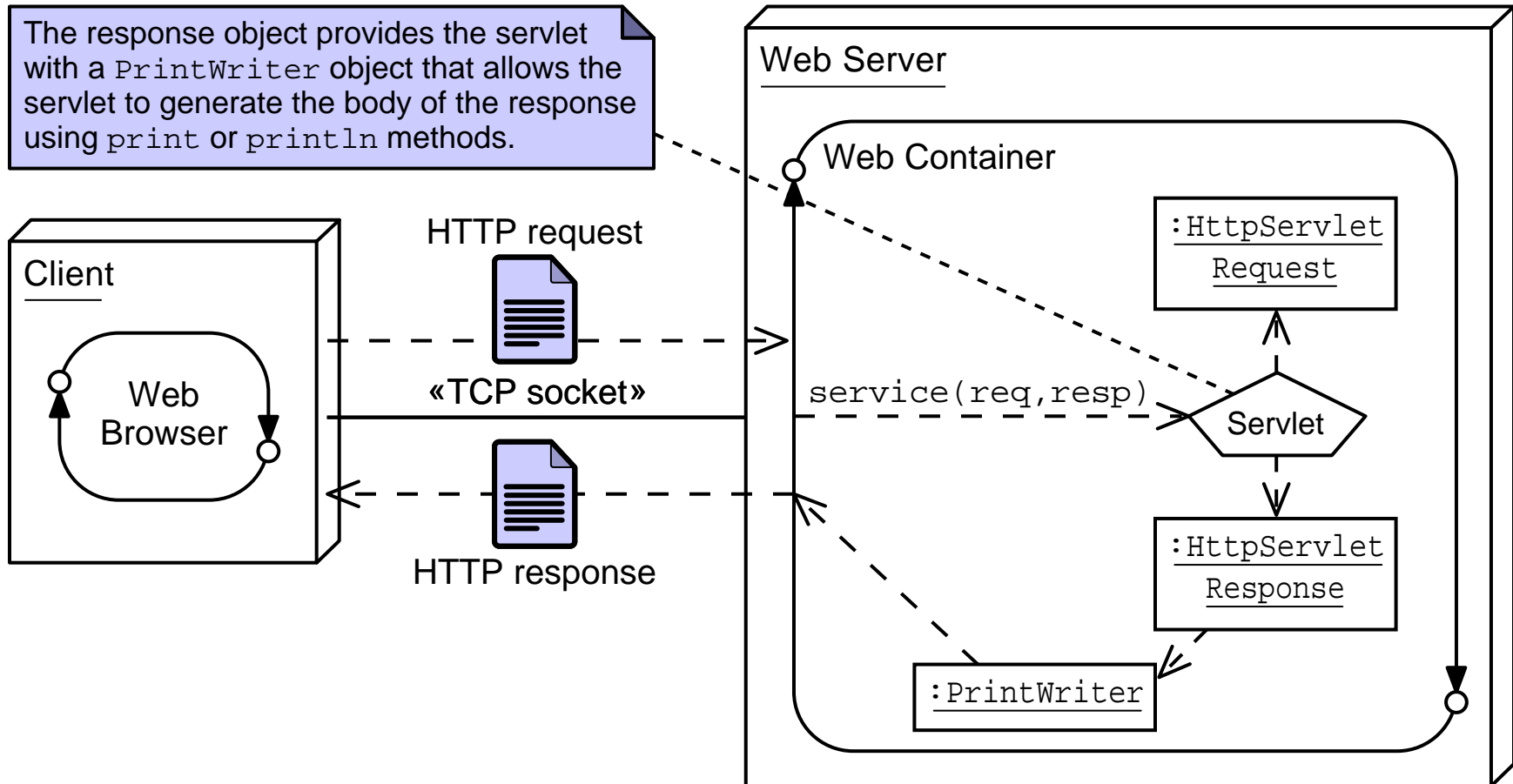


# Request and Response Process



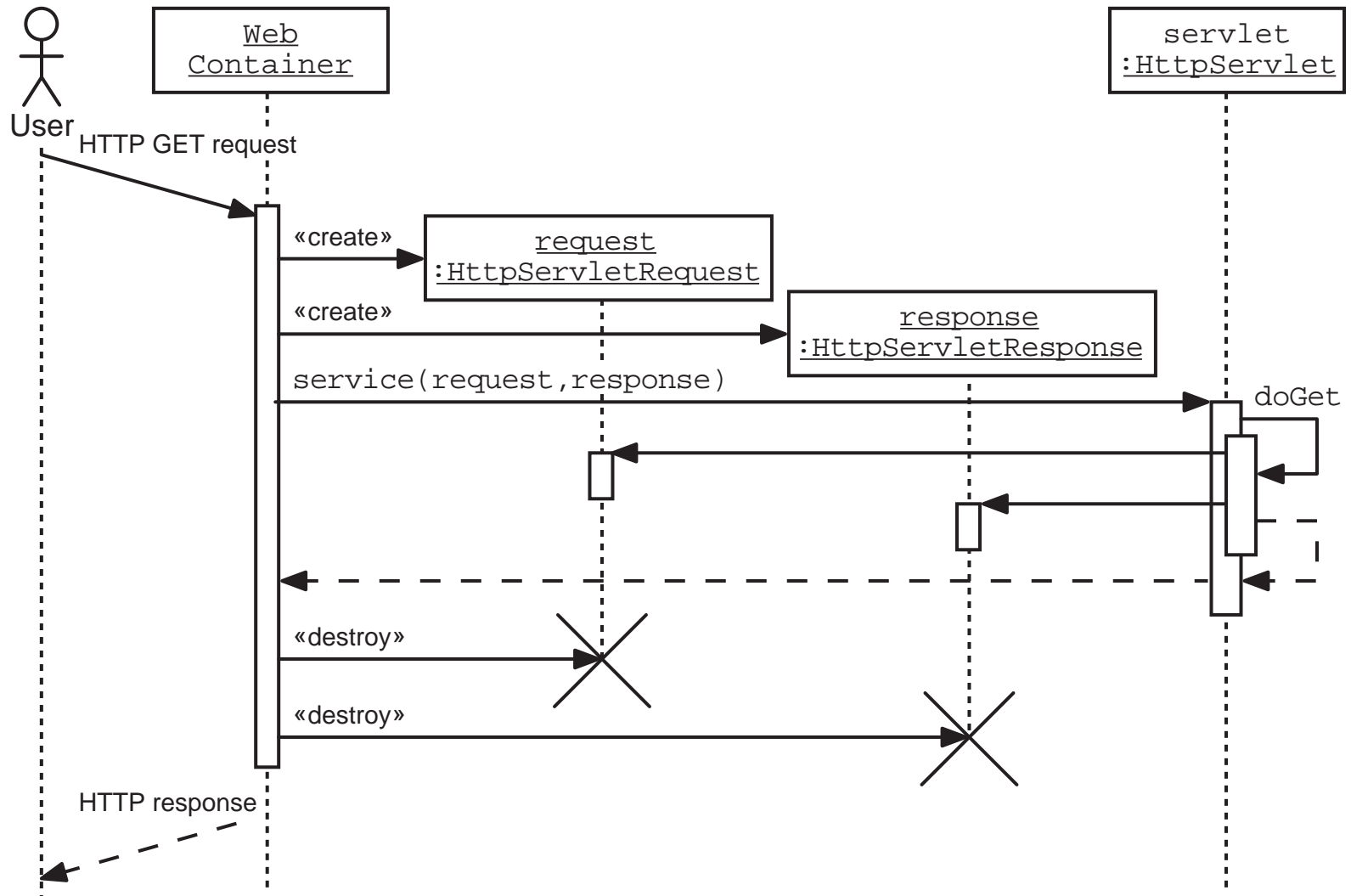


# Request and Response Process





# Sequence Diagram of an HTTP GET Request

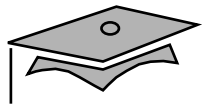




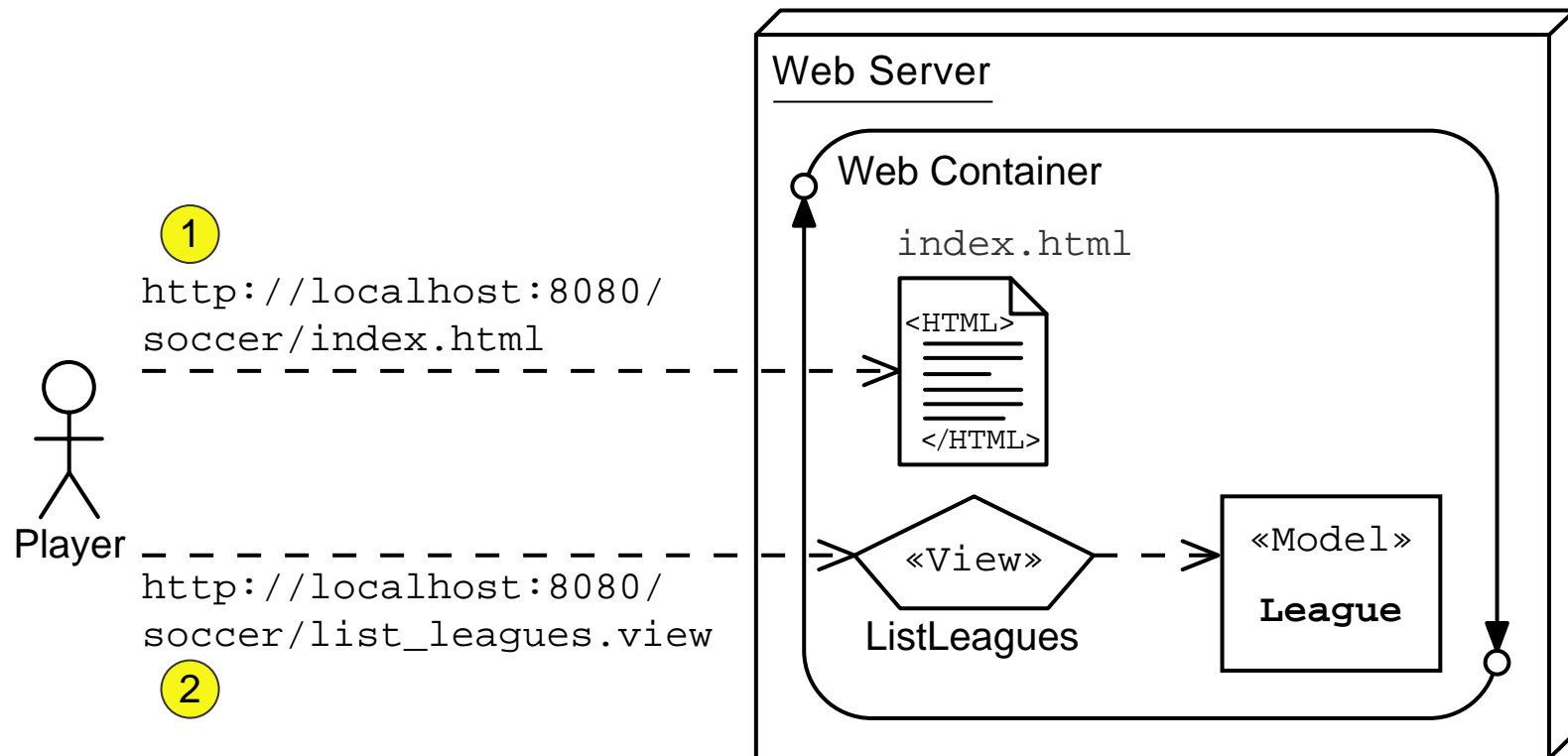
## Objectives

- Design a view component
- Describe the Hypertext Transfer Protocol
- Describe the web container behavior
- **Develop a view servlet**
- Configure and deploy a servlet

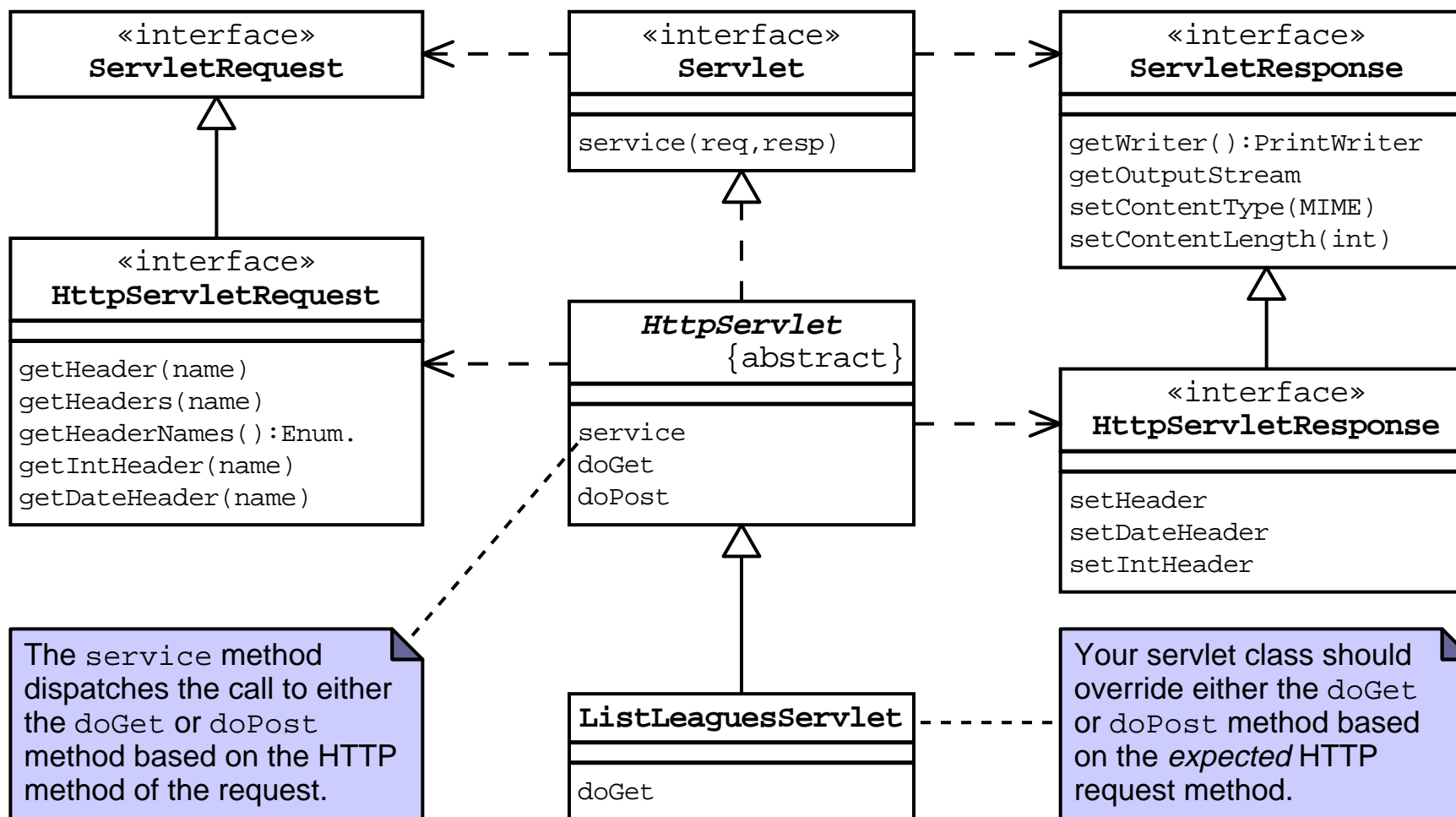




# List Leagues Architecture Model



# The HttpServlet API





## The ListLeaguesServlet Code

```
1  package sl314.view;
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  // Model classes
10 import sl314.model.League;
11 import java.util.List;
12 import java.util.LinkedList;
13 import java.util.Iterator;
14
15 public class ListLeaguesServlet extends HttpServlet {
16
17     private List leagueList = null;
18
19     public void doGet(HttpServletRequest request,
20                       HttpServletResponse response)
21         throws IOException {
```



## The ListLeaguesServlet Code (Part 2)

```
16
17     private List leagueList = null;
18
19     public void doGet(HttpServletRequest request,
20                       HttpServletResponse response)
21         throws IOException {
22
23         // Create the set of leagues
24         leagueList = new LinkedList();
25         leagueList.add( new League(2003, "Spring",
26                                   "Soccer League (Spring `03)") );
27         leagueList.add( new League(2003, "Summer",
28                                   "Summer Soccer Fest 2003") );
29         leagueList.add( new League(2003, "Fall",
30                                   "Fall Soccer League (2003)") );
31         leagueList.add( new League(2004, "Spring",
32                                   "Soccer League (Spring `04)") );
33         leagueList.add( new League(2004, "Summer",
34                                   "The Summer of Soccer Love 2004") );
35         leagueList.add( new League(2004, "Fall",
36                                   "Fall Soccer League (2004)") );
```



## The ListLeaguesServlet Code (Part 3)

```
36         "Fall Soccer League (2004)") );
37
38     // Set page title
39     String pageTitle = "Duke's Soccer League: List Leagues";
40
41     // Specify the content type is HTML
42     response.setContentType("text/html");
43     PrintWriter out = response.getWriter();
44
45     // Generate the HTML response
46     out.println("<html>");
47     out.println("<head>");
48     out.println("  <title>" + pageTitle + "</title>");
49     out.println("</head>");
50     out.println("<body bgcolor='white'>");
51
52     // Generate page heading
53     out.println("<!-- Page Heading -->");
54     out.println("<table border='1' cellpadding='5' cellspacing='0'
width='400'>");
55     out.println("<tr bgcolor='#CCCCFF' align='center' valign='center'
```



## The ListLeaguesServlet Code (Part 4)

```
height='20'>");
56     out.println("  <td><h3>" + pageTitle + "</h3></td>");
57     out.println("</tr>");
58     out.println("</table>");
59
60     // Generate main body
61     out.println("<p>");
62     out.println("The set of soccer leagues are:");
63     out.println("</p>");
64
65     out.println("<ul>");
66     Iterator items = leagueList.iterator();
67     while ( items.hasNext() ) {
68         League league = (League) items.next();
69         out.println("  <li>" + league.getTitle() + "</li>");
70     }
71     out.println("</ul>");
72
73     out.println("</body>");
74     out.println("</html>");
75 } // END of doGet method
```



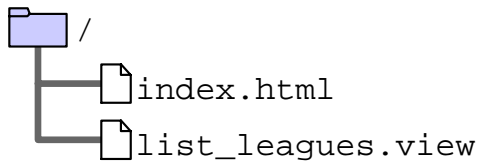
## Objectives

- Design a view component
- Describe the Hypertext Transfer Protocol
- Describe the web container behavior
- Develop a view servlet
- **Configure and deploy a servlet**

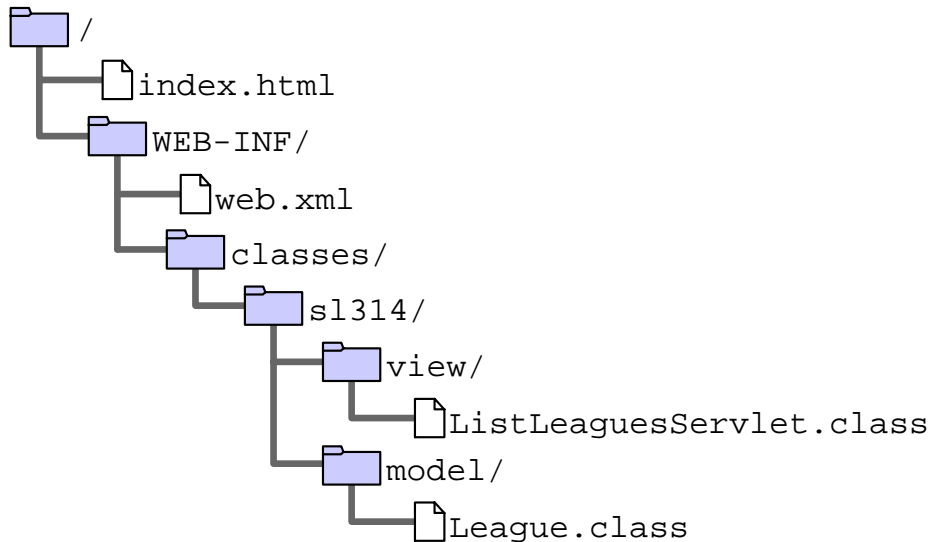


# Soccer League Web Application Structure

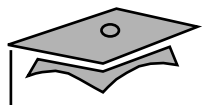
The logical web application hierarchy:



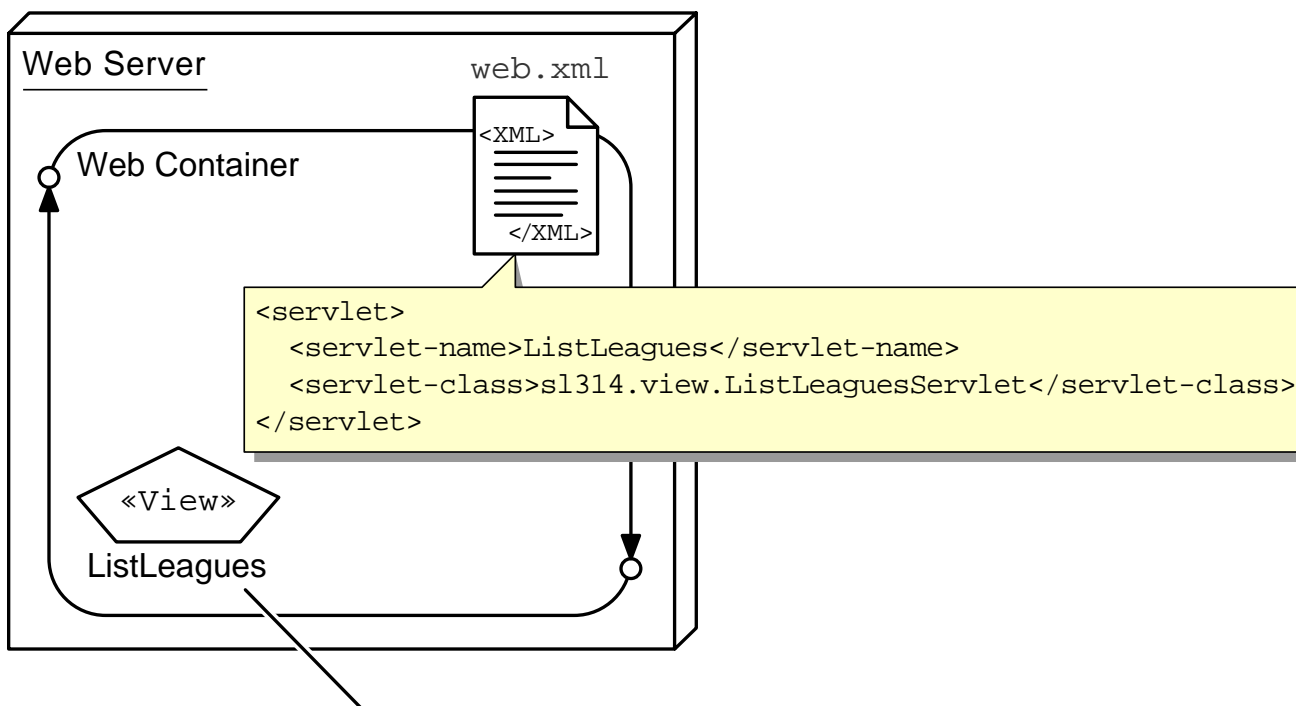
The physical web application hierarchy:







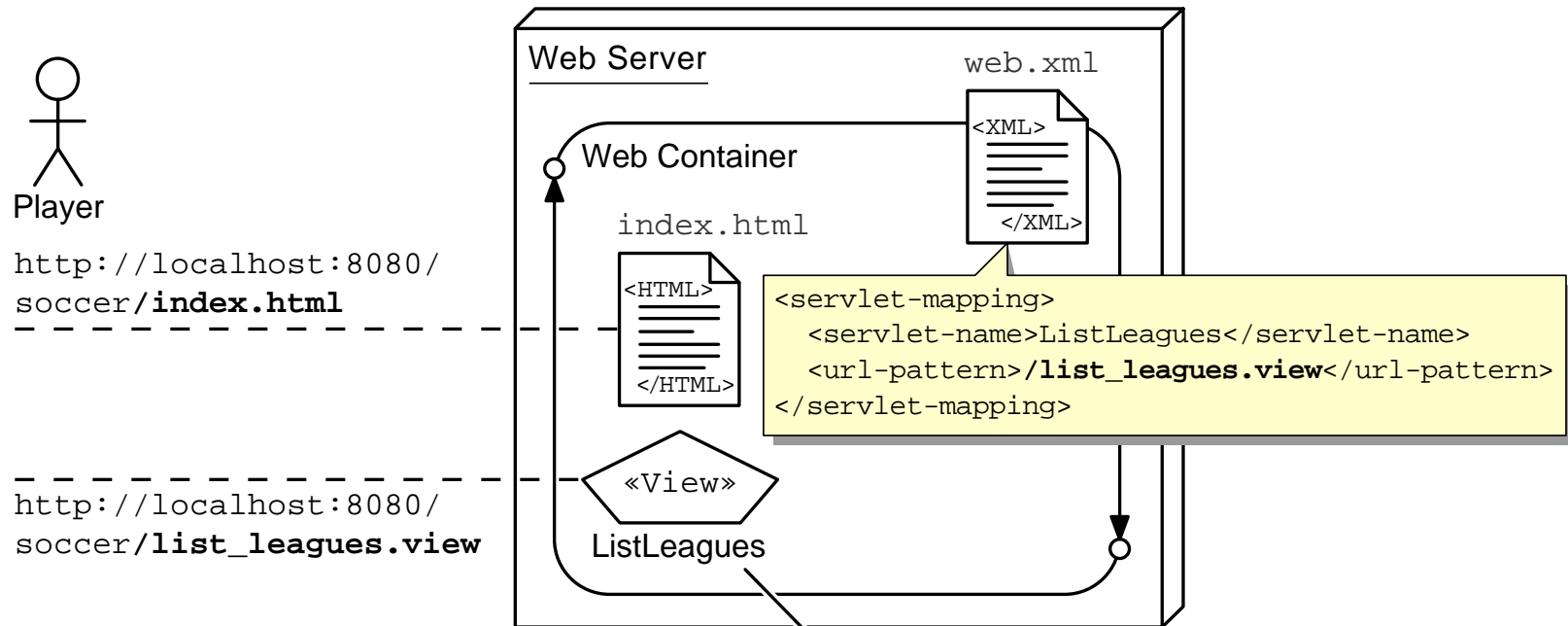
# Configuring a Servlet Definition



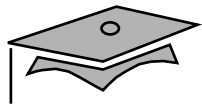
The web container will create one, and only one, servlet object for each definition in the deployment descriptor.



# Configuring a Servlet Mapping



The web container maps the URL of each request to either a physical web resource (such as an HTML page) or to a logical web resource (such as a servlet).



# Complete Deployment Descriptor

```
1  <?xml version="1.0" encoding="ISO-8859-1"?>
2
3  <web-app
4      xmlns="http://java.sun.com/xml/ns/j2ee"
5      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
6      xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
7                          http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
8      version="2.4">
9
10     <display-name>SL-314 WebApp Example</display-name>
11     <description>
12         This Web Application demonstrates a single View servlet.
13     </description>
14
15     <servlet>
16         <servlet-name>ListLeagues</servlet-name>
17         <servlet-class>sl314.view.ListLeaguesServlet</servlet-class>
18     </servlet>
19
```

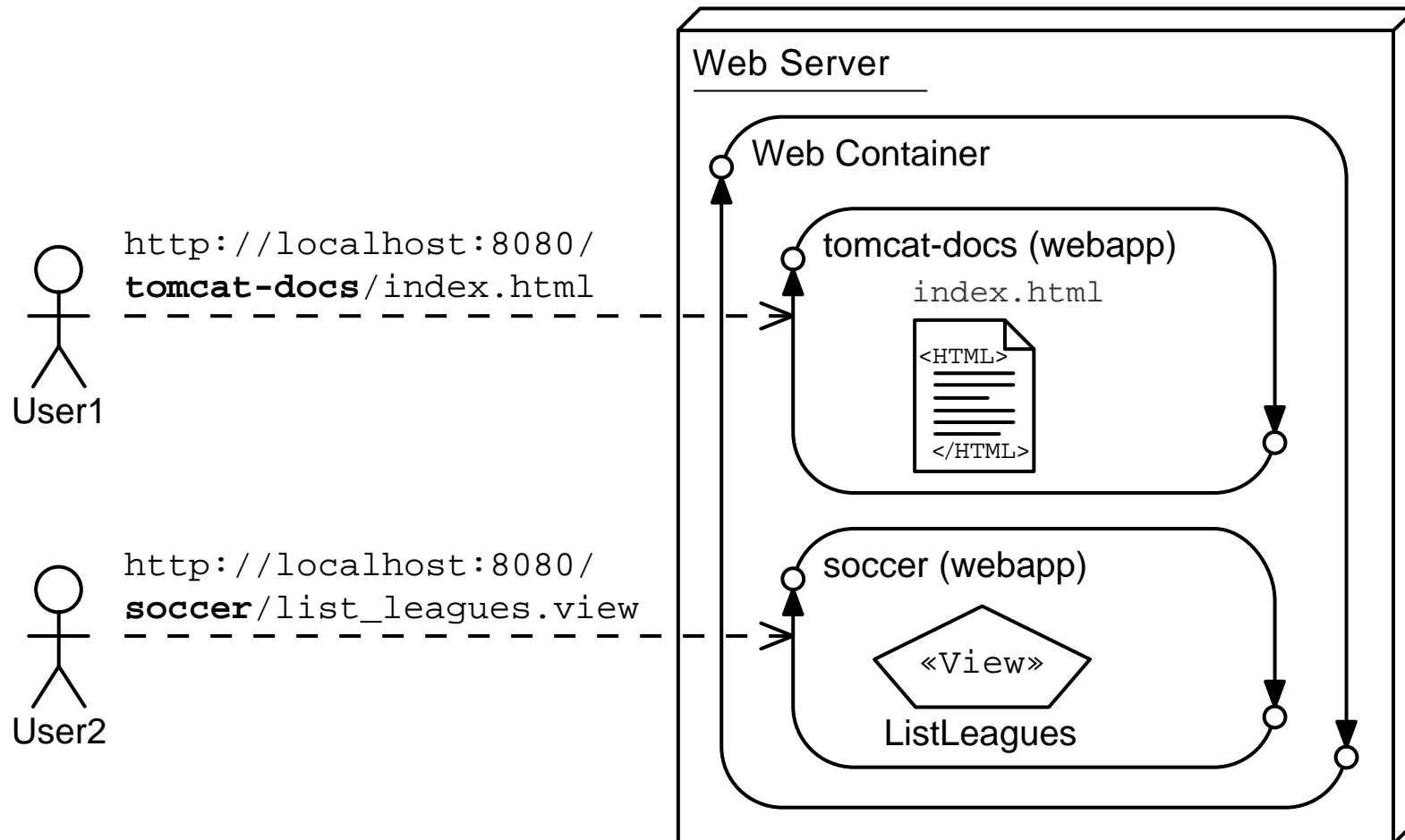


# Complete Deployment Descriptor

```
9
10 <display-name>SL-314 WebApp Example</display-name>
11 <description>
12     This Web Application demonstrates a single View servlet.
13 </description>
14
15 <servlet>
16     <servlet-name>ListLeagues</servlet-name>
17     <servlet-class>sl314.view.ListLeaguesServlet</servlet-class>
18 </servlet>
19
20 <servlet-mapping>
21     <servlet-name>ListLeagues</servlet-name>
22     <url-pattern>/list_leagues.view</url-pattern>
23 </servlet-mapping>
24
25 </web-app>
```

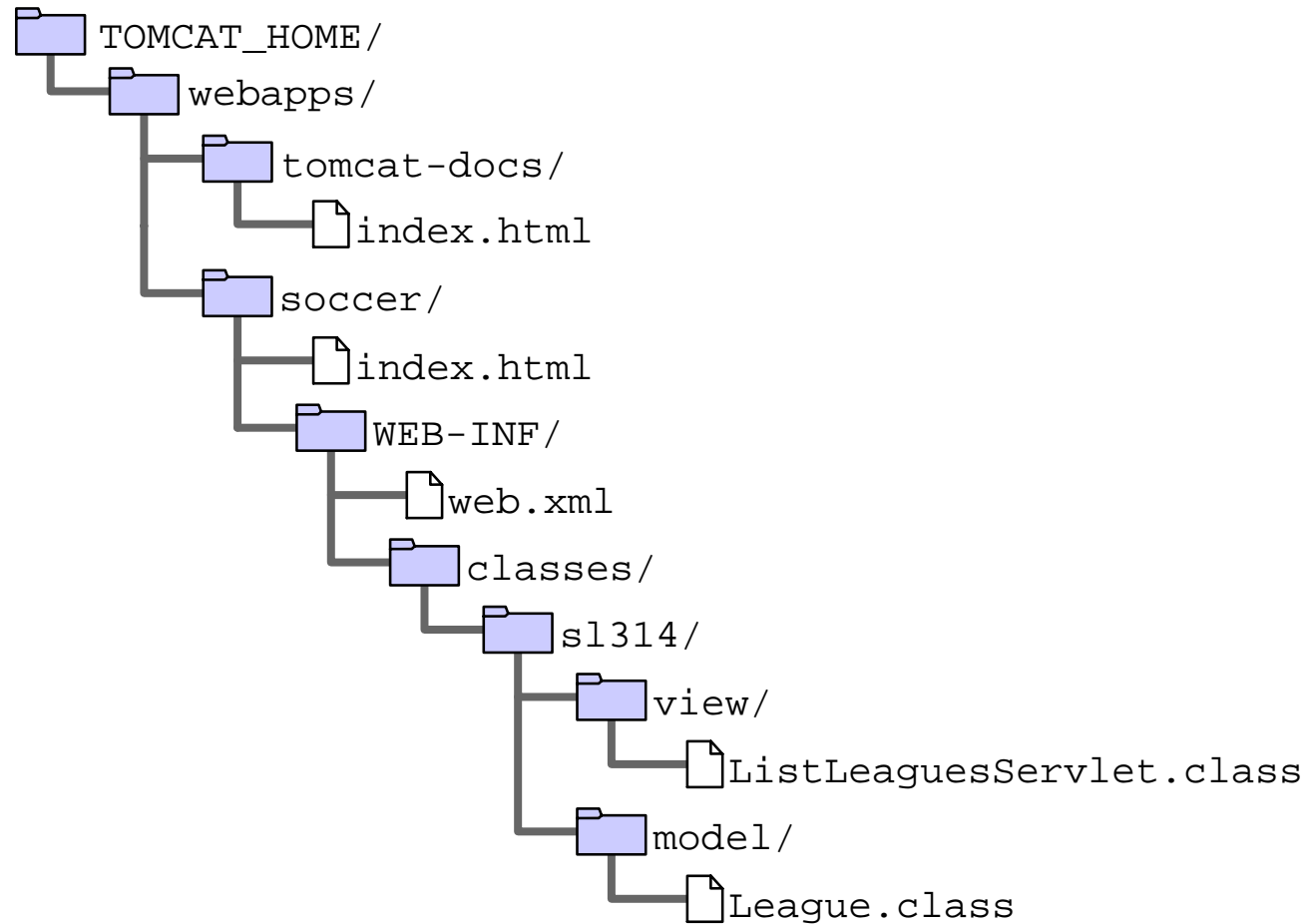


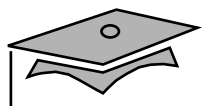
# Web Application Context Root



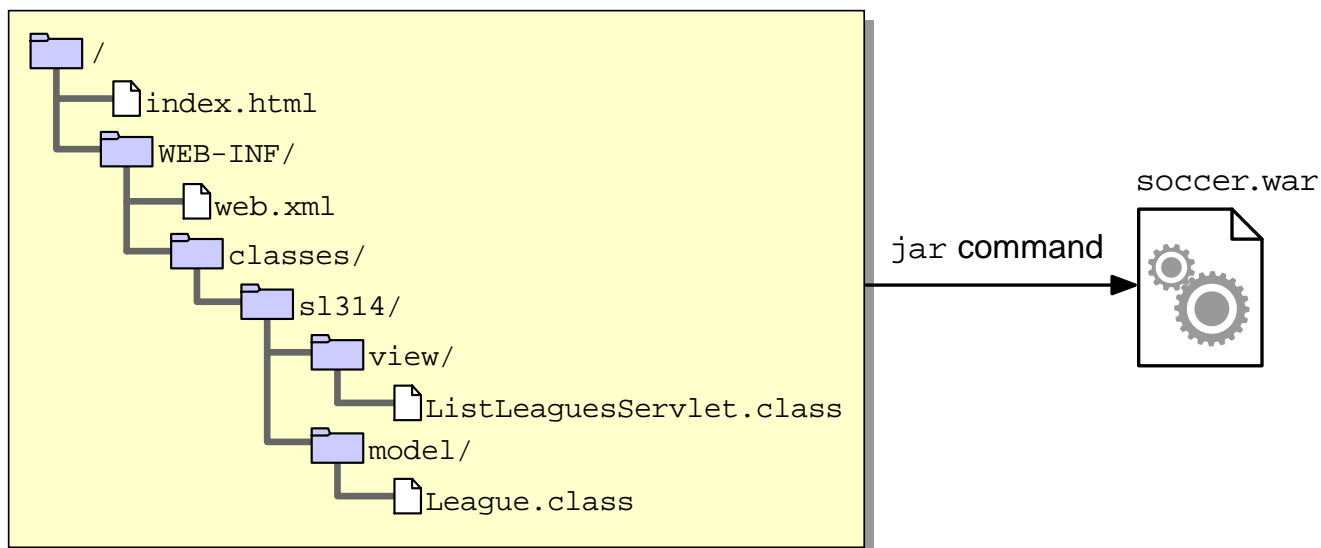


# Tomcat Deployment

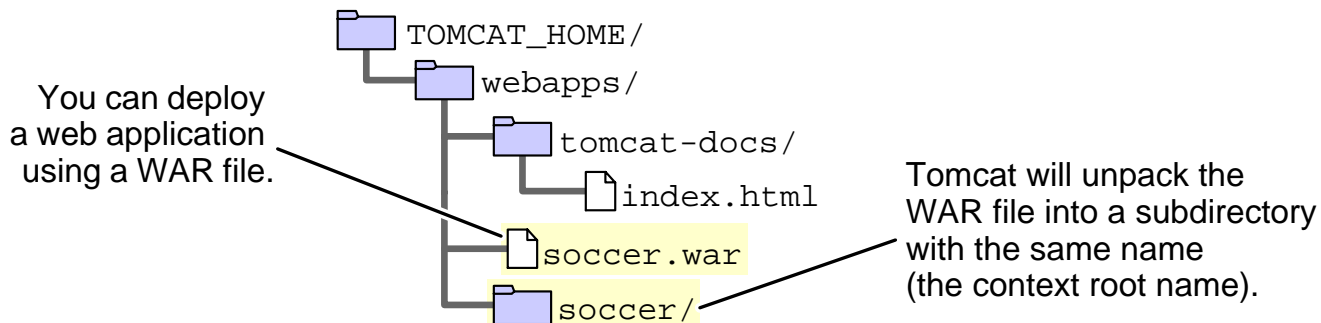




# WAR Files for Deployment



## Tomcat deployment of a WAR file:





## Activating the Servlet in a Web Browser

Request for `http://localhost:8080/soccer/index.html` presents:

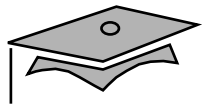


HTML:

```
20 <h3>Players</h3>
21
22 <ul>
23   <li><a href='list_leagues.view'>List all leagues</a></li>
24   <li>Register for a league (TBA)</li>
25 </ul>
```

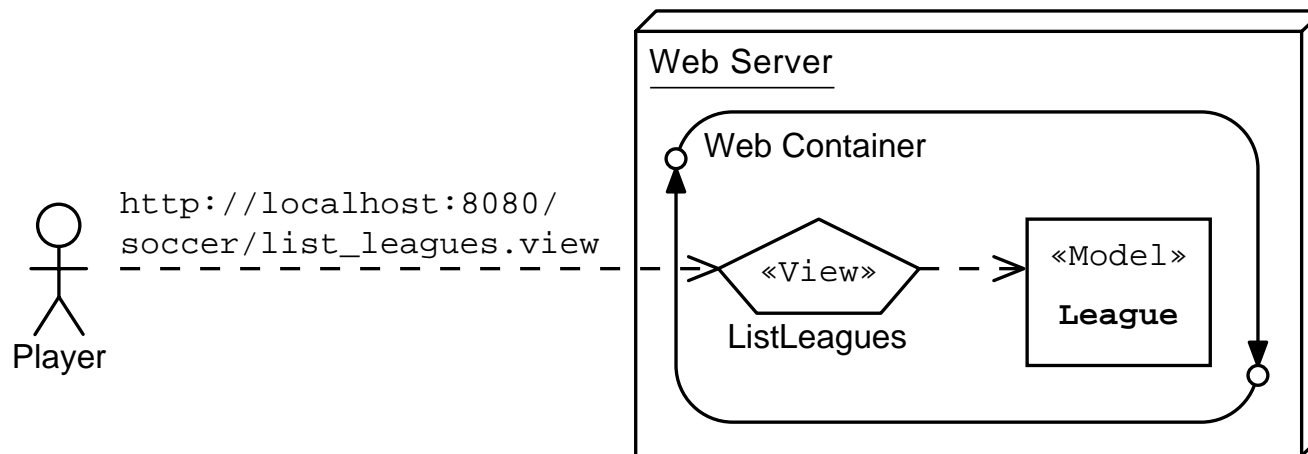
Clicking on List performs a GET request for the URL:  
`http://localhost:8080/soccer/list_leagues.view`





## Activating the ListLeagues View

Request for the `list_league.view` is sent to the container:



This servlet generates this view:

The set of soccer leagues are:

- Soccer League (Spring '03)
- Summer Soccer Fest 2003
- Fall Soccer League (2003)
- Soccer League (Spring '04)
- The Summer of Soccer Love 2004
- Fall Soccer League (2004)



## Summary

- You can use a view component to display data, present a form, present informational messages, and so on.
- The HTTP protocol provides a mechanism to request static or dynamic views.
- The web container intercepts the HTTP request and activates the necessary servlet.
- You can develop a servlet class that implements the `doGet` method to process a request.
- You can access data from the request stream using the request object provided by the web container.
- You can generate a view by writing to the output stream of the request object provided by the container.



# Module 3

## Developing a Controller Component



## Objectives

- Design a controller component
- Create an HTML form
- Describe how HTML form data is sent in the HTTP request
- Develop a controller servlet
- Dispatch from a controller servlet to a view servlet



## Relevance

- What is a controller component?
- What types of controller components are you familiar with?

