Web Component Development With Servlet and JSP™ Technologies

SL314
Revision C

Course Contents

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
- 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
- Developing Web Applications Using JavaServer™ Faces

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 Using UML*
- Java Platform, Enterprise Edition – Covered in WJT-310: *Java™ Platform, Enterprise Edition: Technology Overview*
- Enterprise JavaBeans™ technology – Covered in SL-351: *Enterprise JavaBeans™ Programming*
- JavaServer™ Faces technology – Covered in DTJ-3108: *Developing JavaServer™ Faces Components With AJAX*

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 such as 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 Platform, Standard Edition (Java SE platform), servlet, and JSP APIs
- 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 JSP 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 (continued)

- ***`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 System (Solaris OS) is different from the Microsoft Windows platform, both commands are shown.

Module 1

Introduction to Web Technologies

Objectives

- Describe web applications
- Describe Java Platform, Enterprise Edition 5 (Java EE 5)
- Describe web application program execution methods and the advantages and disadvantages of each
- Describe Java servlet technology
- Describe JavaServer Pages technology
- Define three-tier architecture
- Define Model-View-Controller (MVC) architecture

Relevance

- What web applications have you developed?
- Did your web technology allow you to achieve your goals?

Web Application Technologies

- HTML over HTTP
- Common Gateway Interface (CGI)
- Servlets
- JavaServer Pages (JSP) technology
- JSP Standard Tag Library (JSTL)
- XML
- Struts
- JavaServer Faces

Java™ EE 5

- Java EE is the industry standard for developing portable, robust, scalable and secure server-side Java applications. Java EE is built on the solid foundation of Java Platform, Standard Edition (Java SE).
- Java EE is a set of coordinated technologies which includes the following web application technologies:
  - Java Servlet 2.5 (Java Specification Requests [JSR] 154)
  - JavaServer Pages 2.1 (JSR 245)
  - JavaServer Pages Standard Tag Library (JSR 52)
  - JavaServer Faces 1.2 (JSR 252)

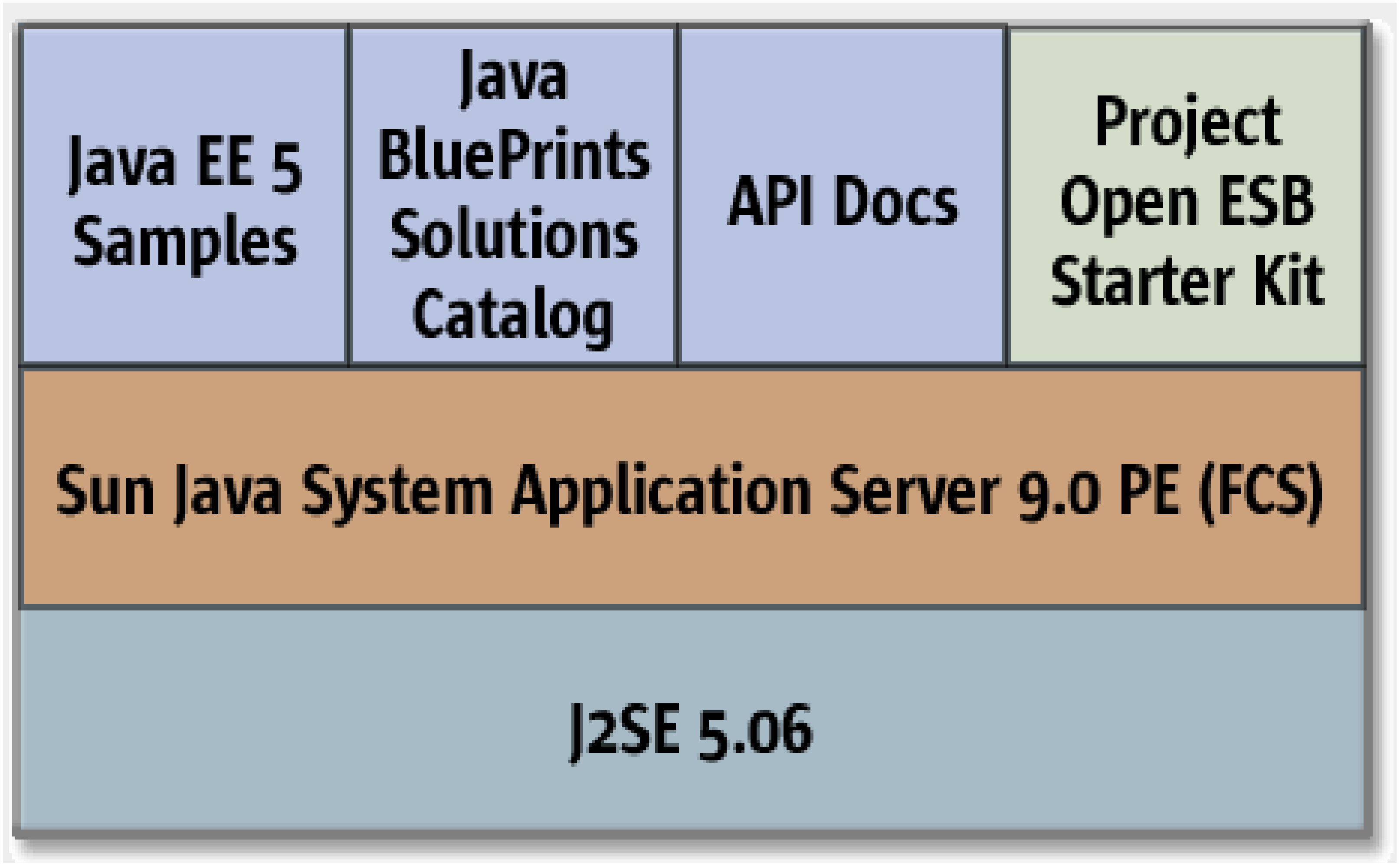
Java EE 5 (continued)

For a complete list of Java technologies, go to:

`http://java.sun.com/javaee/technologies/`

`http://java.sun.com/javase/technologies/`

Java EE 5 SDK

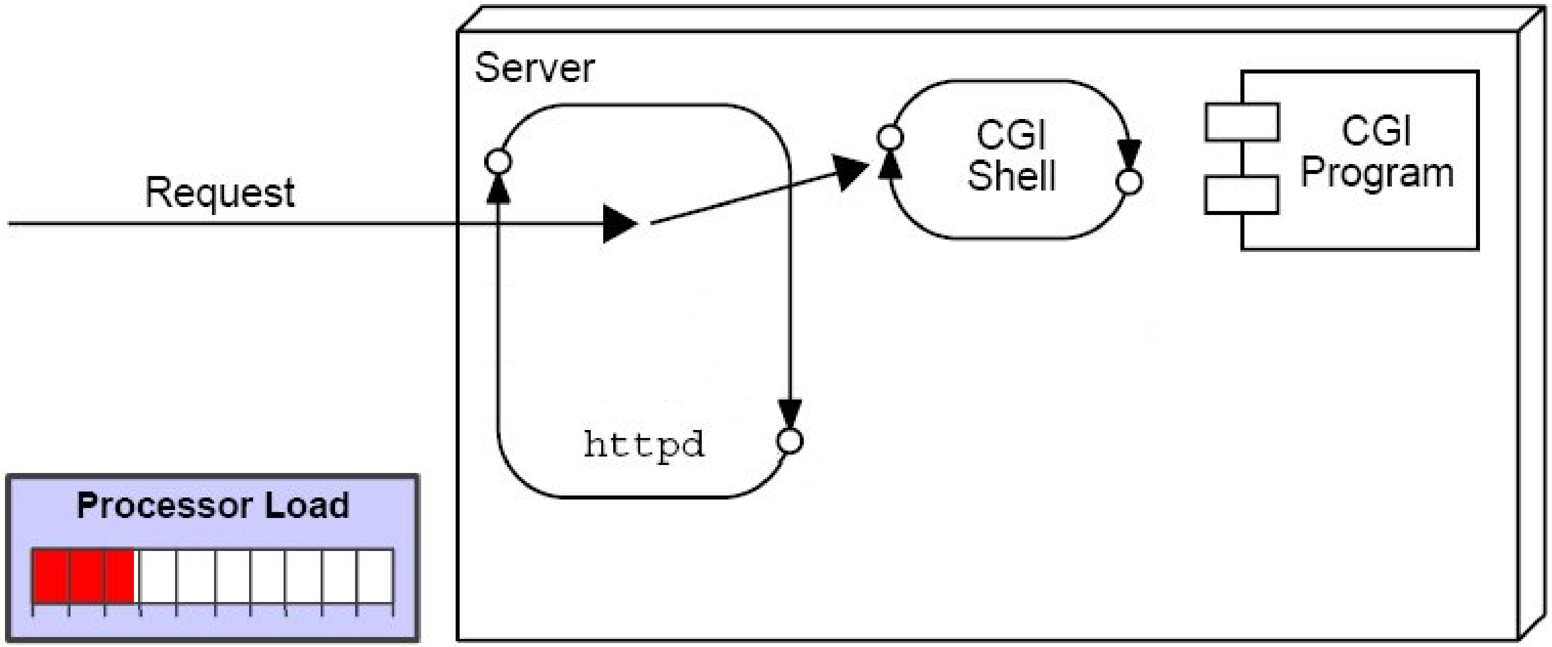

Java EE 5 Samples
Java BluePrints Solutions Catalog
API Docs
Project Open ESB Starter Kit
Sun Java System Application Server 9.0 PE (FCS)
J2SE 5.06

Web Sites and Web Applications

- A web site is a collection of *static* files, HTML pages, graphics, and various other files.
- A web application is a web site with *dynamic* functionality on the server.
- A web application run programs on the server, for example:
  - A browser makes a request, to the server, for an HTML form.
  - The server responds by sending the HTML form back to the browser in an HTTP request stream.
  - Next, the browser sends another request, with data from the HTML form, to the server.
  - The server passes the request and data to a program that responds by sending data back to the browser.

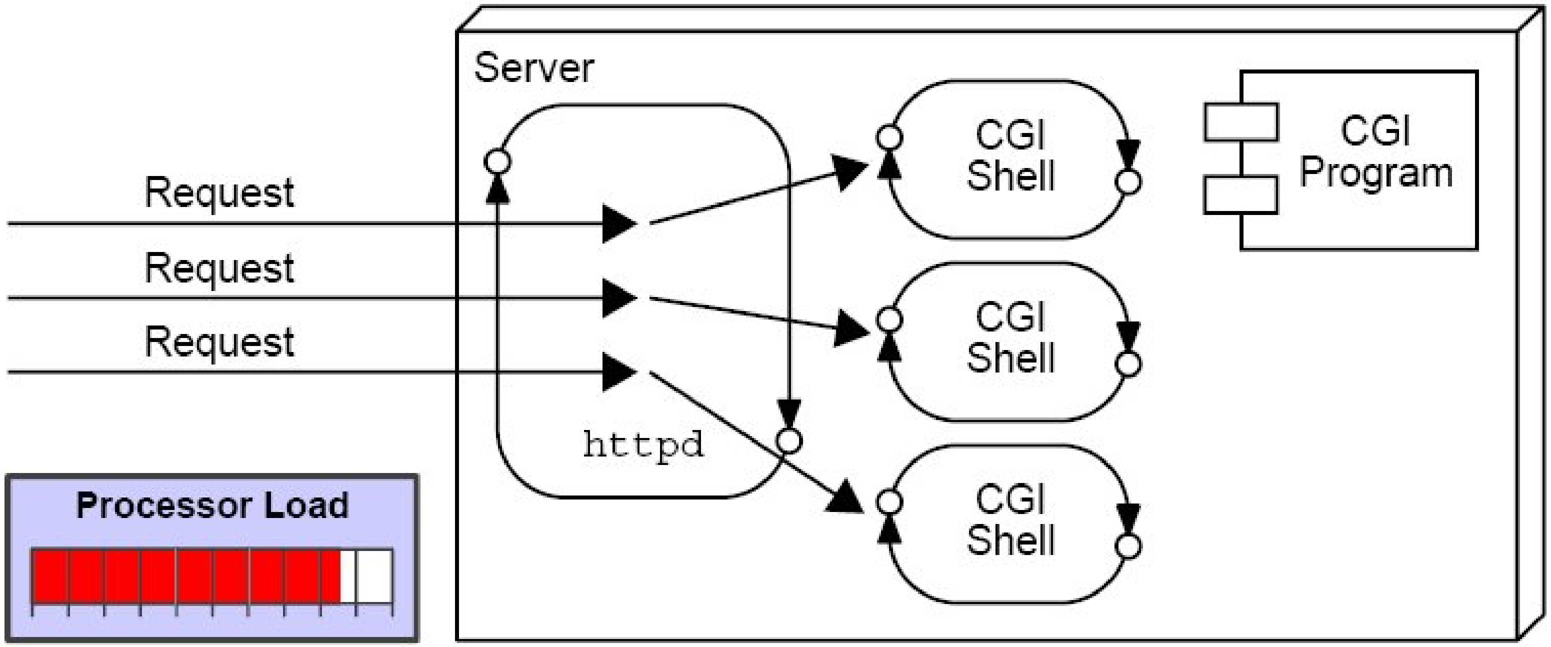
Execution of CGI Programs

One request to one CGI program:



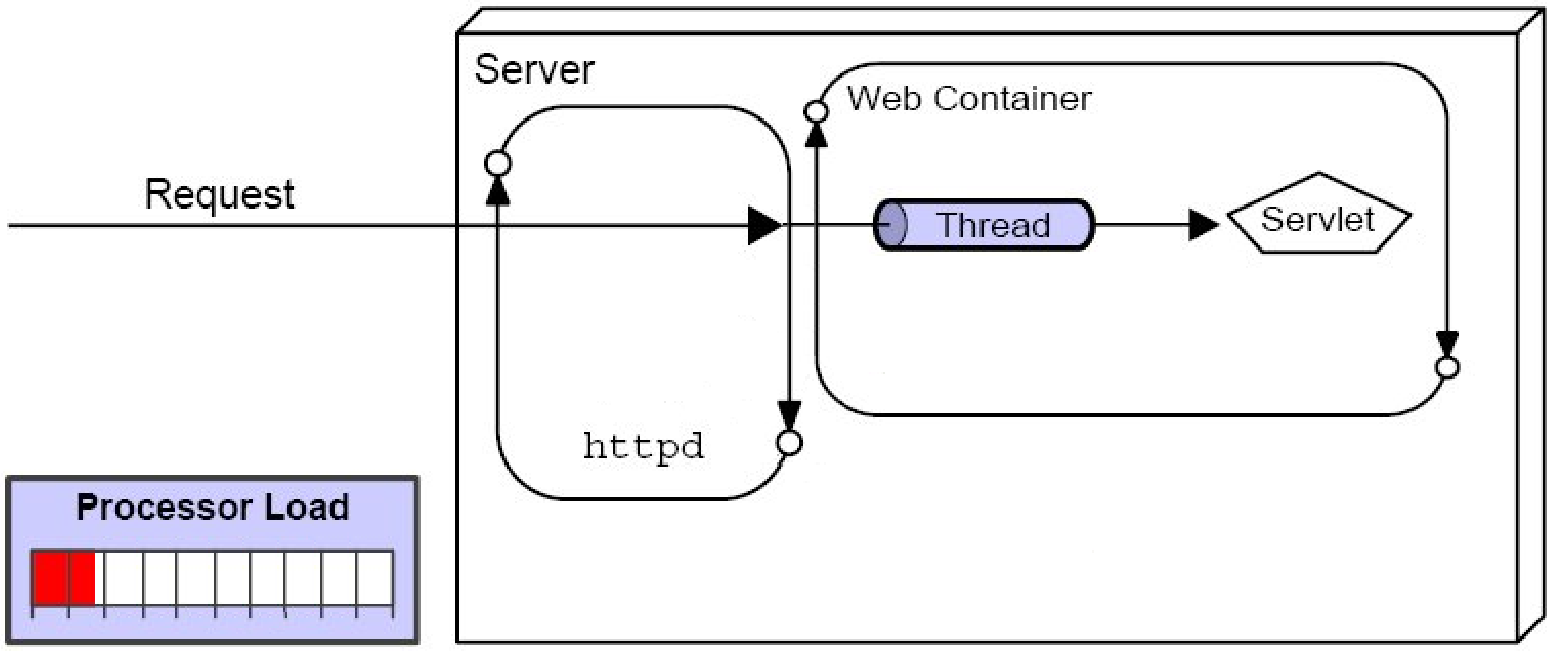
Execution of CGI Programs (continued)

Many requests to one CGI program:



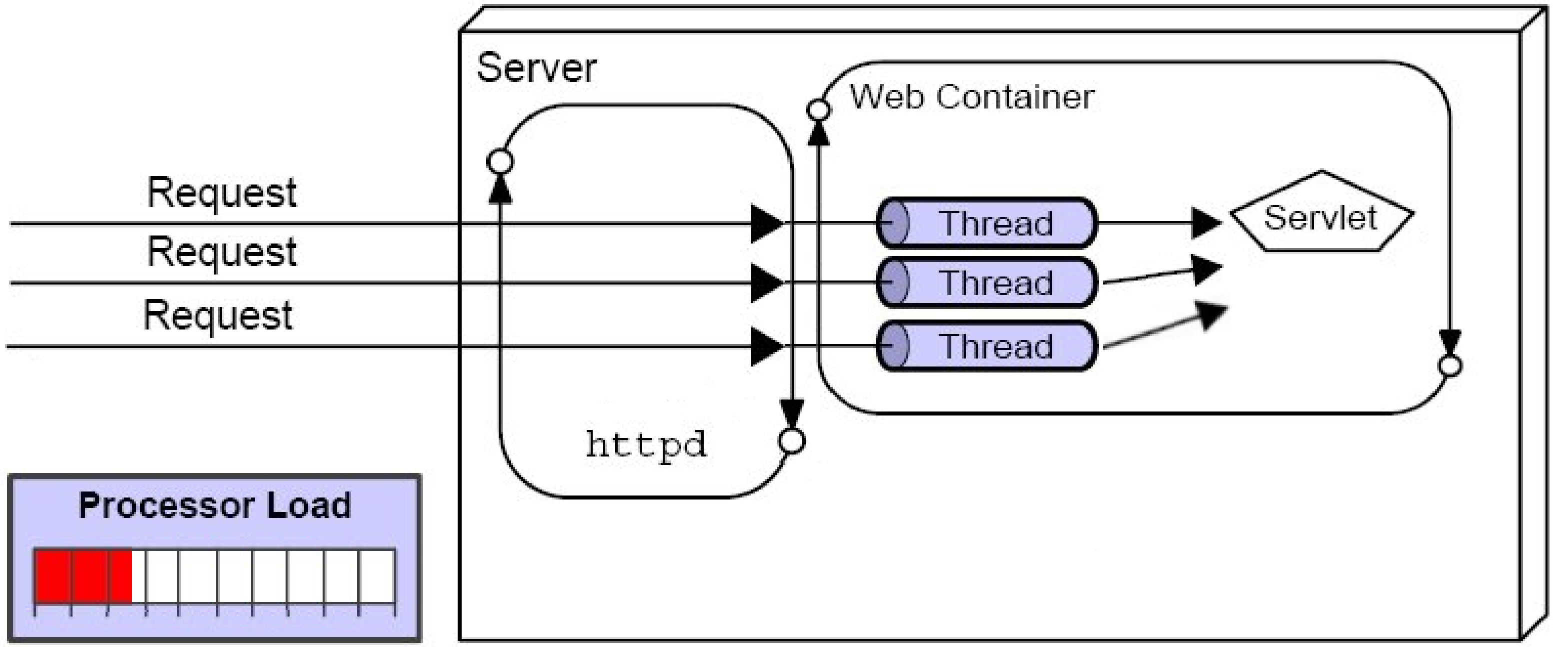
Execution of Java Servlets

One request to one servlet program:



Execution of Java Servlets (continued)

Many requests to one servlet program:



Using Separate Processes or Using Threads

- Advantages of running programs in separate processes over threads:
  - Programs can be written in a variety of languages
  - Web designers can easily reference programs that run in separate processes.
- Advantages of running servlet programs in threads compared with other languages not in threads:
  - The CPU requirements are lower.
  - Java technologies separate processing code (business logic) from the HTML (presentation logic).
  - The Java language is robust and object-oriented.
  - The Java language is platform-independent.

Java Servlets

- A servlet is a Java technology component that executes on the server.
- Servlet programs perform the following:
  - Process HTTP requests
  - Generate dynamic HTTP responses
- A web container is a special Java Virtual Machine (JVM™) tool interface that manages the servlets and a thread pool.

JavaServer Pages™ Technology

- JSP pages are translated into Java servlet classes that are compiled and execute as servlets in the web container.
- JSP pages should focus on the presentation logic, not on the business logic. This makes for a good design.
- In JSP pages, custom tags and JSP Expression Language provide for reusable code and separation of concerns.
- Java code can be embedded into JSP pages.
- In a Java technology web application, JSP pages are often used in conjunction with servlets and business objects in a Model-View-Controller pattern.

Concerns When Using Servlets and JSP™ Technology

Advantages of JSP technology:

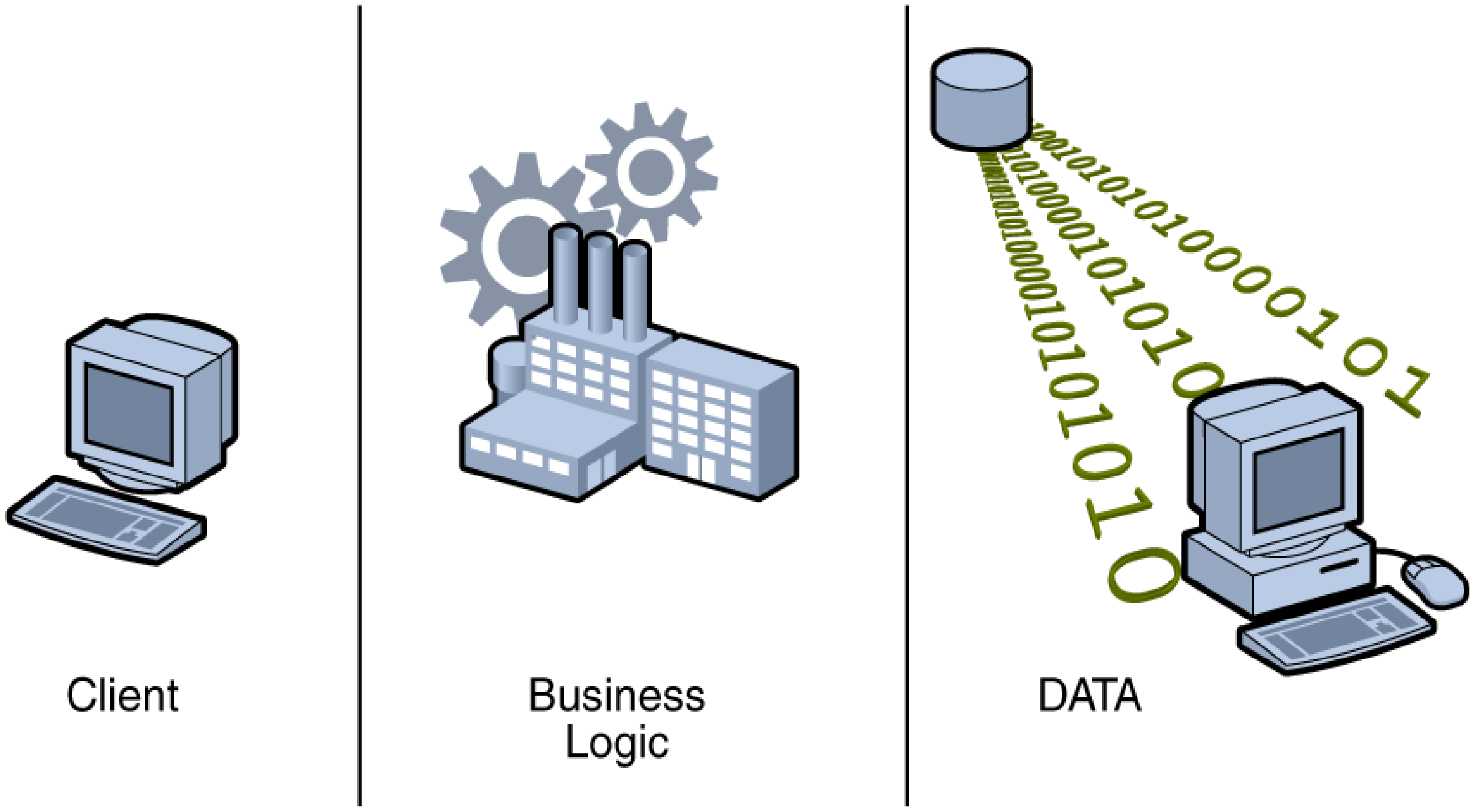
- Provides high performance and scalability because threads are used
- Is built on Java technology, so it is platform-independent.
- Can take advantage of the object-oriented language and its APIs

Concerns When Using Servlets and JSP Technology (continued)

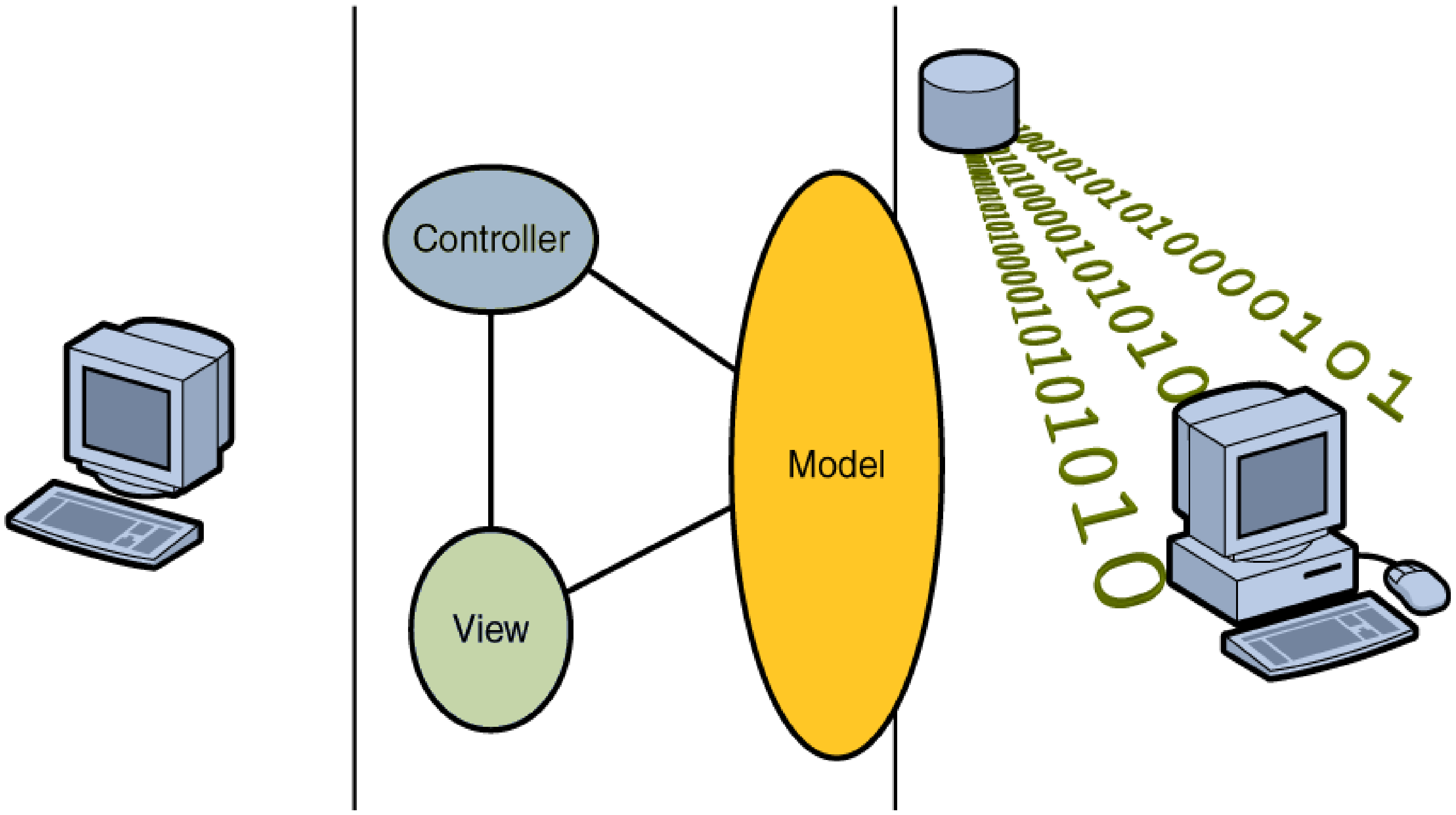
Disadvantages of JSP technology:

- If JSP pages are used in isolation, then the scripting code that performs business and control logic can become cumbersome in the JSP pages. JSP pages are also difficult to debug.
- There is separation of concerns into business logic and presentation logic.
- There are concurrency issues.

Web Application – Three-Tier Architecture

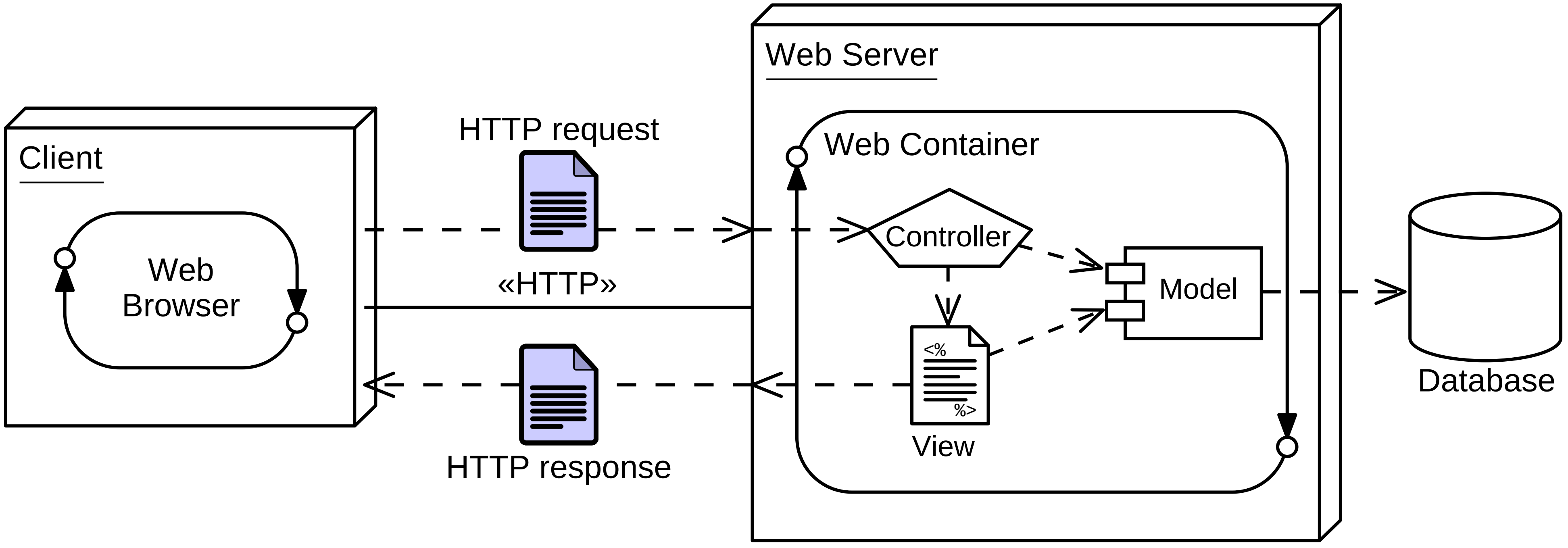


Model-View-Controller (MVC) Architecture in a Web Application



Model 2 Architecture

Deployment diagram of a web container using Model 2 architecture:



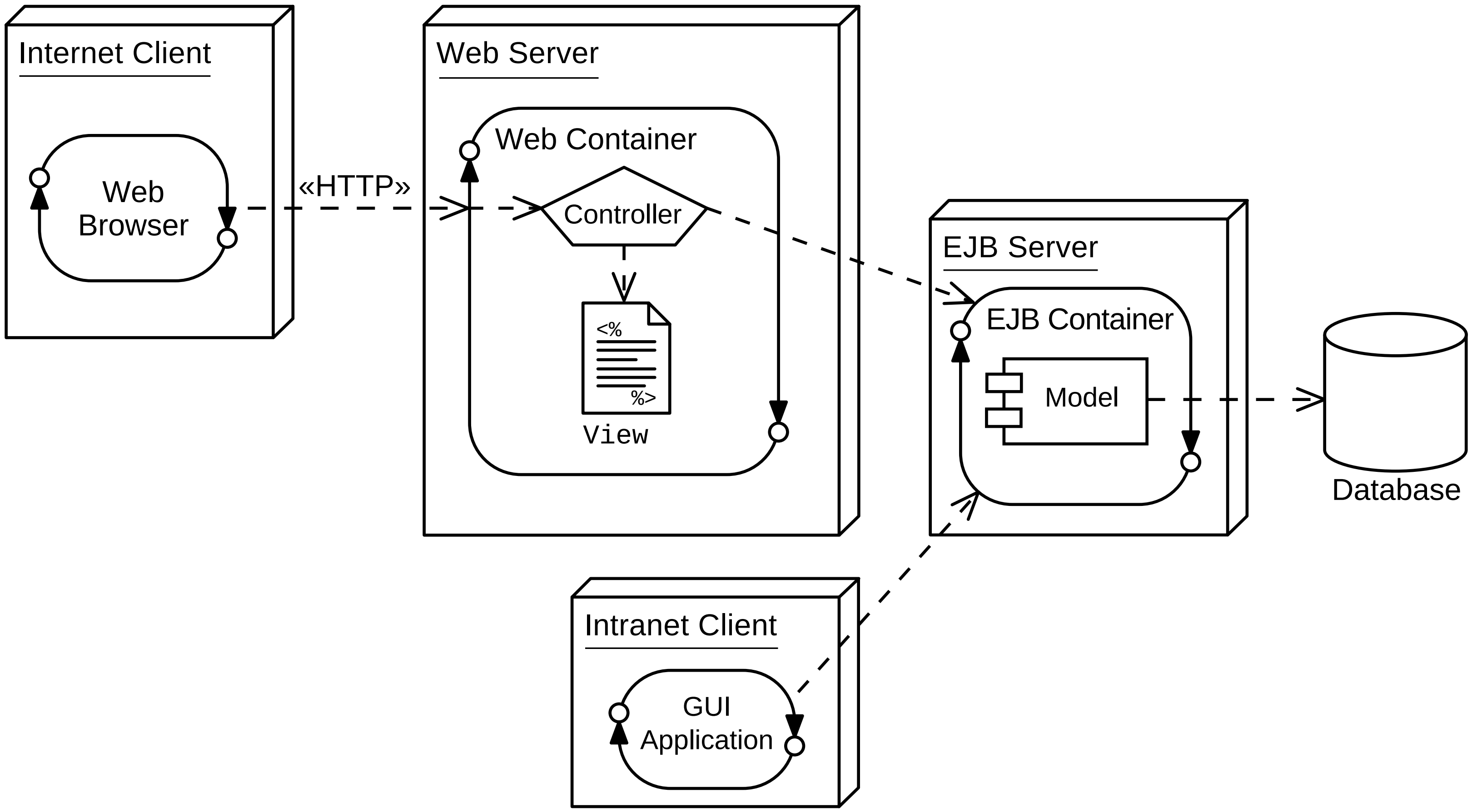
Model 2 Frameworks

- 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

Java EE Containers

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

Java EE Architecture Example



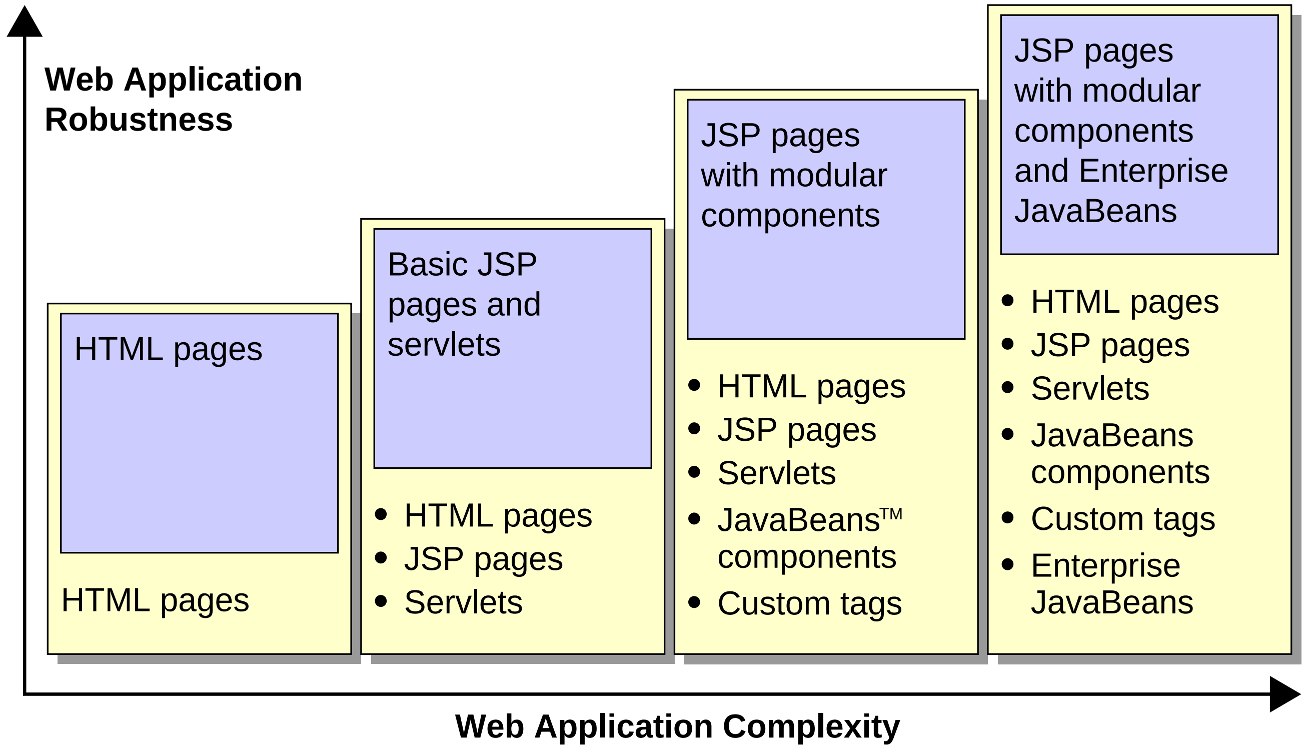
Job Roles

The modularity of Java EE 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

- CGI provided hooks for web servers to execute application programs.
- Java servlets are similar to CGI, but they execute in a JVM using threading.
- JSP pages are similar to 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 Java EE architectures.

Module 2

Developing a View Component

Objectives

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

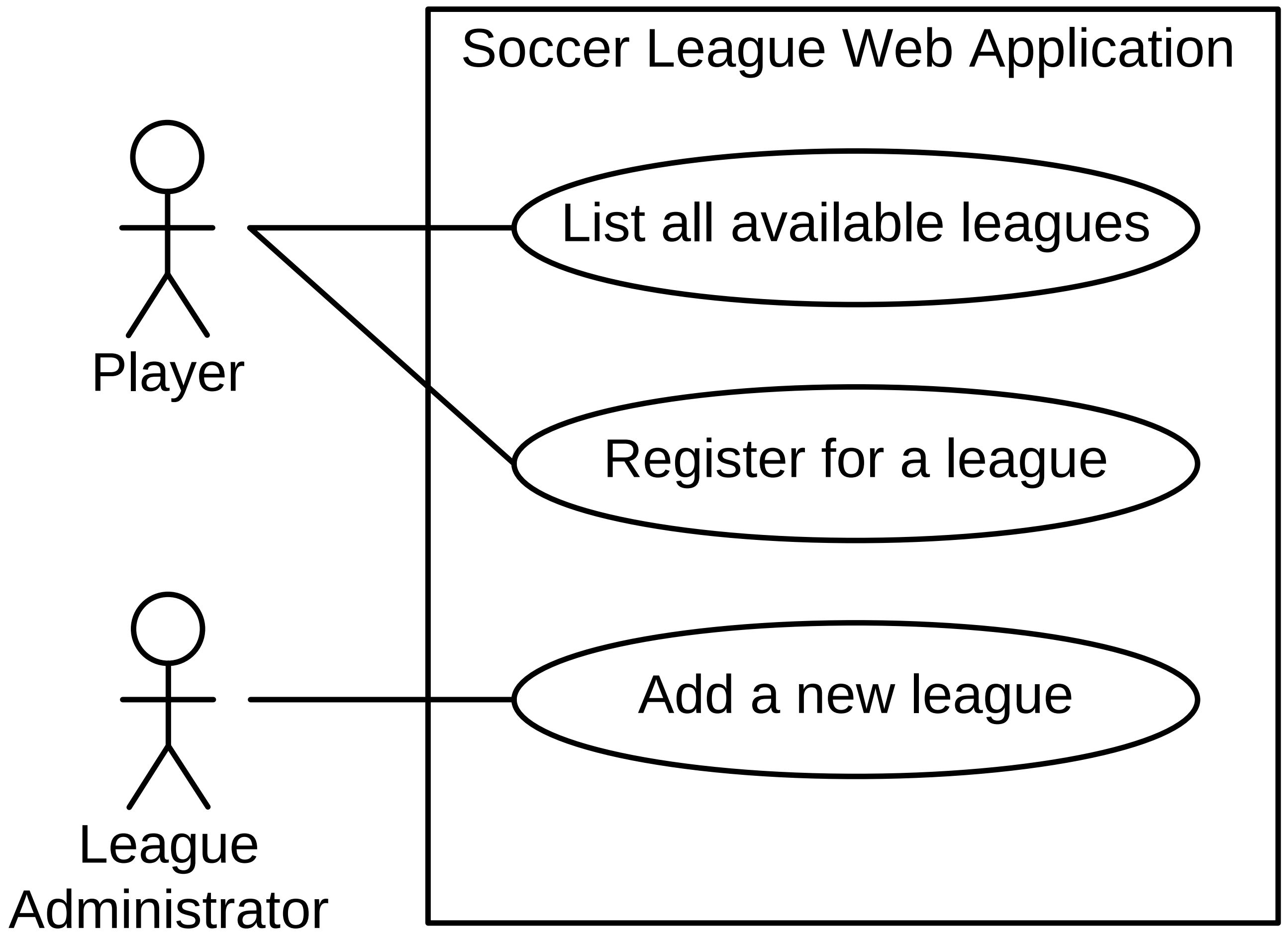
Relevance

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

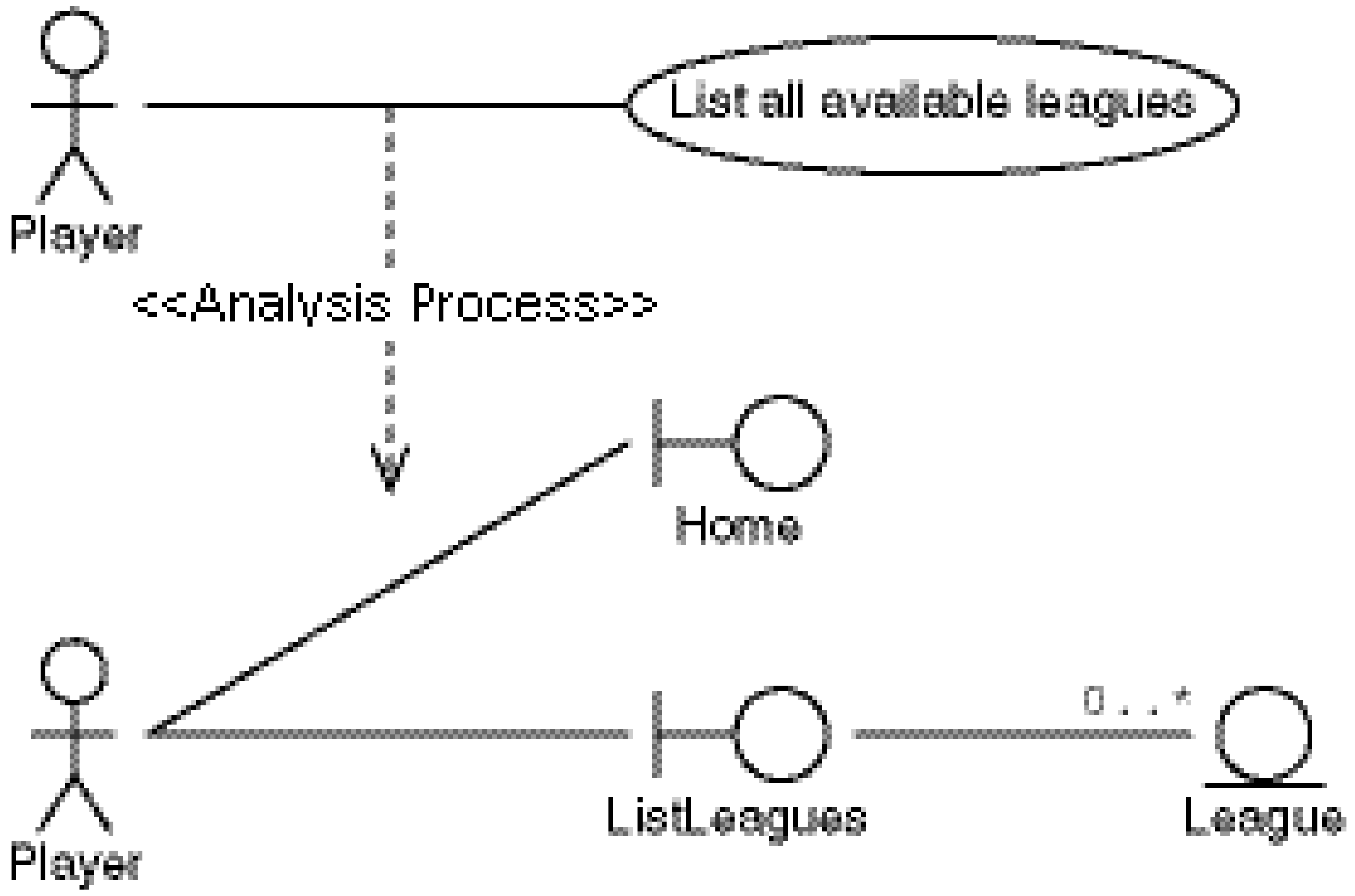
Types of View Components

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

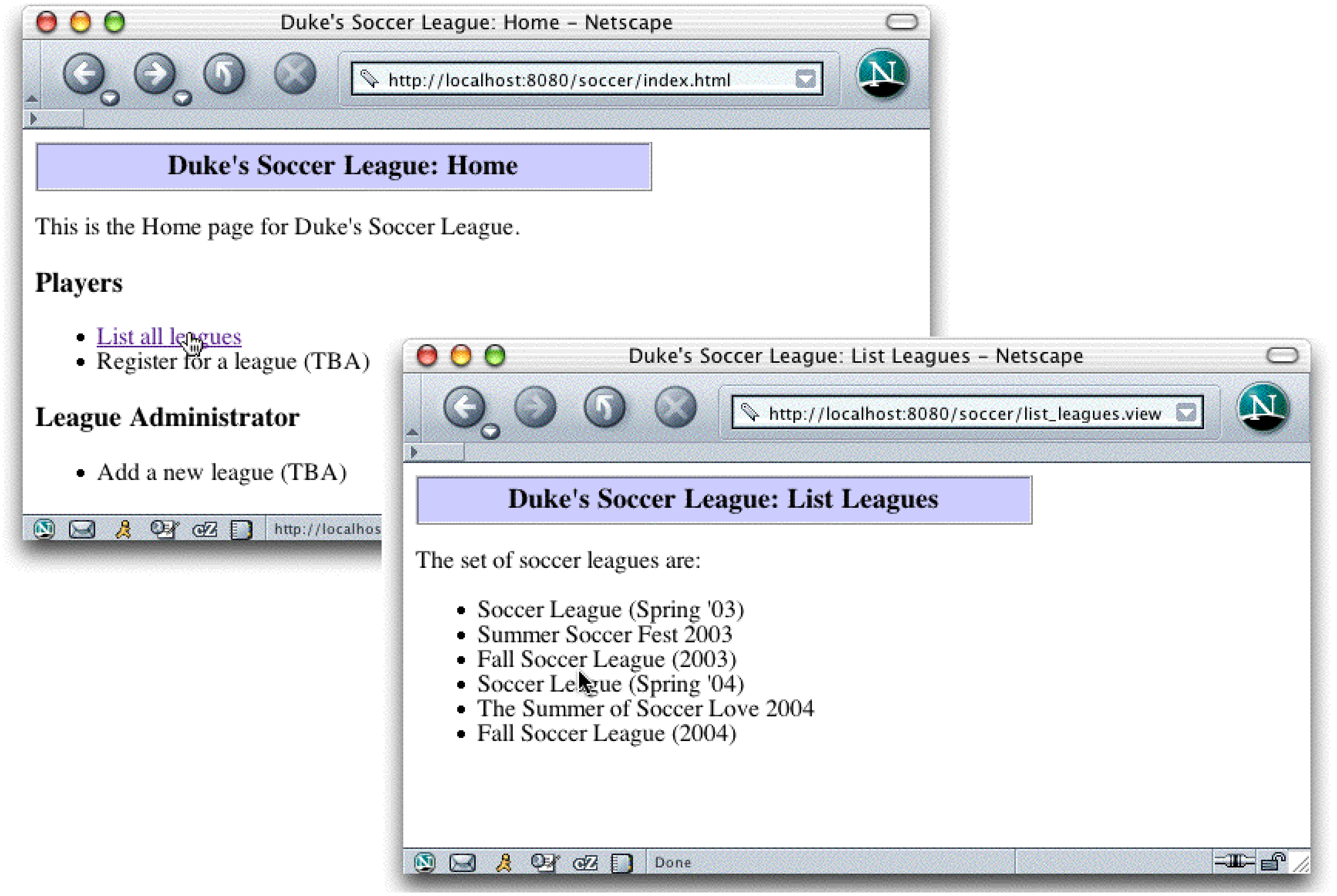
Soccer League Case Study



List Leagues Analysis Model



List Leagues Page Flow


Duke's Soccer League: Home - Netscape
http://localhost:8080/soccer/index.html

Duke's Soccer League: Home

This is the Home page for Duke's Soccer League.

