Module 3 Content

Concept of HTTP Protocol and Web Containers

Originally Servlets were intended to work with any server. However, practically, Servlets are used only with web servers. Therefore, Servlets typically respond to HTTP requests.

HTTP

- HTTP is a very simple and lightweight protocol.
- The client is the one that always initiates a request. The server can never make a callback connection to the client.
- The HTTP requires a client to establish a connection prior to each request and a server to close the connection after sending the response. Also, either a client or a server can prematurely terminate a connection.
- Originally HTTP was developed to serve static information and is stateless by its very nature.
- In modern times we often want to build web applications that generate dynamic content responses to client's requests. This requirement calls for a significant change in the way we design application components and specifically servlets. We want servlets to maintain persistence, meaning to be able to store a state of the servlet object across multiple requests from a client.
- HTTP defines certain types of requests that a client can send to servers. The following is a list of request methods in HTTP/1.1:

  GET, POST, HEAD, OPTIONS, PUT, TRACE, DELETE and CONNECT.

  The first two of them, GET and POST, are the most commonly used.

  - The GET request method, usually accompanied by parameters, can be used to retrieve static or dynamic information, for example:

    http://www.apl.jhu.edu/Java_EE_course/get_example

    or

    http://www.apl.jhu.edu/Java_EE_course/get_example?firstname=joe&lastname=doe

    In this example, **Universal Resource Identifier (URI)** specifies a resource name, such as **/Java_EE_course/get_example**, that follows the URL, **www.apl.jhu.edu**.

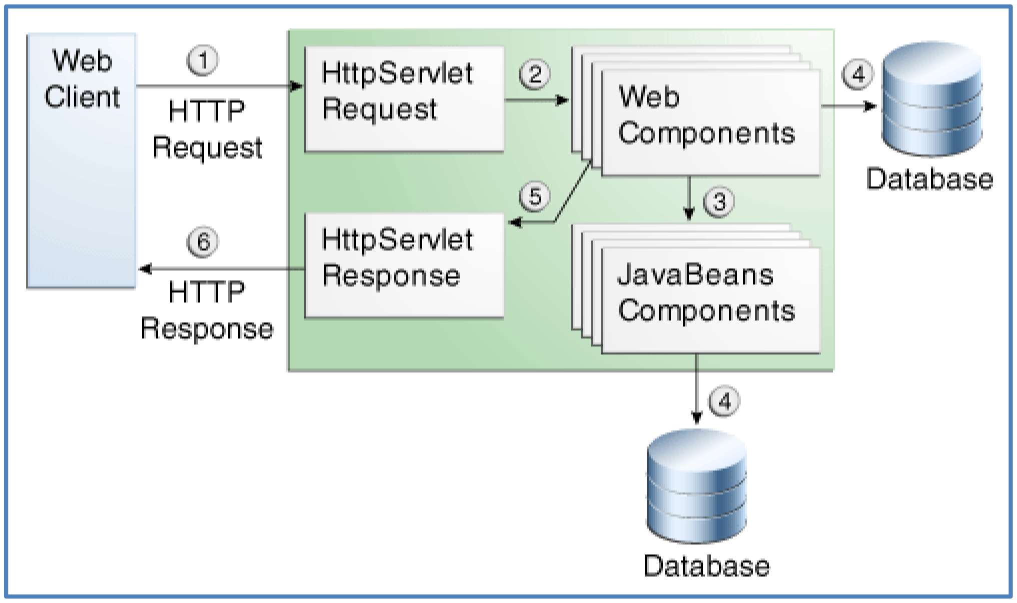
    The web server can process two parameters, firstname and lastname, with values "joe" and "doe" respectively and send a response back to the browser. Notice that with the GET method, request parameters are transmitted as a query string appended to the requested URL. Sometimes this has bad ramifications because of sensitivity of the information. Also some servers might have a restriction on a length of input information that can be transferred this way.