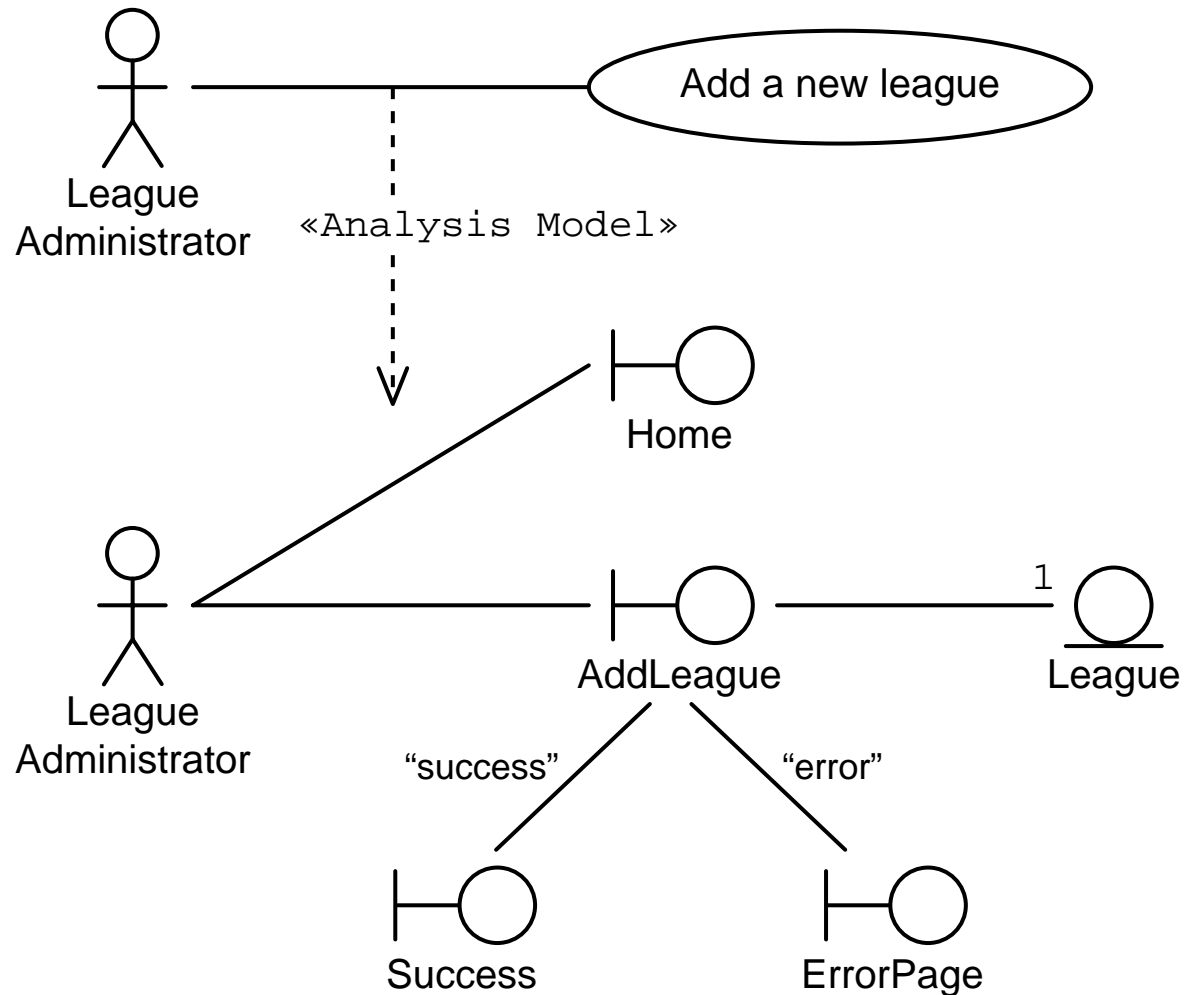


## Types of Controller Components

- Process input from a user
- Screen navigation
- Preparing data for view components

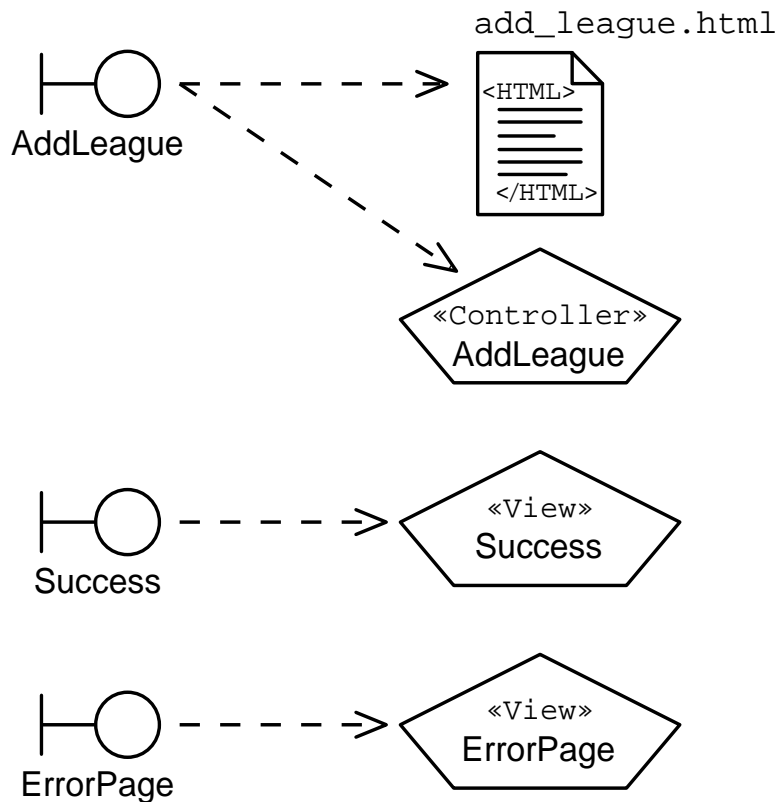


# Add a New League Analysis Model

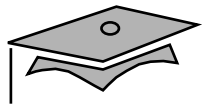




# Add League Boundary Components

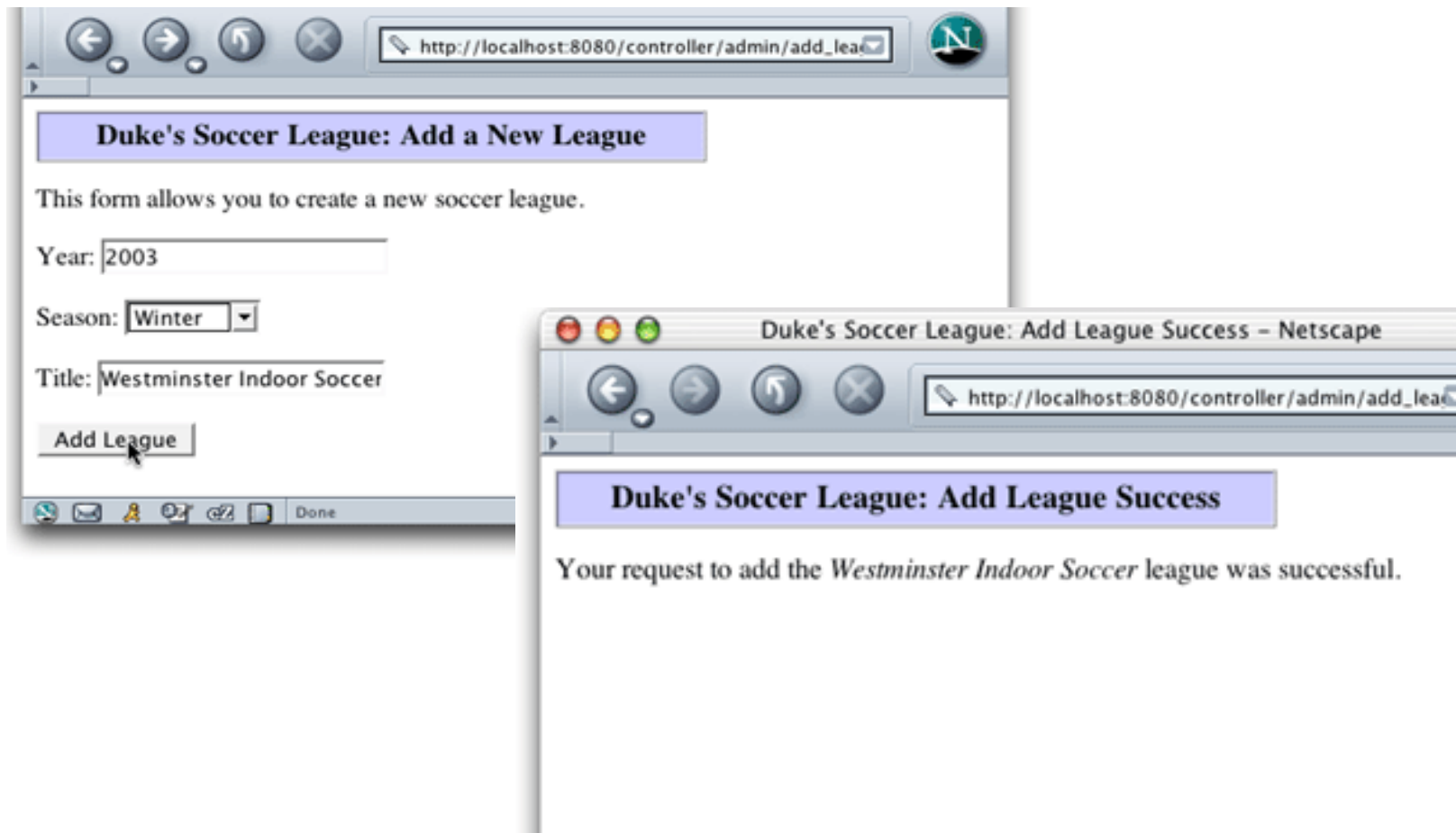






# Add a New League Page Flow

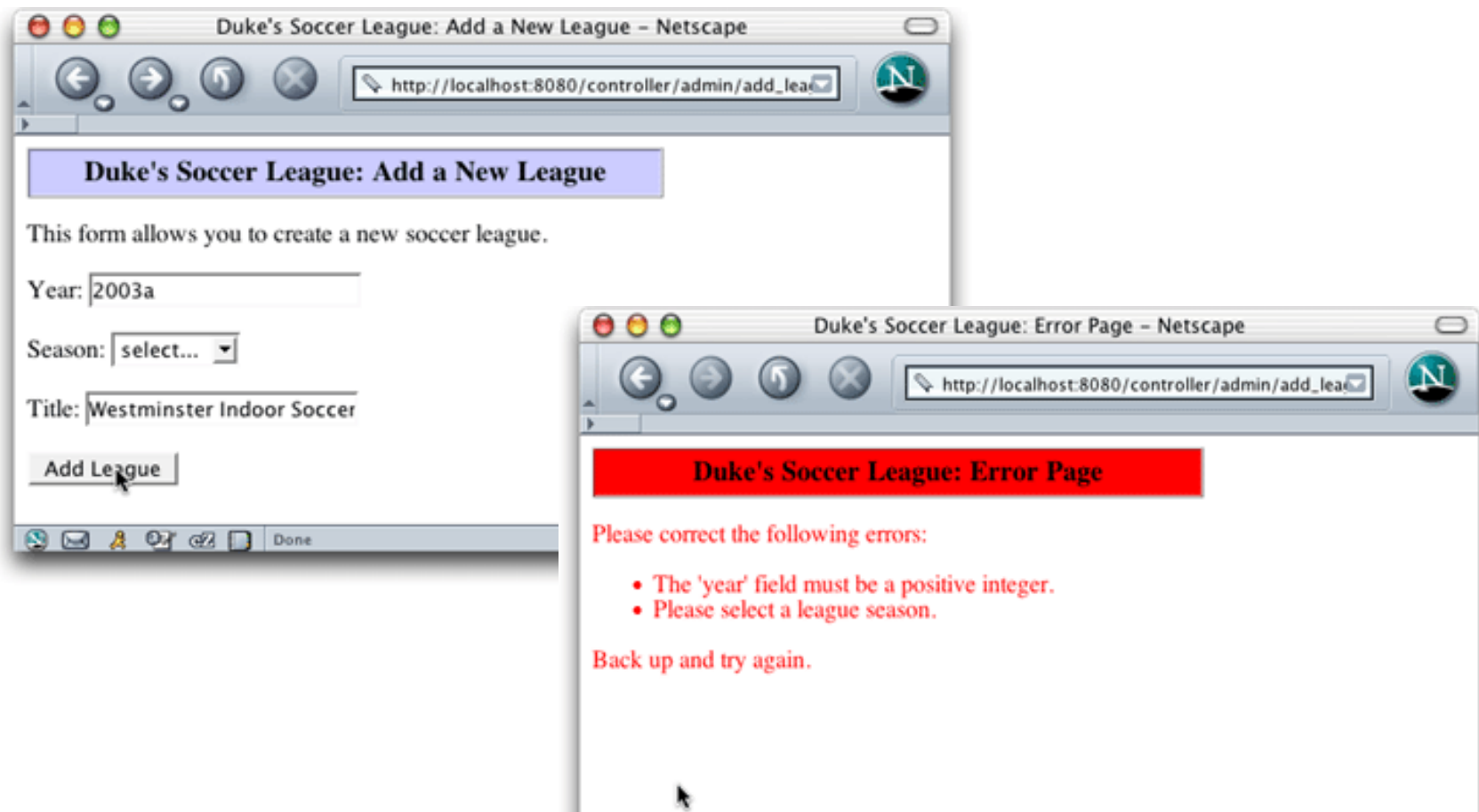
Success path:





# Add a New League Page Flow

Error path:



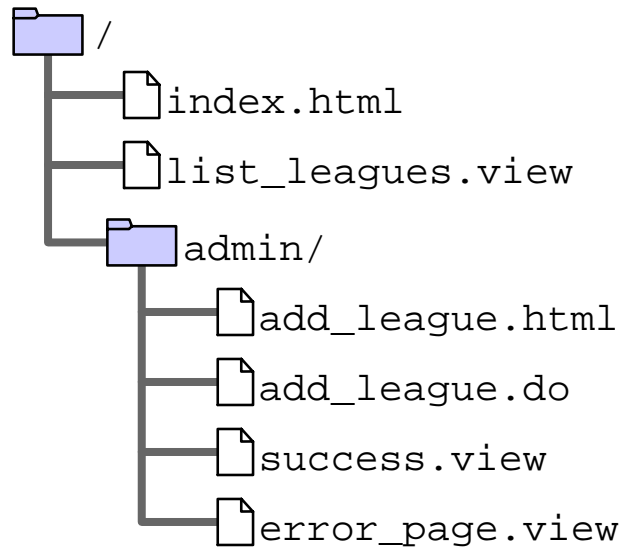


## Form Verification

- What are the drawbacks of using server-side verification?
- What is an alternative to server-side verification?
- What are the drawbacks of using client-side verification?
- What is the solution?

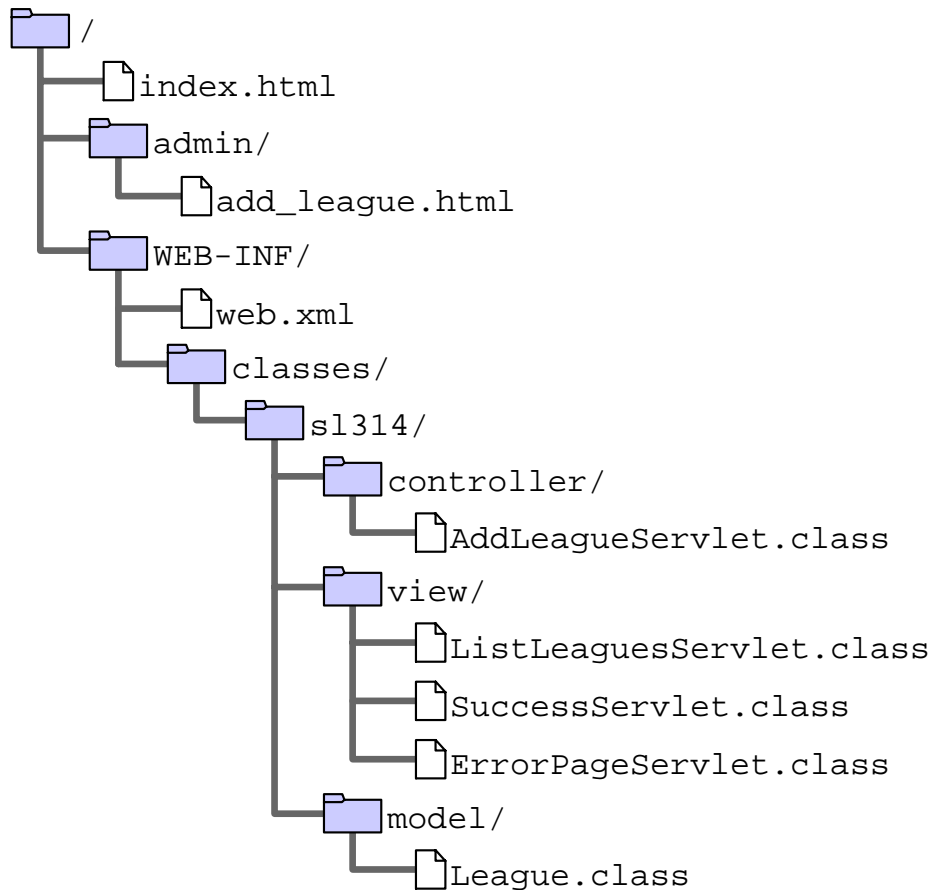


# Soccer League Web Structure





# Soccer League Web Structure





## Objectives

- Design a controller component
- Create an HTML form
- Describe how HTML form data is sent in the HTTP request
- Develop a controller servlet
- Dispatch from a controller servlet to a view servlet



# Creating an HTML Form

The screenshot shows a Netscape browser window titled "Duke's Soccer League: Add a New League - Netscape". The address bar contains the URL "http://localhost:8080/controller/admin/add\_lea...". The main content area features a purple header bar with the text "Duke's Soccer League: Add a New League". Below the header, a paragraph reads "This form allows you to create a new soccer league." The form consists of three input fields: "Year:" with the value "2003", "Season:" with a dropdown menu showing "Winter", and "Title:" with the value "Westminster Indoor Soccer". At the bottom of the form is a button labeled "Add League". The browser's status bar at the bottom shows "Done" and various navigation icons.



## The FORM Tag

The following is a partial structure of an HTML form:

```
<form action='URL TO CONTROLLER' method='GET or POST' >
<!-- PUT FORM COMPONENT TAGS HERE -->
</form>
```

For example:

```
<form action='add_league.do' method='POST' >
Year: [textfield tag]
Season: [drop-down list tag]
Title: [textfield tag]
[submit button tag]
</form>
```

A single web page can contain many forms.





## Textfield Component

In Netscape, a textfield component looks like this:

This form allows you to create a new soccer league.

Year:

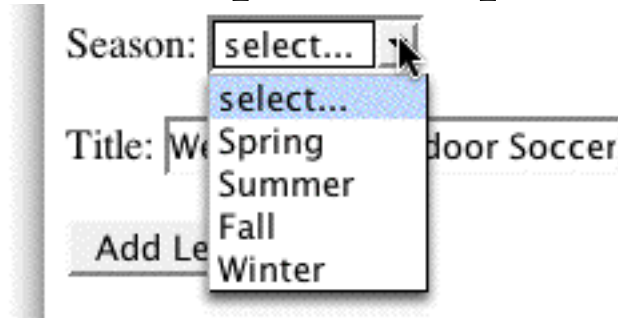
The HTML content for this component is:

```
16 <p>
17 This form allows you to create a new soccer league.
18 </p>
19
20 <form action='add_league.do' method='POST'>
21 Year: <input type='text' name='year' /> <br/><br/>
```



## Drop-Down List Component

In Netscape, a drop-down list component looks like this:



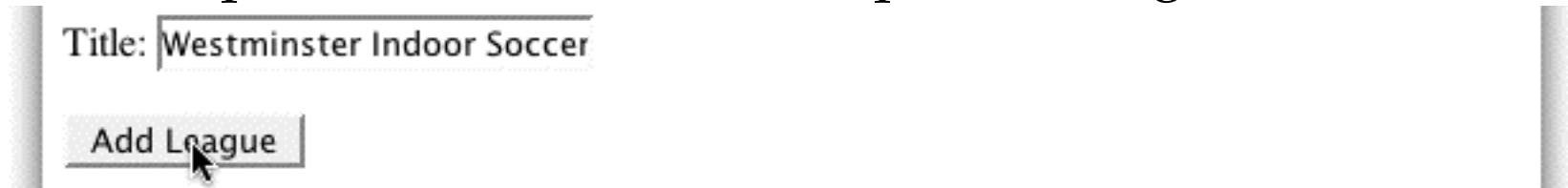
The HTML content for this component is:

```
22 Season: <select name='season'>
23     <option value='UNKNOWN'>select...</option>
24     <option value='Spring'>Spring</option>
25     <option value='Summer'>Summer</option>
26     <option value='Fall'>Fall</option>
27     <option value='Winter'>Winter</option>
28 </select> <br/><br/>
```



## Submit Button

In Netscape, a submit button component might look like this:



The HTML content for this component is:

```
29 Title: <input type='text' name='title' /> <br/><br/>
30 <input type='submit' value='Add League' />
31 </form>
```



## Complete Add a New League Form

```
16 <p>
17 This form allows you to create a new soccer league.
18 </p>
19
20 <form action='add_league.do' method='POST'>
21 Year: <input type='text' name='year' /> <br/><br/>
22 Season: <select name='season'>
23     <option value='UNKNOWN'>select...</option>
24     <option value='Spring'>Spring</option>
25     <option value='Summer'>Summer</option>
26     <option value='Fall'>Fall</option>
27     <option value='Winter'>Winter</option>
28 </select> <br/><br/>
29 Title: <input type='text' name='title' /> <br/><br/>
30 <input type='submit' value='Add League' />
31 </form>
```



## Objectives

- Design a controller component
- Create an HTML form
- Describe how HTML form data is sent in the HTTP request
- Develop a controller servlet
- Dispatch from a controller servlet to a view servlet



## 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:

`username=Fred&password=C1r5z`

`season=Winter&year=2004&title=Westminster+Indoor+Soccer+(2004)`



# HTTP GET Method Request

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

```
GET /admin/add_league.do?year=2003&season=Winter&title=Westminster+Indoor-
HTTP/1.1
Host: localhost:8080
User-Agent: Mozilla/5.0 (Macintosh; U; PPC Mac OS X Mach-O; en-US; rv:1.4.
20030624 Netscape/7.1
Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,te
plain;q=0.8,video/x-mng,image/png,image/jpeg,image/gif;q=0.2,*/*;q=0.1
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
```



# HTTP POST Method Request

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

```
POST /admin/add_league.do HTTP/1.1
Host: localhost:8080
User-Agent: Mozilla/5.0 (Macintosh; U; PPC Mac OS X Mach-O; en-US; rv:1.4)
20030624 Netscape/7.1
Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text
plain;q=0.8,video/x-mng,image/png,image/jpeg,image/gif;q=0.2,*/*;q=0.1
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Referer: http://localhost:8080/controller/admin/add_league.html
Content-Type: application/x-www-form-urlencoded
Content-Length: 55
```

```
year=2003&season=Winter&title=Westminster+Indoor+Soccer
```





## HTTP GET and POST Methods

The HTTP GET method is used when:

- The processing of the request is idempotent.
- 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).



## Objectives

- Design a controller component
- Create an HTML form
- Describe how HTML form data is sent in the HTTP request
- **Develop a controller servlet**
- Dispatch from a controller servlet to a view servlet



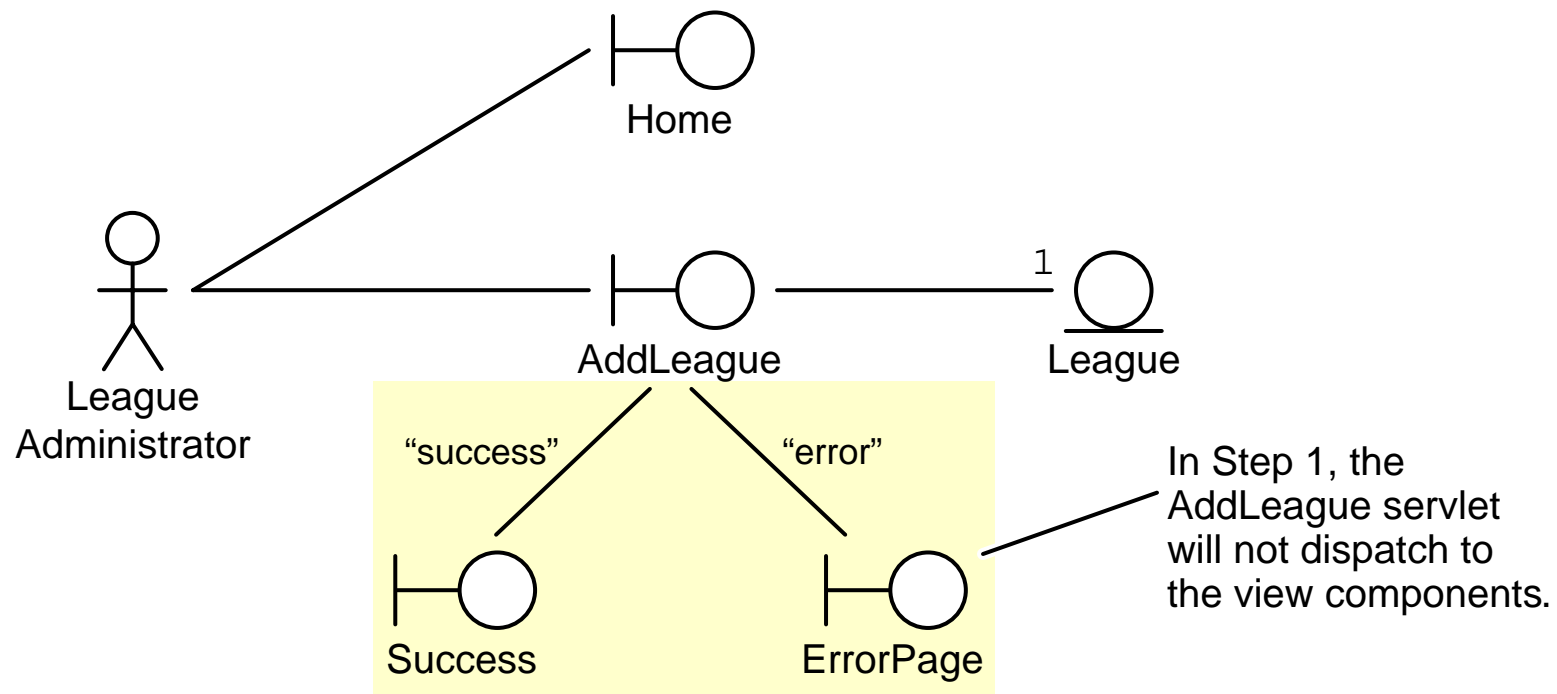
## Developing a Controller Servlet

A form-processing (controller) servlet needs to:

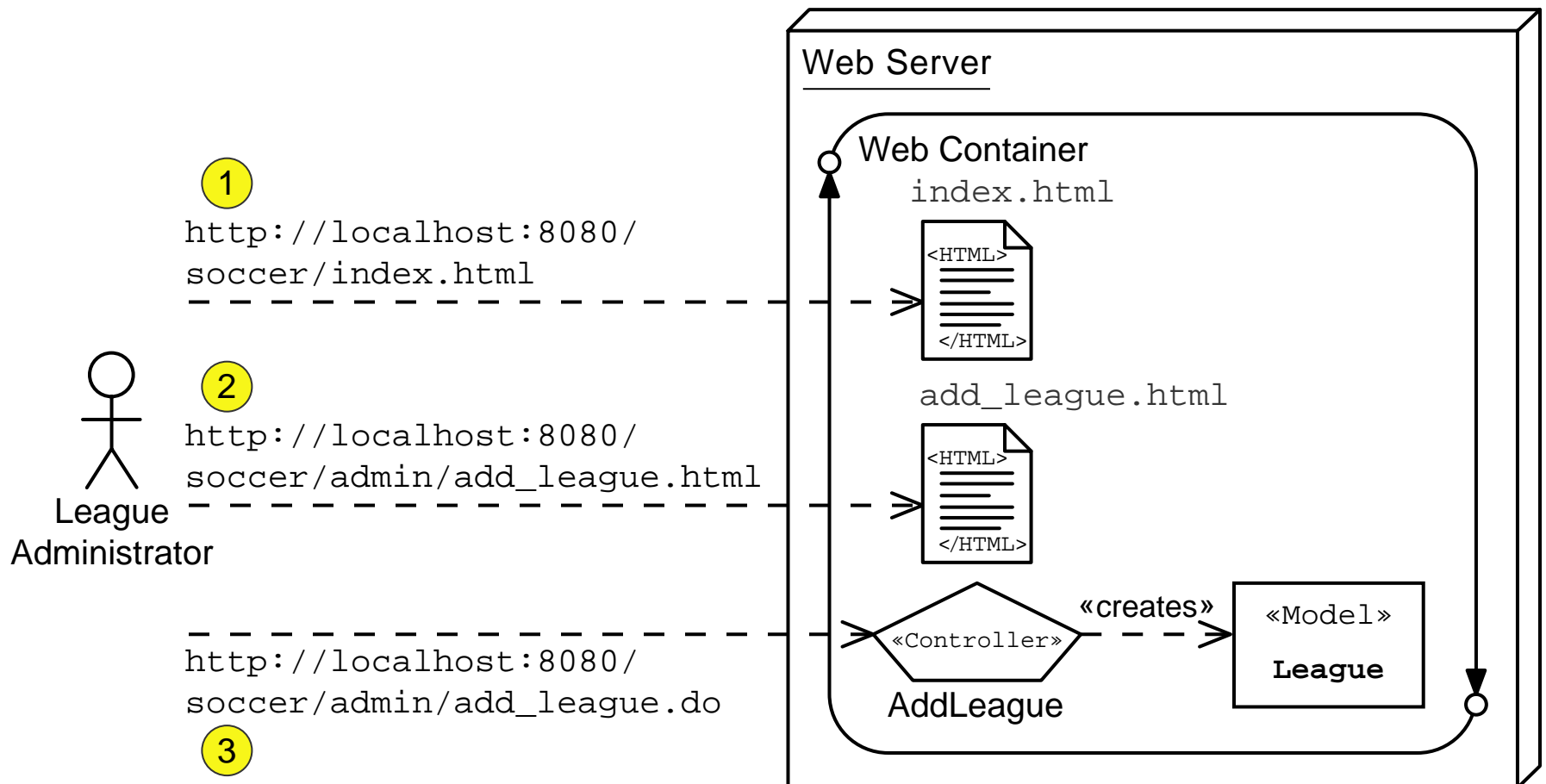
1. Retrieve form parameters from the HTTP request.
2. Perform any data conversion on the form parameters.
3. Verify the form parameters.
4. Execute the business logic.
5. Dispatch to the next view component based on the results of the previous steps.

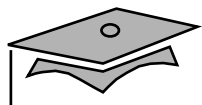


# Add League Analysis Model (Stage 1)

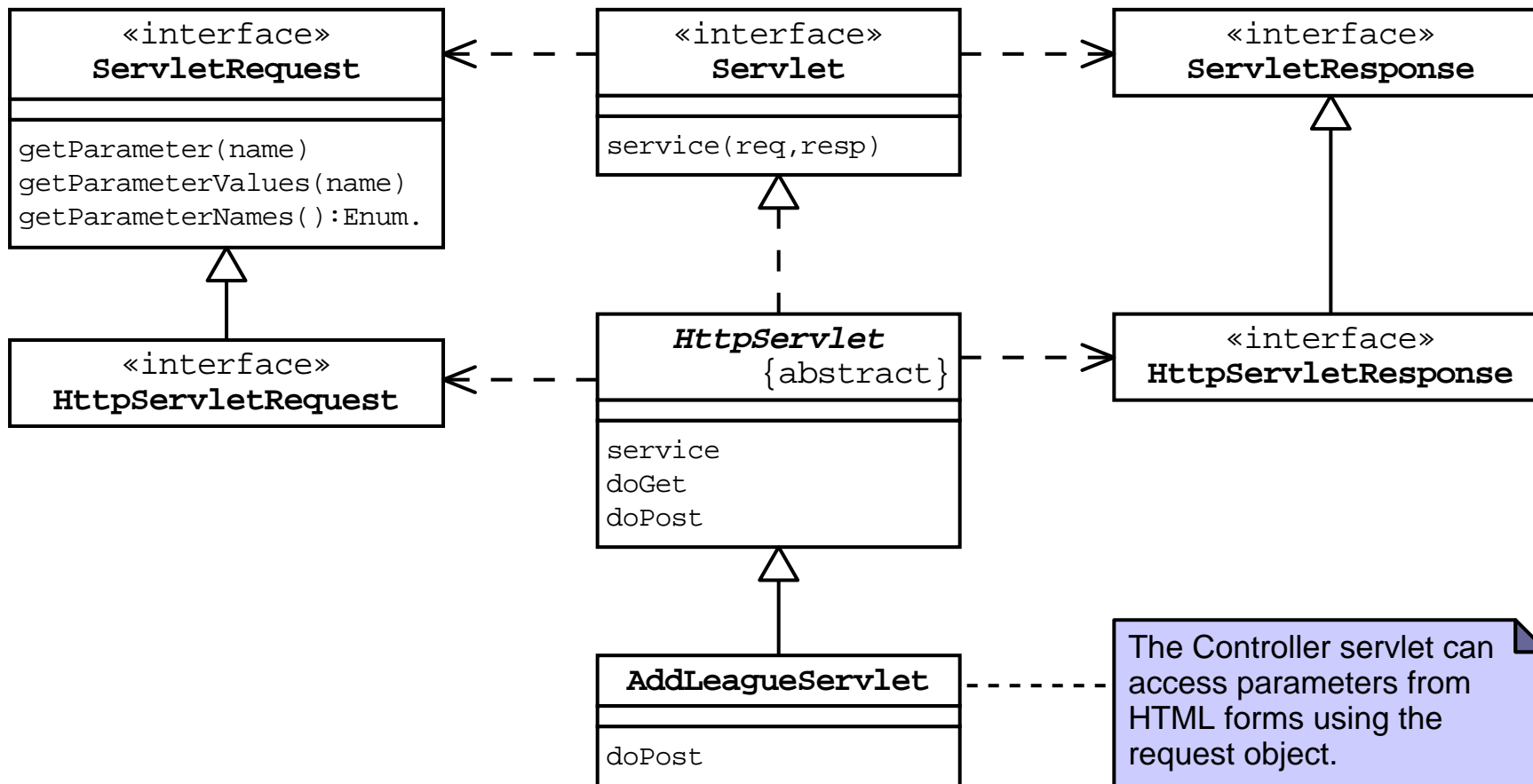


# Add League Architecture Model (Stage 1)





# Servlet API to Retrieve Form Parameters





## The AddLeagueServlet Class Declaration

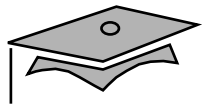
```
1  package sl314.controller;
2
3  import javax.servlet.http.HttpServlet;
4  import javax.servlet.http.HttpServletRequest;
5  import javax.servlet.http.HttpServletResponse;
6  import javax.servlet.ServletException;
7  // Support classes
8  import java.io.IOException;
9  import java.io.PrintWriter;
10 // Model classes
11 import sl314.model.League;
12 import java.util.List;
13 import java.util.LinkedList;
14
15 public class AddLeagueServlet extends HttpServlet {
16     public void doPost(HttpServletRequest request,
17                       HttpServletResponse response)
18         throws IOException, ServletException {
19
20         // Keep a set of strings to record form processing errors.
21         List errorMsgs = new LinkedList();
```



## Retrieving Form Parameters and Data Conversion

```
22
23     try {
24
25         // Retrieve form parameters.
26         String yearStr = request.getParameter("year").trim();
27         String season = request.getParameter("season").trim();
28         String title = request.getParameter("title").trim();
29
30         // Perform data conversions.
31         int year = -1;
32         try {
33             year = Integer.parseInt(yearStr);
34         } catch (NumberFormatException nfe) {
35             errorMsgs.add("The 'year' field must be a positive integer.");
36         }
37
```





## Performing Form Parameters

```
37
38     // Verify form parameters
39     if ( (year != -1) && ((year < 2000) || (year > 2010)) ) {
40         errorMsgs.add("The 'year' field must within 2000 to 2010.");
41     }
42     if ( season.equals("UNKNOWN") ) {
43         errorMsgs.add("Please select a league season.");
44     }
45     if ( title.length() == 0 ) {
46         errorMsgs.add("Please enter the title of the league.");
47     }
48
49     // Send the ErrorPage view if there were errors
50     if ( ! errorMsgs.isEmpty() ) {
51         // dispatch to the ErrorPage
52         PrintWriter out = response.getWriter();
53         out.println("ERROR PAGE");
54         return;
55     }
56
```



## Performing the Business Logic

```
57
58     // Perform business logic
59     League league = new League(year, season, title);
60
61     // Send the Success view
62     PrintWriter out = response.getWriter();
63     out.println("SUCCESS");
64     return;
65
```



## Handling an Exception

```
65
66     // Handle any unexpected exceptions
67     } catch (RuntimeException e) {
68         errorMsgs.add(e.getMessage());
69         // dispatch to the ErrorPage
70         PrintWriter out = response.getWriter();
71         out.println("ERROR PAGE");
72
73         // Log stack trace
74         e.printStackTrace(System.err);
75
76     } // END of try-catch block
77 } // END of doPost method
78 } // END of AddLeagueServlet class
```

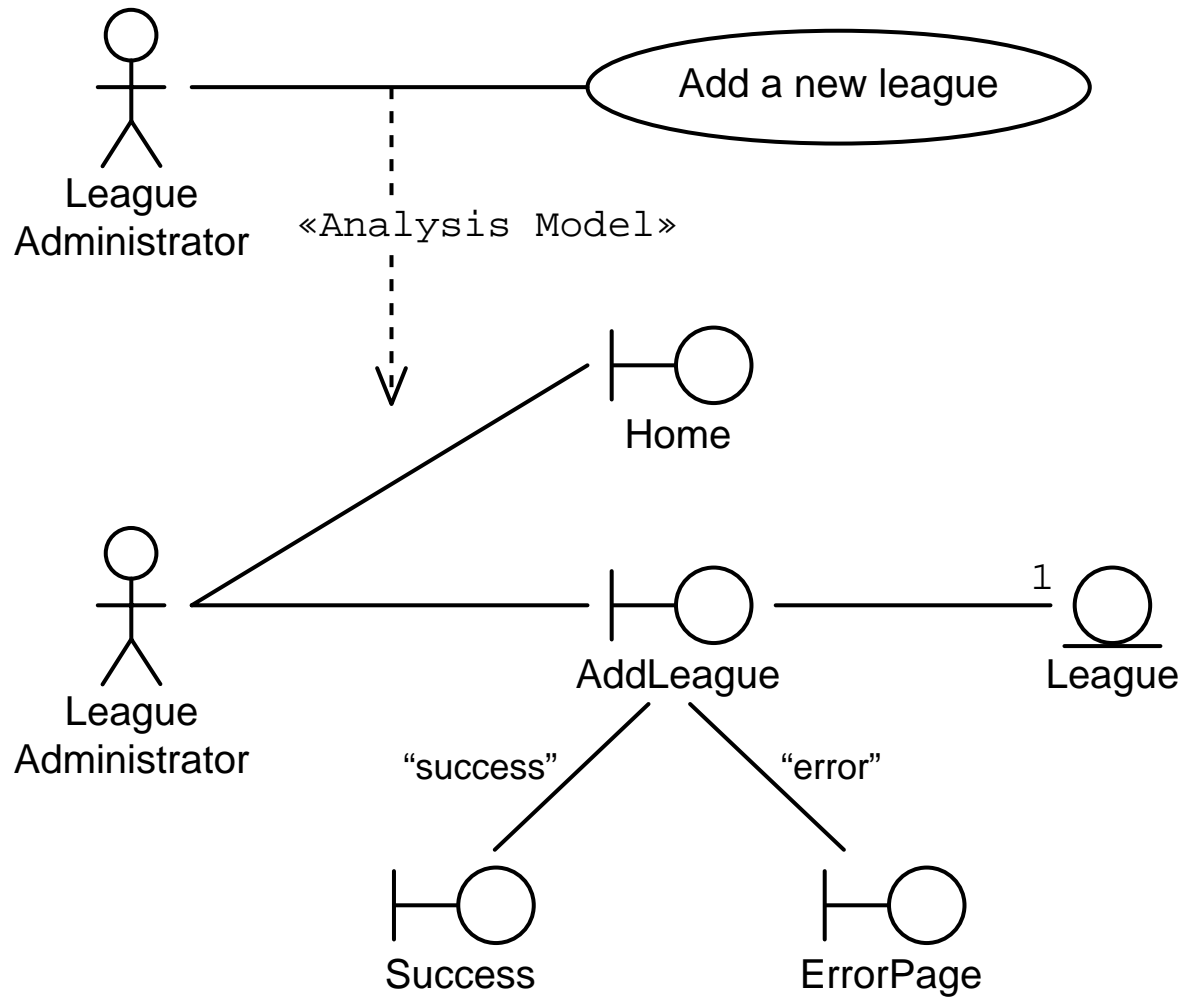


## Objectives

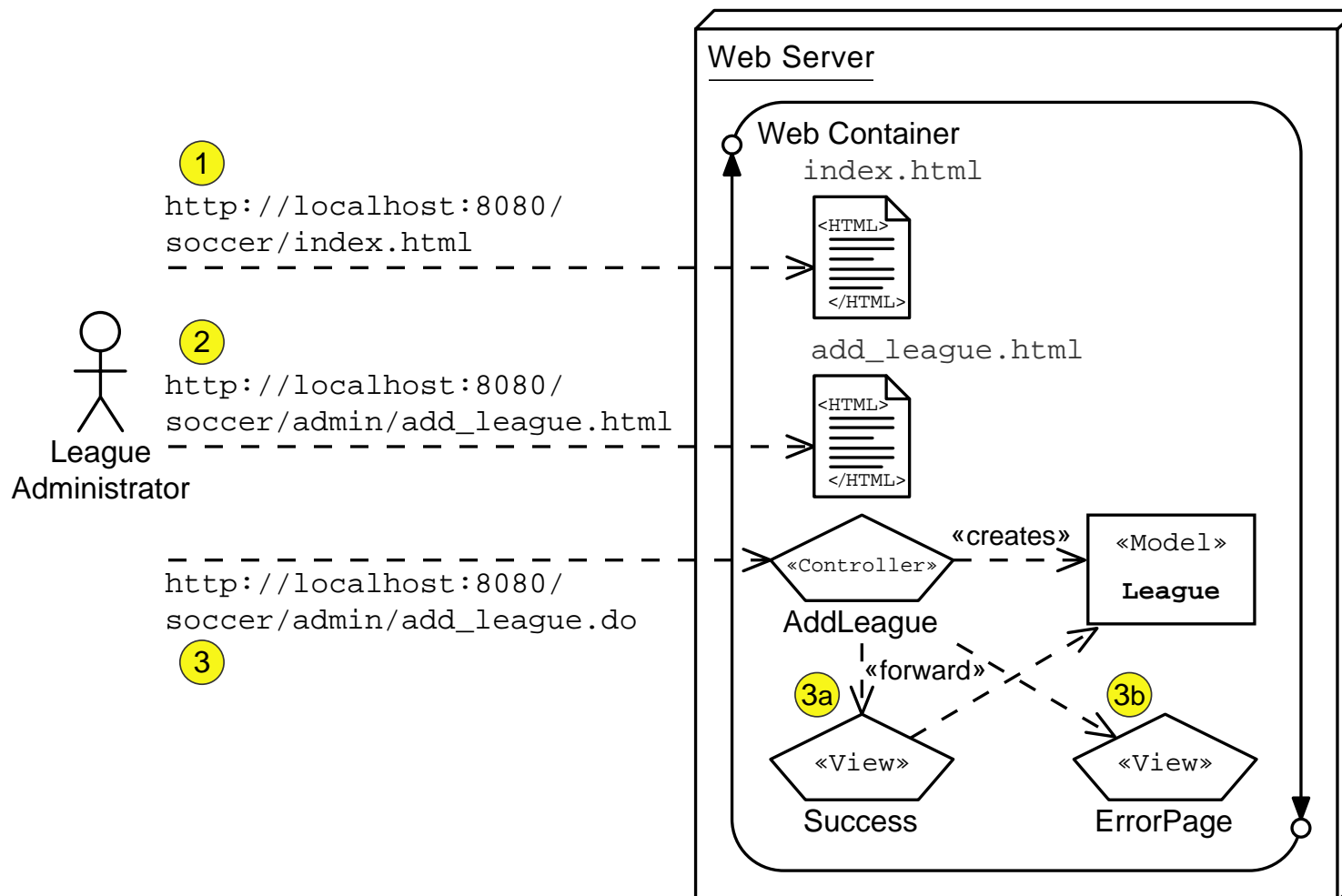
- Design a controller component
- Create an HTML form
- Describe how HTML form data is sent in the HTTP request
- Develop a controller servlet
- Dispatch from a controller servlet to a view servlet



## Add League Analysis Model (Stage 2)

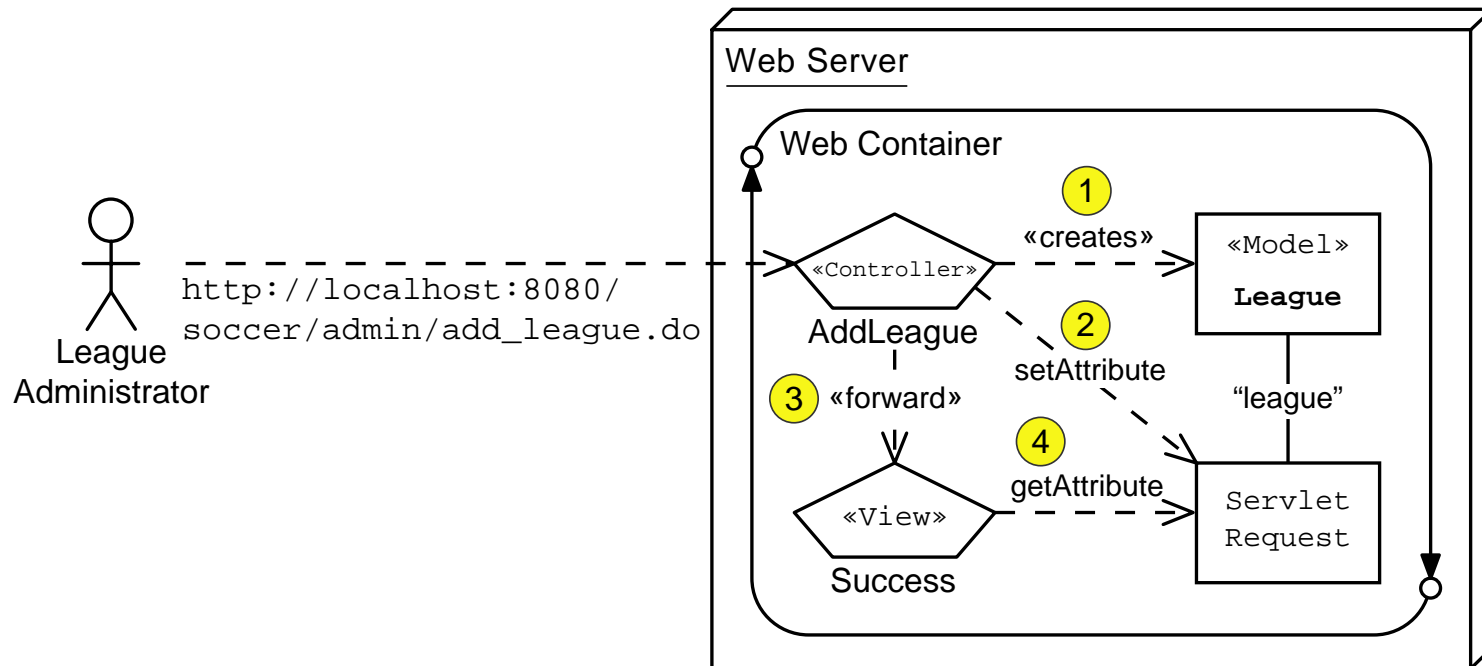


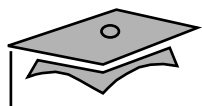
## Add League Architecture Model (Stage 2)





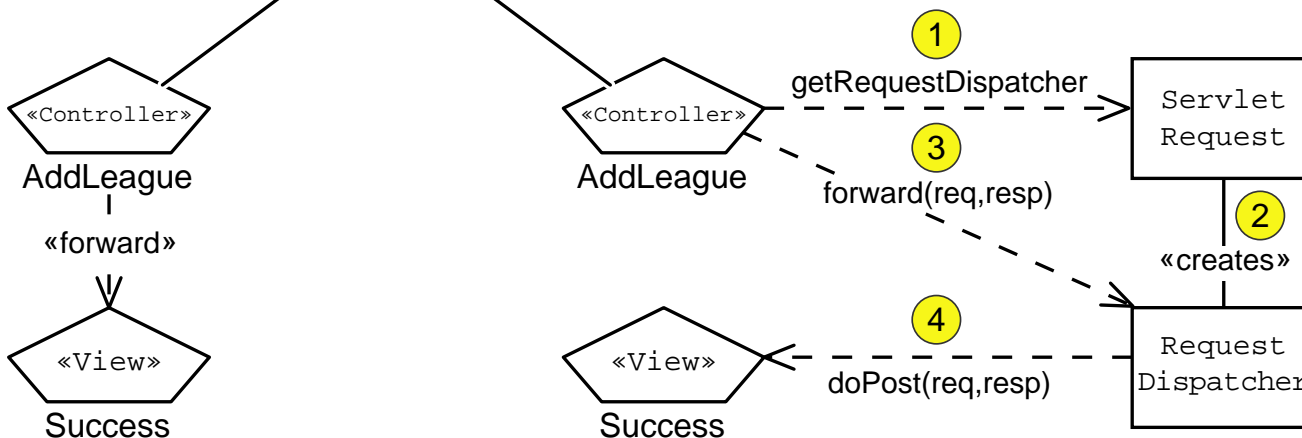
# Request Scope





# Using a Request Dispatcher

You must use a request dispatcher provided by the web container to forward the request to a view component.







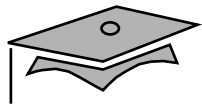
## Developing the AddLeagueServlet Code

```
6  import javax.servlet.RequestDispatcher;
7  import javax.servlet.ServletException;
8  // Support classes
9  import java.io.IOException;
10 import java.io.PrintWriter;
11 // Model classes
12 import sl314.model.League;
13 import java.util.List;
14 import java.util.LinkedList;
15
16 public class AddLeagueServlet extends HttpServlet {
17     public void doPost(HttpServletRequest request,
18                       HttpServletResponse response)
19         throws IOException, ServletException {
20
21         // Keep a set of strings to record form processing errors.
22         List errorMsgs = new LinkedList();
23         // Store this set in the request scope, in case we need to
24         // send the ErrorPage view.
25         request.setAttribute("errorMsgs", errorMsgs);
26
```



## Developing the AddLeagueServlet Code

```
21     // Keep a set of strings to record form processing errors.
22     List errorMsgs = new LinkedList();
23     // Store this set in the request scope, in case we need to
24     // send the ErrorPage view.
25     request.setAttribute("errorMsgs", errorMsgs);
26
27     try {
28
29         // Retrieve form parameters.
30         String yearStr = request.getParameter("year").trim();
31         String season = request.getParameter("season").trim();
32         String title = request.getParameter("title").trim();
33
34         // Perform data conversions.
35         int year = -1;
36         try {
37             year = Integer.parseInt(yearStr);
38         } catch (NumberFormatException nfe) {
39             errorMsgs.add("The 'year' field must be a positive integer.")
40         }
41
```



## Developing the AddLeagueServlet Code

```
41
42     // Verify form parameters
43     if ( (year != -1) && ((year < 2000) || (year > 2010)) ) {
44         errorMsgs.add("The 'year' field must within 2000 to 2010.");
45     }
46     if ( season.equals("UNKNOWN") ) {
47         errorMsgs.add("Please select a league season.");
48     }
49     if ( title.length() == 0 ) {
50         errorMsgs.add("Please enter the title of the league.");
51     }
52
53     // Send the ErrorPage view if there were errors
54     if ( ! errorMsgs.isEmpty() ) {
55         RequestDispatcher view
56             = request.getRequestDispatcher("error_page.view");
57         view.forward(request, response);
58         return;
59     }
60
```



## Developing the AddLeagueServlet Code

```
60
61     // Perform business logic
62     League league = new League(year, season, title);
63     // Store the new league in the request-scope
64     request.setAttribute("league", league);
65
66     // Send the Success view
67     RequestDispatcher view
68     = request.getRequestDispatcher("success.view");
69     view.forward(request, response);
70     return;
71
72     // Handle any unexpected exceptions
73     } catch (RuntimeException e) {
74         errorMsgs.add(e.getMessage());
75         RequestDispatcher view
76         = request.getRequestDispatcher("error_page.view");
77         view.forward(request, response);
78
79         // Log stack trace
80         e.printStackTrace(System.err);
```



## The SuccessServlet Code

```
11
12 public class SuccessServlet extends HttpServlet {
13
14     public void doGet(HttpServletRequest request,
15                       HttpServletResponse response)
16         throws IOException {
17         generateView(request, response);
18     }
19
20     public void doPost(HttpServletRequest request,
21                       HttpServletResponse response)
22         throws IOException {
23         generateView(request, response);
24     }
25
26     public void generateView(HttpServletRequest request,
27                              HttpServletResponse response)
28         throws IOException {
29
```



## The SuccessServlet Code (Part 2)

```
25
26 public void generateView(HttpServletRequest request,
27                          HttpServletResponse response)
28     throws IOException {
29
30     // Set page title
31     String pageTitle = "Duke's Soccer League: Add League Success";
32
33     // Retrieve the 'league' from the request-scope
34     League league = (League) request.getAttribute("league");
35
36     // Specify the content type is HTML
37     response.setContentType("text/html");
38     PrintWriter out = response.getWriter();
39
40     // Generate the HTML response
41     out.println("<html>");
```



## The SuccessServlet Code (Part 3)

```
54
55     // Generate main body
56     out.println("<p>");
57     out.print("Your request to add the ");
58     out.print("<i>" + league.getTitle() + "</i>");
59     out.println(" league was successful.");
60     out.println("</p>");
61
62     out.println("</body>");
63     out.println("</html>");
64
65 } // END of generateResponse method
66
67 } // END of SuccessServlet class
```



## Summary

- You can use a controller component to process forms, manage screen navigation, prepare data for views, and so on.
- You can create web forms using the HTML form tags.
- Usually, you should use the POST HTTP method to send form data to your servlets.
- You can access form data on the request stream using the `getParameter` method on the request object.
- You can use the request scope to communicate from a controller to a view component.
- You can use a `RequestDispatcher` object to forward the request from the controller to the view component.