Players

- List all leagues
- Register for a league (TBA)

League Administrator

- Add a new league (TBA)

Duke's Soccer League: List Leagues - Netscape
http://localhost:8080/soccer/list_leagues.view

Duke's Soccer League: List Leagues

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)

Home Page HTML

```
<html>

<head>
<title>Duke’s Soccer League: Home</title>
</head>

<body bgcolor=’white’>

<!-- Page Heading -->
<table border=’1’ cellpadding=’5’ cellspacing=’0’ width=’400’>
<tr bgcolor=’#CCCCFF’ align=’center’ valign=’center’ height=’20’>
  <td><h3>Duke’s Soccer League: Home</h3></td>
</tr>
</table>

```

Home Page HTML (Part 2)

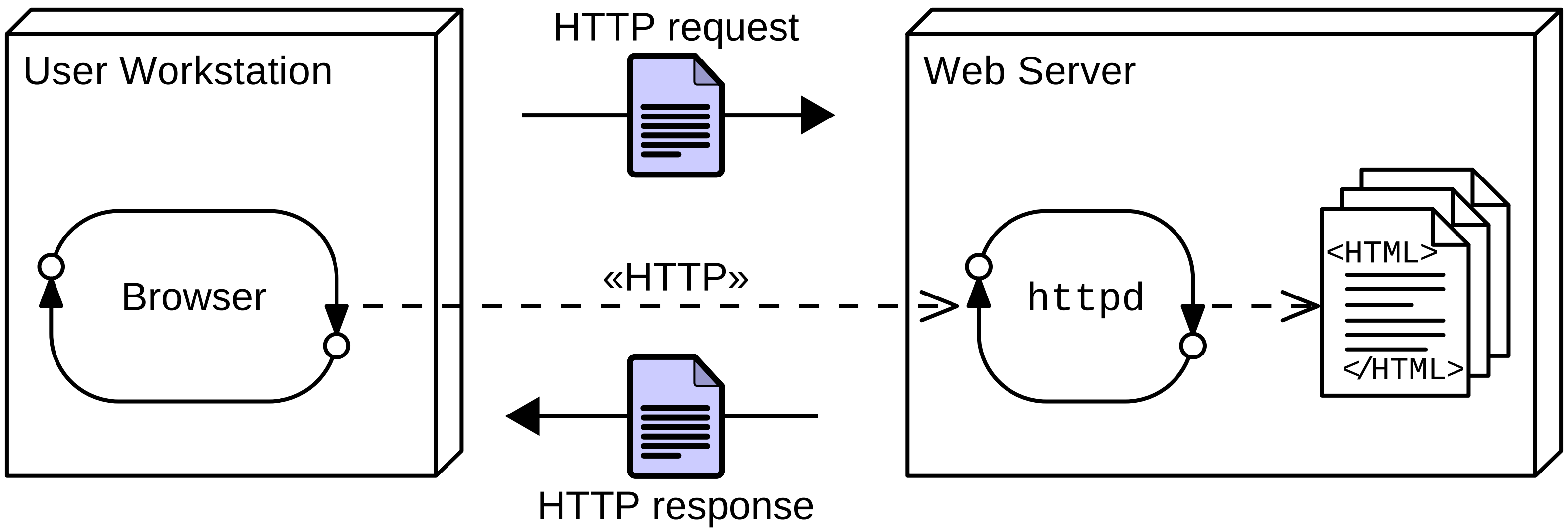
```
<p>
This is the Home page for Duke’s Soccer League.
</p>

<h3>Players</h3>

<ul>
  <li><a href=’list_leagues.view’>List all leagues</a></li>
  <li>Register for a league (TBA)</li>
</ul>

<h3>League Administrator</h3>

<ul>
  <li>Add a new league (TBA)</li>
</ul>

</body>

</html>
```

List Leagues Page HTML

```
<!-- Page Heading -->
<table border='1' cellpadding='5' cellspacing='0' width='400'>
<tr bgcolor='#CCCCFF' align='center' valign='center' height='20'>
  <td><h3>Duke's Soccer League: List Leagues</h3></td>
</tr>
</table>

<p>
The set of soccer leagues are:
</p>

<ul>
  <li>The Summer of Soccer Love 2004</li>
  <li>Fall Soccer League (2003)</li>
  <li>Fall Soccer League (2004)</li>
  <li>Soccer League (Spring '03)</li>
  <li>Summer Soccer Fest 2003</li>
  <li>Soccer League (Spring '04)</li>
</ul>

</body>
</html>
```

Hypertext Transfer Protocol

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

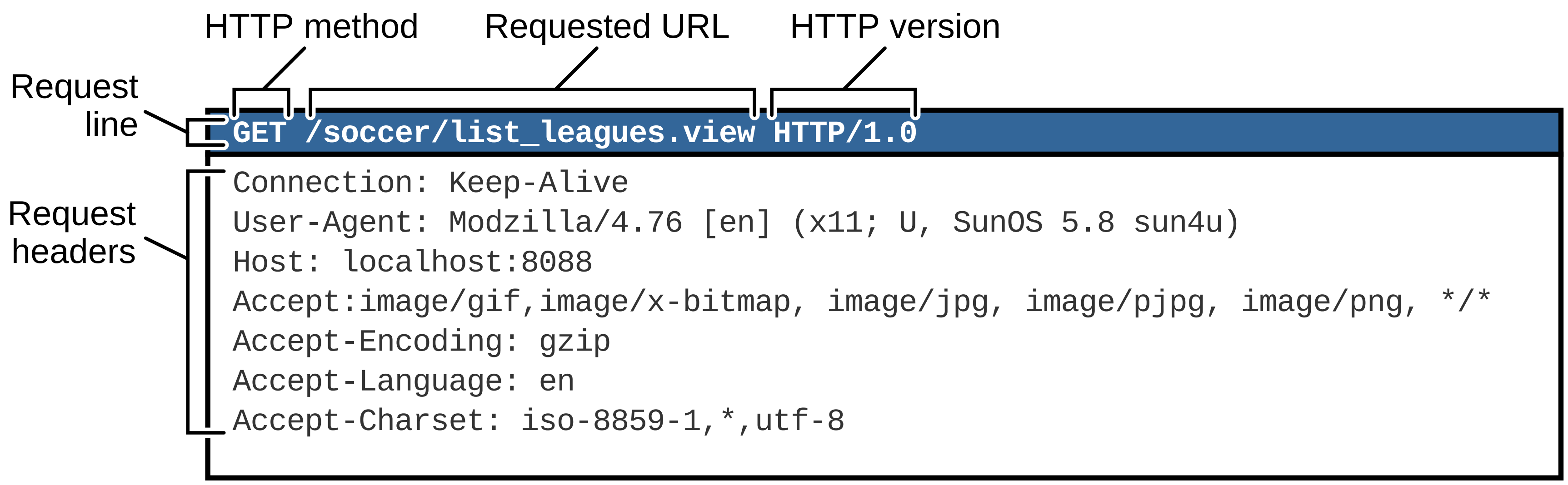


HTTP GET Method

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



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

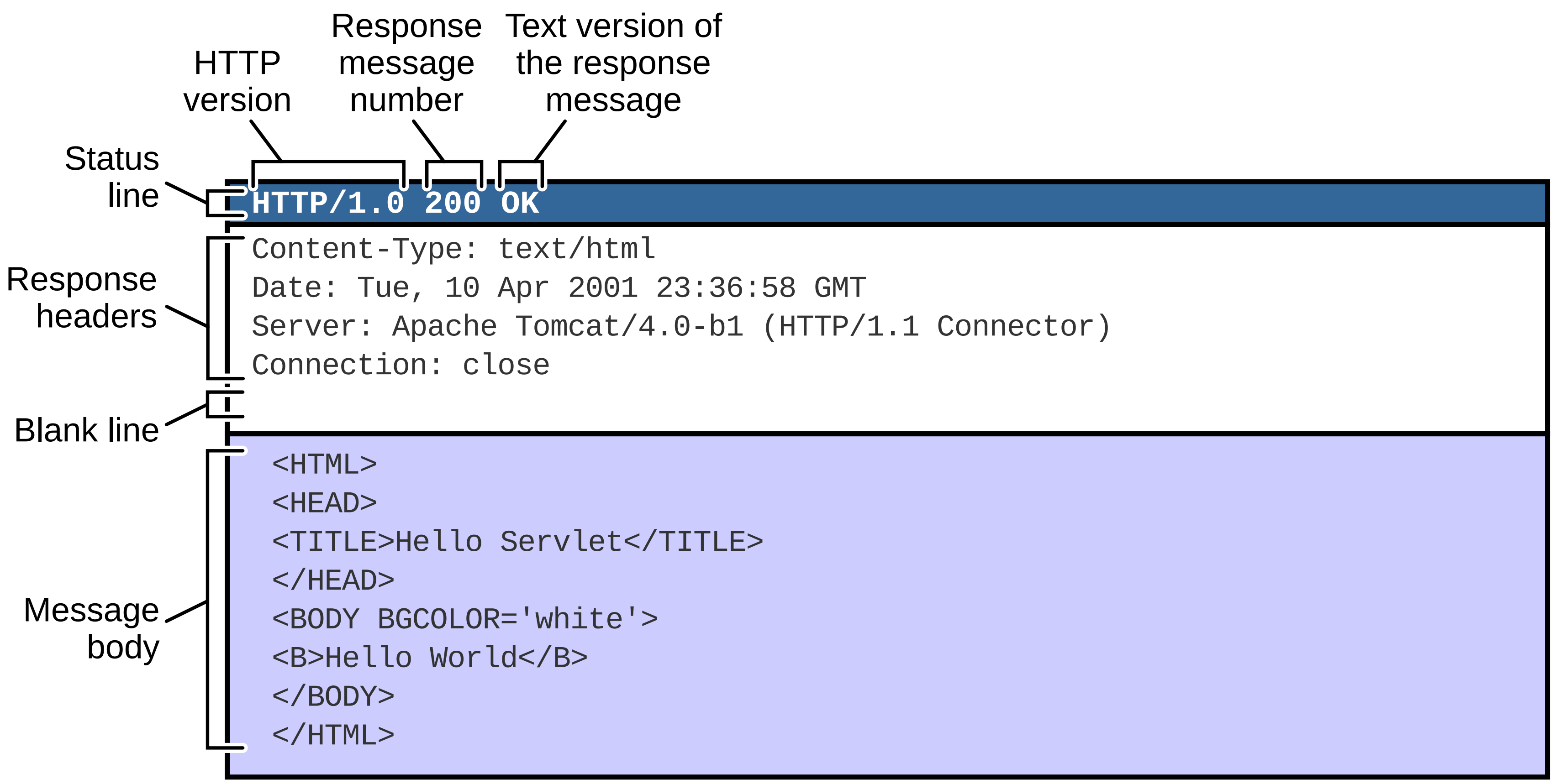
HTTP Request Headers

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

Example headers:

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

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

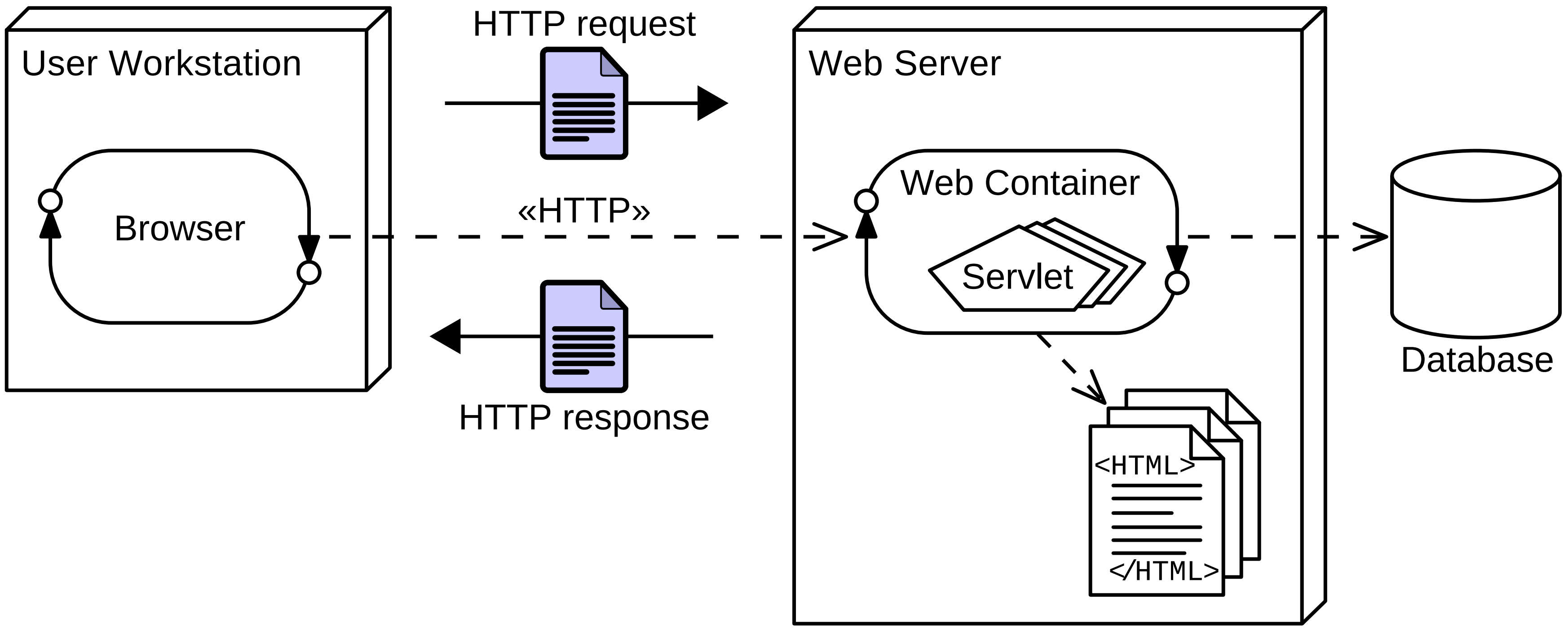
HTTP Response Headers

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

Example headers:

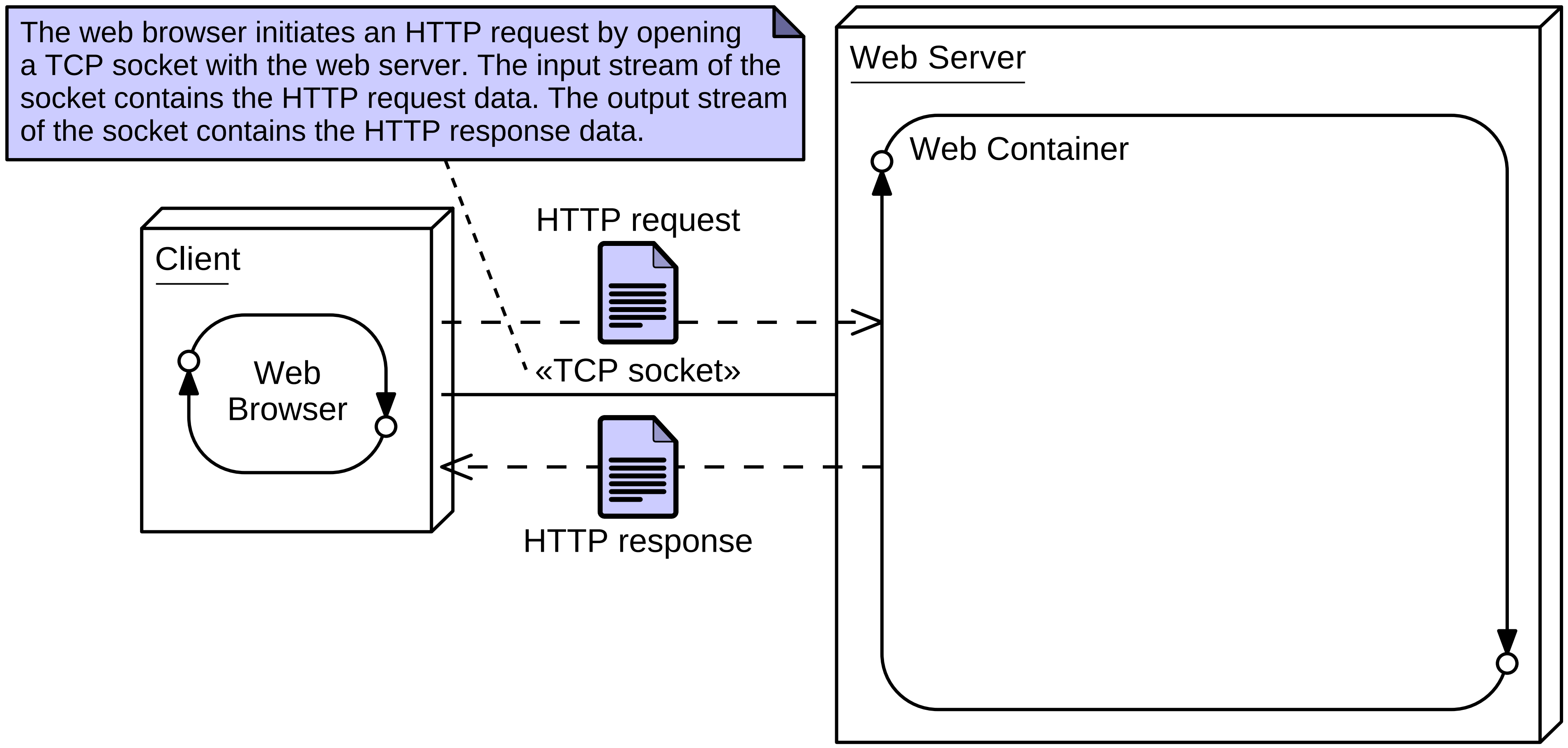
Header	**Use**
Content-Type	A MIME type (such as `text/html`) which classifies the type of data in the response
Content-Length	The length (in bytes) of the payload of the response
Server	An informational string about the server that responded to this HTTP request
Cache-Control	A directive for the web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

Web Container Architecture

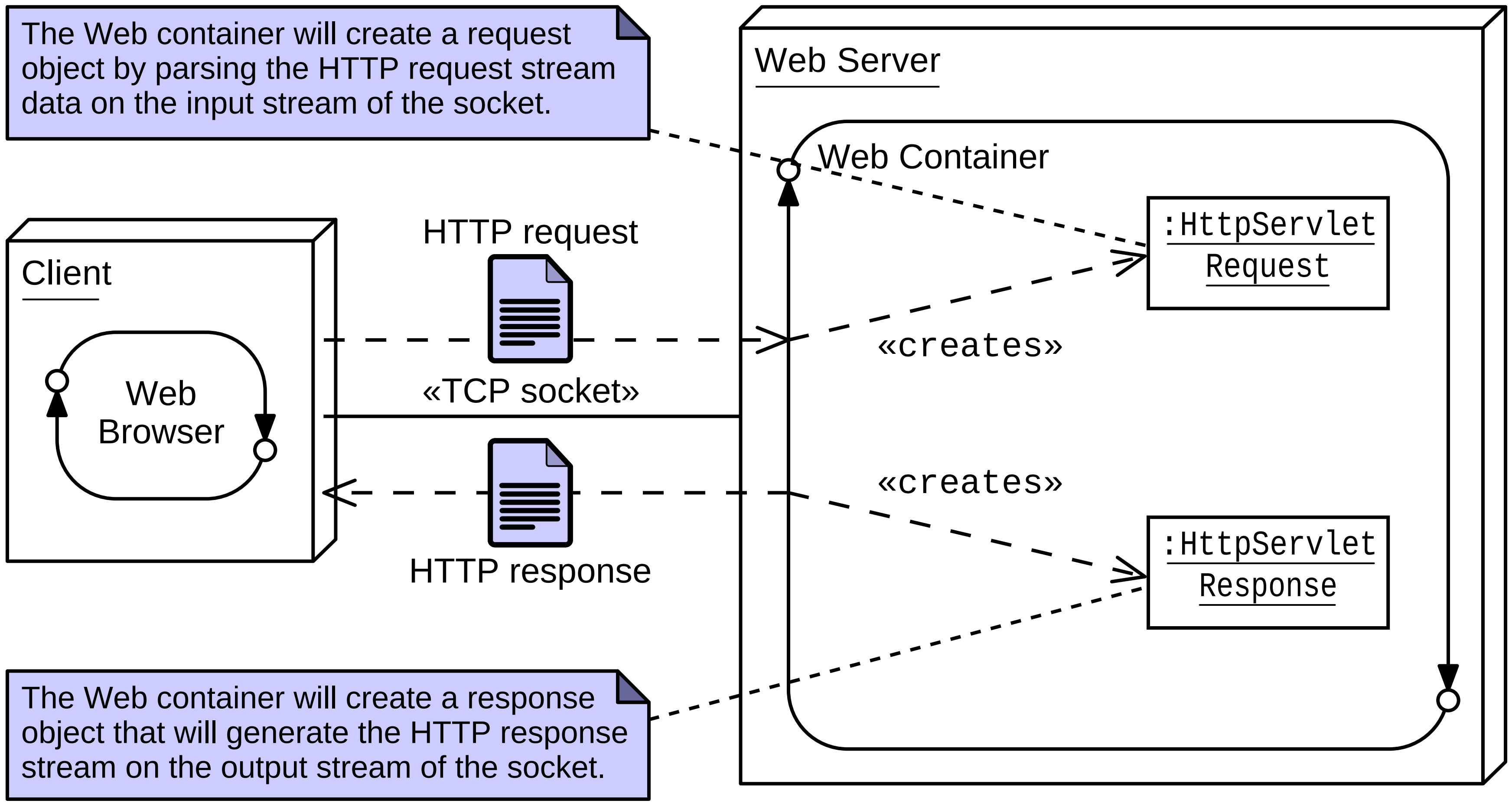


A web container can be used to process HTTP requests by executing the `service` method on an `HttpServlet` object.

Request and Response Process

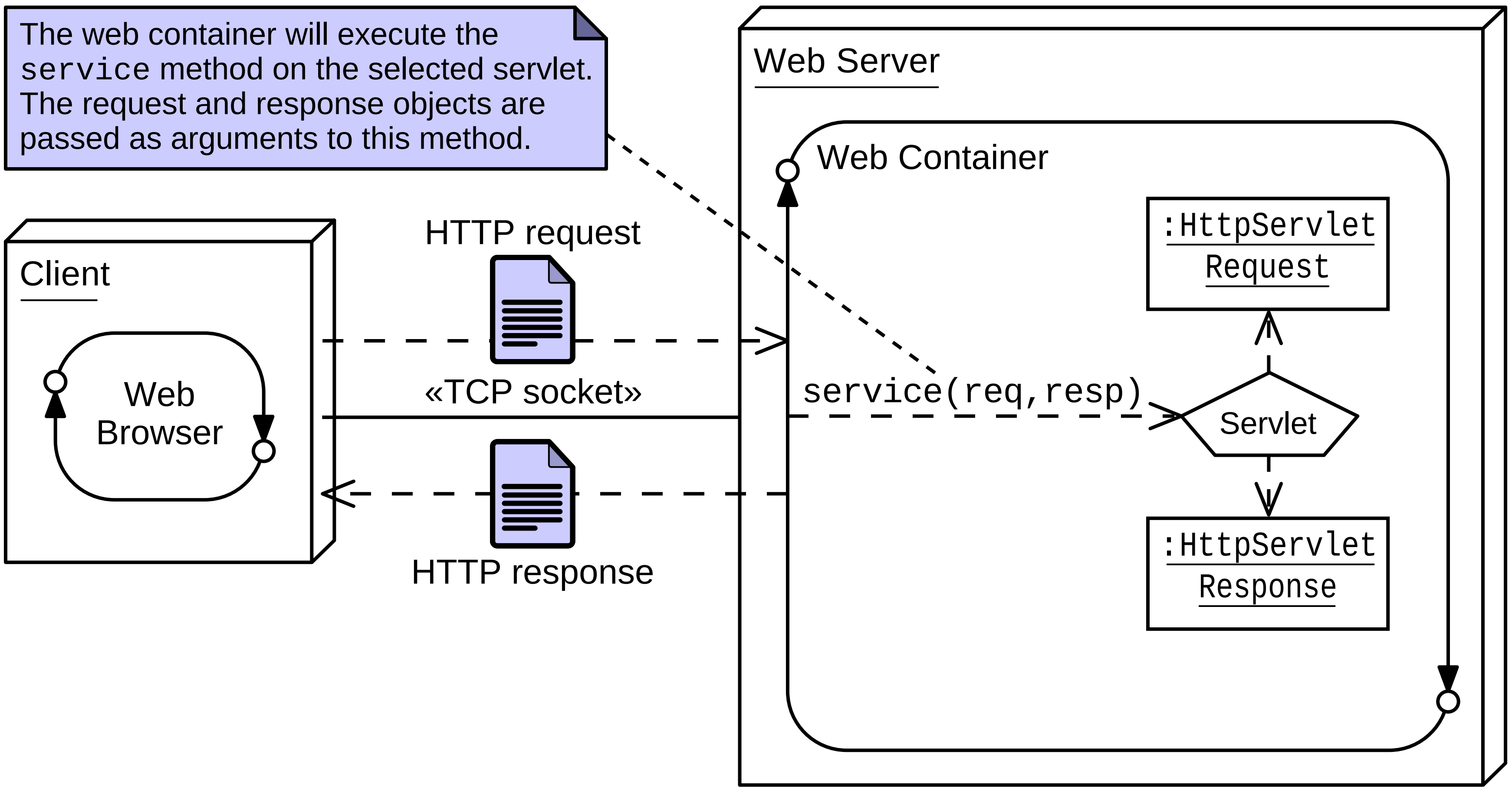

The web browser initiates an HTTP request by opening a TCP socket with the web server. The input stream of the socket contains the HTTP request data. The output stream of the socket contains the HTTP response data.

Request and Response Process (Part 2)


The Web container will create a request object by parsing the HTTP request stream data on the input stream of the socket.
Web Server
Web Container
:HttpServlet Request
Client
HTTP request
Web Browser
«creates»
«TCP socket»
«creates»
:HttpServlet Response
HTTP response
The Web container will create a response object that will generate the HTTP response stream on the output stream of the socket.

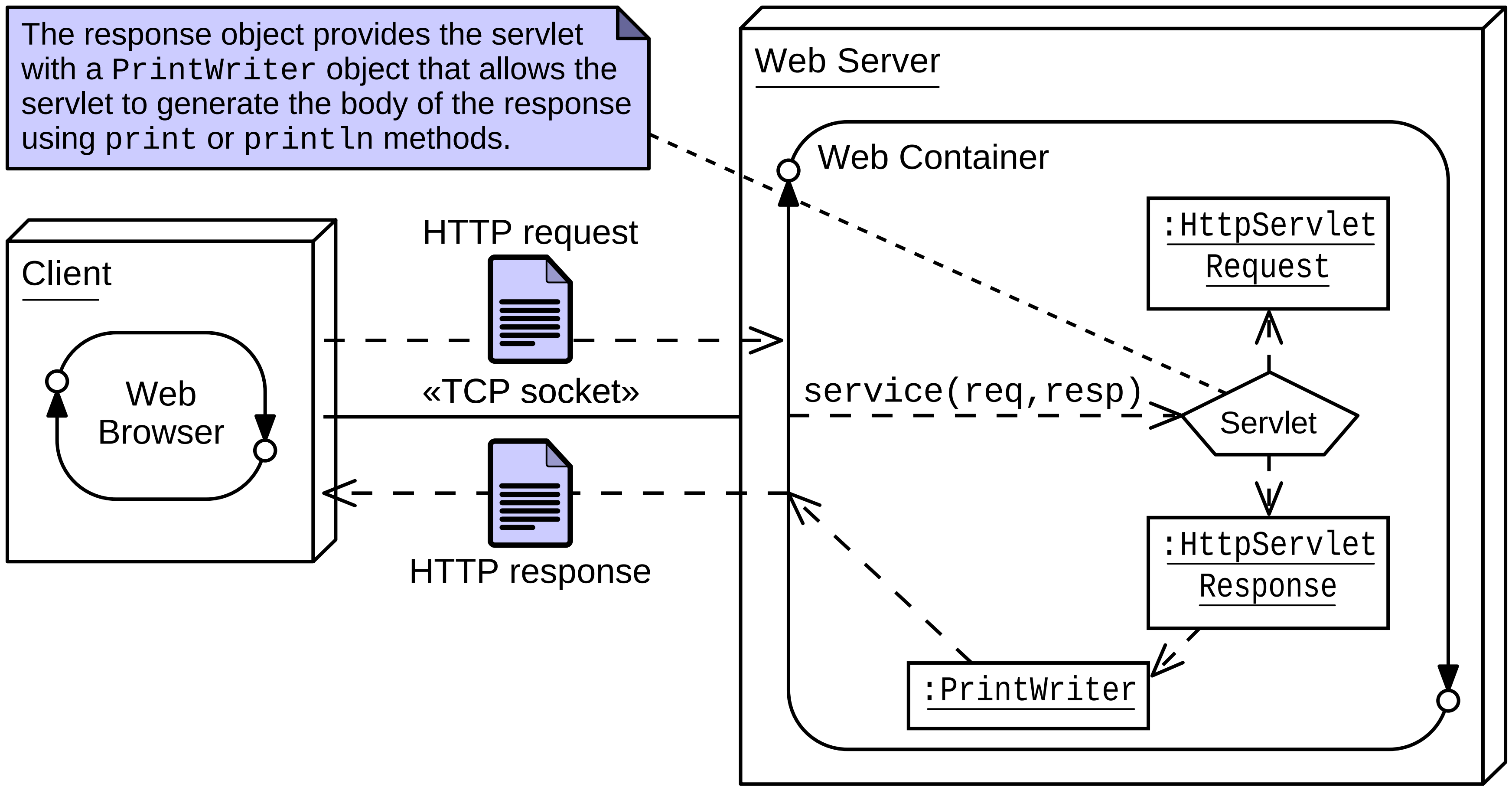
Request and Response Process (Part 3)

The web container will execute the `service` method on the selected servlet. The request and response objects are passed as arguments to this method.

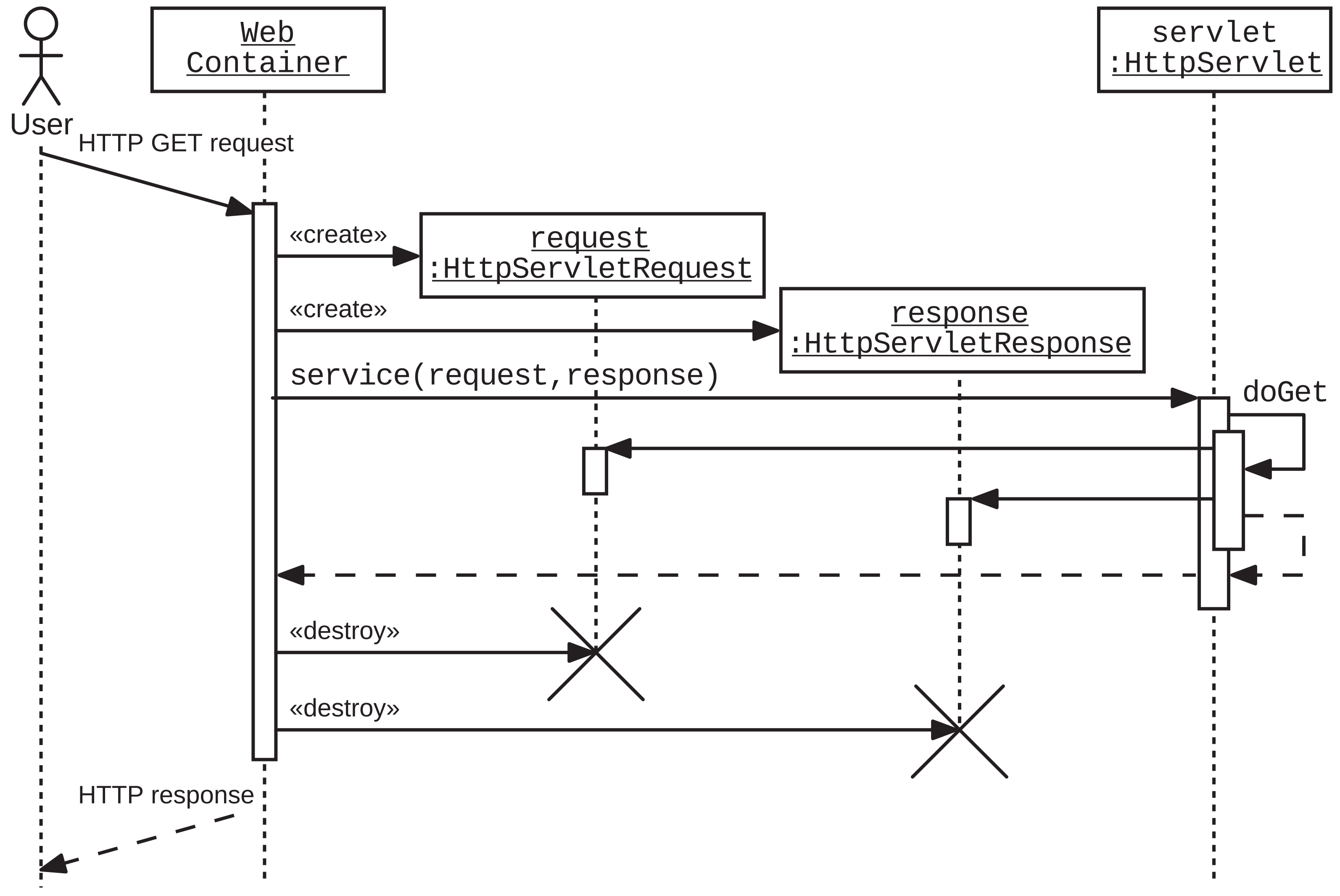


Request and Response Process (Part 4)

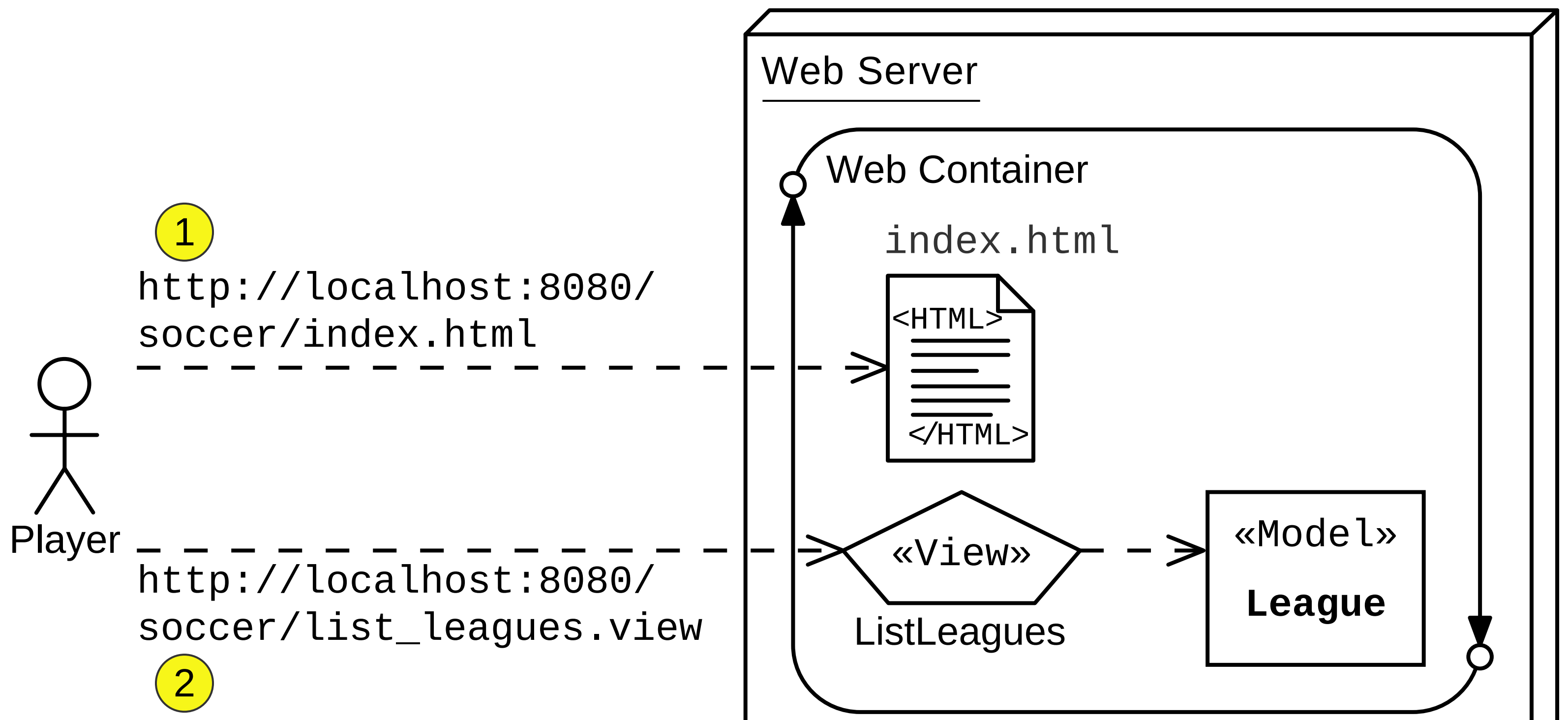
The response object provides the servlet with a `PrintWriter` object that allows the servlet to generate the body of the response using `print` or `println` methods.



Sequence Diagram of an HTTP GET Request



List Leagues Architecture Model



The ListLeaguesServlet Code

```java
package sl314.view;

import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
// Support classes
import java.io.IOException;
import java.io.PrintWriter;
// Model classes
import sl314.model.League;
import java.util.List;
import java.util.LinkedList;
import java.util.Iterator;

public class ListLeaguesServlet extends HttpServlet {

  private List leagueList = null;

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
           throws IOException {
```

The ListLeaguesServlet Code (Part 2)

```
public class ListLeaguesServlet extends HttpServlet {

  private List leagueList = null;

  public void doGet(HttpServletRequest request,
                     HttpServletResponse response)
         throws IOException {

    // Create the set of leagues
    leagueList = new LinkedList();
    leagueList.add( new League(2003, “Spring”,
                               “Soccer League (Spring ‘03)”) );
    leagueList.add( new League(2003, “Summer”,
                               “Summer Soccer Fest 2003”) );
    leagueList.add( new League(2003, “Fall”,
                               “Fall Soccer League (2003)”) );
    leagueList.add( new League(2004, “Spring”,
                               “Soccer League (Spring ‘04)”) );
    leagueList.add( new League(2004, “Summer”,
                               “The Summer of Soccer Love 2004”) );
    leagueList.add( new League(2004, “Fall”,
                               “Fall Soccer League (2004)”) );
```

The `ListLeaguesServlet` Code (Part 3)

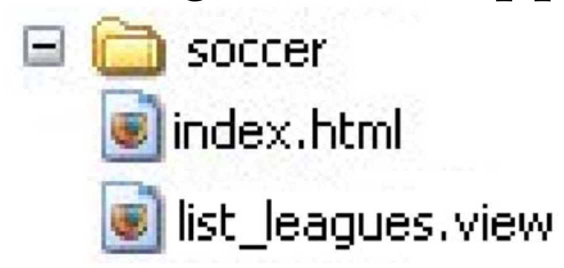
```

// Set page title
String pageTitle = “Duke’s Soccer League: List Leagues”;

// Specify the content type is HTML
response.setContentType(“text/html”);
PrintWriter out = response.getWriter();

// Generate the HTML response
out.println(“<html>”);
out.println(“<head>”);
out.println(“  <title>” + pageTitle + “</title>”);
out.println(“</head>”);
out.println(“<body bgcolor=’white’>”);

// Generate page heading
out.println(“<!-- Page Heading -->”);
out.println(“<table border=’1’ cellpadding=’5’ cellspacing=’0’
width=’400’>”);
out.println(“<tr bgcolor=’#CCCCFF’ align=’center’ valign=’center’
height=’20’>”);
```

The ListLeaguesServlet Code (Part 4)

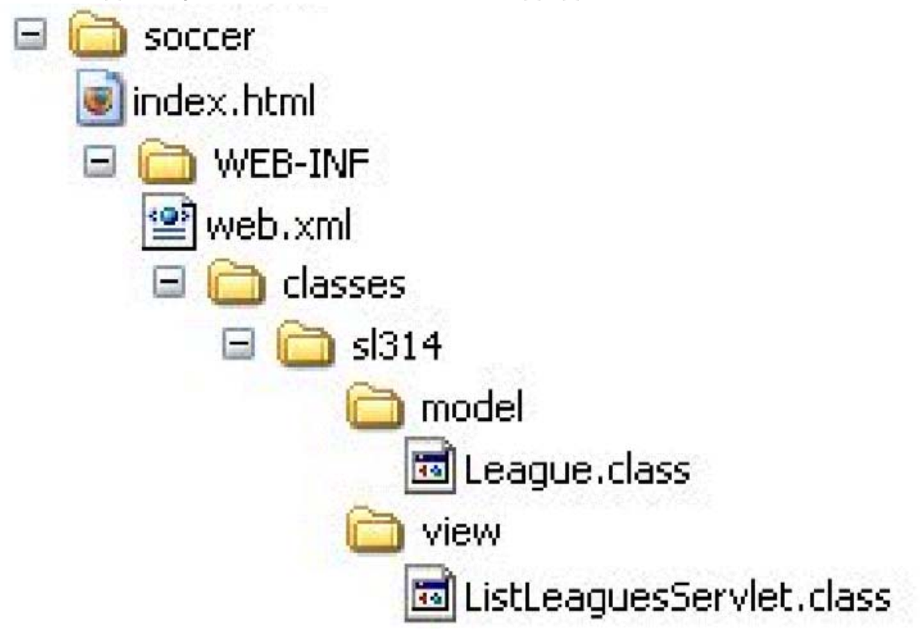
```
        out.println("  <td><h3>" + pageTitle + "</h3></td>");
        out.println("</tr>");
        out.println("</table>");

        // Generate main body
        out.println("<p>");
        out.println("The set of soccer leagues are:");
        out.println("</p>");

        out.println("<ul>");
        Iterator items = leagueList.iterator();
        while ( items.hasNext() ) {
          League league = (League) items.next();
          out.println("  <li>" + league.getTitle() + "</li>");
        }
        out.println("</ul>");

        out.println("</body>");
        out.println("</html>");
      } // END of doGet method
```

Soccer League Web Application Structure

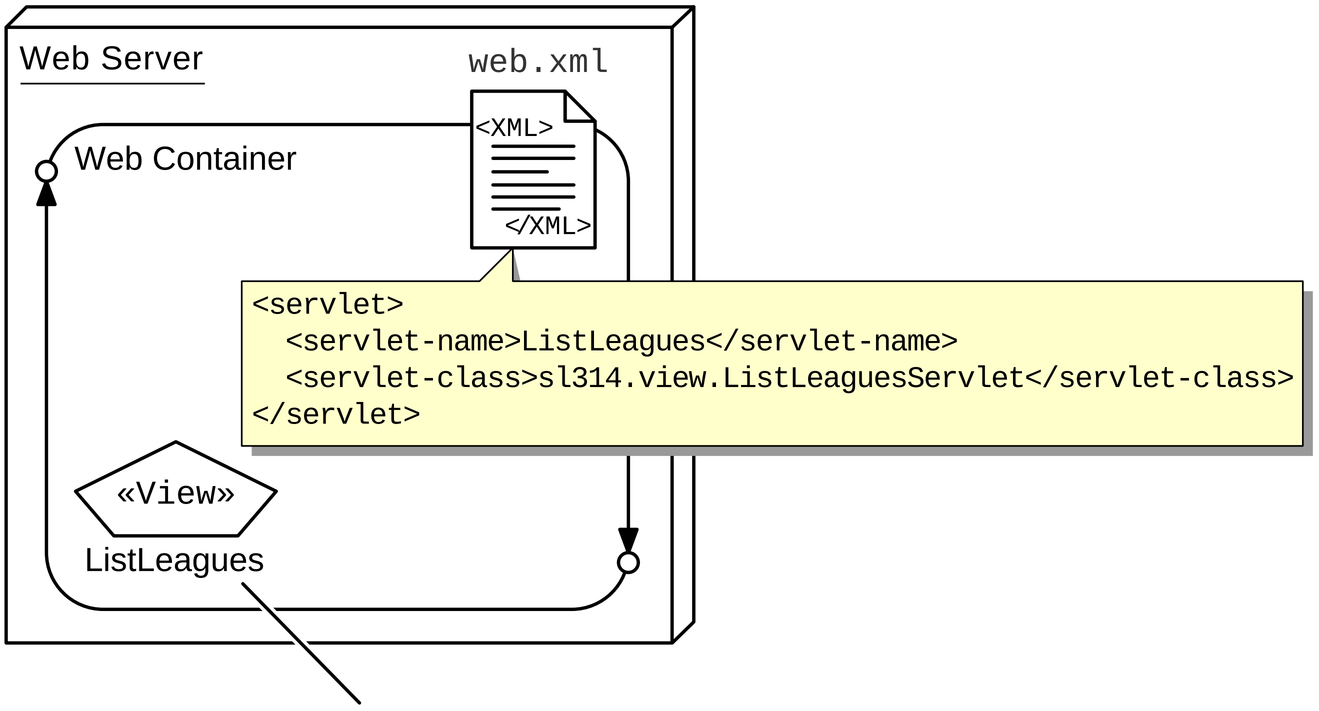
The logical web application hierarchy:



The physical web application hierarchy:



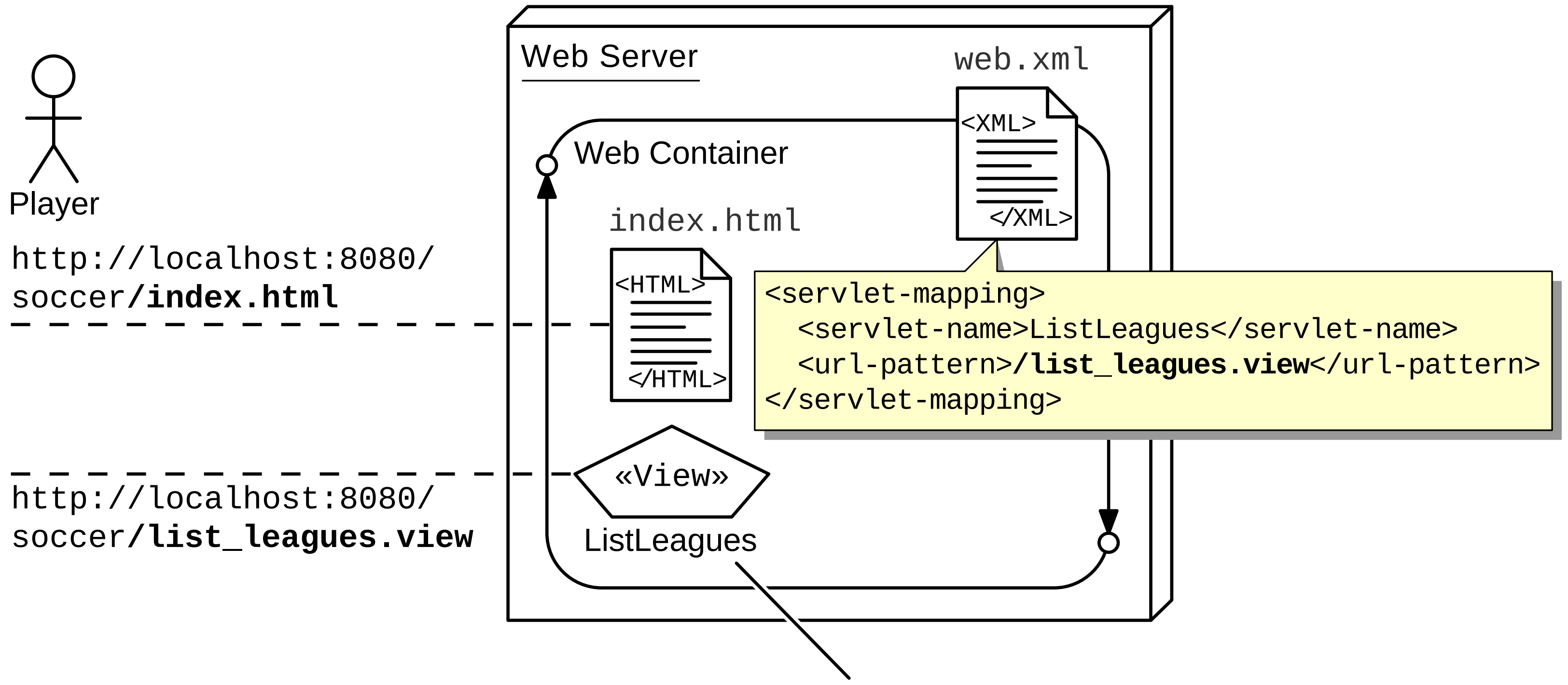
Configuring a Servlet Definition



```
<servlet>
  <servlet-name>ListLeagues</servlet-name>
  <servlet-class>sl314.view.ListLeaguesServlet</servlet-class>
</servlet>
```

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

Configuring a Servlet Mapping



http://localhost:8080/
soccer/**index.html**

```
<servlet-mapping>
  <servlet-name>ListLeagues</servlet-name>
  <url-pattern>/list_leagues.view</url-pattern>
</servlet-mapping>
```

http://localhost:8080/
soccer/**list_leagues.view**

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).

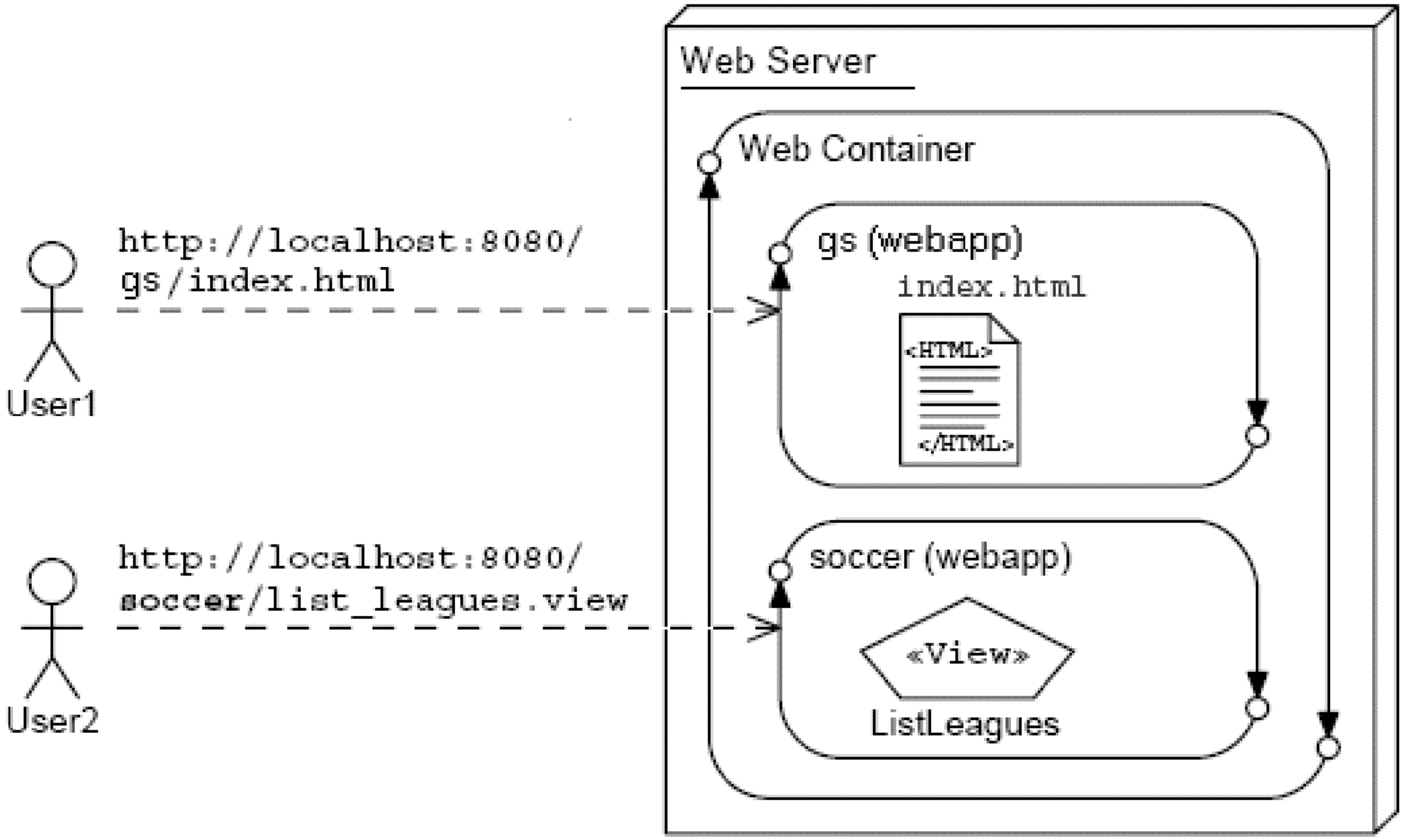
Complete Deployment Descriptor

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>

<web-app
    xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
                        http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
    version="2.4">

  <display-name>SL-314 WebApp Example</display-name>
  <description>
    This Web Application demonstrates a single View servlet.
  </description>

  <servlet>
    <servlet-name>ListLeagues</servlet-name>
    <servlet-class>sl314.view.ListLeaguesServlet</servlet-class>
  </servlet>

```

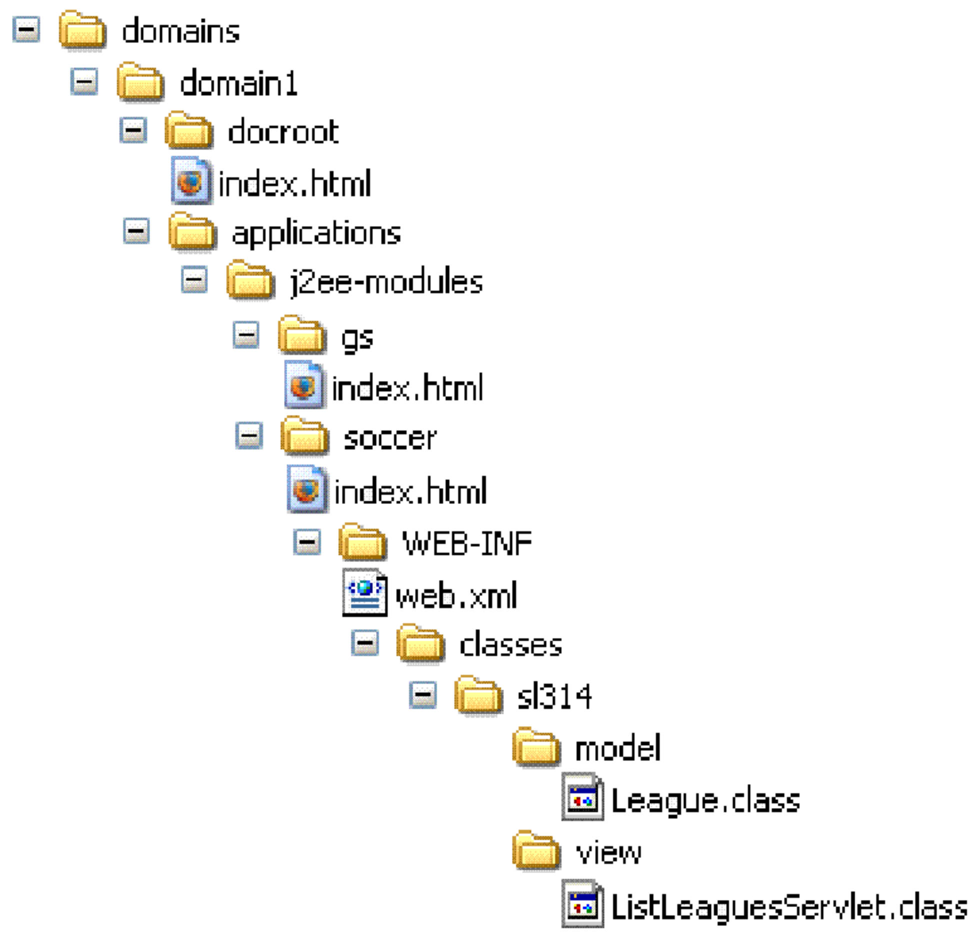
Complete Deployment Descriptor

```
<?xml version=”1.0” encoding=”UTF-8”?>
<web-app version=”2.5” xmlns=”http://java.sun.com/xml/ns/javaee” xmlns:xsi=”http://www.w3.org/2001/XMLSchema-instance” xsi:schemaLocation=”http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd”>

  <servlet>
    <servlet-name>ListLeagues</servlet-name>
    <servlet-class>sl314.view.ListLeaguesServlet</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>ListLeagues</servlet-name>
    <url-pattern>/list_leagues.view</url-pattern>
  </servlet-mapping>

```

Web Application Context Root

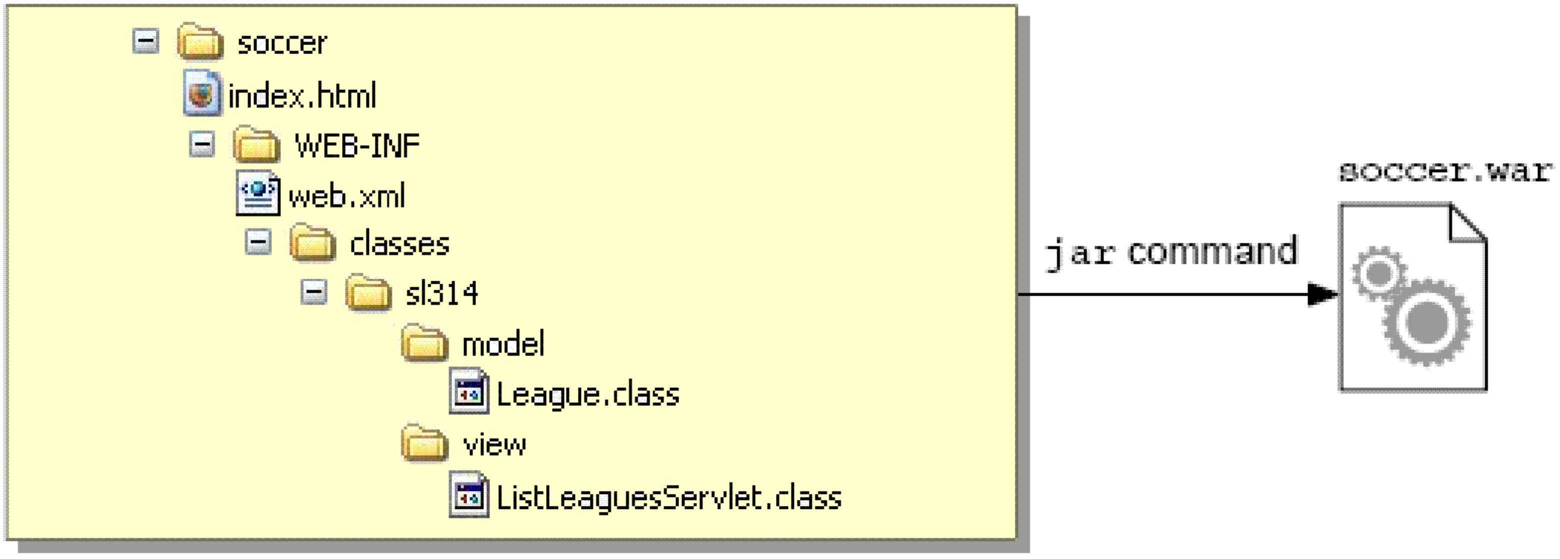


Sun Java™ System Application Server Deployment

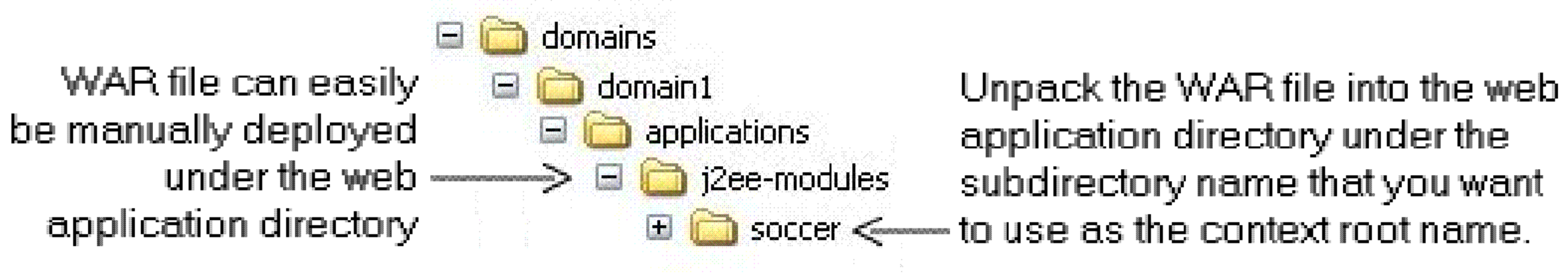


```
domains
  domain1
    docroot
      index.html
    applications
      j2ee-modules
        gs
          index.html
        soccer
          index.html
          WEB-INF
            web.xml
            classes
              sl314
                model
                  League.class
                view
                  ListLeaguesServlet.class
```

WAR Files for Deployment



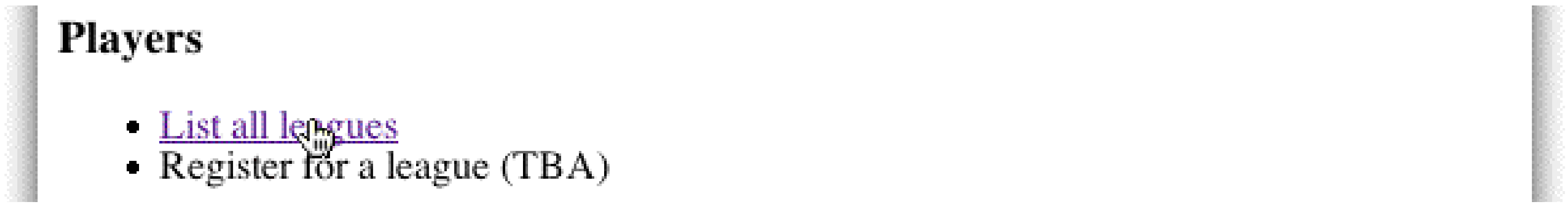
Application Server deployment of a WAR file:



domains
domain1
applications
j2ee-modules
soccer
WAR file can easily be manually deployed under the web application directory
Unpack the WAR file into the web application directory under the subdirectory name that you want to use as the context root name.

Activating the Servlet in a Web Browser

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



Players

- List all leagues
- Register for a league (TBA)

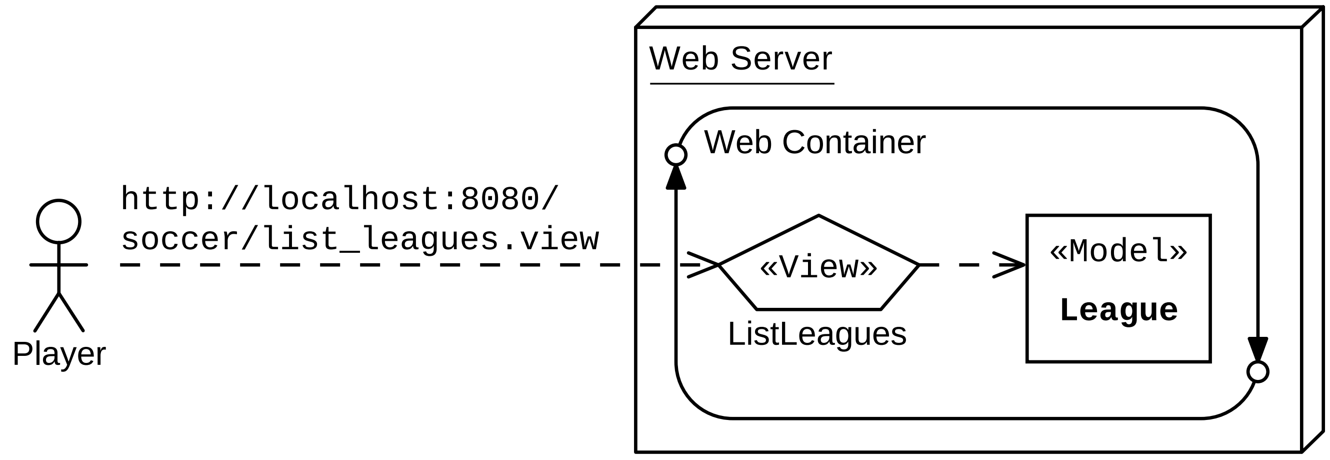
HTML:

```
<h3>Players</h3>

<ul>
  <li><a href='list_leagues.view'>List all leagues</a></li>
  <li>Register for a league (TBA)</li>
</ul>
```

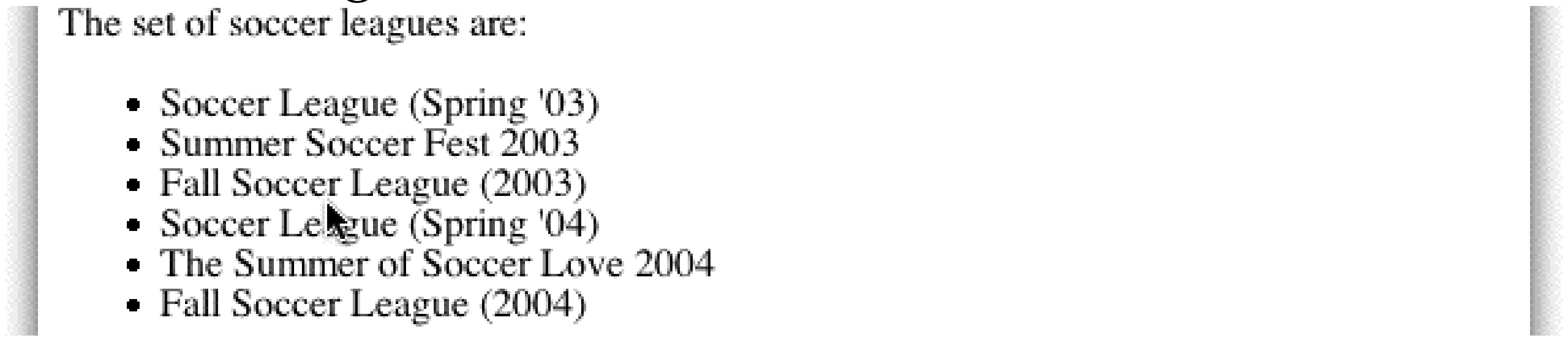
Clicking on List performs a GET request for the URL:
`http://localhost:8080/soccer/list_leagues.view`

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

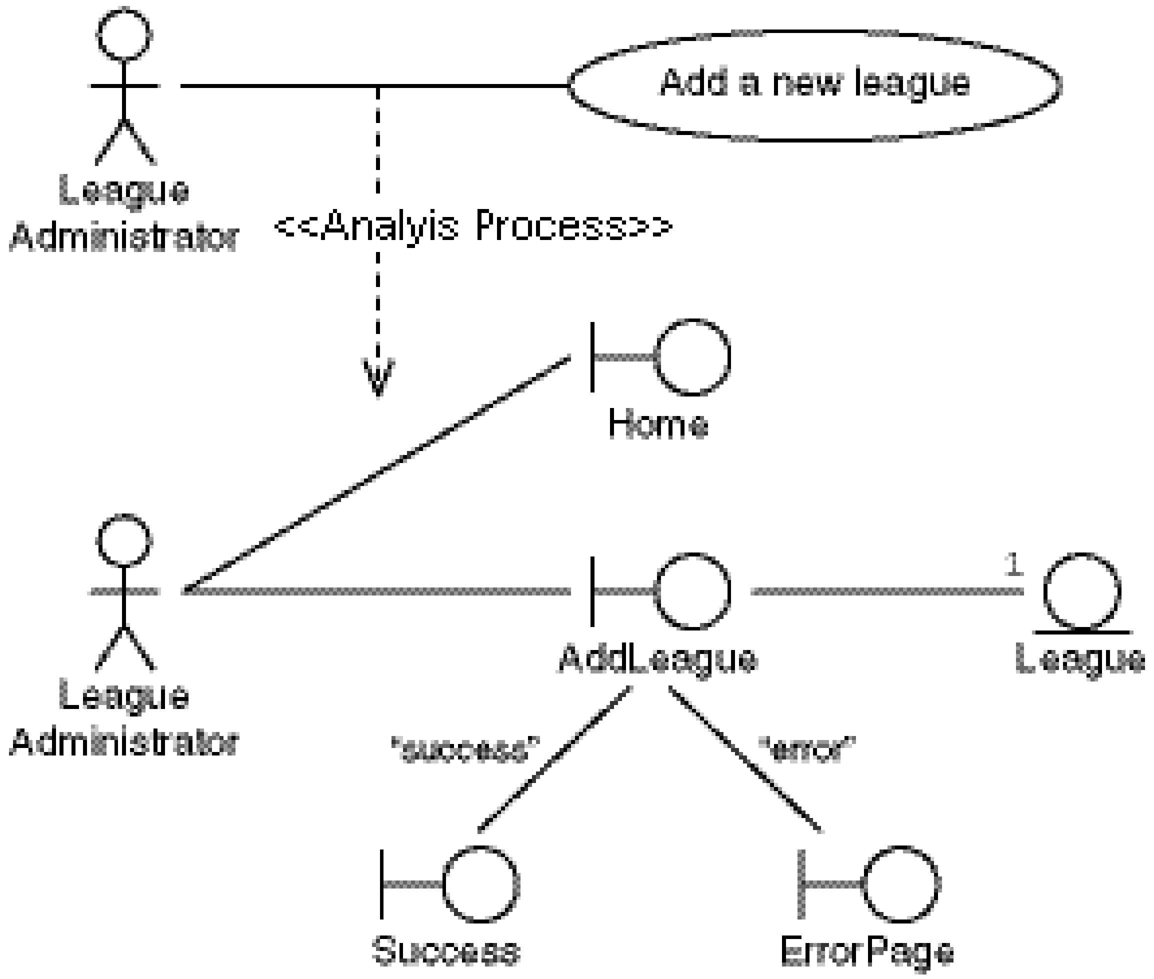
Relevance

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

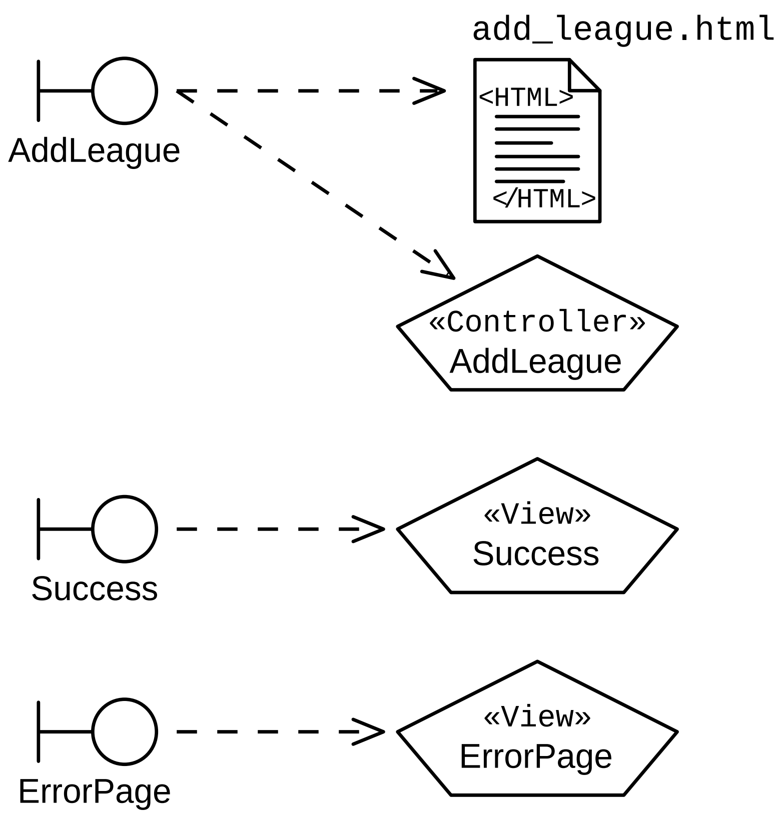
Types of Controller Components

- Process input from a user
- Support screen navigation
- Prepare data for view components

Add a New League Analysis Model

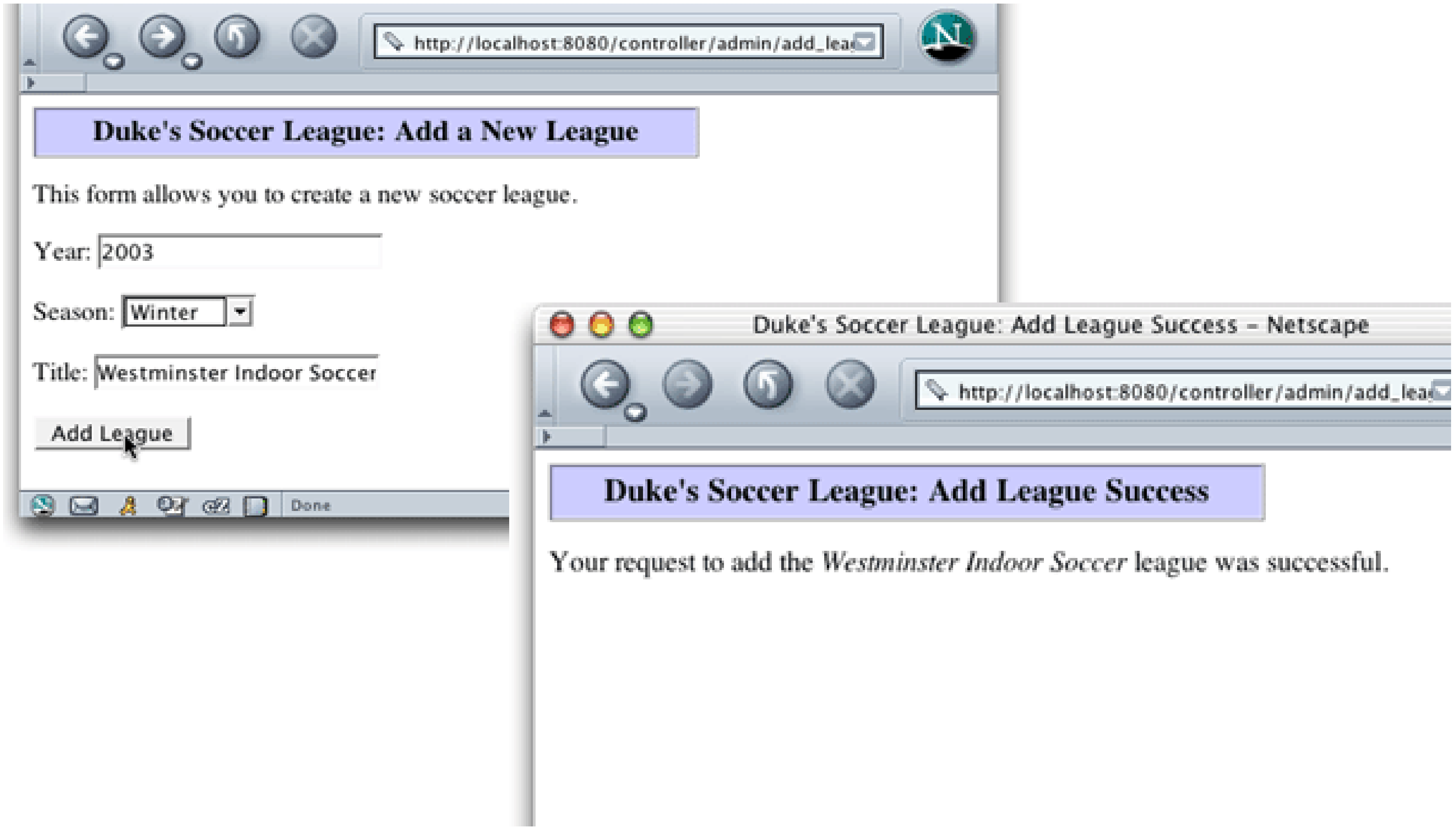


Add League Boundary Components



Add a New League Page Flow

Success path:



http://localhost:8080/controller/admin/add_lea

Duke's Soccer League: Add a New League

This form allows you to create a new soccer league.

Year: 2003

Season: Winter

Title: Westminster Indoor Soccer

Duke's Soccer League: Add League Success - Netscape

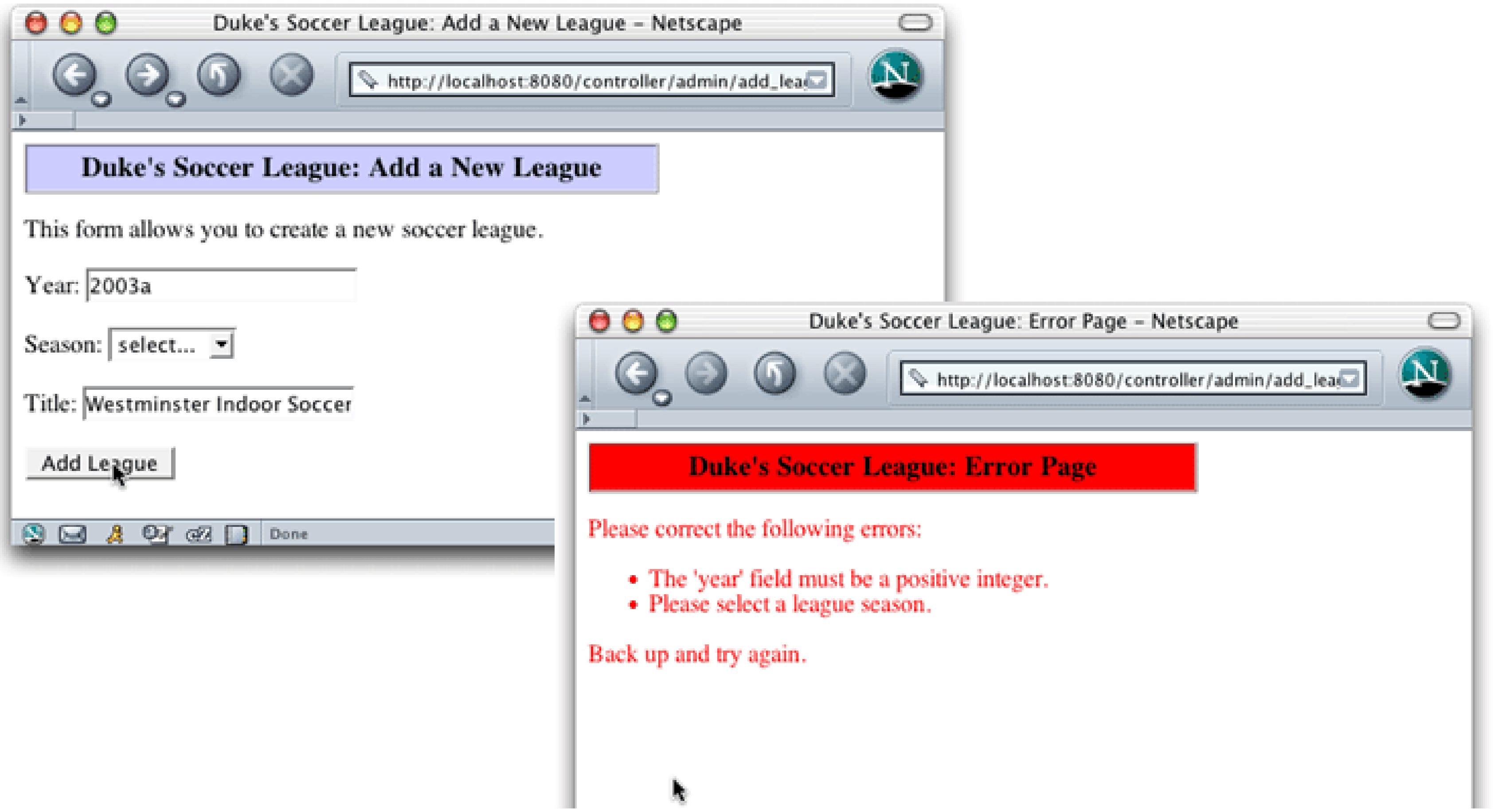
http://localhost:8080/controller/admin/add_lea

Duke's Soccer League: Add League Success

Your request to add the *Westminster Indoor Soccer* league was successful.

Add a New League Page Flow (continued)

Error path:


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

http://localhost:8080/controller/admin/add_lea

Duke's Soccer League: Add a New League

This form allows you to create a new soccer league.

Year: 2003a

Season: select...

Title: Westminster Indoor Soccer

Add League

Done

Duke's Soccer League: Error Page - Netscape

http://localhost:8080/controller/admin/add_lea

Duke's Soccer League: Error Page

Please correct the following errors:

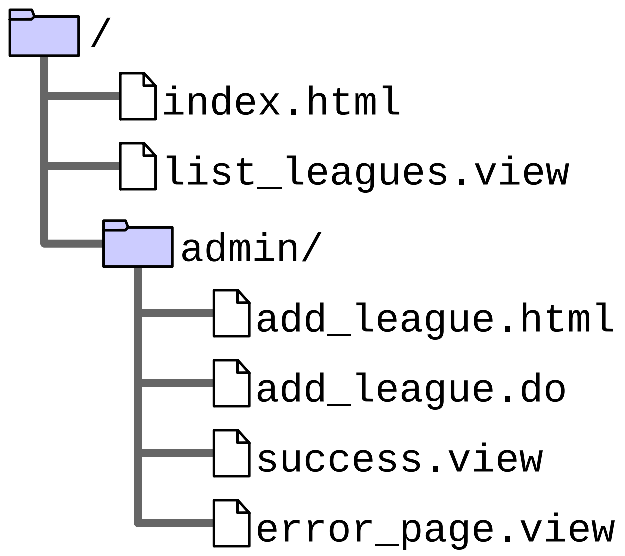
- The 'year' field must be a positive integer.
- Please select a league season.

Back up and try again.

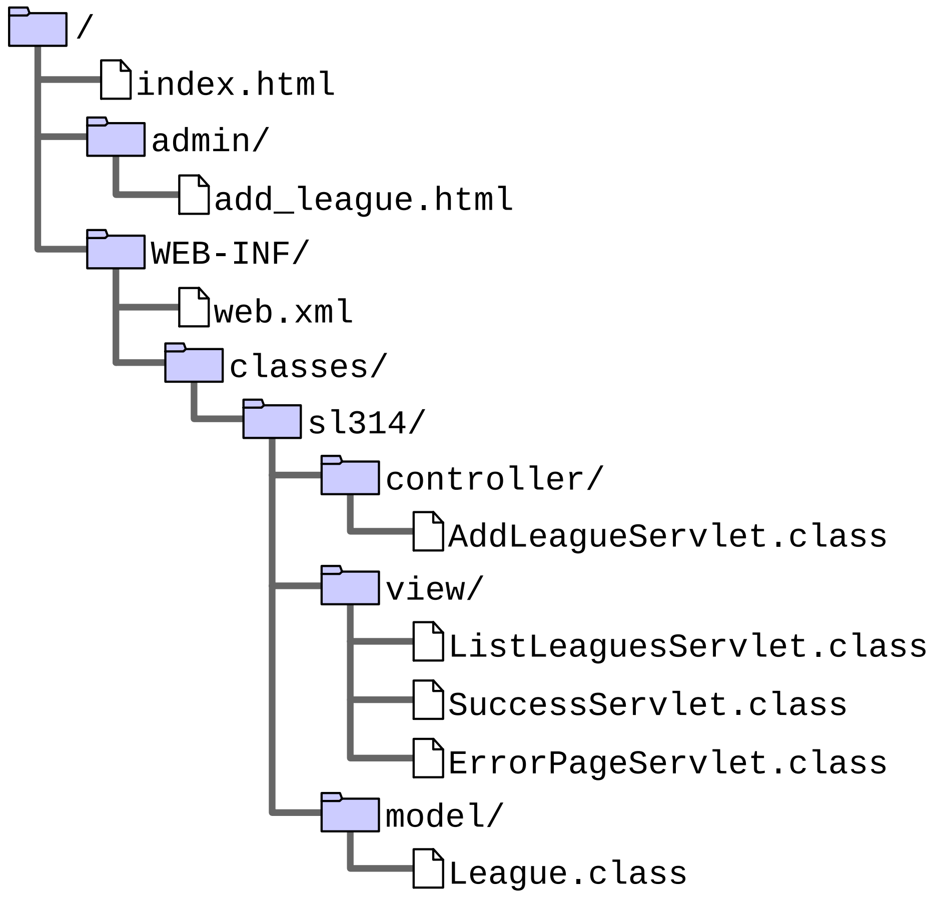
Form Verification

- 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?

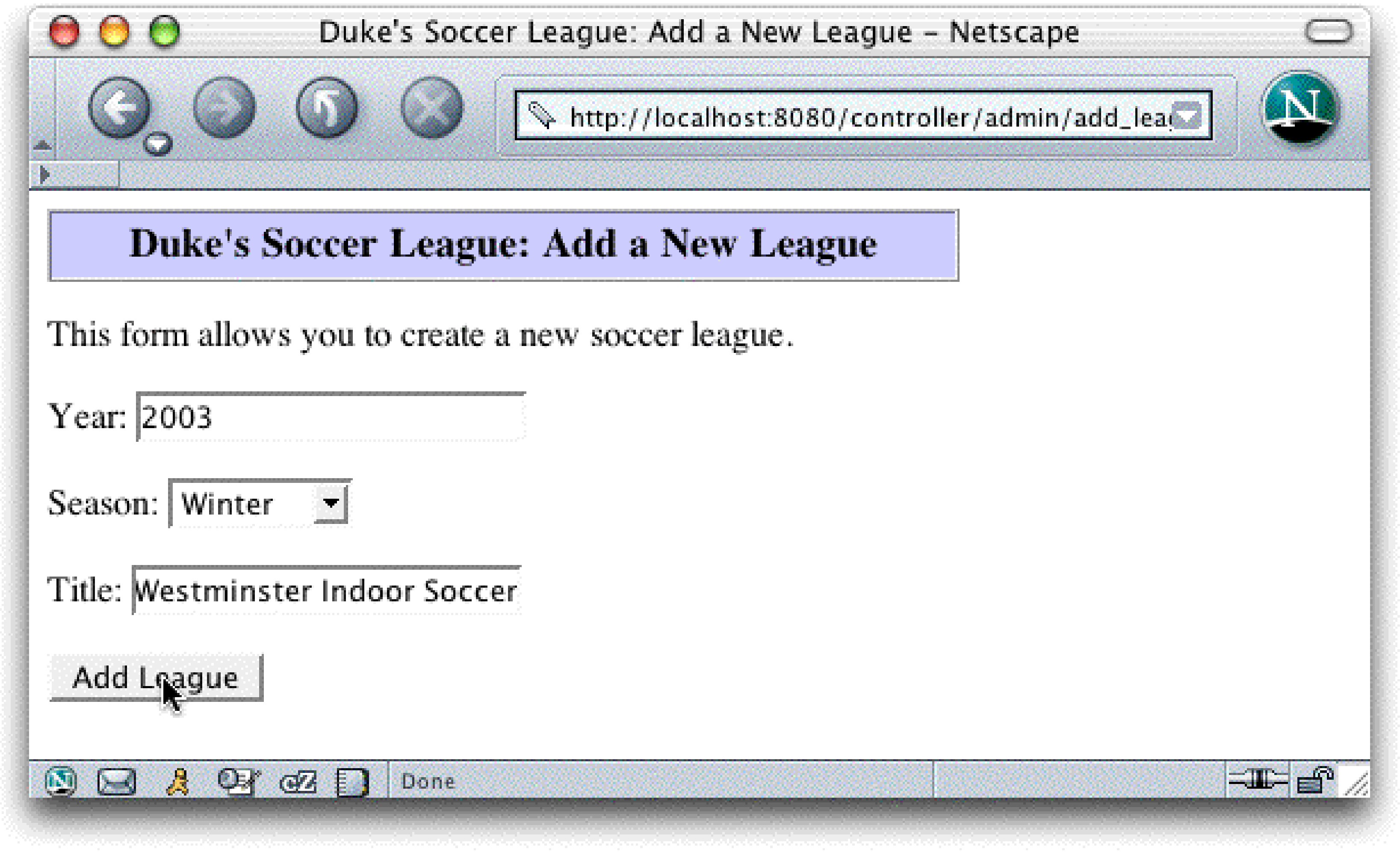
Soccer League Web Structure



Soccer League Web Structure (continued)



Creating an HTML Form



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

http://localhost:8080/controller/admin/add_lea

Duke's Soccer League: Add a New League

This form allows you to create a new soccer league.

Year: 2003

Season: Winter

Title: Westminster Indoor Soccer

Add League

Done

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>
```

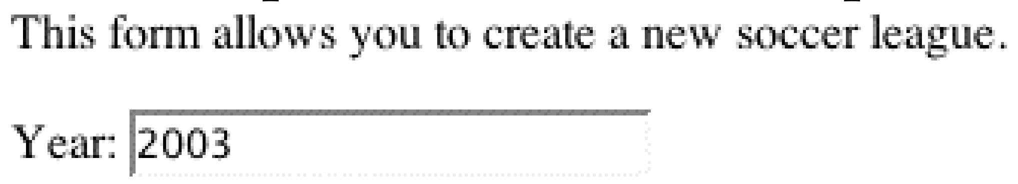
For example:

```
<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.

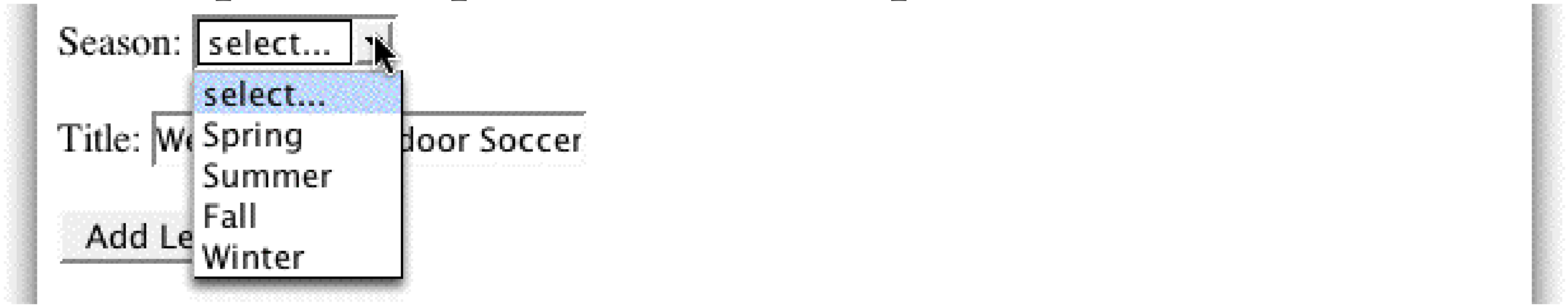
Year: 2003

The HTML content for this component is:

```
<p>
This form allows you to create a new soccer league.
</p>

<form action='add_league.do' method='POST'>
Year: <input type='text' name='year' /> <br/><br/>
```

Drop-Down List Component

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

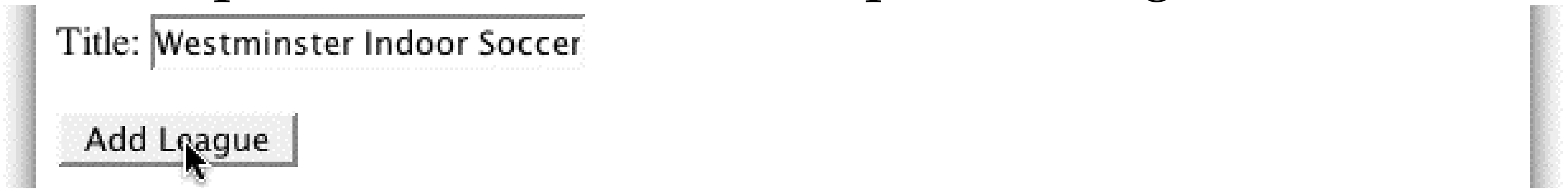


The HTML content for this component is:

```
Season: <select name='season'>
          <option value='UNKNOWN'>select...</option>
          <option value='Spring'>Spring</option>
          <option value='Summer'>Summer</option>
          <option value='Fall'>Fall</option>
          <option value='Winter'>Winter</option>
        </select> <br/><br/>
```

Submit Button

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


Title: Westminster Indoor Soccer

The HTML content for this component is:

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

Complete Add a New League Form

```
<p>
This form allows you to create a new soccer league.
</p>

<form action='add_league.do' method='POST'>
Year: <input type='text' name='year' /> <br/><br/>
Season: <select name='season'>
          <option value='UNKNOWN'>select...</option>
          <option value='Spring'>Spring</option>
          <option value='Summer'>Summer</option>
          <option value='Fall'>Fall</option>
          <option value='Winter'>Winter</option>
        </select> <br/><br/>
Title: <input type='text' name='title' /> <br/><br/>
<input type='submit' value='Add League' />
</form>
```

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,tex
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,tex
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:

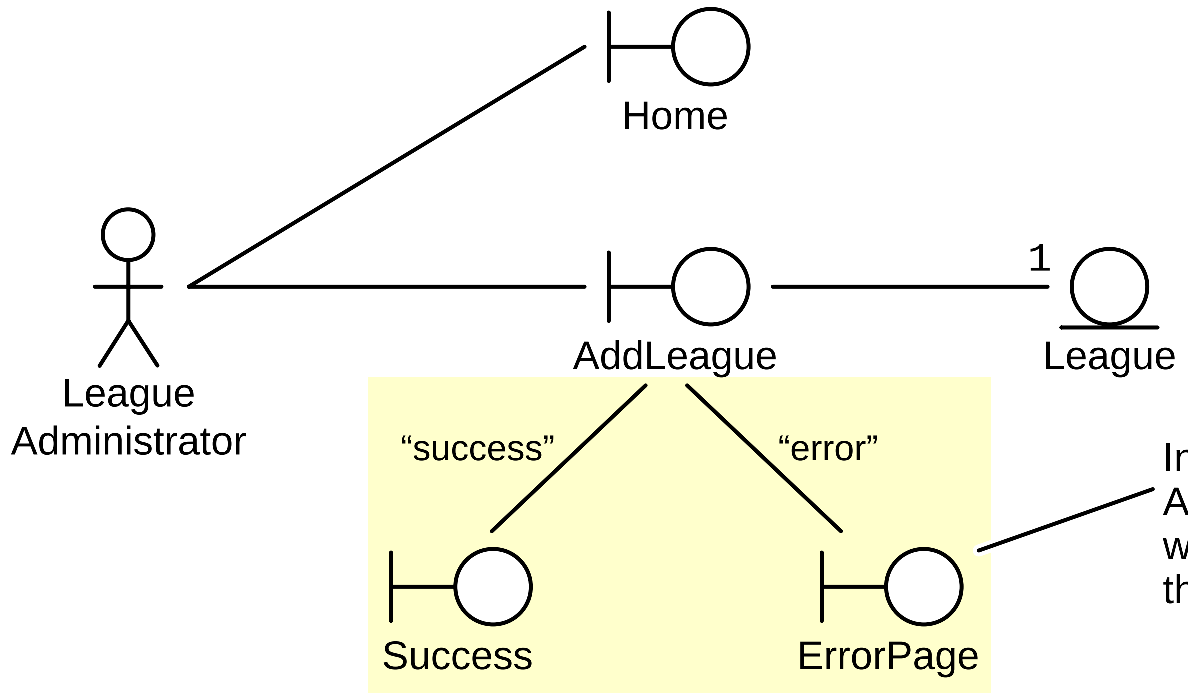
- The processing of the request changes the state of the server, such as storing data in a database.
- The amount of form data is large.
- The contents of the data should not be visible in the URL (for example, passwords).

Developing a Controller Servlet

A form-processing (controller) servlet needs to:

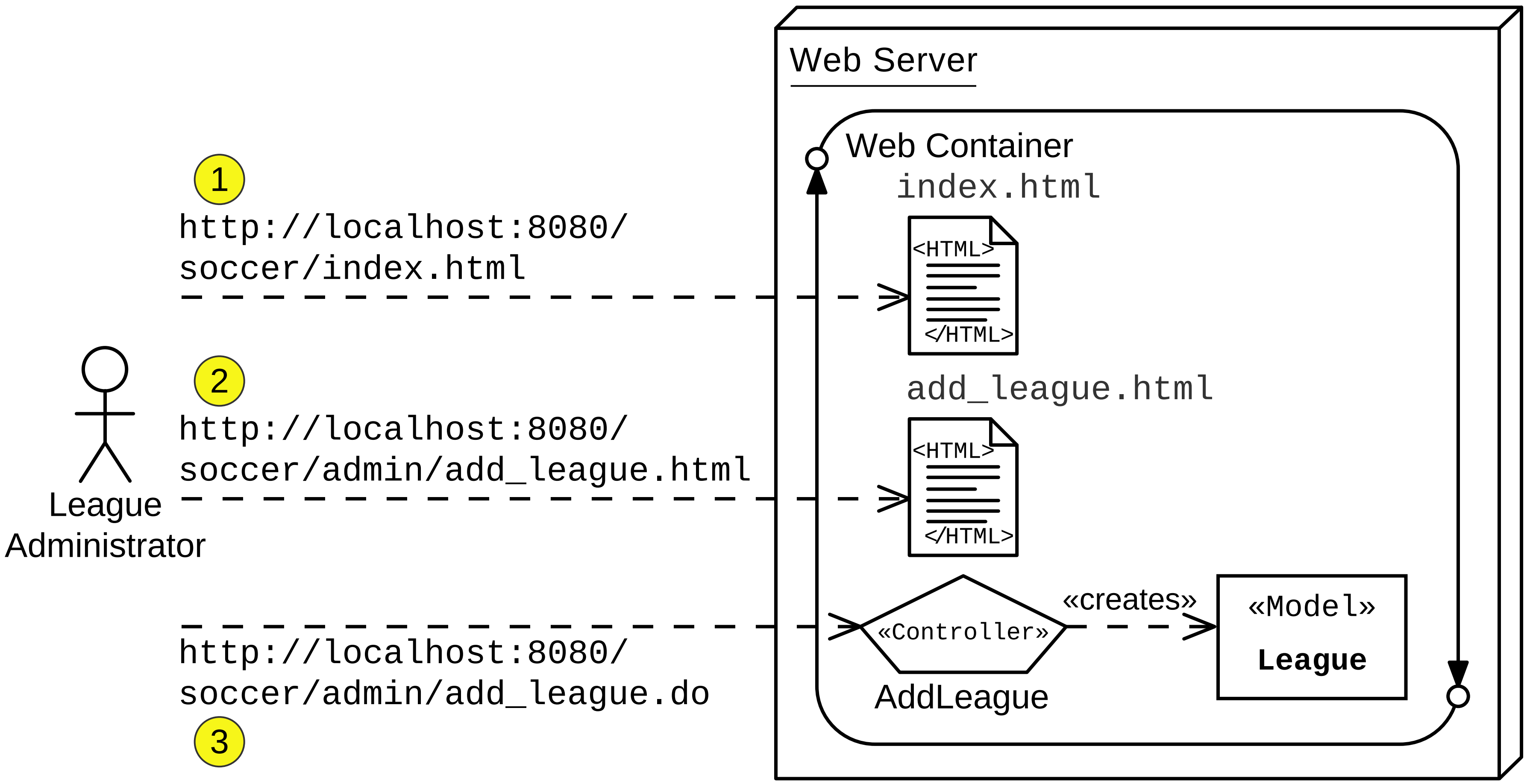
1. Retrieve form parameters from the HTTP request.
2. Perform any data conversion on the form parameters.
3. Verify the form parameters.
4. Execute the business logic.
5. Dispatch to the next view component based on the results of the previous steps.

Add League Analysis Model (Stage 1)

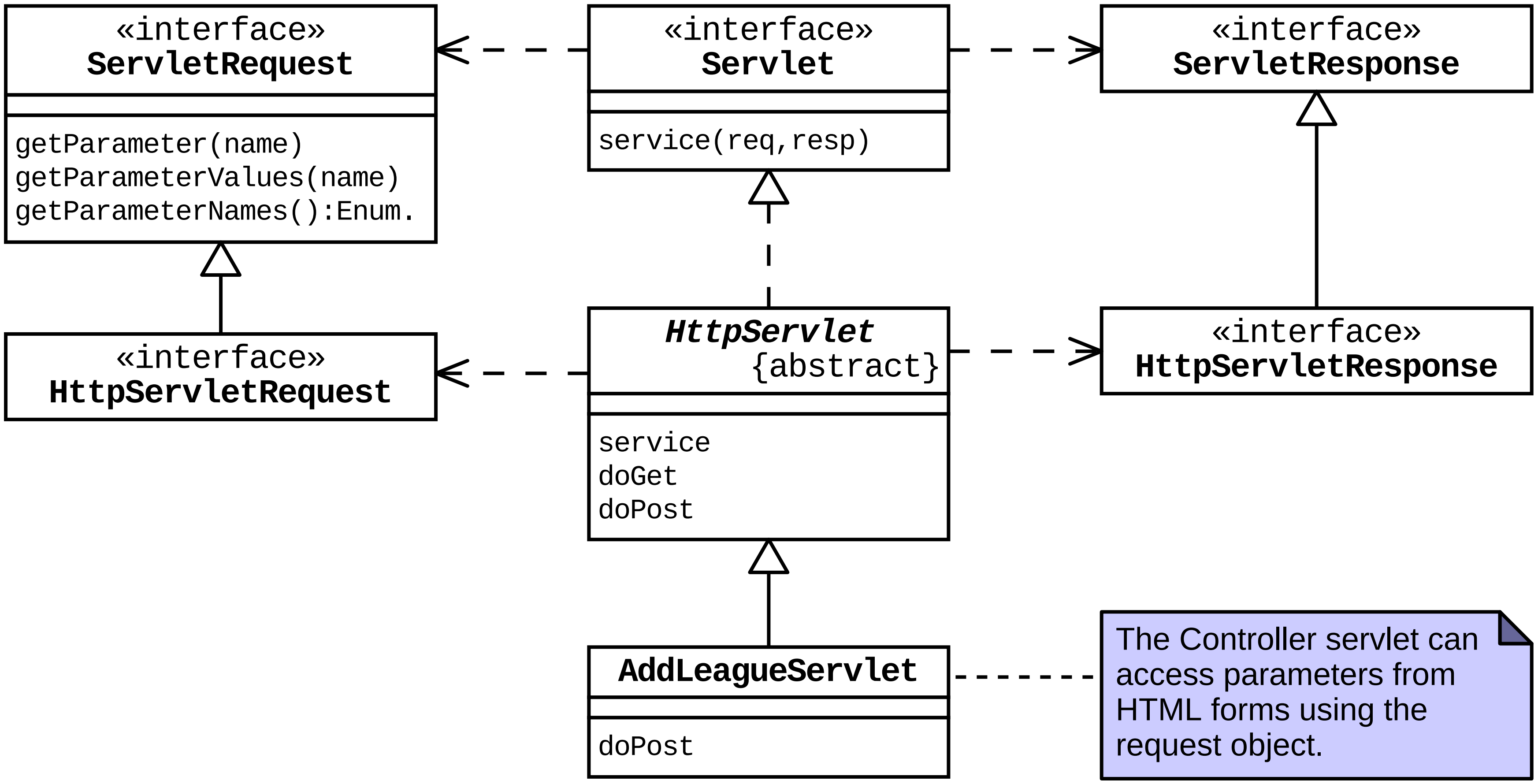


In Step 1, the AddLeague servlet will not dispatch to the view components.

Add League Architecture Model (Stage 1)(continued)



Servlet API to Retrieve Form Parameters



The Controller servlet can access parameters from HTML forms using the request object.

The AddLeagueServlet Class Declaration

```
package sl314.controller;

import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.ServletException;
// Support classes
import java.io.IOException;
import java.io.PrintWriter;
// Model classes
import sl314.model.League;
import java.util.List;
import java.util.LinkedList;

public class AddLeagueServlet extends HttpServlet {
  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
         throws IOException, ServletException {

    // Keep a set of strings to record form processing errors.
    List errorMsgs = new LinkedList();
```

Retrieving Form Parameters and Data Conversion

```

    try {

      // Retrieve form parameters.
      String yearStr = request.getParameter(“year”).trim();
      String season = request.getParameter(“season”).trim();
      String title = request.getParameter(“title”).trim();

      // Perform data conversions.
      int year = -1;
      try {
        year = Integer.parseInt(yearStr);
      } catch (NumberFormatException nfe) {
       errorMsgs.add(“The ‘year’ field must be a positive integer.”);
      }

```

Performing Form Validations

```

    // Verify form parameters
    if ( (year != -1) && ((year < 2000) || (year > 2010)) ) {
      errorMsgs.add(“The ‘year’ field must within 2000 to 2010.”);
    }
    if ( season.equals(“UNKNOWN”) ) {
      errorMsgs.add(“Please select a league season.”);
    }
    if ( title.length() == 0 ) {
      errorMsgs.add(“Please enter the title of the league.”);
    }

    // Send the ErrorPage view if there were errors
    if ( ! errorMsgs.isEmpty() ) {
      // dispatch to the ErrorPage
      PrintWriter out = response.getWriter();
      out.println(“ERROR PAGE”);
      return;
    }

```

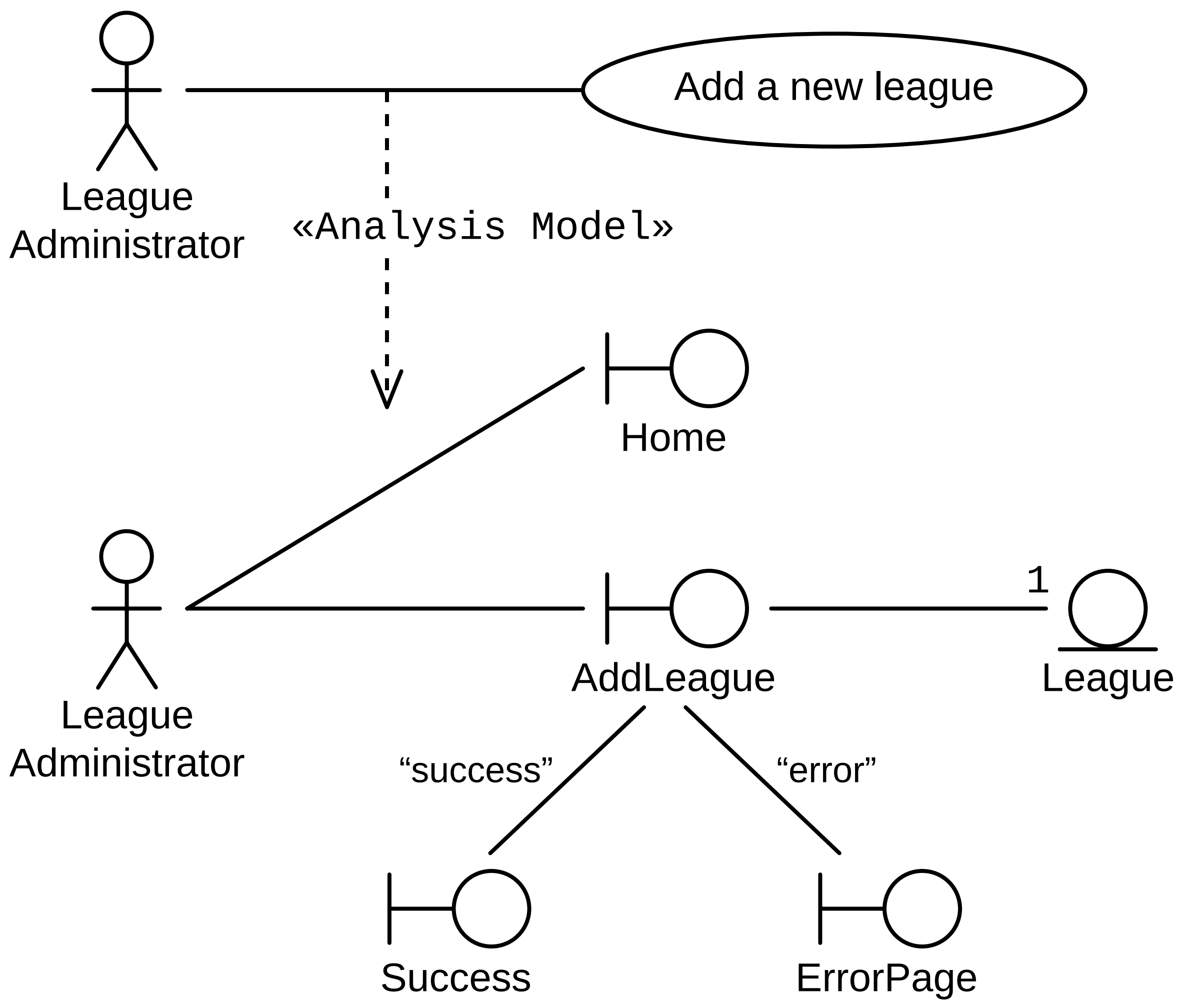
Performing the Business Logic

```

        // Perform business logic
        League league = new League(year, season, title);

        // Send the Success view
        PrintWriter out = response.getWriter();
        out.println(“SUCCESS”);
        return;

```

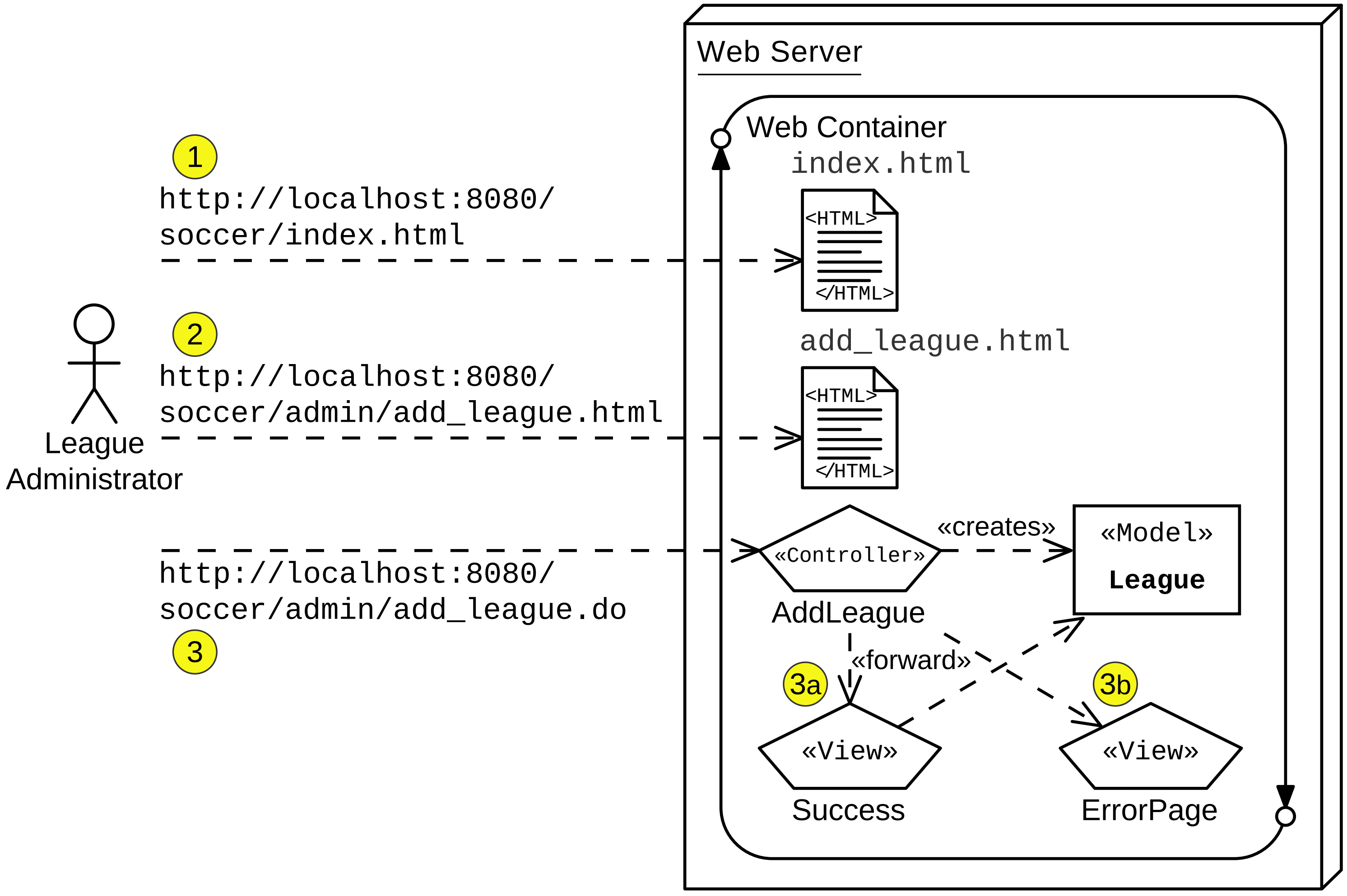
Handling an Exception

```

    // Handle any unexpected exceptions
    } catch (RuntimeException e) {
      errorMsgs.add(e.getMessage());
      // dispatch to the ErrorPage
      PrintWriter out = response.getWriter();
      out.println(“ERROR PAGE”);

      // Log stack trace
      e.printStackTrace(System.err);

    } // END of try-catch block
  } // END of doPost method
} // END of AddLeagueServlet class
```

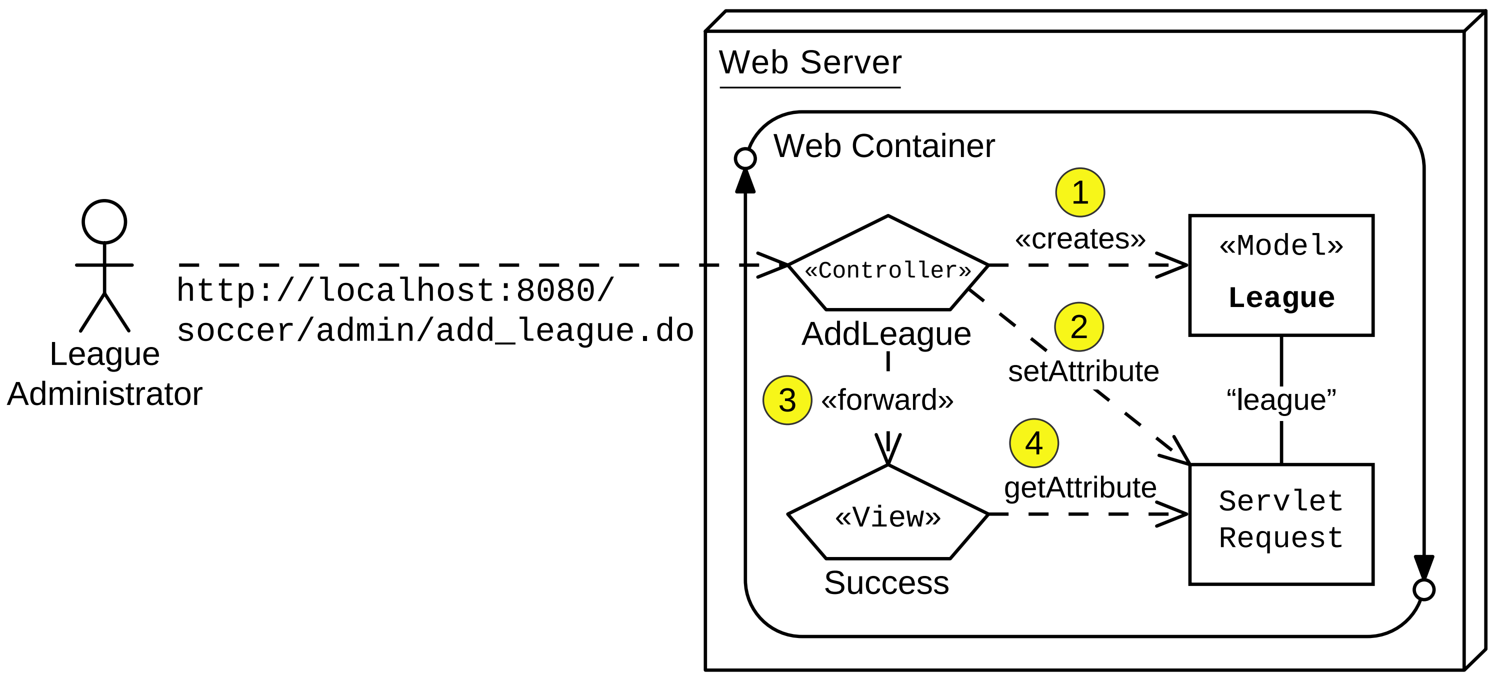
Add League Analysis Model (Stage 2)



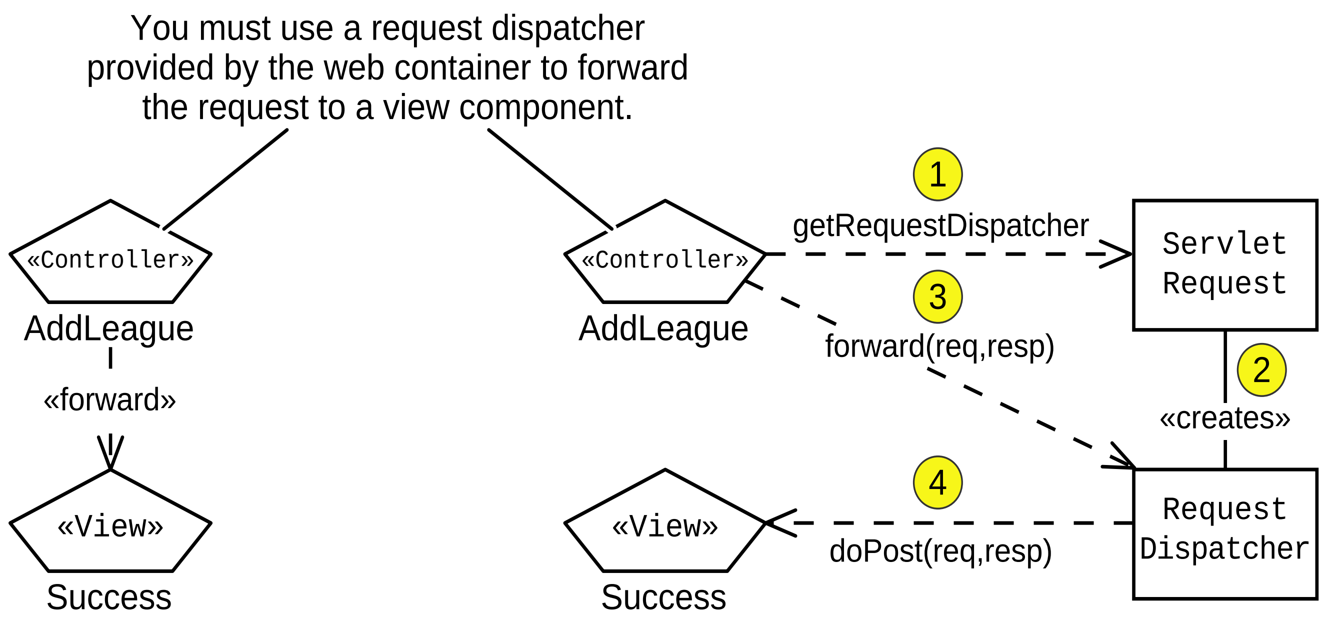
Add League Architecture Model (Stage 2)



Request Scope



Using a Request Dispatcher



Developing the AddLeagueServlet Code

```
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
// Support classes
import java.io.IOException;
import java.io.PrintWriter;
// Model classes
import sl314.model.League;
import java.util.List;
import java.util.LinkedList;

public class AddLeagueServlet extends HttpServlet {
  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
         throws IOException, ServletException {

    // Keep a set of strings to record form processing errors.
    List errorMsgs = new LinkedList();
    // Store this set in the request scope, in case we need to
    // send the ErrorPage view.
    request.setAttribute(“errorMsgs”, errorMsgs);

```

Developing the AddLeagueServlet Code (Part 2)

```
      try {

        // Retrieve form parameters.
        String yearStr = request.getParameter(“year”).trim();
        String season = request.getParameter(“season”).trim();
        String title = request.getParameter(“title”).trim();

        // Perform data conversions.
        int year = -1;
        try {
          year = Integer.parseInt(yearStr);
        } catch (NumberFormatException nfe) {
          errorMsgs.add(“The ‘year’ field must be a positive integer.”)
        }

```

Developing the AddLeagueServlet Code (Part 3)

```

        // Verify form parameters
        if ( (year != -1) && ((year < 2000) || (year > 2010)) ) {
          errorMsgs.add("The 'year' field must within 2000 to 2010.");
        }
        if ( season.equals("UNKNOWN") ) {
          errorMsgs.add("Please select a league season.");
        }
        if ( title.length() == 0 ) {
          errorMsgs.add("Please enter the title of the league.");
        }

        // Send the ErrorPage view if there were errors
        if ( ! errorMsgs.isEmpty() ) {
          RequestDispatcher view
            = request.getRequestDispatcher("error_page.view");
          view.forward(request, response);
          return;
        }

```

Developing the AddLeagueServlet Code (Part 4)

```
        // Perform business logic
        League league = new League(year, season, title);
        // Store the new league in the request-scope
        request.setAttribute(“league”, league);

        // Send the Success view
        RequestDispatcher view
          = request.getRequestDispatcher(“success.view”);
        view.forward(request, response);
        return;

      // Handle any unexpected exceptions
      } catch (RuntimeException e) {
        errorMsgs.add(e.getMessage());
        RequestDispatcher view
          = request.getRequestDispatcher(“error_page.view”);
        view.forward(request, response);

        // Log stack trace
        e.printStackTrace(System.err);
```

The SuccessServlet Code

```

public class SuccessServlet extends HttpServlet {

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
         throws IOException {
    generateView(request, response);
  }

  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
         throws IOException {
    generateView(request, response);
  }

  public void generateView(HttpServletRequest request,
                           HttpServletResponse response)
         throws IOException {

```

The SuccessServlet Code (Part 2)

```
        // Set page title
        String pageTitle = “Duke’s Soccer League: Add League Success”;

        // Retrieve the ‘league’ from the request-scope
        League league = (League) request.getAttribute(“league”);

        // Specify the content type is HTML
        response.setContentType(“text/html”);
        PrintWriter out = response.getWriter();

        // Generate the HTML response
        out.println(“<html>”);
```

The SuccessServlet Code (Part 3)

```

      // Generate main body
      out.println(“<p>”);
      out.print(“Your request to add the “);
      out.print(“<i>” + league.getTitle() + “</i>”);
      out.println(“ league was successful.”);
      out.println(“</p>”);

      out.println(“</body>”);
      out.println(“</html>”);

    } // END of generateResponse method

  } // 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

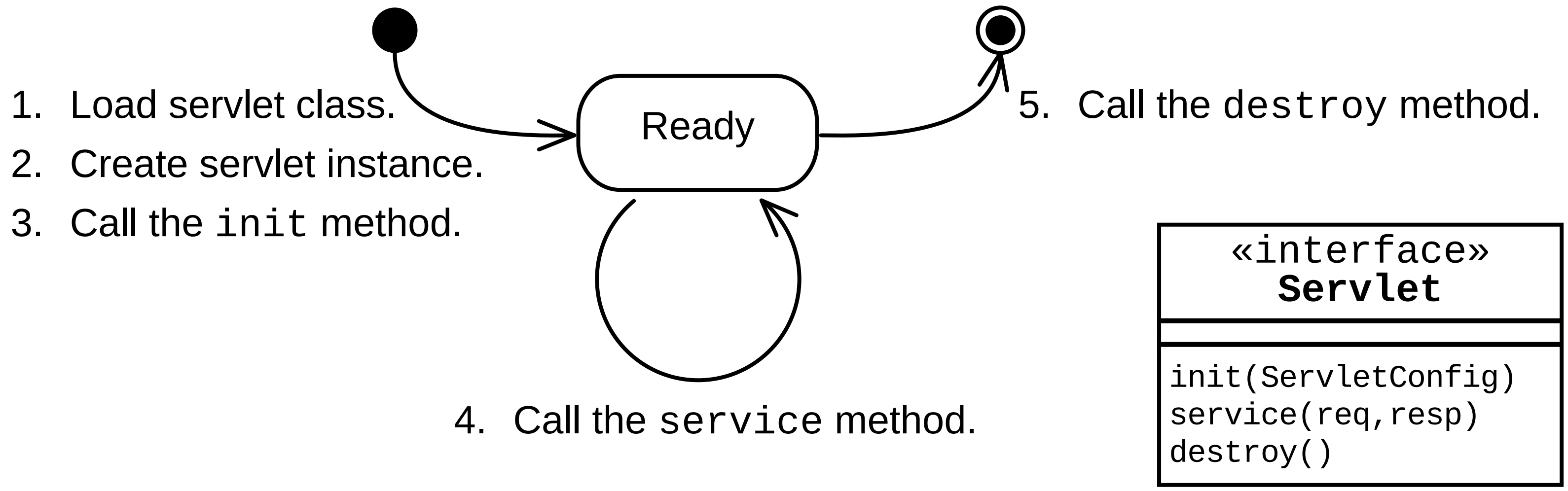
Objectives

- 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?