  - The POST request method is commonly used for accessing dynamic resources. It allows transmitting large amounts of input information to the server. It allows encapsulating the multi-part messages into the request body.
  - Along with its response (as a response body content) the server sends back a status and some meta-information describing the response embedded into a response header. This

meta-information could be helpful controlling the behavior of the browser, like "Content-Type", "Expires", "Date", etc.

Web Containers

- The web container is a hosting facility for the Java Servlets, the Java Server Pages (JSPs), and other web presentation tier software components of the web-enabled applications. It also supports deployment processes, which include installation and customization of run-time behavior of the components.
- There are several types of web containers:
    - web container in a separate runtime. This is an old pre-Java EE style "servlet engine" with the web server having plug-in extensions to communicate to the servlets environment;
    - web container built into web servers. An example is Sun's Java WebServer, Jakarta Tomcat implementation (from http://jakarta.apache.org);
    - web container as a part of a Java EE application server, such as Oracle's WebLogic Server or IBM's WebSphere application server.
- A web container provides such services as request dispatching, security, concurrency, and lifecycle management.
- A web container also gives web components access to such APIs as naming, transactions, and email.
- Every web application has to be installed, or *deployed*, on the web container. Deployment package provides configuration information that can be specified using Java EE annotations or can be maintained in a text file in XML format called a web application deployment descriptor (DD). A web application DD must conform to the schema described in the Java Servlet specification.
- The following figure represents typical web application request processing:



Adopted from Oracle's Java EE 7 Tutorial

1) The Web client sends an HTTP request to the web server.
2) A web server that implements Java Servlet and JavaServer Pages technology converts the request into an HTTPServletRequest object.

3) This object is delivered to a web component, which can interact with JavaBeans components or 4) a database to generate dynamic content.
5) The web component can then generate an HTTPServletResponse or can pass the request to another web component. A web component eventually generates a HTTPServletResponse object.
6) The web server converts this object to an HTTP response and returns it to the client.

Java Servlets

Basic Java Servlet API and Programming Techniques

- As a reference, see the latest Sun's Java Servlet API in The Java Servlet Technology: http://www.oracle.com/technetwork/java/index-jsp-135475.html.
- The API includes two packages:
  - javax.servlet - protocol independent,
  - javax.servelt.http - contains classes and interfaces specific to HTTP protocol.

  In these two packages, a set of interfaces and classes serves its main purposes:

  - creating an environment to manage the lifecycle of servlets,
  - supporting application logic based on a request/response paradigm,
  - dealing with exceptions,
  - managing an interaction between servlets and an environment.
- The typical order of steps in a servlet algorithm is as follows:
  - to receive a request object,
  - to extract any input parameters,
  - to process any application logic,
  - to conduct session tracking and handling,
  - to generate the response.
- When writing a servlet we need to either directly or indirectly implement a javax.servlet, a servlet interface in order for a web container to communicate with our servlet. Most likely you will extend one of two classes: javax.servlet.GenericServelt or javax.servlet.http.HttpServlet.
- The javax.servelt.Servlet interface enforces the five following methods:

public void init (ServletConfig config)	Called once when servlet instance is instantiated
public void service (ServletRequest request, ServletResponse response)	Called to handle a request
public void destroy ()	Called when web container is ready to remove a servlet instance out of service
public ServletConfig getServletConfig ()	To return the ServletConfig object that was passed to the servlet during the init () method
public String getServletInfo ()	For returning a String object containing Information about the servlet

  - All servlet methods from the servlet interface will be called at specific times and in a specific order during the servlet lifecycle.
  - The simplest way to write a servlet is to extend an abstract class javax.servlet.http.HttpServlet. Then all we need to do is to provide an implementation to some of its methods which would be used in our concrete servlet class:
    - service,