# Module 4

## Developing Dynamic Forms



# Objectives

- Describe the servlet life cycle
- Customize a servlet with initialization parameters
- Explain error reporting within the web form
- Repopulating the web form

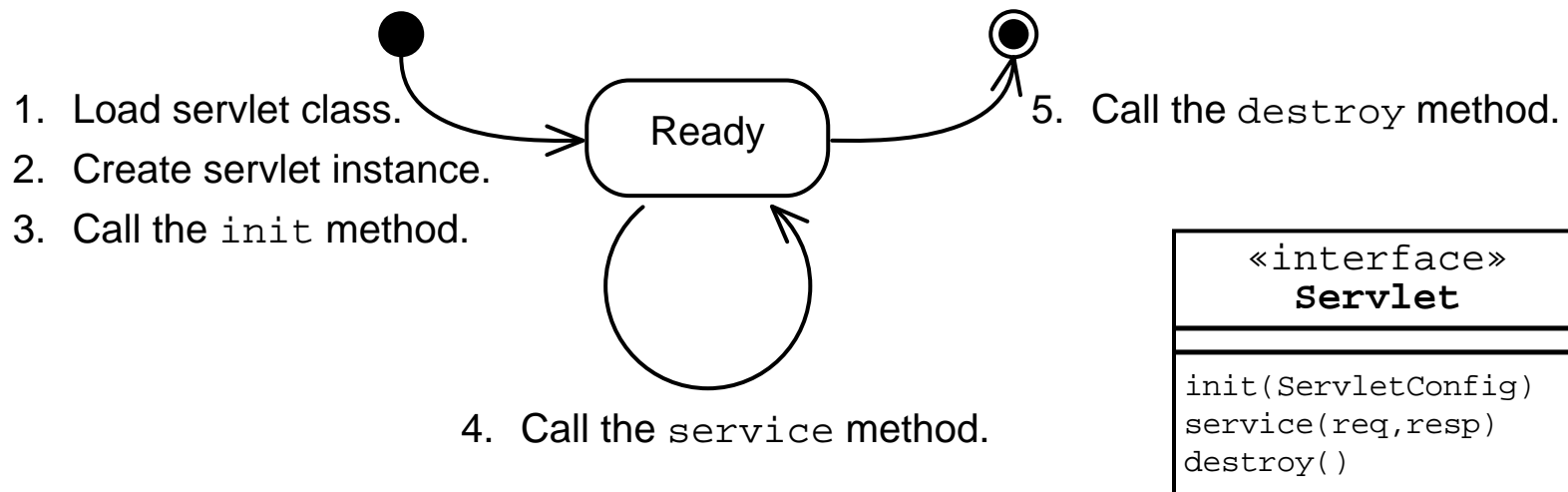


## Relevance

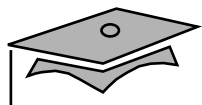
- What is a dynamic form?
- What elements of a form can be customized?
- How can a form report any processing errors?
- Have you ever seen a form that re-populated or pre-populated the form fields?



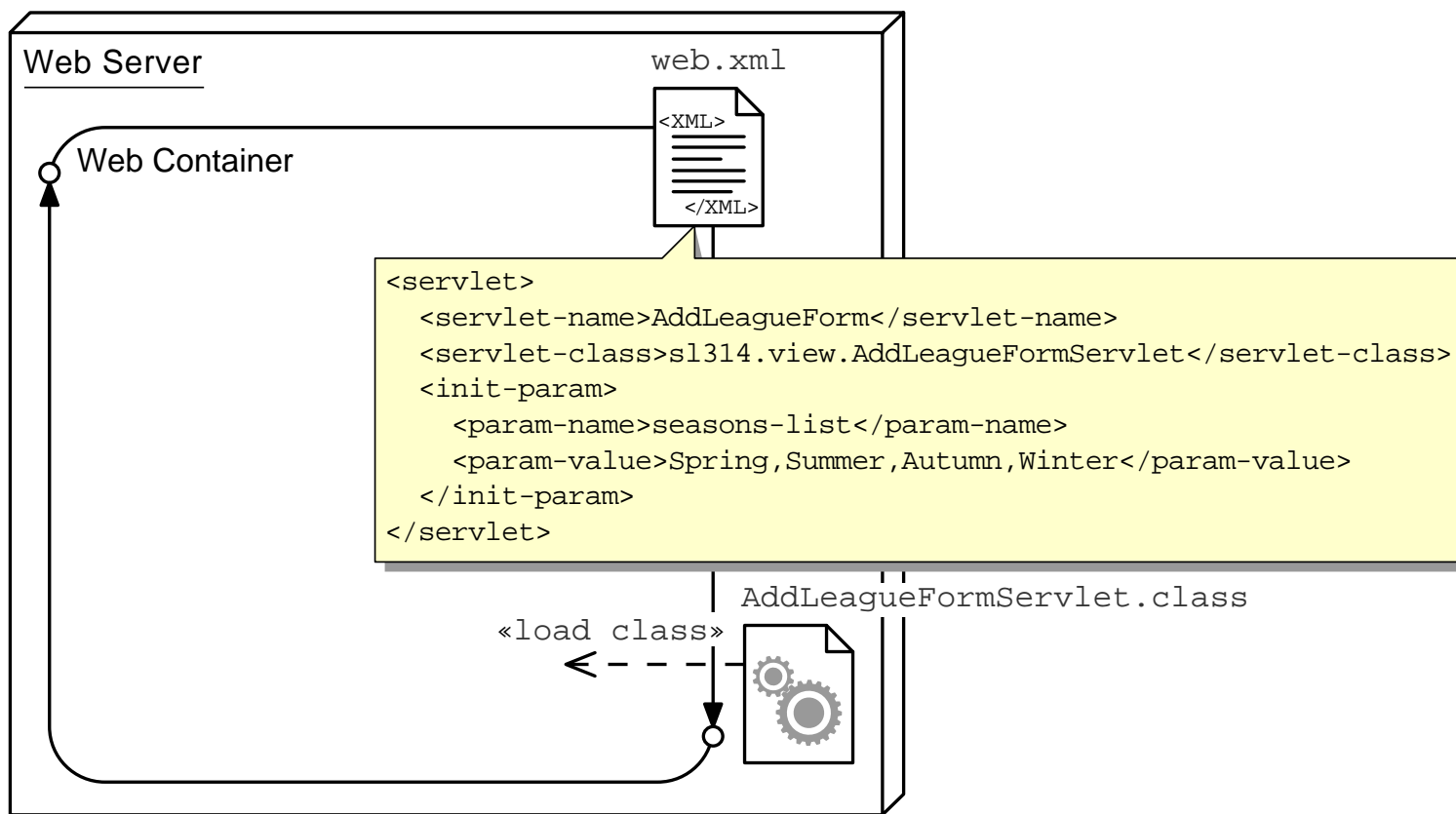
# Servlet Life Cycle Overview



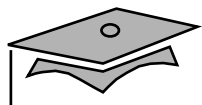
The web container manages the life cycle of a servlet instance. These methods should not be called by your code.



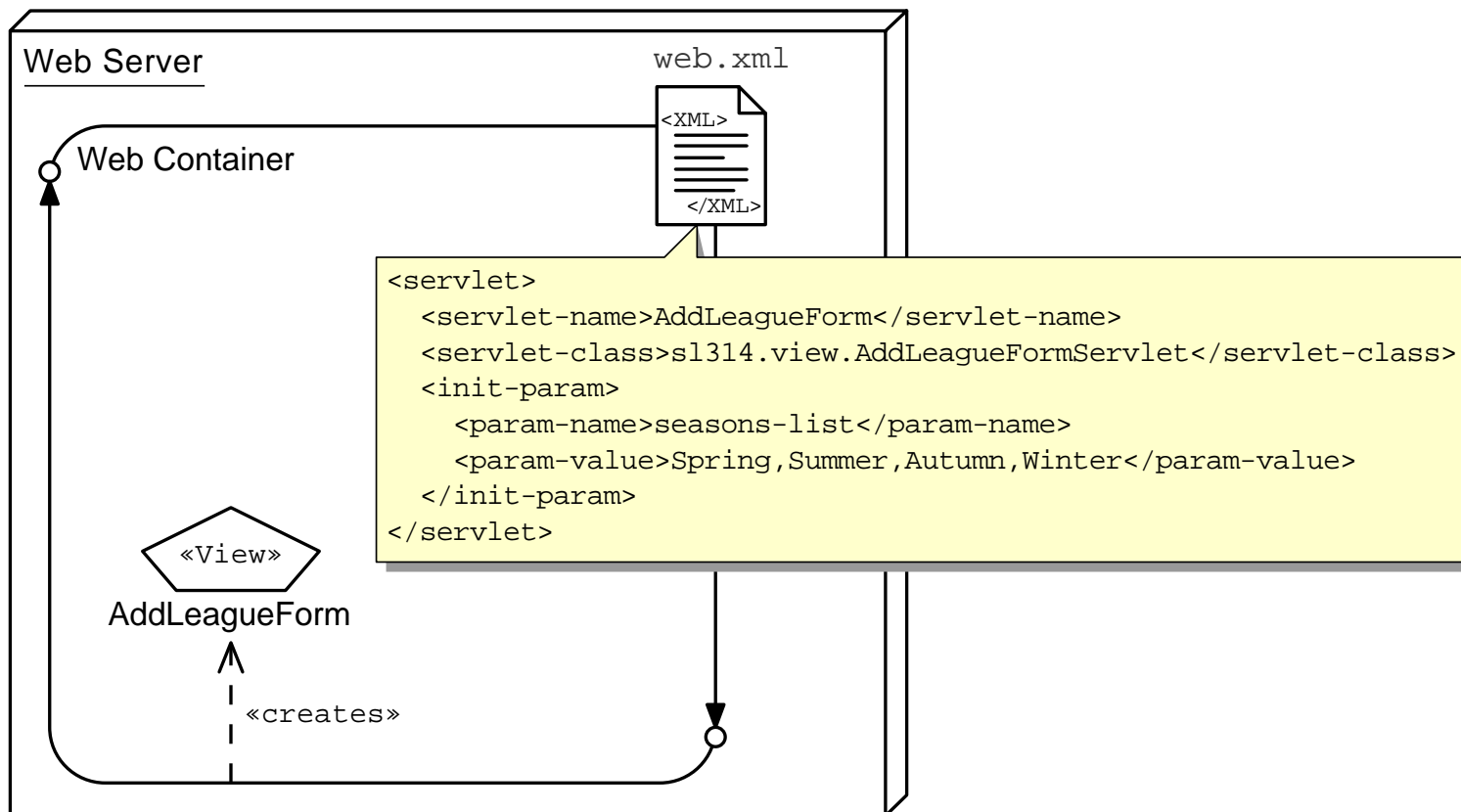
# Servlet Class Loading



Classes can be in: WEB-INF/classes/, WEB-INF/lib/\*.jar, plus J2SE classes, and container classes.

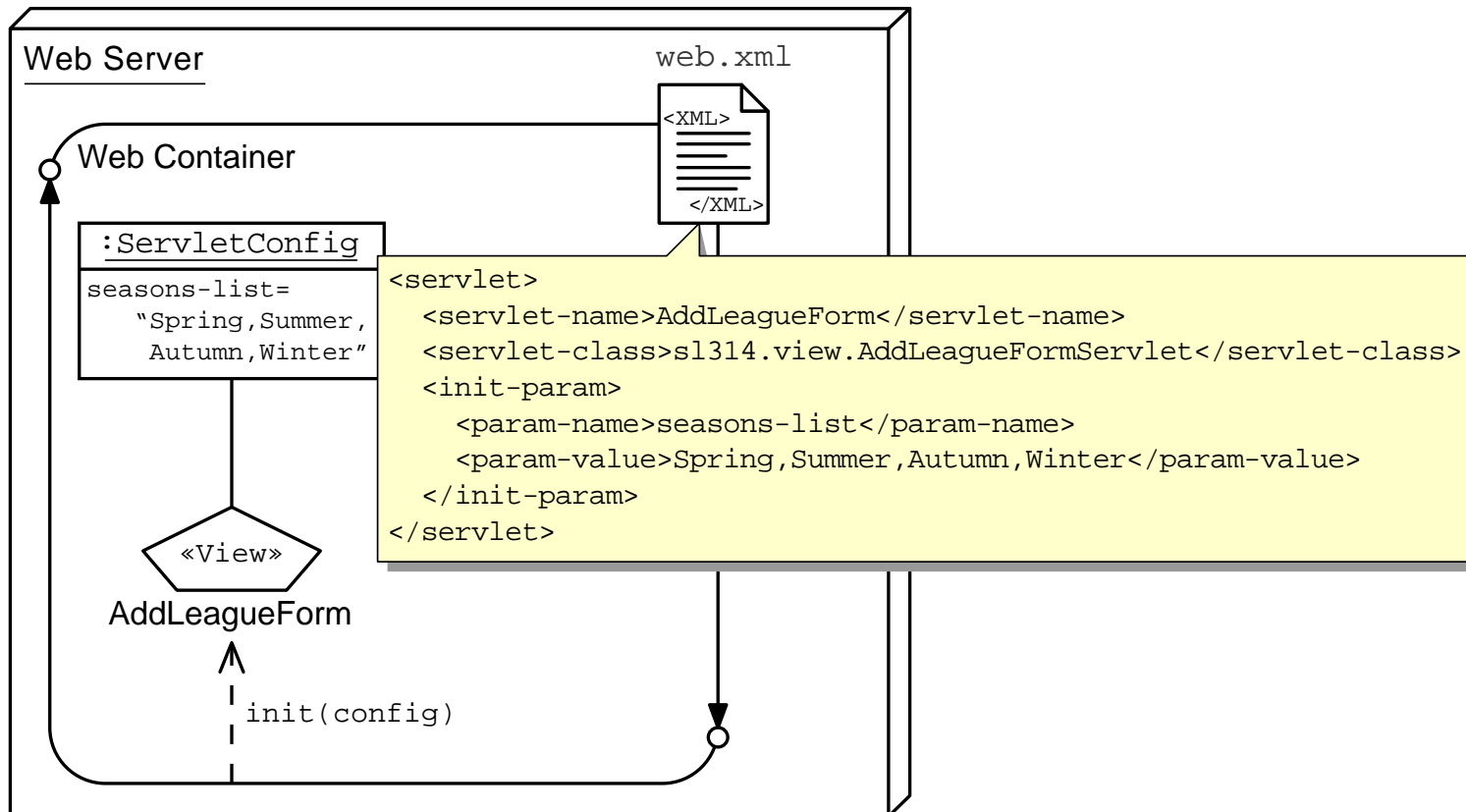


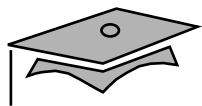
# One Instance Per Servlet Definition



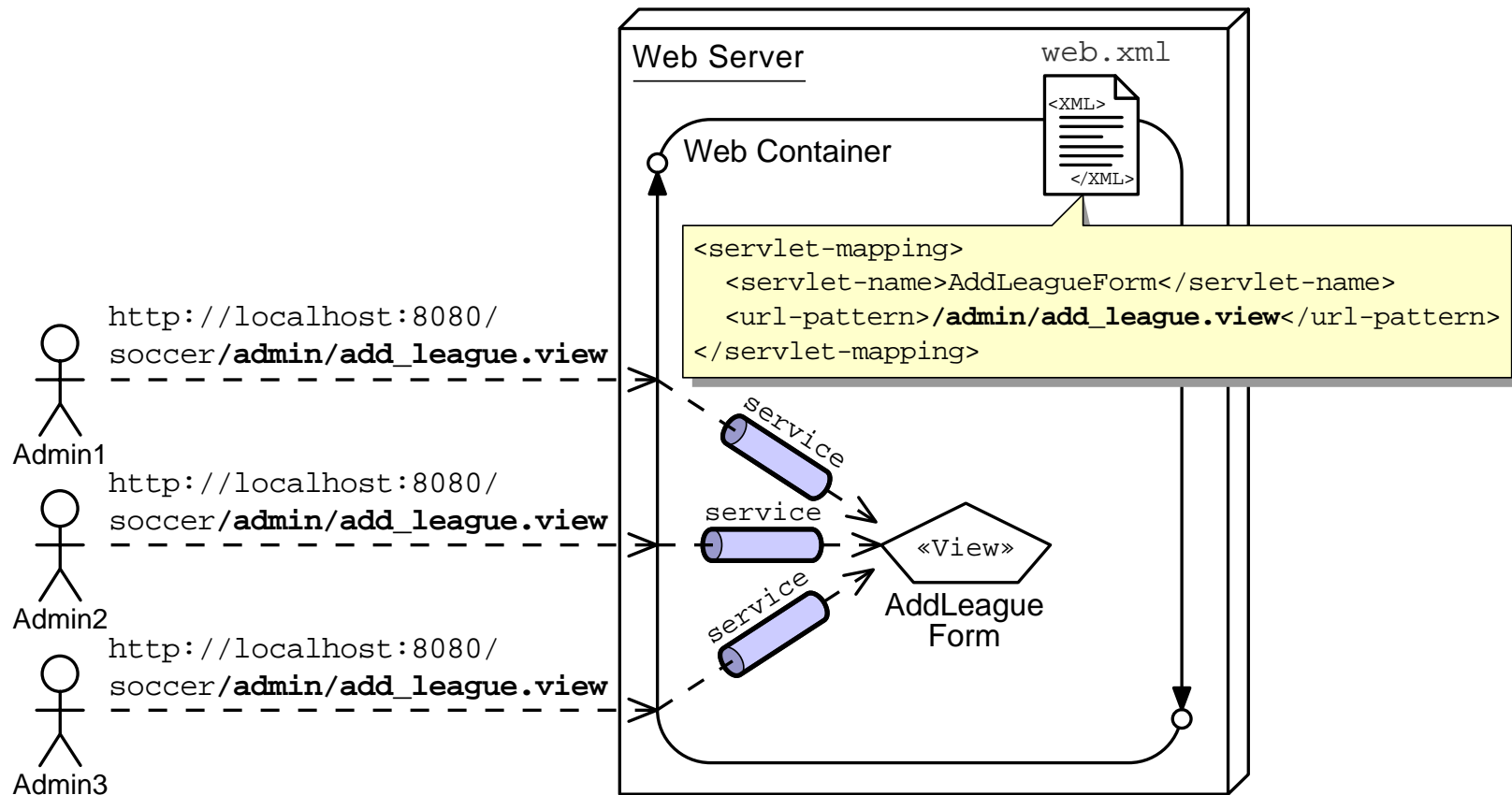


# The `init` Life Cycle Method

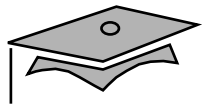




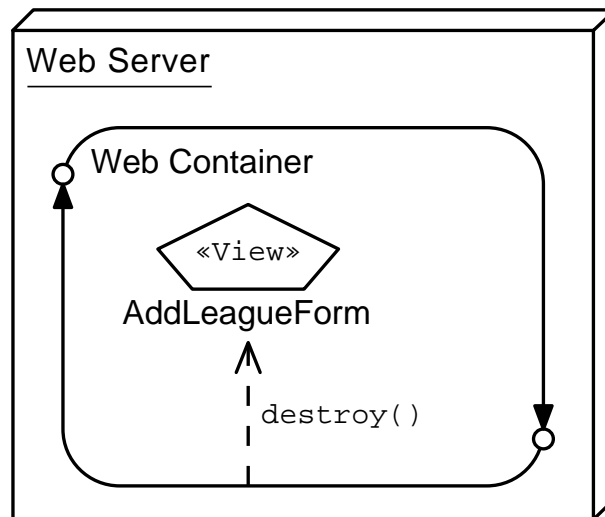
# The service Life Cycle Method







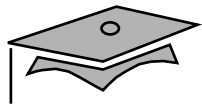
# The destroy Life Cycle Method





## Objectives

- Describe the servlet life cycle
- Customize a servlet with initialization parameters
- Error reporting within the web form
- Repopulating the web form



# Customizing the Add a New League Form

US-centric season names:

This form allows you to create a new soccer league.

Year:

Season:

Title:

- select...
- Spring
- Summer
- Fall
- Winter

Customized season names:

This form allows you to create a new soccer league.

Year:

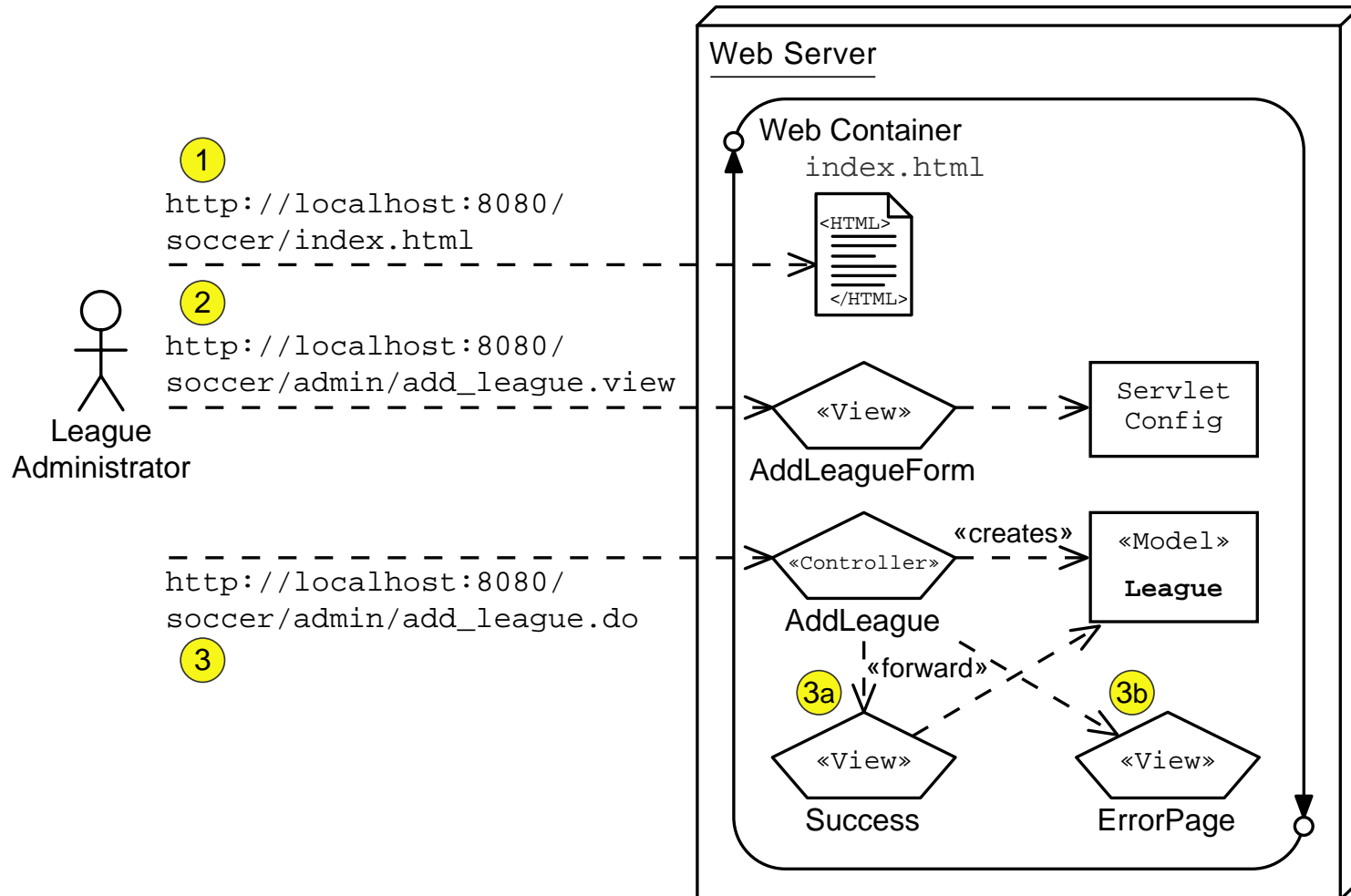
Season:

Title:

- select...
- Spring
- Summer
- Autumn
- Winter



# Add League Architecture Model (Step 1)





## The AddLeagueFormServlet Code

```
74 out.println("</p>");
75 out.println("<form action='add_league.do' method='POST'>");
76
77 // Display the year field
78 out.println("Year: <input type='text' name='year' /> <br/><br/>");
79
80 // Customize the season drop-down menu
81 out.println("Season: <select name='season'>");
82 out.println("    <option value='UNKNOWN'>select...</option>");
83 for ( int i = 0; i < SEASONS.length; i++ ) {
84     out.print("    <option value='" + SEASONS[i] + "'>");
85     out.println(">" + SEASONS[i] + "</option>");
86 }
87 out.println("    </select> <br/><br/>");
88
89 // Display the title field
90 out.println("Title: <input type='text' name='title' /> <br/><br/>");
91
92 out.println("<input type='Submit' value='Add League' />");
93 out.println("</form>");
```



## Configuring Initialization Parameters

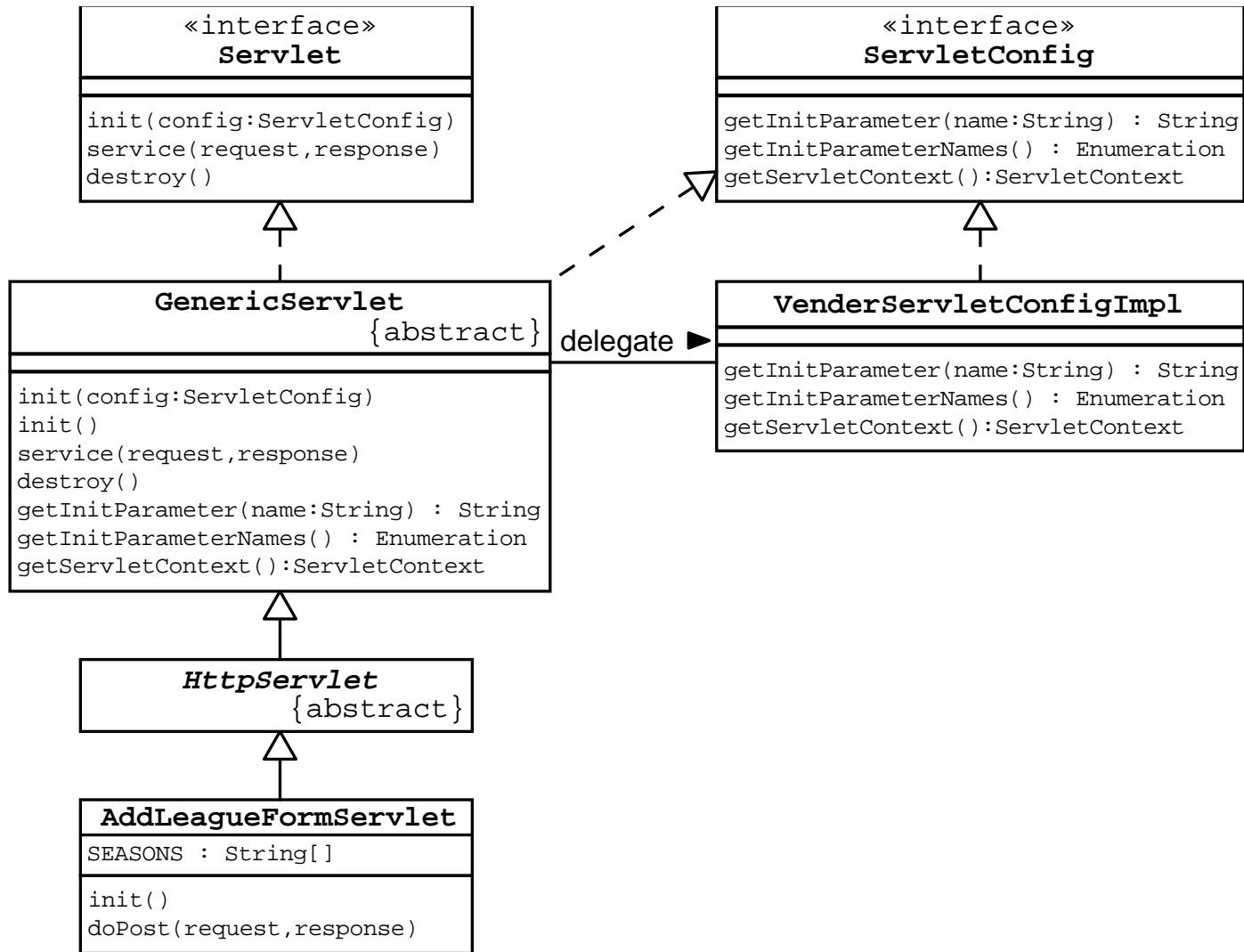
Deployment descriptor for a servlet initialization parameter:

```
20     <servlet>
21         <servlet-name>AddLeagueForm</servlet-name>
22         <servlet-class>sl314.view.AddLeagueFormServlet</servlet-class>
23         <init-param>
24             <param-name>seasons-list</param-name>
25             <param-value>Spring,Summer,Autumn,Winter</param-value>
26         </init-param>
27     </servlet>
```

A servlet can have any number of initialization parameters.



# The ServletConfig API





## The AddLeagueFormServlet Code

```
11
12 public class AddLeagueFormServlet extends HttpServlet {
13
14     /** There are the default seasons for the US. */
15     private static final String DEFAULT_SEASONS
16         = "Spring,Summer,Fall,Winter";
17
18     /** This variable holds the set of seasons. */
19     private String[] SEASONS;
20
21     /** The init method configures the set of seasons. */
22     public void init() {
23         String seasons_list = getInitParameter("seasons-list");
24         if ( seasons_list == null ) {
25             seasons_list = DEFAULT_SEASONS;
26         }
27         SEASONS = seasons_list.split(",");
28     }
29
```



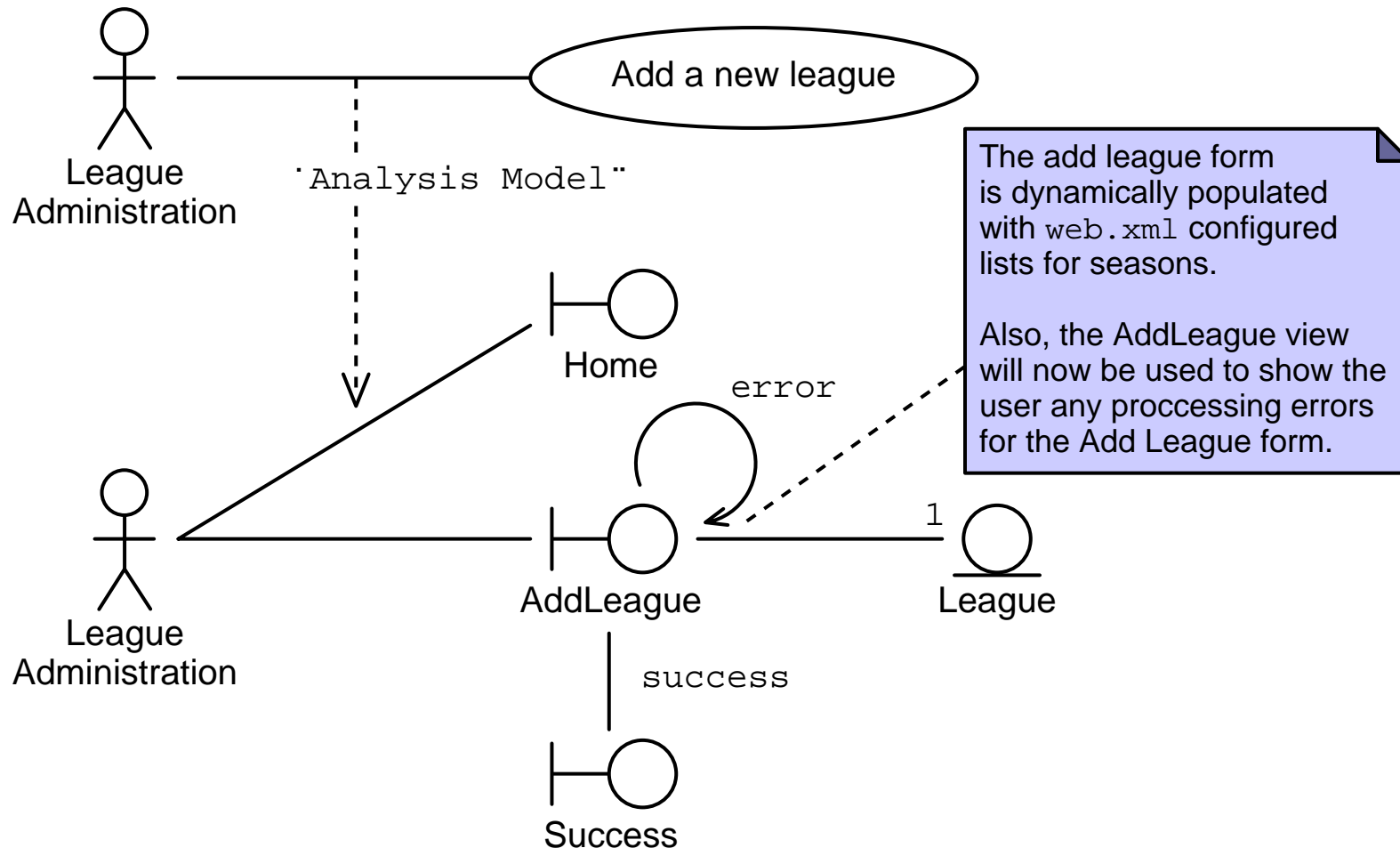


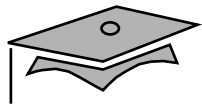
## Objectives

- Describe the servlet life cycle
- Customize a servlet with initialization parameters
- **Error reporting within the web form**
- Repopulating the web form



## Add League Analysis Model (Stage 2)





# Error Handling Screen Shots

Duke's Soccer League: Add a New League - Netscape

http://localhost:8080/config2/admin/add\_leagu

**Duke's Soccer League: Add a New League**

This form allows you to create a new soccer league.

Year:

Season:

Title:

Done

Duke's Soccer League: Add a New League - Netscape

http://localhost:8080/config2/admin/add\_le

**Duke's Soccer League: Add a New League**

Please correct the following errors:

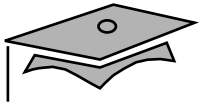
- The 'year' field must be a positive integer.
- Please select a league season.

This form allows you to create a new soccer league.

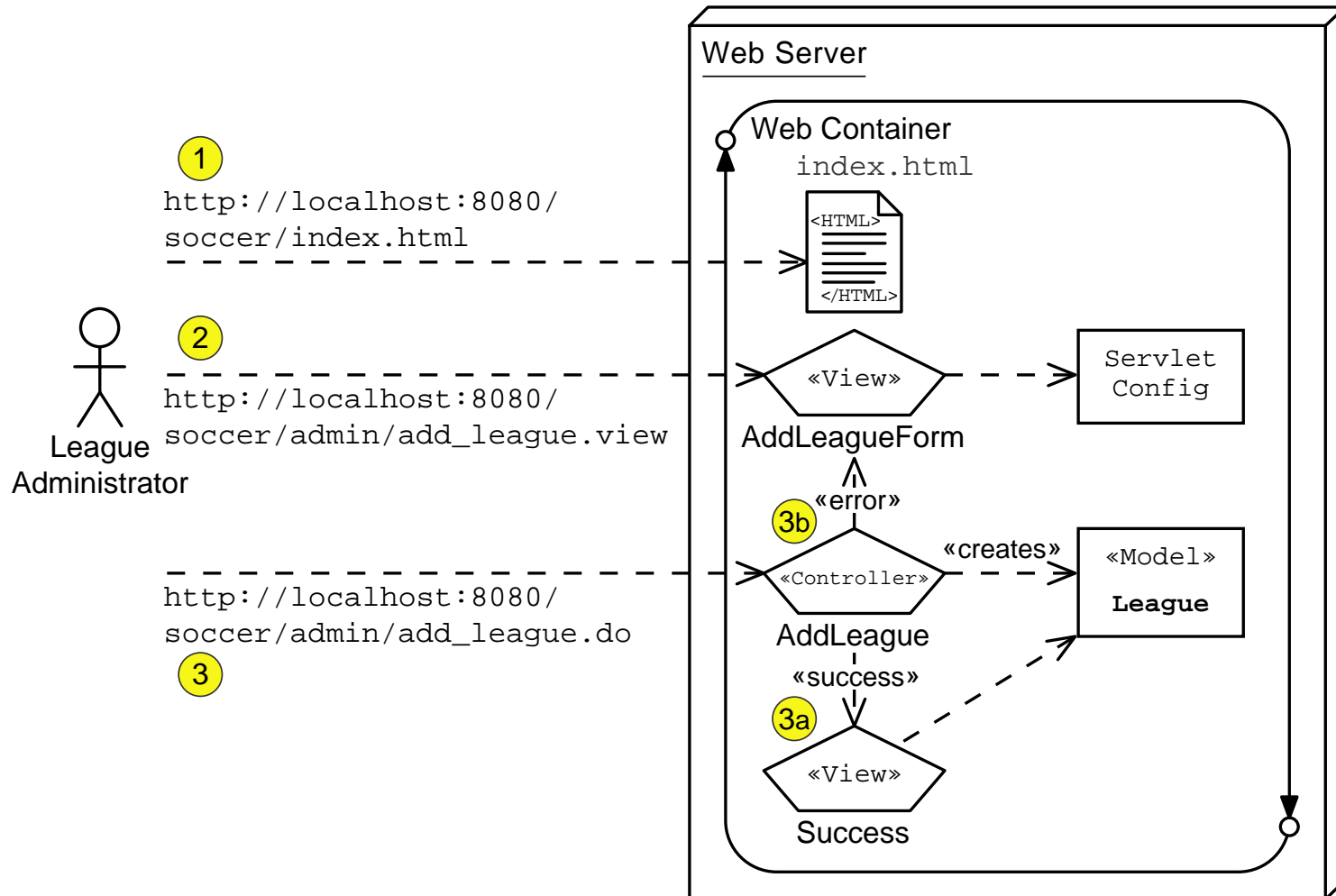
Year:

Season:

Title:

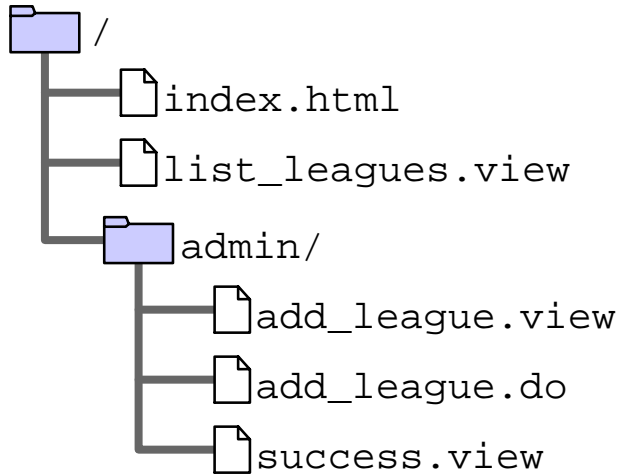


# Add League Architecture Model (Stage 2)



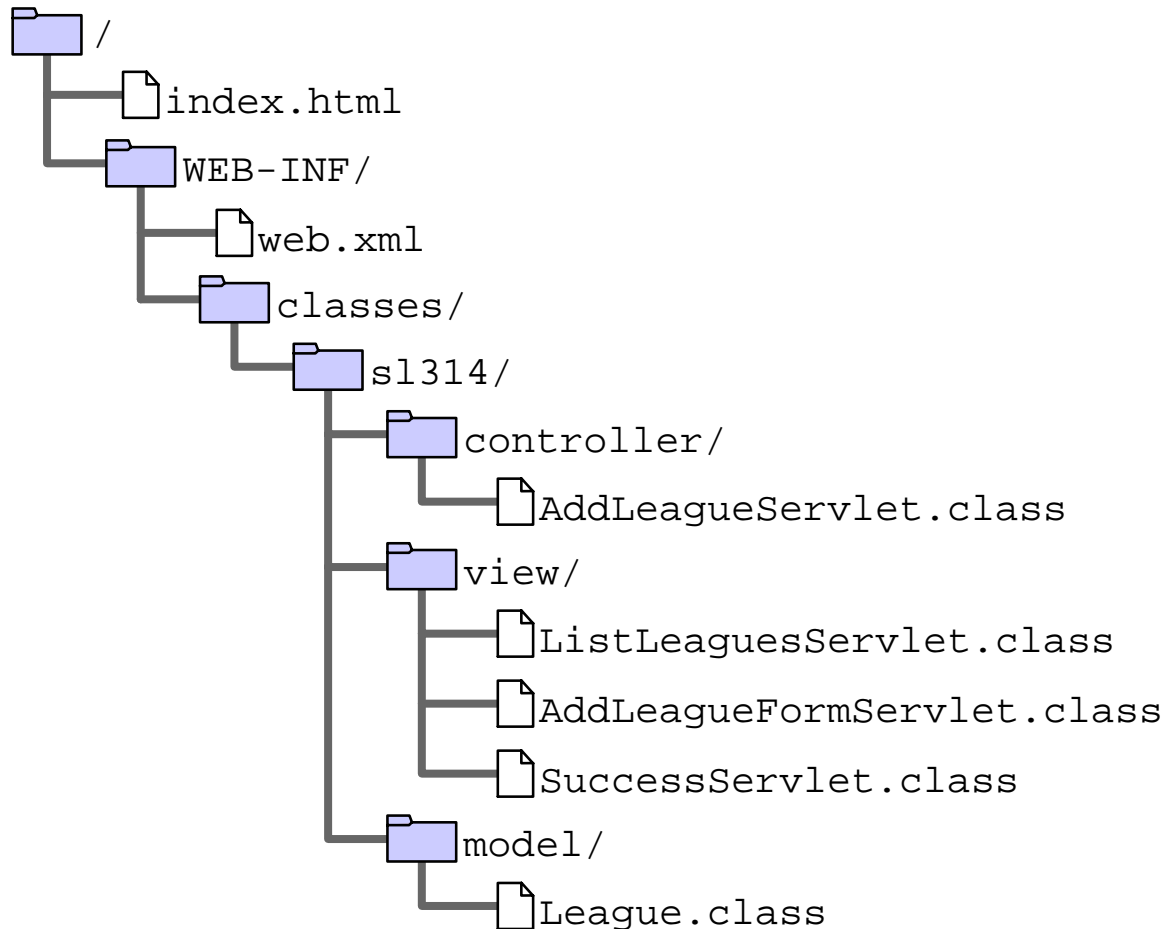


# Soccer League Web Application Structure





# Soccer League Web Application Structure





## The AddLeagueServlet Code

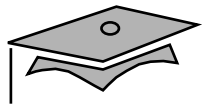
```
43
44     // Verify form parameters
45     if ( (year != -1) && ((year < 2000) || (year > 2010)) ) {
46         errorMsgs.add("The 'year' field must within 2000 to 2010.");
47     }
48     if ( season.equals("UNKNOWN") ) {
49         errorMsgs.add("Please select a league season.");
50     }
51     if ( title.length() == 0 ) {
52         errorMsgs.add("Please enter the title of the league.");
53     }
54
55     // Send the user back to the AddDVD form, if there were errors
56     if ( ! errorMsgs.isEmpty() ) {
57         RequestDispatcher view
58             = request.getRequestDispatcher("add_league.view");
59         view.forward(request, response);
60         return;
61     }
```



## The AddLeagueFormServlet Code

```
28
29 public void doGet(HttpServletRequest request,
30                  HttpServletResponse response)
31     throws IOException {
32     generateView(request, response);
33 }
34
35 public void doPost(HttpServletRequest request,
36                  HttpServletResponse response)
37     throws IOException {
38     generateView(request, response);
39 }
40
41 public void generateView(HttpServletRequest request,
42                          HttpServletResponse response)
43     throws IOException {
```





## The AddLeagueFormServlet Code

```
41 public void generateView(HttpServletRequest request,
42                         HttpServletResponse response)
43     throws IOException {
44
45     // Set page title
46     String pageTitle = "Duke's Soccer League: Add a New League";
47
48     // Retrieve the errorMsgs from the request-scope
49     List errorMsgs = (List) request.getAttribute("errorMsgs");
50
51     // Specify the content type is HTML
52     response.setContentType("text/html");
53     PrintWriter out = response.getWriter();
54
55     // Generate the HTML response
56     out.println("<html>");
57     out.println("<head>");
58     out.println("  <title>" + pageTitle + "</title>");
59     out.println("</head>");
60     out.println("<body bgcolor='white'>");
61
```



## The AddLeagueFormServlet Code

```
69
70     // Report any errors (if any)
71     if ( errorMsgs != null ) {
72         out.println("<p>");
73         out.println("<font color='red'>Please correct the following
errors:</font>");
74         out.println("<ul>");
75         Iterator items = errorMsgs.iterator();
76         while ( items.hasNext() ) {
77             String message = (String) items.next();
78             out.println("  <li>" + message + "</li>");
79         }
80         out.println("</ul>");
81         out.println("</font>");
82         out.println("</p>");
83     }
84
```



## Objectives

- Describe the servlet life cycle
- Customize a servlet with initialization parameters
- Error reporting within the web form
- Repopulating the web form



# Repopulating Web Forms

Duke's Soccer League: Add a New League - Netscape

http://localhost:8080/config2/admin/add\_leag

**Duke's Soccer League: Add a New League**

This form allows you to create a new soccer league.

Year:

Season:

Title:

Done

Duke's Soccer League: Add a New League - Netscape

http://localhost:8080/config2/admin/add\_le

**Duke's Soccer League: Add a New League**

Please correct the following errors:

- The 'year' field must be a positive integer.

This form allows you to create a new soccer league.

Year:

Season:

Title:



## Repopulating a Text Field

```
84
85     // Generate main body
86     out.println("<p>");
87     out.println("This form allows you to create a new soccer league.");
88     out.println("</p>");
89     out.println("<form action='add_league.do' method='POST'>");
90
91     // Repopulate the year field
92     String year = request.getParameter("year");
93     if ( year == null ) {
94         year = "";
95     }
96     out.println("Year: <input type='text' name='year' value='"
97                 + year + "` /> <br/><br/>");
98
```



## Repopulating a Drop-Down List

```
98
99 // Repopulate the season drop-down menu
100 String season = request.getParameter("season");
101 out.println("Season: <select name='season'>");
102 if ( (season == null) || season.equals("UNKNOWN") ) {
103     out.println("        <option value='UNKNOWN'>select...</option>");
104 }
105 for ( int i = 0; i < SEASONS.length; i++ ) {
106     out.print("        <option value='" + SEASONS[i] + "'");
107     if ( SEASONS[i].equals(season) ) {
108         out.print(" selected");
109     }
110     out.println(">" + SEASONS[i] + "</option>");
111 }
112 out.println("        </select> <br/><br/>");
113
```



## Summary

- Usually, web forms should be dynamic to allow for customization, error reporting, and repopulating fields after an error.
- You can use servlet initialization parameters to help customize forms, but `init` parameters can be used for many more purposes.
- You can use the `init()` method to read the `init` parameters and perform servlet configuration.



# Module 5

## Sharing Application Resources Using the Servlet Context





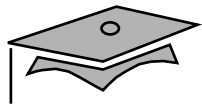
## Objectives

- Describe the purpose and features of the servlet context
- Develop a servlet context listener to initialize a shared application resource



## Relevance

- How can you share application data in a web application?
- When should this shared data be loaded into working memory?