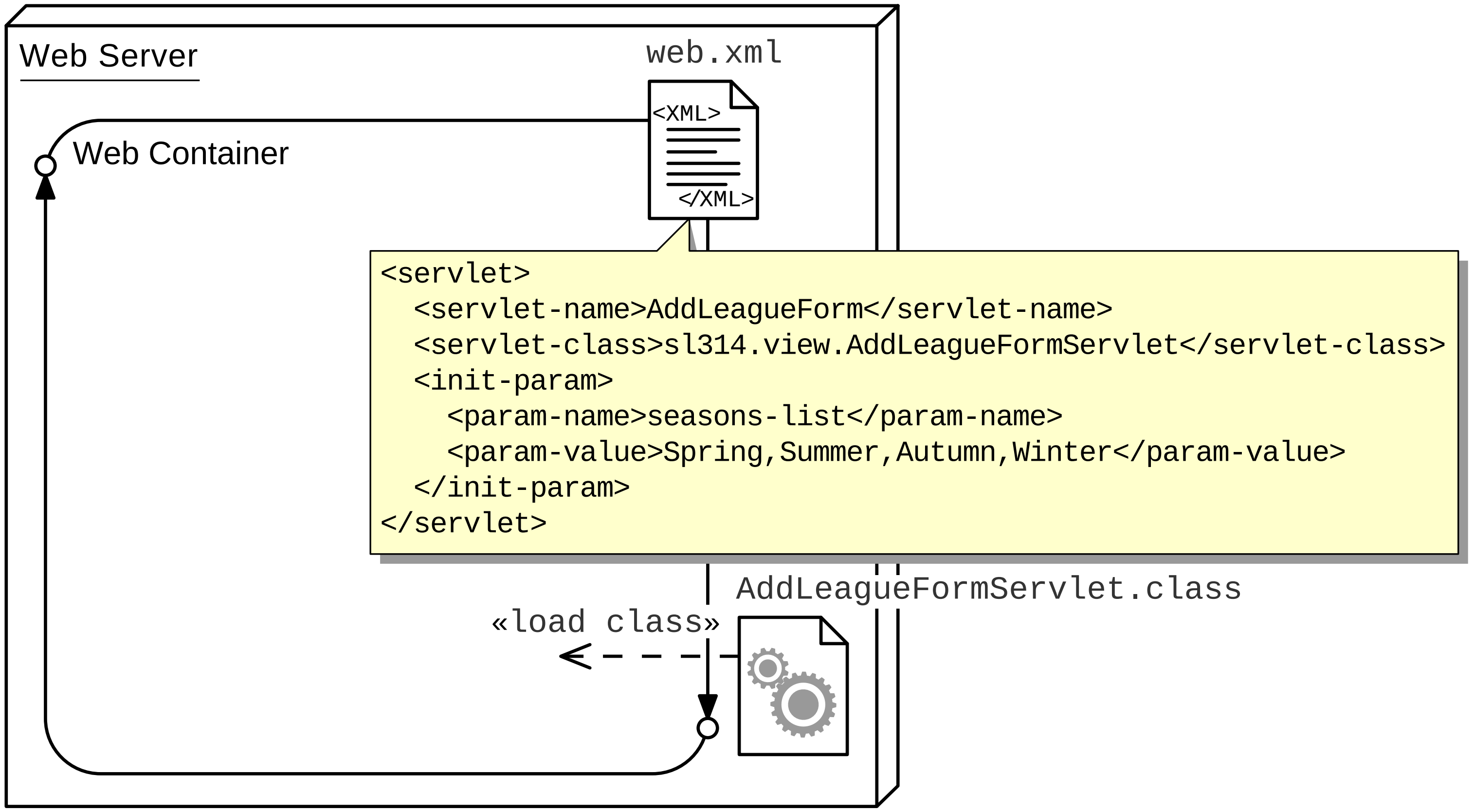
Servlet Life Cycle Overview


1. Load servlet class.
2. Create servlet instance.
3. Call the `init` method.
Ready
5. Call the `destroy` method.
4. Call the `service` method.

```
«interface»
Servlet

init(ServletConfig)
service(req,resp)
destroy()
```

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

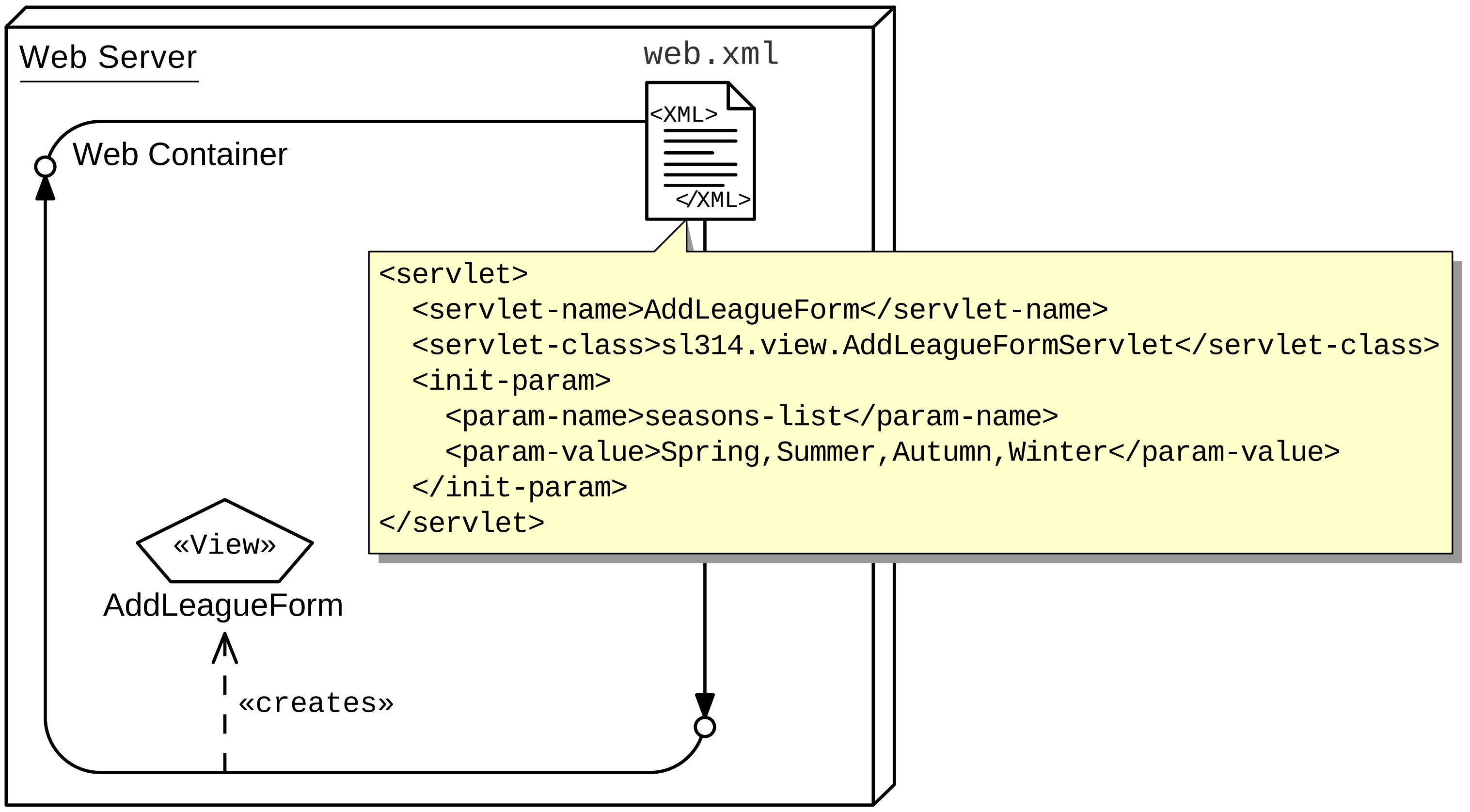
Servlet Class Loading



```
<servlet>
  <servlet-name>AddLeagueForm</servlet-name>
  <servlet-class>sl314.view.AddLeagueFormServlet</servlet-class>
  <init-param>
    <param-name>seasons-list</param-name>
    <param-value>Spring,Summer,Autumn,Winter</param-value>
  </init-param>
</servlet>
```

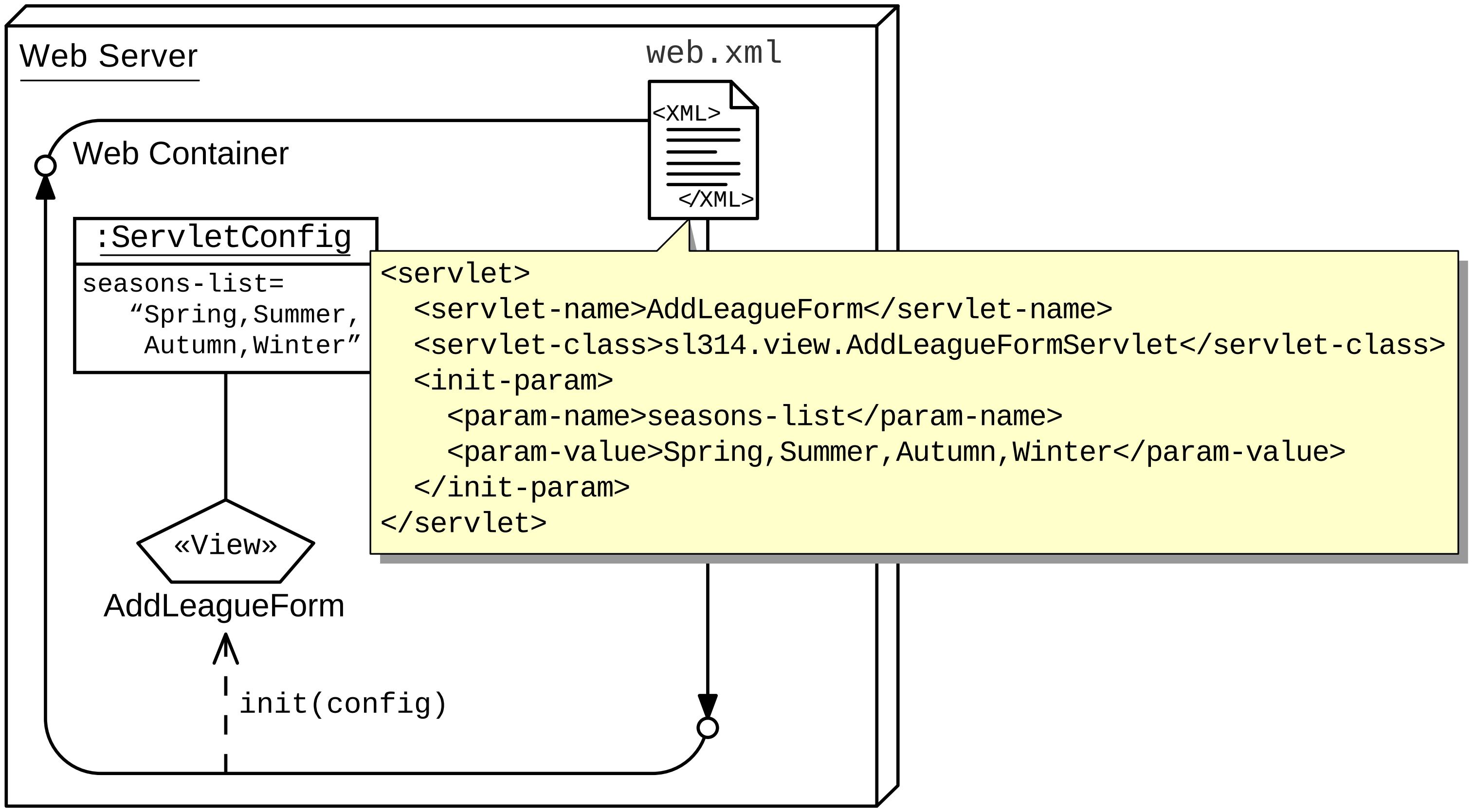
Classes can be in: `WEB-INF/classes/`, `WEB-INF/lib/*.jar`, plus Java SE classes, and container classes.

One Instance Per Servlet Definition



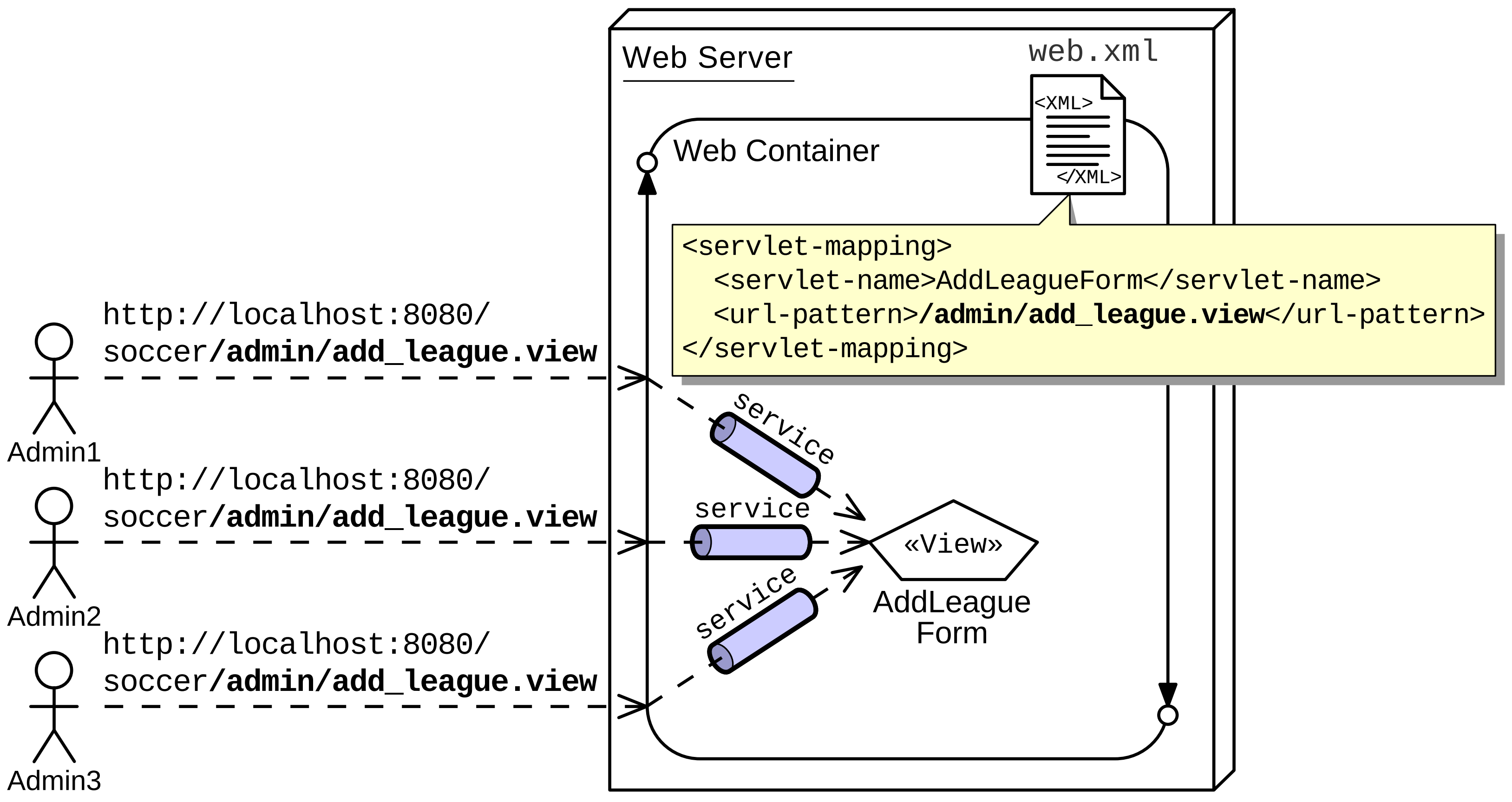
```
<servlet>
  <servlet-name>AddLeagueForm</servlet-name>
  <servlet-class>sl314.view.AddLeagueFormServlet</servlet-class>
  <init-param>
    <param-name>seasons-list</param-name>
    <param-value>Spring,Summer,Autumn,Winter</param-value>
  </init-param>
</servlet>
```

The `init` Life Cycle Method



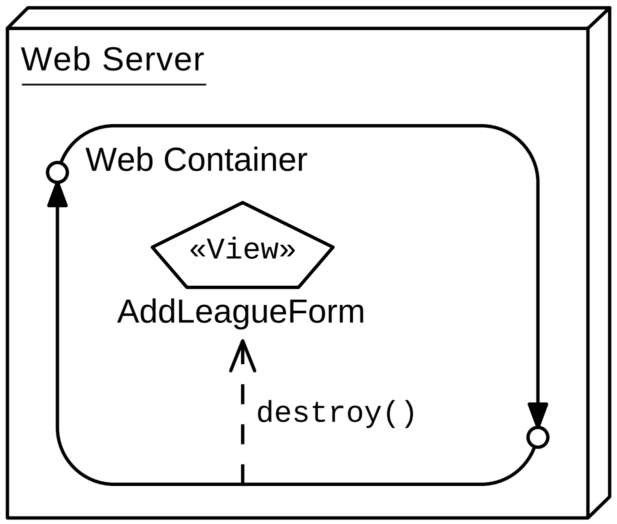
```
<servlet>
  <servlet-name>AddLeagueForm</servlet-name>
  <servlet-class>sl314.view.AddLeagueFormServlet</servlet-class>
  <init-param>
    <param-name>seasons-list</param-name>
    <param-value>Spring,Summer,Autumn,Winter</param-value>
  </init-param>
</servlet>
```

The `service` Life Cycle Method


Web Server
web.xml
<XML>
</XML>
Web Container
http://localhost:8080/ soccer/admin/add_league.view
Admin1
http://localhost:8080/ soccer/admin/add_league.view
Admin2
http://localhost:8080/ soccer/admin/add_league.view
Admin3
service
service
service
«View»
AddLeague Form

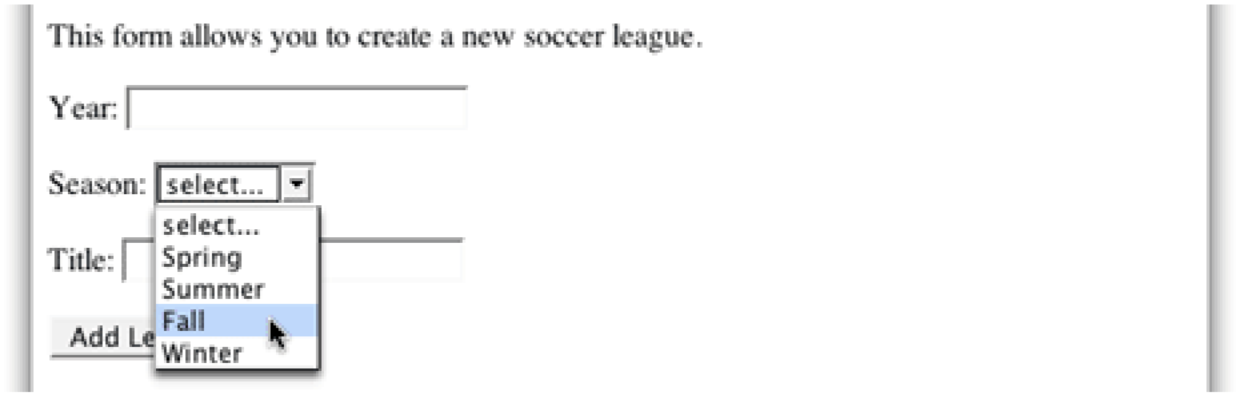
```
<servlet-mapping>
  <servlet-name>AddLeagueForm</servlet-name>
  <url-pattern>/admin/add_league.view</url-pattern>
</servlet-mapping>
```

The `destroy` Life Cycle Method



Customizing the Add a New League Form

US-centric season names:

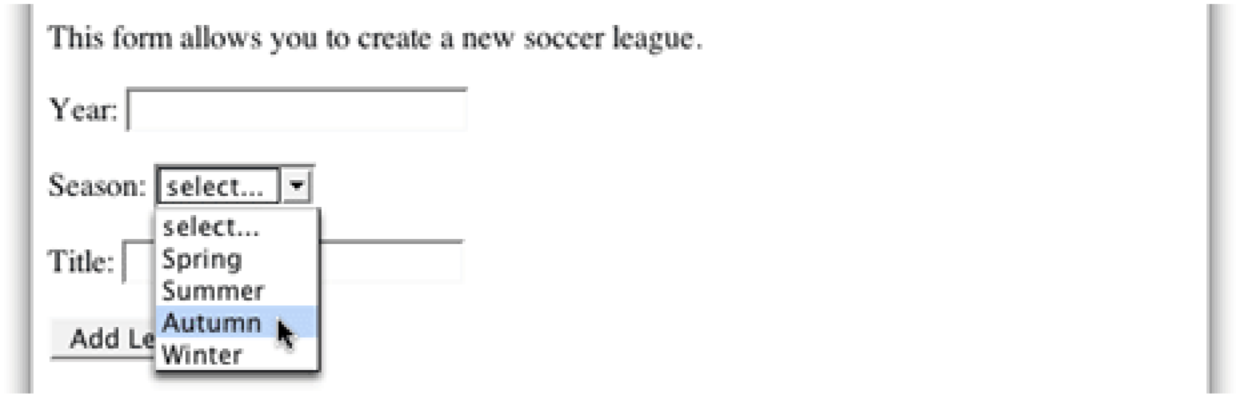


This form allows you to create a new soccer league.

Year:

Season: select...

Customized season names:

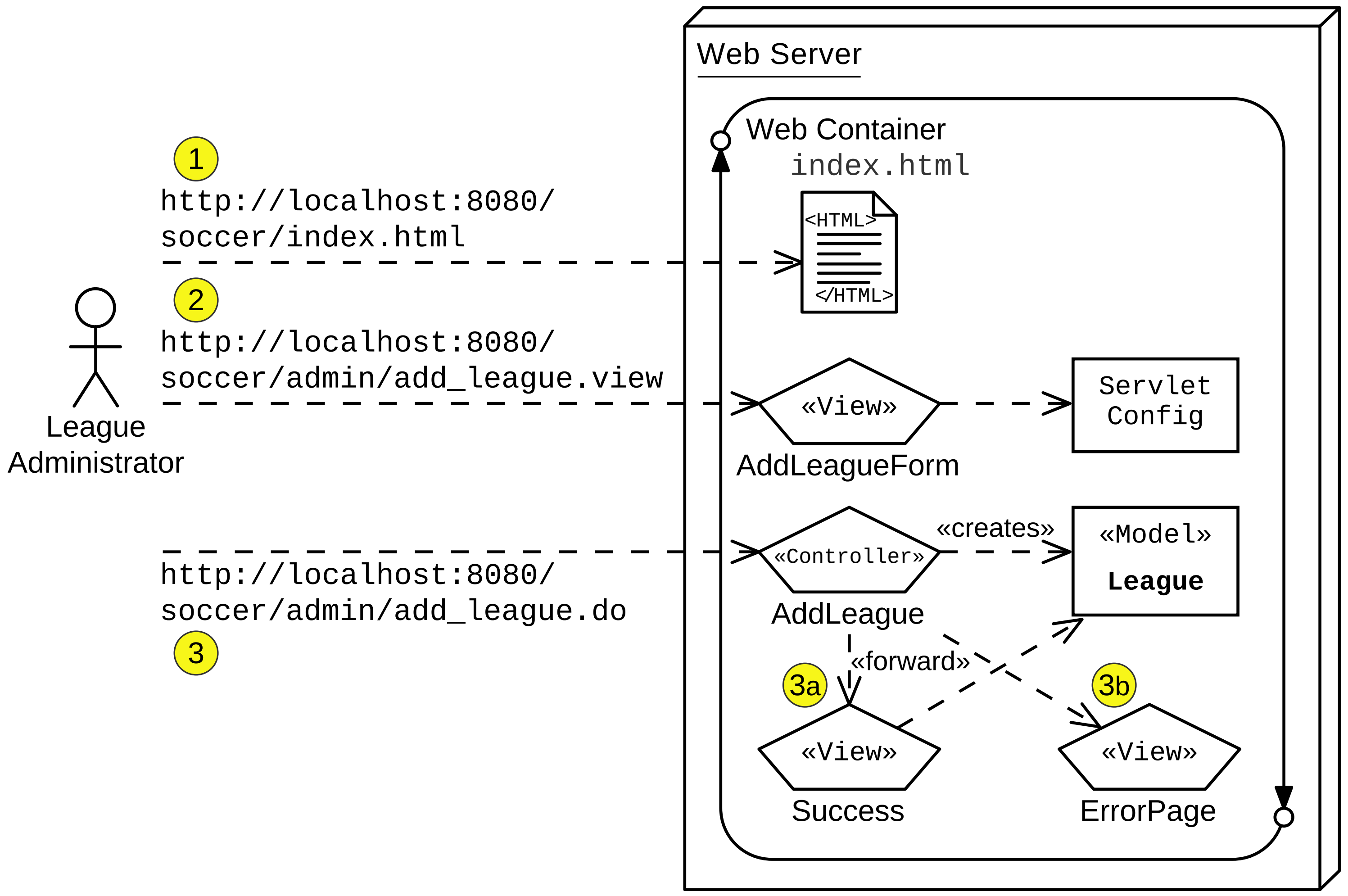


This form allows you to create a new soccer league.

Year:

Season: select...

Add League Architecture Model (Step 1)



The AddLeagueFormServlet Code

```
out.println("</p>");
out.println("<form action='add_league.do' method='POST'>");

// Display the year field
out.println("Year: <input type='text' name='year' /> <br/><br/>");

// Customize the season drop-down menu
out.println("Season: <select name='season'>");
out.println("          <option value='UNKNOWN'>select...</option>");
for ( int i = 0; i < SEASONS.length; i++ ) {
  out.print("          <option value='" + SEASONS[i] + "'");
  out.println(">" + SEASONS[i] + "</option>");
}
out.println("        </select> <br/><br/>");

// Display the title field
out.println("Title: <input type='text' name='title' /> <br/><br/>");

out.println("<input type='Submit' value='Add League' />");
out.println("</form>");
```

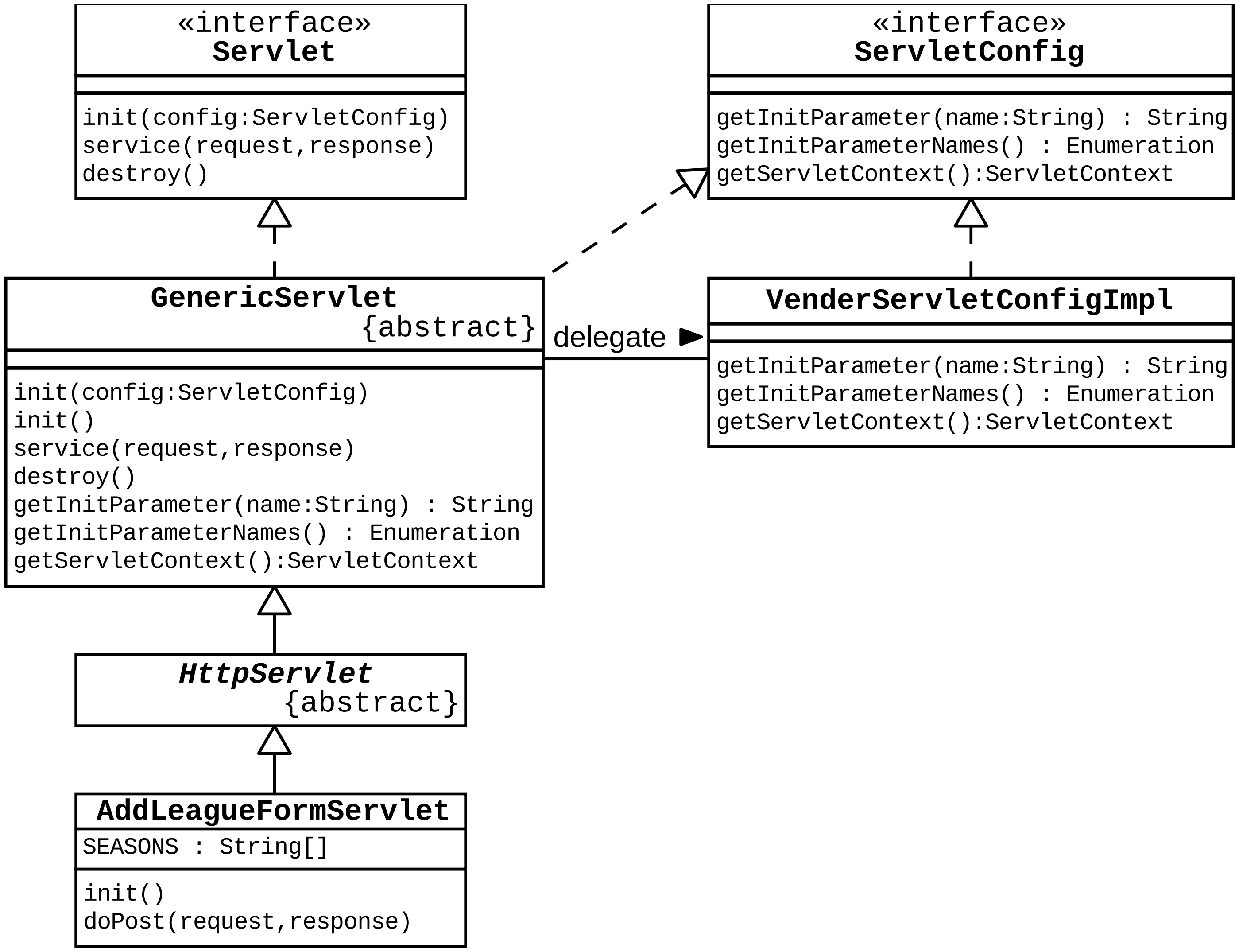
Configuring Initialization Parameters

Deployment descriptor for a servlet initialization parameter:

```
<servlet>
  <servlet-name>AddLeagueForm</servlet-name>
  <servlet-class>sl314.view.AddLeagueFormServlet</servlet-class>
  <init-param>
    <param-name>seasons-list</param-name>
    <param-value>Spring,Summer,Autumn,Winter</param-value>
  </init-param>
</servlet>
```

A servlet can have any number of initialization parameters.

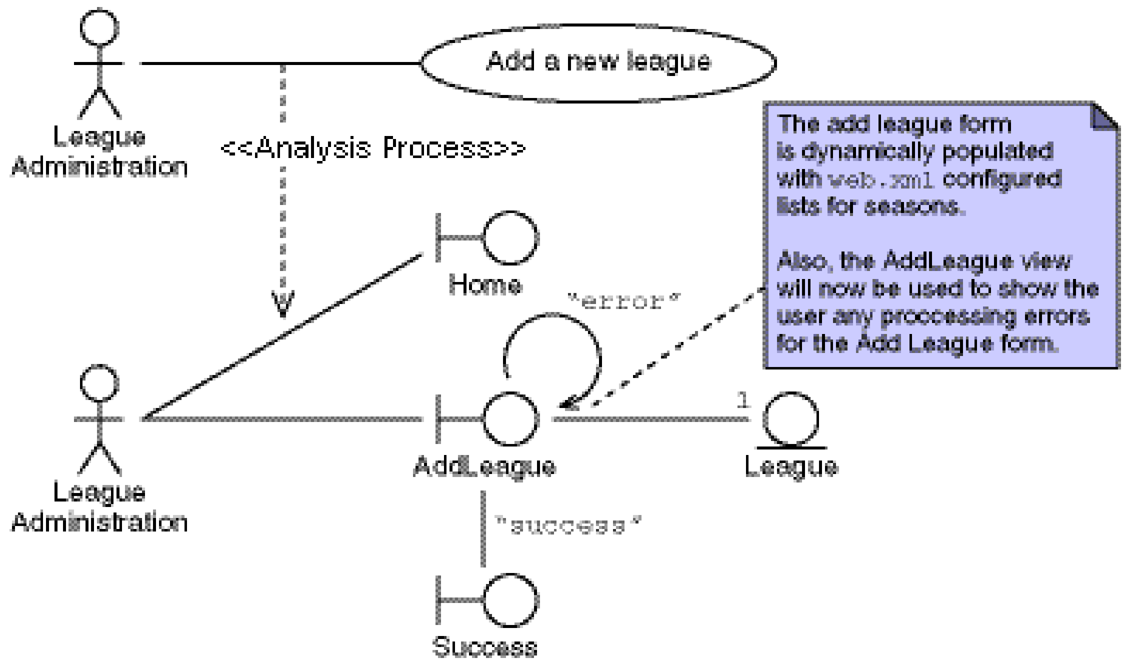
The ServletConfig API


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

The AddLeagueFormServlet Code

```

public class AddLeagueFormServlet extends HttpServlet {

  /** There are the default seasons for the US. */
  private static final String DEFAULT_SEASONS
    = “Spring,Summer,Fall,Winter”;

  /** This variable holds the set of seasons. */
  private String[] SEASONS;

  /** The init method configures the set of seasons. */
  public void init() {
    String seasons_list = getInitParameter(“seasons-list”);
    if ( seasons_list == null ) {
      seasons_list = DEFAULT_SEASONS;
    }
    SEASONS = seasons_list.split(“,”);
  }

```

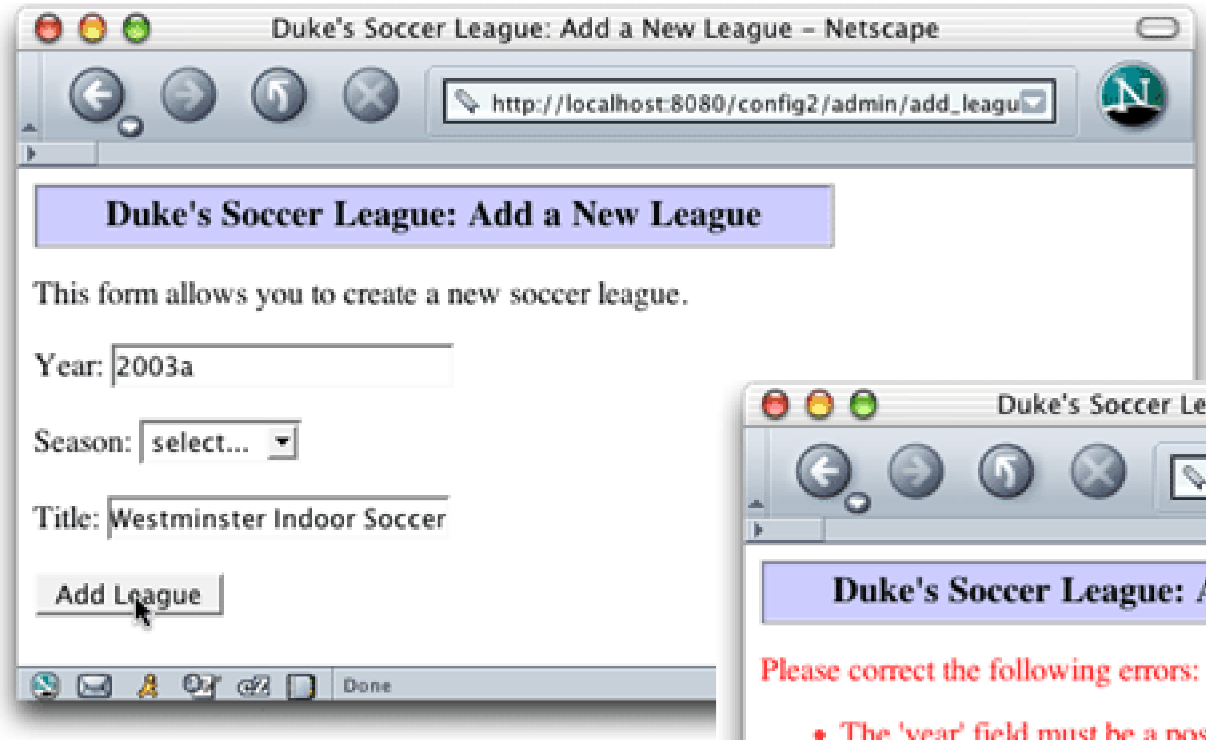
Add League Analysis Model (Stage 2)



The add league form is dynamically populated with `web.xml` configured lists for seasons.

Also, the AddLeague view will now be used to show the user any proccessing errors for the Add League form.

Error Handling Screen Shots



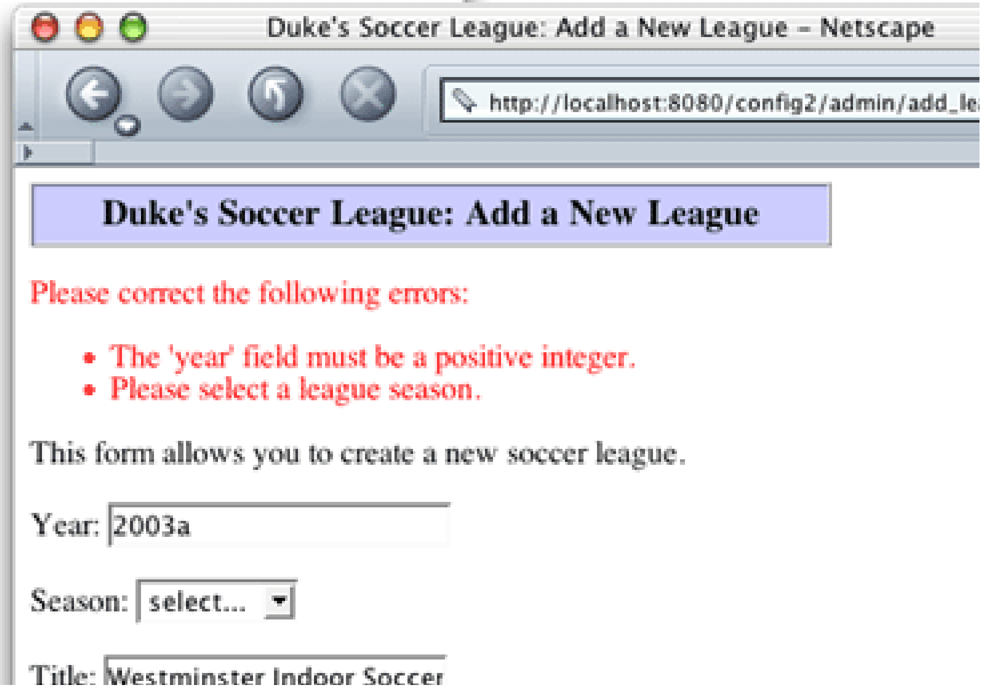
Duke's Soccer League: Add a New League – Netscape

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:	2003a
Season:	select...
Title:	Westminster Indoor Soccer



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

http://localhost:8080/config2/admin/add_le

Duke's Soccer League: Add a New League

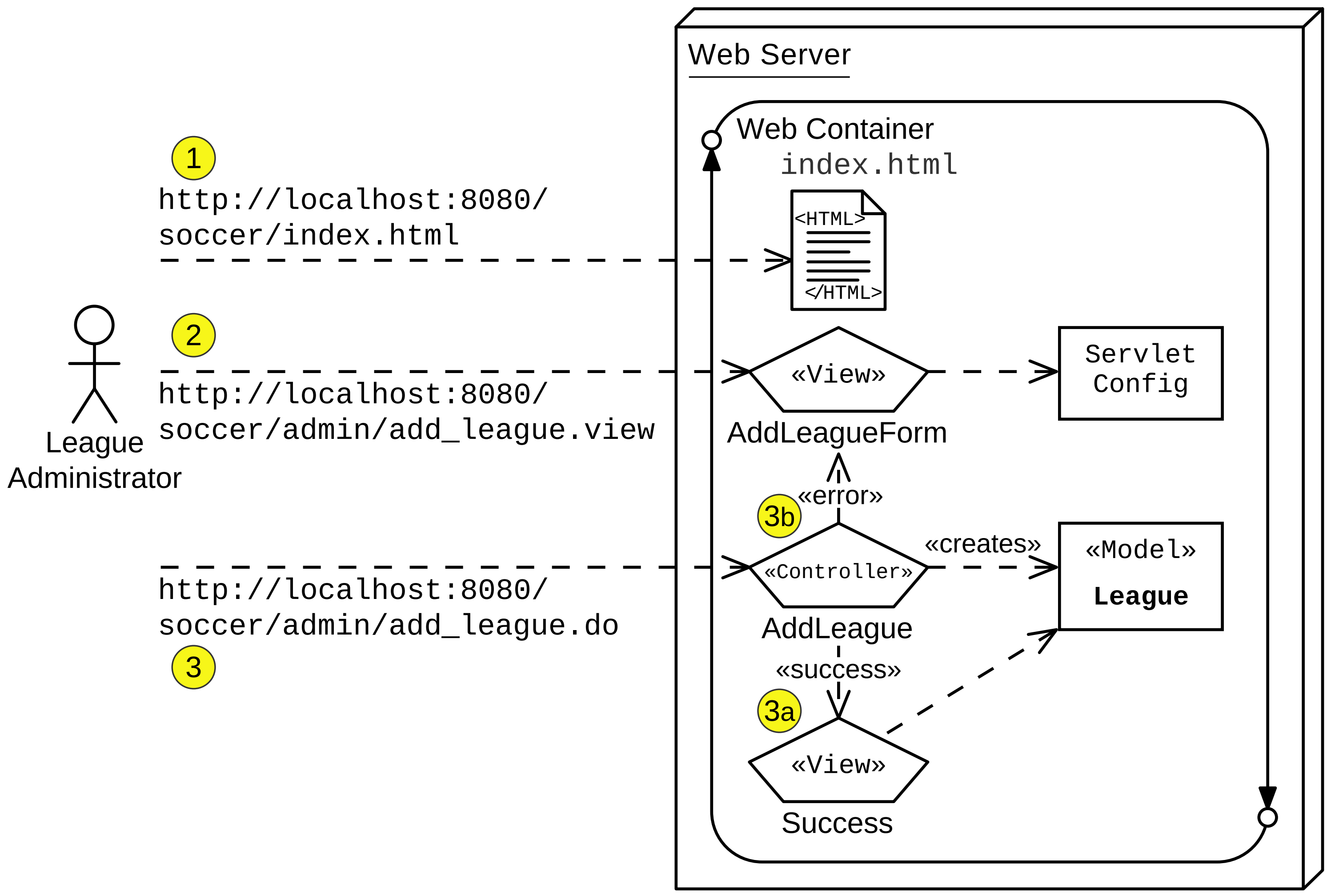
Please correct the following errors:

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

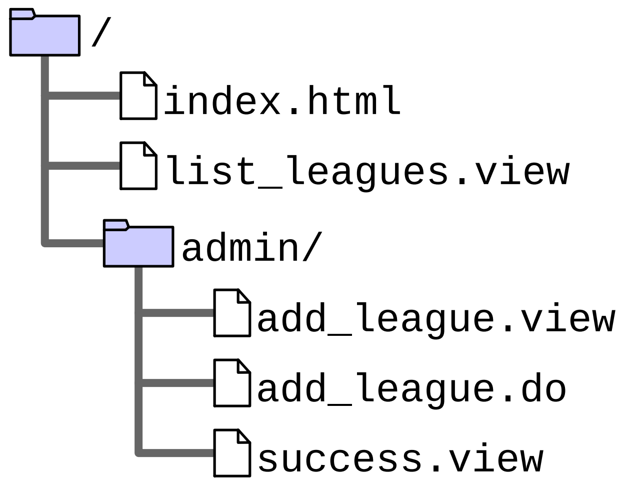
This form allows you to create a new soccer league.

Year:	2003a
Season:	select...
Title:	Westminster Indoor Soccer

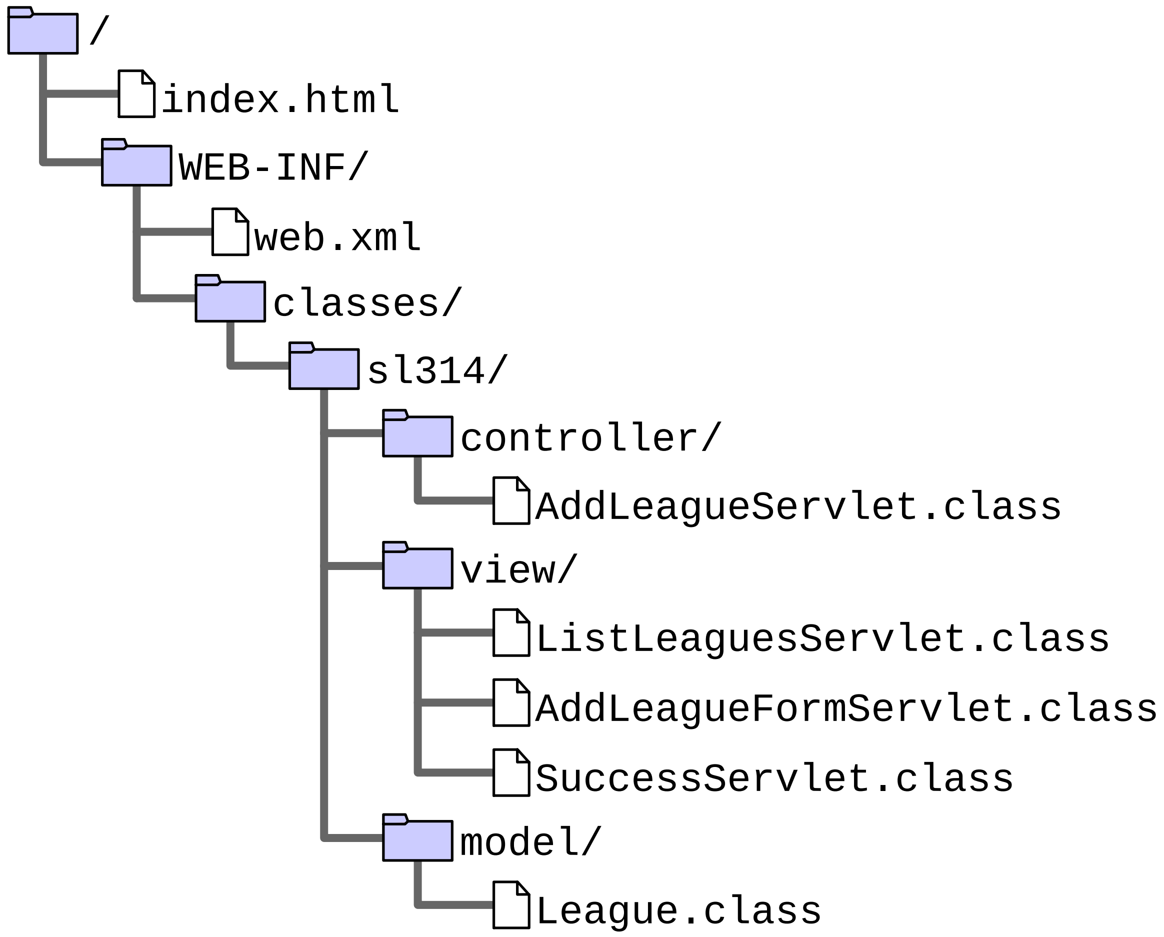
Add League Architecture Model (Stage 2)



Soccer League Web Application Structure



Soccer League Web Application Structure (continued)



The AddLeagueServlet Code

```

        // Verify form parameters
        if ( (year != -1) && ((year < 2000) || (year > 2010)) ) {
          errorMsgs.add(“The ‘year’ field must within 2000 to 2010.”);
        }
        if ( season.equals(“UNKNOWN”) ) {
          errorMsgs.add(“Please select a league season.”);
        }
        if ( title.length() == 0 ) {
          errorMsgs.add(“Please enter the title of the league.”);
        }

        // Send the user back to the AddDVD form, if there were errors
        if ( ! errorMsgs.isEmpty() ) {
          RequestDispatcher view
            = request.getRequestDispatcher(“add_league.view”);
          view.forward(request, response);
          return;
        }
```

The AddLeagueFormServlet Code

```

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
           throws IOException {
      generateView(request, response);
    }

    public void doPost(HttpServletRequest request,
                       HttpServletResponse response)
           throws IOException {
      generateView(request, response);
    }

    public void generateView(HttpServletRequest request,
                             HttpServletResponse response)
           throws IOException {
```

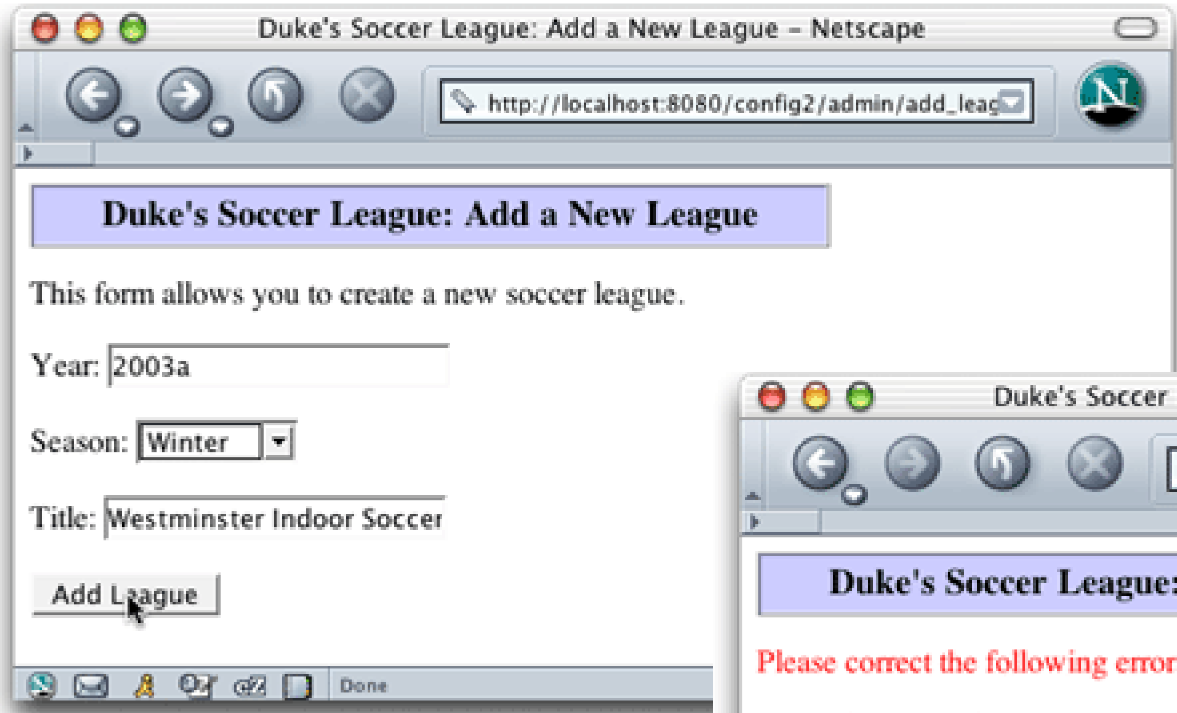
The AddLeagueFormServlet Code (Part 2)

```
  public void generateView(HttpServletRequest request,
                           HttpServletResponse response)
         throws IOException {

    // Set page title
    String pageTitle = “Duke’s Soccer League: Add a New League”;

    // Retrieve the errorMsgs from the request-scope
    List errorMsgs = (List) request.getAttribute(“errorMsgs”);

    // Specify the content type is HTML
    response.setContentType(“text/html”);
    PrintWriter out = response.getWriter();

    // Generate the HTML response
    out.println(“<html>”);
    out.println(“<head>”);
    out.println(“  <title>” + pageTitle + “</title>”);
    out.println(“</head>”);
    out.println(“<body bgcolor=’white’>”);

```

The `AddLeagueFormServlet` Code (Part 3)

```

    // Report any errors (if any)
    if ( errorMsgs != null ) {
      out.println(“<p>”);
      out.println(“<font color=’red’>Please correct the following errors:”);
      out.println(“<ul>”);
      Iterator items = errorMsgs.iterator();
      while ( items.hasNext() ) {
        String message = (String) items.next();
        out.println(“  <li>” + message + “</li>”);
      }
      out.println(“</ul>”);
      out.println(“</font>”);
      out.println(“</p>”);
    }

```

Repopulating Web Forms


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

http://localhost:8080/config2/admin/add_leag

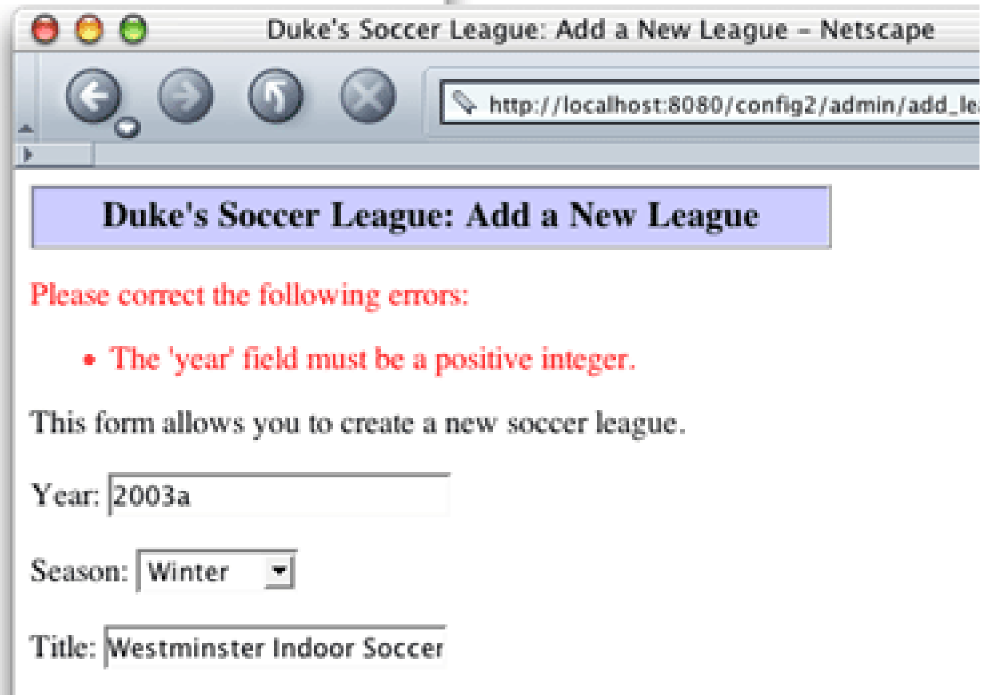
Duke's Soccer League: Add a New League

This form allows you to create a new soccer league.