# Soccer League Demonstration



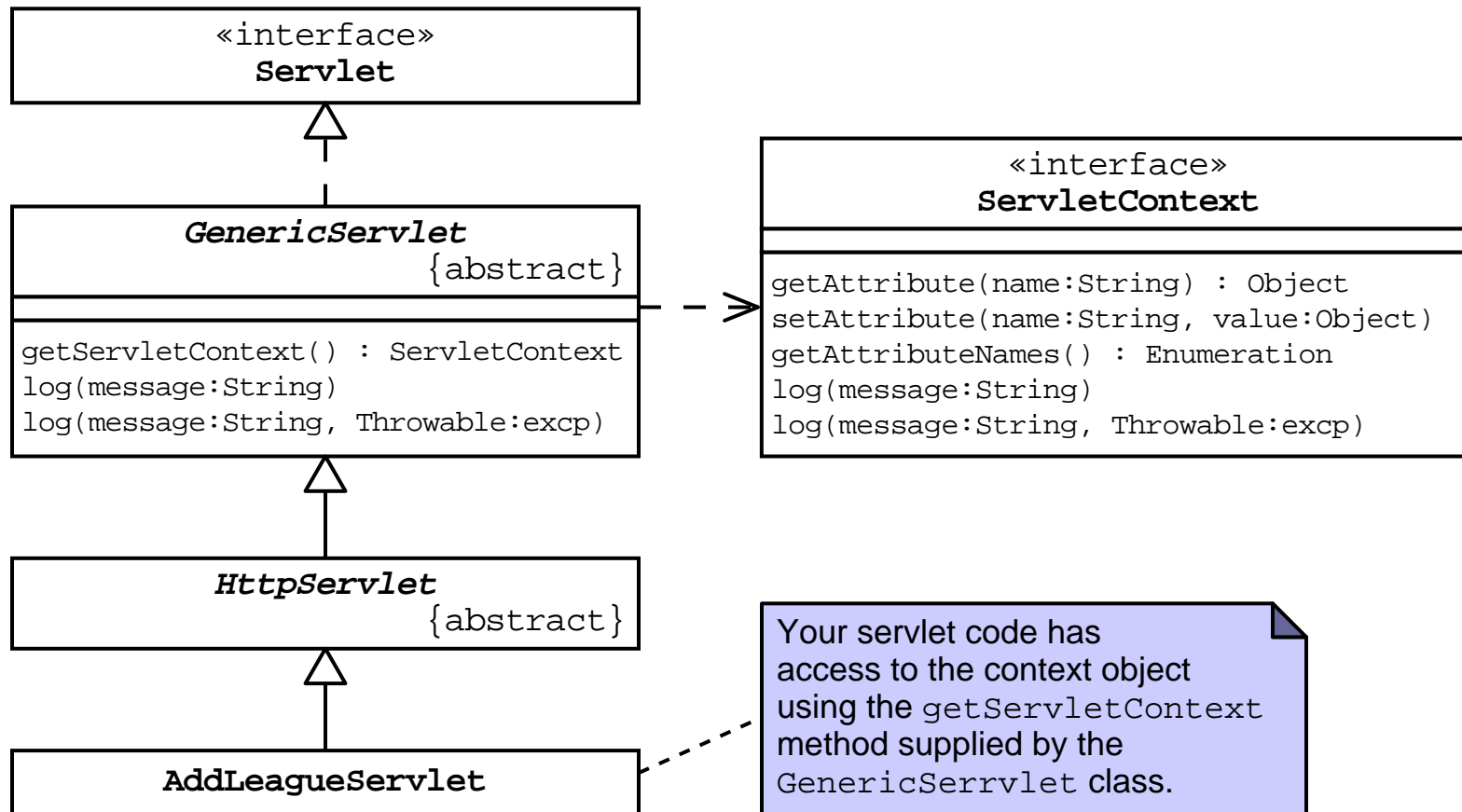


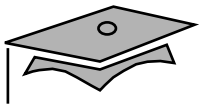
## Servlet Context

- A web application is a self-contained collection of static and dynamic resources.
- 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.

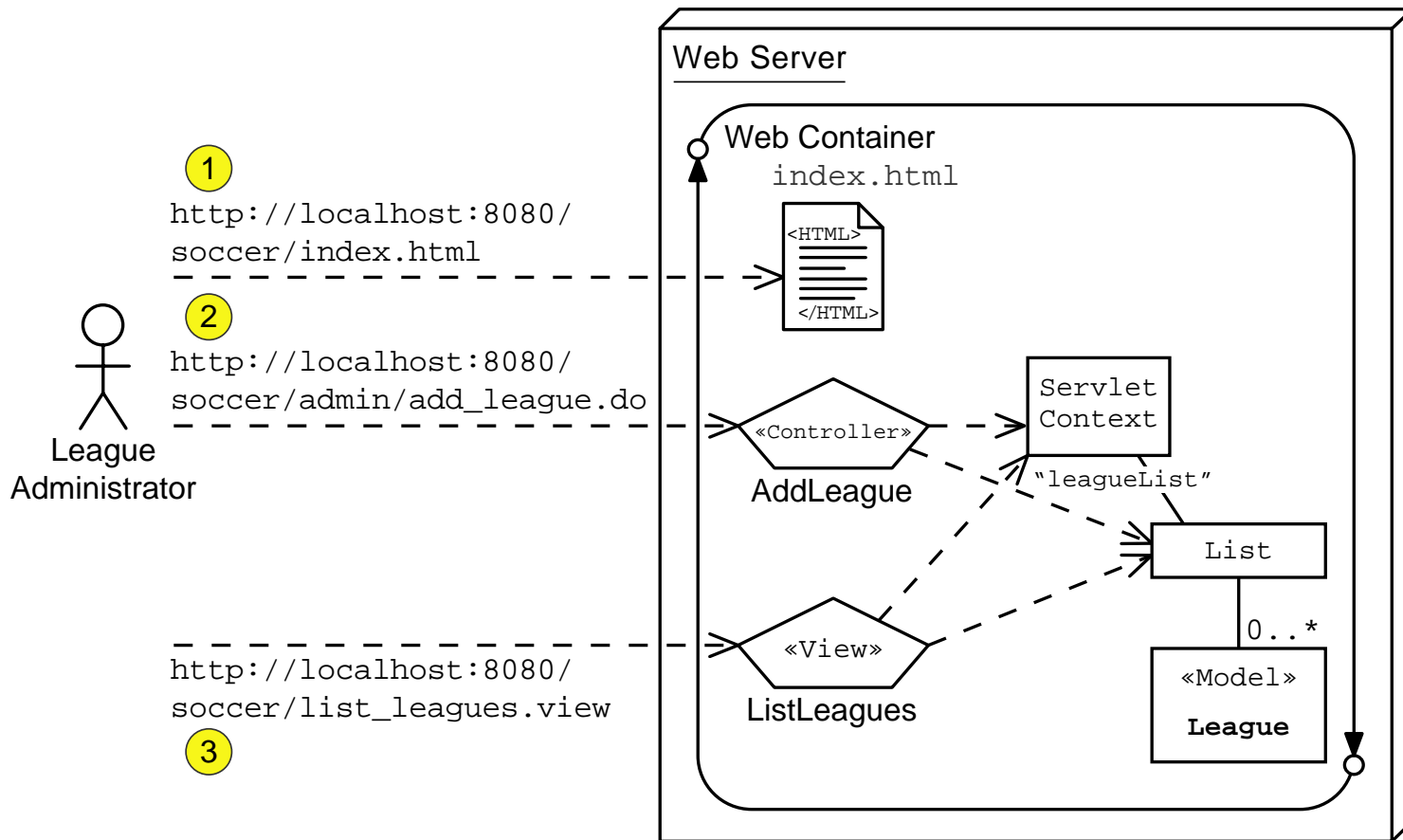


# The ServletContext API





# Soccer League Architecture Model





## Modified AddLeagueServlet Code

```
63
64     // Perform business logic
65     League league = new League(year, season, title);
66     // Store the new league in the request-scope
67     request.setAttribute("league", league);
68
69     // Store the new league in the leagueList context-scope attribute
70     ServletContext context = getServletContext();
71     List leagueList = (List) context.getAttribute("leagueList");
72     leagueList.add(league);
73
74     // Send the Success view
75     RequestDispatcher view
76     = request.getRequestDispatcher("success.view");
77     view.forward(request, response);
78     return;
```



## Modified ListLeaguesServlet Code

```
14
15 public class ListLeaguesServlet extends HttpServlet {
16
17     public void doGet(HttpServletRequest request,
18                       HttpServletResponse response)
19         throws IOException {
20
21         // Set page title
22         String pageTitle = "Duke's Soccer League: List Leagues";
23
24         // Retrieve the list of leagues from the context-scope
25         ServletContext context = getServletContext();
26         List leagueList = (List) context.getAttribute("leagueList");
27
28         // Specify the content type is HTML
29         response.setContentType("text/html");
30         PrintWriter out = response.getWriter();
31
32         // Generate the HTML response
33         out.println("<html>");
```





## Modified ListLeaguesServlet Code (Part 2)

```
46
47     // Generate main body
48     out.println("<p>");
49     out.println("The set of soccer leagues are:");
50     out.println("</p>");
51
52     out.println("<ul>");
53     Iterator items = leagueList.iterator();
54     while ( items.hasNext() ) {
55         League league = (League) items.next();
56         out.println("  <li>" + league.getTitle() + "</li>");
57     }
58     out.println("</ul>");
```



## Objectives

- Describe the purpose and features of the servlet context
- Develop a servlet context listener to initialize a shared application resource



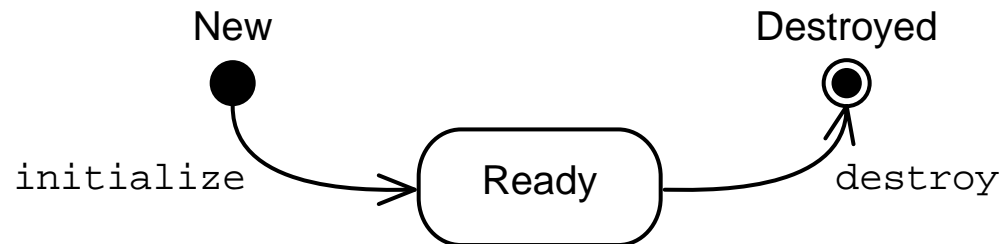
## League List Initialization Example

The following tasks need to be performed to initialize the `leagueList` context-scoped attribute:

1. Determine the location of the `leagues.txt` file.
2. Read the `leagues.txt` file.
3. Create `League` objects for each row in the `leagues.txt` file and store them in a `List` object.
4. Store the list of leagues in the `leagueList` context attribute.
5. Log the fact that the list was initialized, or log any exception thrown by this code.



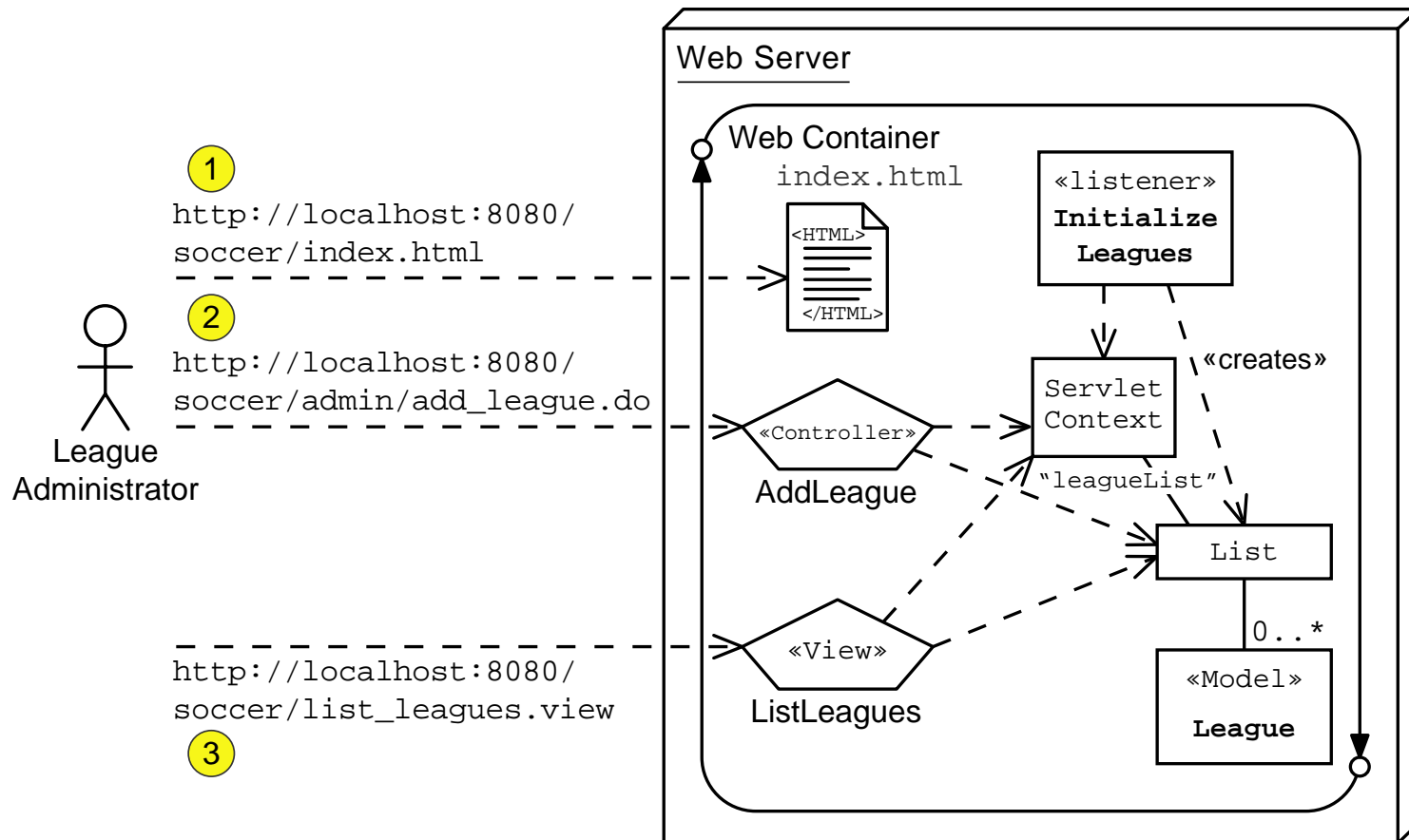
# Web Application Life Cycle

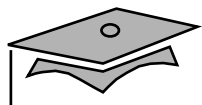


- When the web container is started, each web application is initialized.
- When the web container is shutdown, each web application is destroyed.
- A servlet context listener can be used to receive these web application life cycle events.

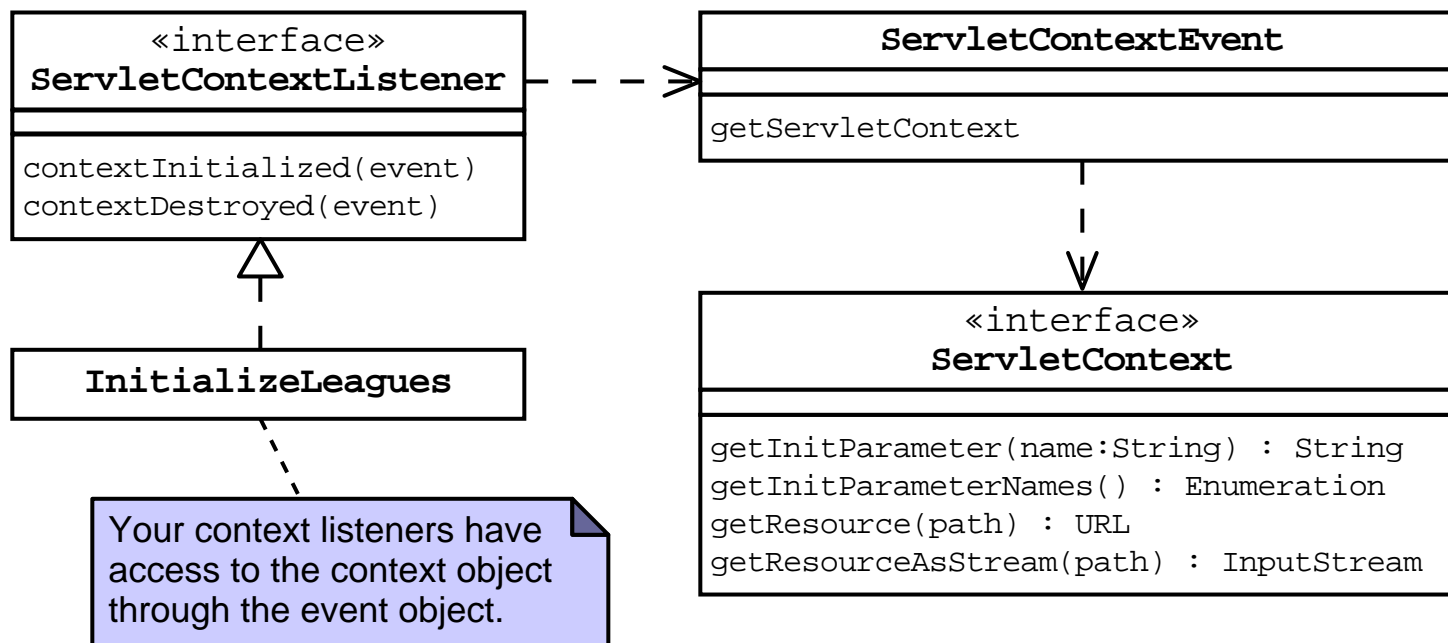


# Soccer League Architecture Model (Revisited)





# The ServletContextListener API





## The InitializeLeagues Code

```
3  import javax.servlet.ServletContextListener;
4  import javax.servlet.ServletContextEvent;
5  import javax.servlet.ServletContext;
6  // Support classes
7  import java.io.InputStream;
8  import java.io.InputStreamReader;
9  import java.io.BufferedReader;
10 // Model classes
11 import sl314.model.League;
12 import java.util.List;
13 import java.util.LinkedList;
14
15 public class InitializeLeagues implements ServletContextListener {
16
17     public void contextInitialized(ServletContextEvent event) {
18         ServletContext context = event.getServletContext();
19         List leagueList = new LinkedList();
20         String leaguesFile = context.getInitParameter("leagues-file");
21         InputStream is = null;
22         BufferedReader reader = null;
23
```



## The InitializeLeagues Code (Part 2)

```
21     InputStream is = null;
22     BufferedReader reader = null;
23
24     try {
25         is = context.getResourceAsStream(leaguesFile);
26         reader = new BufferedReader(new InputStreamReader(is));
27         String record;
28
29         // Read every record (one per line)
30         while ( (record = reader.readLine()) != null ) {
31             String[] fields = record.split("\t");
32
33             // Extract the data fields for the record
34             int year = Integer.parseInt(fields[0]);
35             String season = fields[1];
36             String title = fields[2];
37
38             // Add the new League item to the list
39             League item = new League(year, season, title);
40             leagueList.add(item);
41         }
```





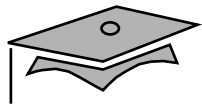
## The InitializeLeagues Code (Part 3)

```
42
43     context.setAttribute("leagueList", leagueList);
44
45     context.log("The league list has been loaded.");
46
47     } catch (Exception e) {
48         context.log("Exception occurred while processing the leagues
file.", e);
49
50     } finally {
51         if ( is != null ) {
52             try { is.close(); } catch (Exception e) {}
53         }
54         if ( reader != null ) {
55             try { reader.close(); } catch (Exception e) {}
56         }
57     }
58
59     } // END of contextInitialized
```

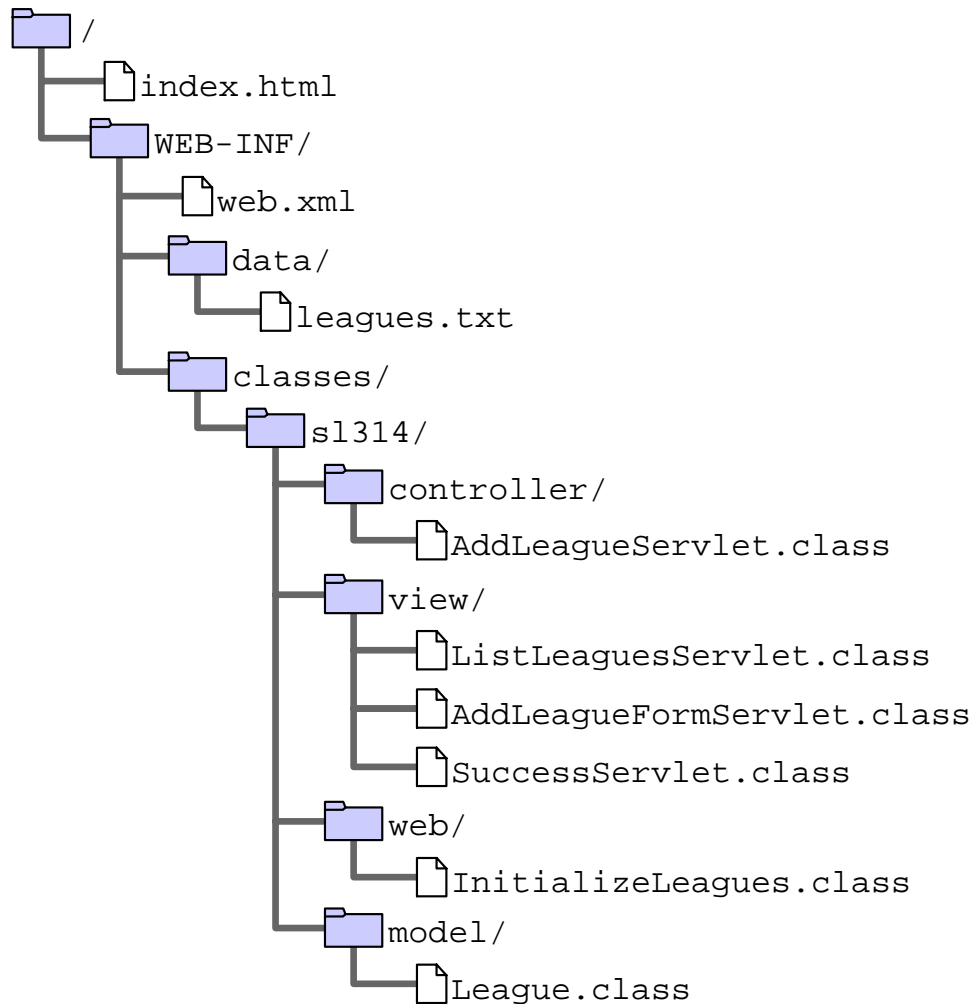


# Soccer League Deployment Descriptor

```
3 <web-app
4     xmlns="http://java.sun.com/xml/ns/j2ee"
5     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
6     xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
7         http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
8     version="2.4">
9
10    <display-name>SL-314 WebApp Example</display-name>
11    <description>
12        This Web Application demonstrates using the context-scope to store
13        a common resource: the "leagueList" for the Soccer League webapp.
14    </description>
15
16    <context-param>
17        <param-name>leagues-file</param-name>
18        <param-value>/WEB-INF/data/leagues.txt</param-value>
19    </context-param>
20
21    <listener>
22        <listener-class>sl314.web.InitializeLeagues</listener-class>
23    </listener>
```



# Soccer League Physical Hierarchy





## Summary

- The `ServletContext` object can store application attributes (name/object pairs) globally across all web components.
- You can initialize shared application resources by creating a class that implements the `ServletContextListener` interface.



# Module 6

## Designing the Business Tier



# Objectives

- Describe the Analysis model
- Design entity components
- Design service components



## Relevance




- What domain entities are required for the Register for a League use case?
- How might this data be persisted?
- What types of operations cannot be performed by entity classes?
- What type of components might you use to perform these operations?



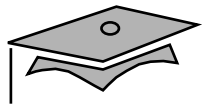
## Describing the Analysis Model

An Analysis model is used to bridge the gap between use case analysis and component design.

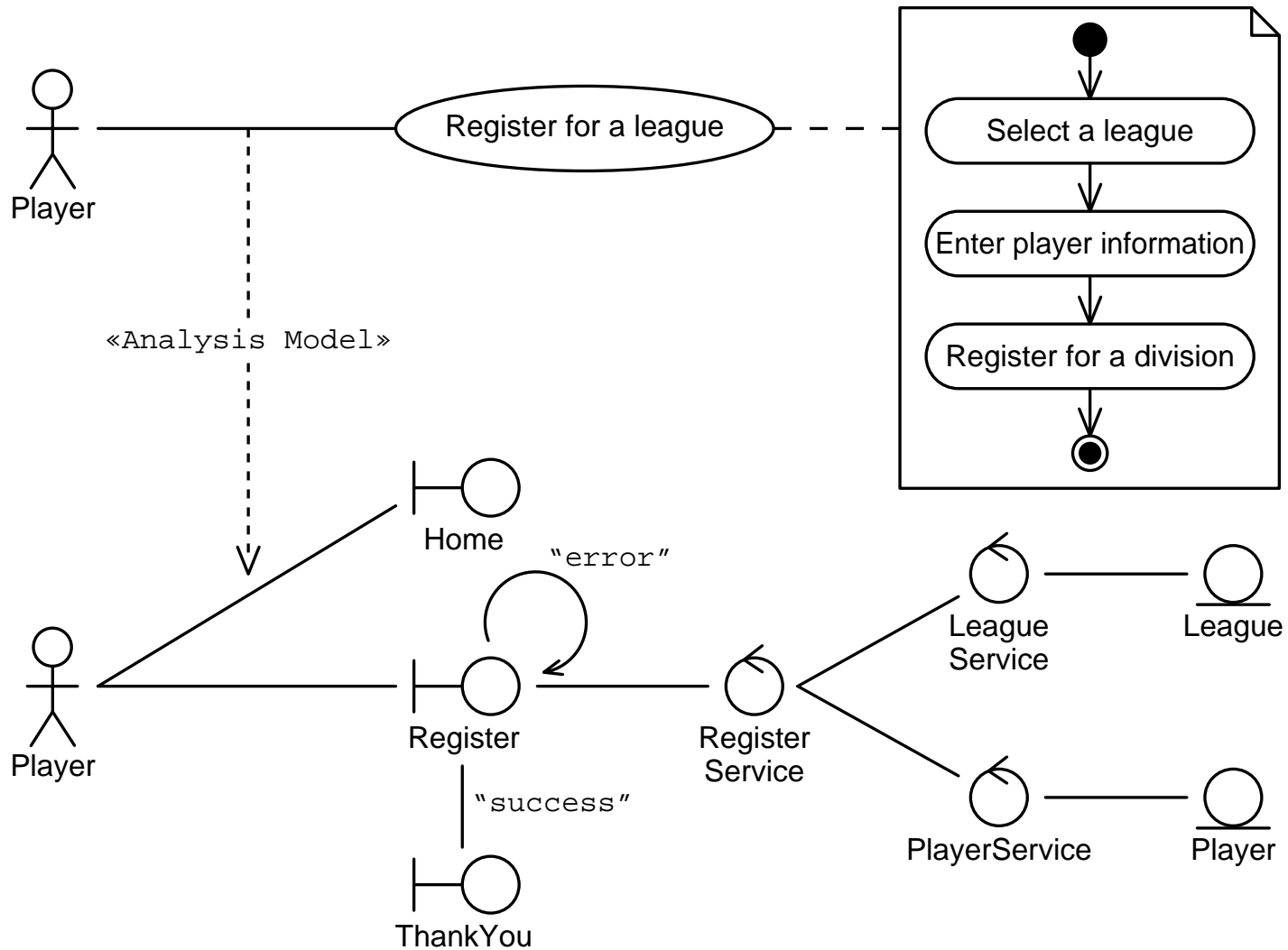
An Analysis model consists of three abstract component types:

Component	Symbol	Description
Boundary		Communicates between the user and the system.
Service		Provides a services-oriented layer between boundary and entity components.
Entity		Represents domain objects and persistent data.



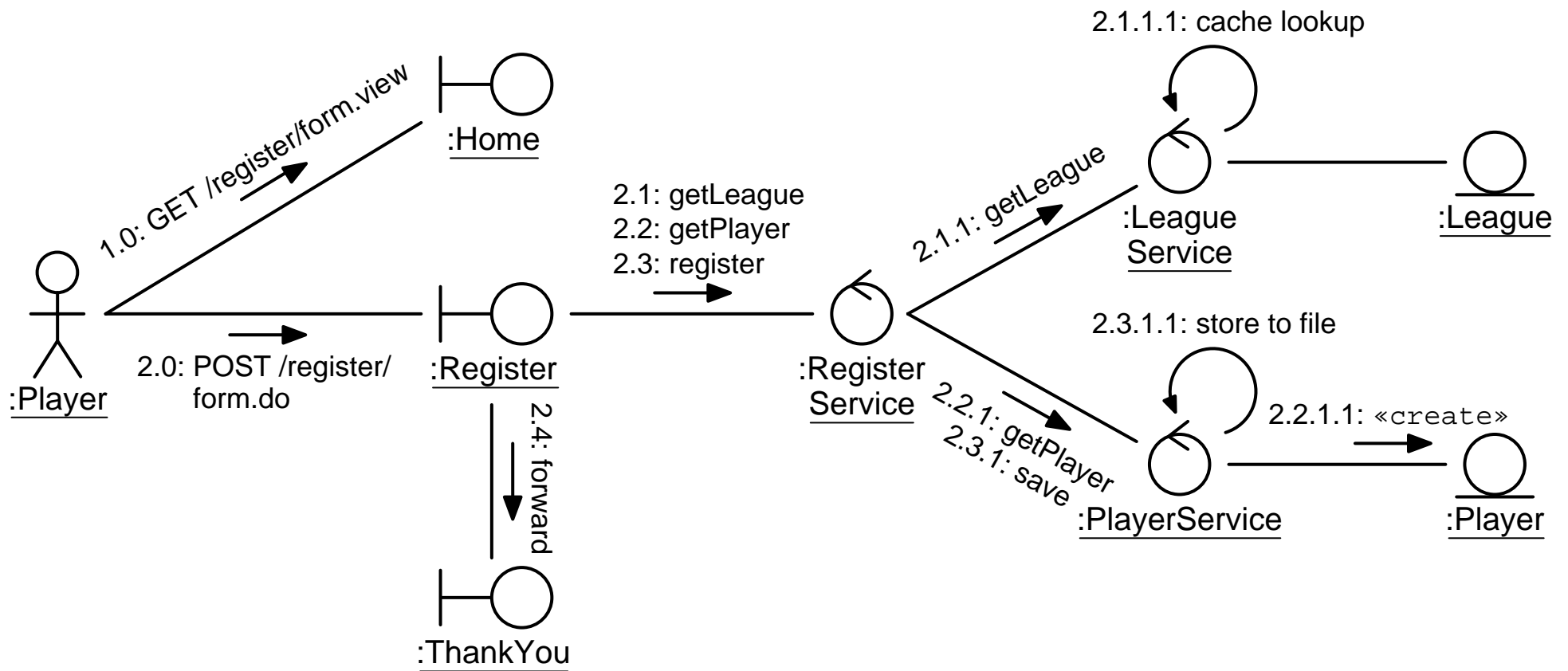


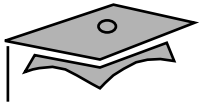
# Registration Use Case Analysis Model



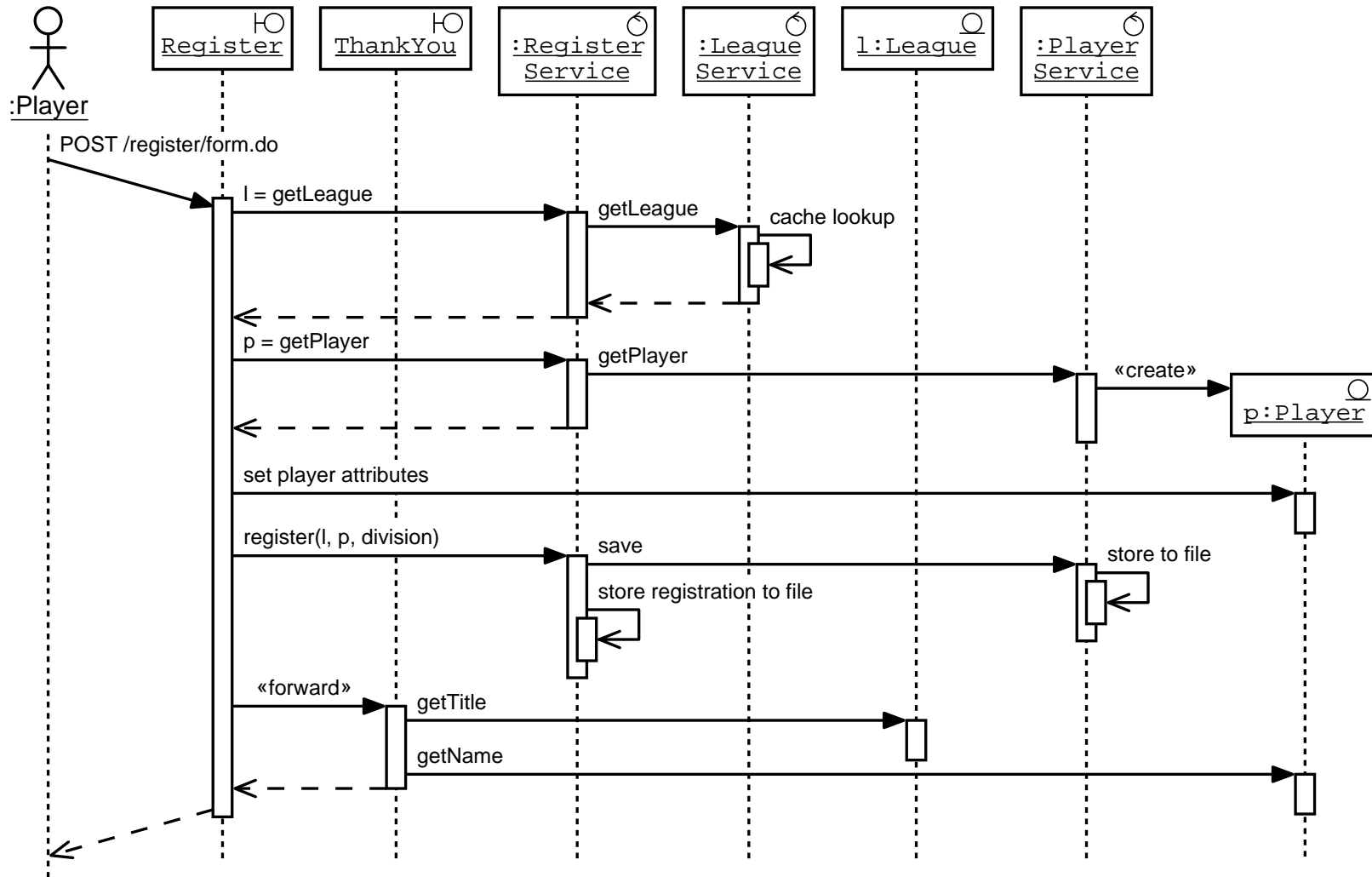


# Detailed Analysis Model



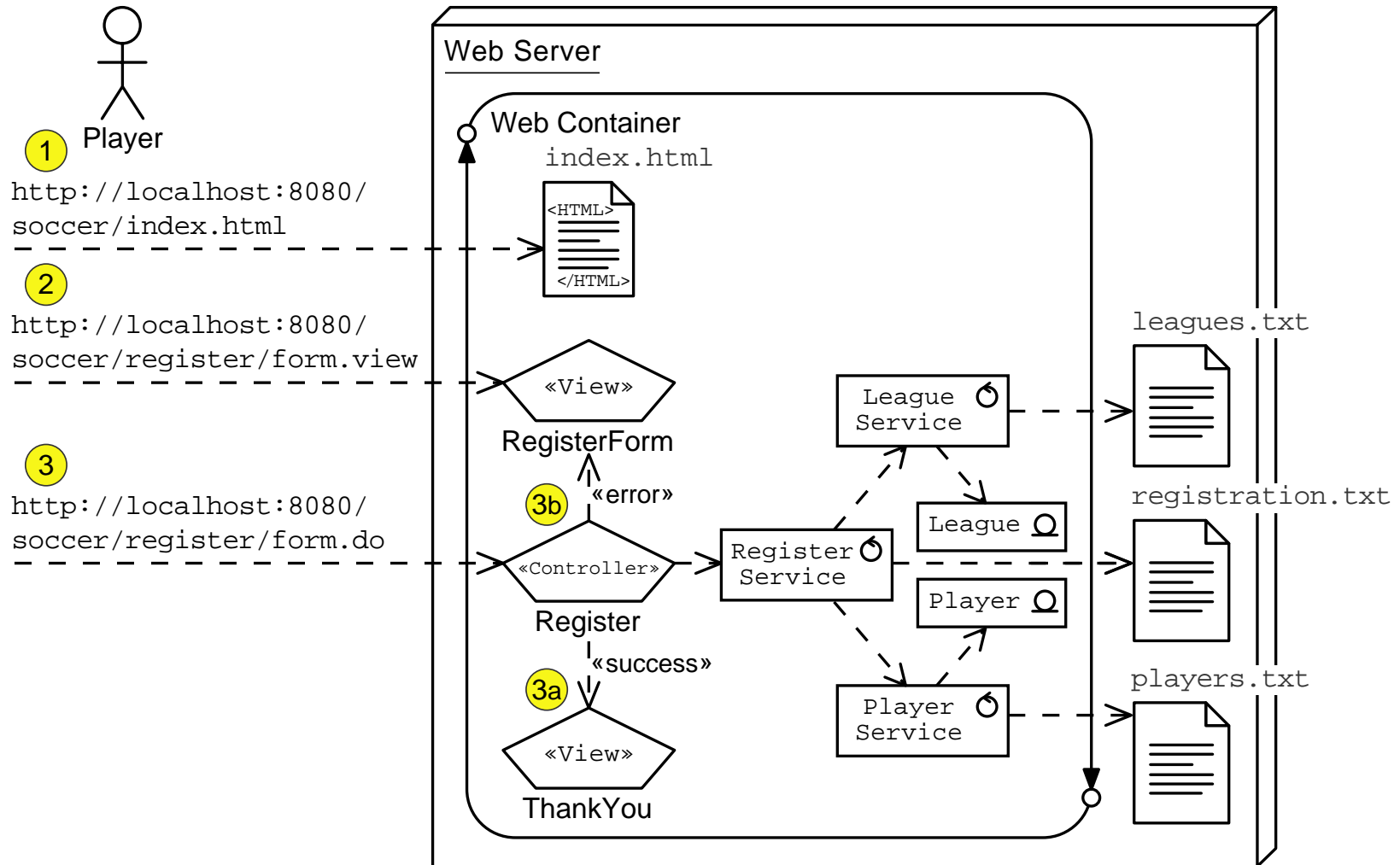


# Another View: UML Sequence Diagram





# Another View: UML Deployment Diagram





# Objectives

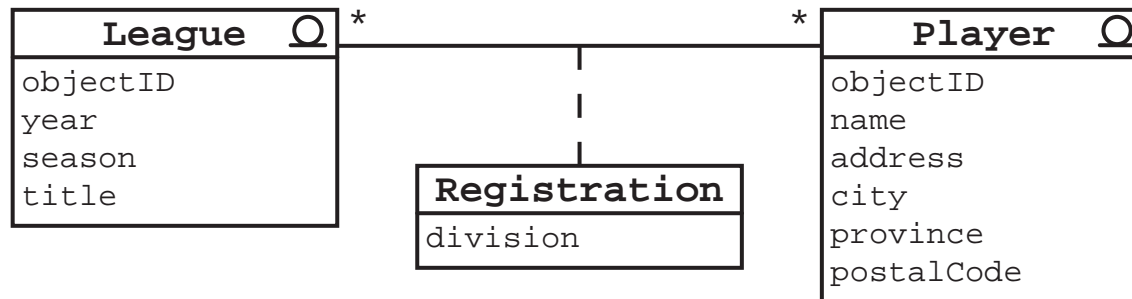
- Describe the Analysis model
- Design entity components
- Design service components



## Domain Entities

Domain entities are real world business objects.

For example:



Can you name other entities that might exist in a Soccer League application?

What about other domains? (such as retail, financial, and so on)



## The Player Code

```
1 package sl314.model;
2
3 /**
4  * This domain object represents a player in a soccer league.
5  */
6 public class Player {
7
8     String name = "";
9     String address = "";
10    String city = "";
11    String province = "";
12    String postalCode = "";
13
14    /**
15     * This is the constructor. It is package-private to prevent misuse.
16     * The PlayerService.getPlayer method should be used to create a
17     * new player object.
18     */
19    Player(String name) {
20        this(name, "", "", "", "");
21    }
```



## The Player Code (Part 2)

```
14    /**
15     * This is the constructor.  It is package-private to prevent misuse.
16     * The PlayerService.getPlayer method should be used to create a
17     * new player object.
18     */
19    Player(String name) {
20        this(name, "", "", "", "");
21    }
22
23    /**
24     * This is the full constructor.
25     */
26    Player(String name, String address, String city,
27           String province, String postalCode) {
28        this.name = name;
29        this.address = address;
30        this.city = city;
31        this.province = province;
32        this.postalCode = postalCode;
33    }
34
```





## The Player Code (Part 3)

```
32     this.postalCode = postalCode;
33 }
34
35 public String getName() {
36     return name;
37 }
38 public void setName(String value) {
39     name = value;
40 }
41 public String getAddress() {
42     return address;
43 }
44 public void setAddress(String value) {
45     address = value;
46 }
47 public String getCity() {
48     return city;
49 }
50 public void setCity(String value) {
51     city = value;
52 }
```



## Objectives

- Describe the Analysis model
- Design entity components
- Design service components



## Entity Service

Some entity-related operations cannot be performed by the entity component itself:

- Creation – Creating a new instance of the entity
- Retrieval – Retrieving a unique instance in the data store
- Selection – Retrieving a set of instances in the data store
- Aggregation – Performing a calculation (such as an average) over a set of instances
- Deletion – Removing an instance from the data store



## The LeagueService Code

```
1  package sl314.model;
2
3  import java.util.List;
4  import java.util.LinkedList;
5  import java.util.Iterator;
6  import java.util.Collections;
7  import java.io.File;
8  import java.io.FileReader;
9  import java.io.BufferedReader;
10 import java.io.FileWriter;
11 import java.io.PrintWriter;
12 import java.io.IOException;
13
14 /**
15  * This object performs a variety of league services, such as looking
16  * up league objects and creating new ones.
17  */
18 public class LeagueService {
19
```



## The LeagueService Code (Part 2)

```
18 public class LeagueService {
19
20     /** The cache of League objects. */
21     private static final List LEAGUES_CACHE = new LinkedList();
22
23     public LeagueService() {
24         // Make sure that the leagues cache has been initialized
25         synchronized ( LEAGUES_CACHE ) {
26             if ( LEAGUES_CACHE.isEmpty() ) {
27                 cacheLeagues();
28             }
29         }
30     }
31
32     /**
33      * This method returns a complete set of leagues.
34      */
35     public List getAllLeagues() {
36         // Return an immutable List; which makes this read-only
37         return Collections.unmodifiableList(LEAGUES_CACHE);
38     }
}
```



## The LeagueService Code (Part 3)

```
39
40  /**
41   * This method finds the specified League object from the
42   * complete set of leagues.
43   */
44  public League getLeague(int year, String season)
45      throws ObjectNotFoundException {
46
47      // Search in the cache.
48      Iterator set = LEAGUES_CACHE.iterator();
49      while ( set.hasNext() ) {
50          League l = (League) set.next();
51          if ( season.equals(l.getSeason()) && (year == l.getYear()) ) {
52              return l;
53          }
54      }
55
56      // Throw an exception if the league was not found.
57      throw new ObjectNotFoundException();
58  }
59
```



## The LeagueService Code (Part 4)

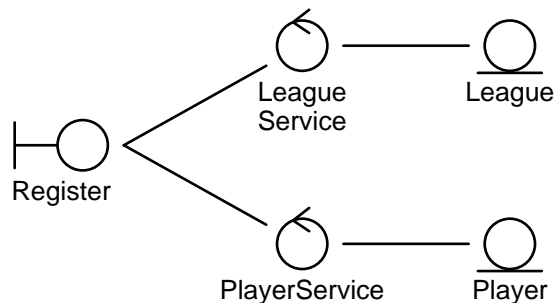
```
59
60  /**
61   * This method adds a new League object.
62   */
63  public League createLeague(int year, String season, String title) {
64
65      // Determine the next league objectID
66      int nextID = LEAGUES_CACHE.size() + 1;
67
68      // Create new league object
69      League league = new League(nextID, year, season, title);
70
71      // Store the league object
72      storeLeague(league);
73
74      // Record the league in the cache for easy retrieval
75      LEAGUES_CACHE.add(league);
76
77      return league;
78  }
79
```



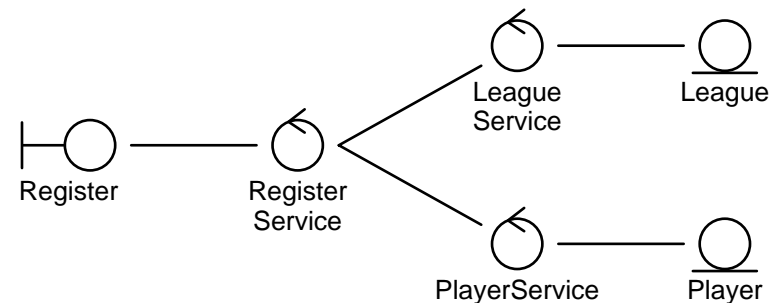
# Façade Service

A façade service might be used to reduce coupling between boundary components and other services.

## High Coupling



## Low Coupling







## The RegisterService Code

```
1 package sl314.model;
2
3 import java.io.File;
4 import java.io.FileWriter;
5 import java.io.PrintWriter;
6 import java.io.IOException;
7
8 /**
9  * This object performs a variety of league registration services.
10 * It acts a Facade into the business logic of registering a Player for
11 * a League.
12 */
13 public class RegisterService {
14
15     public RegisterService() {
16         // do nothing
17     }
18 }
```



## The RegisterService Code (Part 2)

```
18
19  /**
20   * This method finds the specified league, by delegating to the
21   * LeagueService object.
22   */
23  public League getLeague(int year, String season)
24      throws ObjectNotFoundException {
25      LeagueService leagueSvc = new LeagueService();
26      return leagueSvc.getLeague(year, season);
27  }
28
29  /**
30   * This method return a Player object for the named person, by
31   * delegating to the PlayerService object.
32   */
33  public Player getPlayer(String name) {
34      PlayerService playerSvc = new PlayerService();
35      return playerSvc.getPlayer(name);
36  }
```



## The RegisterService Code (Part 3)

```
37
38  /**
39   * This method stores the registration information for the player,
40   * based on the league and division information.
41   */
42  public void register(League league, Player player, String division) {
43
44      // Use the player service to save the player object
45      PlayerService playerSvc = new PlayerService();
46      playerSvc.save(player);
47
48      // Record the registration
49      insertRegistration(league, player, division);
50  }
51
```



## Summary

- An Analysis model bridges the gap between analysis (of use cases) and design (of application components).
- Boundary components have two aspects: views and controllers.
- Entity components represent real world business objects.
- Service components provide functional services to the boundary components for manipulating entities.



# Module 7

## Developing Web Applications Using Struts



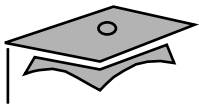
# Objectives

- Design a web application using the Struts MVC framework
- Develop a Struts action class
- Configure the Struts action mappings

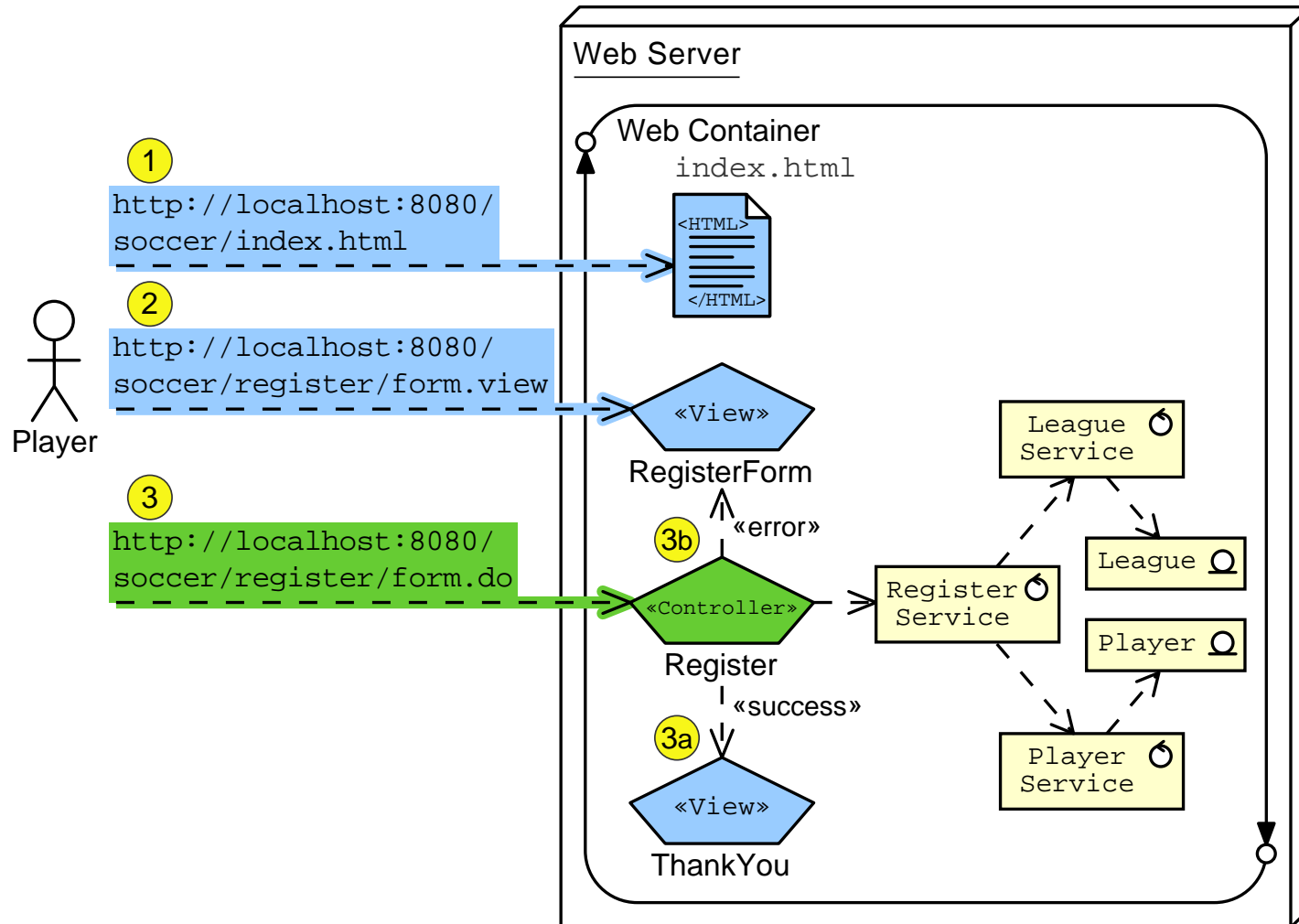


## Relevance

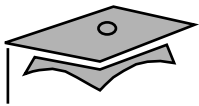
- What types of application components have you seen so far in this class?
- How many servlets are required in the web application architecture you have seen so far in this class?



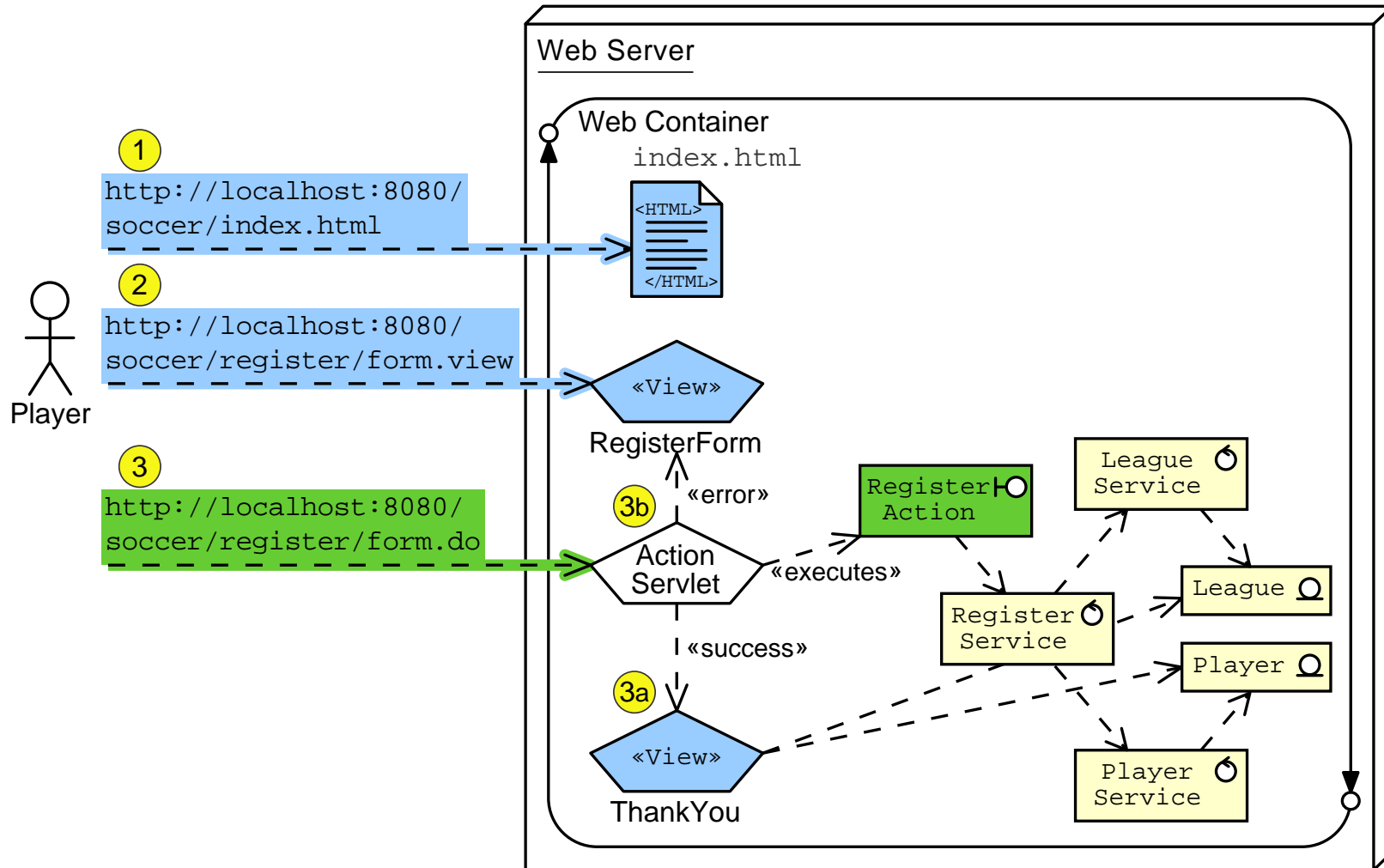
# Model-View-Controller Pattern





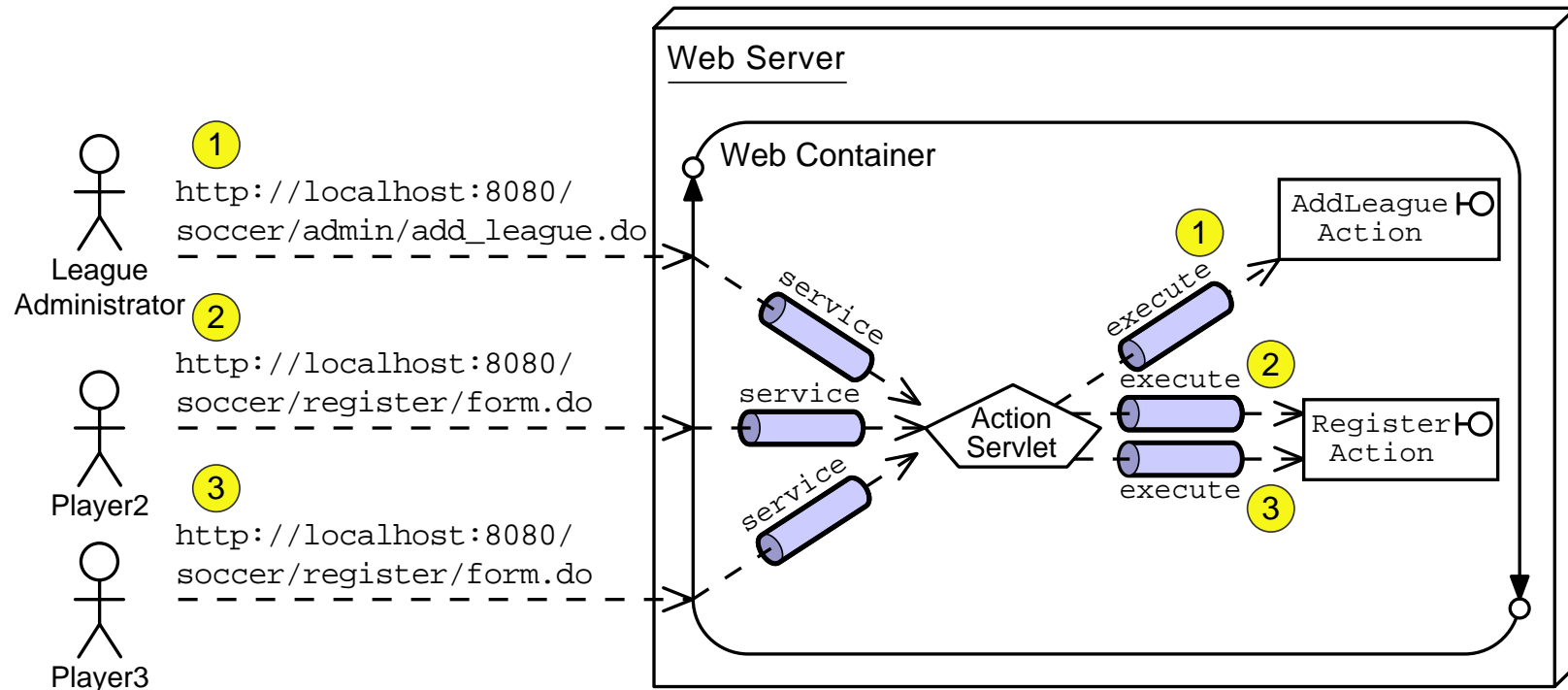


# Struts MVC Framework





# Front Controller Pattern



Controller requests are handled by the Struts ActionServlet, which acts as an infrastructure controller to dispatch to the application controller actions.

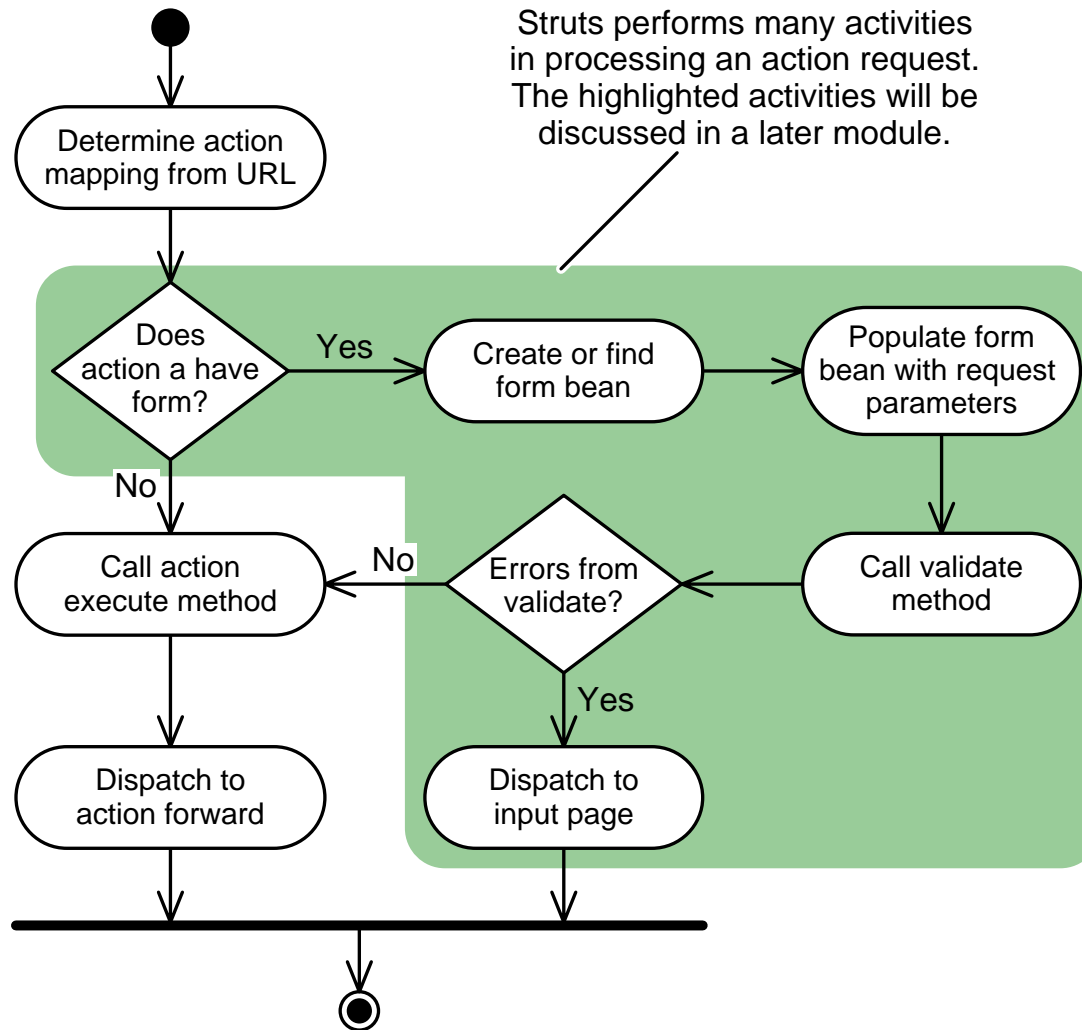


## Struts MVC Framework

- Framework provides the following elements:
  - Infrastructure servlet controller
  - Base classes
  - Configuration files
- Why use a framework like Struts?
  - Provides flexible, extensible infrastructure for MVC
  - Lets you focus on what is important to your application, such as:
    - Application controllers
    - Model components
    - Views



# Struts Activity Diagram



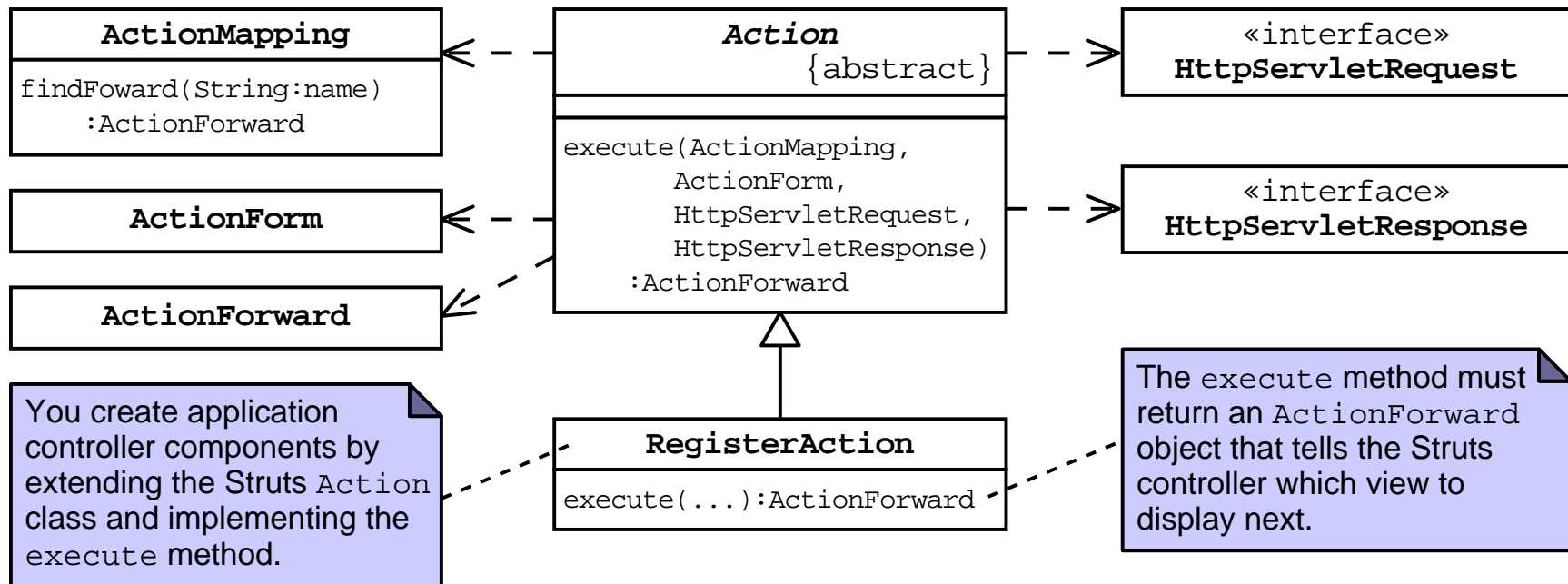


## Objectives

- Design a web application using the Struts MVC framework
- **Develop a Struts action class**
- Configure the Struts action mappings



# Struts Action Class





## The AddLeagueAction Code

```
1  package sl314.controller;
2
3  import javax.servlet.http.HttpServletRequest;
4  import javax.servlet.http.HttpServletResponse;
5  // Struts classes
6  import org.apache.struts.action.Action;
7  import org.apache.struts.action.ActionForward;
8  import org.apache.struts.action.ActionMapping;
9  import org.apache.struts.action.ActionForm;
10 // Model classes
11 import sl314.model.LeagueService;
12 import sl314.model.League;
13 import java.util.Set;
14 import java.util.List;
15 import java.util.LinkedList;
16
17
18 public class AddLeagueAction extends Action {
19
```



## The AddLeagueAction Code (Part 2)

```
17
18 public class AddLeagueAction extends Action {
19
20     public ActionForward execute(ActionMapping mapping,
21                                 ActionForm form,
22                                 HttpServletRequest request,
23                                 HttpServletResponse response) {
24
25         // Keep a set of strings to record form processing errors.
26         List errorMsgs = new LinkedList();
27         // Store this set in the request scope, in case we need to
28         // send the ErrorPage view.
29         request.setAttribute("errorMsgs", errorMsgs);
30
31         try {
32
33             // Retrieve form parameters.
34             String yearStr = request.getParameter("year").trim();
35             String season = request.getParameter("season").trim();
36             String title = request.getParameter("title").trim();
37
```





## The AddLeagueAction Code (Part 3)

```
45
46     // Verify form parameters
47     if ( (year != -1) && ((year < 2000) || (year > 2010)) ) {
48         errorMsgs.add("The 'year' field must within 2000 to 2010.");
49     }
50     if ( season.equals("UNKNOWN") ) {
51         errorMsgs.add("Please select a league season.");
52     }
53     if ( title.length() == 0 ) {
54         errorMsgs.add("Please enter the title of the league.");
55     }
56
57     // Send the ErrorPage view if there were errors
58     if ( ! errorMsgs.isEmpty() ) {
59         return mapping.findForward("error");
60     }
61
```



## The AddLeagueAction Code (Part 4)

```
61
62     // Perform business logic
63     LeagueService leagueSvc = new LeagueService();
64     League league = leagueSvc.createLeague(year, season, title);
65     // Store the new league in the request-scope
66     request.setAttribute("league", league);
67
68     // Send the Success view
69     return mapping.findForward("success");
70
71     // Handle any unusual exceptions
72     } catch (RuntimeException e) {
73
74         // Log stack trace
75         e.printStackTrace(System.err);
76
77         // Send the ErrorPage view if there were errors
78         errorMsgs.add(e.getMessage());
79         return mapping.findForward("error");
80
81     } // END of try-catch block
```



## Objectives

- Design a web application using the Struts MVC framework
- Develop a Struts action class
- **Configure the Struts action mappings**



# Configuring the Struts Action Mappings

You need to do the following:

1. Configure the Struts infrastructure controller.
2. Configure a servlet mapping for the Struts controller.
3. Configure the action mappings.
4. Install the Struts library files.



# Configuring the Infrastructure Controller

Configured in the `web.xml` deployment descriptor:

```
24
25     <!-- Declare the Struts ActionServlet (Front Controller) -->
26     <servlet>
27         <servlet-name>FrontController</servlet-name>
28         <servlet-class>
29             org.apache.struts.action.ActionServlet
30         </servlet-class>
31         <!-- Path of the struts configuration file -->
32         <init-param>
33             <param-name>config</param-name>
34             <param-value>/WEB-INF/struts-config.xml</param-value>
35         </init-param>
36         <!-- Load the servlet on startup -->
37         <load-on-startup>1</load-on-startup>
38     </servlet>
39
```



## Front Controller Servlet Mapping

Also, configured in the `web.xml` deployment descriptor:

```
79
80     <!-- Standard Front Controller Mapping -->
81     <servlet-mapping>
82         <servlet-name>FrontController</servlet-name>
83         <url-pattern>*.do</url-pattern>
84     </servlet-mapping>
85
```

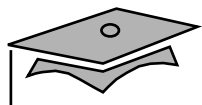
This servlet mapping ensures that all `*.do` requests go to the Struts infrastructure controller.



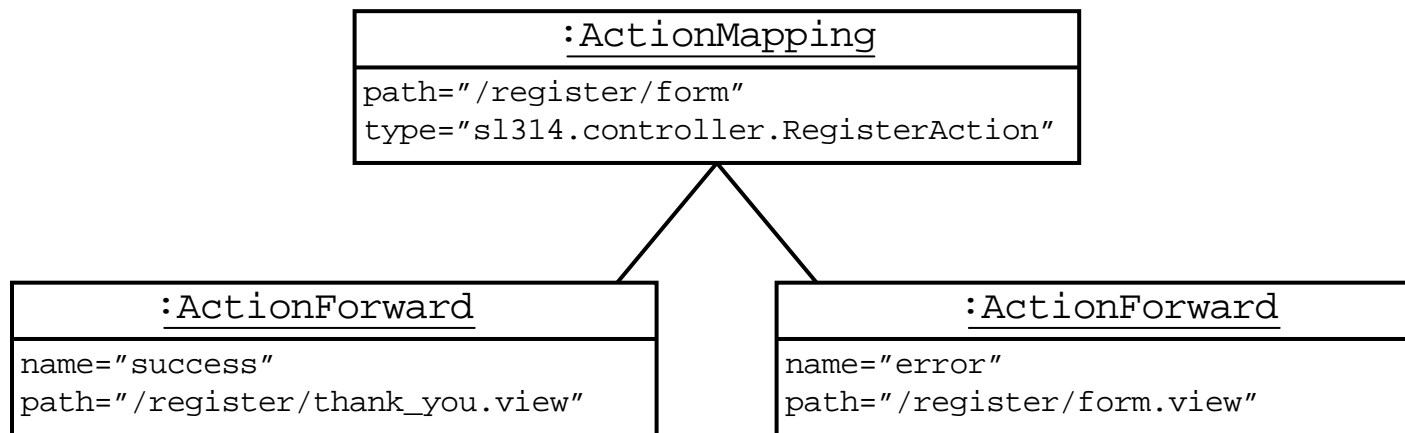
# Configuring Action Mappings

Configured in the `struts-config.xml` file:

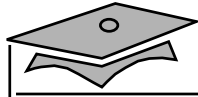
```
9      <action-mappings>
10
11      <!-- Declare the /register/form.do action -->
12      <action path="/register/form"
13              type="sl314.controller.RegisterAction">
14          <forward name="success" path="/register/thank_you.view"/>
15          <forward name="error" path="/register/form.view"/>
16      </action>
17
18      <!-- Declare the /admin/add_league.do action -->
19      <action path="/admin/add_league"
20              type="sl314.controller.AddLeagueAction">
21          <forward name="success" path="/admin/success.view"/>
22          <forward name="error" path="/admin/add_league.view"/>
23      </action>
24
25  </action-mappings>
```



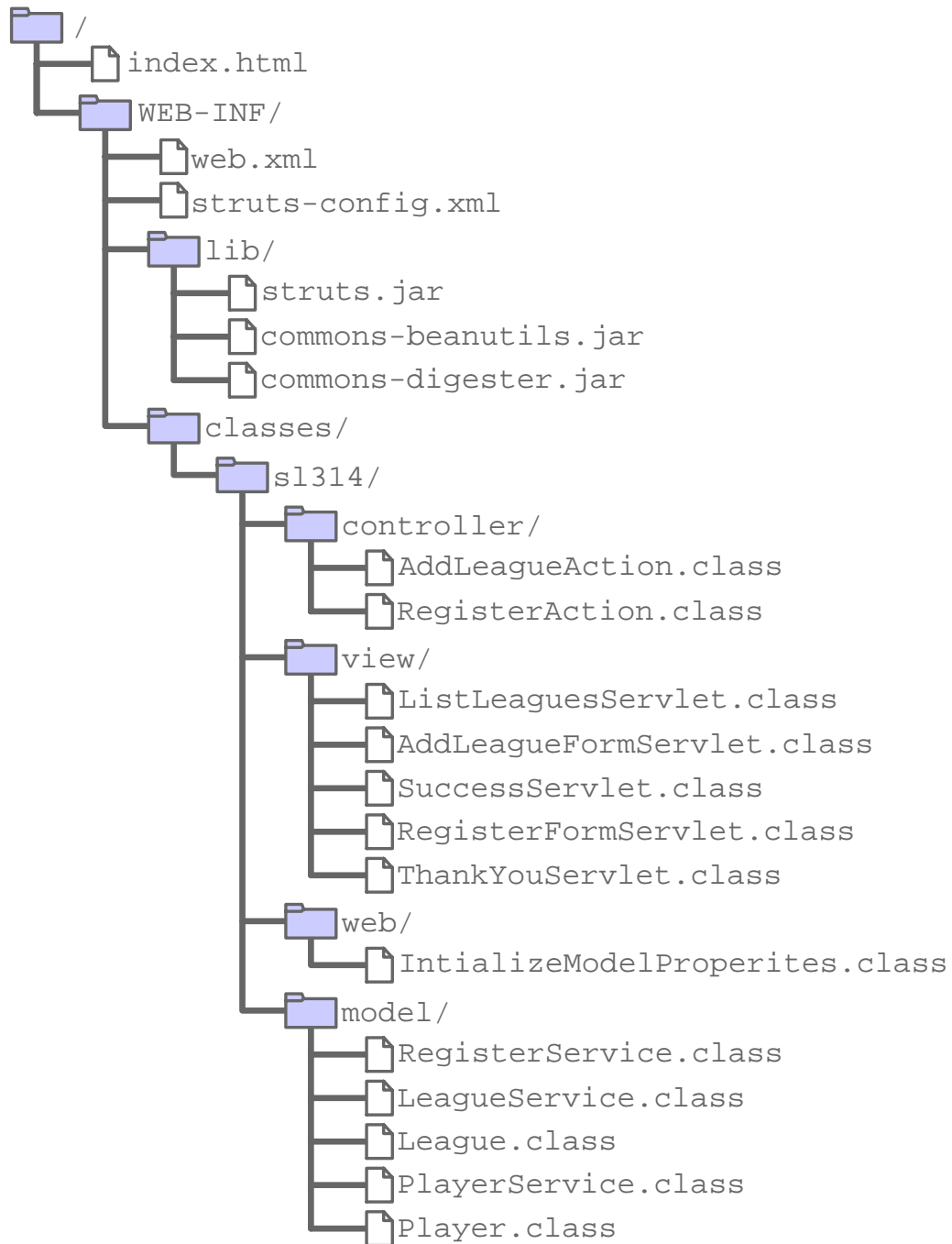
# Action Mapping Object Representation







# Physical Web Hierarchy





## Summary

- Struts is a framework that provides an implementation of the Front Controller pattern and supports the development of MVC-based web applications.
- Using Struts, you create a subclass of `Action` for each application controller.
- You can then configure the set of actions and their *forwards* in the `struts-config.xml` file.
- You also need to configure the Struts infrastructure controller servlet in the `web.xml` file.
- Finally, Struts is a big framework. In this module, you were introduced only to the essential aspects of Struts.



# 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



## Relevance

- What mechanism do you currently use for maintaining communications across requests?
- How much additional development is needed to use that communication mechanism?

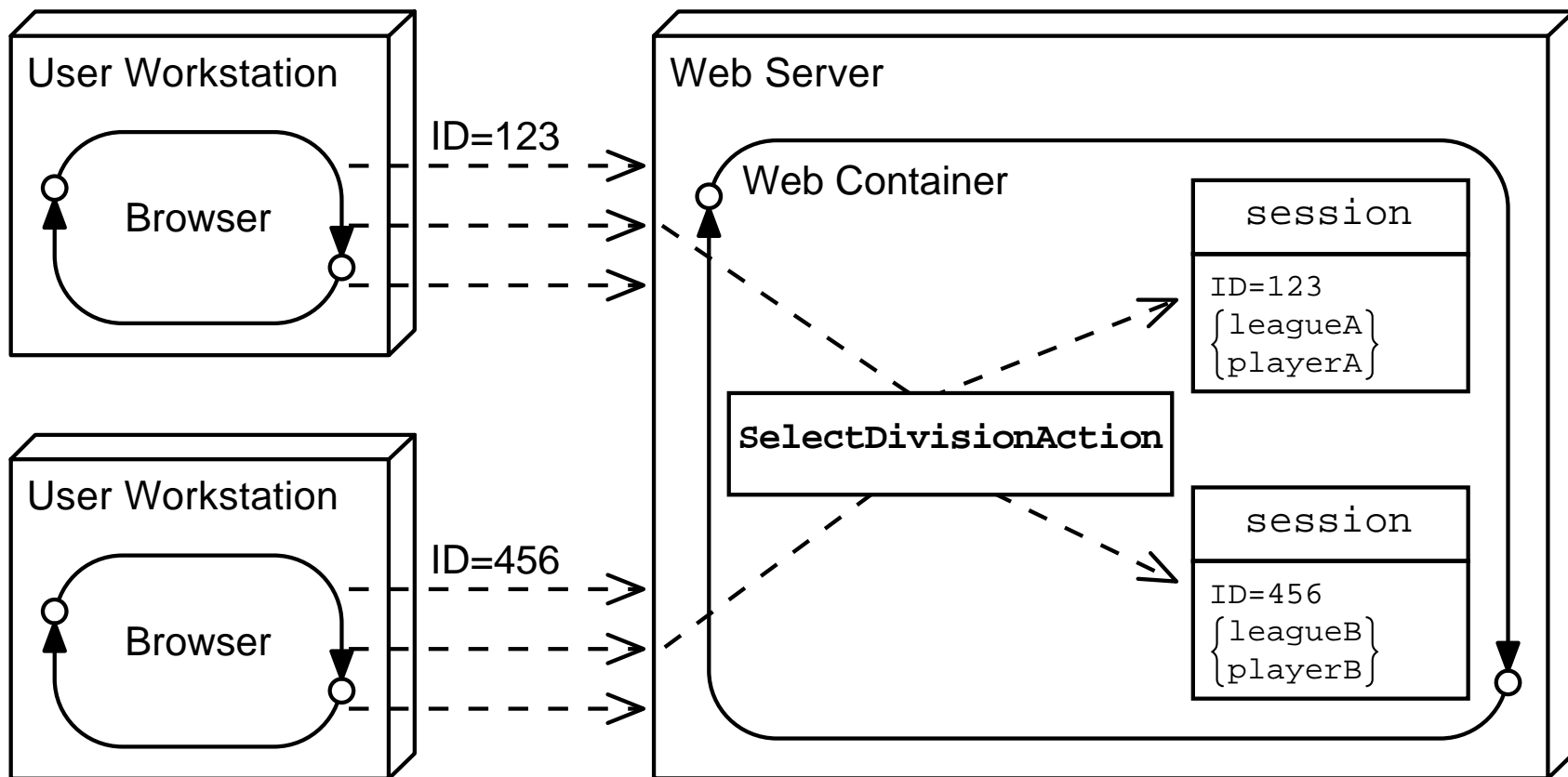


# HTTP and Session Management

HTTP is a stateless protocol. Each request and response message connection is independent of all others. Therefore, the web container must create a mechanism to store session information for a particular user.

# Web Container Sessions

The web container can keep a session object for each user:





## 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





# Designing Web Applications

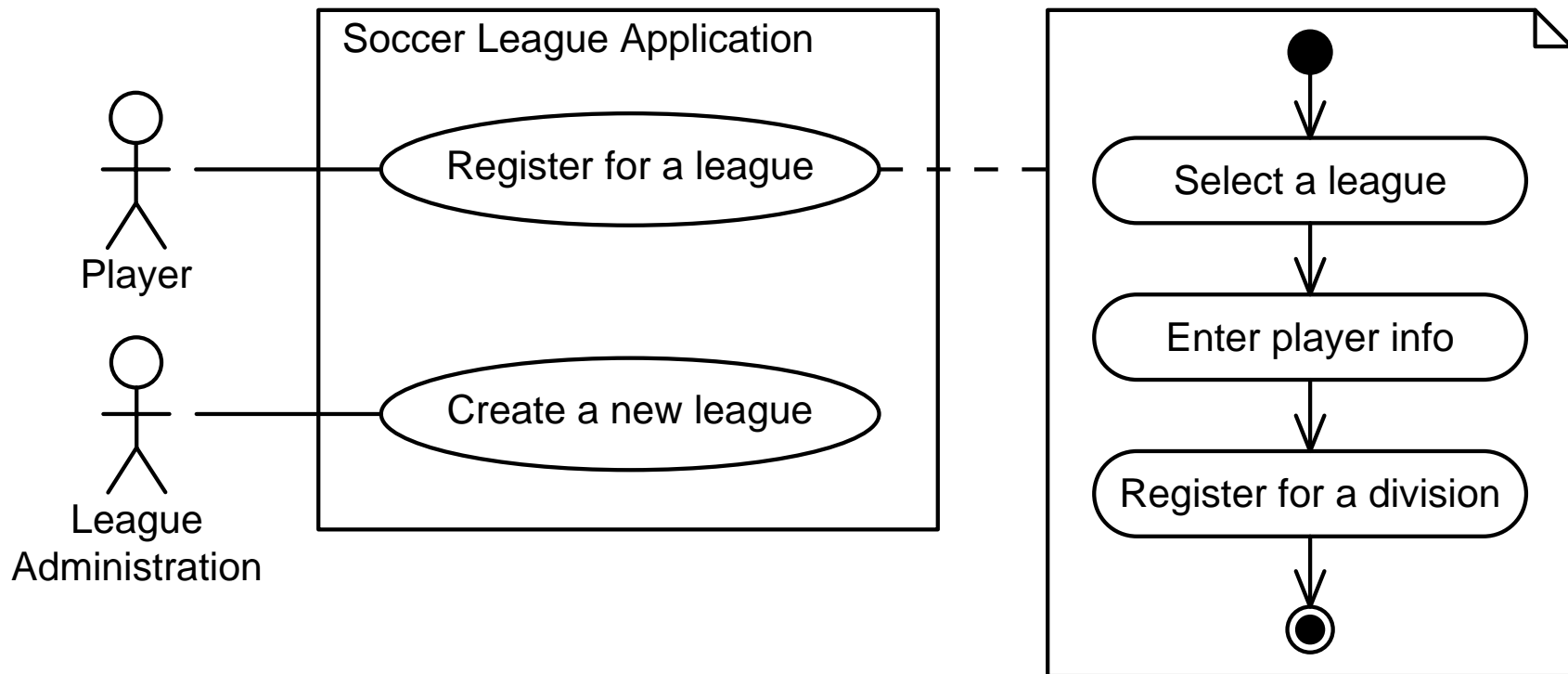
The following is just one technique for designing web applications using session management. There are three steps to this design process:

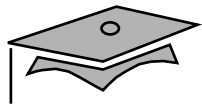
1. Design multiple, interacting views for a use case.
2. Create a Struts application controller for each activity in the use case.
3. Create a unique Struts URL for each activity in the use case.



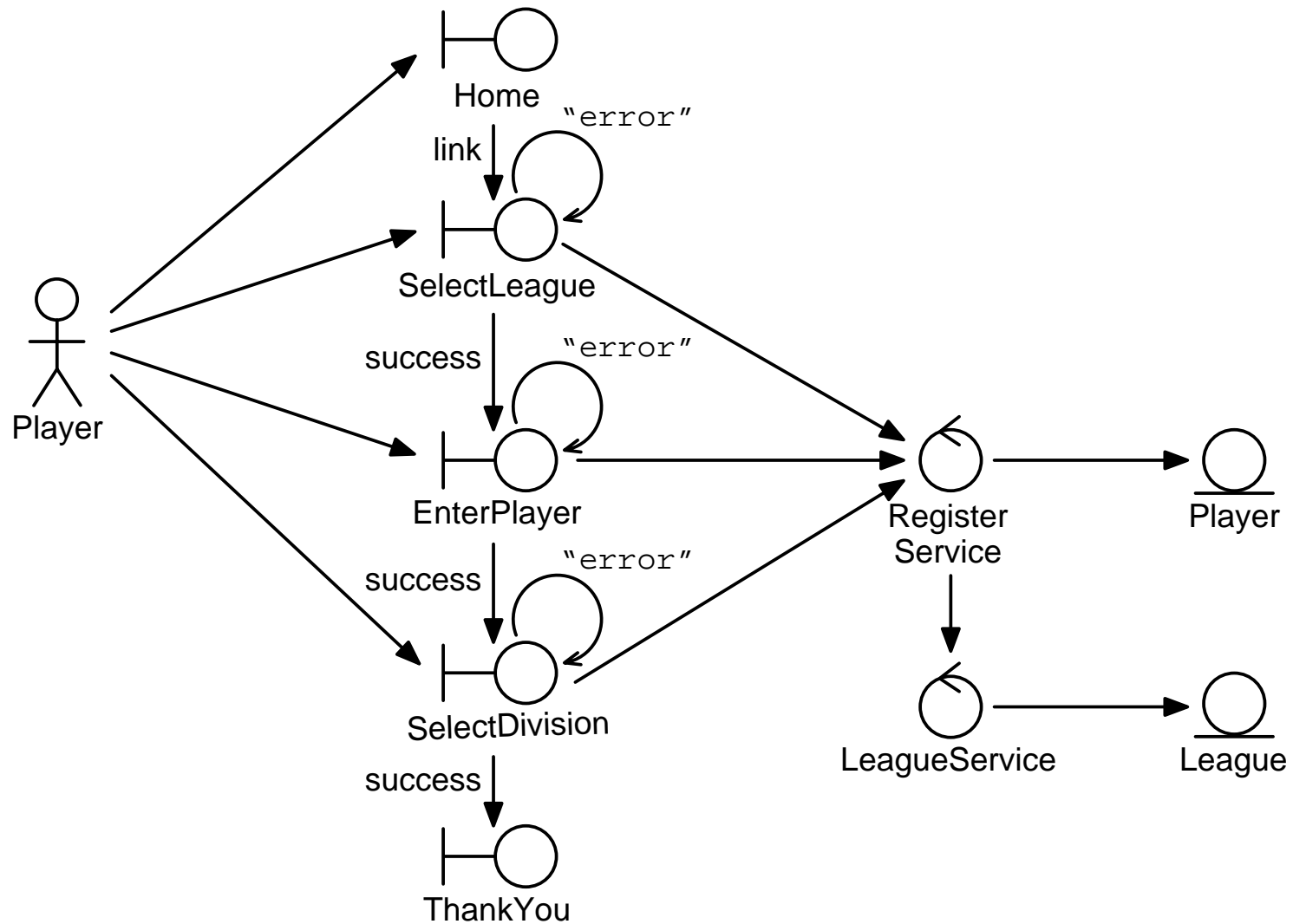
# Registration Use Case Example

The following is the use case for on-line league registration:





# Registration Use Case Analysis Model





## Objectives

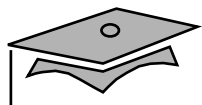
- Describe the purpose of session management
- Design a web application that uses session management
- **Develop servlets using session management**
- Describe the cookies implementation of session management
- Describe the URL-rewriting implementation of session management



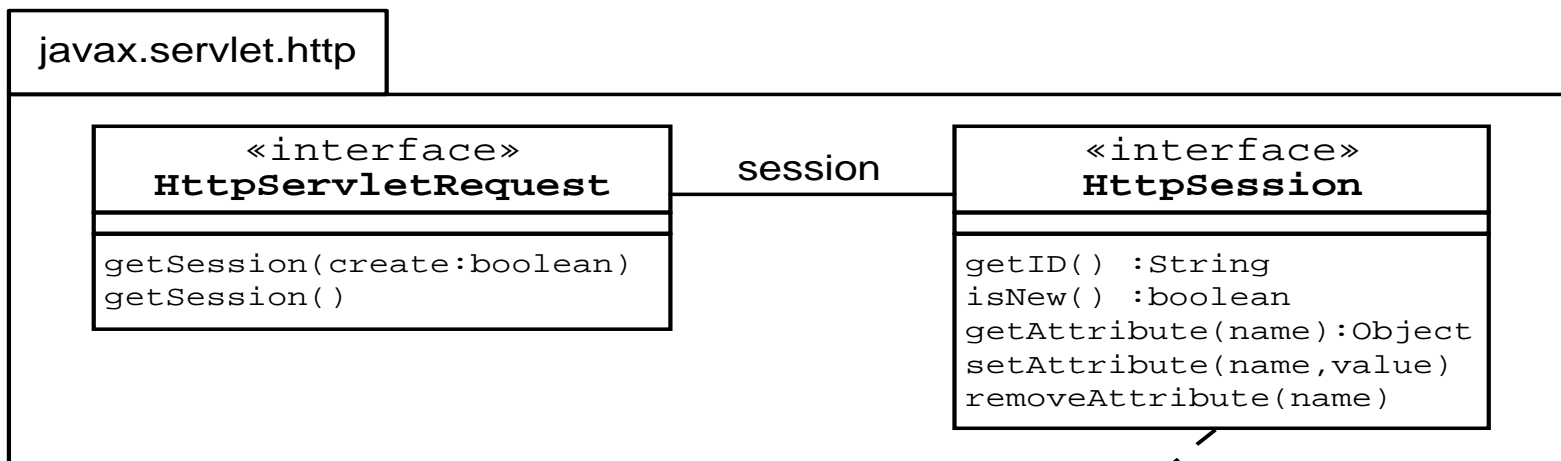
# Developing Web Applications

Using session management:

- Each activity-specific action must store attributes (name/object pairs) that are used by other requests within the session.
- Any action can access an attribute that has already been set by processing a previous request.
- At the end of the session, the action *might* destroy the session object.



# Session API



The session object can hold any number of objects using the `xyzAttribute` methods.

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



## Storing Session Attributes

```
58     // Perform business logic
59     RegisterService registerSvc = new RegisterService();
60
61     // Retrieve the league
62     League league = registerSvc.getLeague(year, season);
63
64     // Store the league object in the session-scope
65     HttpSession session = request.getSession();
66     session.setAttribute("league", league);
67
68     // Send the Success view
69     return mapping.findForward("success");
```

- Looks up the league object (line 62)
- Retrieves the session object (line 65)
- Stores it in the league attribute in the session (line 66)
- Directs the FrontController to the next view (line 69)

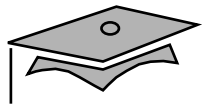


## Accessing Session Attributes

The `SelectDivisionAction` retrieves the league and player objects from the session:

```
47         // Retrieve the league and player objects from the session-scope
48         HttpSession session = request.getSession();
49         League league = (League) session.getAttribute("league");
50         Player player = (Player) session.getAttribute("player");
51
52         // Perform business logic
53         RegisterService registerSvc = new RegisterService();
54         // Perform the registration
55         registerSvc.register(league, player, division);
```





## Accessing Session Attributes

Views (such as the ThankYou component) might also:

- Access session attributes:

```
34
35     // Retrieve the 'league' and 'player' from the session-scope
36     HttpSession session = request.getSession();
37     League league = (League) session.getAttribute("league");
38     Player player = (Player) session.getAttribute("player");
39
```

- Generate a dynamic response using the attributes:

```
58
59     // Present the main body
60     out.println("<p>");
61     out.print("Thank you, " + player.getName() + ", for registering ");
62     out.println("for the <i>" + league.getTitle() + "</i> league.");
63     out.println("</p>");
64
```



## Destroying the Session

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

```
129 <session-config>
130   <session-timeout>10</session-timeout>
131 </session-config>
```

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

<code>&lt;&lt;interface&gt;&gt;</code> <b>HttpSession</b>
<code>invalidate()</code> <code>getCreationTime() :long</code> <code>getLastAccessedTime() :long</code> <code>getMaxInactiveInterval() :int</code> <code>setMaxInactiveInterval(int)</code>



## Destroying the Session

- Session objects can be shared across multiple actions (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).



## Objectives

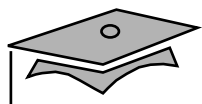
- Describe the purpose of session management
- Design a web application that uses session management
- Develop servlets using session management
- Describe the cookies implementation of session management
- Describe the URL-rewriting implementation of session management



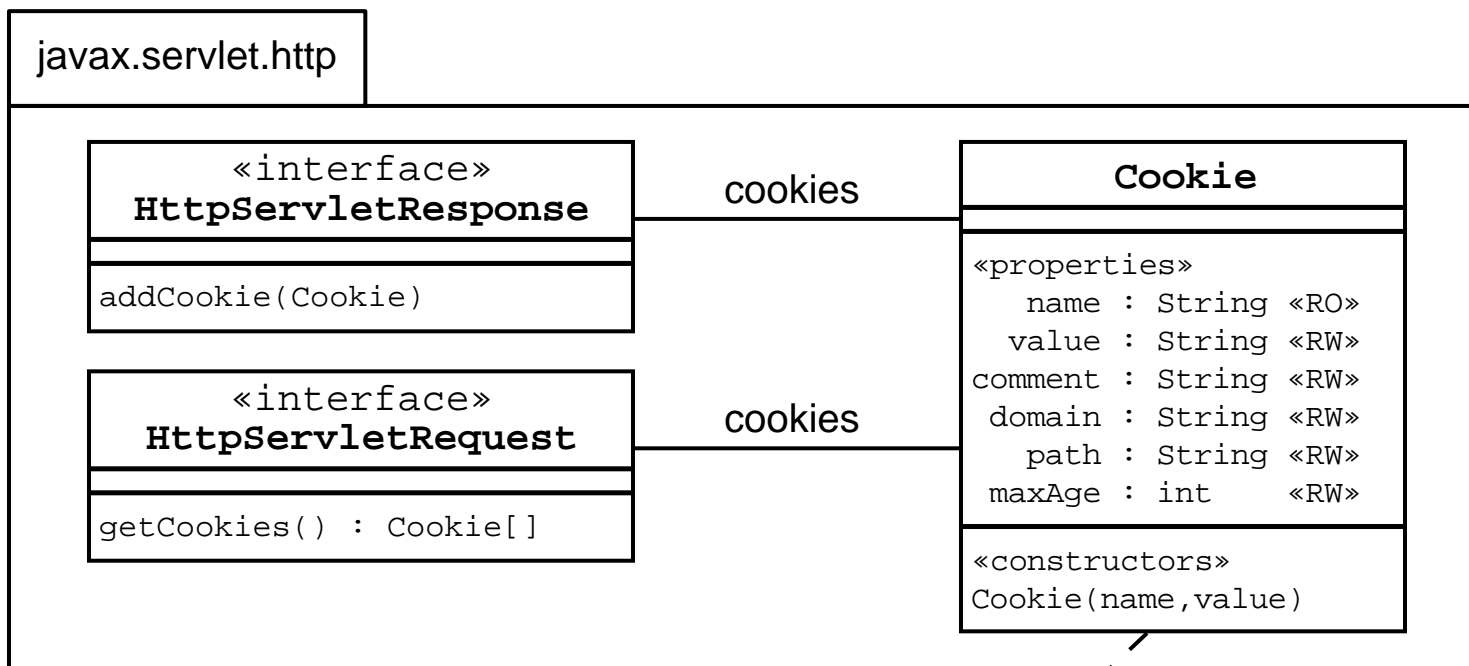
# Using Cookies for Session Management

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 are flushed by the client browser at the end of that lifespan.



# Cookie API



A Cookie object has accessors and mutators for each property.