Year: 2003a

Season: Winter

Title: Westminster Indoor Soccer


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: 2003a

Season: Winter

Title: Westminster Indoor Soccer

Repopulating a Text Field

```

        // Generate main body
        out.println(“<p>”);
        out.println(“This form allows you to create a new soccer league.”);
        out.println(“</p>”);
        out.println(“<form action=’add_league.do’ method=’POST’>”);

        // Repopulate the year field
        String year = request.getParameter(“year”);
        if ( year == null ) {
          year = ““;
        }
        out.println(“Year: <input type=’text’ name=’year’ value=’”
                    + year + “‘ /> <br/><br/>”);

```

Repopulating a Drop-Down List

```

    // Repopulate the season drop-down menu
    String season = request.getParameter(“season”);
    out.println(“Season: <select name=’season’>”);
    if ( (season == null) || season.equals(“UNKNOWN”) ) {
      out.println(“          <option value=’UNKNOWN’>select...</option>”);
    }
    for ( int i = 0; i < SEASONS.length; i++ ) {
      out.print(“          <option value=’” + SEASONS[i] + “‘”);
      if ( SEASONS[i].equals(season) ) {
        out.print(“ selected”);
      }
      out.println(“>” + SEASONS[i] + “</option>”);
    }
    out.println(“        </select> <br/><br/>”);

```

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

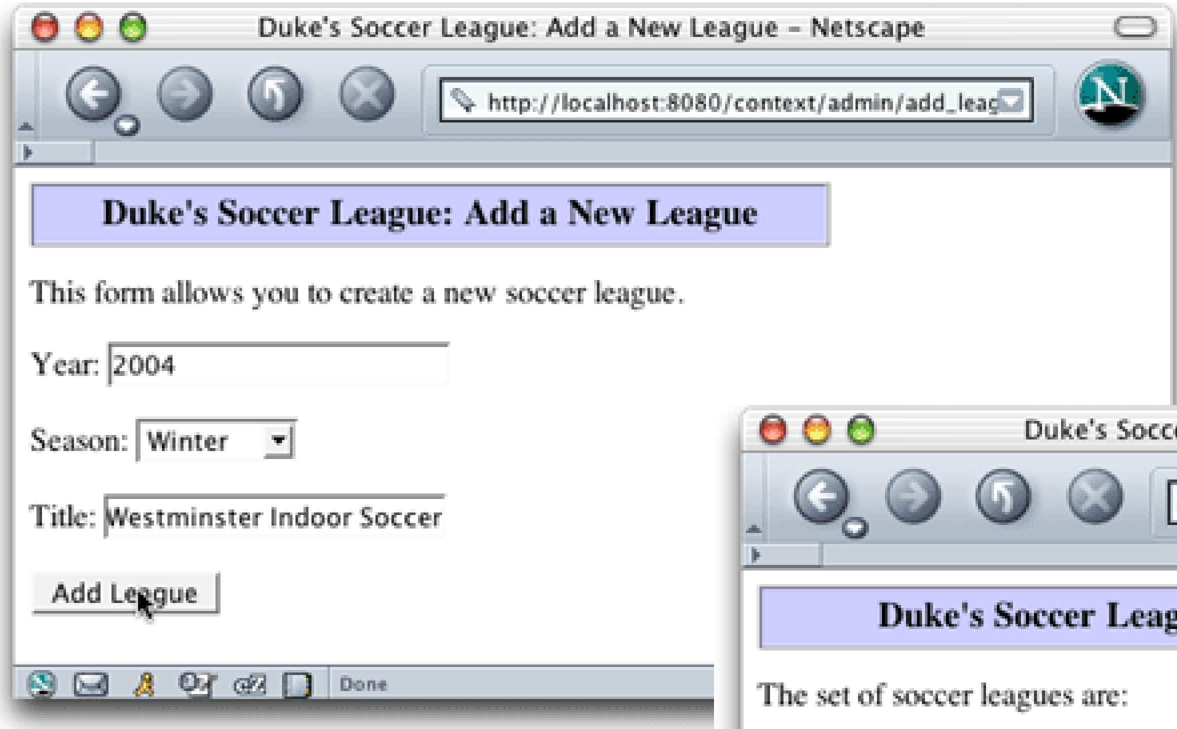
Objectives

- 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



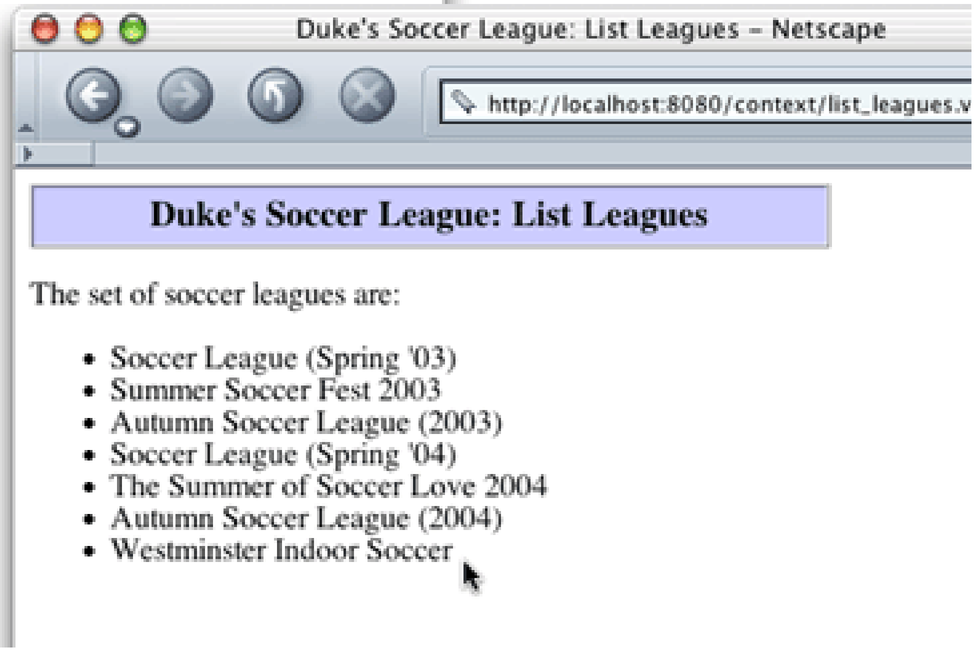
Duke's Soccer League: Add a New League – Netscape

http://localhost:8080/context/admin/add_leag

Duke's Soccer League: Add a New League

This form allows you to create a new soccer league.

Year:	2004
Season:	Winter
Title:	Westminster Indoor Soccer



Duke's Soccer League: List Leagues – Netscape

http://localhost:8080/context/list_leagues.v

Duke's Soccer League: List Leagues

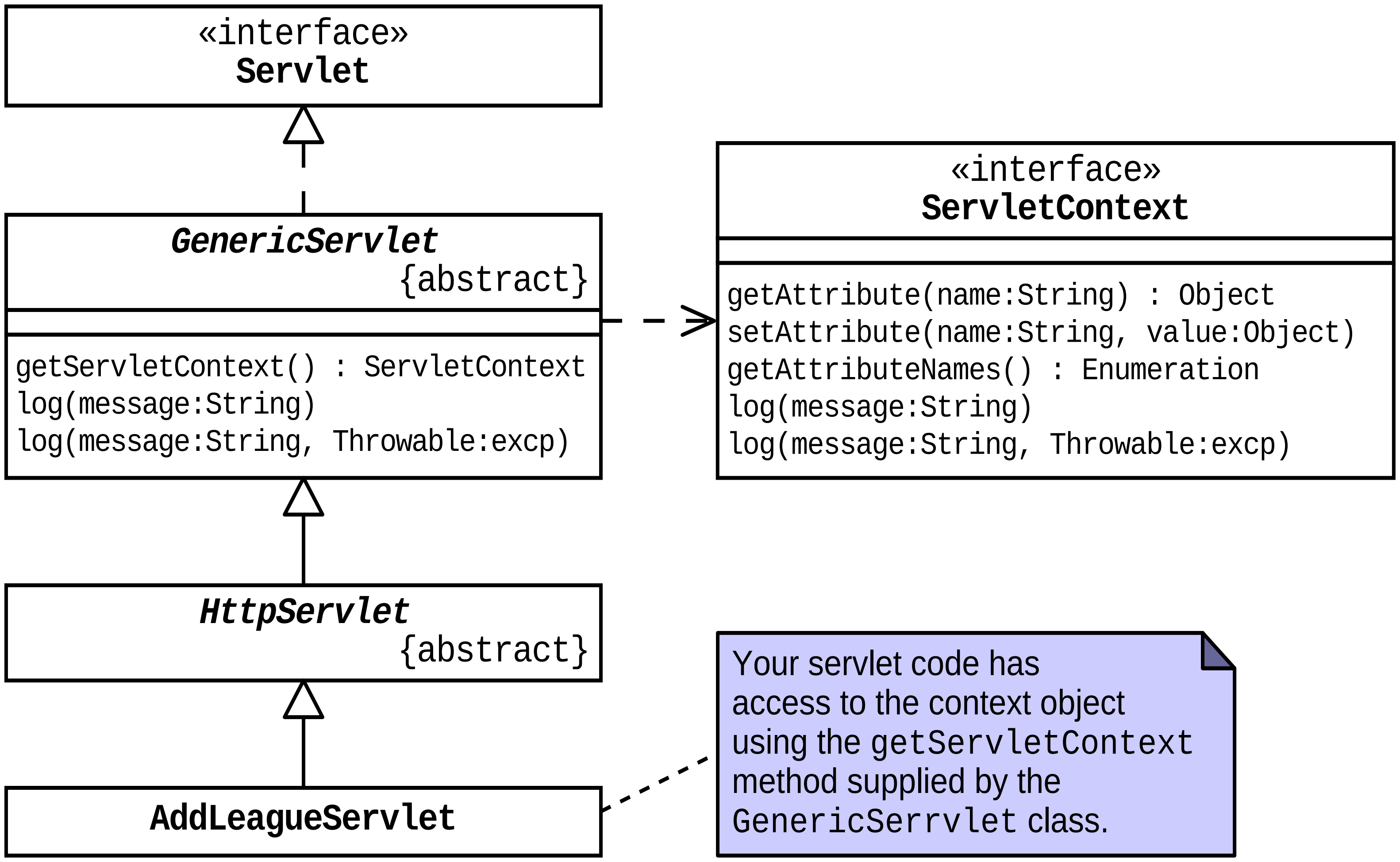
The set of soccer leagues are:

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

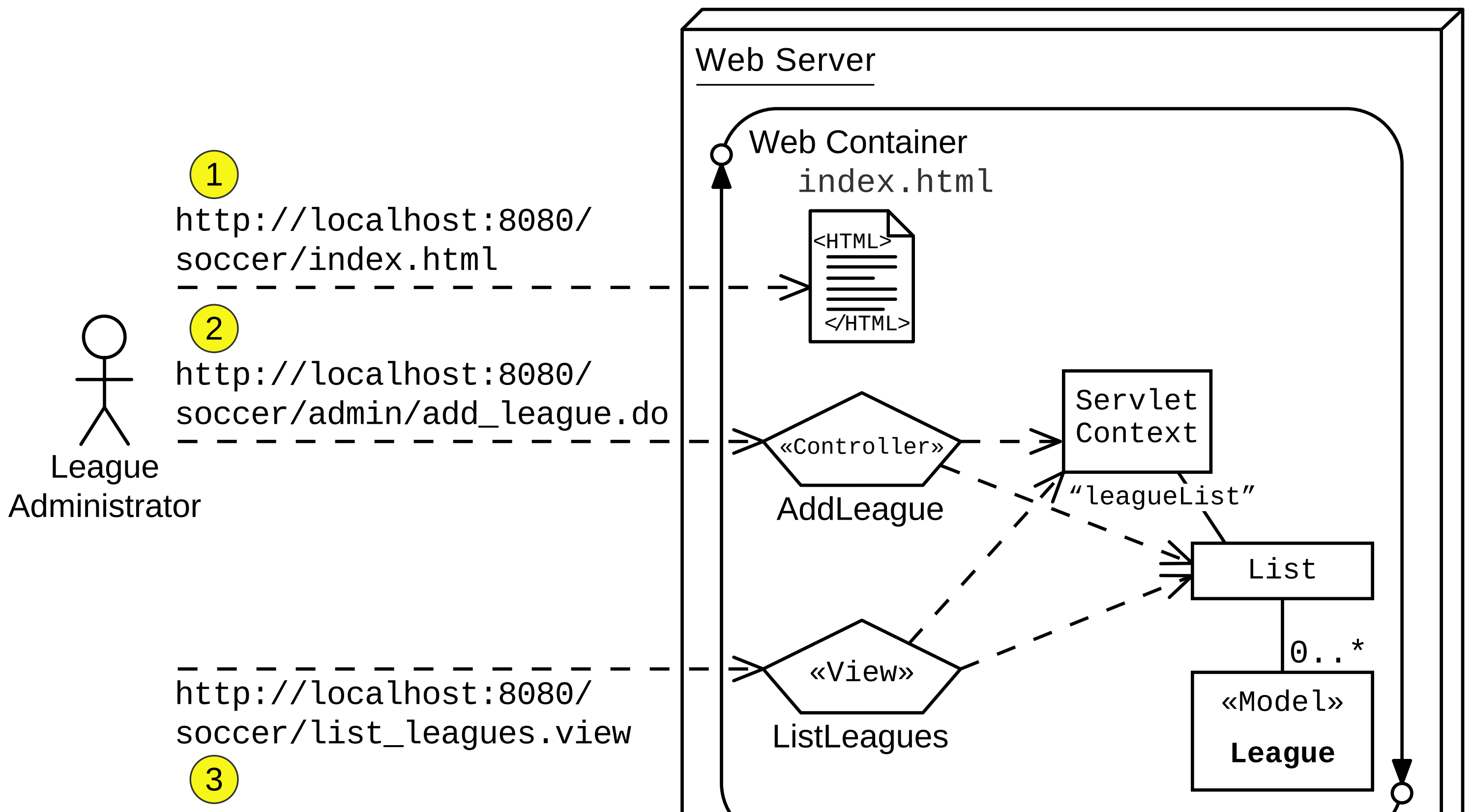
Servlet Context

- 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


«interface»
Servlet
GenericServlet
{abstract}
getServletContext() : ServletContext
log(message:String)
log(message:String, Throwable:excp)
HttpServlet
{abstract}
AddLeagueServlet
«interface»
ServletContext
getAttribute(name:String) : Object
setAttribute(name:String, value:Object)
getAttributeNames() : Enumeration
log(message:String)
log(message:String, Throwable:excp)
Your servlet code has access to the context object using the getServletContext method supplied by the GenericSerrvlet class.

Soccer League Architecture Model



Modified `AddLeagueServlet` Code

```

        // Perform business logic
        League league = new League(year, season, title);
        // Store the new league in the request-scope
        request.setAttribute(“league”, league);

        // Store the new league in the leagueList context-scope attribute
        ServletContext context = getServletContext();
        List leagueList = (List) context.getAttribute(“leagueList”);
        leagueList.add(league);

        // Send the Success view
        RequestDispatcher view
          = request.getRequestDispatcher(“success.view”);
        view.forward(request, response);
        return;
```

Modified ListLeaguesServlet Code

```

public class ListLeaguesServlet extends HttpServlet {

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
         throws IOException {

    // Set page title
    String pageTitle = “Duke’s Soccer League: List Leagues”;

    // Retrieve the list of leagues from the context-scope
    ServletContext context = getServletContext();
    List leagueList = (List) context.getAttribute(“leagueList”);

    // Specify the content type is HTML
    response.setContentType(“text/html”);
    PrintWriter out = response.getWriter();

    // Generate the HTML response
    out.println(“<html>”);
```

Modified `ListLeaguesServlet` Code (Part 2)

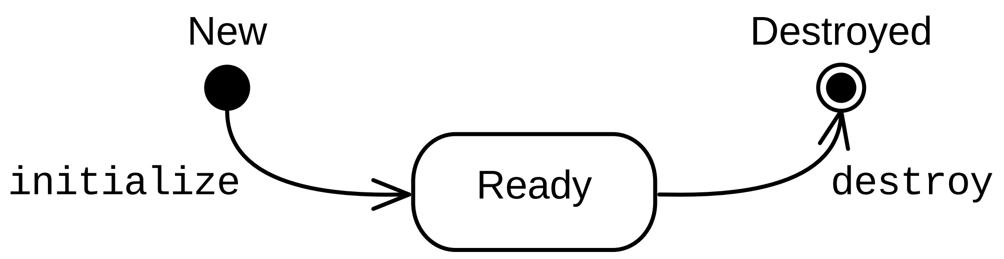
```

    // Generate main body
    out.println(“<p>”);
    out.println(“The set of soccer leagues are:”);
    out.println(“</p>”);

    out.println(“<ul>”);
    Iterator items = leagueList.iterator();
    while ( items.hasNext() ) {
      League league = (League) items.next();
      out.println(“  <li>” + league.getTitle() + “</li>”);
    }
    out.println(“</ul>”);
```

League List Initialization Example

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

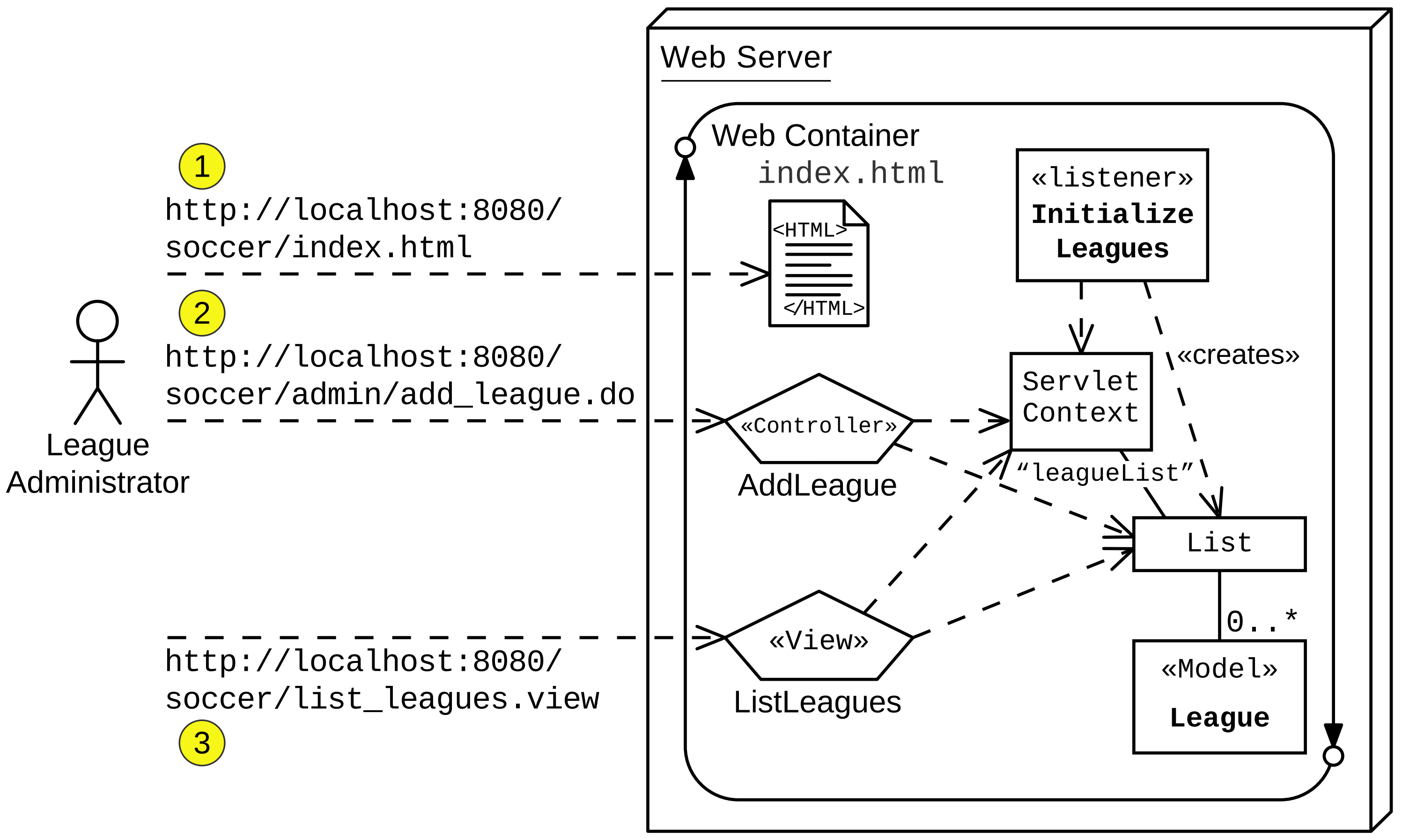
1. Determine the location of the `leagues.txt` file.
2. Read the `leagues.txt` file.
3. Create `League` objects for each row in the `leagues.txt` file and store them in a `List` object.
4. Store the list of leagues in the `leagueList` context attribute.
5. Log the fact that the list was initialized,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

Web Application Life Cycle

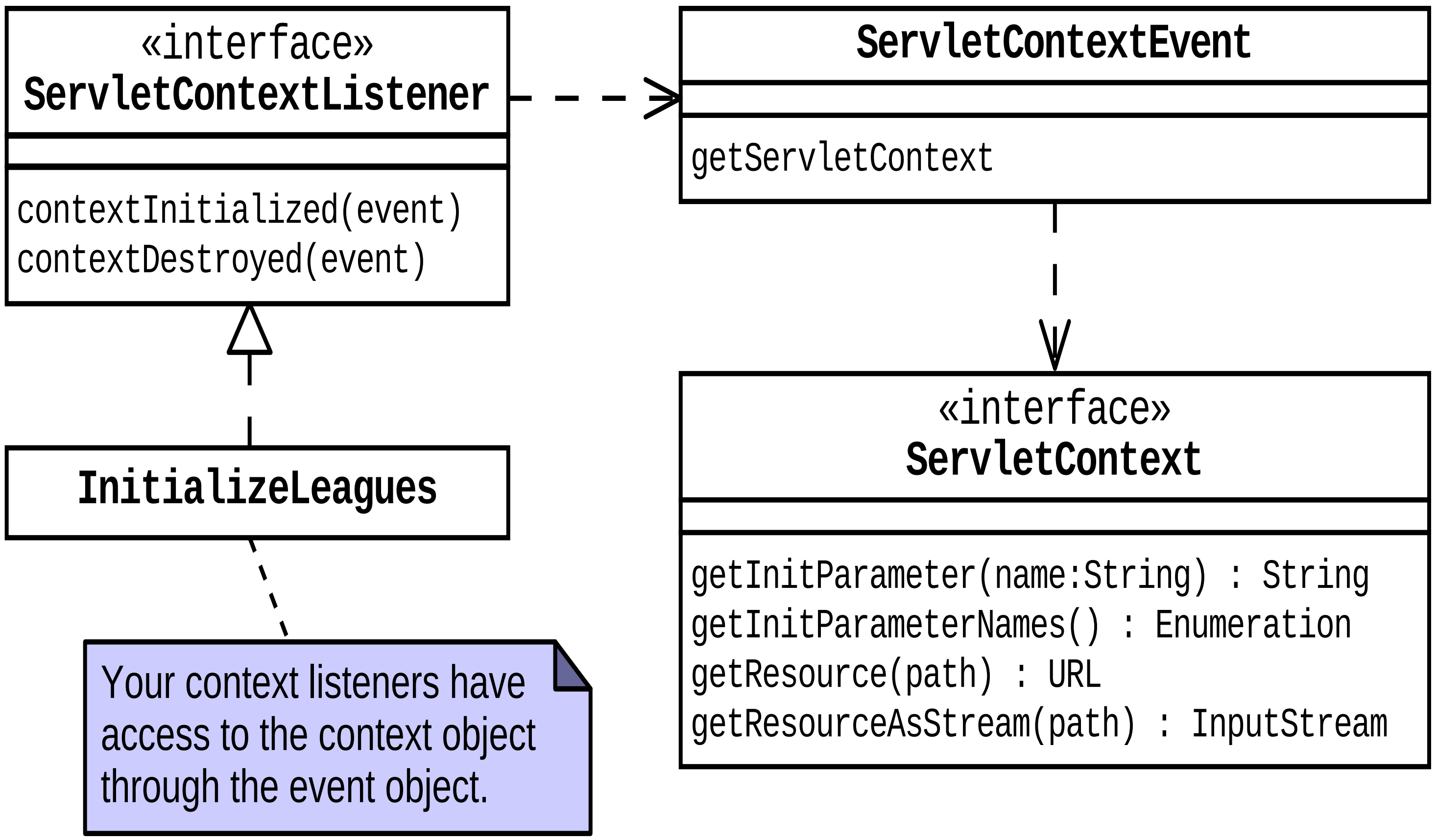


- When the web container is started, each web application is initialized.
- When the web container is shut down, each web application is destroyed.
- A servlet context listener can be used to receive these web application life cycle events.

Soccer League Architecture Model (Revisited)



The ServletContextListener API


«interface»
ServletContextListener
contextInitialized(event)
contextDestroyed(event)
ServletContextEvent
getServletContext
InitializeLeagues
«interface»
ServletContext
getInitParameter(name:String) : String
getInitParameterNames() : Enumeration
getResource(path) : URL
getResourceAsStream(path) : InputStream
Your context listeners have access to the context object through the event object.

The InitializeLeagues Code

```
import javax.servlet.ServletContextListener;
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContext;
// Support classes
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.BufferedReader;
// Model classes
import sl314.model.League;
import java.util.List;
import java.util.LinkedList;

public class InitializeLeagues implements ServletContextListener {

  public void contextInitialized(ServletContextEvent event) {
    ServletContext context = event.getServletContext();
    List leagueList = new LinkedList();
    String leaguesFile = context.getInitParameter(“leagues-file”);
    InputStream is = null;
    BufferedReader reader = null;

```

The InitializeLeagues Code (Part 2)

```

    try {
      is = context.getResourceAsStream(leaguesFile);
      reader = new BufferedReader(new InputStreamReader(is));
      String record;

      // Read every record (one per line)
      while ( (record = reader.readLine()) != null ) {
        String[] fields = record.split(“\t”);

        // Extract the data fields for the record
        int year = Integer.parseInt(fields[0]);
        String season = fields[1];
        String title = fields[2];

        // Add the new League item to the list
        League item = new League(year, season, title);
        leagueList.add(item);
      }
```

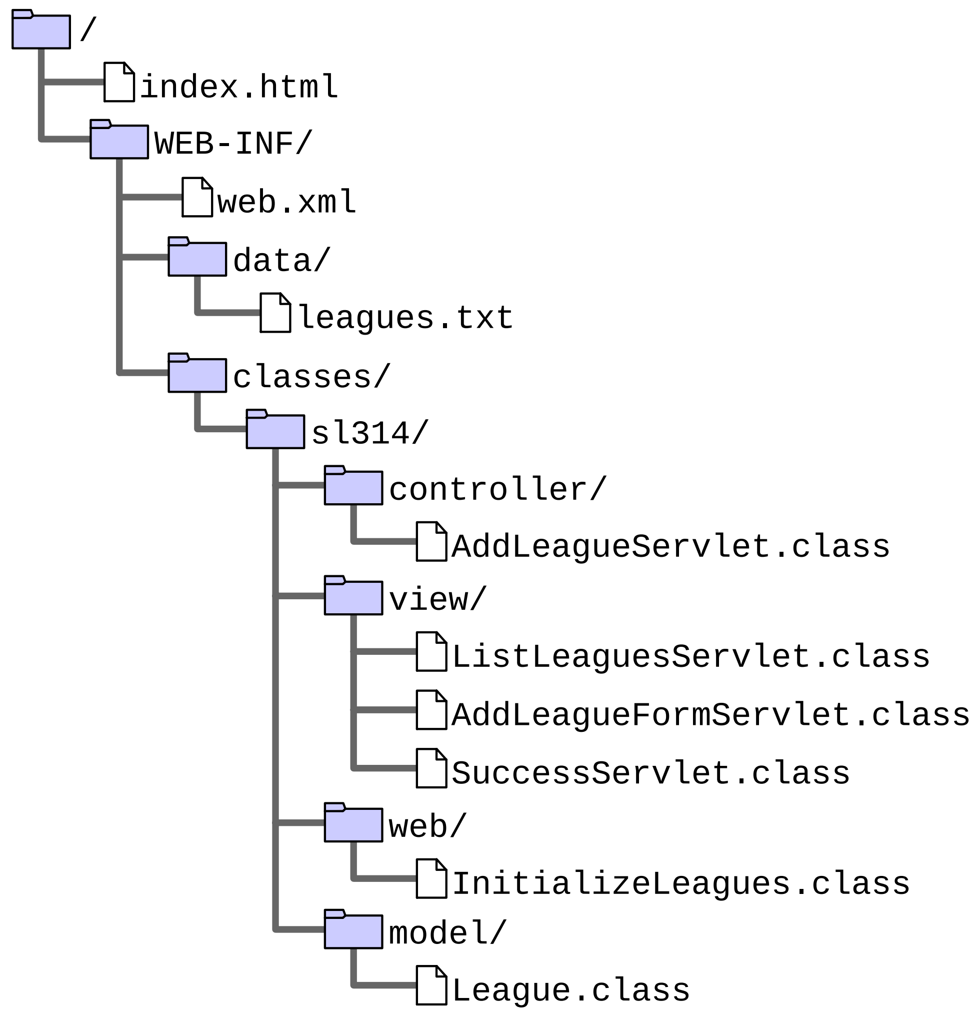
The InitializeLeagues Code (Part 3)

```
        context.setAttribute(“leagueList”, leagueList);

        context.log(“The league list has been loaded.”);

      } catch (Exception e) {
        context.log(“Exception occured while processing the leagues file.”, e);

      } finally {
        if ( is != null ) {
          try { is.close(); } catch (Exception e) {}
        }
        if ( reader != null ) {
          try { reader.close(); } catch (Exception e) {}
        }
      }

    } // END of contextInitialized
```

Soccer League Deployment Descriptor

```
<web-app
    xmlns=”http://java.sun.com/xml/ns/j2ee”
    xmlns:xsi=”http://www.w3.org/2001/XMLSchema-instance”
    xsi:schemaLocation=”http://java.sun.com/xml/ns/j2ee
                     http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd”
    version=”2.4”>

  <display-name>SL-314 WebApp Example</display-name>
  <description>
    This Web Application demonstrates using the context-scope to store
    a common resource: the “leagueList” for the Soccer League webapp.
  </description>

  <context-param>
    <param-name>leagues-file</param-name>
    <param-value>/WEB-INF/data/leagues.txt</param-value>
  </context-param>

  <listener>
    <listener-class>sl314.web.InitializeLeagues</listener-class>
  </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

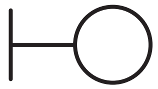
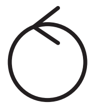
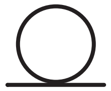
Relevance

- 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?

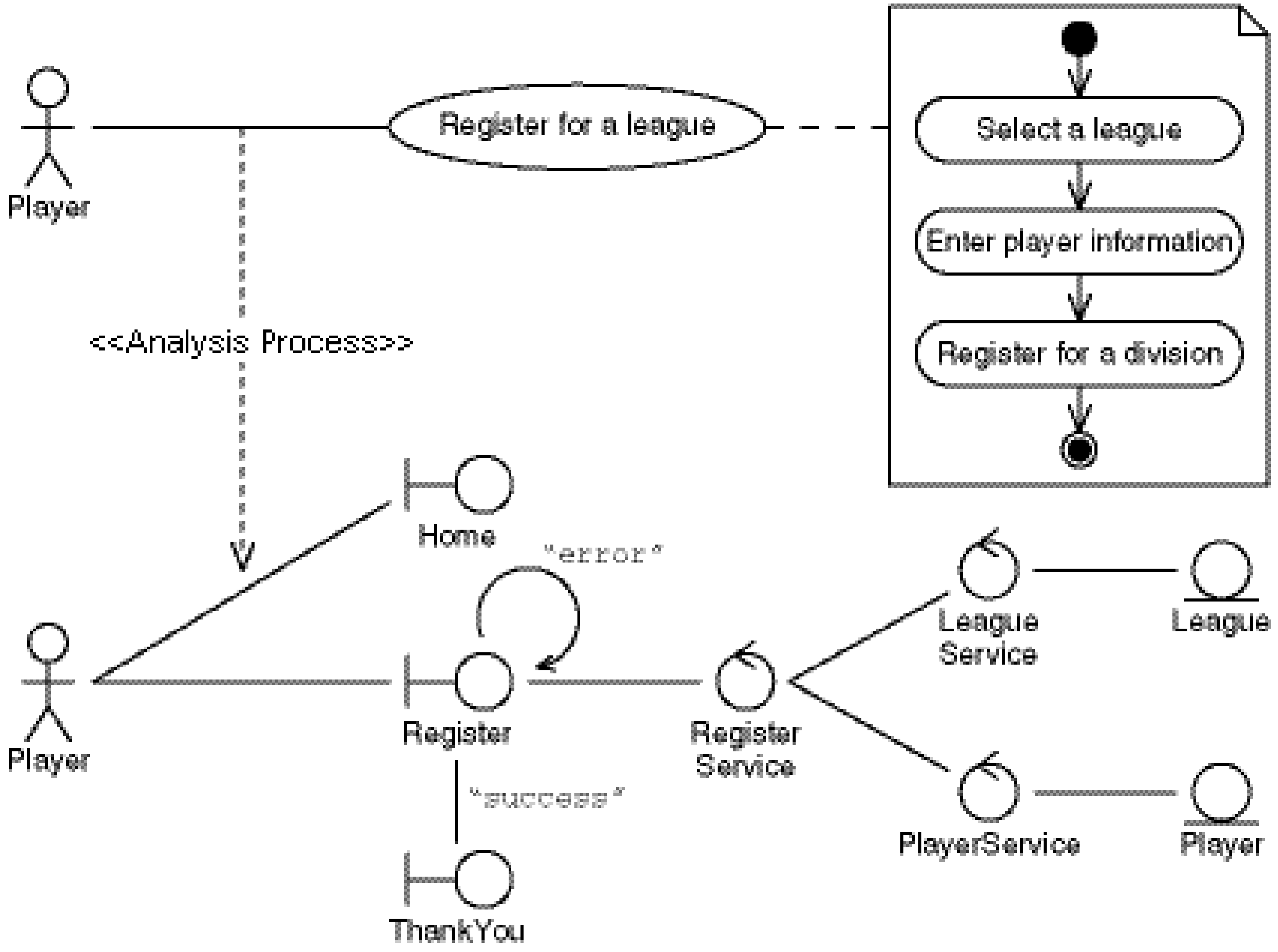
Describing the Analysis Model

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

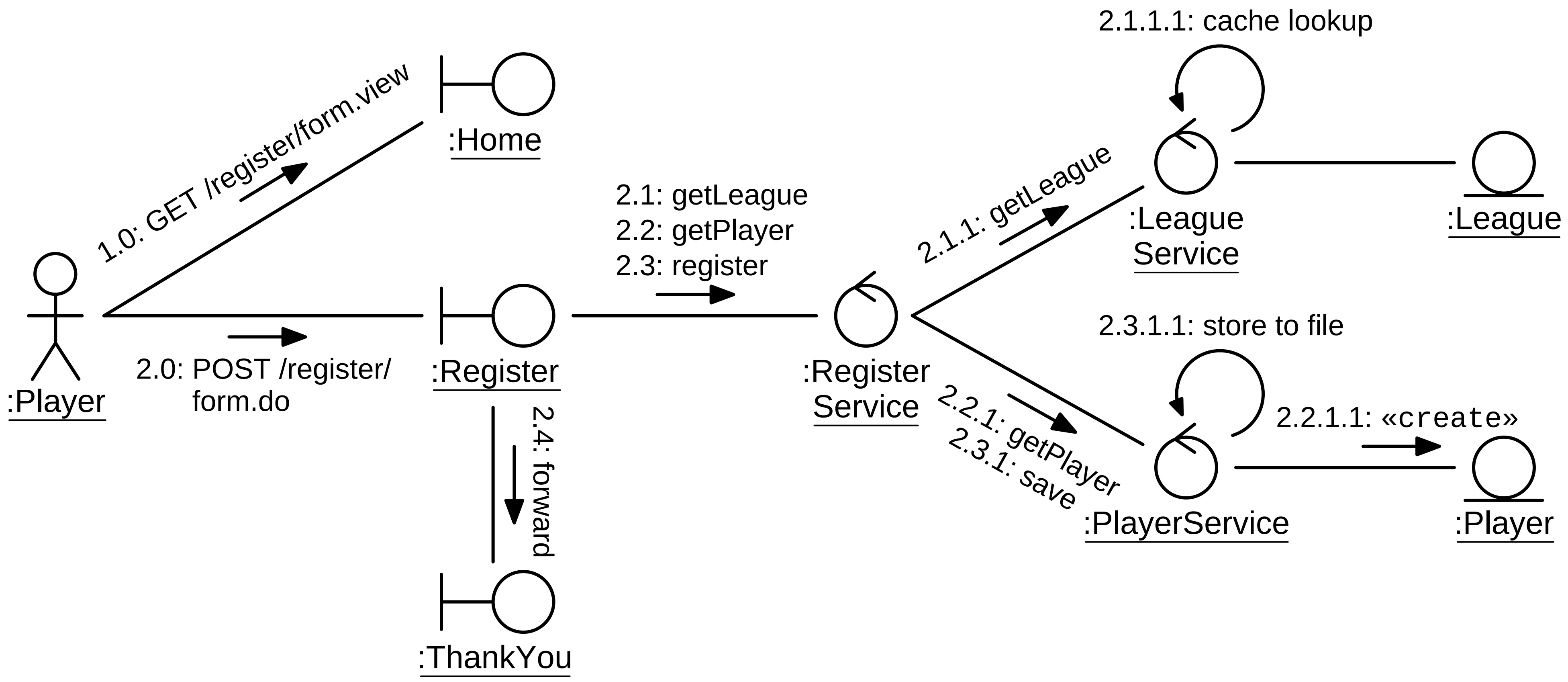
An Analysis model consists of three abstract component types:

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.

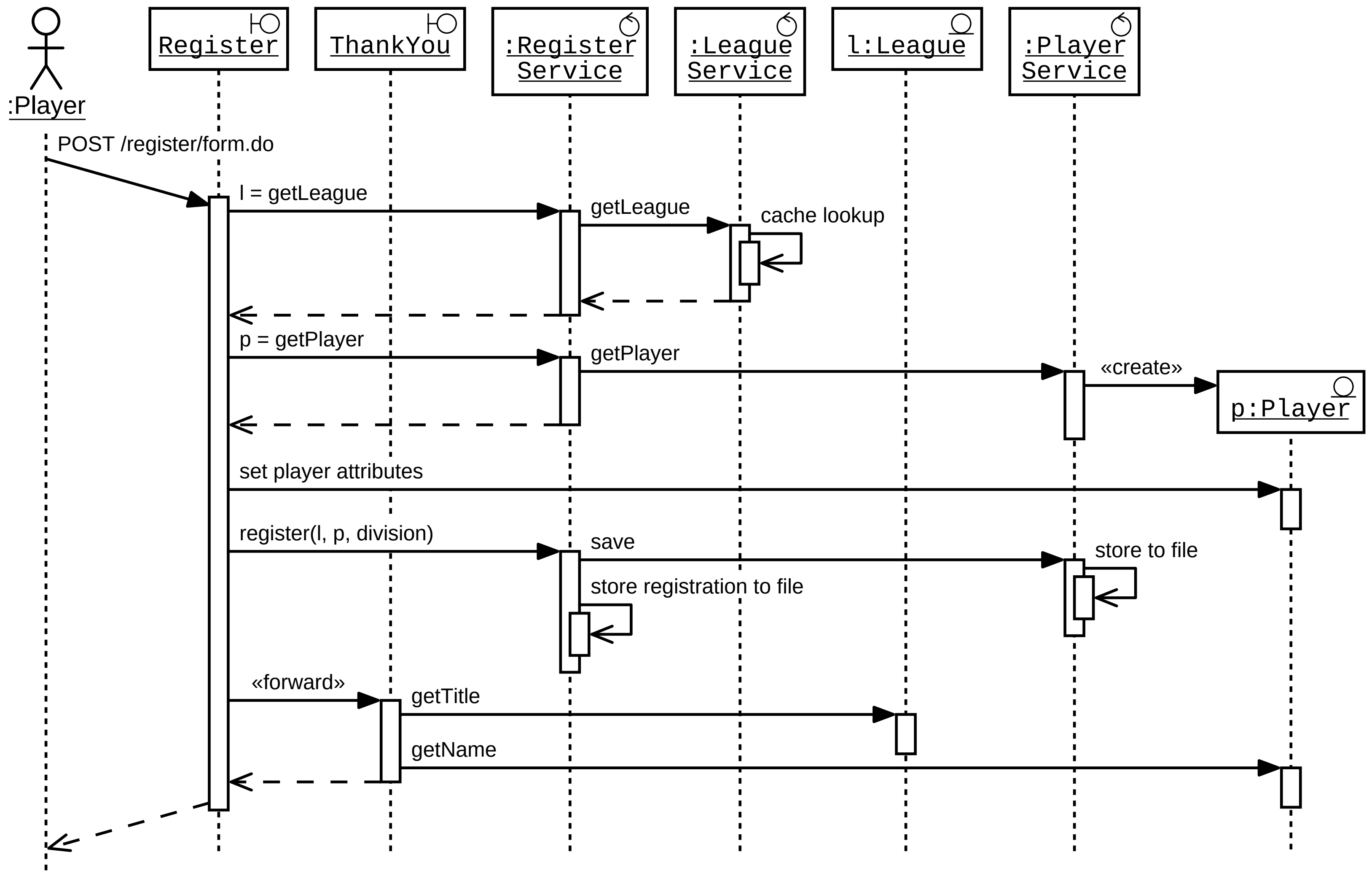
Registration Use Case Analysis Process



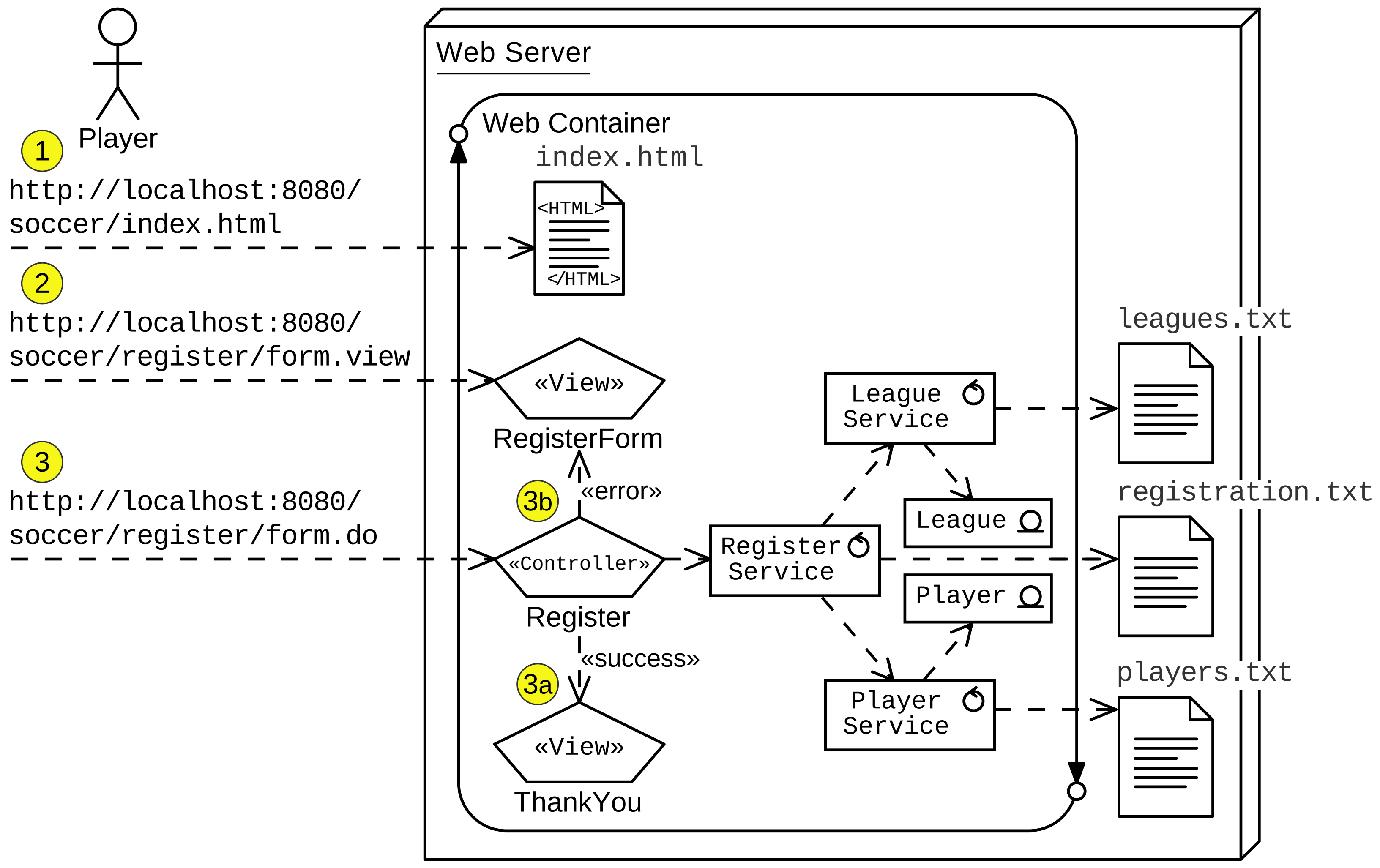
Detailed Analysis Model



Another View: UML Sequence Diagram



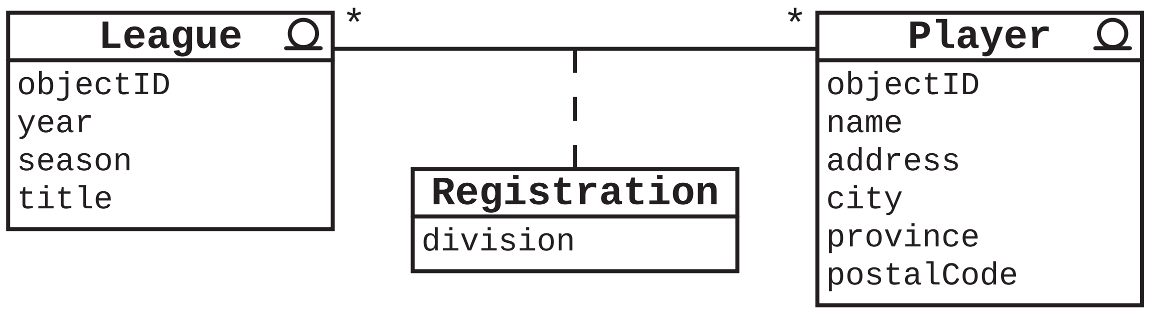
Another View: UML Deployment Diagram



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

```
package sl314.model;

/**
 * This domain object represents a player in a soccer league.
 */
public class Player {

    String name = "";
    String address = "";
    String city = "";
    String province = "";
    String postalCode = "";

    /**
     * This is the constructor.  It is package-private to prevent misuse.
     * The PlayerService.getPlayer method should be used to create a
     * new player object.
     */
    Player(String name) {
        this(name, "", "", "", "");
    }
```

The Player Code (Part 2)

```
    /**
     * This is the constructor.  It is package-private to prevent misuse.
     * The PlayerService.getPlayer method should be used to create a
     * new player object.
     */
    Player(String name) {
        this(name, "", "", "", "");
    }

    /**
     * This is the full constructor.
     */
    Player(String name, String address, String city,
            String province, String postalCode) {
        this.name = name;
        this.address = address;
        this.city = city;
        this.province = province;
        this.postalCode = postalCode;
    }

```

The `Player` Code (Part 3)

```
    public String getName() {
        return name;
    }
    public void setName(String value) {
        name = value;
    }
    public String getAddress() {
        return address;
    }
    public void setAddress(String value) {
        address = value;
    }
    public String getCity() {
        return city;
    }
    public void setCity(String value) {
        city = value;
    }
    public String getProvince() {
        return province;
    }
```

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

```
package sl314.model;

import java.util.List;
import java.util.LinkedList;
import java.util.Iterator;
import java.util.Collections;
import java.io.File;
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.FileWriter;
import java.io.PrintWriter;
import java.io.IOException;

/**
 * This object performs a variety of league services, such as looking
 * up league objects and creating new ones.
 */
public class LeagueService {

```

The LeagueService Code (Part 2)

```java
public class LeagueService {

    /** The cache of League objects. */
    private static final List LEAGUES_CACHE = new LinkedList();
    private String dataDirectory;

    public LeagueService(String dataDirectory) {
        this.dataDirectory = dataDirectory;

        // Make sure that the leagues cache has been initialized
        synchronized ( LEAGUES_CACHE ) {
            if ( LEAGUES_CACHE.isEmpty() ) {
                cacheLeagues();
            }
        }
    }

    /**
     * This method returns a complete set of leagues.
     */
    public List getAllLeagues() {
        // Return an immutable List; which makes this read-only
```

The LeagueService Code (Part 3)

```
        // Return an immutable List; which makes this read-only
        return Collections.unmodifiableList(LEAGUES_CACHE);
    }

    /**
     * This method finds the specified League object from the
     * complete set of leagues.
     */
    public League getLeague(int year, String season)
    throws ObjectNotFoundException {

        // Search in the cache.
        Iterator set = LEAGUES_CACHE.iterator();
        while ( set.hasNext() ) {
            League l = (League) set.next();
            if ( season.equals(l.getSeason()) && (year == l.getYear()) ) {
                return l;
            }
        }

        // Throw an exception if the league was not found.
        throw new ObjectNotFoundException();
    }
```

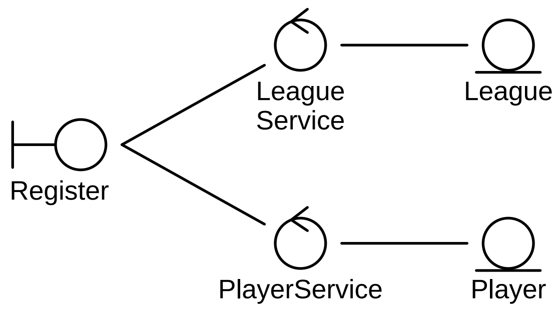
The LeagueService Code (Part 4)

```

    /**
     * This method adds a new League object.
     */
    public League createLeague(int year, String season, String title) {

        // Determine the next league objectID
        int nextID = LEAGUES_CACHE.size() + 1;

        // Create new league object
        League league = new League(nextID, year, season, title);

        // Store the league object
        storeLeague(league);

        // Record the league in the cache for easy retrieval
        LEAGUES_CACHE.add(league);

        return league;
    }


```

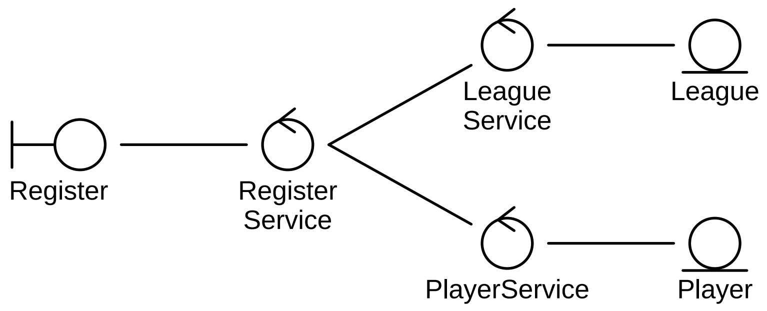
Façade Service

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

High Coupling



Low Coupling



The RegisterService Code

```
package sl314.model;

import java.io.File;
import java.io.FileWriter;
import java.io.PrintWriter;
import java.io.IOException;

/**
 * This object performs a variety of league registration services.
 * It acts a Facade into the business logic of registering a Player for
 * a League.
 */
public class RegisterService {
    private String dataDirectory;

    public RegisterService(String dataDirectory) {
        this.dataDirectory = dataDirectory;
        // do nothing
    }

```

The RegisterService Code (Part 2)

```

    /**
     * This method finds the specified league, by delegating to the
     * LeagueService object.
     */
    public League getLeague(int year, String season)
    throws ObjectNotFoundException {
        LeagueService leagueSvc = new LeagueService(dataDirectory);
        return leagueSvc.getLeague(year, season);
    }

    /**
     * This method return a Player object for the named person, by
     * delegating to the PlayerService object.
     */
    public Player getPlayer(String name) {
        PlayerService playerSvc = new PlayerService(dataDirectory);
        return playerSvc.getPlayer(name);
    }

```

The RegisterService Code (Part 3)

```
    /**
     * This method stores the registration information for the player,
     * based on the league and division information.
     */
    public void register(League league, Player player, String division) {

        // Use the player service to save the player object
        PlayerService playerSvc = new PlayerService(dataDirectory);
        playerSvc.save(player);