## Using Cookies Example

- The code to store a cookie in the response:

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

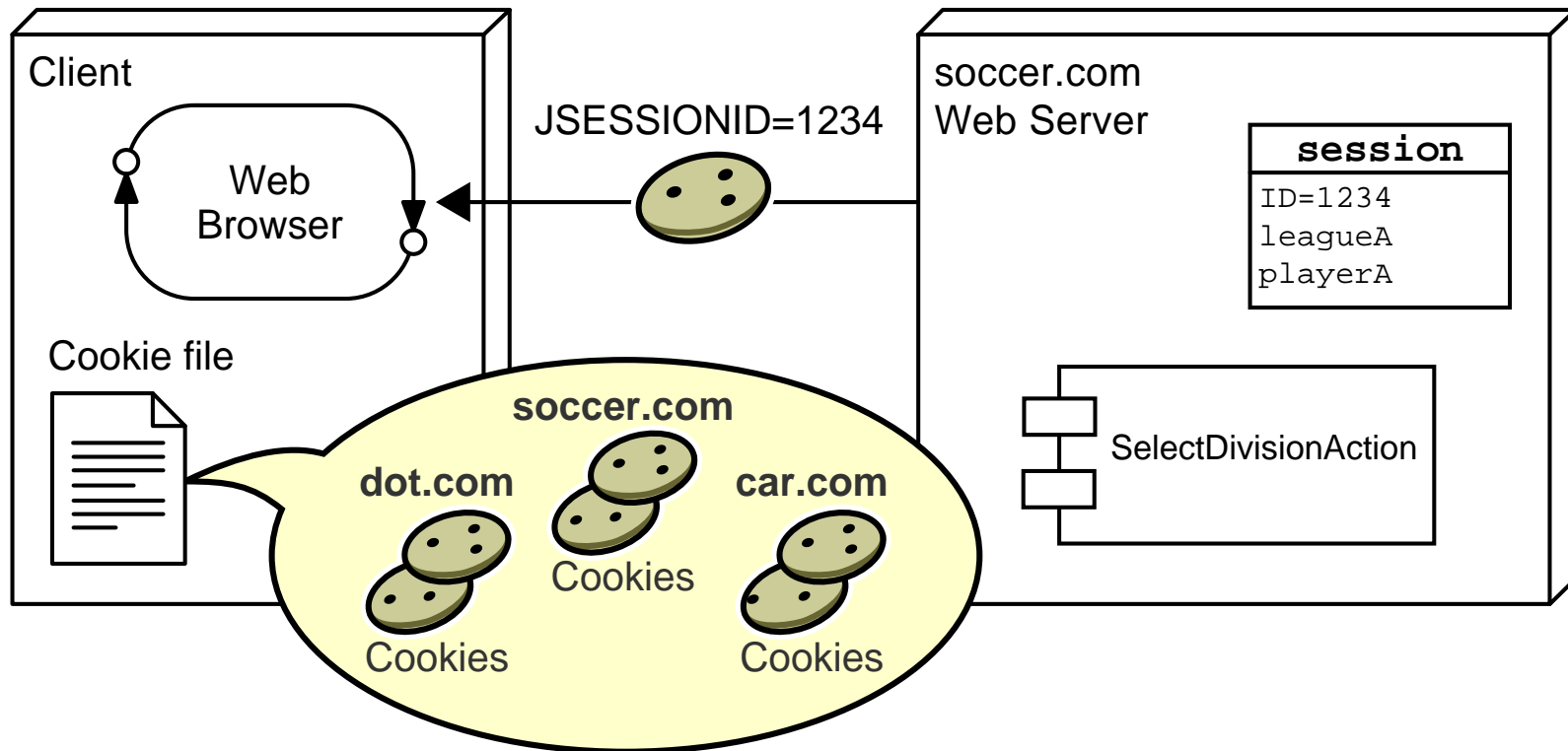
- The code to retrieve a cookie from the request:

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

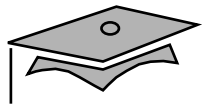


# Performing Session Management Using Cookies

The web container sends a JSESSIONID Cookie to the client:

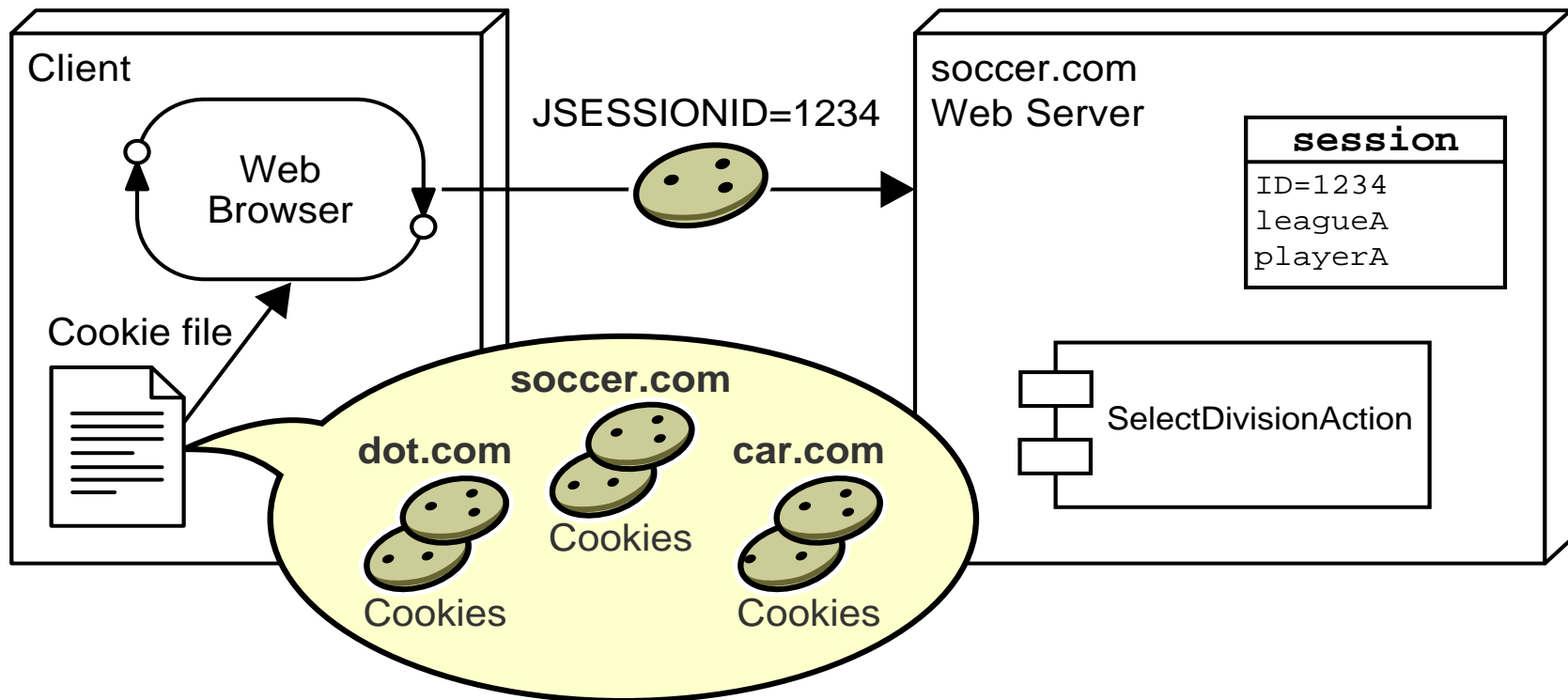






# Performing Session Management Using Cookies

The JSESSIONID cookie is sent in all subsequent requests:





# Performing Session Management Using Cookies

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



## 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

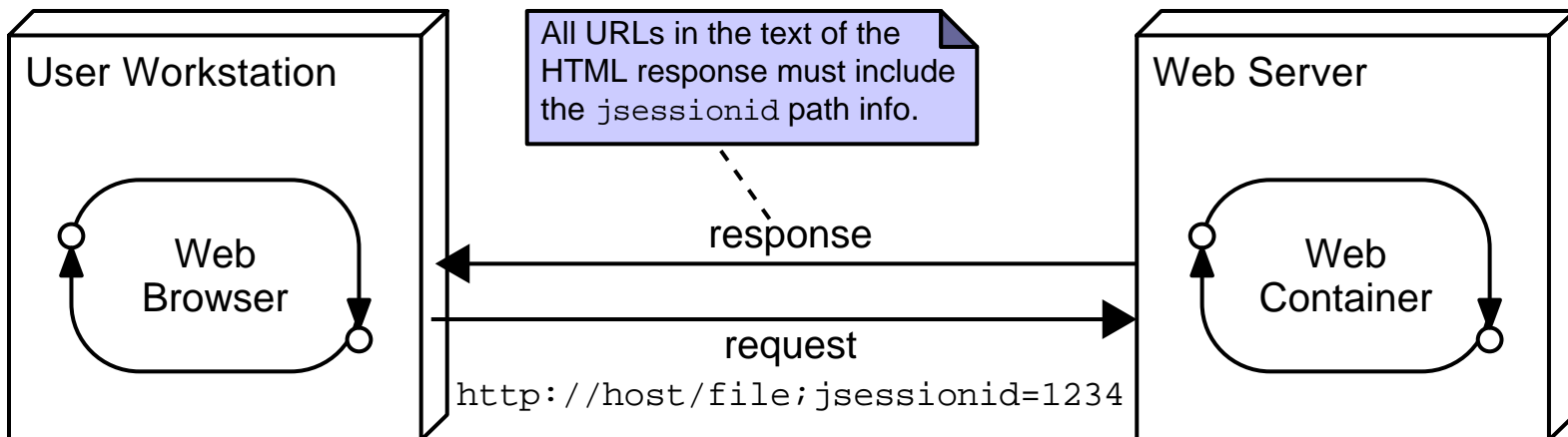


## Using URL-Rewriting for Session Management

- URL-rewriting is used when Cookies cannot be used.
- The server appends extra data on the end of each URL.
- The 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`.



# Using URL-Rewriting for Session Management





## URL-Rewriting Implications

- 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 `EnterPlayerForm` view the `action` attribute on the form tag must be encoded:

```
86
87 // Present the form
88 out.println("<form action=' "
89           + response.encodeURL("enter_player.do")
90           + "' method='POST'>");
91
```



## Summary

- Use cases that must share data across multiple HTTP requests require session management.
- The web container supplies a session management mechanism because HTTP is a stateless protocol.
- A web application can store and retrieve session-scoped data in the `HttpSession` object which is retrieved from the request object.
- The default session management mechanism uses HTTP cookies.
- Web containers must also support URL-rewriting for session management when the client has cookies turned off.



# Module 9

## Using Filters in Web Applications





# Objectives

- Describe the web container request cycle
- Describe the Filter API
- Develop a filter class
- Configure a filter in the `web.xml` file



## Relevance

- What should you do if you want an operation to occur every time a particular request is made?
- What should you do if that operation must be performed on other requests in the web application?
- What should you do if you want to allow this operation to be turned off at deployment?



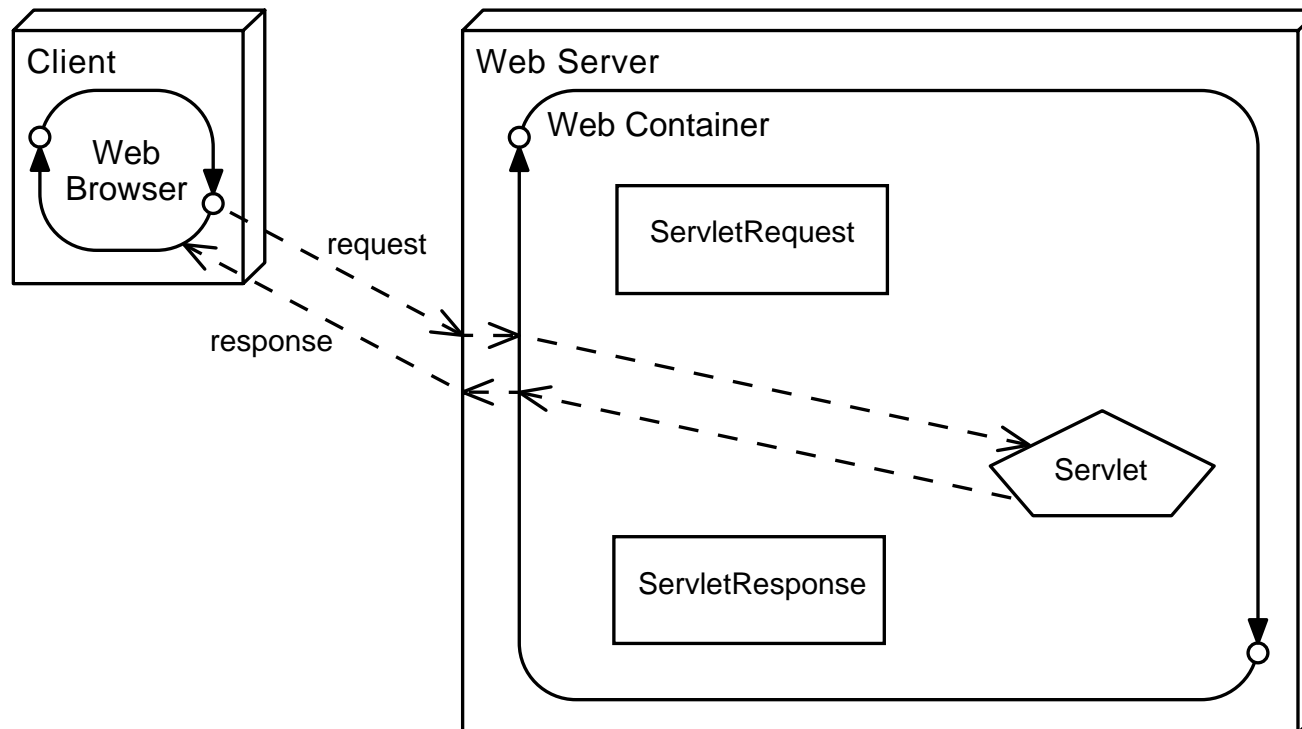
## Web Container Request Cycle

- Request processing by the web container
- Applying filters to an incoming request
- Applying filters to a dispatched request



# Web Container Request Processing

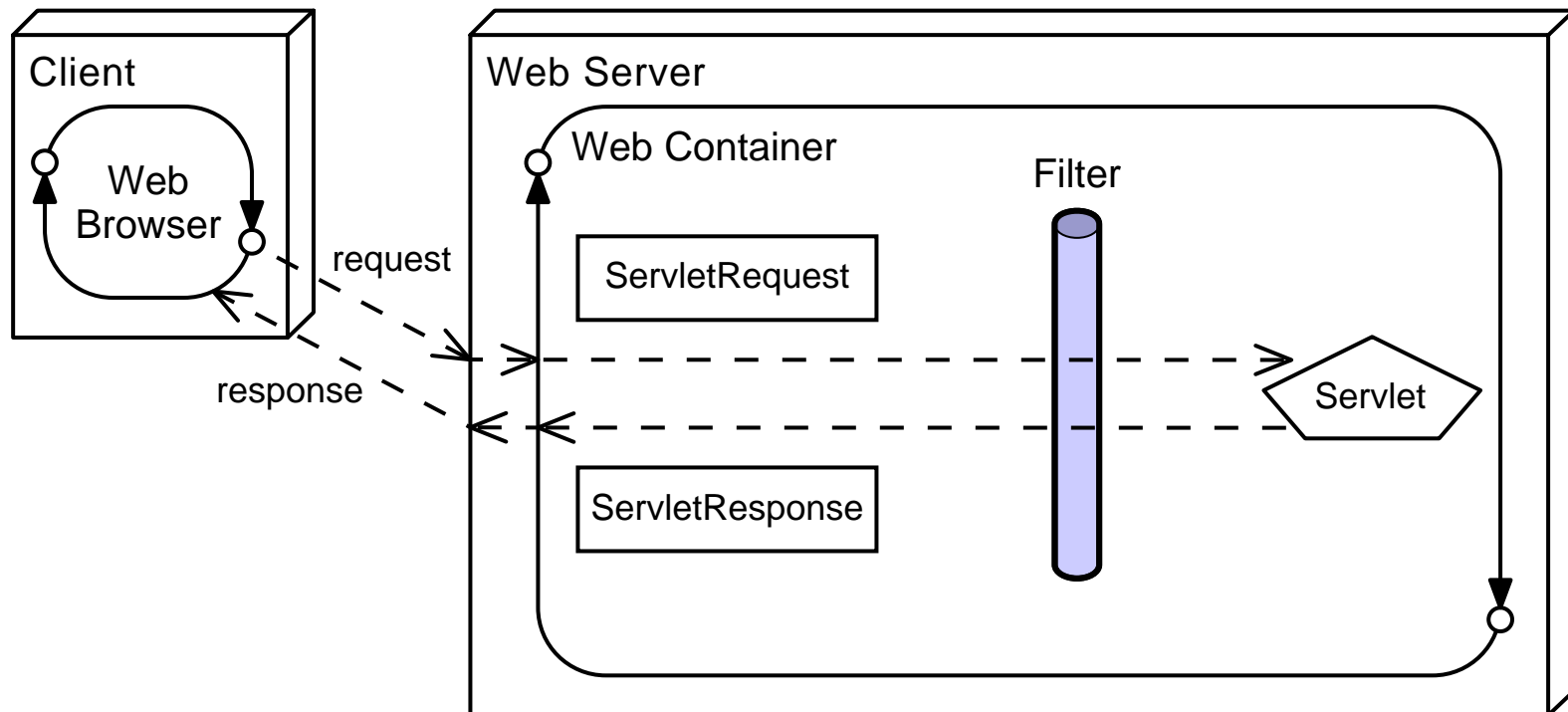
Request and response objects are created for each incoming request.





## Applying Filters to an Incoming Request

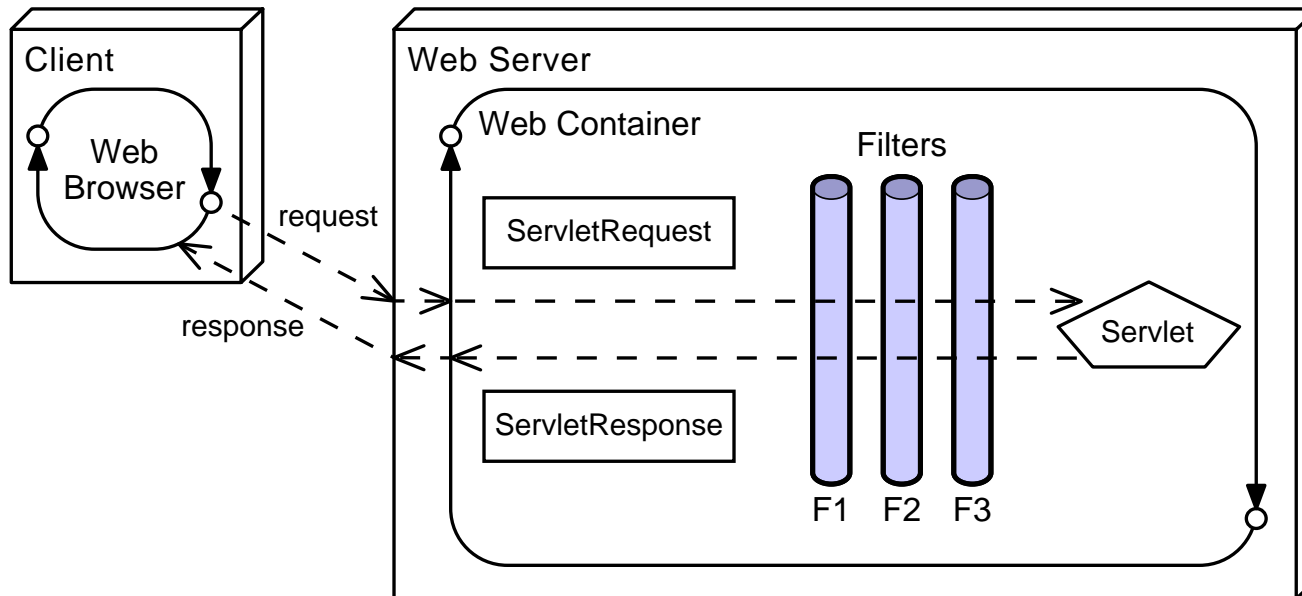
- A filter intercepts the request before it gets to the requested resource
- A response is returned to the client through the filter





# Applying Filters to an Incoming Request

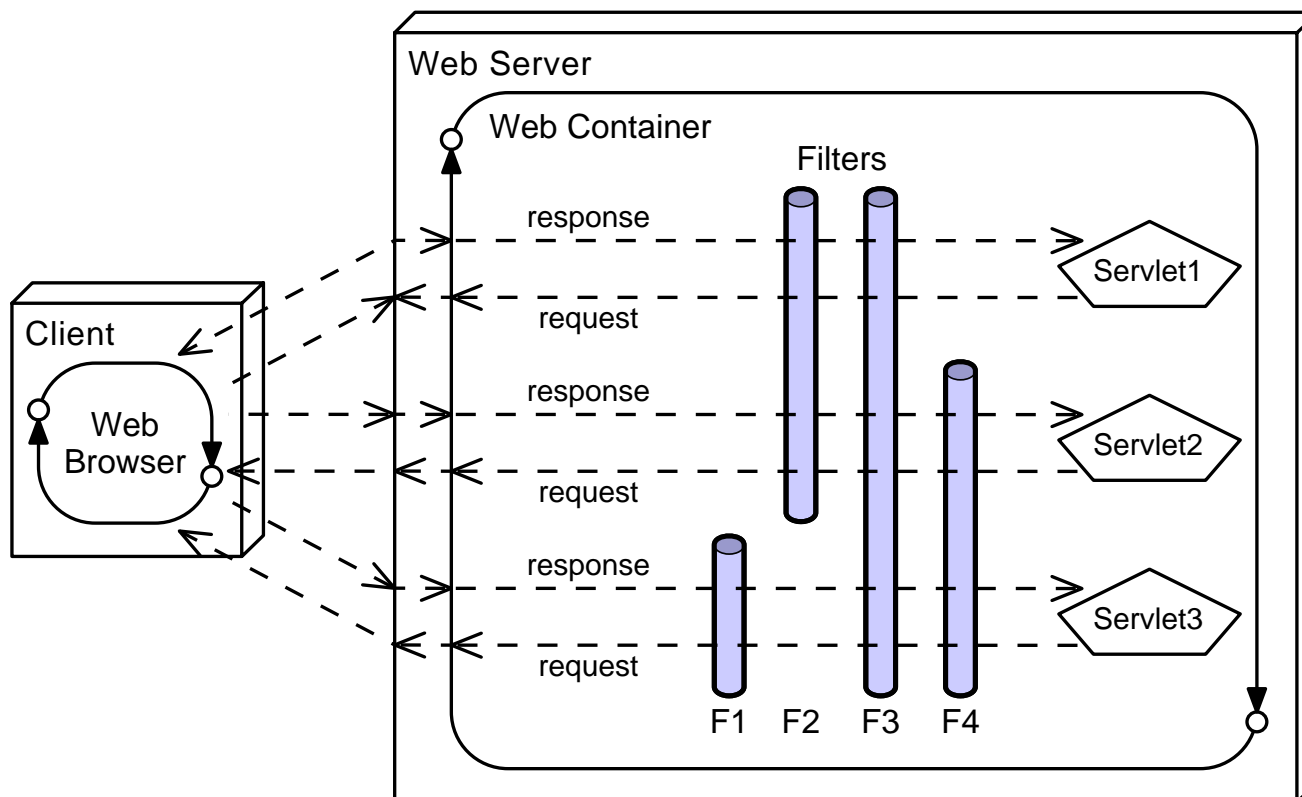
Multiple filters can intercept a given request.



This provides for modularity and reuse of code.

# Applying Filters to an Incoming Request

Filters can be applied to different requests in different combinations.



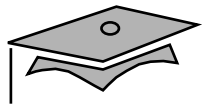


## Applying Filters to an Incoming Request

Filters can be used for many activities in a web application, such as:

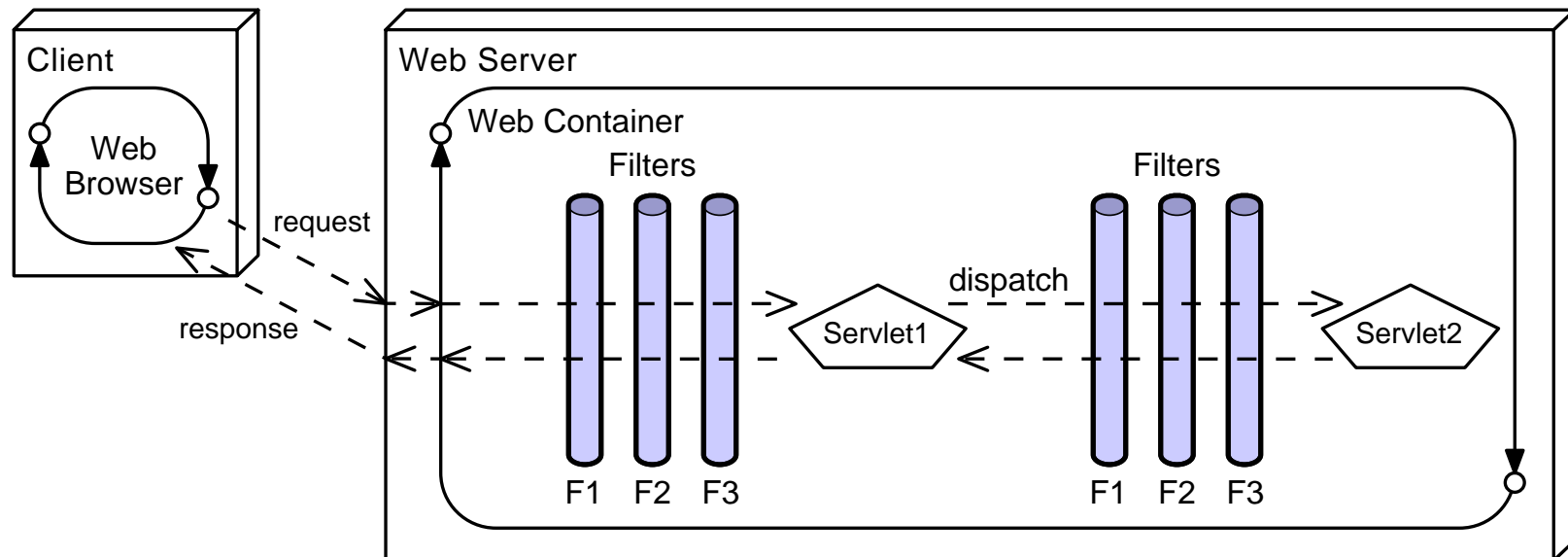
- Blocking access to a resource based on user identity or role membership
- Auditing incoming requests
- Compressing the response data stream
- Transforming the response
- Measuring and logging servlet performance





## Filters Applied to a Dispatch

Filters can be applied to an internal dispatch, such as a request forward or include.

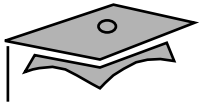


This behavior is determined by the information in the deployment descriptor.

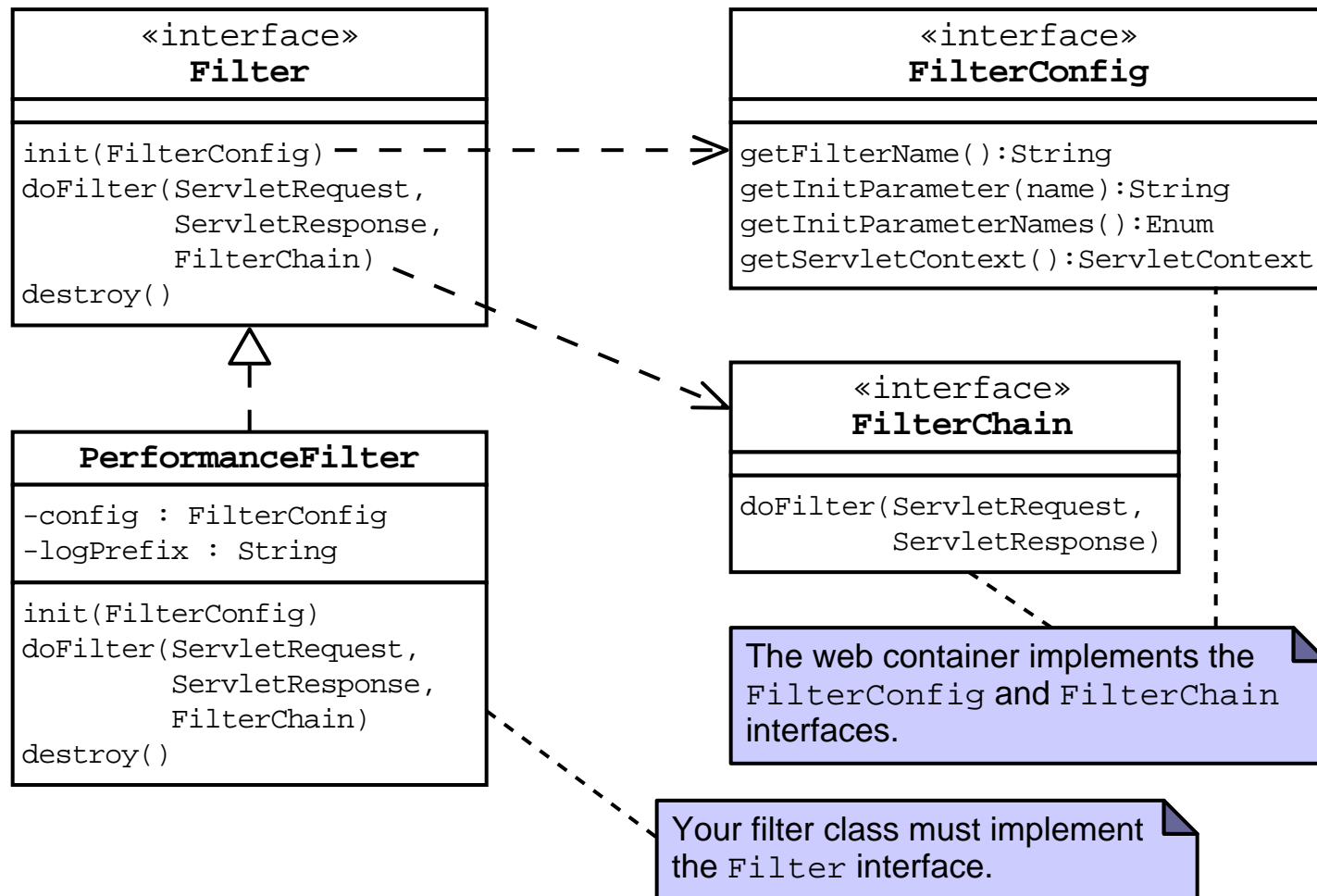


## Objectives

- Describe the web container request cycle
- Describe the Filter API
- Develop a filter class
- Configure a filter in the web.xml file



# Filter API





## Objectives

- Describe the web container request cycle
- Describe the Filter API
- **Develop a filter class**
- Configure a filter in the `web.xml` file



## The PerformanceFilter Class

```
1 package sl314.web;
2
3 import java.io.IOException;
4
5 import javax.servlet.ServletException;
6 import javax.servlet.ServletResponse;
7 import javax.servlet.ServletException;
8 import javax.servlet.http.HttpServletRequest;
9
10 import javax.servlet.Filter;
11 import javax.servlet.FilterChain;
12 import javax.servlet.FilterConfig;
13
14 public class PerformanceFilter implements Filter {
15
16     private FilterConfig config;
17     private String logPrefix;
18
```



## The `init` Method

The `init` method is called once when the filter instance is first created.

Use the `init` method to:

- Perform one-time initialization of resources the filter uses over its lifetime.
- Retrieve the initialization parameters configured in the deployment descriptor.

```
19 public void init(FilterConfig config)
20     throws ServletException {
21     this.config = config;
22     logPrefix = config.getInitParameter("Log Entry Prefix");
23 }
```



## The doFilter Method

- The filter equivalent of a servlet's `service` method
- As a developer, you implement the `doFilter` method to do the following:
  - Perform the operations you want to occur every time the filter is invoked.
  - Decide whether to pass the request to the next component in the filter chain or halt the request entirely.

To pass on the request, call the `doFilter` method on the `FilterChain` reference.



## The doFilter Method

```
24
25 public void doFilter(ServletRequest request,
26     ServletResponse response, FilterChain chain)
27     throws ServletException, IOException {
28
29     long begin = System.currentTimeMillis();
30     chain.doFilter(request, response);
31     long end = System.currentTimeMillis();
32
33     StringBuffer logMessage = new StringBuffer();
34     if (request instanceof HttpServletRequest) {
35         logMessage = ((HttpServletRequest)request).getRequestURL();
36     }
37     logMessage.append(": ");
38     logMessage.append(end - begin);
39     logMessage.append(" ms");
40
41     if(logPrefix != null) {
42         logMessage.insert(0,logPrefix);
43     }
44
45     config.getServletContext().log(logMessage.toString());
46 }
```





## The destroy Method

The destroy method is the last method called in the lifecycle of a filter instance.

Use the destroy method to clean up any resources allocated in the init method.

```
48     public void destroy() {  
49         config = null;  
50         logPrefix = null;  
51     }
```



## Objectives

- Describe the web container request cycle
- Describe the Filter API
- Develop a filter class
- Configure a filter in the `web.xml` file



## Configuring the Filter

- You declare the filter in the deployment descriptor
- You can supply initialization parameters in the declaration

```
25 <filter>
26   <filter-name>perfFilter</filter-name>
27   <filter-class>sl314.web.PerformanceFilter</filter-class>
28   <init-param>
29     <param-name>Log Entry Prefix</param-name>
30     <param-value>Performance: </param-value>
31   </init-param>
32 </filter>
```



## Configuring the Filter

- Mappings can be:
  - URL based – Use the exact URL or a wildcard (\*)
  - Servlet name-based – Specify the name of the servlet to which the filter is applied

```
34 <filter-mapping>  
35   <filter-name>perfFilter</filter-name>  
36   <url-pattern>*.do</url-pattern>  
37 </filter-mapping>
```

- For a given request, if multiple filter mappings match:
  - URL-based filters applied before servlet name-based filters
  - Filters applied in the order in which the mappings occur in the deployment descriptor



## Configuring the Filter

Given these servlet mappings, what happens if the client requests `/admin/add_league.do`?

```
<servlet-mapping>
  <servlet-name>FrontController</servlet-name>
  </url-pattern>*.do</url-pattern>
</servlet-mapping>
<filter-mapping>
  <filter-name>perfFilter</filter-name>
  <servlet-name>FrontController</servlet-name>
</filter-mapping>
<filter-mapping>
  <filter-name>auditFilter</filter-name>
  <url-pattern>*.do</url-pattern>
</filter-mapping>
<filter-mapping>
  <filter-name>transformFilter</filter-name>
  <url-pattern>*.do</url-pattern>
</filter-mapping>
```



## Configuring the Filter

Typically, filters are applied to requests from a client. You can specify the `dispatcher` element in a filter mapping. This determines what type (or types) of requests invoke the filter. Valid values are:

- `REQUEST` – The filter is applied if the request is from a client
- `INCLUDE` – The filter is applied if the request is from a request dispatcher include
- `FORWARD` – The filter is applied if the request is from a request dispatcher forward
- `ERROR` – The filter is applied if the request is a result of an error condition



## Configuring the Filter

You can use a combination of `dispatcher` elements to specify when filters should be applied.

Given:

```
<filter-mapping>
  <filter-name>auditFilter</filter-name>
  <url-pattern>*.do</url-pattern>
  <dispatcher>INCLUDE</dispatcher>
  <dispatcher>FORWARD</dispatcher>
</filter-mapping>
```

When would the `auditFilter` be applied?



## Summary

- Filters permit you to augment the default request processing model.
- You can create a filter as follows:
  - Implementing the `javax.servlet.Filter` interface
  - Configuring a filter instance in the deployment descriptor
  - Configuring one or more filter mappings
- Filters can also be applied to dispatched requests.





# Module 10

## Integrating Web Applications With Databases



## Objectives

- Understand what a database management system (DBMS) does
- Design a web application to integrate with a DBMS
- Configure a DataSource and Java Naming and Directory Interface™ (JNDI) API



## Relevance

- Have you ever developed an application that integrates with the resource tier? How did you develop the access logic to the RDBMS?
- Did you ever have to change the database design? How did that affect the various tiers in your application?



## 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



## JDBC™ API

- The JDBC™ API 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 are described for designing and developing a web application, in which the JDBC technology code is encapsulated using the Data Access Object design pattern.

An incorrect technique is to create a connection object for each request, but this approach is extremely slow and does not scale well.



## Objectives

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



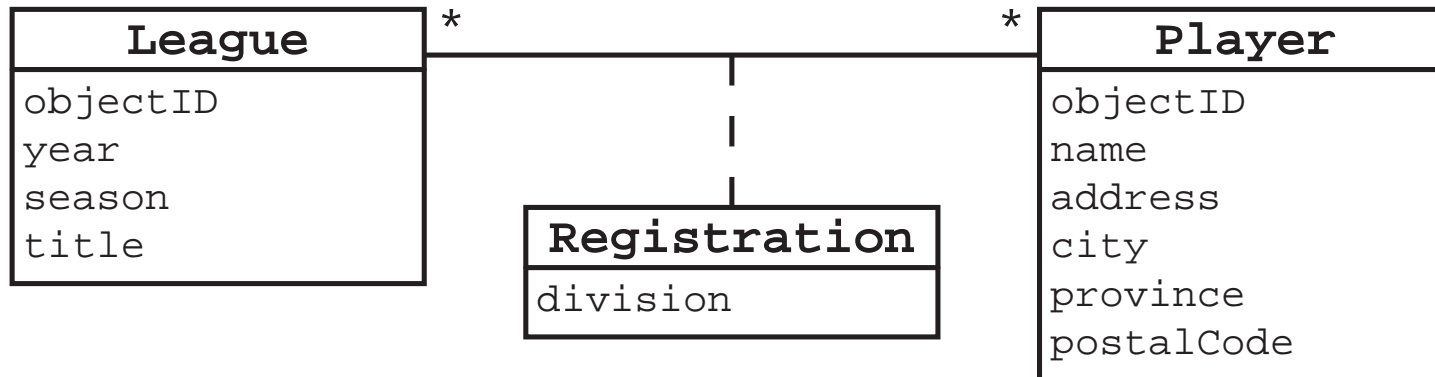
## Designing a Web Application

- 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 data access object (DAO) pattern



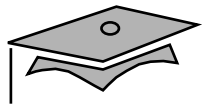
## Domain Objects

The following are the domain objects in Soccer League web application:



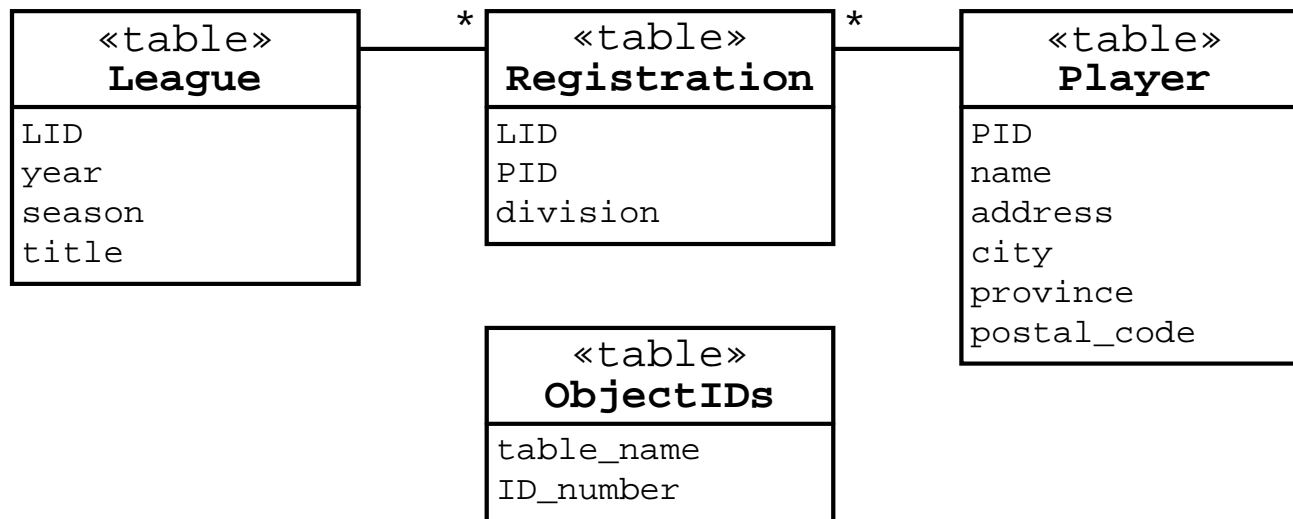
The objectID has been added to the classes to provide a unique ID in the database (DB) table for each of these entities.



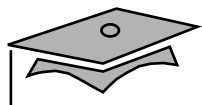


# Database Tables

The following is one possible DB design for the domain objects:



The objectID in the Java technology object corresponds to the ID in the database table. For example, the objectID 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



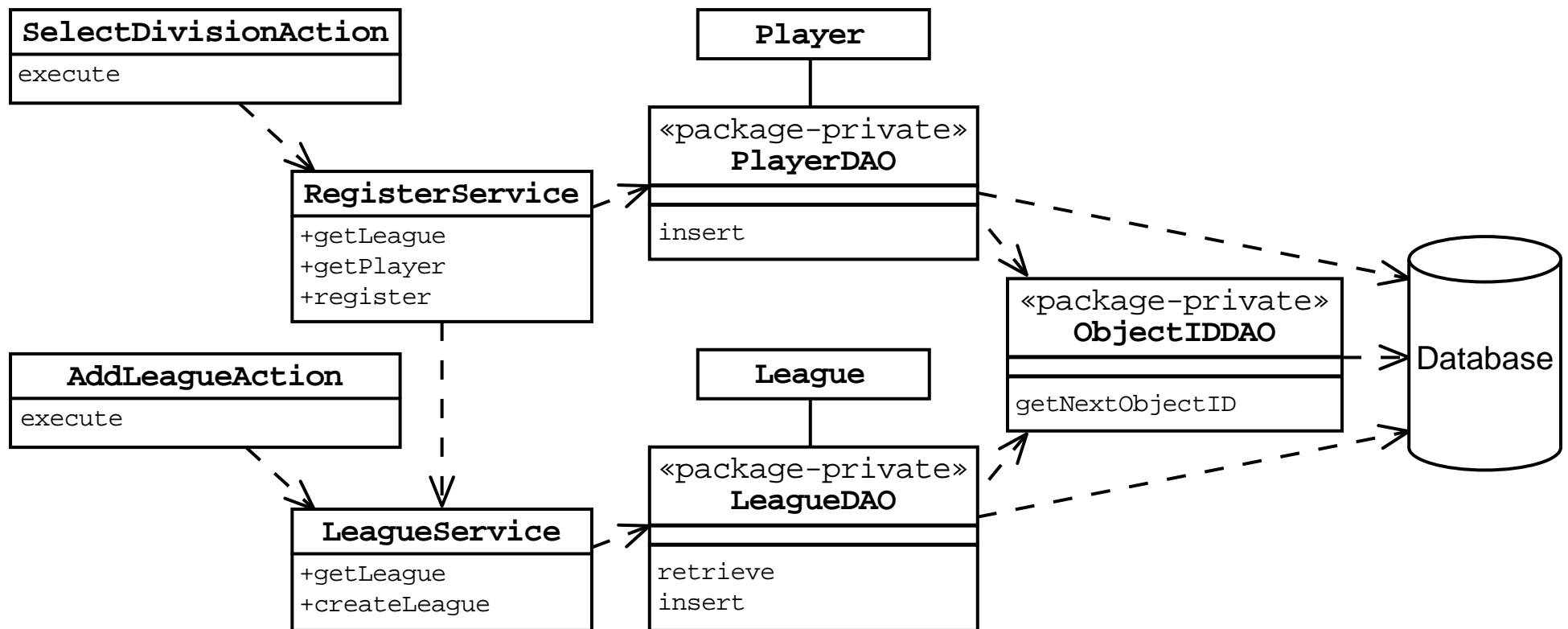
## Data Access Object Pattern

- 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





## DAO Pattern Advantages

- 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 permits changes to front-end technologies.
- This design permits changes to back-end technologies.



## Objectives

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



## Traditional Approaches to Database Connections

- Have you developed a web application that connects to a database?
- How did you make connections in the web application?
- What problems did you experience?



## Traditional Approaches to Database Connections

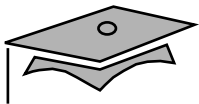
- Use `DriverManager.getConnection` to create database connections with every request.
- Create a connection and store it as a member variable of the servlet.
- Use a connection pool to recycle connections.
- Can use servlet context to store the connection pool
  - A custom connection pool might present maintenance problems
  - Servlet context is not available to business tier components (such as DAOs)



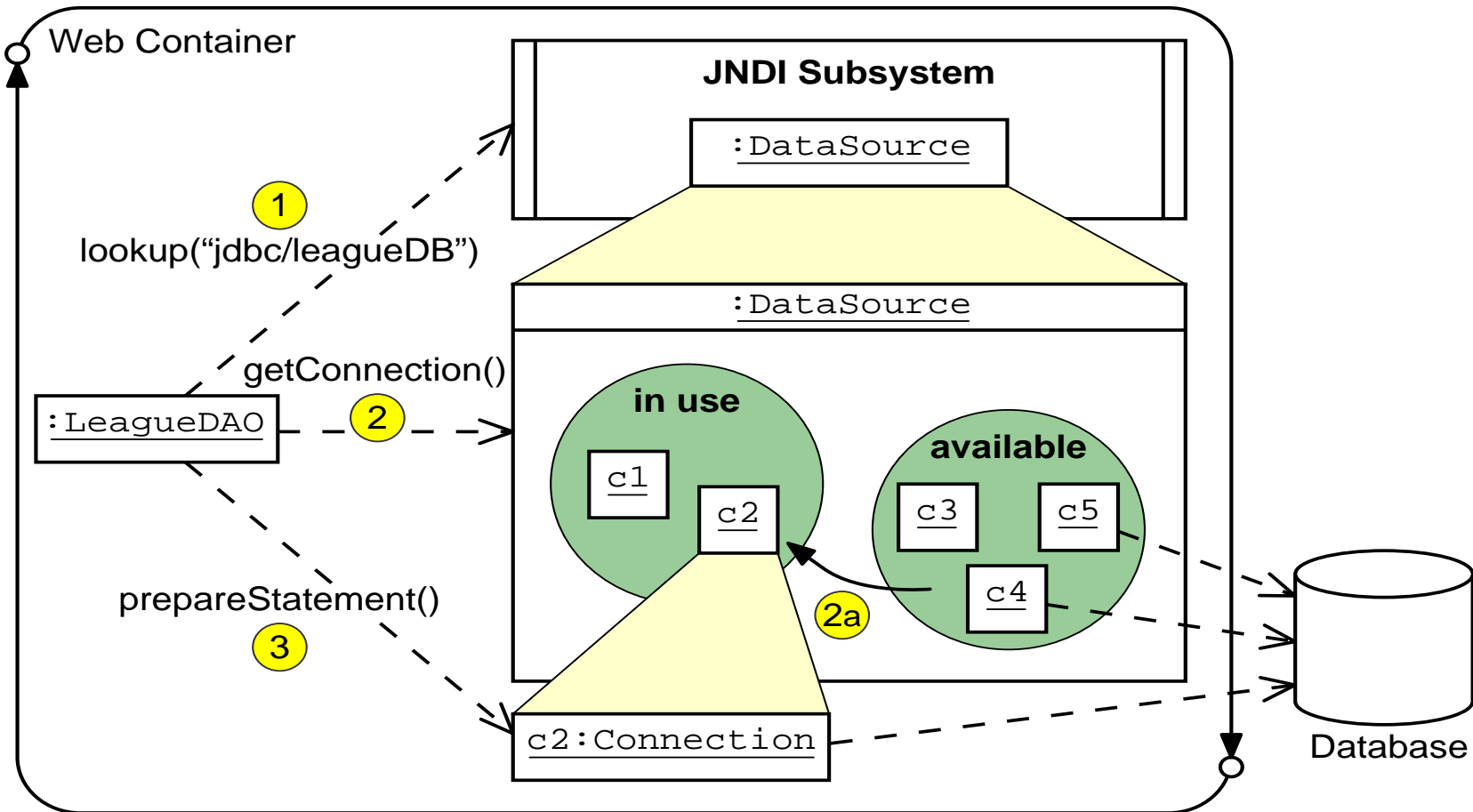


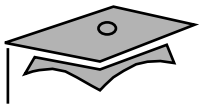
## Using a DataSource and JNDI API

- J2EE application servers provide a namespace, which can be accessed using JNDI APIs.
- J2EE application servers must support storing DataSource resources in JNDI namespace.
- DataSource is an object which encapsulates the information to connect to the database:
  - Database URL
  - Driver
  - User name and password
- Most servers provide a database connection pool that is accessed using the DataSource.

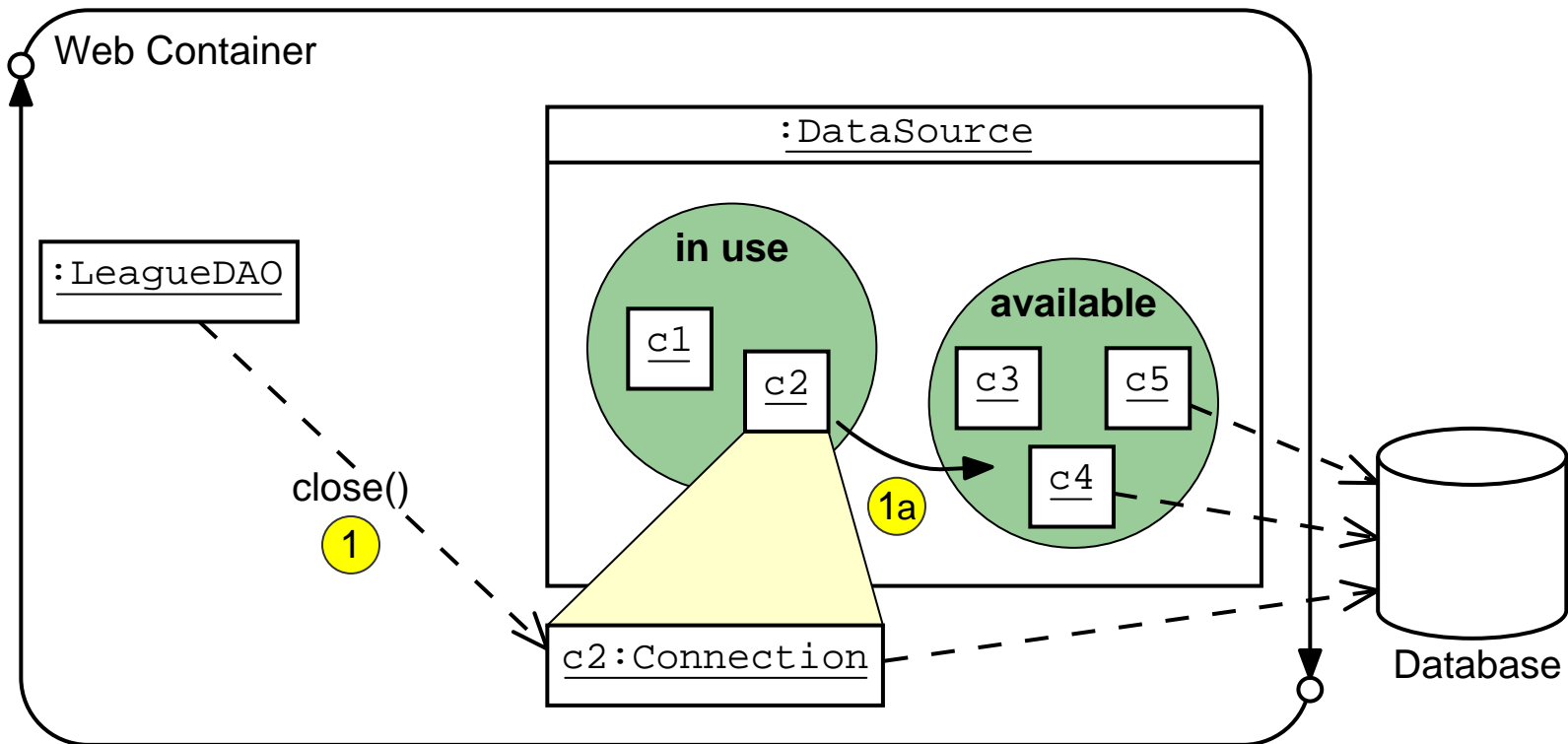


# Application DataSource Use





# Application DataSource Use





## Application DataSource Use

- The DataSource API:

<code>javax.sql.DataSource</code>
<code>getConnection(): java.sql.Connection</code> <code>getConnection(username: String, password: String): java.sql.Connection</code>

- Locate DataSource using JNDI lookup:

```
52     Context ctx = new InitialContext();
53     if ( ctx == null ) {
54         throw new RuntimeException("JNDI Context could not be found.");
55     }
56     ds = (DataSource)ctx.lookup("java:comp/env/jdbc/leagueDB");
57     if ( ds == null ) {
58         throw new RuntimeException("DataSource could not be found.");
59     }
```



## Configuring a DataSource and JNDI

- JNDI lookup needs to be defined in the `web.xml` deployment descriptor:

```
197 <resource-ref>
198   <description>
199     This defines a JNDI resource reference for
200     java:comp/env/jdbc/leagueDB DataSource which
201     is formally declared in the database.xml in
202     the TOMCAT_HOME/conf/Catalina/localhost directory.
203   </description>
204   <res-ref-name>jdbc/leagueDB</res-ref-name>
205   <res-type>javax.sql.DataSource</res-type>
206   <res-auth>Container</res-auth>
207 </resource-ref>
```

- In Tomcat, a DataSource is configured in the application configuration file under `TOMCAT_HOME/conf/Catalina/localhost`.



# Tomcat DataSource Configuration

```
1  <?xml version='1.0' encoding='utf-8'?>
2  <Context docBase="database" path="/database">
3      <Resource name="jdbc/leagueDB" type="javax.sql.DataSource"/>
4      <ResourceParams name="jdbc/leagueDB">
5          <parameter>
6              <name>driverClassName</name>
7              <value>com.pointbase.jdbc.jdbcUniversalDriver</value>
8          </parameter>
9          <parameter>
10             <name>url</name>
11             <value>jdbc:pointbase://localhost:9093/league</value>
12          </parameter>
13          <parameter>
14             <name>username</name>
15             <value>PBPUBLIC</value>
16          </parameter>
17          <parameter>
18             <name>password</name>
19             <value>PBPUBLIC</value>
20          </parameter>
```



## Tomcat DataSource Configuration

```
21     <parameter>
22         <name>maxWait</name>
23         <value>5000</value>
24     </parameter>
25     <parameter>
26         <name>maxActive</name>
27         <value>4</value>
28     </parameter>
29     <parameter>
30         <name>maxIdle</name>
31         <value>2</value>
32     </parameter>
33 </ResourceParams>
34 </Context>
```



## Summary

- Most web applications need to interface to a resource tier (usually a relational database).
- The DAO pattern separates the business tier components from the resource tier.
- In J2EE technology-compliant web containers, the best solution to access a DB connection is by using a DataSource object that is stored under JNDI.
- The DataSource object provides a pool of DB connections.
- You must configure a JNDI DataSource resource in the deployment descriptor, but you also have to configure it in the web container.





# Module 11

## Developing JSP™ Pages



# Objectives

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



## Relevance

- What problems exist in generating an HTML response in a servlet?
- How do template page technologies (and JSP technology in particular) solve these problems?



# JavaServer Pages 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.



# Hello World Servlet

```
10
11 public class HelloServlet 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
27     public void generateResponse(HttpServletRequest request,
28                                HttpServletResponse response)
29         throws IOException {
```



## Hello World Servlet

```
30
31     String name = request.getParameter("name");
32     if ( (name == null) || (name.length() == 0) ) {
33         name = DEFAULT_NAME;
34     }
35
36     response.setContentType("text/html");
37     PrintWriter out = response.getWriter();
38
39     out.println("<HTML>");
40     out.println("<HEAD>");
41     out.println("<TITLE>Hello Servlet</TITLE>");
42     out.println("</HEAD>");
43     out.println("<BODY BGCOLOR='white'>");
44     out.println("<B>Hello, " + name + "</B>");
45     out.println("</BODY>");
46     out.println("</HTML>");
47
48     out.close();
49 }
```



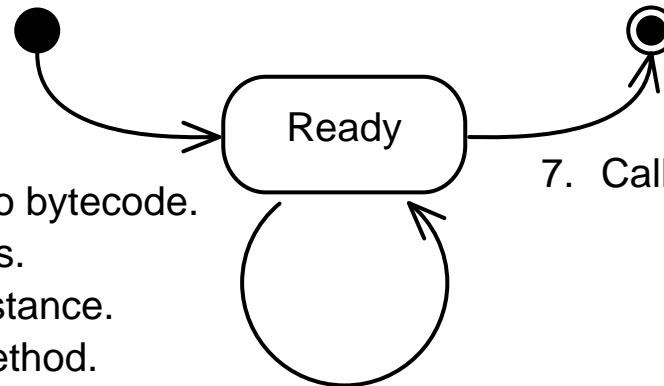
## The hello.jsp Page

```
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
```



# Steps of JSP Page Processing

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

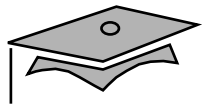


6. Call the `_jspService` method.

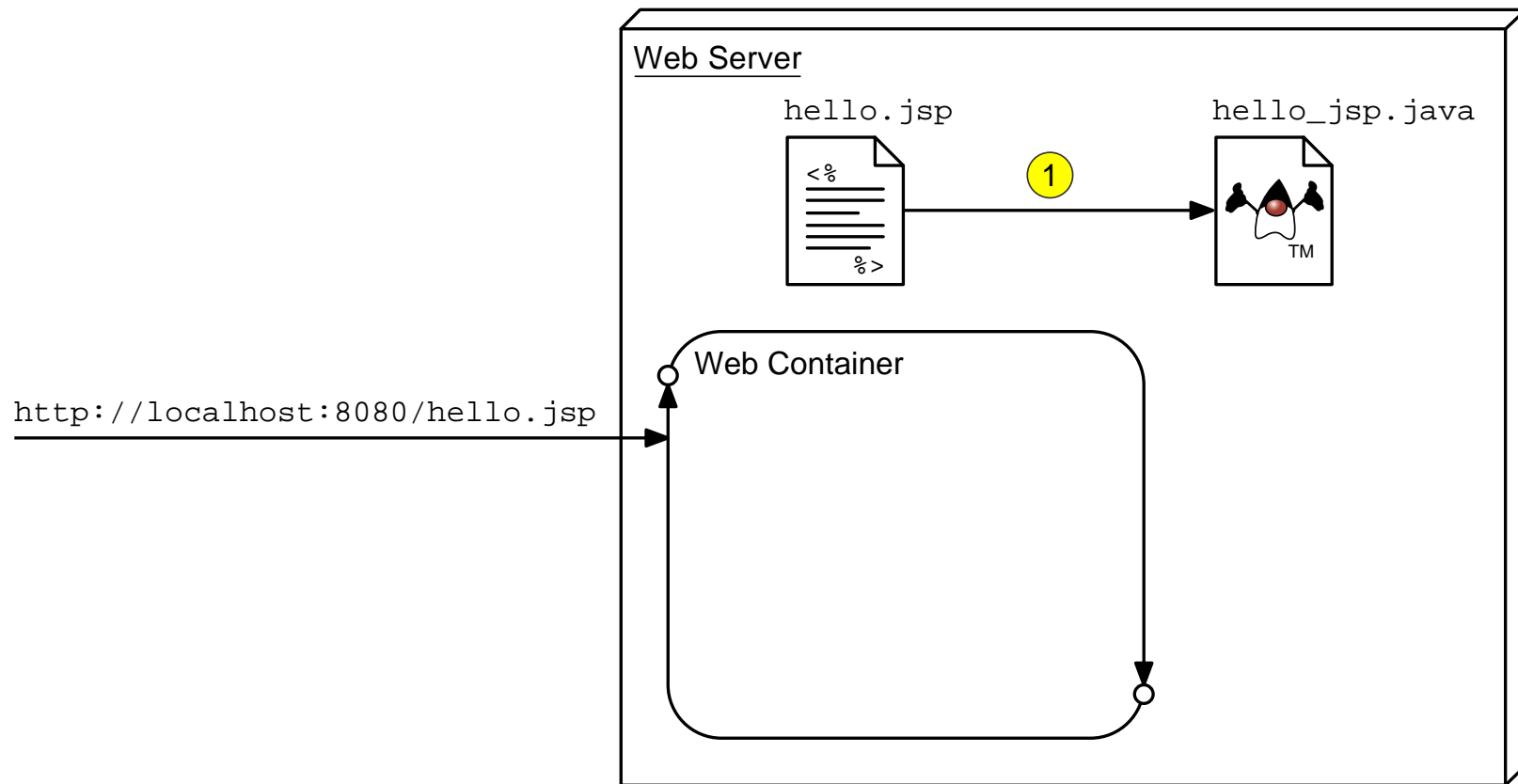
7. Call the `jspdestroy` method.

```
«interface»  
HttpJspPage  
  
jspInit()  
_jspService(req, resp)  
jspDestroy()
```



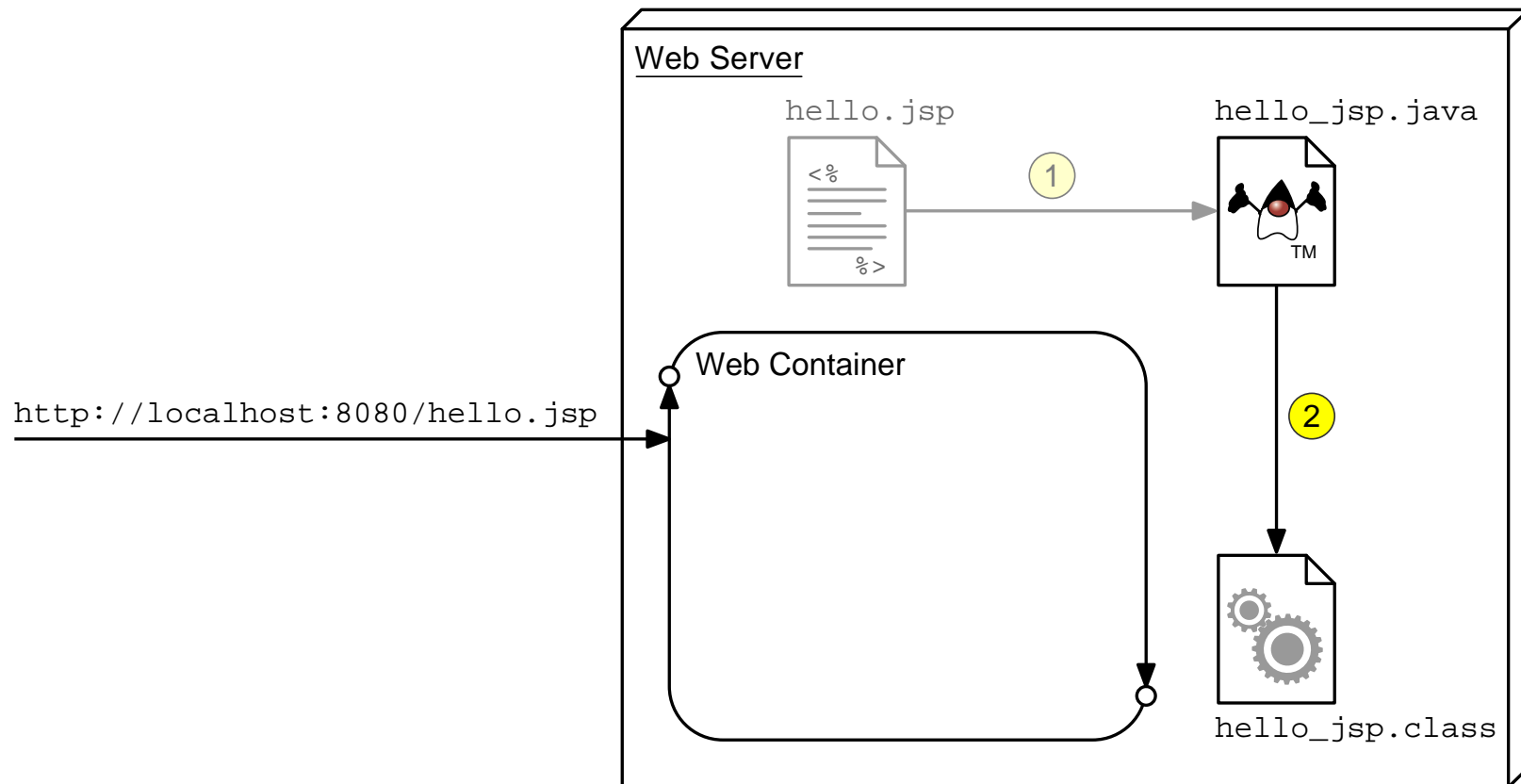


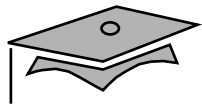
# JSP Page Translation



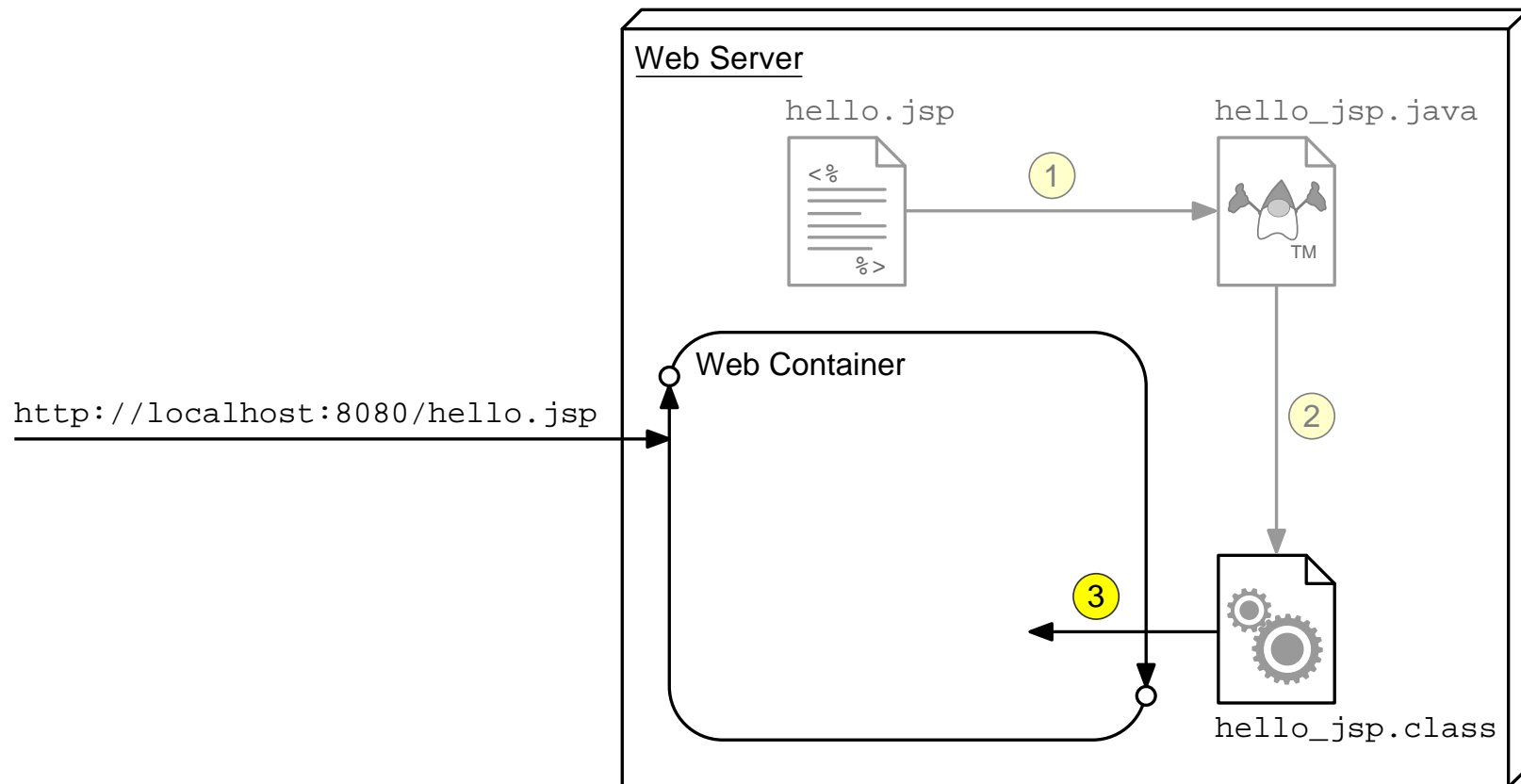


# JSP Page Compilation



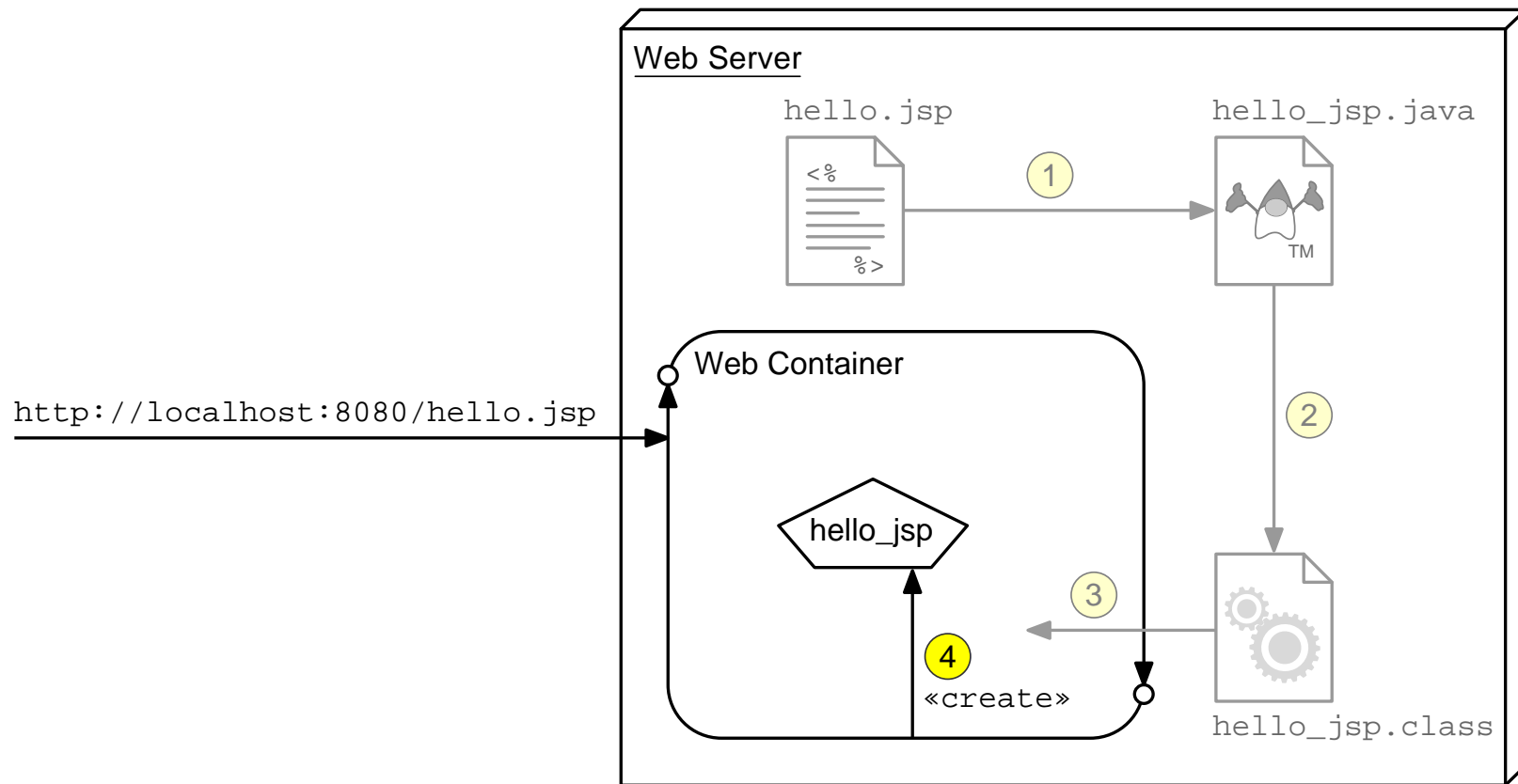


# JSP Page Class Loading

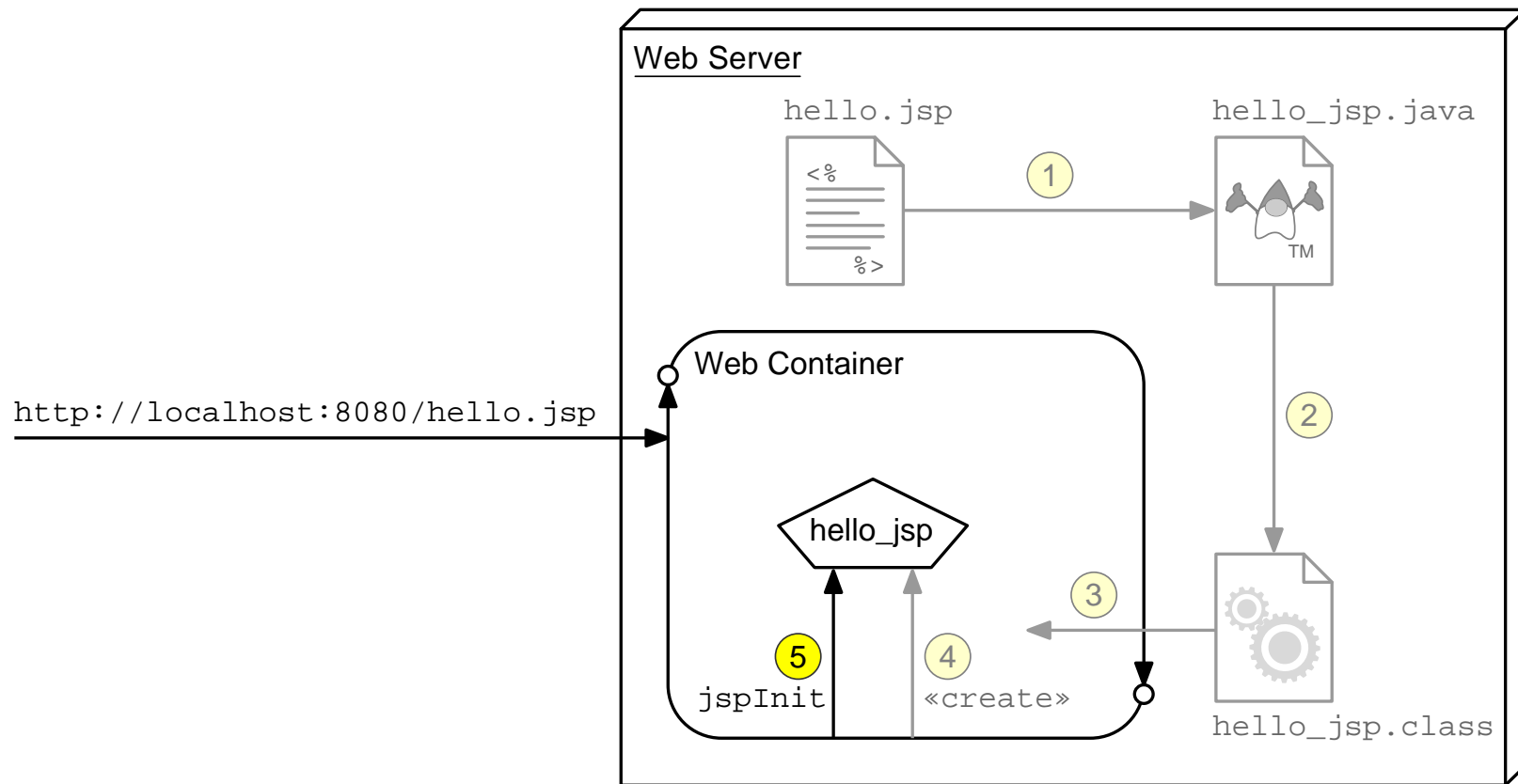




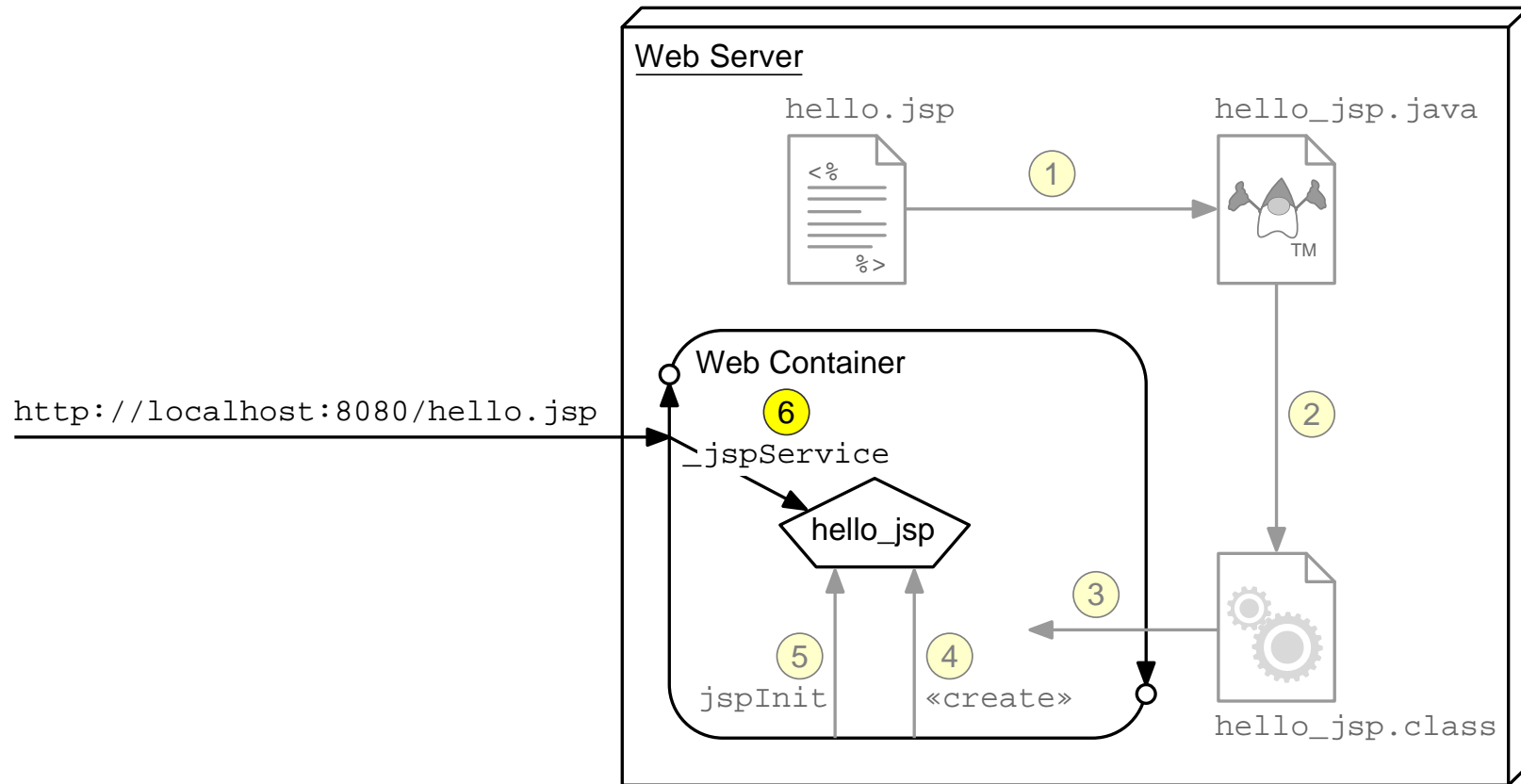
# JSP Page Servlet Instance



# JSP Page Initialization

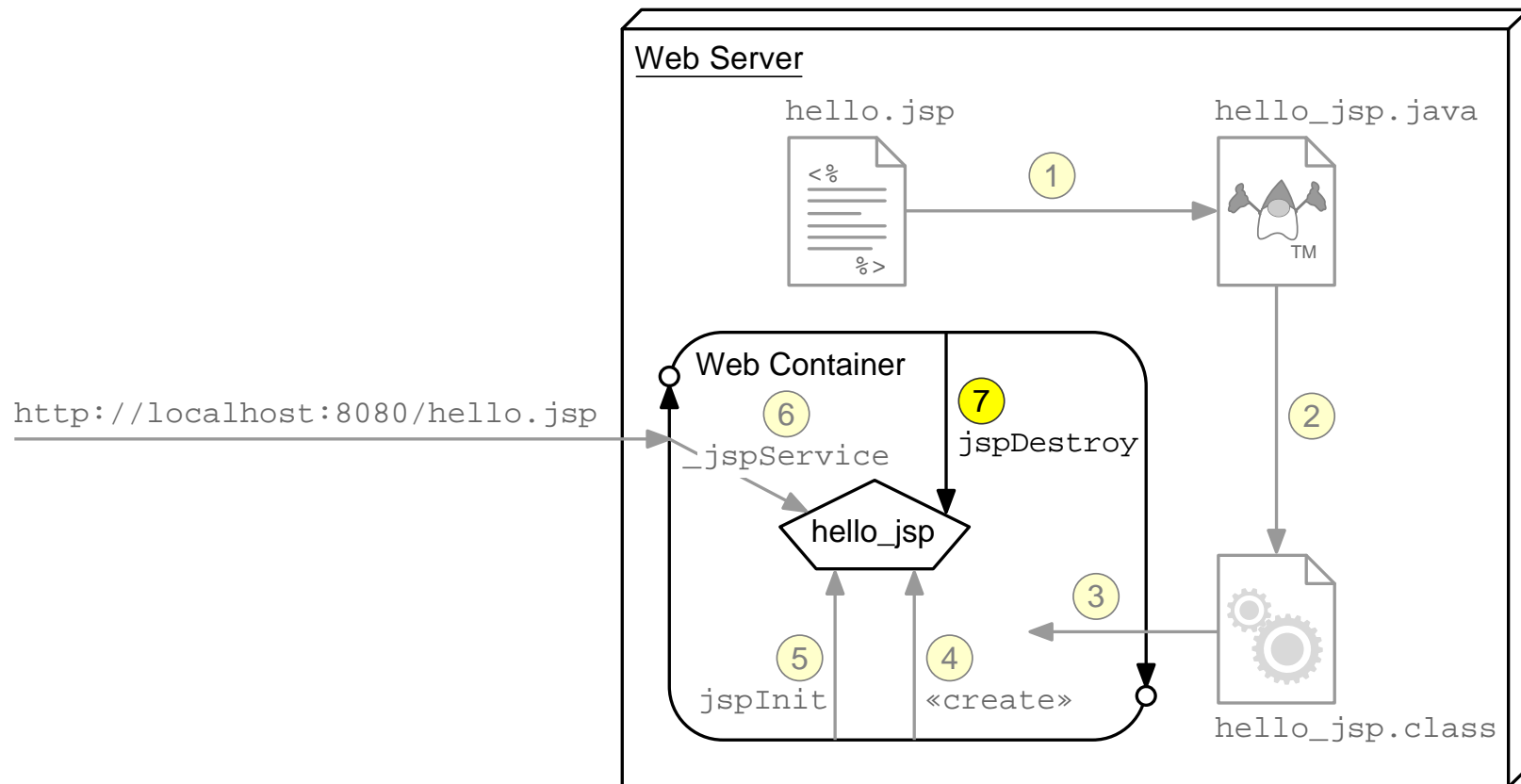


# JSP Page Service





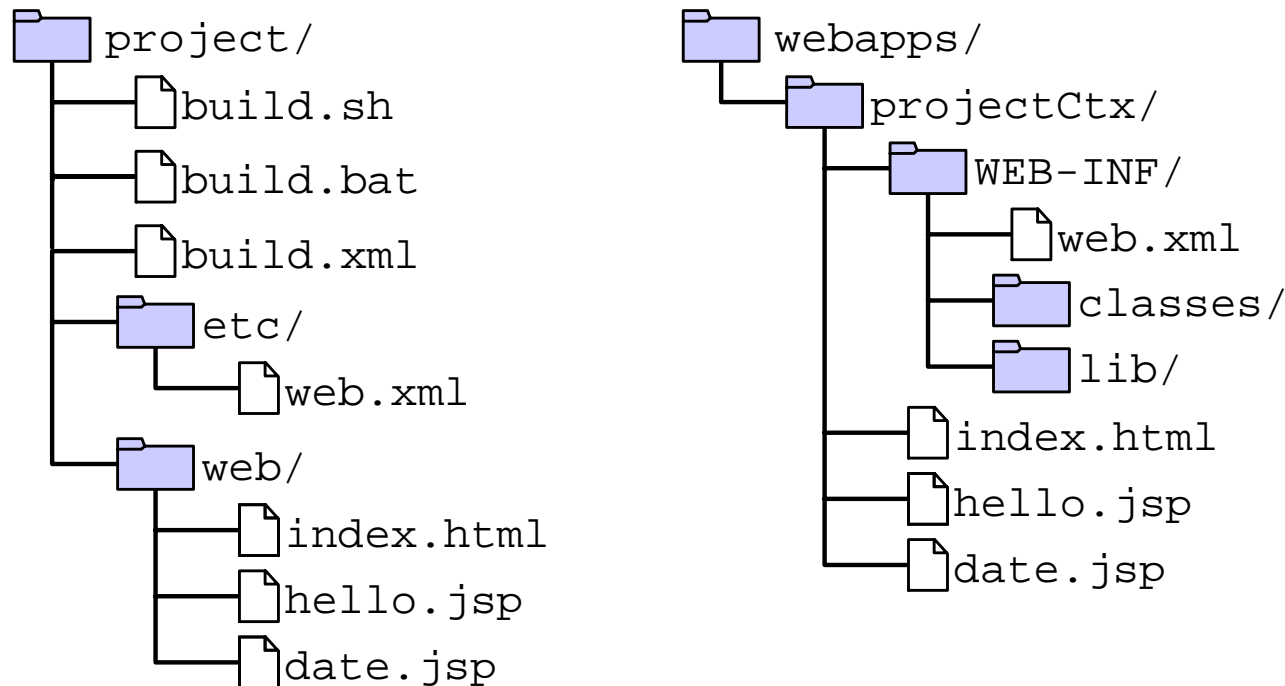
# JSP Page Destroyed





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



## Writing JSP Scripting Elements

JSP scripting elements `<% %>` are processed by the JSP engine.

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

There are five types of scripting elements:

Scripting Element	Scripting Syntax
Comment	<code>&lt;%-- <i>comment</i> --%&gt;</code>
Directive	<code>&lt;%@ <i>directive</i> %&gt;</code>
Declaration	<code>&lt;%! <i>decl</i> %&gt;</code>
Scriptlet	<code>&lt;% <i>code</i> %&gt;</code>
Expression	<code>&lt;%= <i>expr</i> %&gt;</code>



## Comments

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

- 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

A directive tag affects the JSP page translation phase.

- Syntax:

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

- Examples:

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

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



## Declaration Tag

A declaration tag lets the JSP page developer 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

A scriptlet tag lets the JSP page developer 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

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.

Variable Name	Description
<code>request</code>	The <code>HttpServletRequest</code> object associated with the request.
<code>response</code>	The <code>HttpServletResponse</code> object associated with the response that is sent back to the browser.
<code>out</code>	The <code>JspWriter</code> object associated with the output stream of the response.
<code>session</code>	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.
<code>application</code>	The <code>ServletContext</code> object for the web application.





# Implicit Variables

Additional variables:

<b>Variable Name</b>	<b>Description</b>
<code>config</code>	The <code>ServletConfig</code> object associated with the servlet for this JSP page.
<code>pageContext</code>	The <code>pageContext</code> object encapsulates the environment of a single request for this JSP page.
<code>page</code>	The <code>page</code> variable is equivalent to the <code>this</code> variable in the Java programming language.
<code>exception</code>	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 JSP technology
- Write JSP code using scripting elements
- Write JSP code using the page directive
- Write JSP code using standard tags
- Write JSP code using the Expression Language
- Configure the JSP environment in the web.xml file



## Using 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 (the `import` attribute is the one exception).
- 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.



## Using the page Directive

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

Attribute	Use
language	Defines the scripting language to be used in the page. The value <code>java</code> is the only value currently defined and is the default.
extends	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.
buffer	Defines the size of the buffer used in the output stream (a <code>JspWriter</code> object). The value is either <code>none</code> or <code>Mkb</code> . The default buffer size is 8 KB or greater. For example: <code>buffer="8kb"</code> or <code>buffer="none"</code>



## Using the page Directive

Attribute	Use
autoFlush	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> .
session	Defines whether the JSP page is participating in an HTTP session. The value can be either <code>true</code> (the default) or <code>false</code> .
import	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>



## Using the `page` Directive

Attribute	Use
<code>isThreadSafe</code>	Allows the JSP page developer to declare whether or not the JSP page is thread-safe.
<code>info</code>	Defines an informational string about the JSP page.
<code>contentType</code>	Defines the MIME type of the output stream. The default is <code>text/html</code> .
<code>pageEncoding</code>	Defines the character encoding of the output stream. The default is <code>ISO-8859-1</code> .
<code>isELIgnored</code>	Specifies whether EL elements are ignored on the page. The value is either <code>true</code> or <code>false</code> (default). If set to <code>true</code> , EL on the page is not evaluated.



## Using the `page` Directive

Attribute	Use
<code>isErrorPage</code>	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.
<code>errorPage</code>	Indicates another JSP page that handles 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)



## Objectives

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





## Using Standard Tags

The JSP specification provides standard tags for use within your JSP pages.

- In the `jsp:` namespace
- Available in every JSP container
- Reduces the need to use scriptlets in JSP pages
- EL and JSTL reduce the need for standard tags

In this module, you see the standard tags for handling components based on JavaBeans™ component architecture (JavaBeans components/bean).



## JavaBeans™ Components

A JavaBeans component is a Java class that:

- Has properties defined with accessor and mutator methods (get and set methods)
- Has a no-argument constructor
- Has no public instance variables
- Implements the `java.io.Serializable` interface

A JavaBeans component is not a component based on the Enterprise JavaBeans™ specification (EJB™ component) component



# The CustomerBean JavaBeans Component

```
1  package sl314.beans;
2
3  import java.io.Serializable;
4
5  public class CustomerBean implements Serializable {
6
7      private String name;
8      private String email;
9      private String phone;
10
11     public CustomerBean() {
12         this.name = "";
13         this.email = "";
14         this.phone = "";
15     }
16
```



# The CustomerBean JavaBeans Component

```
16
17     public void setName(String name) {
18         this.name = name;
19     }
20     public String getName() {
21         return name;
22     }
23
24     public void setEmail(String email) {
25         this.email = email;
26     }
27     public String getEmail() {
28         return email;
29     }
30
```



## The useBean Tag

If you want to interact with a JavaBeans instance using the standard tags in a JSP page, you must first declare the bean. You do this by using the useBean standard tag.

- Create or locate a JavaBeans instance for use on the page
- Syntax for the tag:

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

- id: name of bean
- scope: location of bean (default is page)
- class: fully qualified classname



## The useBean Tag

The useBean standard tag allows you to retrieve or create a JavaBean object:

- Given

```
<jsp:useBean id="myBean"
             scope="request"
             class="s1314.beans.CustomerBean" />
```

- Java equivalent:

```
CustomerBean myBean
    = (CustomerBean) request.getAttribute("myBean");
if ( myBean == null ) {
    myBean = new CustomerBean();
    request.setAttribute("myBean", myBean);
}
```



## The useBean Tag

The useBean tag in a JSP Page can have a body:

```
1  <jsp:useBean id="cust" scope="request"
2      class="sl314.beans.CustomerBean">
3      <%
4          cust.setName(request.getParameter("name"));
5          cust.setEmail(request.getParameter("email"));
6          cust.setPhone(request.getParameter("phone"));
7      %>
8  </jsp:useBean>
```

- The body is only evaluated if the bean is created
- If the bean is located in the named scope, the body is skipped



## The setProperty Tag

The setProperty tag stores attributes in a JavaBeans component.

- Syntax:

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

- The *property\_expression* is one of:
  - `property=" * "`
  - `property="propertyName"`
  - `property="propertyName" param="parameterName"`
  - `property="propertyName" value="propertyValue"`





## The setProperty Tag

The setProperty tag:

- Given:

```
<jsp:setProperty name="cust"  
property="email" />
```

- Java equivalent:

```
cust.setEmail(request.getParameter("email"));
```



## The getProperty Tag

The `getProperty` tag retrieves an attribute from a JavaBeans component.

- Syntax:

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

- Given:

```
<jsp:getProperty name="cust"  
property="email" />
```

- Java equivalent:

```
out.print(cust.getEmail());
```



## The getProperty Tag

The useBean tag output appears along with template text.

```
15 <H2>Customer Information:</H2>
16 Name: <jsp:getProperty name="cust" property="name" /><BR>
17 Email: <jsp:getProperty name="cust" property="email" /><BR>
18 Phone: <jsp:getProperty name="cust" property="phone" /><BR>
```



## Objectives

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



## Using Expression Language Elements

The purpose of EL is to aid in producing scriptless JSP pages.

- Syntax of EL in a JSP page:  
 `$\${expr}$`
- You can escape the expression:  
 `$\$\{expr\}$`
- Expressions can be used in two ways:
  - Attribute values in custom and standard actions
  - Within template text



## Bean Access Using EL

Beans within the namespace available to the JSP can be accessed easily using EL.

- Beans can be accessed by way of dot notation:  
`${bean.attribute}`
- Beans can be located by searching through the scopes:  
page, request, session and application
- Bean scope can be specified by preceding the bean name with the scope  
`${sessionScope.cust.firstName}`



# EL Implicit Objects

EL defines several objects:

<b>Implicit Object</b>	<b>Description</b>
<code>pageContext</code>	The <code>PageContext</code> object
<code>pageScope</code>	A Map containing page-scoped attributes and their values
<code>requestScope</code>	A Map containing request-scoped attributes and their values
<code>sessionScope</code>	A Map containing session-scoped attributes and their values
<code>applicationScope</code>	A Map containing application-scoped attributes and their values
<code>param</code>	A Map containing request parameters and single string values



# EL Implicit Objects

Additional objects:

<b>Implicit Object</b>	<b>Description</b>
<code>paramValues</code>	A Map containing request parameters and their corresponding string arrays
<code>header</code>	A Map containing header names and single string values
<code>headerValues</code>	A Map containing header names and their corresponding string arrays
<code>cookie</code>	A Map containing cookie names and their values





## EL Implicit Objects

For example,

```
${param.username}
```

If the bean returns an array, and element can specify its index using [] notation:

```
${paramValues.fruit[2]}
```



# Arithmetic Operators

Five arithmetic operators defined:

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



# Arithmetic Operators

Example operations:

<b>EL Expression</b>	<b>Result</b>
<code>#{3 div 4}</code>	0.75
<code>#{1 + 2 * 4}</code>	9
<code>#{(1 + 2) * 4}</code>	12
<code>#{32 mod 10}</code>	2



# Comparisons and Logical Operators

EL has six comparison operators:

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



## Comparisons and Logical Operators

- EL has three logical operators

Logical Operation	Operator
and	&& and and
or	and or
not	! and not

- Comparison and logical operations return a boolean
- Typically used as value for custom tag attribute
- Inserts true or false in output stream if used within template text



## Objectives

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



## Configuring the JSP Environment

This section outlines the deployment descriptor configuration for the JSP environment.

- Defined within `jsp-config` tag
- `jsp-property-group` defines a set of JSP pages:
  - The `url-pattern` – Specifies pages that belong to a group
  - The `scripting-invalid` – Turns scripting on or off
  - The `el-ignored` – Turns EL interpretation on or off
  - The `include-prelude` – Adds the specified JSP fragment to the beginning of every resource in the group
  - The `include-coda` – Adds the specified JSP fragment to the end of every resource in the group



# Configuring the JSP Environment

Multiple `jsp-property-group` elements available:

```
13     <jsp-config>
14         <jsp-property-group>
15             <url-pattern>/scripting_off/*</url-pattern>
16             <scripting-invalid>>true</scripting-invalid>
17         </jsp-property-group>
18
19         <jsp-property-group>
20             <url-pattern>/EL_off/*</url-pattern>
21             <el-ignored>>true</el-ignored>
22         </jsp-property-group>
23
24         <jsp-property-group>
25             <url-pattern>/prelude_coda/*</url-pattern>
26             <include-prelude>/prelude_coda/prelude.jspf</include-prelude>
27             <include-coda>/prelude_coda/coda.jspf</include-coda>
28         </jsp-property-group>
29     </jsp-config>
```





## Summary

- JavaServer Pages are dynamic HTML pages that execute on the server.
- JSP pages are converted to raw servlets at runtime.
- You can use scripting elements to embed Java technology code to perform dynamic content generation.
- You can also use standard actions and the Expression Language to reduce the amount of Java technology code.
- The ultimate goal of JSP technology is to allow non-programmers to create dynamic HTML.



# Module 12

## Developing JSP Pages Using Custom Tags



## Objectives

- Understand the job roles involved in web application development
- Understand the rationale in using custom tags
- Develop a JSP page using an existing custom tag library



## Relevance

- Who in your organization will be creating JSP pages?
- Suppose you start with a small number of JSP pages in a web application and have a significant amount of scripting code in these pages. What problems can you foresee as the web application grows?