        // Record the registration
        insertRegistration(league, player, division);
    }

```

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

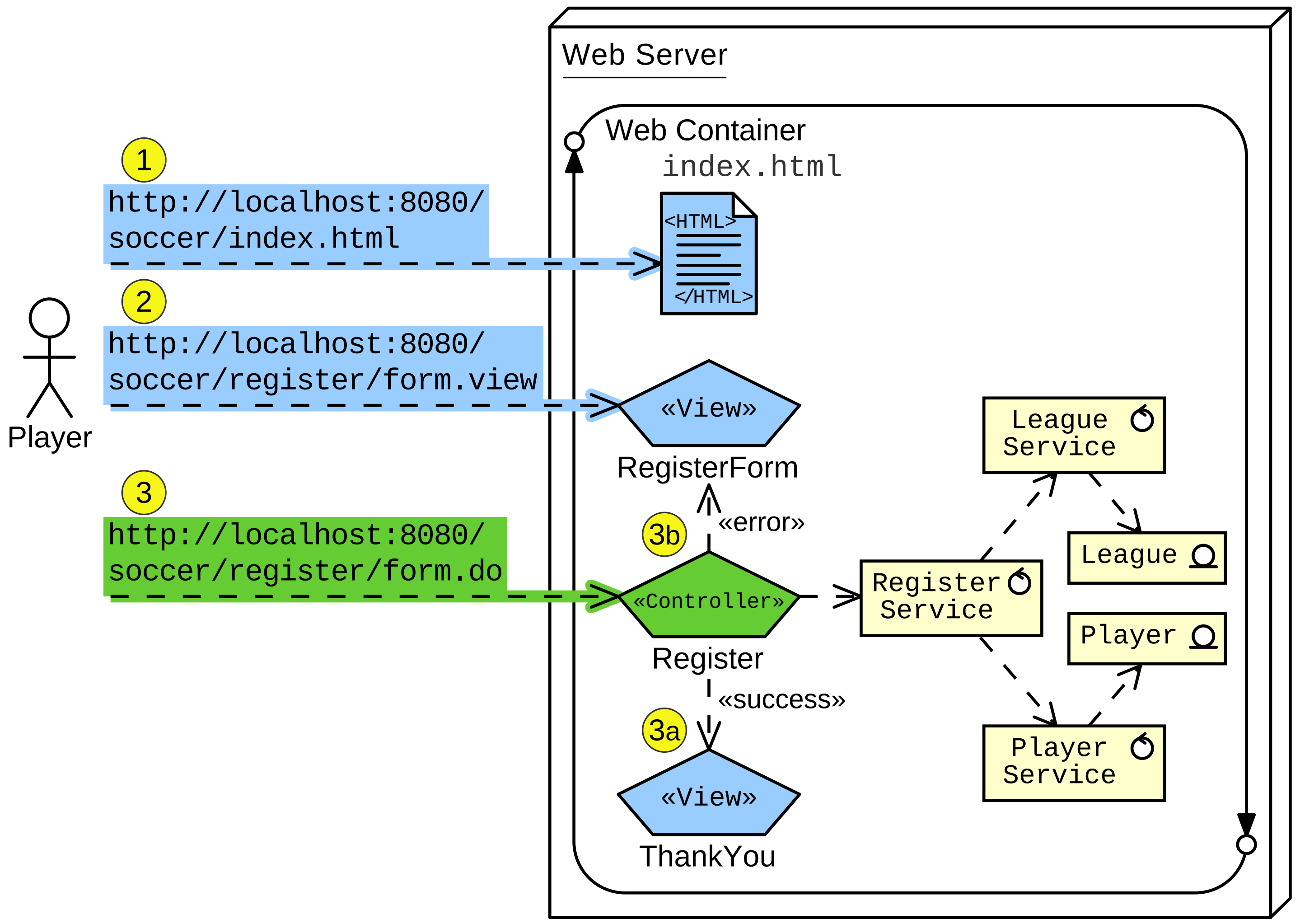
Objectives

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

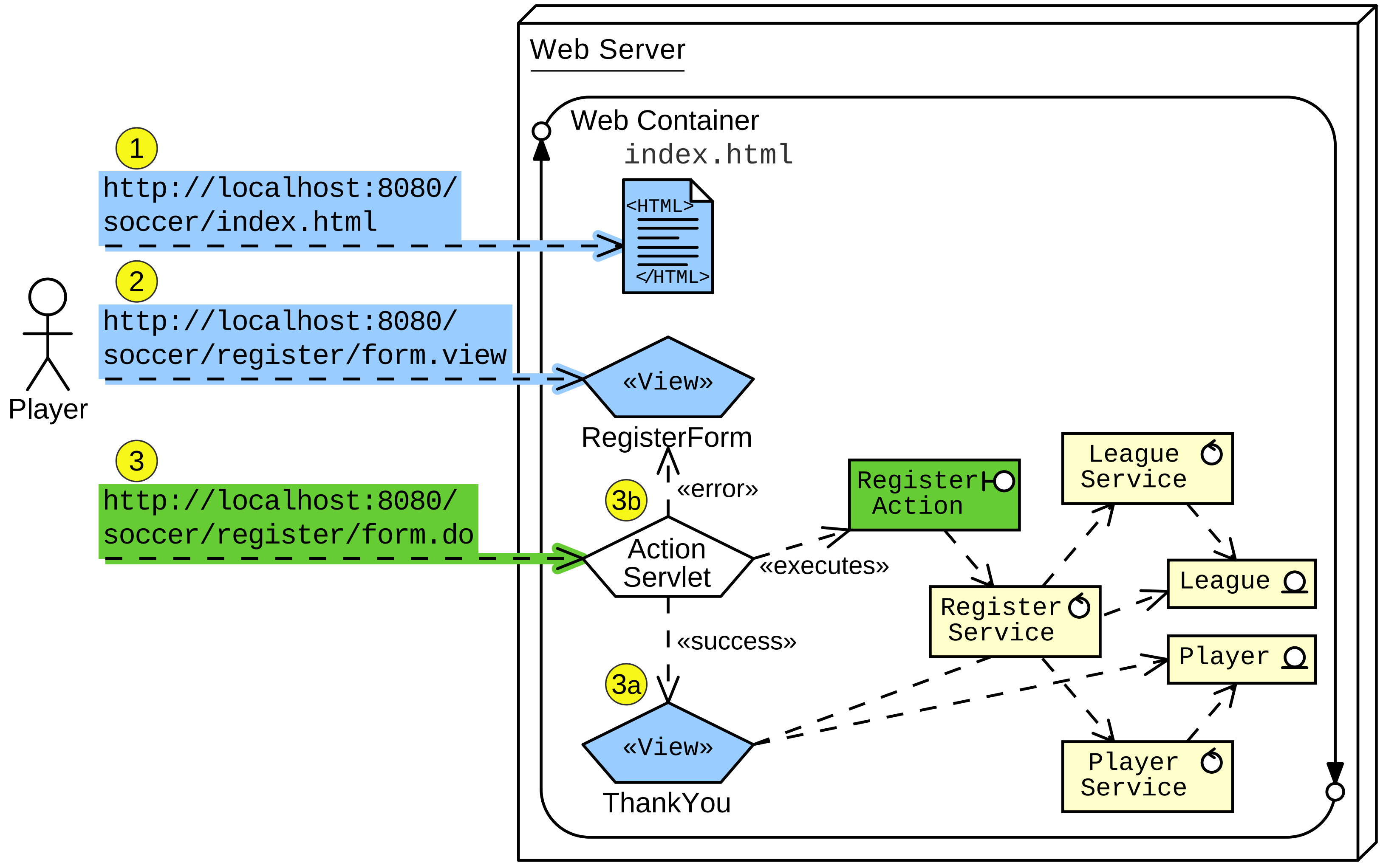
Relevance

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

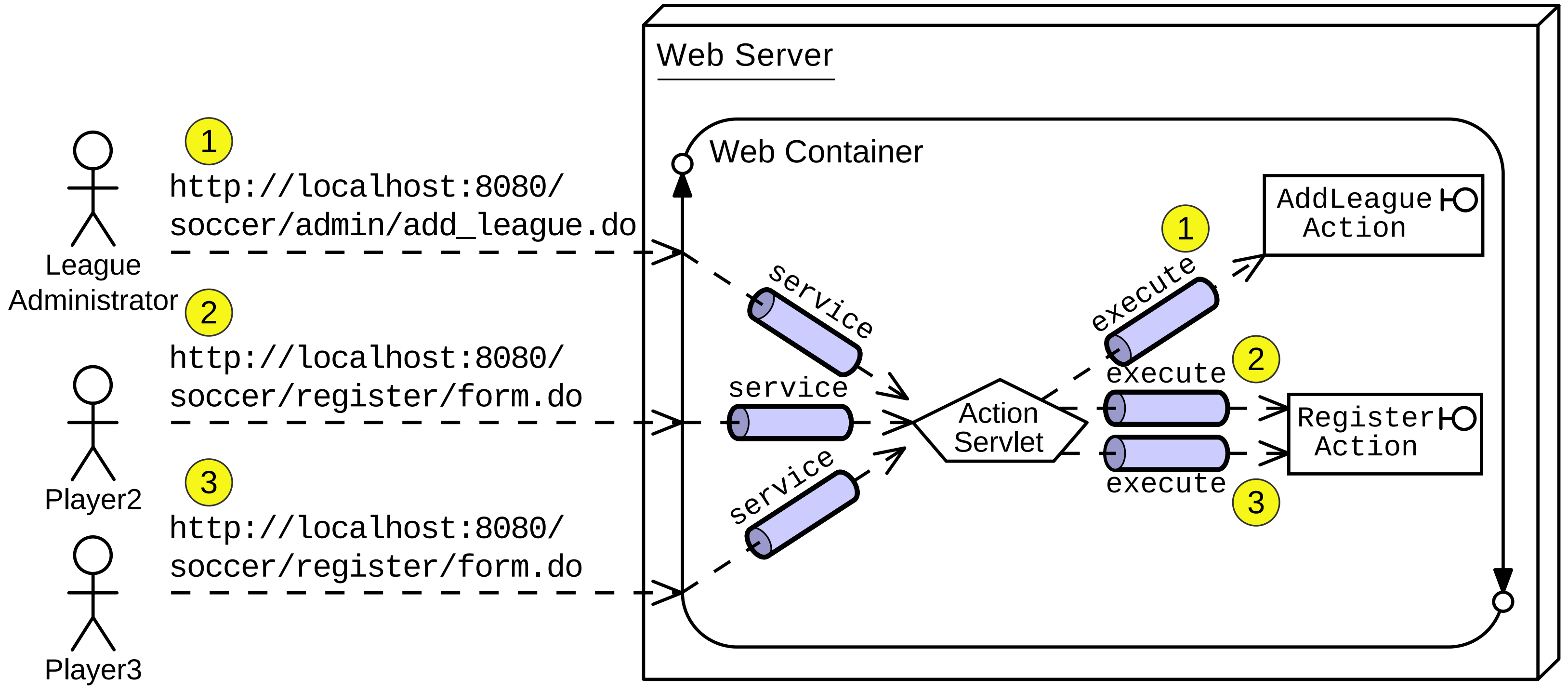
Model-View-Controller Pattern



Struts MVC Framework



Front Controller Pattern

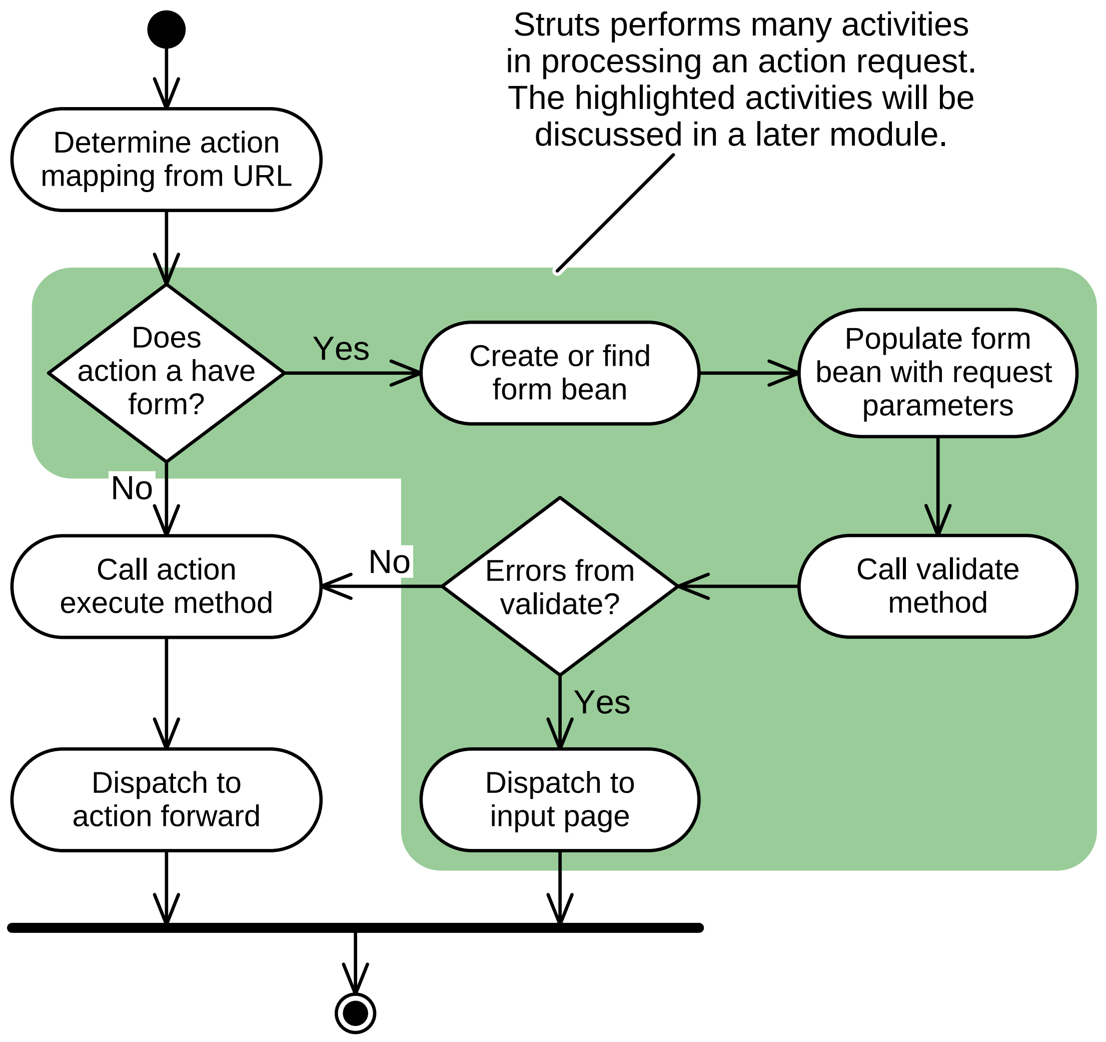


Controller requests are handled by the Struts `ActionServlet`, which acts as an infrastructure controller to dispatch to the application controller actions.

Struts MVC Framework

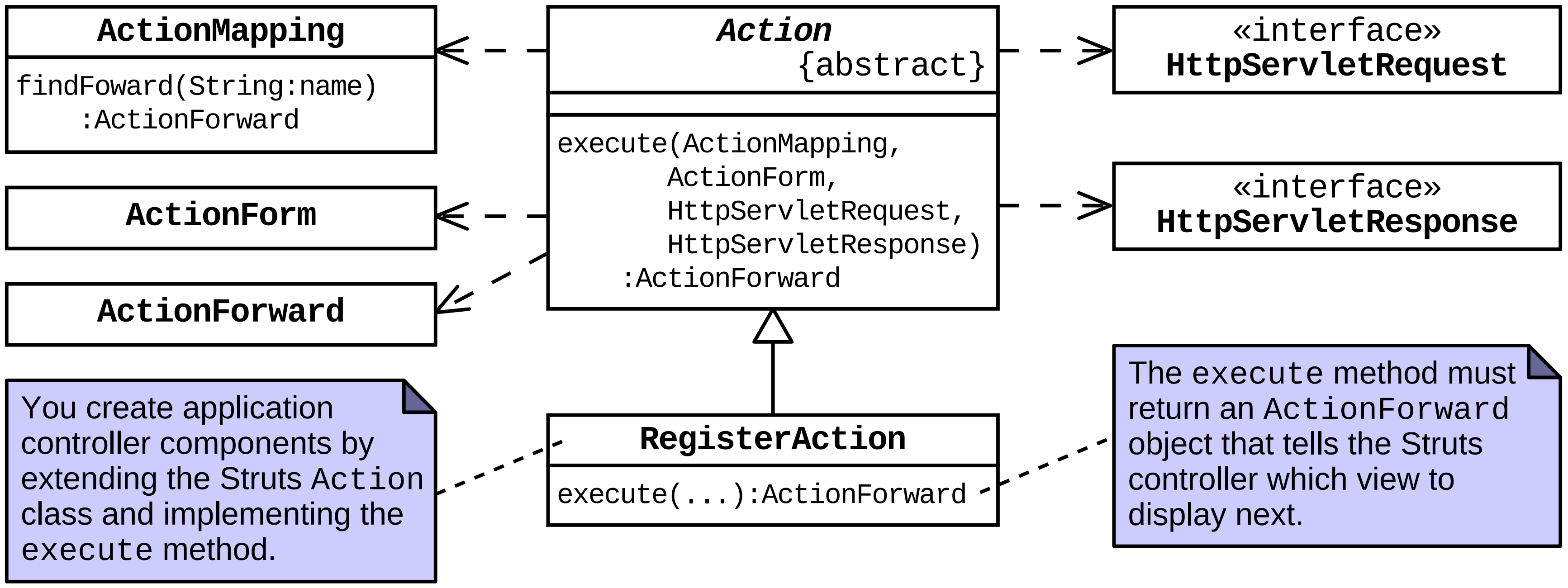
- Framework provides the following elements:
  - Infrastructure servlet controller
  - Base classes
  - Configuration files
- Why use a framework like Struts?
  - Provides flexible, extensible infrastructure for MVC
  - Lets you focus on what is important to your application, such as:
    - Application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

Struts Activity Diagram



Struts performs many activities in processing an action request. The highlighted activities will be discussed in a later module.
Determine action mapping from URL
Does action a have form?
Yes
Create or find form bean
Populate form bean with request parameters
No
Call action execute method
No
Errors from validate?
Call validate method
Yes
Dispatch to action forward
Dispatch to input page

Struts Action Class



You create application controller components by extending the Struts `Action` class and implementing the `execute` method.

The `execute` method must return an `ActionForward` object that tells the Struts controller which view to display next.

The AddLeagueAction Code

```
package sl314.controller;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
// Struts classes
import org.apache.struts.action.Action;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.action.ActionForm;
// Model classes
import sl314.model.LeagueService;
import sl314.model.League;
import java.util.List;
import java.util.LinkedList;
import javax.servlet.ServletContext;


public class AddLeagueAction extends Action {

```

The AddLeagueAction Code (Part 2)

```
public class AddLeagueAction extends Action {

    public ActionForward execute(ActionMapping mapping,
            ActionForm form,
            HttpServletRequest request,
            HttpServletResponse response) {

        // Keep a set of strings to record form processing errors.
        List errorMsgs = new LinkedList();
        // Store this set in the request scope, in case we need to
        // send the ErrorPage view.
        request.setAttribute(“errorMsgs”, errorMsgs);

        try {

            // Retrieve form parameters.
            String yearStr = request.getParameter(“year”).trim();
            String season = request.getParameter(“season”).trim();
            String title = request.getParameter(“title”).trim();

```

The AddLeagueAction Code (Part 3)

```
// Perform data conversions.
int year = -1;
try {
    year = Integer.parseInt(yearStr);
} catch (NumberFormatException nfe) {
    errorMsgs.add(“The ‘year’ field must be a positive integer.”);
}

// Verify form parameters
if ( (year != -1) && ((year < 2000) || (year > 2010)) ) {
errorMsgs.add(“The ‘year’ field must within 2000 to 2010.”)
}
if ( season.equals(“UNKNOWN”) ) {
    errorMsgs.add(“Please select a league season.”);
}
if ( title.length() == 0 ) {
    errorMsgs.add(“Please enter the title of the league.”)
}

// Send the ErrorPage view if there were errors
if ( ! errorMsgs.isEmpty() ) {
    return mapping.findForward(“error”);
```

The AddLeagueAction Code (Part 4)

```
                }

                // Perform business logic
                // Perform business logic
                ServletContext context = getServlet().getServletContext();
                String dataDirectory =
(String)context.getAttribute(“sl314.model.dataDirectory”);
               LeagueService leagueSvc = new LeagueService(dataDirectory);
              League league = leagueSvc.createLeague(year, season, title);
                // Store the new league in the request-scope
                request.setAttribute(“league”, league);

                // Send the Success view
                return mapping.findForward(“success”);

                // Handle any unusual exceptions
            } catch (RuntimeException e) {

                // Log stack trace
                e.printStackTrace(System.err);

```

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:

```
<!-- Declare the Struts ActionServlet (Front Controller) -->
<servlet>
  <servlet-name>FrontController</servlet-name>
  <servlet-class>
    org.apache.struts.action.ActionServlet
  </servlet-class>
  <!-- Path of the struts configuration file -->
  <init-param>
    <param-name>config</param-name>
    <param-value>/WEB-INF/struts-config.xml</param-value>
  </init-param>
  <!-- Load the servlet on startup -->
  <load-on-startup>1</load-on-startup>
</servlet>
```

Front Controller Servlet Mapping

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

```

  <!-- Standard Front Controller Mapping -->
  <servlet-mapping>
    <servlet-name>FrontController</servlet-name>
    <url-pattern>*.do</url-pattern>
  </servlet-mapping>

```

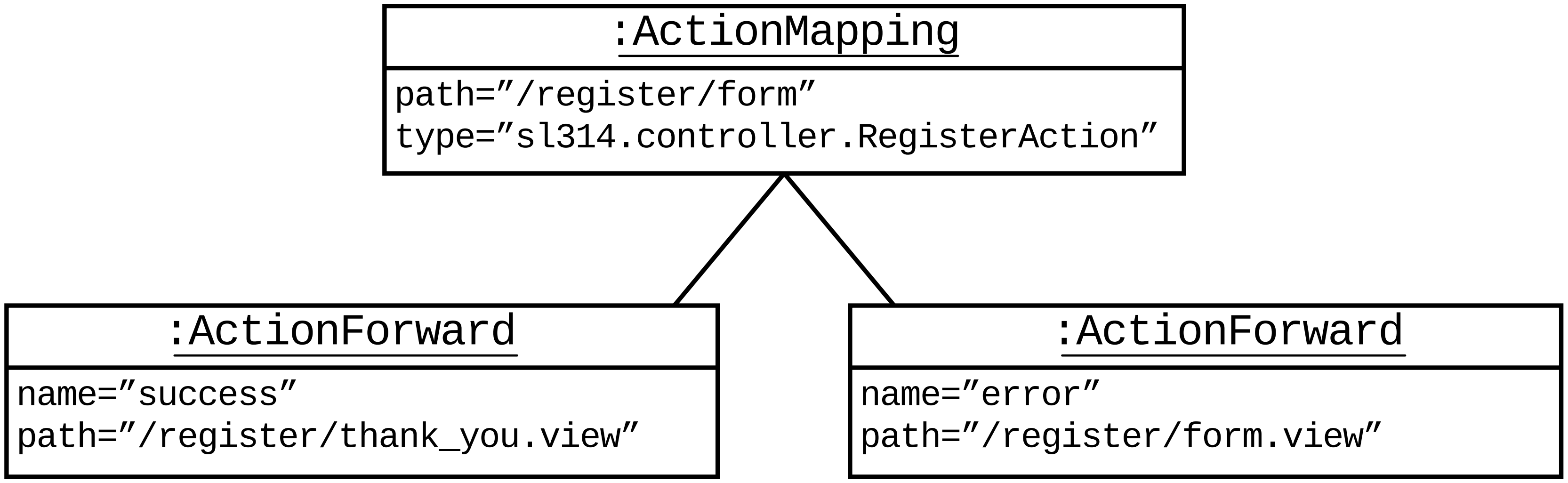
This servlet mapping ensures that all `*.do` requests go to the Struts infrastructure controller.

Configuring Action Mappings

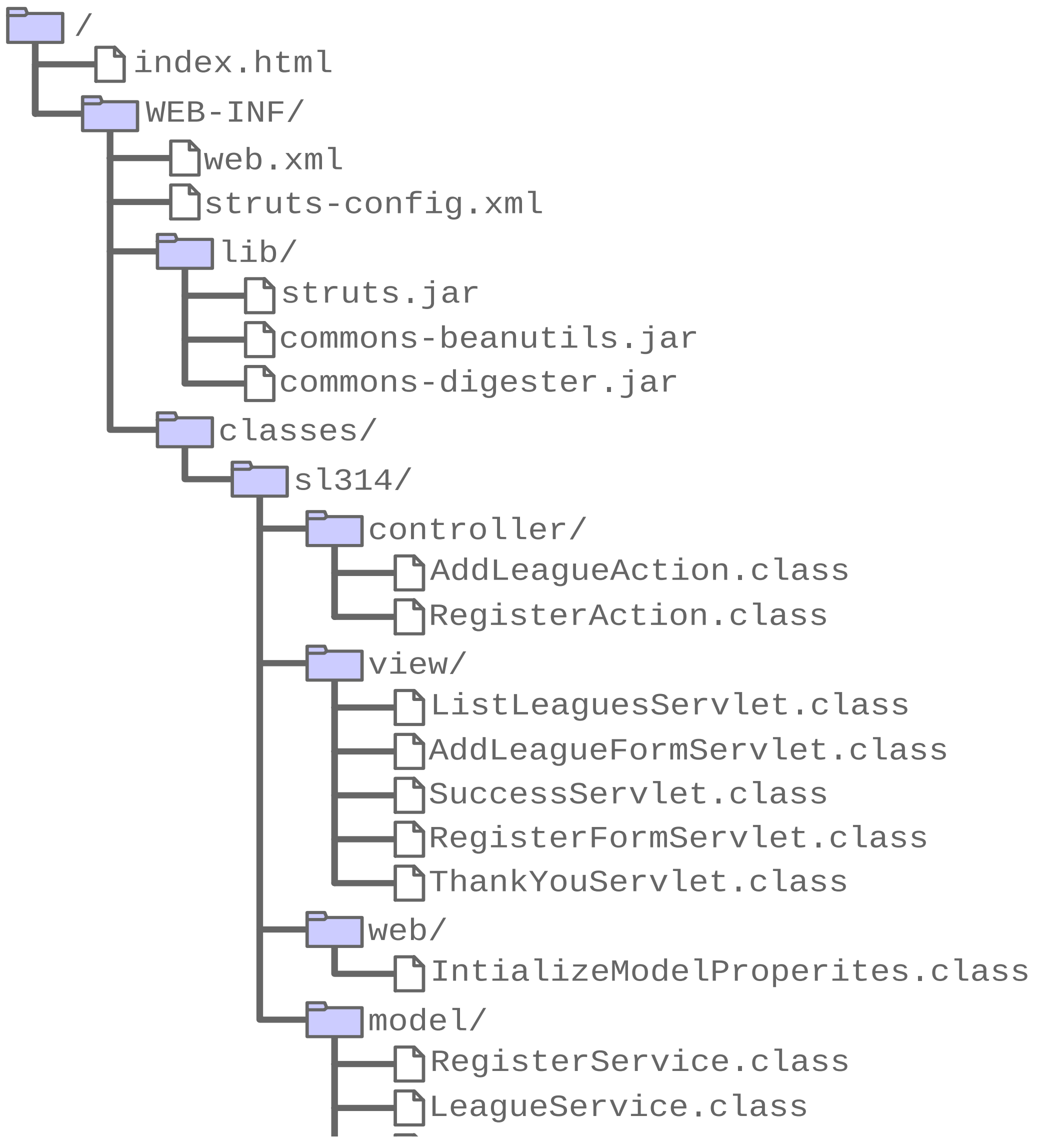
Configured in the `struts-config.xml` file:

```

    <action-mappings>

      <!-- Declare the /register/form.do action -->
      <action path="/register/form"
              type="sl314.controller.RegisterAction">
        <forward name="success" path="/register/thank_you.view"/>
        <forward name="error" path="/register/form.view"/>
      </action>

      <!-- Declare the /admin/add_league.do action -->
      <action path="/admin/add_league"
              type="sl314.controller.AddLeagueAction">
        <forward name="success" path="/admin/success.view"/>
        <forward name="error" path="/admin/add_league.view"/>
      </action>

    </action-mappings>

  </struts-config>
```

Action Mapping Object Representation


:ActionMapping
path=”/register/form”
type=”sl314.controller.RegisterAction”
:ActionForward
name=”success”
path=”/register/thank_you.view”
:ActionForward
name=”error”
path=”/register/form.view”

Installing the Struts Library Files



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

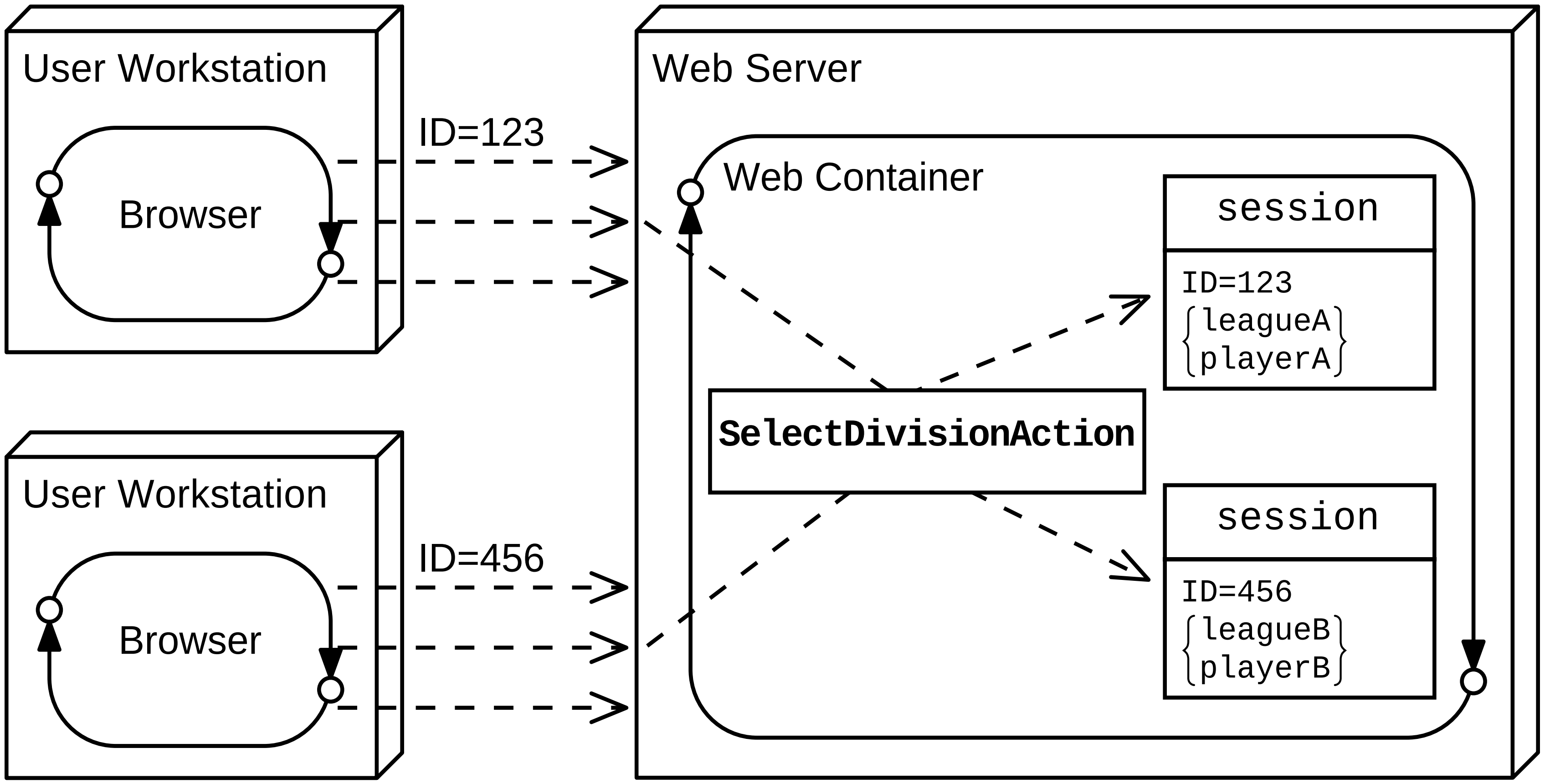
- What mechanism do you currently use for maintaining communications across requests?
- How much additional development is needed to use that communication mechanism?

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:



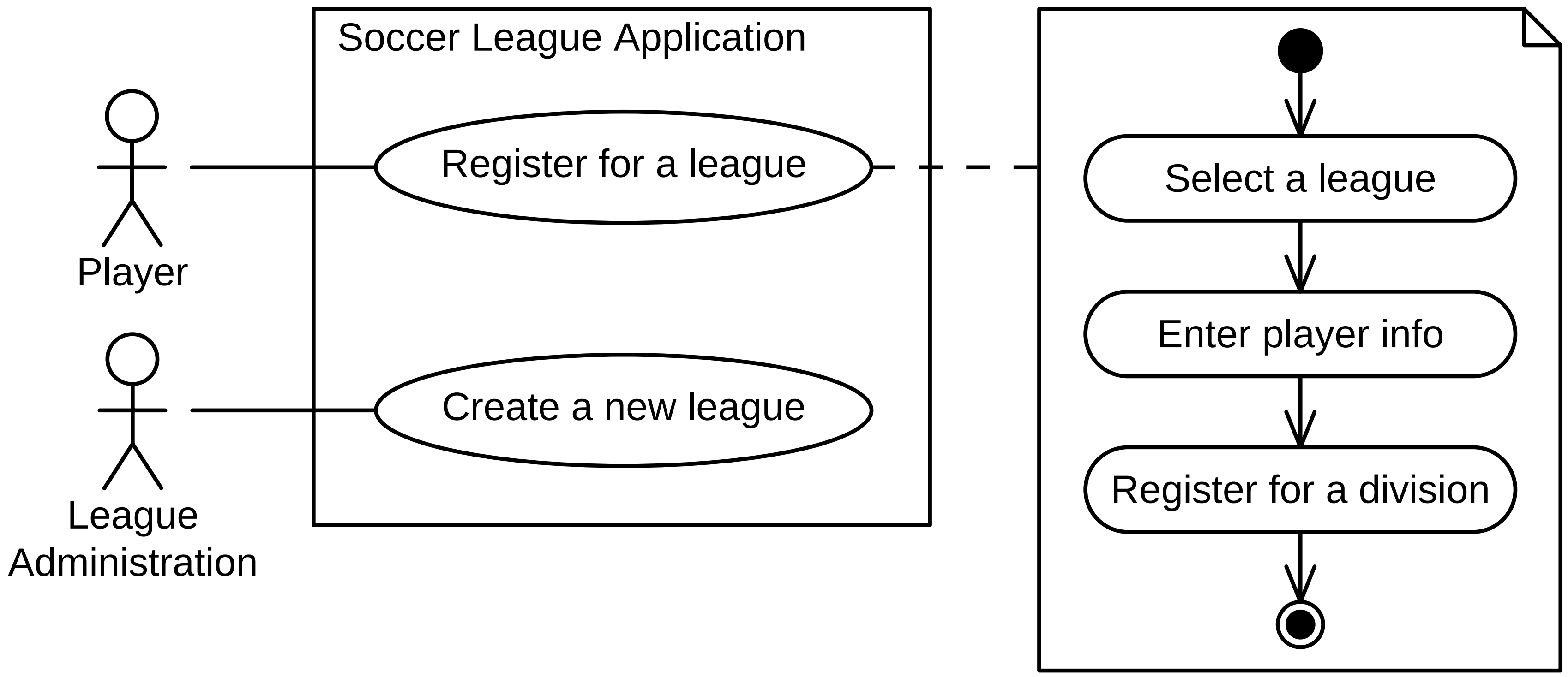
Designing Web Applications

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

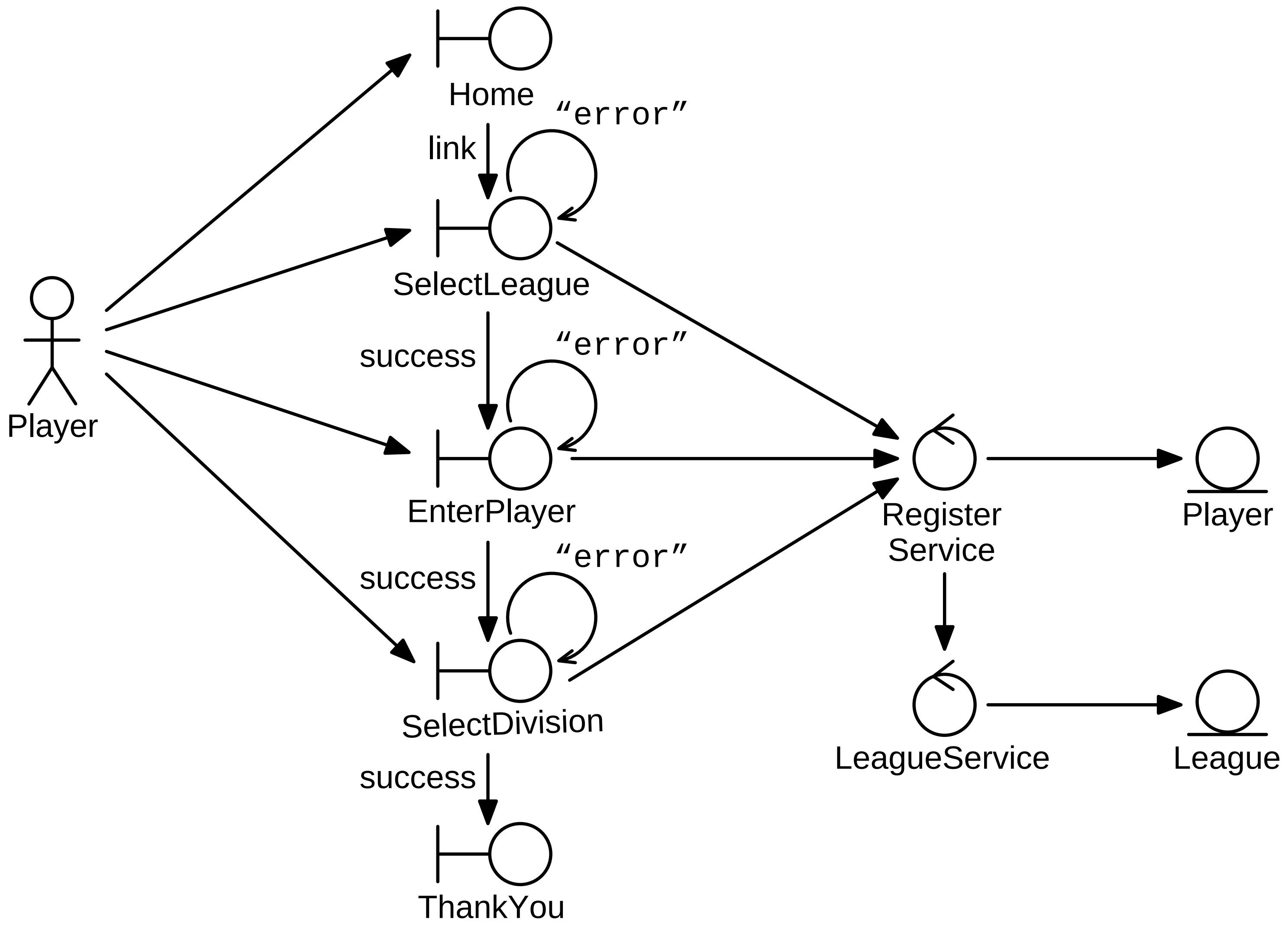
1. Design multiple, interacting views for a use case.
2. Create a Struts application controller for each activity in the use case.
3. Create a unique Struts URL for each activity in the use case.

Registration Use Case Example

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



Registration Use Case Analysis Model

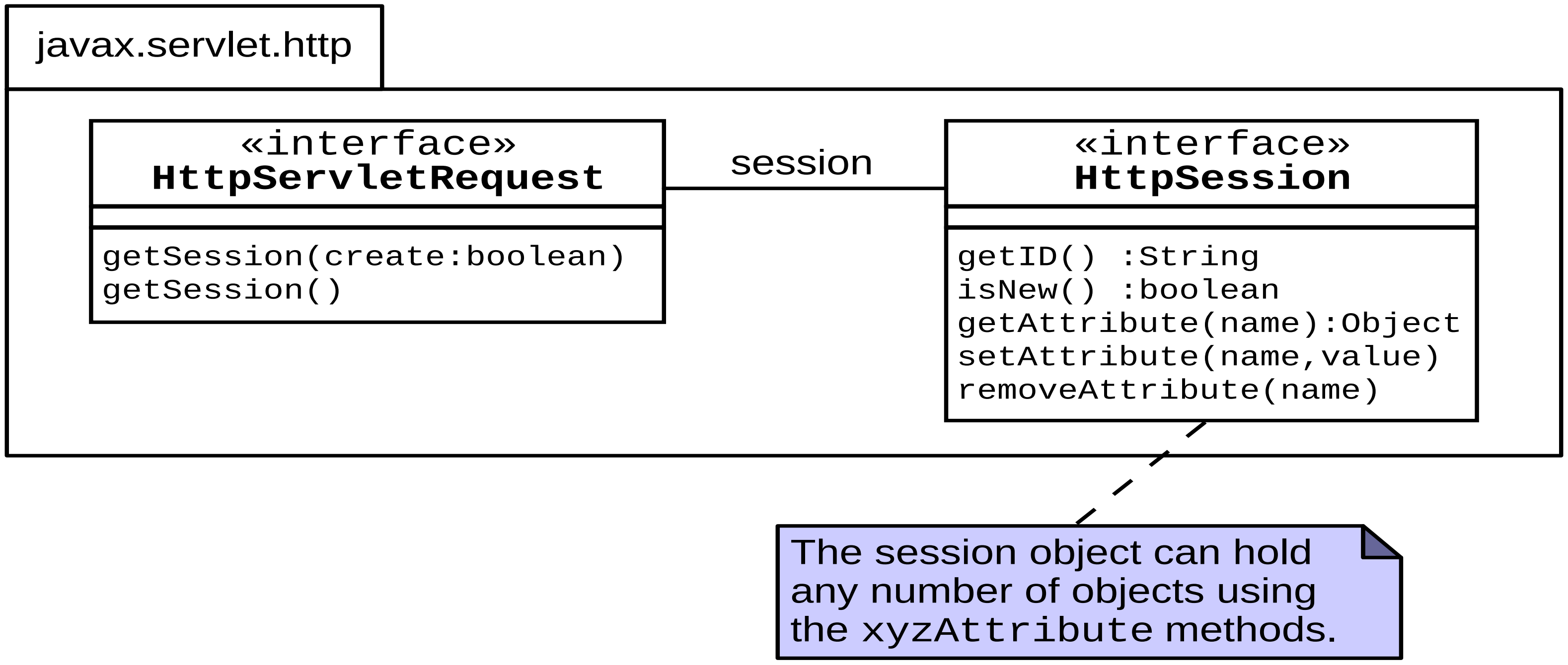


Using Session Management in a Web Application

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



- 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
59                // Perform business logic
60                ServletContext context = getServlet().getServletContext();
61                String dataDirectory =
(String)context.getAttribute(“sl314.model.dataDirectory”);
62                RegisterService registerSvc = new RegisterService(dataDire
63
64                // Retrieve the league
65                League league = registerSvc.getLeague(year, season);
66
67                // Store the league object in the session-scope
68                HttpSession session = request.getSession();
```

- 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:

```
// Retrieve the league and player objects from the session
HttpSession session = request.getSession();
League league = (League) session.getAttribute(“league”);
Player player = (Player) session.getAttribute(“player”);

ServletContext context = getServlet().getServletContext()
String dataDirectory =
(String)context.getAttribute(“sl314.model.dataDirectory”);
RegisterService registerSvc = new RegisterService(dataDirec
```

Accessing Session Attributes (continued)

Views (such as the ThankYou component) might also:

- Access session attributes:

```
// Retrieve the ‘league’ and ‘player’ from the session-scope
HttpSession session = request.getSession();
League league = (League) session.getAttribute(“league”);
Player player = (Player) session.getAttribute(“player”);

```

- Generate a dynamic response using the attributes:

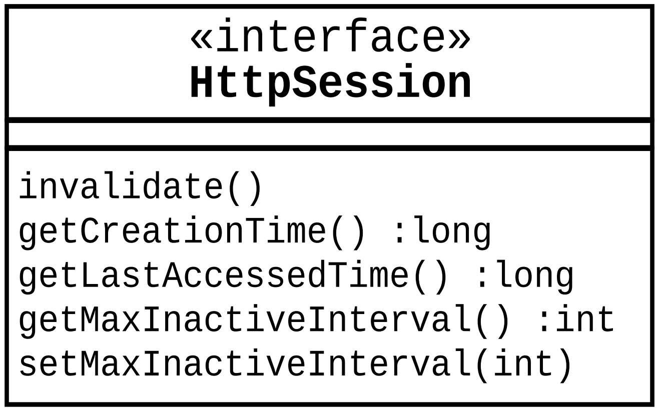
```
// Present the main body
out.println(“<p>”);
out.print(“Thank you, “ + player.getName() + “, for registering “);
out.println(“for the <i>” + league.getTitle() + “</i> league.”);
out.println(“</p>”);

```

Destroying the Session

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

```

</web-app>

```

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


«interface»
HttpSession
invalidate()
getCreationTime() :long
getLastAccessedTime() :long
getMaxInactiveInterval() :int
setMaxInactiveInterval(int)

Destroying the Session (continued)

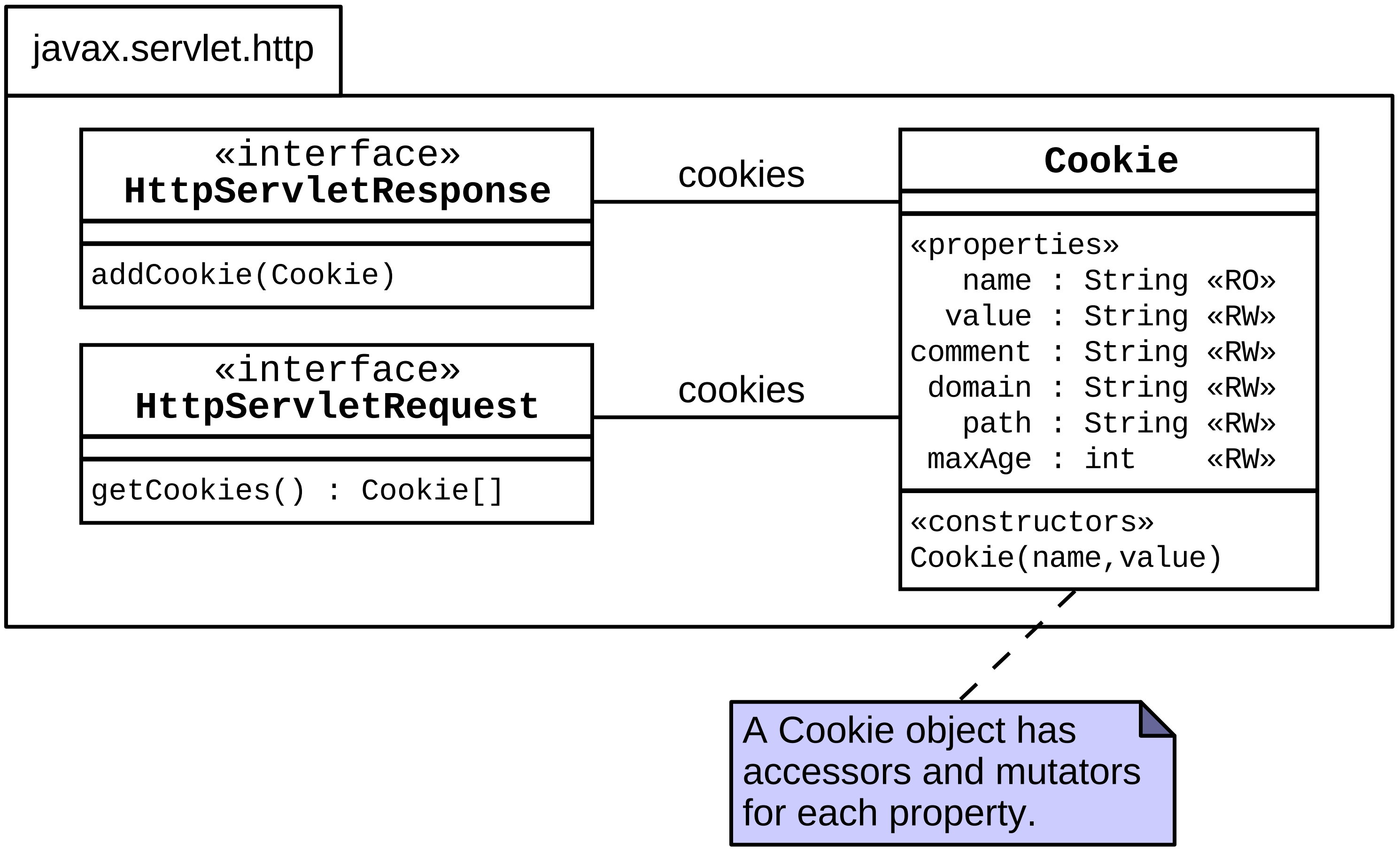
- Session objects can be shared across multiple actions (for different use cases) within the same web application.
- Session objects are not shared across multiple web applications within the same web container.
- Destroying a session using the `invalidate` method might cause disruption to other servlets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

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


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

Using Cookies Example

- The code to store a cookie in the response:

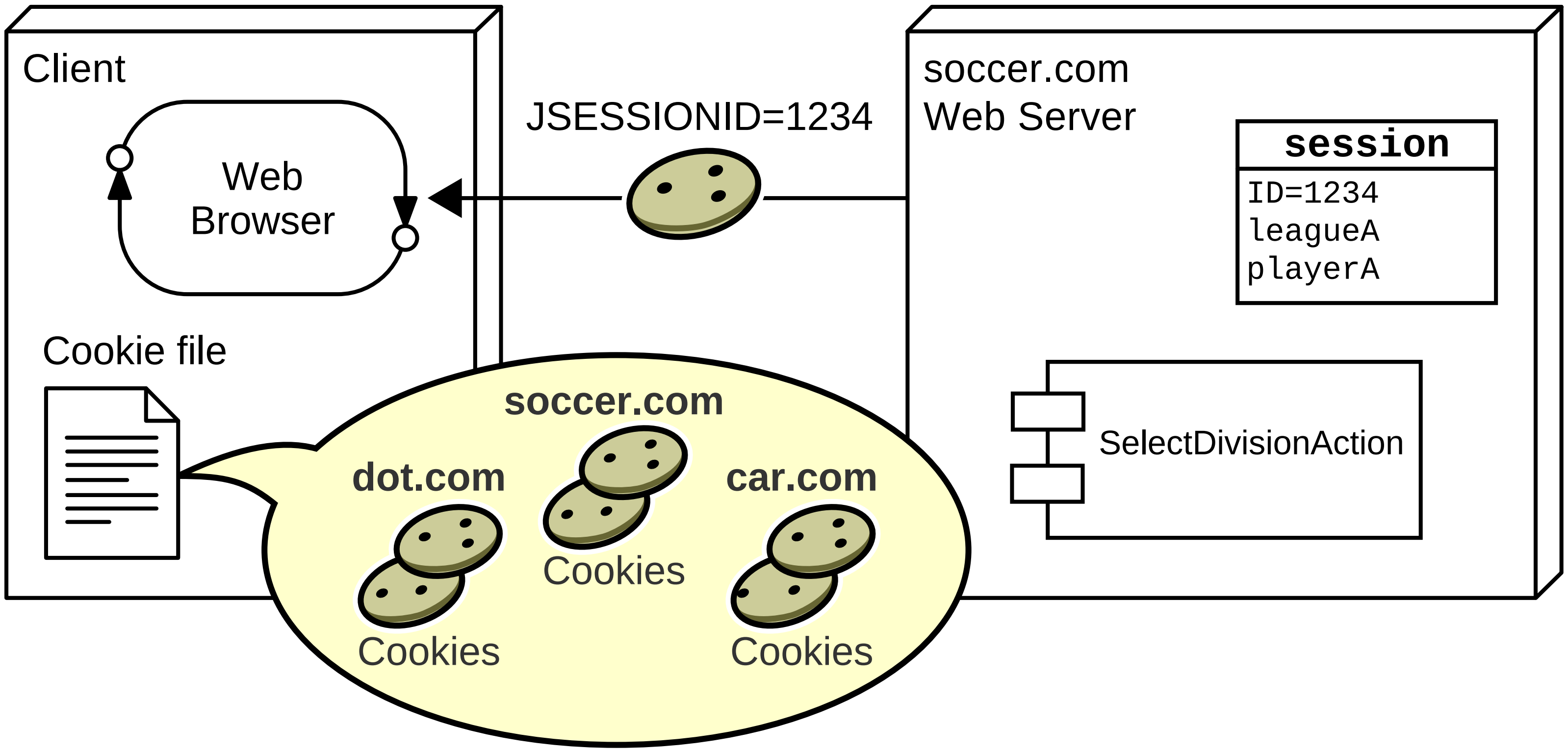
```
String name = request.getParameter("firstName");
Cookie c = new Cookie("yourname", name);
response.addCookie(c);
```

- 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();
  }
}
```

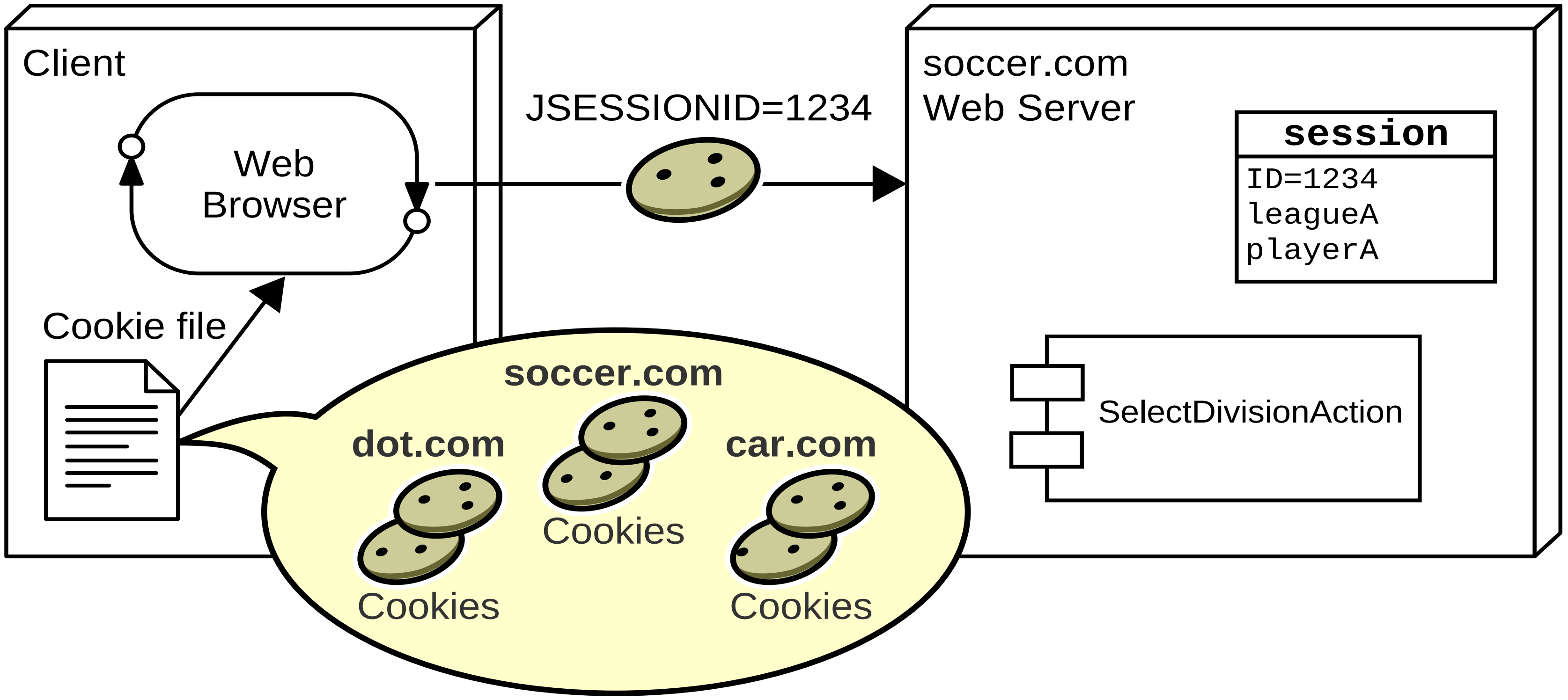
Performing Session Management Using Cookies

The web container sends a `JSESSIONID` cookie to the client:



Performing Session Management Using Cookies (continued)

The `JSESSIONID` cookie is sent in all subsequent requests:



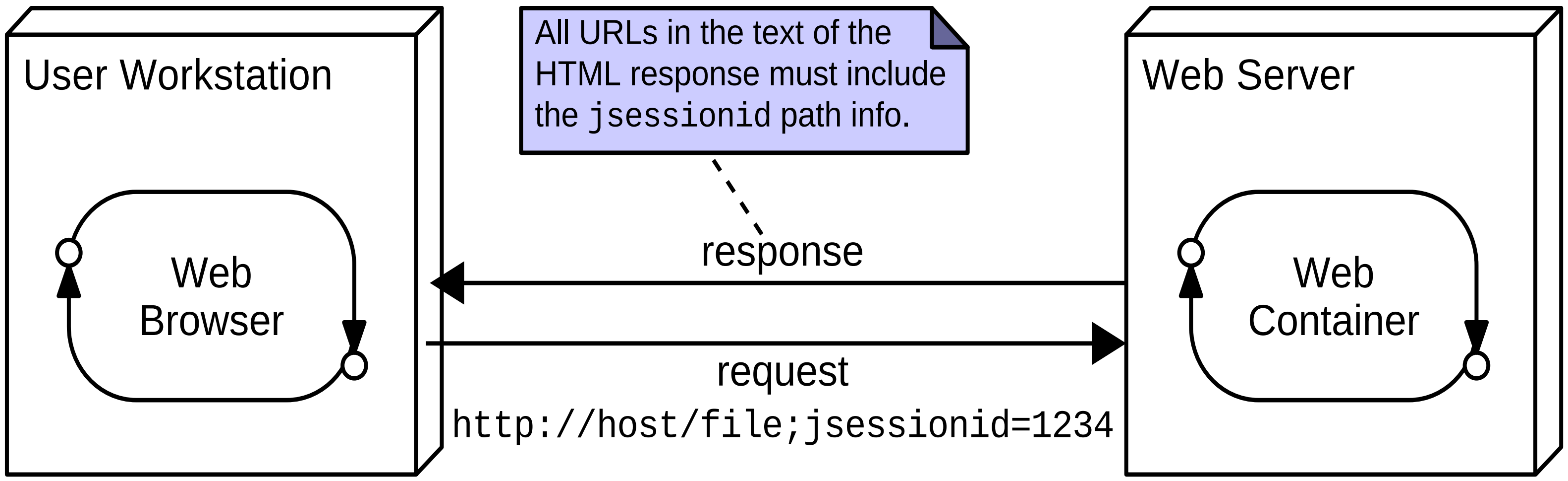
Performing Session Management Using Cookies (continued)

- 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.

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 (continued)


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

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:

```

    // Present the form
    out.println("<form action='"
                + response.encodeURL("enter_player.do")
                + "' method='POST'>");

```

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

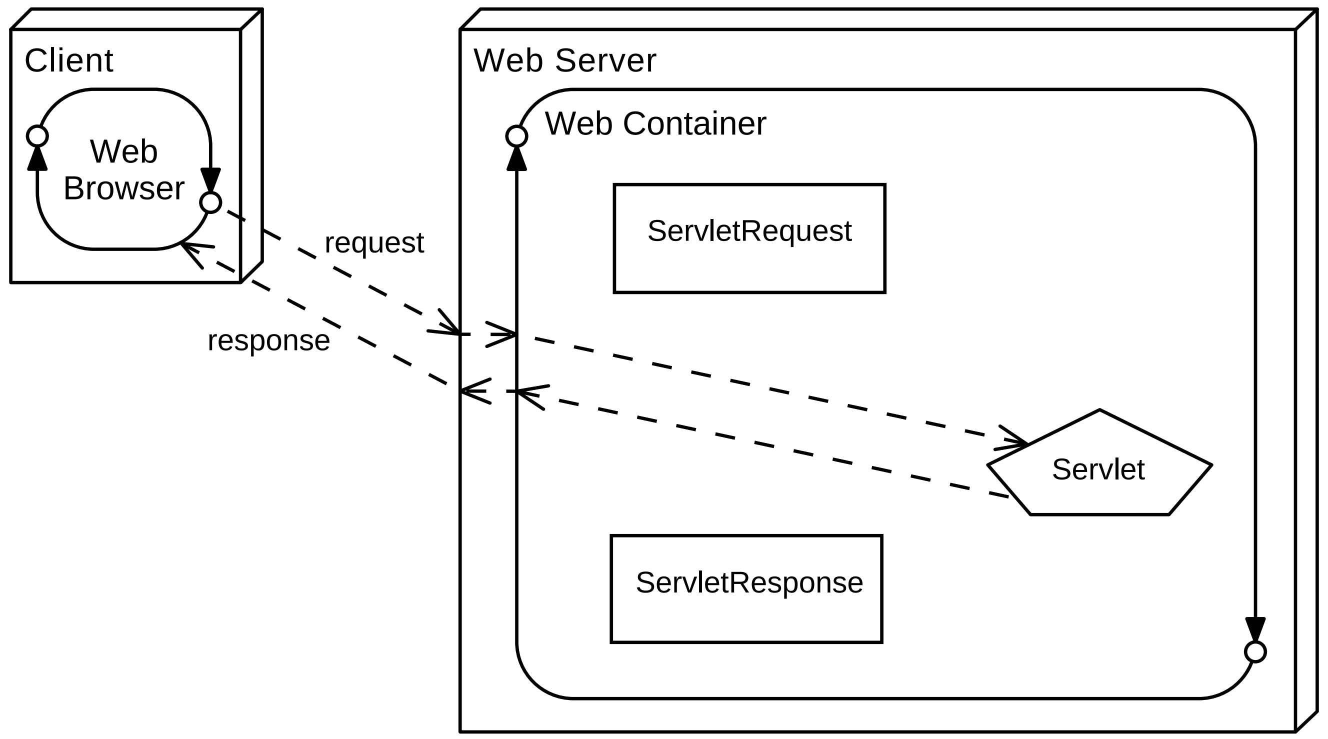
- What should you do if you want an operation to occur every time a particular request is made?
- What should you do if that operation must be performed on other requests in the web application?
- What should you do if you want to allow this operation to be turned off at deployment?

Web Container Request Cycle

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

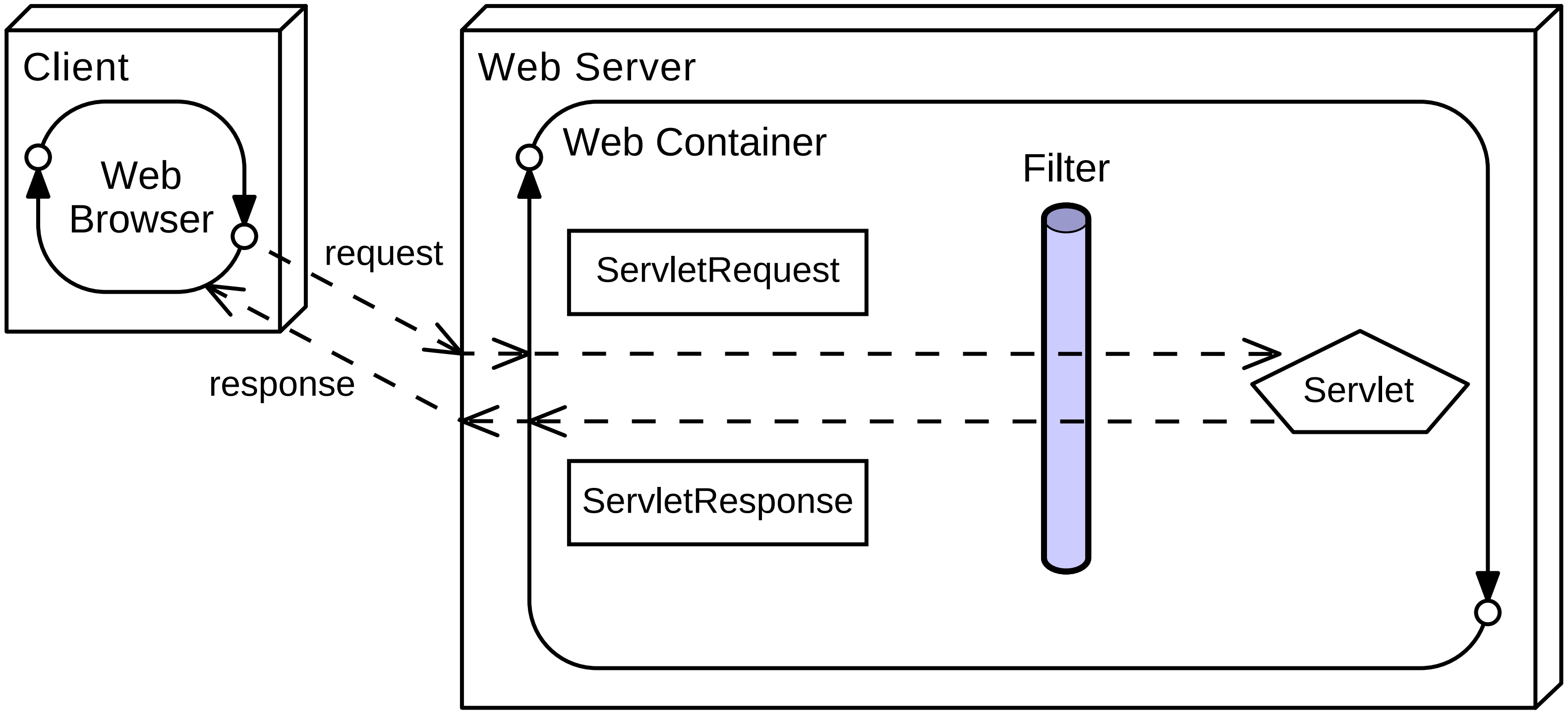
Web Container Request Processing

Request and response objects are created for each incoming request.



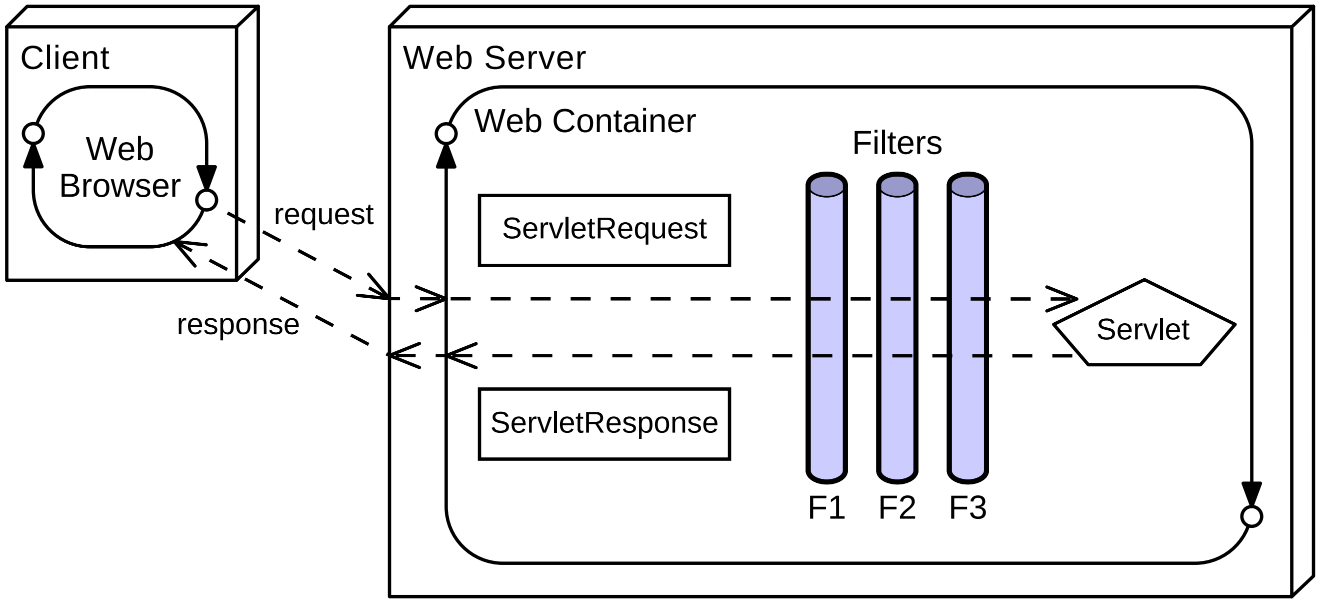
Applying Filters to an Incoming Request

- 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 (continued)

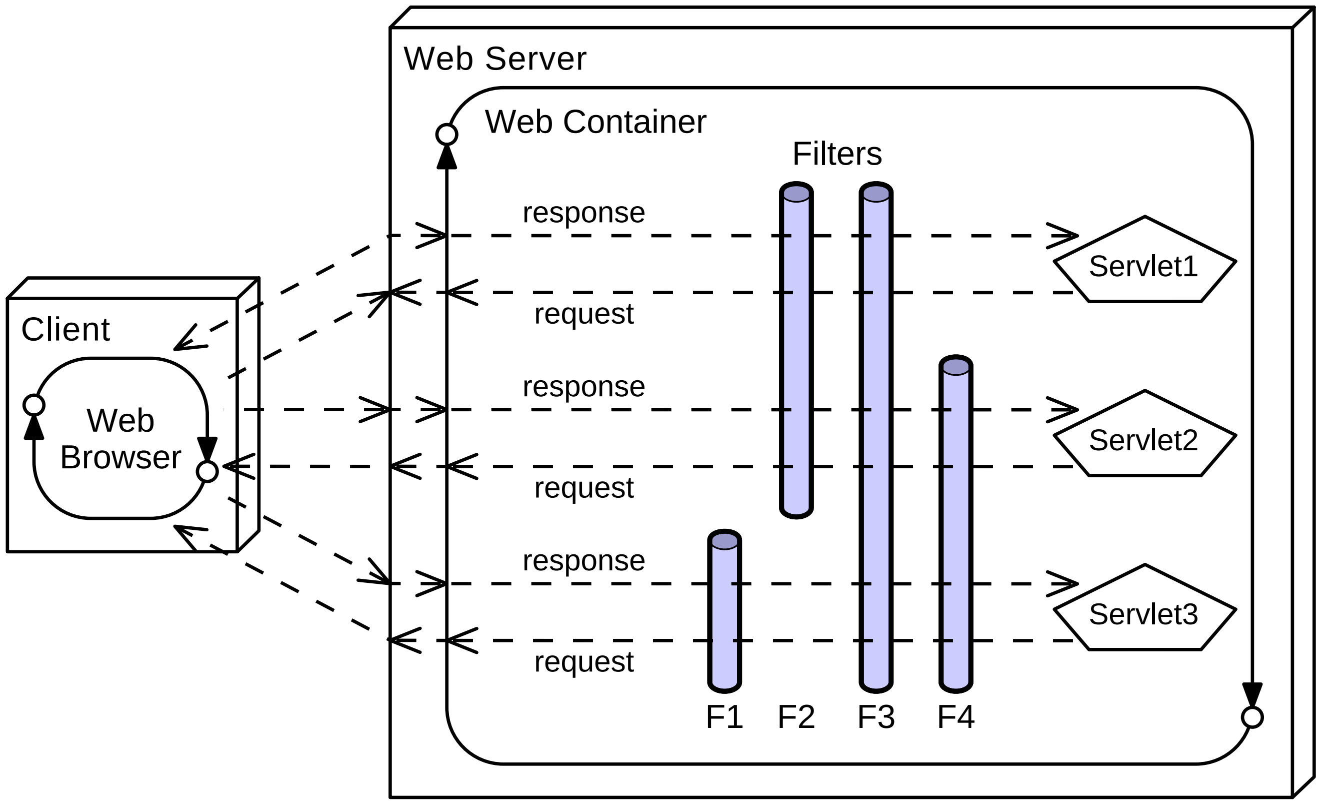
Multiple filters can intercept a given request.



This provides for modularity and reuse of code.

Applying Filters to an Incoming Request (continued)

Filters can be applied to different requests in different combinations.



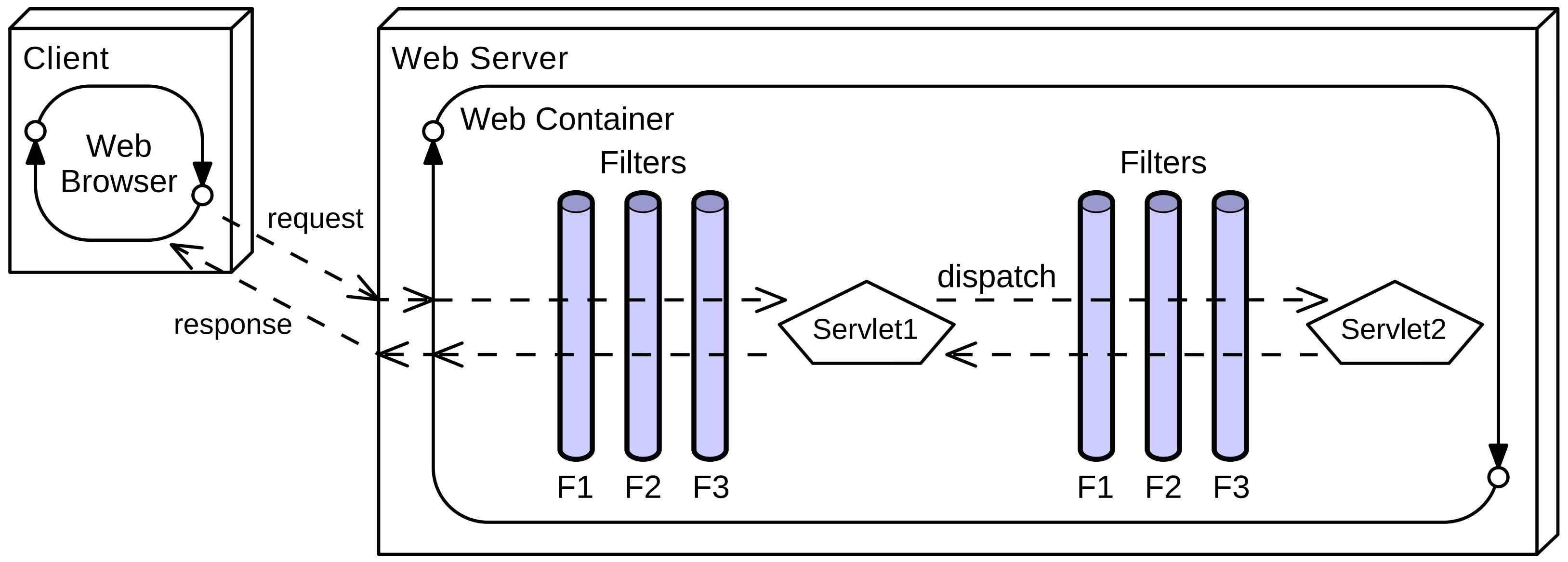
Applying Filters to an Incoming Request (continued)

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

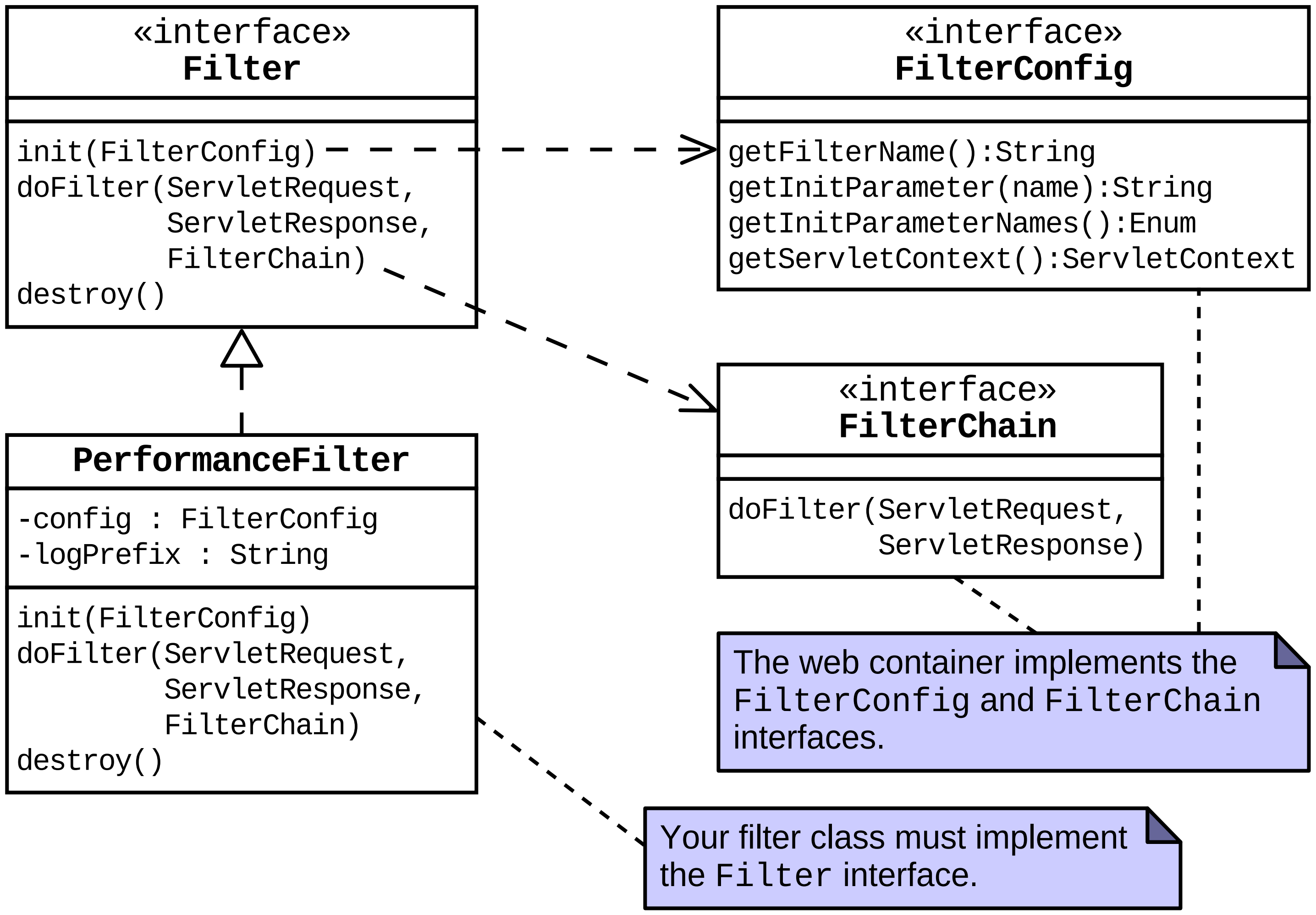
Filters Applied to a Dispatch

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.

Filter API



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

The `PerformanceFilter` Class

```java
package sl314.web;

import java.io.IOException;

import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;

import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;

public class PerformanceFilter implements Filter {

  private FilterConfig config;
  private String logPrefix;

```

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

```
public void init(FilterConfig config)
  throws ServletException {
  this.config = config;
  logPrefix = config.getInitParameter(“Log Entry Prefix”);
}
```

The `doFilter` Method

- The `doFilter` method is 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 (continued)

```

public void doFilter(ServletRequest request,
  ServletResponse response, FilterChain chain)
  throws ServletException, IOException {

  long begin = System.currentTimeMillis();
  chain.doFilter(request, response);
  long end = System.currentTimeMillis();

  StringBuffer logMessage = new StringBuffer();
  if (request instanceof HttpServletRequest) {
    logMessage = ((HttpServletRequest)request).getRequestURL();
  }
  logMessage.append(": ");
  logMessage.append(end - begin);
  logMessage.append(" ms");

  if(logPrefix != null) {
    logMessage.insert(0,logPrefix);
  }

  config.getServletContext().log(logMessage.toString());
}
```

The `destroy` Method

The `destroy` method is the last method called in the life cycle of a filter instance.

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

```
    public void destroy() {
      config = null;
      logPrefix = null;
    }
```

Configuring the Filter

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

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

Configuring the Filter (continued)

- 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

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

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 (continued)

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 (continued)

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

- Map sample data structure into database entities
- Design a web application to integrate with a DBMS
- Configure a DataSource and Java Naming and Directory Interface™ (JNDI) API

Relevance

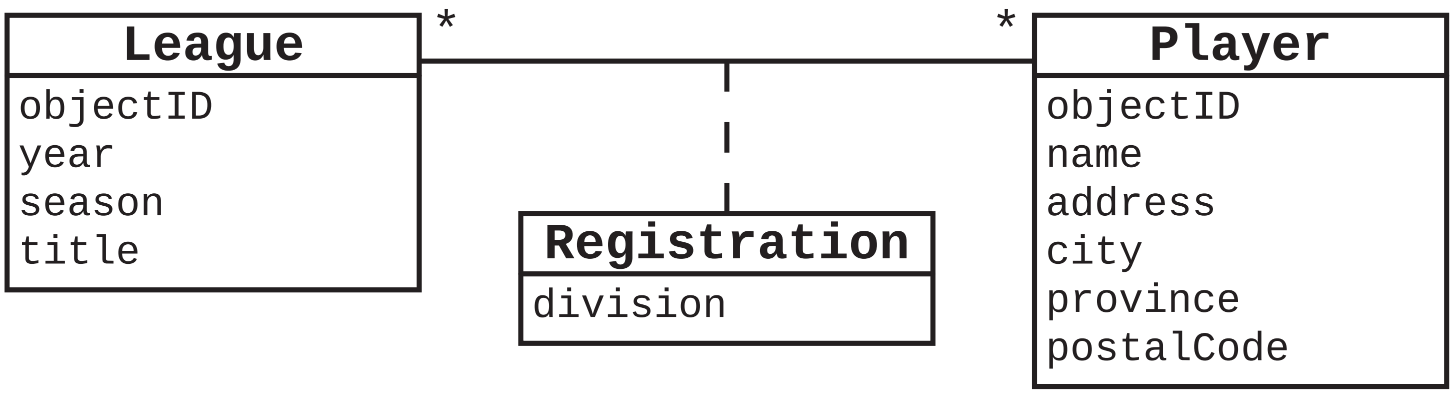
- Have you ever developed an application that integrates with the resource tier? How did you develop the access logic to the RDBMS?
- Did you ever have to change the database design? How did that affect the various tiers in your application?

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

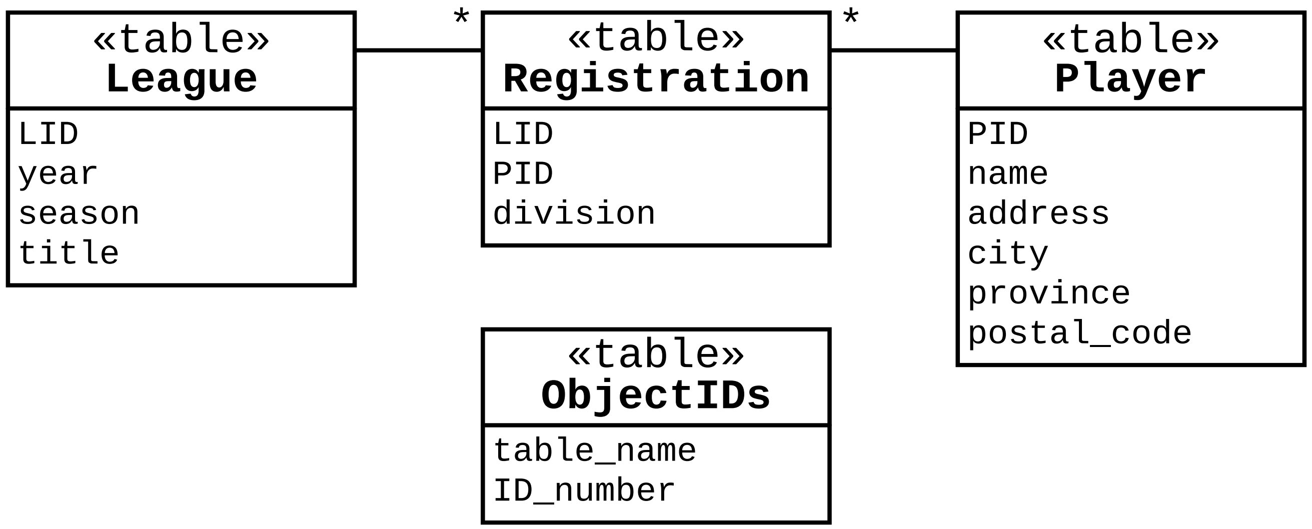
The following are the domain objects in Soccer League web application:



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 (continued)

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

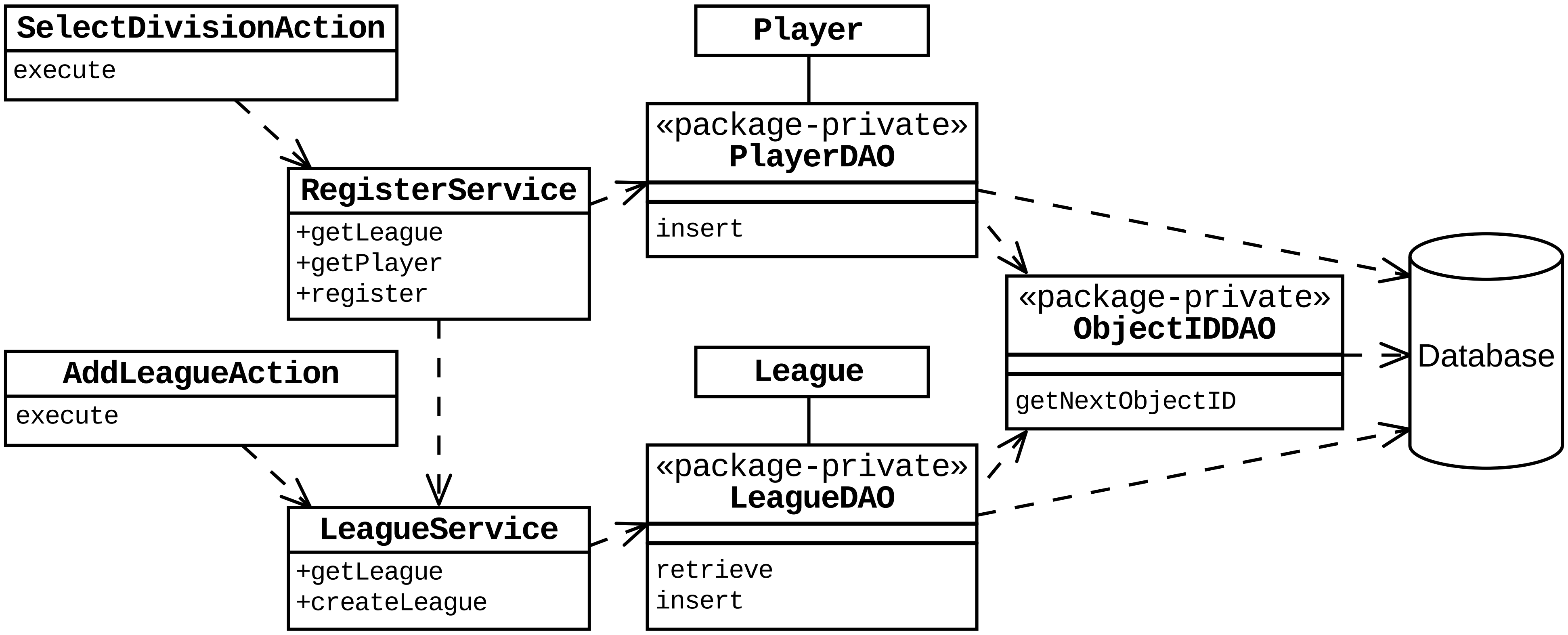
table_name	ID_number
League	005
Player	050

Data Access Object (DAO) 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.

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 DAO 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.

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?

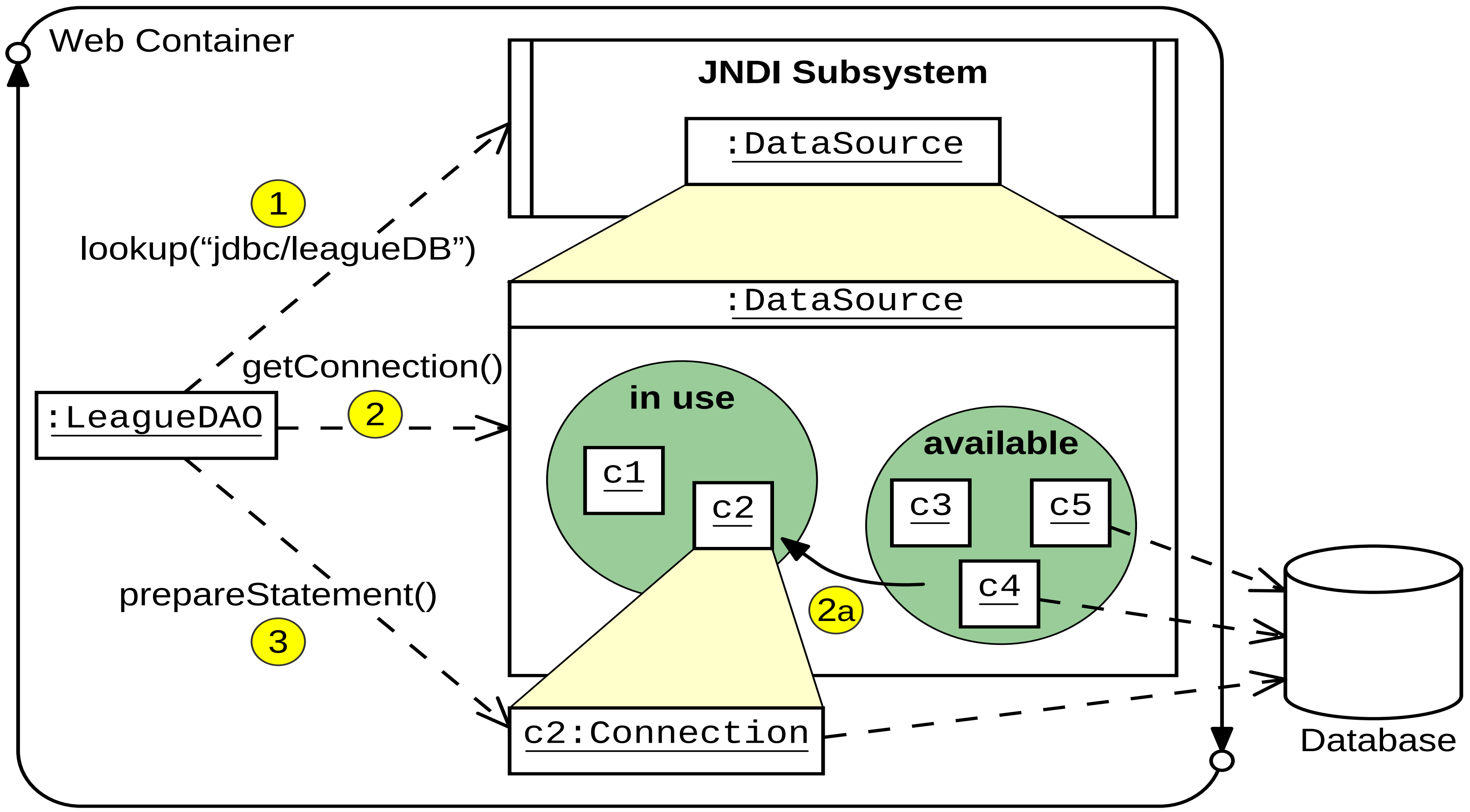
Traditional Approaches to Database Connections

- 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).

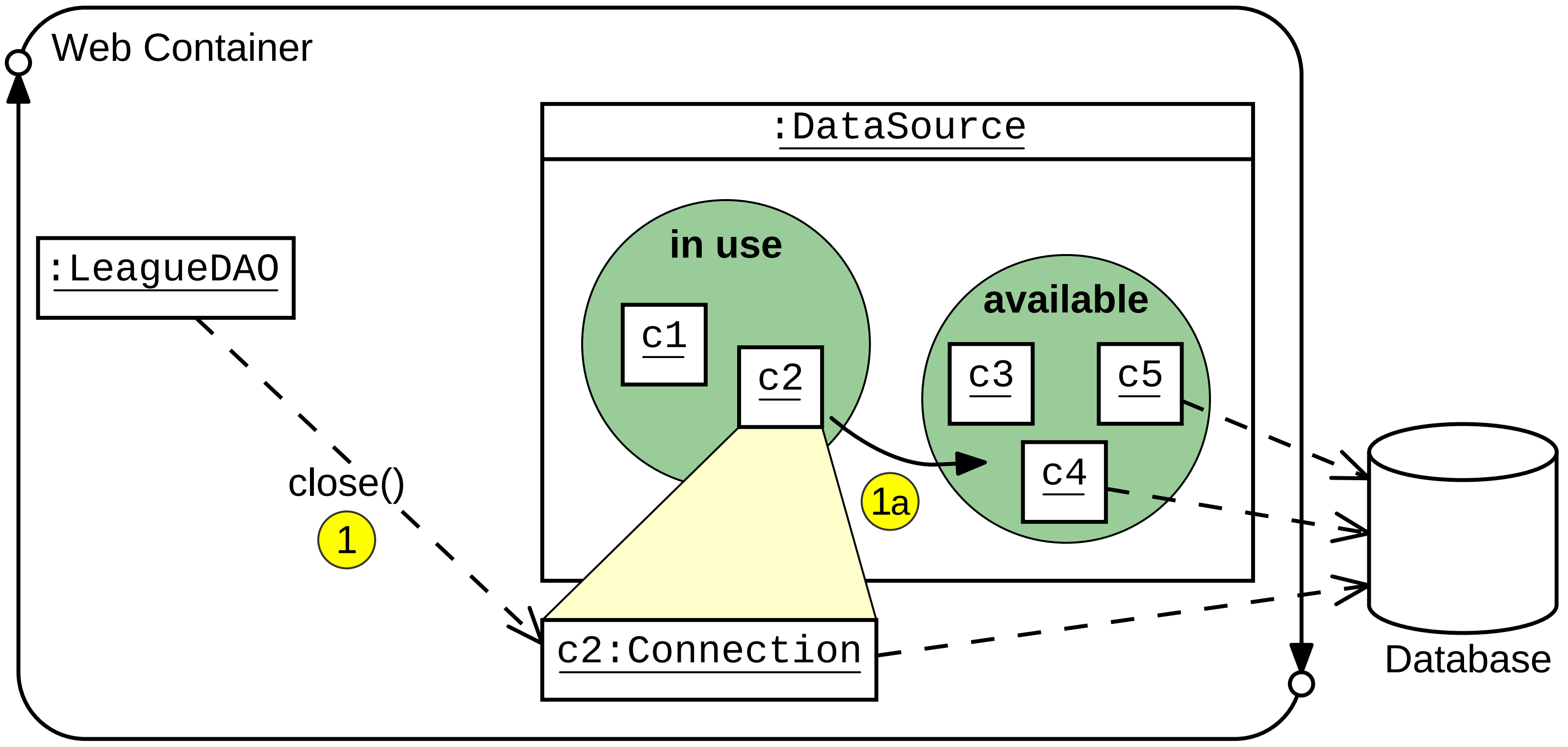
Using a DataSource and JNDI API

- Java EE application servers provide a namespace, which can be accessed using JNDI APIs.
- Java EE 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:

```
javax.sql.DataSource
---------------------------------------------------------------------------
getConnection():java.sql.Connection
getConnection(username: String, password: String): java.sql.Connection
```

- Locate `DataSource` using JNDI lookup:

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

Configuring a Sun Java Application Server DataSource and JNDI

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

```
                <taglib-location>/WEB-INF/struts-tiles.tld</taglib-locat
            </taglib>
        </jsp-config>
        <resource-ref>
            <res-ref-name>jdbc/dvdLibraryDB</res-ref-name>
            <res-type>javax.sql.DataSource</res-type>
            <res-auth>Container</res-auth>
            <res-sharing-scope>Shareable</res-sharing-scope>
        </resource-ref>
        </web-app>

```

Sun Java Application Server DataSource `sun-web.xml` Configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sun-web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Application Server 8.1 Servlet 2.4//EN" "http://www.sun.com/software/appserver/dtds/sun-web-app_2_4-1.dtd">
<sun-web-app error-url="">
  <context-root>/dvd</context-root>
  <resource-ref>
    <res-ref-name>jdbc/dvdLibraryDB</res-ref-name>
    <jndi-name>jdbc/dvdLibraryDB</jndi-name>
  </resource-ref>
  <class-loader delegate="true"/>
  <jsp-config>
    <property name="classdebuginfo" value="true">
      <description>Enable debug info compilation in the generated servlet class</description>
    </property>
```

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 Java EE 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 technology enables 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

```
public class HelloServlet extends HttpServlet {

  private static final String DEFAULT_NAME = “World”;

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
         throws IOException {
      generateResponse(request, response);
  }

  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
         throws IOException {
      generateResponse(request, response);
  }

  public void generateResponse(HttpServletRequest request,
                               HttpServletResponse response)
         throws IOException {

```

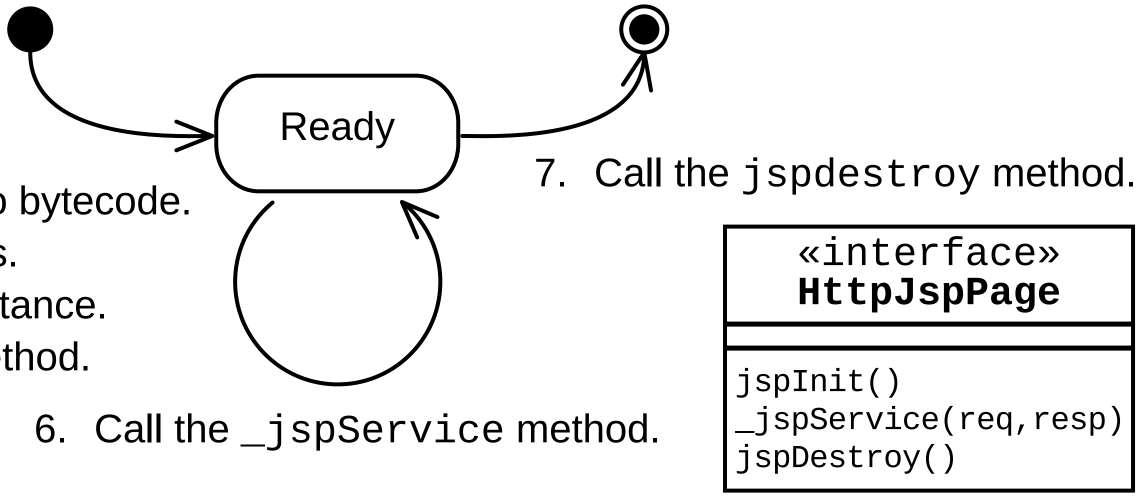
Hello World Servlet (continued)

```

        String name = request.getParameter(“name”);
        if ( (name == null) || (name.length() == 0) ) {
          name = DEFAULT_NAME;
        }

        response.setContentType(“text/html”);
        PrintWriter out = response.getWriter();

        out.println(“<HTML>”);
        out.println(“<HEAD>”);
        out.println(“<TITLE>Hello Servlet</TITLE>”);
        out.println(“</HEAD>”);
        out.println(“<BODY BGCOLOR=’white’>”);
        out.println(“<B>Hello, “ + name + “</B>”);
        out.println(“</BODY>”);
        out.println(“</HTML>”);

        out.close();
    }
```

The `hello.jsp` Page

```
<%! private static final String DEFAULT_NAME = “World”; %>

<html>

<head>
<title>Hello JavaServer Page</title>
</head>

<%-- Determine the specified name (or use default) --%>
<%
    String name = request.getParameter(“name”);
    if ( (name == null) || (name.length() == 0) ) {
      name = DEFAULT_NAME;
    }
%>

<body bgcolor=’white’>

<b>Hello, <%= name %></b>

</body>

</html>
```

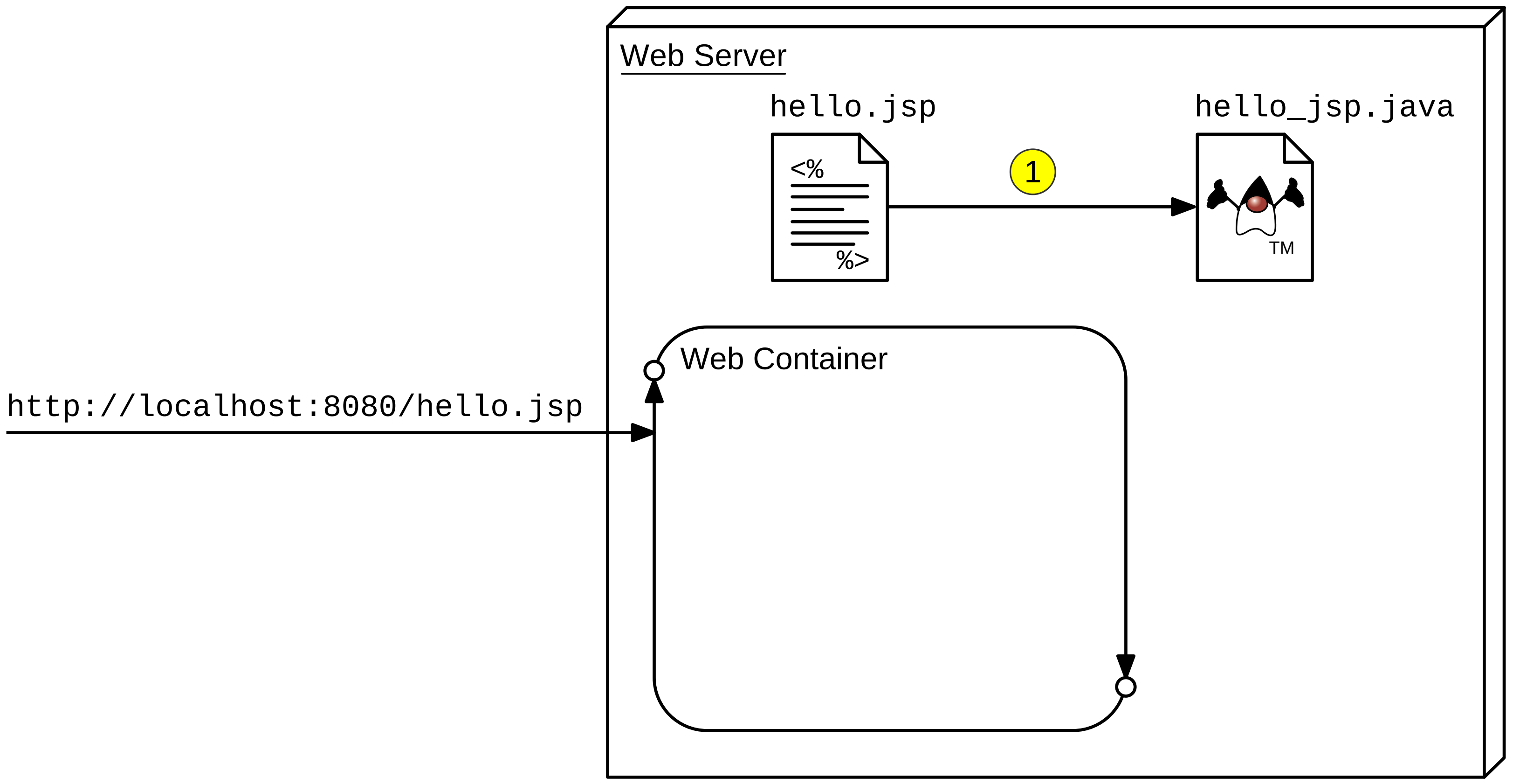
Steps of JSP Page Processing



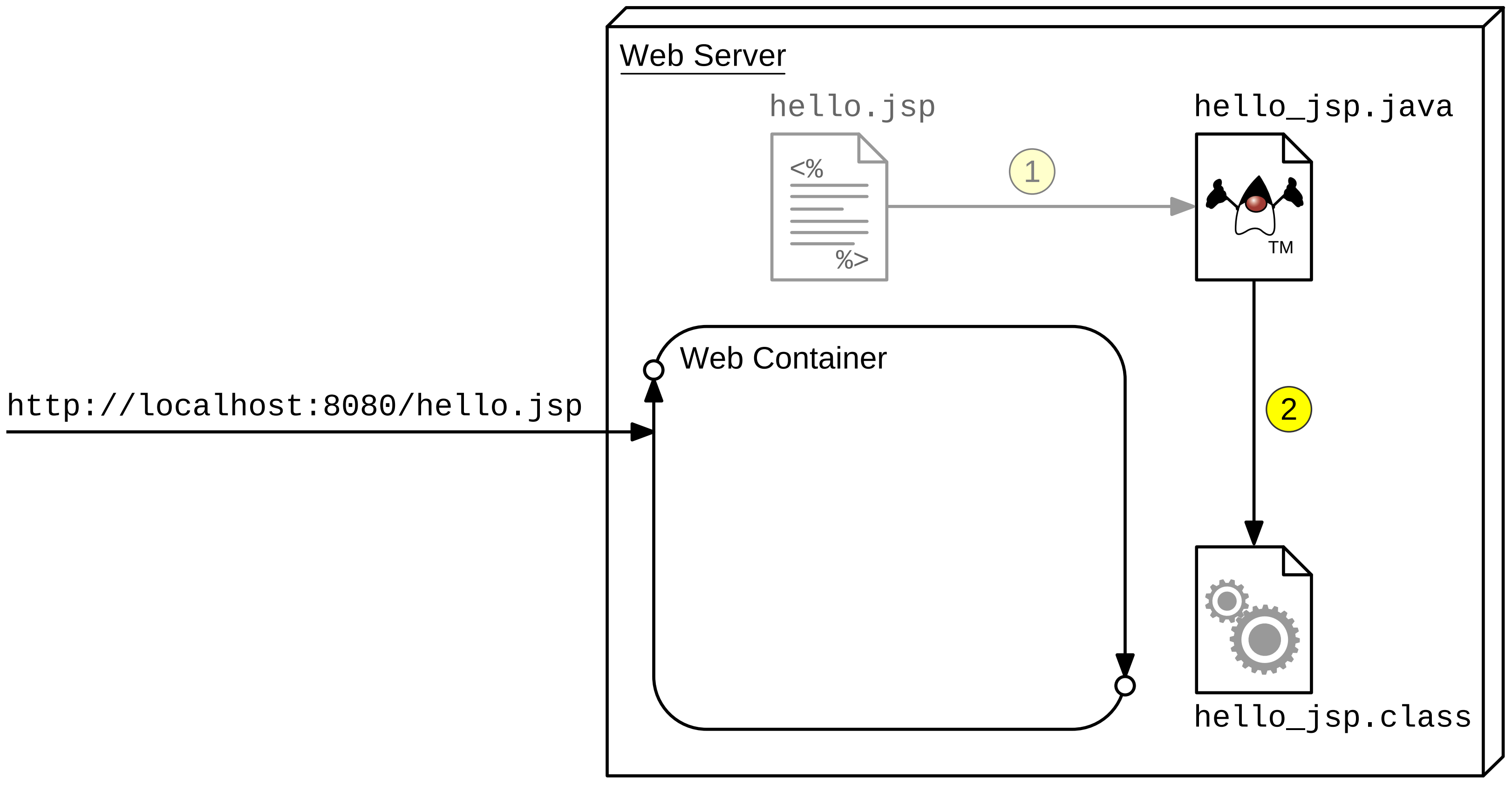
1. Translate the JSP to servlet code.
2. Compile the servlet to bytecode.
3. Load the servlet class.
4. Create the servlet instance.
5. Call the `jspInit` method.
6. Call the `_jspService` method.
7. Call the `jspdestroy` method.

```
«interface»
HttpJspPage

jspInit()
_jspService(req,resp)
jspDestroy()
```

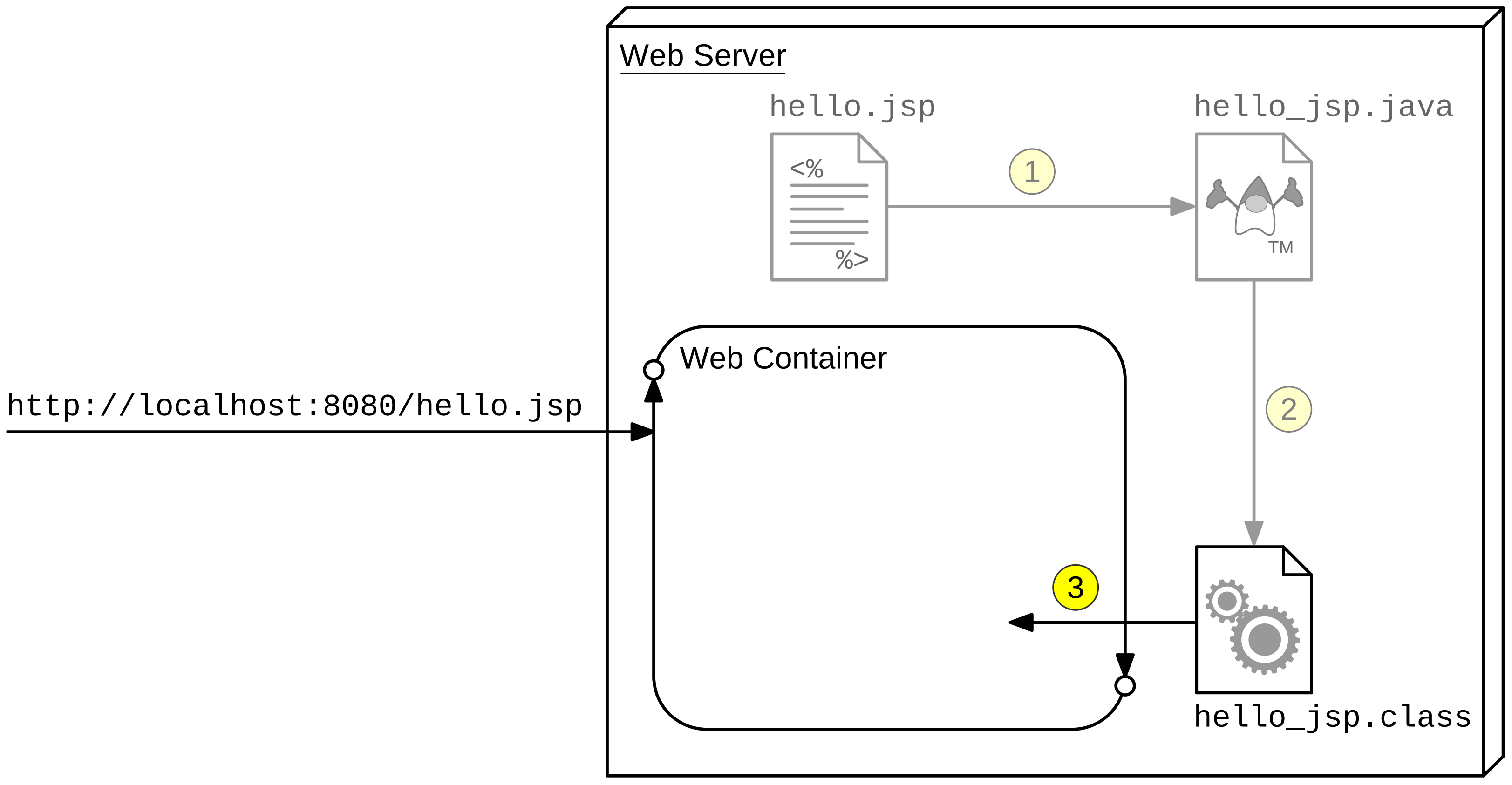
JSP Page Translation



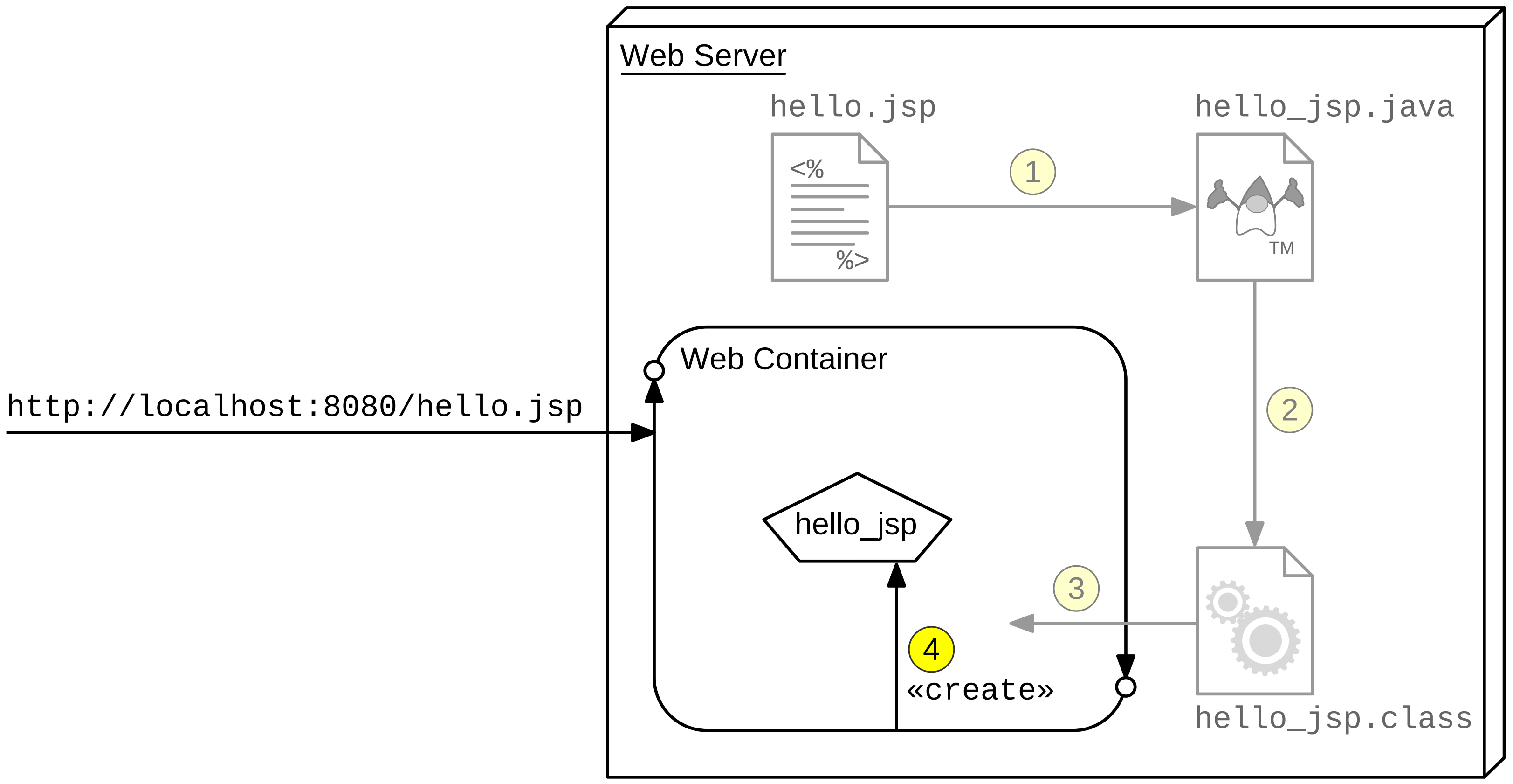
JSP Page Compilation



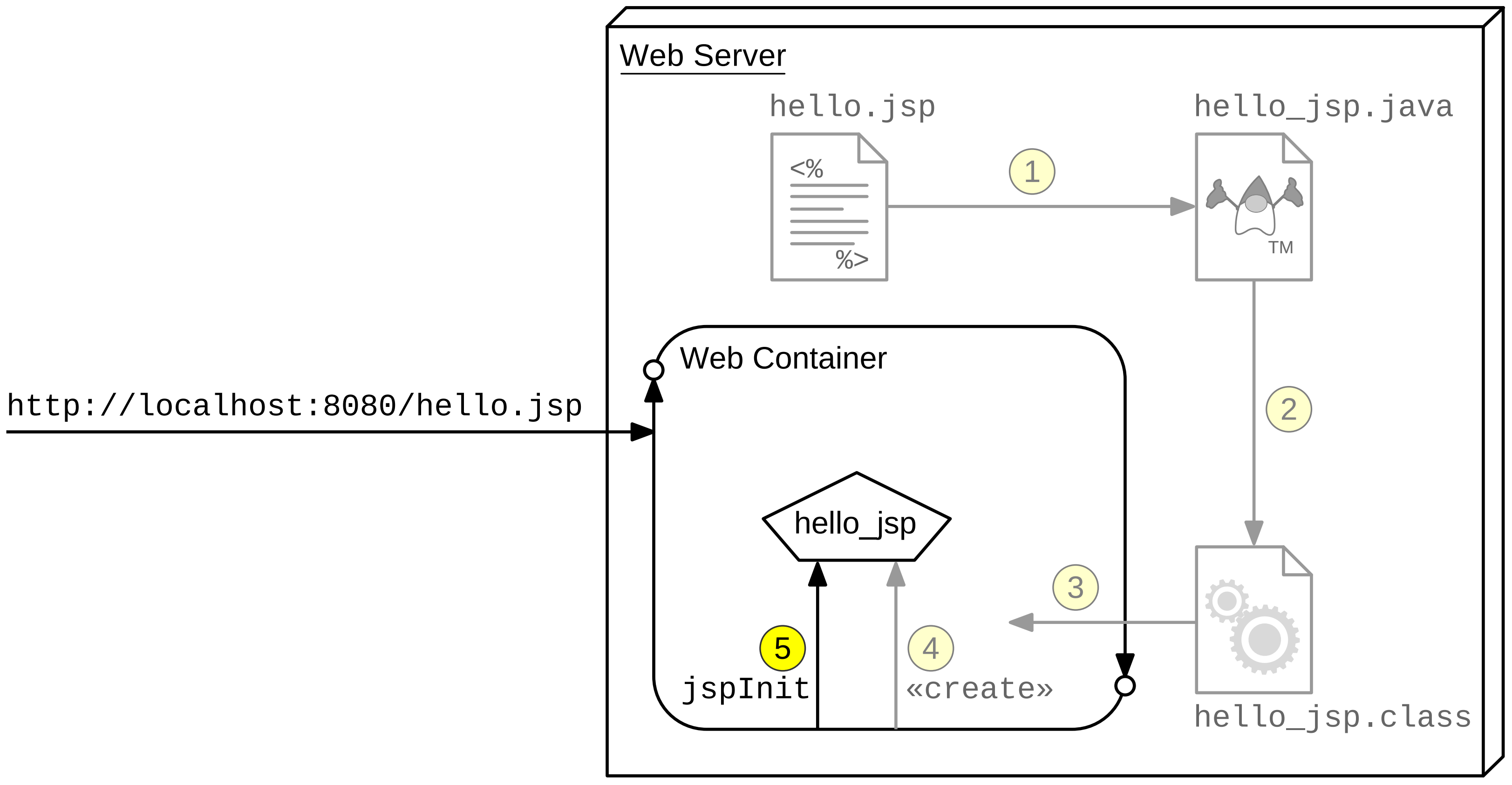
JSP Page Class Loading



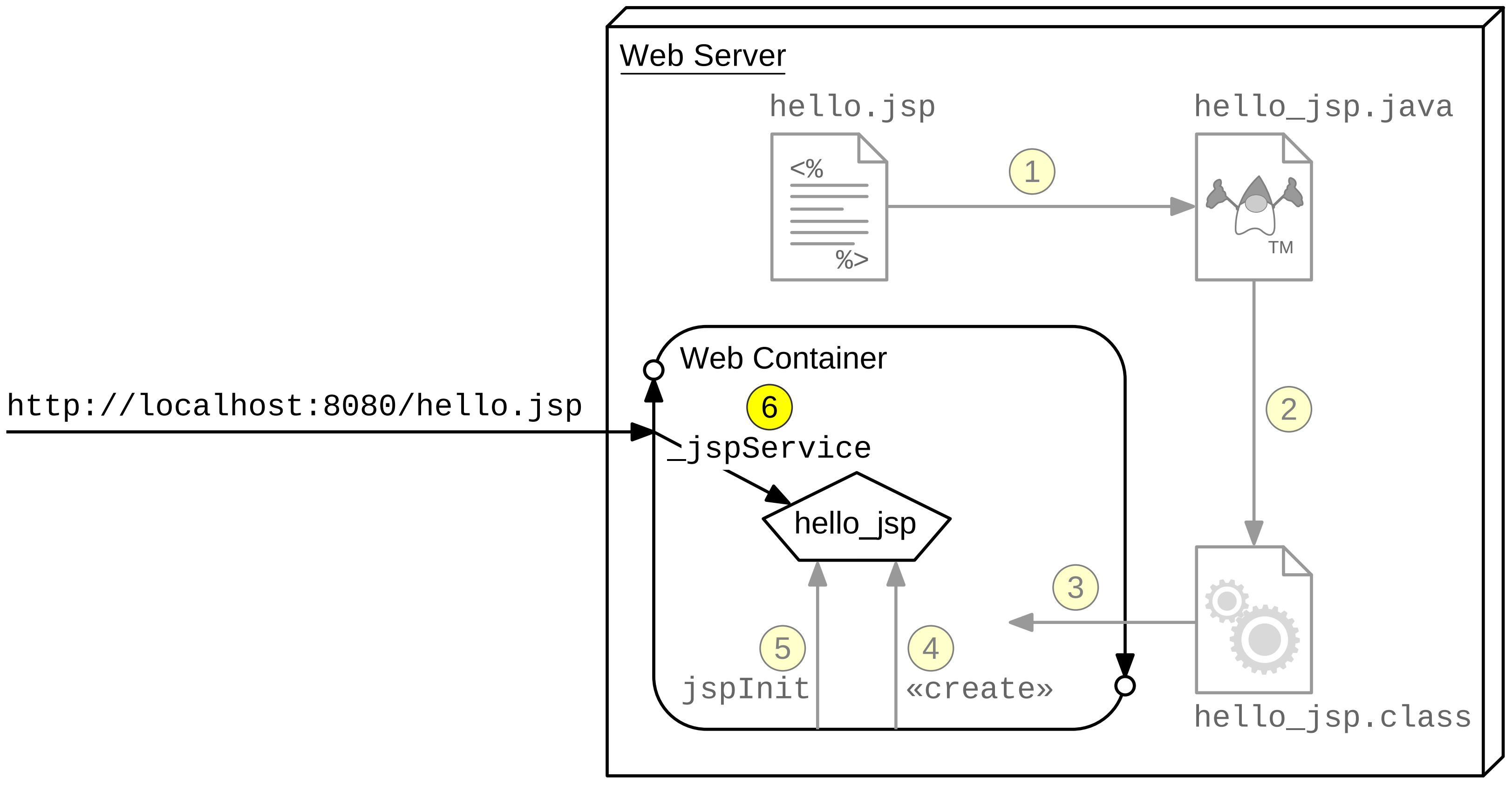
JSP Page Servlet Instance



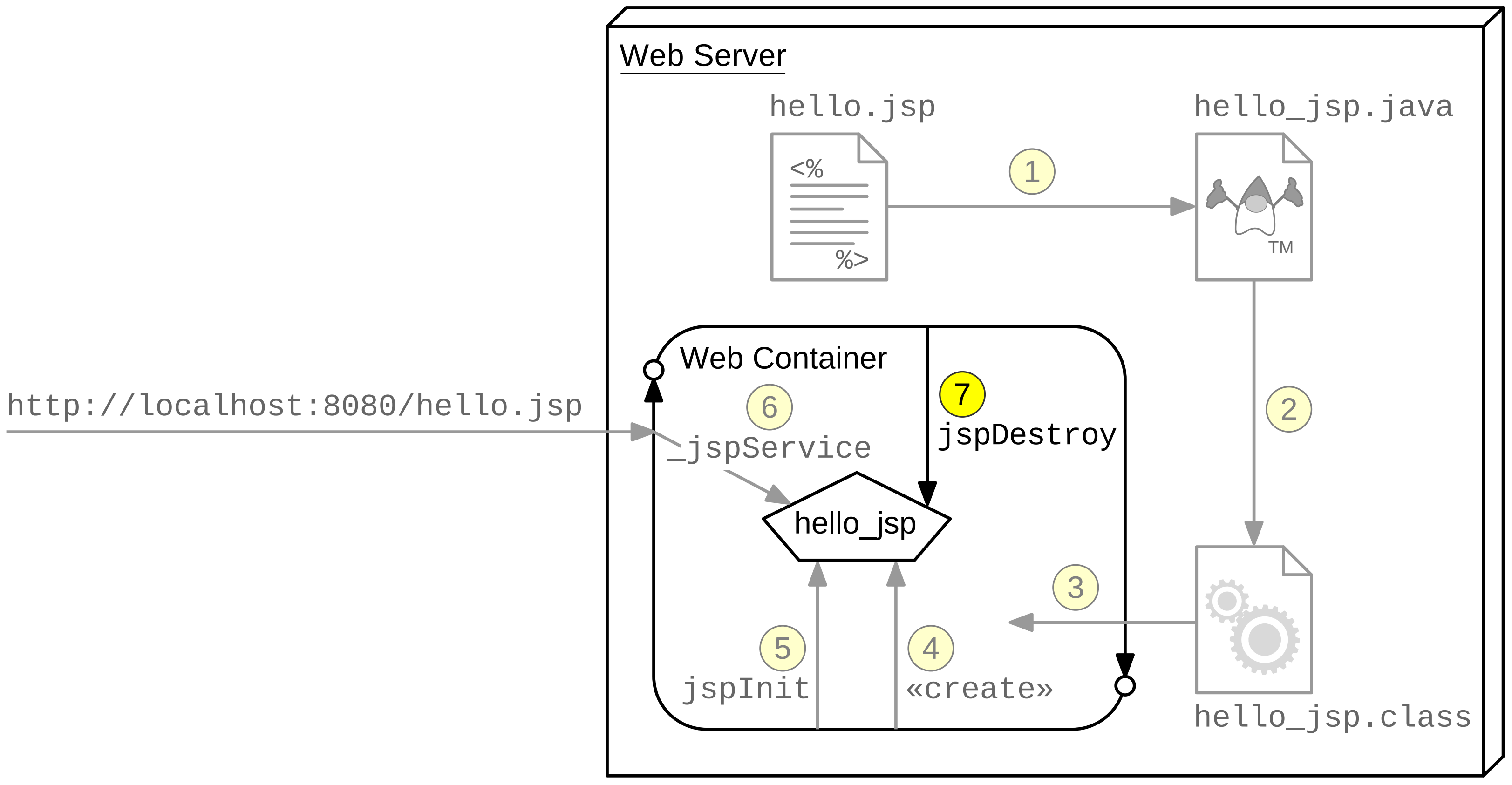
JSP Page Initialization



JSP Page Service

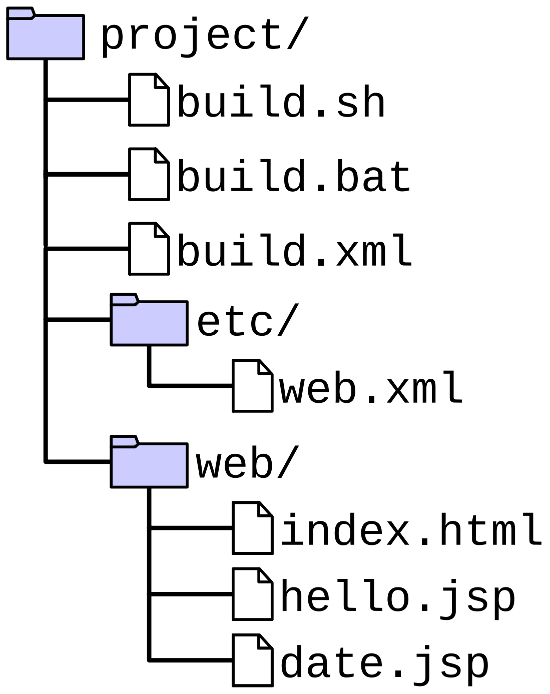


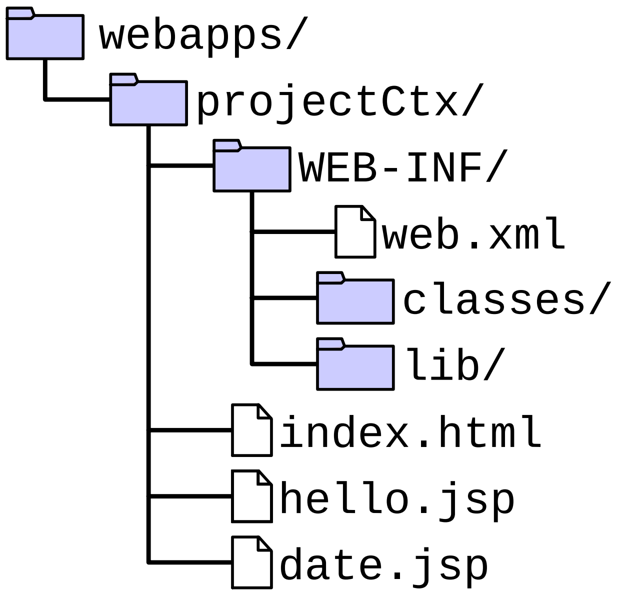
JSP Page Destroyed



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:





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	`<%-- comment --%>`
Directive	`<%@ directive %>`
Declaration	`<%! decl %>`
Scriplet	`<% code %>`
Expression	`<%= expr %>`

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 am a big number.
<% } else { %>
     I am 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
`request`	The `HttpServletRequest` object associated with the request.
`response`	The `HttpServletResponse` object associated with the response that is sent back to the browser.
`out`	The `JspWriter` object associated with the output stream of the response.
`session`	The `HttpSession` object associated with the session for the given user of the request. This variable is only meaningful if the JSP page is participating in an HTTP session.
`application`	The `ServletContext` object for the web application.