## Job Roles Revisited

Job roles for a large web application might include:

- *Web Designers* – Responsible for creating the views of the application, which are primarily composed of HTML pages
- *Web Component Developers* – Responsible for creating the control elements of the application, which is almost exclusively Java technology code
- *Business Component Developers* – Responsible for creating the model elements of the application, which might reside on the web server or on a remote server (such as an EJB technology server)



## Objectives

- Understand the job roles involved in web application development
- Understand the rationale in using custom tags
- Develop a JSP page using an existing custom tag library



# Contrasting Custom Tags and Scriptlet Code

```
42 <%-- Report any errors (if any) --%>
43 <%
44     // Retrieve the errorMsgs from the request-scope
45     List errorMsgs = (List) request.getAttribute("errorMsgs");
46     if ( (errorMsgs != null) && !errorMsgs.isEmpty() ) {
47 %>
48 <p>
49 <font color='red'>Please correct the following errors:
50 <ul>
51 <%
52     Iterator items = errorMsgs.iterator();
53     while ( items.hasNext() ) {
54         String message = (String) items.next();
55 %>
56     <li><%= message %></li>
57 <%
58         } // END of while loop over errorMsgs
59 %>
60 </ul>
61 </font>
62 </p>
63 <%
64     } // END of if errorMsgs is not empty
65 %>
```



# Contrasting Custom Tags and Scriptlet Code

Equivalent custom tag in the registration form:

```
40 <%-- Report any errors (if any) --%>
41 <c:if test="\${not empty errorMsgs}">
42     <p>
43     <font color='red'>Please correct the following errors:
44     <ul>
45     <c:forEach var="message" items="\${errorMsgs}">
46         <li>\${message}</li>
47     </c:forEach>
48     </ul>
49     </font>
50     </p>
51 </c:if>
```





# Contrasting Custom Tags and Scriptlet Code

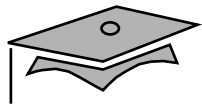
Advantages of custom tags compared to scriptlet code:

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



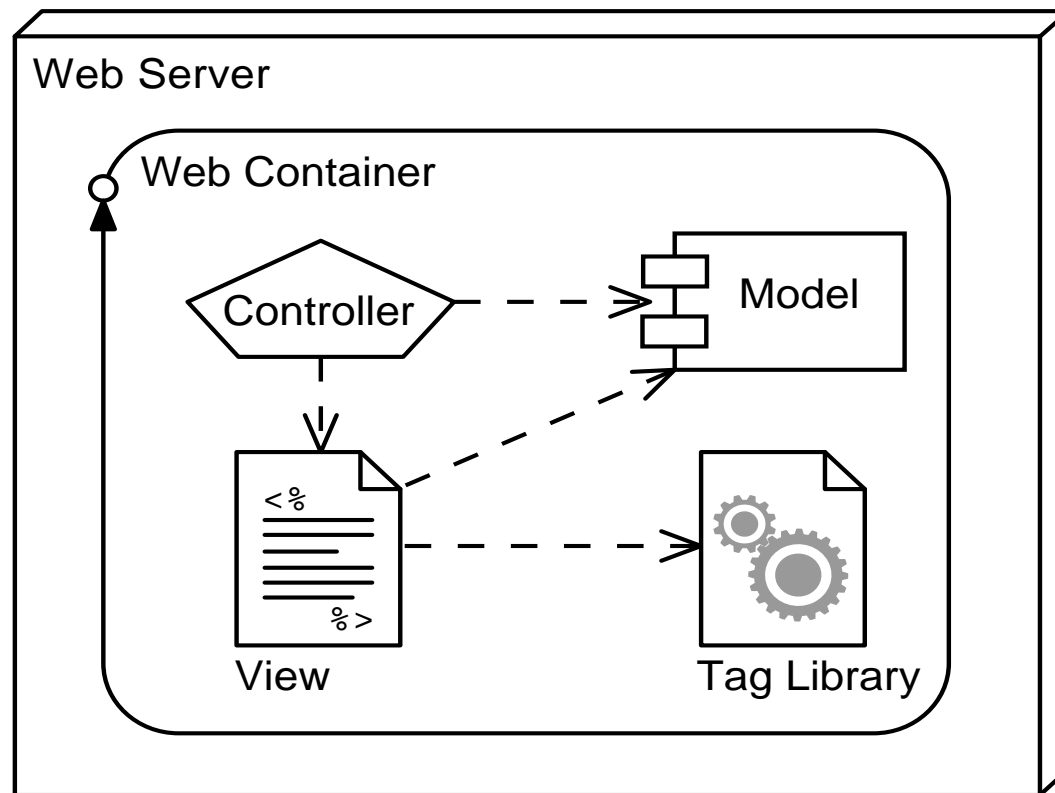
# Developing JSP Pages Using Custom Tags

- 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



# Custom Tag Library Overview

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





## Custom Tag Library Overview

- Custom tag handlers used in a JSP page can access any object that is also accessible to the JSP page.
- This is accomplished by the `pageContext` object that is unique for a given JSP page and for a given request on that JSP page.
- The `pageContext` object provides access to all attribute scopes: page, request, session, and application.
- The `pageContext` object provides access to all implicit objects in the JSP page (request, response, out, and so on).



# 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>
```



## JSTL Sample Tags

This section presents a few of the tags from the core tag library in JSTL:

- `set`
- `if`
- `forEach`
- `url`
- `out`



## The set Tag

You use the set tag to store a variable in a named scope or to update the property of a JavaBeans instance or Map.

- Body content – Empty if the value attribute is supplied. Otherwise, the body is the value.
- The var attribute – This mandatory attribute is the name of the request parameter.
- The value attribute – This optional attribute is an empty tag and provides the value for the variable.
- The scope attribute – This optional attribute supplies the scope location of the variable.



## The set Tag

The following example shows how to use the JSTL set tag:

```
4
5  <%-- Set page title --%>
6  <c:set var="pageTitle">Duke's Soccer League: Registration</c:set>
7
8  <%-- Generate the HTML response --%>
9  <html>
10 <head>
11   <title>${pageTitle}</title>
12 </head>
```





## The `if` Tag

The `if` tag is a conditional tag in JSTL. A test expression is evaluated and the results of the test can be stored for later use. If a body is supplied, the body is only evaluated if the test results in `true`.

- Body content – Optional. If present, only evaluated if test expression is `true`.
- The `test` attribute – This mandatory attribute contains the expression to be evaluated.
- The `var` attribute – This optional attribute is used to store the result of the test.
- The `scope` attribute – This optional attribute supplies the scope location of the `var` attribute.



## The `if` Tag

The following example shows how to use the JSTL `if` tag:

```
39
40 <%-- Report any errors (if any) --%>
41 <c:if test="{not empty errorMsgs}">
42   <p>
43     <font color='red'>Please correct the following errors:
44     <ul>
45       <c:forEach var="message" items="{errorMsgs}">
46         <li>{message}</li>
47       </c:forEach>
48     </ul>
49     </font>
50   </p>
51 </c:if>
```



## The forEach Tag

The `forEach` tag provides iteration capabilities over a body. If a collection is supplied, it can be a `java.util.Collection`, `java.util.Map`, `java.util.Iterator`, `java.util.Enumeration`, array, or comma-delimited string.

- Body content – Contains what will be iterated over.
- The `items` attribute – This optional attribute specifies the collection to be iterated over.
- The `var` attribute – This optional attribute stores the current item in the iteration.
- The `varStatus` attribute – This optional attribute stores information about the step of the iteration.



## The forEach Tag

Additional JSTL forEach tag attributes:

- The `begin` attribute – This attribute specifies the first element in the iteration. If the `items` attribute is not specified, the `begin` attribute is required.
- The `end` attribute – This attribute specifies the last element in the iteration. If the `items` attribute is not specified, the `end` attribute is required.
- The `step` attribute – This optional attribute specifies that the iteration should only include every *n*th item.



## The forEach Tag

The following example shows how to use the JSTL forEach tag:

```
43     <font color='red'>Please correct the following errors:
44     <ul>
45     <c:forEach var="message" items="{errorMsgs}">
46         <li>${message}</li>
47     </c:forEach>
48     </ul>
49     </font>
```



## The `url` Tag

You use the `url` tag to provide a URL with appropriate rewriting for session management. The rewritten URL is typically written to the output stream, but can be stored in a scoped variable for later use.

- The `value` attribute – This mandatory attribute specifies the URL to be rewritten.
- The `var` attribute – This optional attribute is used to store the rewritten URL.
- The `scope` attribute – This optional attribute is used to specify the storage location of the variable.



## The `url` Tag

The following example shows how to use the JSTL `url` tag:

```
52
53  <%-- Present the form --%>
54  <form action='<c:url value="enter_player.do" />' method='POST'>
```

The `value` attribute can also be used with absolute paths (relative to the web application's context root):

```
<form action='<c:url value="/register/enter_player.do" />'
        method='POST'>
...
</form>
```



## The out Tag

The out tag is used to evaluate an expression and write the result to the current `JSPWriter`.

- `Body content` – The body content can contain the default result.
- `value` attribute – This attribute specifies the expression to be evaluated.
- `default` attribute – This optional attribute specifies a result to use if the expression evaluates to `null`.
- `escapeXml` attribute – This optional attribute indicates whether or not the characters (`<`), (`>`), (`&`), (`'`), and (`"`) should be replaced (default is `true`).





## The out Tag

The following example shows how to use the out tag:

```
<c:out value="\${param.email}" default="no email provided" />
```

When displaying content provided by the user, it is best to set the `escapeXml` attribute to `true` to prevent cross-site attacks:

```
<p>  
  <b>Comments:</b> </br>  
  <c:out value="\${user.comments}" escapeXml="true" />  
</p>
```



## Objectives

- Understand the job roles involved in web application development
- Understand the rationale in using custom tags
- Develop a JSP page using an existing custom tag library



## 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.

```
149     <taglib>
150         <taglib-uri>http://www.soccer.org/forms.tld</taglib-uri>
151         <taglib-location>/WEB-INF/forms.tld</taglib-location>
152     </taglib>
```

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

```
2     <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
3     <%@ taglib prefix="forms" uri="http://www.soccer.org/forms.tld" %>
```

Any number of tag libraries might 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. The following code shows that the `set` tag stores the variable `errors` in the page scope.

```
53 <%-- Present the form --%>
54 <form action='<c:url value="enter_player.do" />' method='POST'>
```

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



## Using a Conditional Custom Tag

Partial scriptlet code in the error page:

```
<%  
    if ( errors != null) {  
%>  
    <%-- JSP code --%>  
%>  
    } // end of IF  
%>
```

Equivalent custom tag in the error page:

```
<c:if test="{not empty errorMsgs}">  
    <%-- JSP code --%>  
</c:if>
```



## Using an Iterative Custom Tag

```
26
27 <%
28     // Retrieve the set of leagues the LeagueService
29     LeagueService leagueSvc = new LeagueService();
30     List leagueList = leagueSvc.getAllLeagues();
31 %>
32 <!-- Generate main body -->
33 <p>
34 The set of soccer leagues are:
35 </p>
36 <ul>
37 <%
38     Iterator items = leagueList.iterator();
39     while ( items.hasNext() ) {
40         League league = (League) items.next();
41 %>
42     <li><%= league.getTitle() %></li>
43 <%     }     %>
44 </ul>
```



## Using an Iterative Custom Tag

```
21
22 <!-- Retrieve the set of leagues the LeagueService -->
23 <jsp:useBean id="leagueSvc" scope="page"
24     class="sl314.model.LeagueService" />
25
26 <!-- Generate main body -->
27 <p>
28 The set of soccer leagues are:
29 </p>
30
31 <ul>
32 <c:forEach var="league" items="{leagueSvc.allLeagues}" >
33     <li>${league.title}</li>
34 </c:forEach>
35 </ul>
```



## Summary

- Custom tags are fundamentally the same as standard tags, but you can acquire tag libraries from third parties and even build your own application-specific tags.
- The JSP Standard Tag Library (JSTL) provides a collection of general-purpose tags.
- You can use a tag library in your JSP pages by declaring it using the `<%@ taglib %>` directive.
- Custom tags use standard XML tag syntax.
- With custom tags, standard tags, and the Expression Language, you can eliminate all scriptlet code in your JSP pages.





# Module 13

## Developing Web Applications Using Struts Action Forms



## Objectives

- Describe the components in a Struts application
- Develop an `ActionForm` class
- Develop a JSP page for a View form
- Configure the View forms



## Relevance

- What are the responsibilities of a Struts action class?
- Which of these responsibilities are really View-related aspects of the boundary component?
- Does Struts provide any facilities for separating these View-related aspects for the Controller-related action classes?



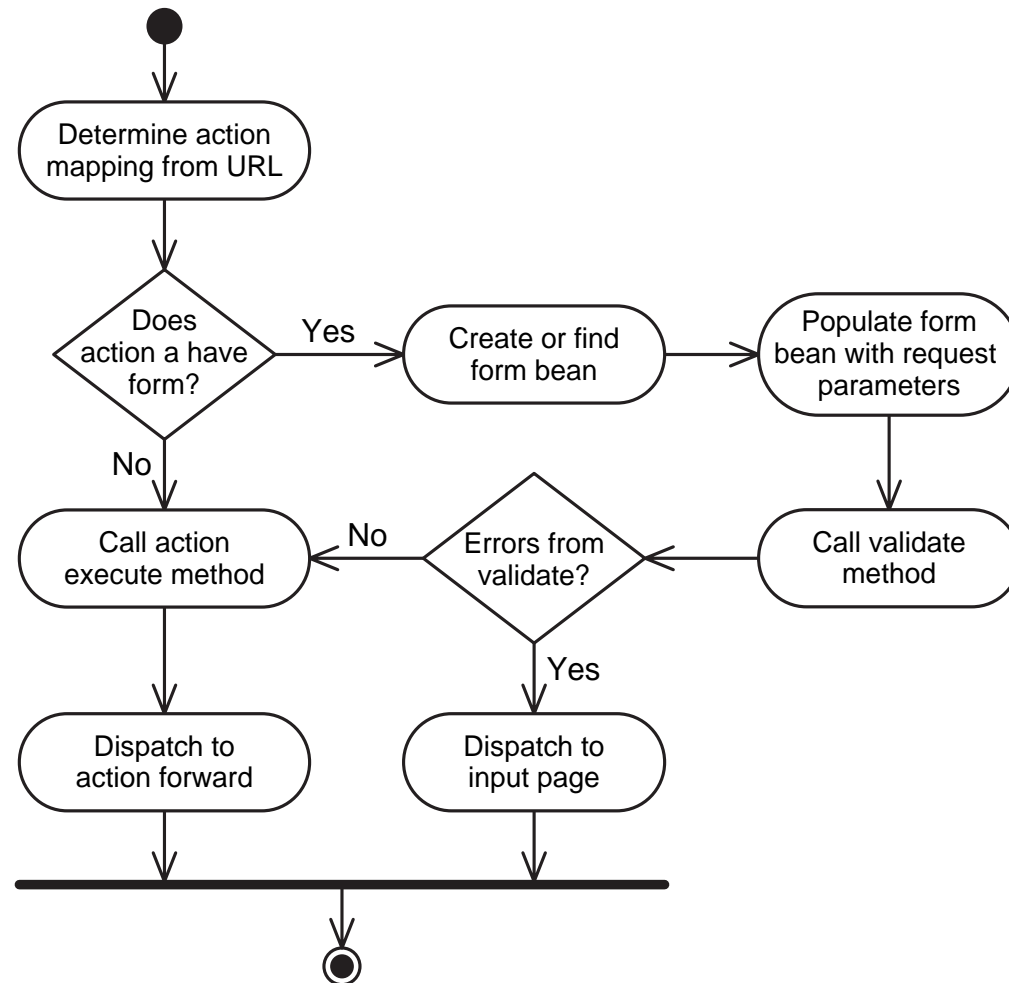
# Struts Application Components

Struts applications consist of the following:

- Model elements
- Control elements
- View elements



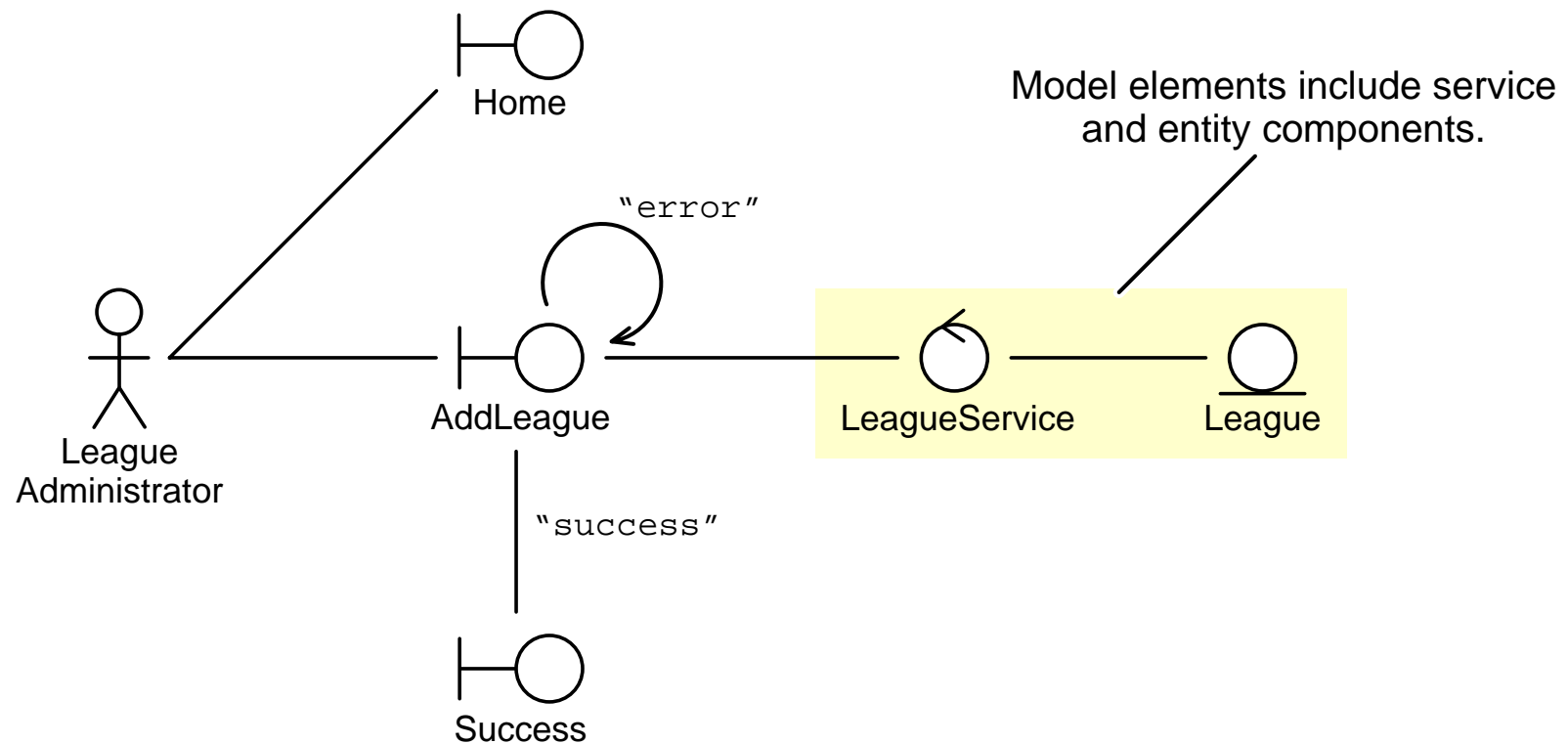
# Struts Activity Diagram





# Model Elements Review

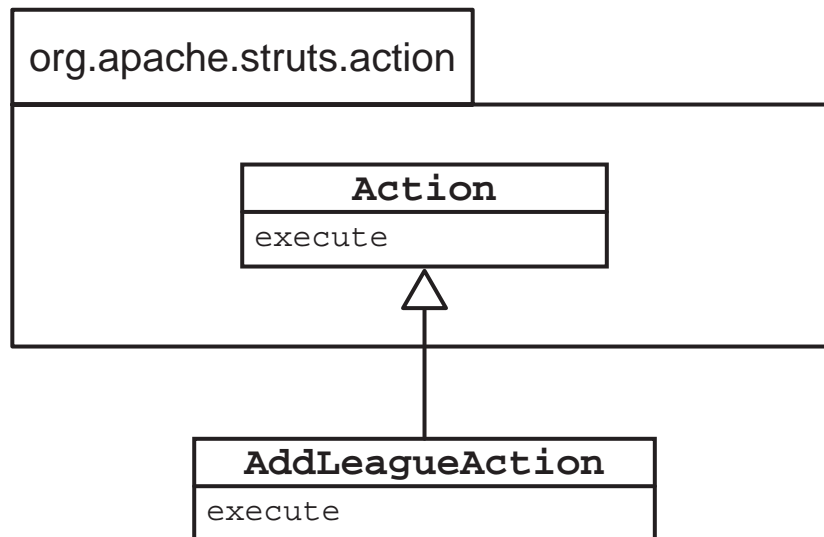
Model elements are the service and entity components in the application.





## Control Elements Review

- The `ActionServlet` control element is part of the Struts infrastructure
- Developers use the Struts subclass `Action` to create custom action classes



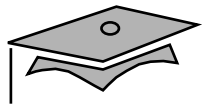


## Control Elements Review

The Action subclasses are configured in the Struts configuration file.

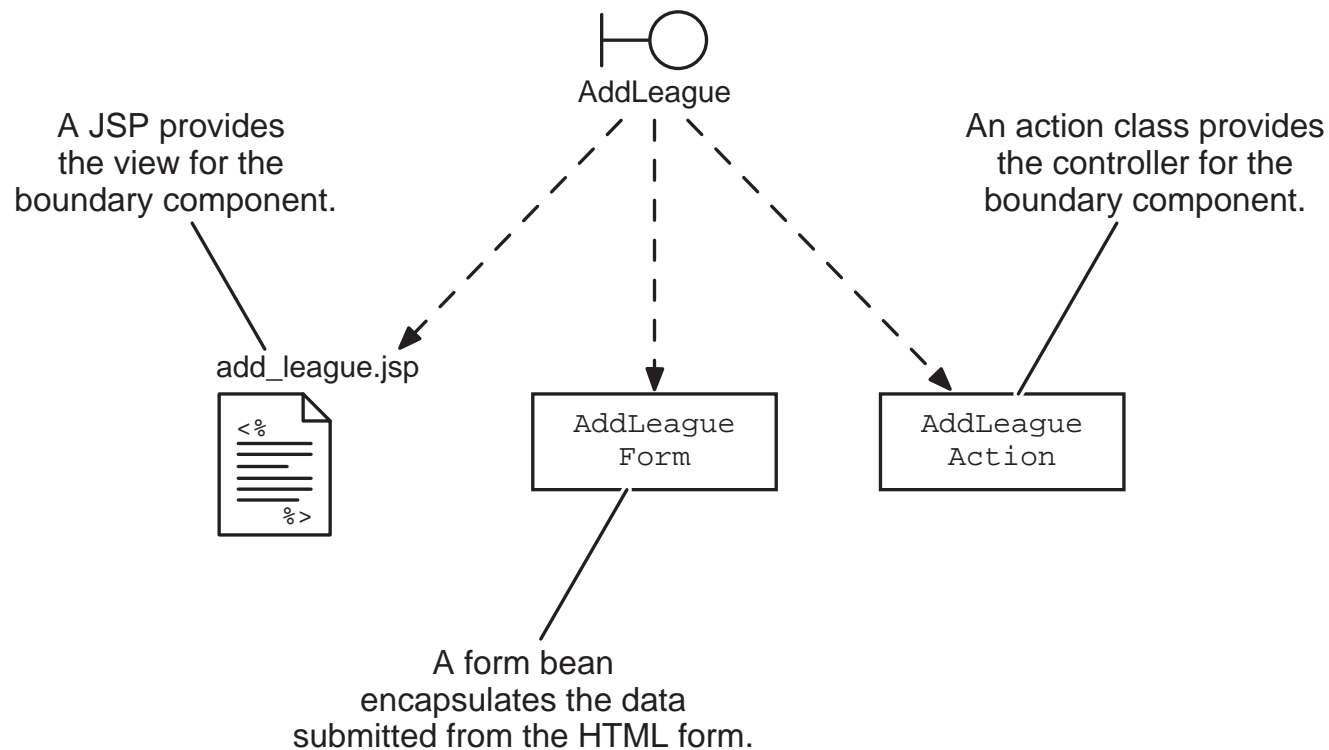
```
23
24     <!-- Declare the Registration actions -->
25     <action path="/register/select_league"
26             type="sl314.controller.SelectLeagueAction"
27             name="selectLeagueForm" scope="request" validate="true"
28             input="/register/select_league.jsp" >
29         <forward name="success" path="/register/enter_player.jsp"/>
30         <forward name="error" path="/register/select_league.jsp"/>
31     </action>
```





## View Elements Review

- Views in Struts can have multiple aspects



- Views can be static web pages, dynamic pages (using forms), and `ActionForm` elements



## Objectives

- Describe the components in a Struts application
- `Develop an ActionForm class`
- Develop a JSP page for a View form
- Configure the View forms



## Developing an ActionForm Class

- Provides an object representation of the elements in an HTML form
- Automatically created or located by the infrastructure controller
- Placed into the scope specified in the Struts configuration file
- The form bean is self-validating



# The Add a New League Form

The screenshot shows a Netscape browser window titled "Duke's Soccer League: Add a New League - Netscape". The address bar contains the URL "http://localhost:8080/controller/admin/add\_lea...". The page content includes a purple header bar with the text "Duke's Soccer League: Add a New League". Below the header, a paragraph states: "This form allows you to create a new soccer league." The form consists of the following fields:

- Year:
- Season:
- Title:

At the bottom of the form is an "Add League" button. The browser's status bar at the bottom shows "Done" and various system icons.



## The AddLeagueForm Class (Parts 1 and 2)

```
1  package sl314.view;
2
3  // Struts imports
4  import org.apache.struts.action.ActionForm;
5  import org.apache.struts.action.ActionMapping;
6  import org.apache.struts.action.ActionError;
7  import org.apache.struts.action.ActionErrors;
8  // Servlet imports
9  import javax.servlet.http.HttpServletRequest;
10
11  /**
12   * This is a Struts form bean for the "Add League" view.
13   */
14  public class AddLeagueForm extends ActionForm {
15
```



## The AddLeagueForm Class (Parts 1 and 2)

```
11  /**
12   * This is a Struts form bean for the "Add League" view.
13   */
14  public class AddLeagueForm extends ActionForm {
15
16      private String season = null;
17      public String getSeason() {
18          return season;
19      }
20      public void setSeason(String season) {
21          this.season = season;
22      }
23
24      private String title = null;
25      public String getTitle() {
26          return title;
27      }
28      public void setTitle(String title) {
29          this.title = title;
30      }
31
```



## The AddLeagueForm Class (Part 3)

```
31
32 // The raw 'year' property
33 private String yearStr = null;
34 public String getYearStr() {
35     return yearStr;
36 }
37 public void setYearStr(String yearStr) {
38     this.yearStr = yearStr;
39 }
40 // The converted 'year' property
41 private int year = -1;
42 public int getYear() {
43     return year;
44 }
45
```



## The AddLeagueForm Class (Part 4)

```
45
46 public ActionErrors validate(ActionMapping mapping,
47                             HttpServletRequest request) {
48     ActionErrors errors = new ActionErrors();
49
50     // Perform data conversions.
51     try {
52         this.year = Integer.parseInt(yearStr);
53     } catch (NumberFormatException nfe) {
54         errors.add("yearStr", new ActionError("error.yearField.required"));
55     }
56
57     // Verify form parameters
58     if ( (year != -1) && ((year < 2000) || (year > 2010)) ) {
59         errors.add("yearStr", new ActionError("error.yearField.range"));
60     }
61     if ( season.equals("UNKNOWN") ) {
62         errors.add("season", new ActionError("error.seasonField.required"));
63     }
64     if ( title.length() == 0 ) {
65         errors.add("title", new ActionError("error.titleField.required"));
66     }
67
68     // Return the errors list. An empty list tells Struts that this form
69     // passed the verification check.
70     return errors;
71 }
```





## Struts ActionError Class

- The `ActionError` objects hold a property key that identifies the application-specific error message.
- These error message keys are located in a resource bundle.

```
# Select League fields
```

```
error.seasonField.required=<li>Please select a league season.</li>
```

```
error.yearField.required=<li>The 'year' field must be a positive integer.<
```

- The `ActionErrors` class is a collection of error objects.
- This is just one piece of Struts i18n and l10n capabilities.
- The JSTL also has a tag library for i18n and l10n support.



## How the Controller Uses the Form Bean

- The `Action` class execute method passes in the form as a generic `ActionForm`. You must cast the form to your application-specific class.
- You can then use the accessor methods on the form bean to access the verified data in the form.
- You can remove all of the form verification code in your controller class because the form bean (and Struts) do it for you.
- The action classes can also use Struts' error classes.



## The AddLeagueAction Class

```
12 // Model classes
13 import sl314.model.LeagueService;
14 import sl314.model.League;
15 import sl314.model.ObjectNotFoundException;
16 // View classes
17 import sl314.view.AddLeagueForm;
18
19
20 public class AddLeagueAction extends Action {
21
22     public ActionForward execute(ActionMapping mapping, ActionForm form,
23                                 HttpServletRequest request,
24                                 HttpServletResponse response) {
25
26         // Use Struts actions to record business processing errors.
27         ActionErrors errors = new ActionErrors();
28         // Store this set in the request scope, in case we need to
29         // send the ErrorPage view.
30         saveErrors(request, errors);
31
32         try {
```



## The AddLeagueAction Class (Part 2)

```
27     ActionErrors errors = new ActionErrors();
28     // Store this set in the request scope, in case we need to
29     // send the ErrorPage view.
30     saveErrors(request, errors);
31
32     try {
33
34         // Cast the form to the application-specific action-form class
35         AddLeagueForm myForm = (AddLeagueForm) form;
36
37         // Perform business logic
38         LeagueService leagueSvc = new LeagueService();
39         League league = leagueSvc.createLeague(myForm.getYear(),
40                                               myForm.getSeason(),
41                                               myForm.getTitle());
42         // Store the new league in the request-scope
43         request.setAttribute("league", league);
44
45         // Send the Success view
46         return mapping.findForward("success");
47
```



## The AddLeagueAction Class (Part 3)

```
46     return mapping.findForward("success");
47
48     // Handle any unusual exceptions
49     } catch (RuntimeException e) {
50
51         // Log stack trace
52         e.printStackTrace(System.err);
53
54         // Record the error
55         errors.add(ActionErrors.GLOBAL_ERROR,
56                 new ActionError("error.unexpectedError",
57                                 e.getMessage()));
58
59         // and forward to the error handling page (the form itself)
60         return mapping.findForward("error");
61
62     } // END of try-catch block
63
64 } // END of execute method
65
66 } // END of AddLeagueAction class
```



## Objectives

- Describe the components in a Struts application
- Develop an `ActionForm` class
- Develop a JSP page for a View form
- Configure the View forms



## Developing the JSP Code for a View Form

- Struts provides several custom tag libraries for use in JSP pages.
- The `html` tag library has tags that make form development easier.
  - Scripting of HTML form components
  - Repopulation of form fields is automatic

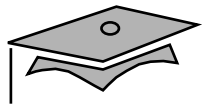


## Struts `html` Tag Library Overview

<b>Tag</b>	<b>Purpose</b>
<code>form</code>	Defines an HTML form
<code>text</code>	Renders a TEXT input element
<code>radio</code>	Renders a radio button input field
<code>submit</code>	Renders a Submit button
<code>image</code>	Renders an image input element
<code>img</code>	Renders an HTML <code>img</code> tag
<code>link</code>	Renders an HTML anchor tag
<code>errors</code>	Displays error messages conditionally

These are only some of the tags in the `html` tag library.





## The add\_league.jsp Page

```
1  <%@ page session="false" %>
2  <%@ taglib prefix="c"
3      uri="http://java.sun.com/jsp/jstl/core" %>
4  <%@ taglib prefix="html"
5      uri="http://jakarta.apache.org/struts/tags-html" %>
6
7  <!-- Set page title --%>
8  <c:set var="pageTitle">Add a New League</c:set>
9
10 <!-- Create business services --%>
11 <jsp:useBean id="leagueSvc" class="sl314.model.LeagueService" />
12
13 <!-- Generate the HTML response --%>
14 <html>
15 <head>
16     <title>Duke's Soccer League: ${pageTitle}</title>
17 </head>
18 <body bgcolor='white'>
19
```



## The add\_league.jsp Page (Part 2)

```
14 <html>
15 <head>
16   <title>Duke's Soccer League: ${pageTitle}</title>
17 </head>
18 <body bgcolor='white'>
19
20 <!-- Generate page heading --%>
21 <!-- Page Heading -->
22 <table border='1' cellpadding='5' cellspacing='0' width='400'>
23 <tr bgcolor='#CCCCFF' align='center' valign='center' height='20'>
24   <td><h3>Duke's Soccer League: ${pageTitle}</h3></td>
25 </tr>
26 </table>
27
28 <!-- Report any errors (if any) --%>
29 <html:errors />
30
31 <!-- Generate main body --%>
32 <p>
33 This form allows you to create a new soccer league.
34 </p>
```



## The add\_league.jsp Page (Part 3)

```
33 This form allows you to create a new soccer league.
34 </p>
35 <html:form action="/admin/add_league.do" method="POST"
36           focus="yearStr">
37 <!-- Repopulate the year field -->
38 Year: <html:text property="yearStr" />
39 <br/><br/>
40 <!-- Repopulate the season drop-down menu -->
41 Season:
42 <html:select property='season'>
43 <c:forEach var="season" items="${leagueSvc.allSeasons}">
44   <html:option value="${season}">${season}</html:option>
45 </c:forEach>
46 </html:select>
47 <br/><br/>
48 <!-- Repopulate the title field -->
49 Title: <html:text property="title" />
50 <br/><br/>
51 <!-- The submit button -->
52 <html:submit value="Add League" />
53 </html:form>
```



## Objectives

- Describe the components in a Struts application
- Develop an `ActionForm` class
- Develop a JSP page for a View form
- **Configure the View forms**



## Configuring the View Forms

Configure the form beans as follows:

- Form beans are configured in the Struts configuration file:

```
10 <form-beans>
11   <form-bean name="selectLeagueForm"
12             type="sl314.view.SelectLeagueForm" />
13   <form-bean name="enterPlayerForm"
14             type="sl314.view.EnterPlayerForm" />
15   <form-bean name="selectDivForm"
16             type="sl314.view.SelectDivisionForm" />
17   <form-bean name="addLeagueForm"
18             type="sl314.view.AddLeagueForm" />
19 </form-beans>
```

- They are named so they can be used later within action elements.



## Configure the View Aspects of the Actions

- Action view aspects are also configured in the Struts configuration file:

```
48
49 <!-- Declare the /admin/add_league.do action -->
50 <action path="/admin/add_league"
51         type="sl314.controller.AddLeagueAction"
52         name="addLeagueForm" scope="request" validate="true"
53         input="/admin/add_league.jsp" >
54     <forward name="success" path="/admin/success.jsp"/>
55     <forward name="error" path="/admin/add_league.jsp"/>
56 </action>
```

- The name, scope, validate, and input attributes are used for this configuration.



## Summary

- Struts provides a mechanism to store form data into a JavaBeans instance. This helps separate view processing logic (parameter retrieval, data conversion, data verification) from the controller logic.
- You create a form bean by extending the Struts `ActionForm` class and providing accessor and mutator methods for each form field.
- You can also perform data conversion within your `ActionForm` class.
- The `validate` method lets you perform verification of the form fields.
- The controller classes can access the form bean for this action.



# Module 14

## Building Reusable Web Presentation Components





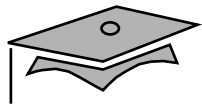
## Objectives

- Describe how to build web page layouts from reusable presentation components
- Include JSP segments
- Develop layouts using the Struts Tiles framework

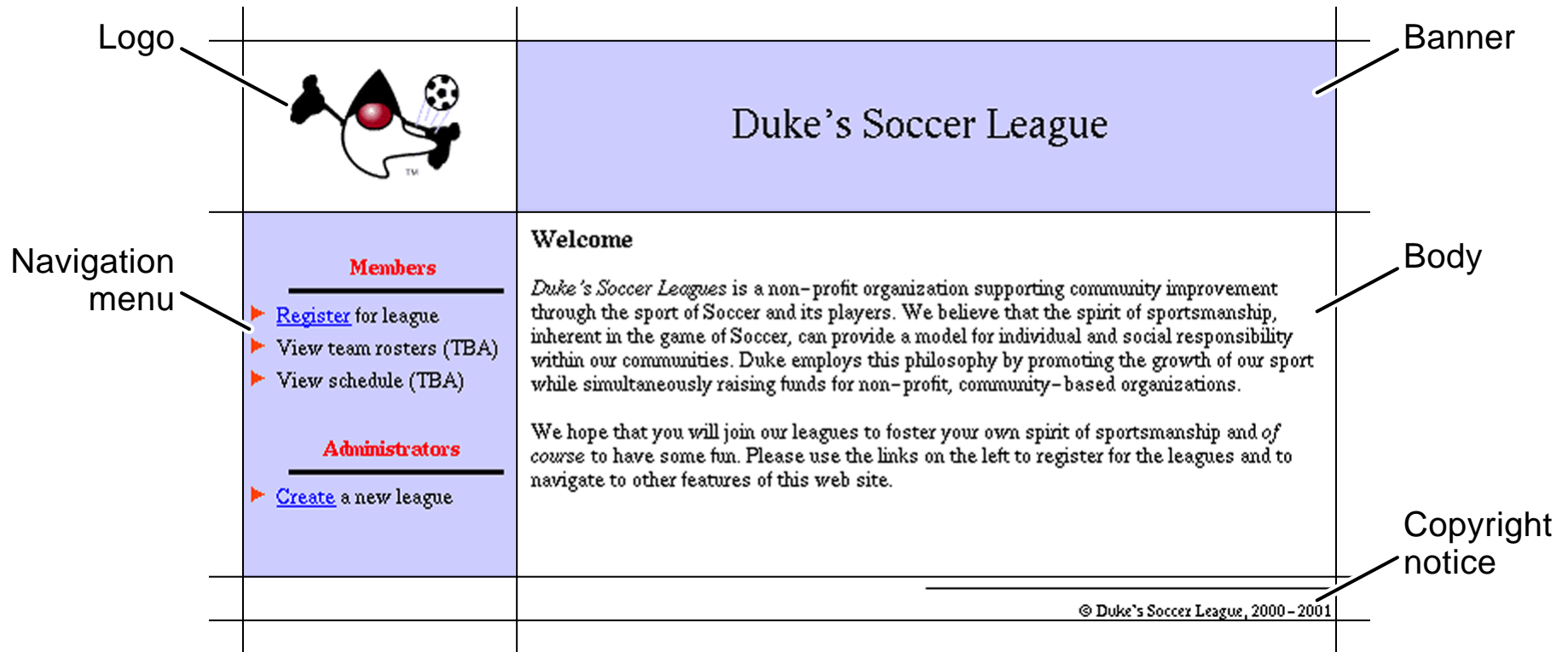


## Relevance

- So far the Soccer League pages have been fairly simple. What HTML technique could you use to facilitate a more rich layout?
- If you have a navigation menu as part of your layout, what issues will you have if you need to build a web application with dozens of pages?
- What if the actual layout of the pages changes? How will you update the layouts of every page in the web application?



# 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'> <!-- navigation 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>
```



## Presentation Segment Overview

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

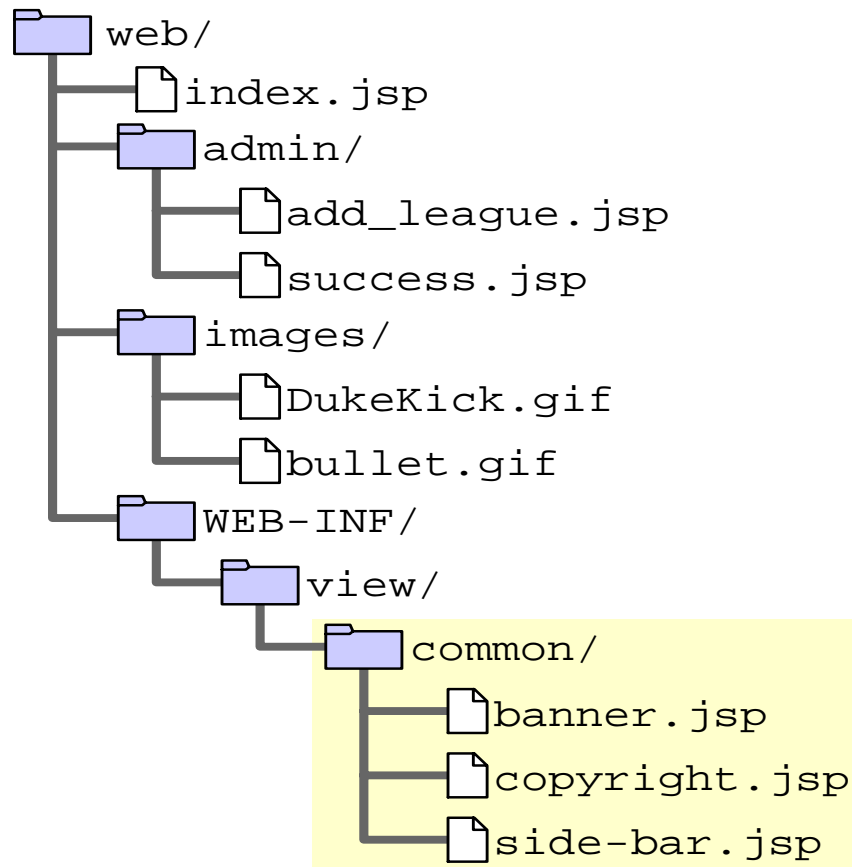
```
1  <%@ taglib prefix="myTags" uri="/WEB-INF/myTags.tld" %>
2
3      <spacer height='15'>
4      <hr width='50%' align='right' size='1' noshade color='blue'>
5      <font size='2' face='Helvetica, san-serif'>
6      &copy; Duke's Soccer League, 2000-<myTags:getCurrentYear />
7      </font>
```

**Note:** Segments should not contain `html`, `head`, or `body` tags.



# Organizing Presentation Segments

You should isolate your reusable segments:





## Organizing Presentation Segments

- Content pages can be anywhere in the web application
- If stored with other content (such as images), the content segments can be accessed directly from a client browser
- You can protect content from direct access by a browser by storing the segments under the `WEB-INF` directory



## Objectives

- Describe how to build web page layouts from reusable presentation components
- **Include JSP segments**
- Develop layouts using the Struts Tiles framework





## Including JSP Page Segments

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

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



## Using the `include` Directive

The `include` directive lets you include a segment into the text of the main JSP page at translation time.

- Syntax:

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

- Example:

```
75 <!-- START of copyright notice -->
76 <td align='right' width='480'>
77     <%@ include file="/WEB-INF/view/common/copyright.jsp" %>
78 </td>
79 <!-- END of copyright notice -->
```



## Using the `jsp:include` Standard Action

The `jsp:include` action lets you include a segment into the text of the HTTP response at runtime.

- **Syntax:**

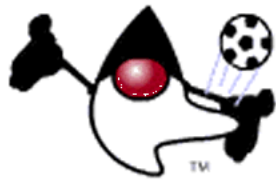
```
<jsp:include page="segmentURL" />
```

- **Example:**

```
36 <!-- START of navigation menu -->
37 <td bgcolor='#CCCCFF' width='160' align='left'>
38   <jsp:include page="/WEB-INF/view/common/navigation.jsp" />
39 </td>
40 <!-- END of navigation menu -->
```



# Using the `jsp:param` Standard Action



Soccer League (Spring '01)

Thank You!

## Members

- ▶ [Register](#) for league
- ▶ View team rosters (TBA)
- ▶ View schedule (TBA)

## Administrators

- ▶ [Create](#) a new league

Thank you, Bryan, for registering in the **Soccer League (Spring '01)** league.



## Using the `jsp:param` Standard Action

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

For example, in the Soccer League home page:

```
24      <!-- START of banner -->
25      <jsp:include page="/WEB-INF/view/common/banner.jsp">
26          <jsp:param name="subTitle" value="Welcome" />
27      </jsp:include>
28      <!-- END of banner -->
```



## Using the `jsp:param` Standard Action

The `subTitle` parameter is attached to the request object:

```
12
13 <font size='5' face='Helvetica, san-serif'>
14   ${bannerTitle}
15 </font>
16
17 <c:if test="${not empty param.subTitle}">
18   <br/><br/>
19   <font size='4' face='Helvetica, san-serif'>
20     ${param.subTitle}
21   </font>
22 </c:if>
```



## Objectives

- Describe how to build web page layouts from reusable presentation components
- Include JSP segments
- Develop layouts using the Struts Tiles framework



## Developing Layouts Using Struts Tiles

The basic idea of Tiles is to have a single (or small number) of layout files, rather than duplicating the layout code from one page to another.

- Views call the layout file
- The layout file provides the layout and dynamically includes information provided by the views





## The layoutPage.jsp Page

```
1  <%@ taglib prefix="tiles"
2      uri="http://jakarta.apache.org/struts/tags-tiles" %>
3  <%@ taglib prefix="c"
4      uri="http://java.sun.com/jsp/jstl/core" %>
5
6  <%-- Generate the HTML response --%>
7  <html>
8
9  <head>
10 <title>Duke's Soccer League: <tiles:getAsString name="subTitle"/></title>
11 </head>
12
13 <body bgcolor='white'>
14
```



## The layoutPage.jsp Page (Part 2)

```
14
15 <table border='0' cellspacing='0' cellpadding='0' width='640'>
16
17 <tr height='100'>
18
19     <td align='center' valign='center' width='160' height='100'>
20         <img src='<c:url value="/images/DukeKick.gif"/>'
21             alt='Duke's Soccer League Logo'>
22     </td>
23
24     <td bgcolor='#CCCCFF' align='center' valign='center' width='480'
height='100'>
25         <!-- START of banner -->
26         <c:set var="subTitle"><tiles:getAsString name='subTitle' /></c:set>
27         <jsp:include page="/WEB-INF/view/common/banner.jsp">
28             <jsp:param name="subTitle" value="\${subTitle}" />
29         </jsp:include>
30         <!-- END of banner -->
31     </td>
32
33 </tr>
```



## The layoutPage.jsp Page (Part 3)

```
34
35 <tr valign='top'>
36
37     <!-- START of navigation menu -->
38     <td bgcolor='#CCCCFF' width='160' align='left'>
39         <jsp:include page="/WEB-INF/view/common/navigation.jsp" />
40     </td>
41     <!-- END of navigation menu -->
42
43     <td width='480' align='left'>
44         <div style='margin-top: 0.1in; margin-left: 0.1in;
45             margin-bottom: 0.1in; margin-right: 0.1in'>
46             <!-- START of main content -->
47             <tiles:insert attribute='body' />
48             <!-- END of main content -->
49         </div>
50     </td>
51
52 </tr>
53
```



## The layoutPage.jsp Page (Part 4)

```
53
54 <tr>
55
56   <td width='160'>
57     <!-- nothing here -->
58   </td>
59
60   <!-- START of copyright notice -->
61   <td align='right' width='480'>
62     <%@ include file="/WEB-INF/view/common/copyright.jsp" %>
63   </td>
64   <!-- END of copyright notice -->
65
66 </tr>
67
68 </table>
69
70 </body>
71 </html>
```



## Tiles Layout

View pages can include the layout page, passing information as Tiles variables.

For example, the Registration Thank You page is:

```
1  <%@ taglib prefix="tiles"
2      uri="http://jakarta.apache.org/struts/tags-tiles" %>
3
4  <tiles:insert page="/WEB-INF/view/layout/layoutPage.jsp">
5      <tiles:put name="subTitle" value="Thank You"/>
6      <tiles:put name="body" value="/WEB-INF/view/register/thank_you.jsp"/>
7  </tiles:insert>
```

- The variables `subTitle` and `body` provide content
- Other views would provide different content



## Content Body

Content files are segments that provide only the content you want to have in that part of the layout.

```
1  <%@ page session="true" %>
2
3  <p>
4  Thank you, ${sessionScope.player.name}, for registering for
5  the <i>${sessionScope.league.title}</i> league.
6  </p>
```



## Summary

- Most modern web sites use graphically rich layouts.
- Graphically rich layouts include a lot of bulky HTML code to structure the hidden tables that create the page layout.
- The Tiles framework can help organize the layout code into a separate, easily maintained file.
- The layout file then includes various presentation segments.
  - Some segments are reusable components, such as banners and navigation menus.
  - Some segments are the actual body content of the page.