Implicit Variables (continued)

Additional variables:

Variable Name	**Description**
`config`	The `ServletConfig` object associated with the servlet for this JSP page.
`pageContext`	The `pageContext` object encapsulates the environment of a single request for this JSP page.
`page`	The `page` variable is equivalent to the `this` variable in the Java programming language.
`exception`	The `Throwable` object that was thrown by some other JSP page. This variable is only available in a JSP error page.

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 (continued)

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 `java` 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. *Do not* use this attribute.
`buffer`	Defines the size of the buffer used in the output stream (a `JspWriter` object). The value is either `none` or `N`kb. The default buffer size is 8 KB or greater. For example: `buffer="8kb"` or `buffer="none"`

Using the page Directive (continued)

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 `true` (automatically flush) or `false` (throw an exception). The default is `true`.
`session`	Defines whether the JSP page is participating in an HTTP session. The value can be either `true` (the default) or `false`.
`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: `import=”java.sql.Date,java.util.*,java.text.*”`

Using the page Directive (continued)

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

Using the page Directive (continued)

Attribute	**Use**
`isErrorPage`	Defines that the JSP page has been designed to be the target of another JSP page's `errorPage` attribute. The value is either `true` or `false` (default). All JSP pages that are an error page automatically have access to the `exception` implicit variable.
`errorPage`	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, `errorPage="error.jsp"` (this is relative to the current hierarchy) or `errorPage="/error/formErrors.jsp"` (this is relative to the context root)

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

```java
package sl314.beans;

import java.io.Serializable;

public class CustomerBean implements Serializable {

  private String name;
  private String email;
  private String phone;

  public CustomerBean() {
    this.name = "";
    this.email = "";
    this.phone = "";
  }

  public void setName(String name) {
    this.name = name;
  }
  public String getName() {
    return name;
  }
```

The CustomerBean JavaBeans Component (continued)

```

  public void setEmail(String email) {
    this.email = email;
  }
  public String getEmail() {
    return email;
  }

  public void setPhone(String phone) {
    this.phone = phone;
  }
  public String getPhone() {
    return phone;
  }

} // END of CustomerBean class
```

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 (continued)

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

- Given

```
<jsp:useBean id="myBean"
             scope="request"
            class="sl314.beans.CustomerBean" />
```

- Java equivalent:

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

The useBean Tag (continued)

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

```
<jsp:useBean id=”cust” scope=”request”
     class=”sl314.beans.CustomerBean”>
  <%
    cust.setName(request.getParameter(“name”));
    cust.setEmail(request.getParameter(“email”));
    cust.setPhone(request.getParameter(“phone”));
  %>
</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 (continued)

The `setProperty` tag:

- Given:

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

- Java technology code 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 technology code equivalent:

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

The `getProperty` Tag (continued)

The `useBean` tag output appears along with template text.

```
<H2>Customer Information:</H2>
Name: <jsp:getProperty name=”cust” property=”name” /><BR>
Email: <jsp:getProperty name=”cust” property=”email” /><BR>
Phone: <jsp:getProperty name=”cust” property=”phone” /><BR>
```

Using Expression Language (EL) 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 page 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:

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

EL Implicit Objects (continued)

Additional objects:

Implicit Object	**Description**
`paramValues`	A Map containing request parameters and their corresponding string arrays
`header`	A Map containing header names and single string values
`headerValues`	A Map containing header names and their corresponding string arrays
`cookie`	A Map containing cookie names and their values

EL Implicit Objects (continued)

For example,

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

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

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

Unified Expression Language

There are two form of expression language

- `#{...}` syntax
- `${...}` syntax

`#{...}`	**`${...}`**
Deferred Expression: Evaluated in a multi-phase request life cycle	Immediate Expression: Evaluated only when rendering output
Read and write values	Read-only value
Useful in JavaServer™ Faces pages	Useful in JavaServer Faces pages
Not useful in traditional JSP pages	Useful in traditional JSP pages

`#{...}` expression syntax is not used in this course.

Arithmetic Operators

Five arithmetic operators are defined:

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

Arithmetic Operators (continued)

Example operations:

EL Expression	**Result**
`${3 div 4}`	0.75
`${1 + 2 * 4}`	9
`${(1 + 2) * 4}`	12
`${32 mod 10}`	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 (continued)

- 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

Configuring the JSP Environment

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

- Defined within the `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 (continued)

Multiple `jsp-property-group` elements are available:

```
<jsp-config>
  <jsp-property-group>
    <url-pattern>/scripting_off/*</url-pattern>
    <scripting-invalid>true</scripting-invalid>
  </jsp-property-group>

  <jsp-property-group>
    <url-pattern>/EL_off/*</url-pattern>
    <el-ignored>true</el-ignored>
  </jsp-property-group>

  <jsp-property-group>
    <url-pattern>/prelude_coda/*</url-pattern>
    <include-prelude>/prelude_coda/prelude.jspf</include-prelude>
    <include-coda>/prelude_coda/coda.jspf</include-coda>
  </jsp-property-group>
</jsp-config>
```

Summary

- JSP 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

- Describe the Java EE job roles involved in web application development
- Design a web application using custom tags
- Use JavaServer Pages Tag Library (JSTL) tags in a JSP Page

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?

The Java EE Job Roles Involved in Web Application Development

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)

Contrasting Custom Tags and Scriptlet Code

```
<%-- Report any errors (if any) --%>
<%
    // Retrieve the errorMsgs from the request-scope
    List errorMsgs = (List) request.getAttribute(“errorMsgs”);
    if ( (errorMsgs != null) && !errorMsgs.isEmpty() ) {
%>
<p>
<font color=’red’>Please correct the following errors:
<ul>
<%
      Iterator items = errorMsgs.iterator();
      while ( items.hasNext() ) {
        String message = (String) items.next();
%>
  <li><%= message %></li>
<%
      } // END of while loop over errorMsgs
%>
</ul>
</font>
</p>
<%
  } // END of if errorMsgs is not empty
%>
```

Contrasting Custom Tags and Scriptlet Code (continued)

Equivalent custom tag in the registration form:

```
<%-- Report any errors (if any) --%>
<c:if test=”${not empty errorMsgs}”>
  <p>
  <font color=’red’>Please correct the following errors:
  <ul>
  <c:forEach var=”message” items=”${errorMsgs}”>
    <li>${message}</li>
  </c:forEach>
  </ul>
  </font>
  </p>
</c:if>
```

Contrasting Custom Tags and Scriptlet Code (continued)

Advantages of custom tags compared to scriptlet code:

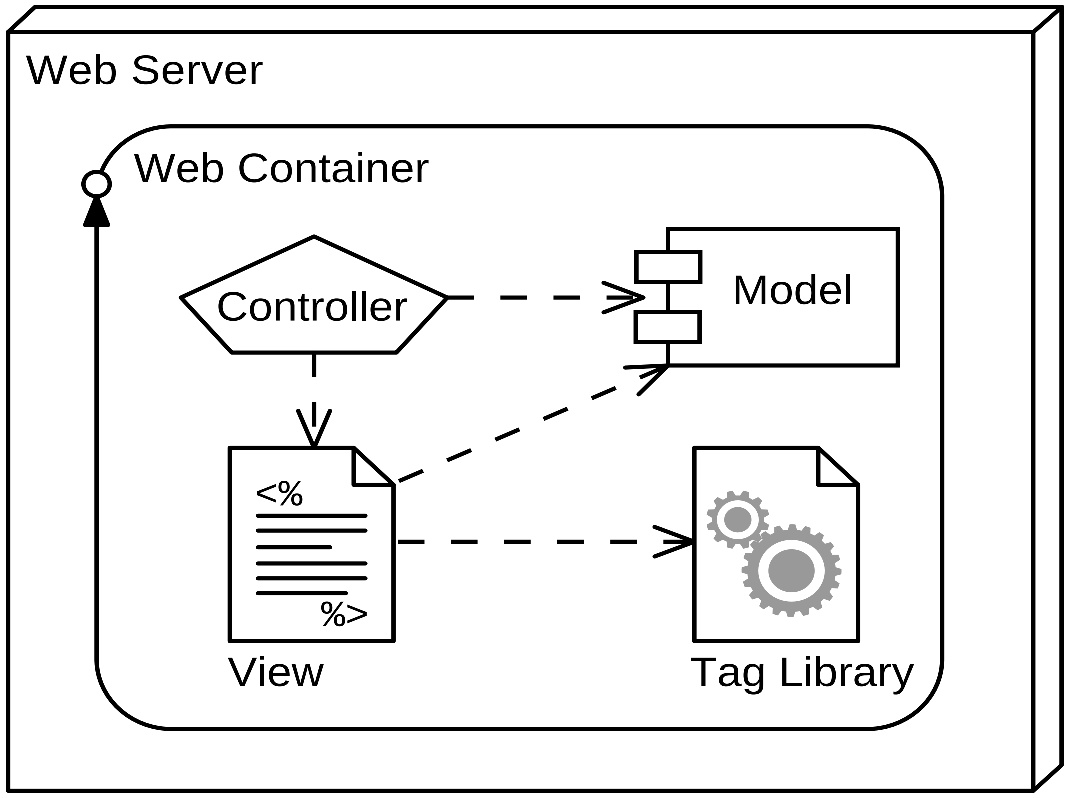
- Java technology code is removed from the JSP page.
- Custom tags are reusable components.
- Standard job roles are supported.

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 (continued)

- 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 (continued)

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

```

<%-- Set page title --%>
<c:set var=”pageTitle”>Duke’s Soccer League: Registration</c:set>

<%-- Generate the HTML response --%>
<html>
<head>
  <title>${pageTitle}</title>
</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 (continued)

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

```

<%-- Report any errors (if any) --%>
<c:if test="${not empty errorMsgs}">
  <p>
  <font color='red'>Please correct the following errors:
  <ul>
  <c:forEach var="message" items="${errorMsgs}">
    <li>${message}</li>
  </c:forEach>
  </ul>
  </font>
  </p>
</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 (continued)

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 (continued)

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

```
<font color='red'>Please correct the following errors:
<ul>
<c:forEach var="message" items="${errorMsgs}">
  <li>${message}</li>
</c:forEach>
</ul>
</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 (continued)

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

```

<%-- Present the form --%>
<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.
- The `value` attribute – This attribute specifies the expression to be evaluated.
- The `default` attribute – This optional attribute specifies a result to use if the expression evaluates to `null`.
- The `escapeXml` attribute – This optional attribute indicates whether or not the characters (`<`), (`>`), (`&`), (`‘`), and (`“`) should be replaced (default is `true`).

The out Tag (continued)

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>
```

Using a Custom Tag Library in JSP Pages

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.

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

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

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

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.

```
<%-- Present the form --%>
<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) {
%>
   <%-- “error messages” JSP code --%>
<%
  } // end of IF
%>
```

Equivalent custom tag in the error page:

```
<c:if test=”${not empty errorMsgs}”>
   <%-- “error messages” JSP code --%>
</c:if>
```

Using an Iterative Custom Tag

```

<%
    // Retrieve the set of leagues the LeagueService
    LeagueService leagueSvc = new LeagueService();
    List leagueList = leagueSvc.getAllLeagues();
%>
<%-- Generate main body --%>
<p>
The set of soccer leagues are:
</p>
<ul>
<%
    Iterator items = leagueList.iterator();
    while ( items.hasNext() ) {
      League league = (League) items.next();
%>
    <li><%= league.getTitle() %></li>
<%    }    %>
</ul>
```

Using an Iterative Custom Tag (continued)

```

<%-- Retrieve the set of leagues the LeagueService --%>
<jsp:useBean id="leagueSvc" scope="page"
     class="sl314.model.LeagueService" />

<%-- Generate main body --%>
<p>
The set of soccer leagues are:
</p>

<ul>
<c:forEach var="league" items="${leagueSvc.allLeagues}" >
  <li>${league.title}</li>
</c:forEach>
</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

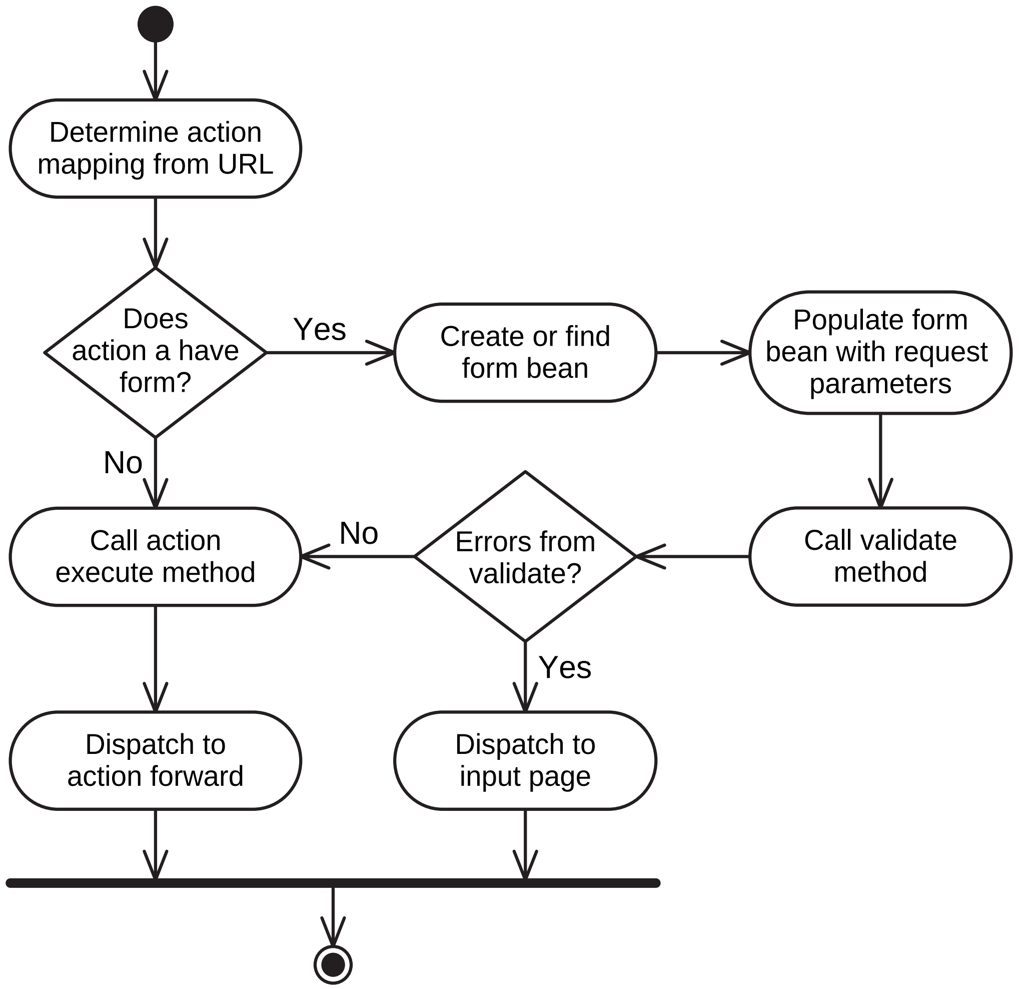
- What are the responsibilities of a Struts action class?
- Which of these responsibilities are really View-related aspects of the boundary component?
- Does Struts provide any facilities for separating these View-related aspects for the Controller-related action classes?

Struts Application Components

Struts applications consist of the following:

- Model elements
- View elements
- Control elements

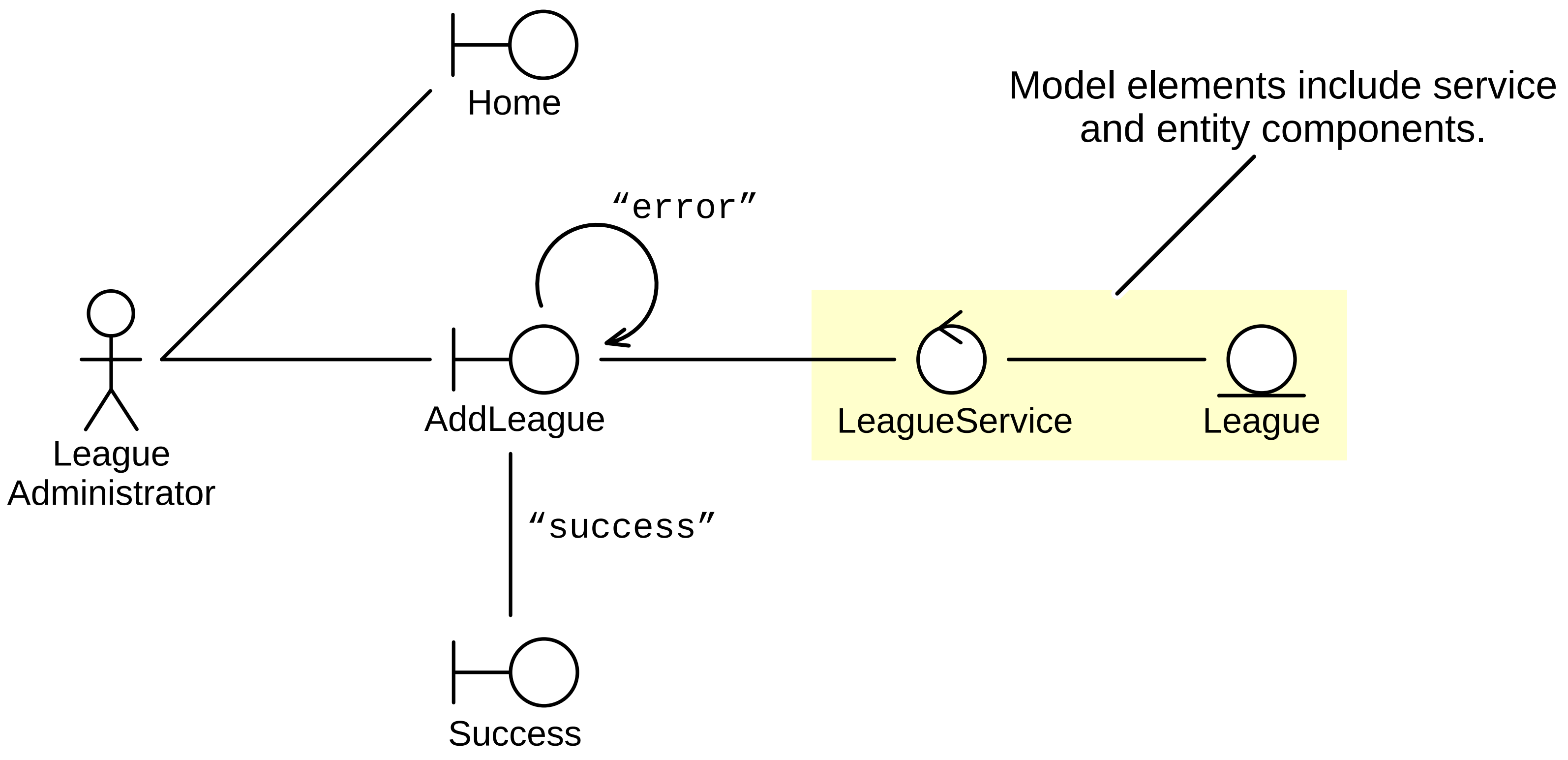
Struts Activity Diagram



Determine action mapping from URL
Does action a have form?
Yes
Create or find form bean
Populate form bean with request parameters
No
Call action execute method
No
Errors from validate?
Call validate method
Yes
Dispatch to action forward
Dispatch to input page

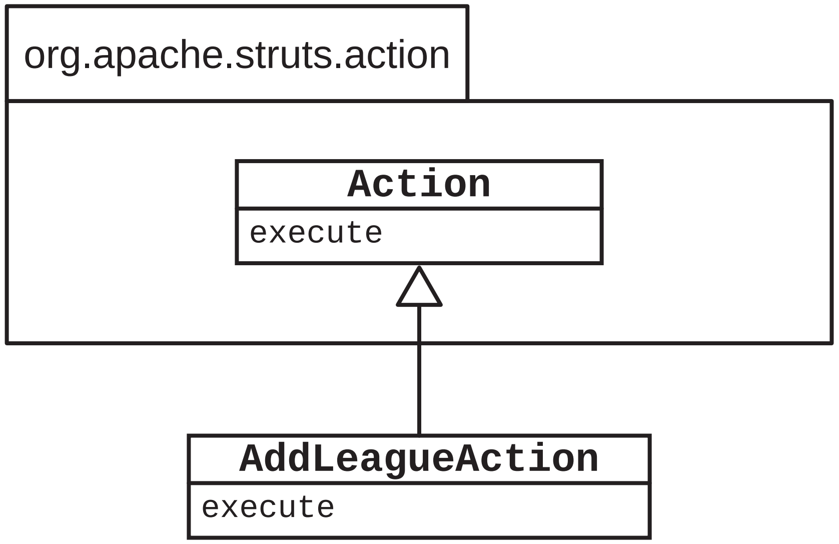
Model Elements Review

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



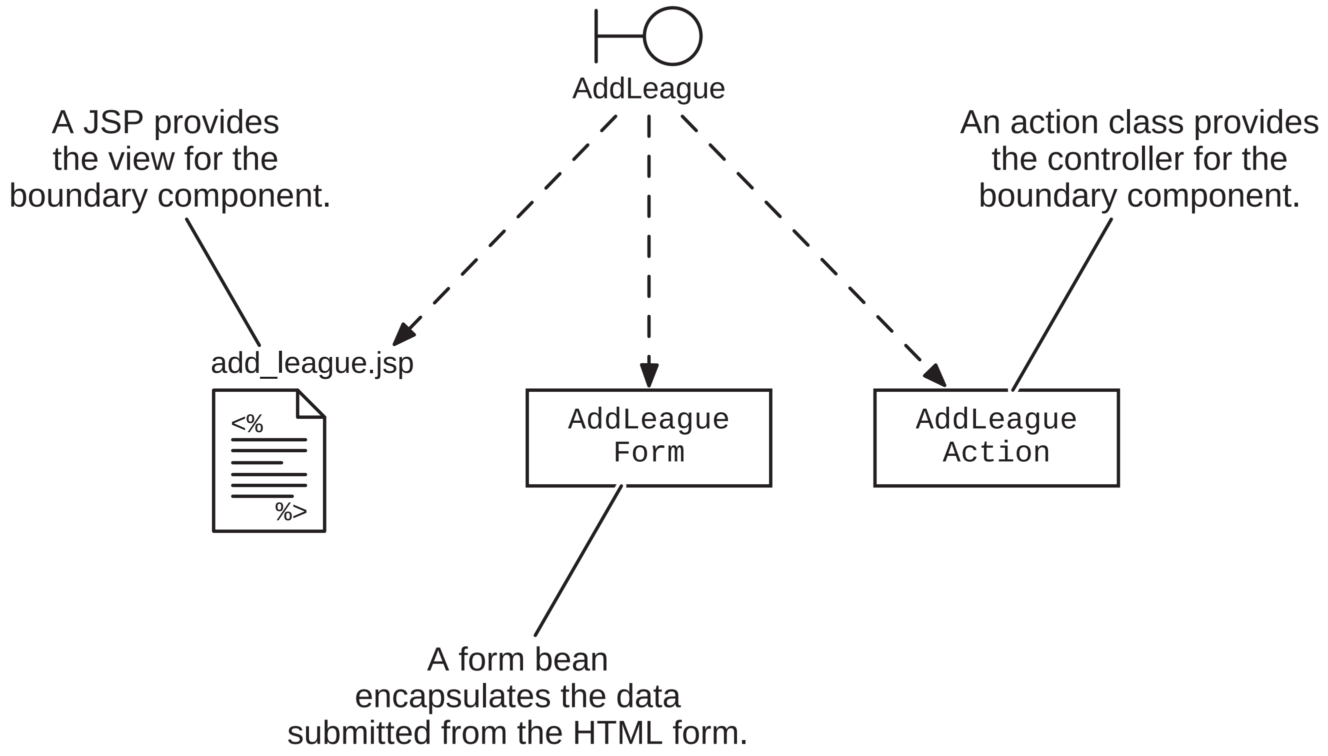
Control Elements Review (continued)

The `Action` subclasses are configured in the Struts configuration file.

```

    <!-- Declare the Registration actions -->
    <action path="/register/select_league"
            type="sl314.controller.SelectLeagueAction"
            name="selectLeagueForm" scope="request" validate="true"
            input="/register/select_league.jsp" >
      <forward name="success" path="/register/enter_player.jsp"/>
      <forward name="error" path="/register/select_league.jsp"/>
    </action>
```

View Elements Review

- Views in Struts can have multiple aspects.

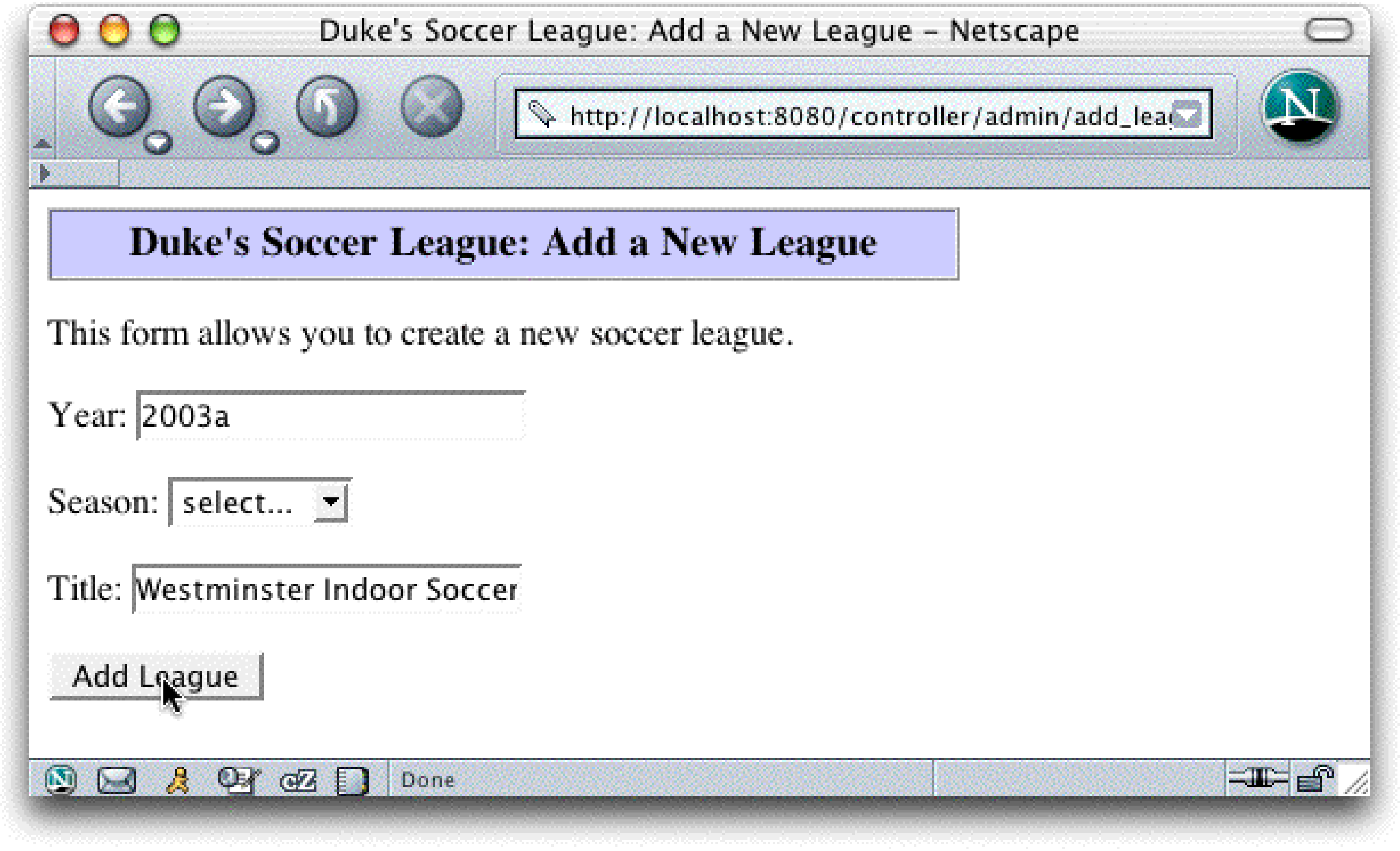

AddLeague
A JSP provides the view for the boundary component.
An action class provides the controller for the boundary component.
add_league.jsp
<%
%>
AddLeague Form
AddLeague Action
A form bean encapsulates the data submitted from the HTML form.

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

Developing an `ActionForm` Class

- `ActionForm` classes provide an object representation of the elements in an HTML form.
- `ActionForm` classes are automatically created or located by the infrastructure controller.
- `ActionForm` classes are placed into the scope specified in the Struts configuration file.
- The form bean is self-validating.

The Add a New League Form



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

http://localhost:8080/controller/admin/add_lea

Duke's Soccer League: Add a New League

This form allows you to create a new soccer league.

Year: 2003a

Season: select...

Title: Westminster Indoor Soccer

Add League

Done

The AddLeagueForm Class

```java
package sl314.view;

// Struts imports
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.action.ActionError;
import org.apache.struts.action.ActionErrors;
// Servlet imports
import javax.servlet.http.HttpServletRequest;

/**
 * This is a Struts form bean for the “Add League” view.
 */
public class AddLeagueForm extends ActionForm {
```

The AddLeagueForm Class (Part 2)

```
/**
 * This is a Struts form bean for the “Add League” view.
 */
public class AddLeagueForm extends ActionForm {

  private String season = null;
  public String getSeason() {
    return season;
  }
  public void setSeason(String season) {
    this.season = season;
  }

  private String title = null;
  public String getTitle() {
    return title;
  }
  public void setTitle(String title) {
    this.title = title;
  }
```

The `AddLeagueForm` Class (Part 3)

```

    // The raw ‘year’ property
    private String yearStr = null;
    public String getYearStr() {
      return yearStr;
    }
    public void setYearStr(String yearStr) {
      this.yearStr = yearStr;
    }
    // The converted ‘year’ property
    private int year = -1;
    public int getYear() {
      return year;
    }

```

The AddLeagueForm Class (Part 4)

```

  public ActionErrors validate(ActionMapping mapping,
                               HttpServletRequest request) {
    ActionErrors errors = new ActionErrors();

    // Perform data conversions.
    try {
      this.year = Integer.parseInt(yearStr);
    } catch (NumberFormatException nfe) {
      errors.add(“yearStr”, new ActionError(“error.yearField.required”));
    }

    // Verify form parameters
    if ( (year != -1) && ((year < 2000) || (year > 2010)) ) {
      errors.add(“yearStr”, new ActionError(“error.yearField.range”));
    }
    if ( season.equals(“UNKNOWN”) ) {
      errors.add(“season”, new ActionError(“error.seasonField.required”));
    }
    if ( title.length() == 0 ) {
      errors.add(“title”, new ActionError(“error.titleField.required”));
    }

    // Return the errors list.  An empty list tells Struts that this form
    // passed the verification check.
    return errors;
  }
```

Struts `ActionError` Class

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

```
// Model classes
import sl314.model.LeagueService;
import sl314.model.League;
import sl314.model.ObjectNotFoundException;
// View classes
import sl314.view.AddLeagueForm;


public class AddLeagueAction extends Action {

  public ActionForward execute(ActionMapping mapping, ActionForm form,
                               HttpServletRequest request,
                               HttpServletResponse response) {

    // Use Struts actions to record business processing errors.
    ActionErrors errors = new ActionErrors();
    // Store this set in the request scope, in case we need to
    // send the ErrorPage view.
    saveErrors(request, errors);

```

The AddLeagueAction Class (Part 2)

```
    // Use Struts actions to record business processing errors.
    ActionErrors errors = new ActionErrors();
    // Store this set in the request scope, in case we need to
    // send the ErrorPage view.
    saveErrors(request, errors);

    try {

      // Cast the form to the application-specific action-form class
      AddLeagueForm myForm = (AddLeagueForm) form;

      // Perform business logic
      LeagueService leagueSvc = new LeagueService();
      League league = leagueSvc.createLeague(myForm.getYear(),
                                             myForm.getSeason(),
                                             myForm.getTitle());
      // Store the new league in the request-scope
      request.setAttribute(“league”, league);

      // Send the Success view
      return mapping.findForward(“success”);
```

The AddLeagueAction Class (Part 3)

```

      // Handle any unusual exceptions
      } catch (RuntimeException e) {

        // Log stack trace
        e.printStackTrace(System.err);

        // Record the error
        errors.add(ActionErrors.GLOBAL_ERROR,
                   new ActionError(“error.unexpectedError”,
                                   e.getMessage()));

        // and forward to the error handling page (the form itself)
        return mapping.findForward(“error”);

      } // END of try-catch block

    } // END of execute method

} // END of AddLeagueAction class
```

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

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

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

The add_league.jsp Page

```
<%@ page session="false" %>
<%@ taglib prefix="c"
           uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="html"
           uri="http://jakarta.apache.org/struts/tags-html" %>

<%-- Set page title --%>
<c:set var="pageTitle">Add a New League</c:set>

<%-- Create business services --%>
<jsp:useBean id="leagueSvc" class="sl314.model.LeagueService" />

<%-- Generate the HTML response --%>
<html>
<head>
  <title>Duke's Soccer League: ${pageTitle}</title>
</head>
<body bgcolor='white'>

```

The add_league.jsp Page (Part 2)

```
<html>
<head>
  <title>Duke’s Soccer League: ${pageTitle}</title>
</head>
<body bgcolor=’white’>

<%-- Generate page heading --%>
<!-- Page Heading -->
<table border=’1’ cellpadding=’5’ cellspacing=’0’ width=’400’>
<tr bgcolor=’#CCCCFF’ align=’center’ valign=’center’ height=’20’>
  <td><h3>Duke’s Soccer League: ${pageTitle}</h3></td>
</tr>
</table>

<%-- Report any errors (if any) --%>
<html:errors />

<%-- Generate main body --%>
<p>
This form allows you to create a new soccer league.
</p>
```

The add_league.jsp Page (Part 3)

```
<%-- Generate main body --%>
<p>
This form allows you to create a new soccer league.
</p>
<html:form action="/admin/add_league.do" method="POST"
           focus="yearStr">
<%-- Repopulate the year field --%>
Year: <html:text property="yearStr" />
<br/><br/>
<%-- Repopulate the season drop-down menu --%>
Season:
<html:select property='season'>
<c:forEach var="season" items="${leagueSvc.allSeasons}">
  <html:option value="${season}">${season}</html:option>
</c:forEach>
</html:select>
<br/><br/>
<%-- Repopulate the title field --%>
Title: <html:text property="title" />
<br/><br/>
<%-- The submit button --%>
<html:submit value="Add League" />
</html:form>
```

Configuring the View Forms

Configure the form beans as follows:

- Form beans are configured in the Struts configuration file.

```
<form-beans>
  <form-bean name="selectLeagueForm"
             type="sl314.view.SelectLeagueForm" />
  <form-bean name="enterPlayerForm"
             type="sl314.view.EnterPlayerForm" />
  <form-bean name="selectDivForm"
             type="sl314.view.SelectDivisionForm" />
  <form-bean name="addLeagueForm"
             type="sl314.view.AddLeagueForm" />
</form-beans>
```

- Form beans are named so that 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.

```
<!-- Declare the /admin/add_league.do action -->
<action path="/admin/add_league"
        type="sl314.controller.AddLeagueAction"
        name="addLeagueForm" scope="request" validate="true"
        input="/admin/add_league.jsp" >
  <forward name="success" path="/admin/success.jsp"/>
  <forward name="error" path="/admin/add_league.jsp"/>
</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

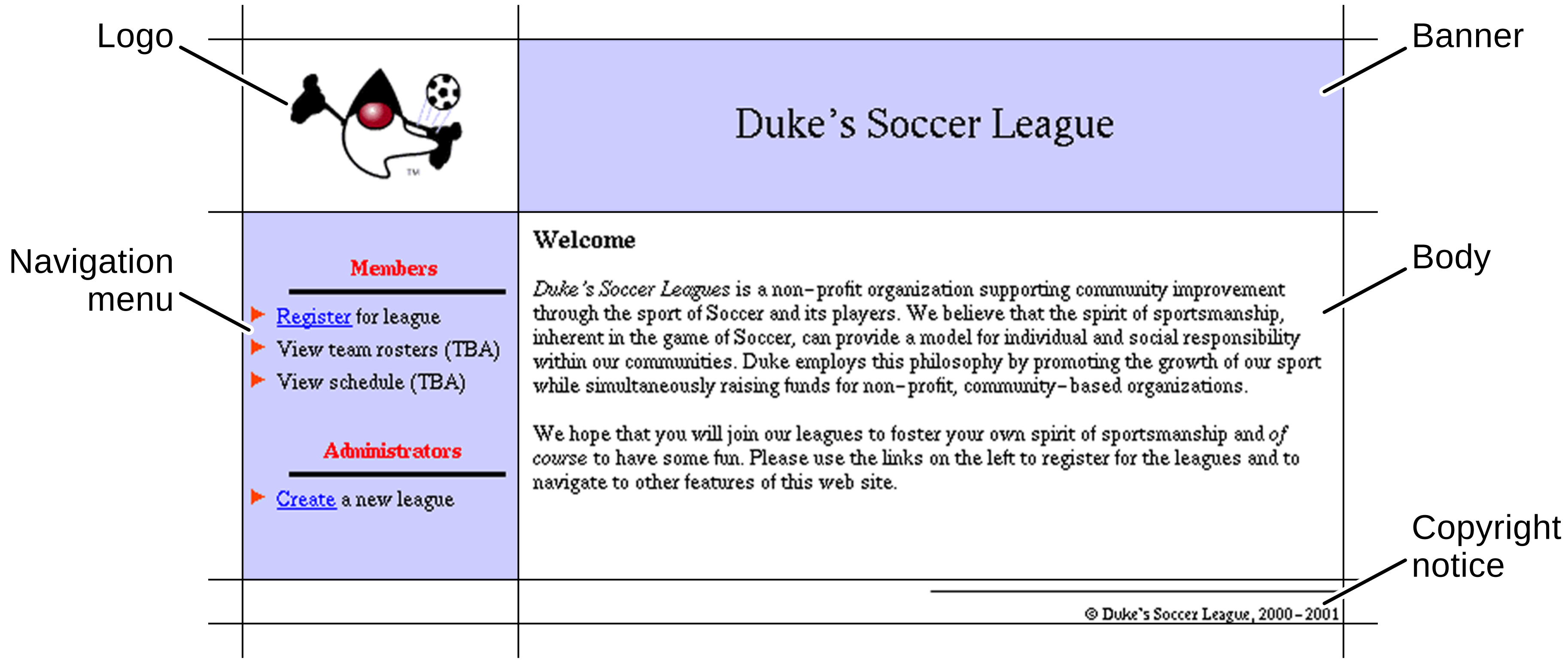
Objectives

- 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



Welcome

Duke's Soccer Leagues is a non-profit organization supporting community improvement through the sport of Soccer and its players. We believe that the spirit of sportsmanship, inherent in the game of Soccer, can provide a model for individual and social responsibility within our communities. Duke employs this philosophy by promoting the growth of our sport while simultaneously raising funds for non-profit, community-based organizations.

We hope that you will join our leagues to foster your own spirit of sportsmanship and *of course* to have some fun. Please use the links on the left to register for the leagues and to navigate to other features of this web site.

© Duke's Soccer League, 2000-2001

Complex Page Layouts (continued)

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

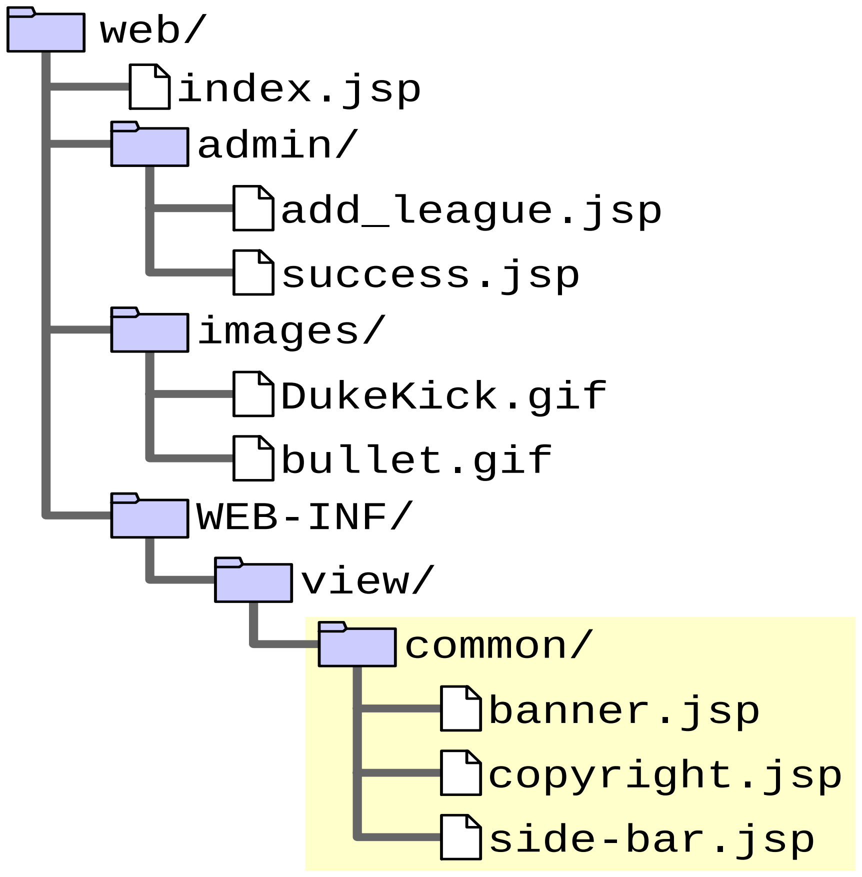
A segment can be any text file that contains static HTML or dynamic JSP technology code:

```
<%@ taglib prefix="myTags" uri="/WEB-INF/myTags.tld" %>

    <spacer height='15'>
    <hr width='50%' align='right' size='1' noshade color='blue'>
    <font size='2' face='Helvetica, san-serif'>
    &copy; Duke's Soccer League, 2000-<myTags:getCurrentYear />
    </font>
```

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

Organizing Presentation Segments

You should isolate your reusable segments.



Organizing Presentation Segments (continued)

- 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.

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:

```
<!-- START of copyright notice -->
<td align=’right’ width=’480’>
  <%@ include file=”/WEB-INF/view/common/copyright.jsp” %>
</td>
<!-- 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.

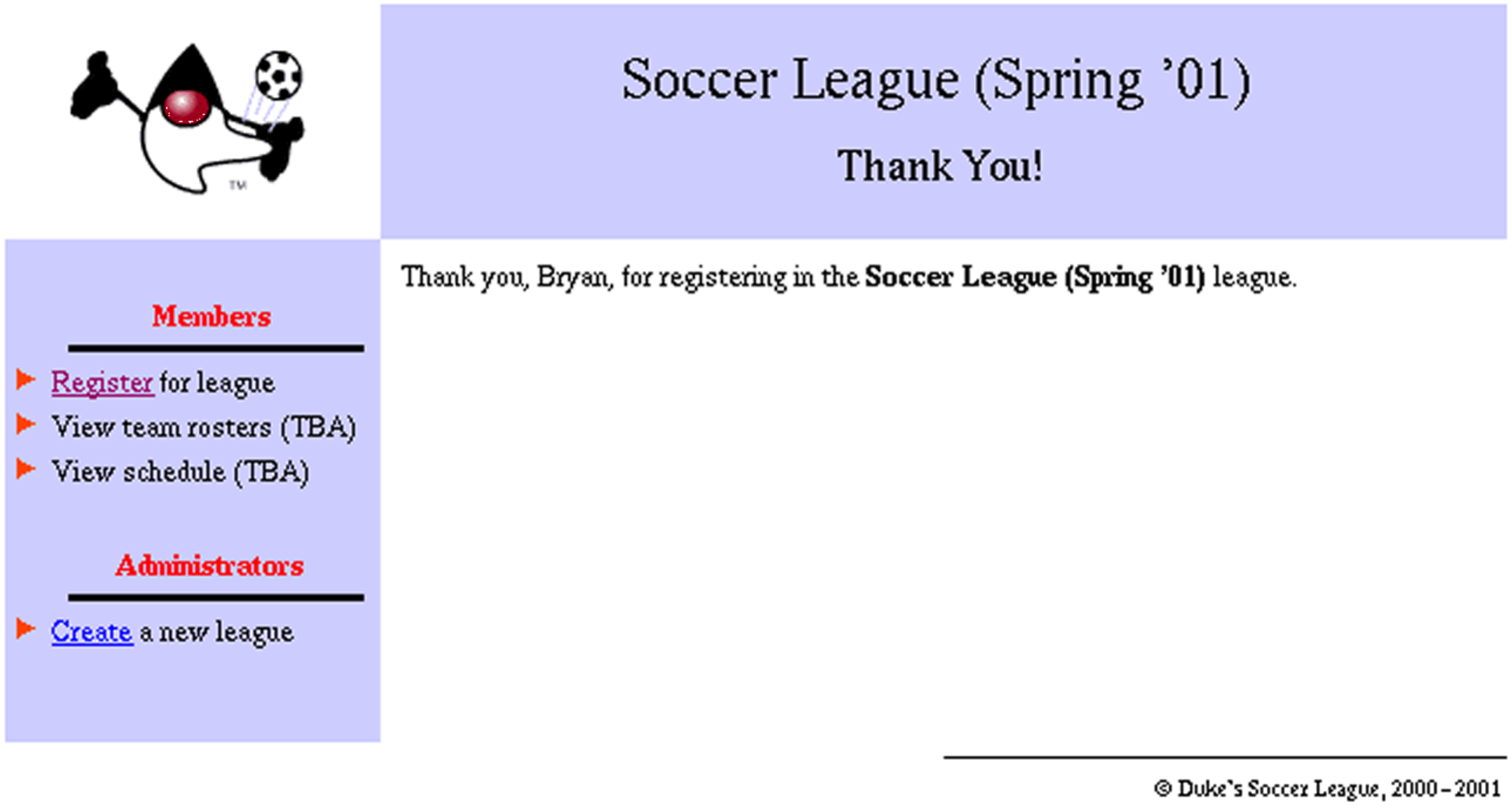
- Syntax:

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

- Example:

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

© Duke's Soccer League, 2000 – 2001

Using the `jsp:param` Standard Action (continued)

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

For example, in the Soccer League home page:

```
<!-- START of banner -->
<jsp:include page=”/WEB-INF/view/common/banner.jsp”>
  <jsp:param name=”subTitle” value=”Welcome” />
</jsp:include>
<!-- END of banner -->
```

Using the `jsp:param` Standard Action (continued)

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

```

<font size='5' face='Helvetica, san-serif'>
${bannerTitle}
</font>

<c:if test="${not empty param.subTitle}">
<br/><br/>
<font size='4' face='Helvetica, san-serif'>
${param.subTitle}
</font>
</c:if>
```

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

```
<%@ taglib prefix=”tiles”
           uri=”http://jakarta.apache.org/struts/tags-tiles” %>
<%@ taglib prefix=”c”
           uri=”http://java.sun.com/jsp/jstl/core” %>

<%-- Generate the HTML response --%>
<html>

<head>
<title>Duke’s Soccer League: <tiles:getAsString name=”subTitle”/></title>
</head>

<body bgcolor=’white’>

```

The `layoutPage.jsp` Page (Part 2)

```

<table border=’0’ cellspacing=’0’ cellpadding=’0’ width=’640’>

<tr height=’100’>

  <td align=’center’ valign=’center’ width=’160’ height=’100’>
    <img src=’<c:url value=”/images/DukeKick.gif”/>’
         alt=’Duke’s Soccer League Logo’>
  </td>

  <td bgcolor=’#CCCCFF’ align=’center’ valign=’center’ width=’480’ height=’100’>
    <!-- START of banner -->
    <c:set var=”subTitle”><tiles:getAsString name=’subTitle’ /></c:set>
    <jsp:include page=”/WEB-INF/view/common/banner.jsp”>
      <jsp:param name=”subTitle” value=”${subTitle}” />
    </jsp:include>
    <!-- END of banner -->
  </td>

</tr>
```

The `layoutPage.jsp` Page (Part 3)

```
<tr valign='top'>

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

  <td width='480' align='left'>
    <div style='margin-top: 0.1in; margin-left: 0.1in;
                margin-bottom: 0.1in; margin-right: 0.1in'>
    <!-- START of main content -->
    <tiles:insert attribute='body' />
    <!-- END of main content -->
    </div>
  </td>

</tr>
```

The `layoutPage.jsp` Page (Part 4)

```
<tr>

  <td width='160'>
    <!-- nothing here -->
  </td>

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

</tr>

</table>

</body>
</html>
```

Tiles Layout

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

For example, the Registration Thank You page is:

```
<%@ taglib prefix="tiles"
           uri="http://jakarta.apache.org/struts/tags-tiles" %>

<tiles:insert page="/WEB-INF/view/layout/layoutPage.jsp">
  <tiles:put name="subTitle" value="Thank You"/>
  <tiles:put name="body" value="/WEB-INF/view/register/thank_you.jsp"/>
</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.

```
<%@ page session=”true” %>

<p>
Thank you, ${sessionScope.player.name}, for registering for
the <i>${sessionScope.league.title}</i> league.
</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.

Web Component Development With Servlet and JSP™ Technologies
SL-314
Revision C

The End