    - doGet,
    - doPost,
    - doDelete,
    - doOptions,
    - doPut,
    - doTrace.
- The javax.servlet.ServletConfig object represents the configuration of a servlet. It contains initialization parameters (as a set of name/value pairs), the name of the servlet,and information about the container.
- To obtain a reference to the ServletConfig object we can do:

```
public init (ServletConfig config) {
super.init (config);
}
```

or

```
getServletConfig ()
```

- To understand handling of a request, let's consider the following HTML document containing an input form:

```
<form action="/servlet/registration" method="post">
 <table>
  <tr>
   <td>First Name:</td>
   <td><input type="text" name="firstName" size="15" /></td>
   <td>Last Name:</td>
   <td><input type="text" name="lastName" size="15" /></td>
  </tr>
 </table>
 <input type="submit" name="submit" value="Submit Request" />
</form>
```

To obtain "firstName" parameter in a servlet we can do:

String firstName = request.getParameter("firstName");

To obtain "lastName" parameter in a servlet:

String lastName = request.getParameter("lastName");

We can also use:

public String[ ] getParameterValues (String key)

or

public Enumeration getParameterNames ()

- To construct a response we use an interface HttpServletResponse with some of the following methods:

addCookie (Cookie cookie)

setContentType (String type)

getWriter ()

setStatus (int statusCode)

sendError (int status)

Java Servlets Session Tracking

- As you know, HTTP is a **"stateless"** protocol. Each client request gets processed by a server as a fresh request without any connection to the previous request from the same user and even from the same session of a given user. This lack of context causes a number of difficulties because in reality web applications are much more complicated; they do require session tracking capabilities. For example, we might want to maintain the information about a client and a session across multiple requests and we have to do it on the server side. The Java Servlet API's session tracking mechanism allows us to handle this requirement.
- **Session** facility identifies a series of requests from a single client to form a single "working" session.
- **State** facility remembers information related to previous requests and other business decisions that are made for requests. That is, the application should be able to associate state with each session.
- There are several approaches to session tracking:
  - Cookies
  - URL rewriting
  - Hidden form fields

All of them differ in implementation details, but are similar in terms of using some kind of information exchange between the server and the client.

- **Cookies** are small bits of textual data that a Web Server sends to a browser and that the browser returns unchanged when later visiting the same Web Server. While using cookies, we can achieve user identification within a given session, or avoid user authentication multiple times within a session, or a website customization. However, cookies have some problems. For example, they might present some level of privacy threat. That's why some customers disable cookie handling by their web browsers. The Servlet Cookie API includes the capabilities of creating cookies, managing cookies attributes, placing cookies in the response headers, and reading cookies from the client.
- **URL-rewriting** allows the clients to append some extra data on the end of each URL that identifies the session, and the server associates that identifier with data it has stored about that session. For example, with ***http://host:port/path/file.html;jsession=9876*** the session information is attached as jsession=9876. This is also an excellent solution and even has the advantage of working when browsers don't support cookies or when the user has disabled them. However, this approach has the same problems as cookies, the server-side program has a lot of straightforward, but tedious, processing to do. It may even break when the user leaves the session and later comes back via a bookmark or link.
- **Hidden form fields** allow the server program (for example, a Java servlet) to include additional invisible (hidden) fields with certain values into an HTML form and send it back to the client, like:

  <input type="hidden" name="MyHiddenFieldID" value="something" />

When this HTML form is submitted, the specified fields with known names and values will be included in the GET and POST data.

Java Servlets technology provides its own an outstanding technical approach for session tracking: the ***HttpSession*** API. This high-level interface is built on top of cookies or URL-rewriting. In fact, most servers use cookies if the browser supports them, but automatically revert to URL-rewriting when cookies are unsupported or explicitly disabled. However, because this is high-level abstraction, a servlet programmer doesn't need to know an/or bother with many of the details, and is automatically given a convenient place to store arbitrary objects that are associated with each session.

The following online Java EE tutorial provides detailed guidance of how to use the HttpSession object and examples of Java Servlet coding: Java Servlet Technology. Maintaining Client State, see http://docs.oracle.com/javaee/6/tutorial/doc/bnagm.html#bnagn. Students can use it as a reference document

Web Applications and Java Servlets Deployment, Standard Directory Structure

Since Java Servlet API Release 2.2, there is one feature that significantly changed the way the server-side presentation components get set up for run-time. This feature is known as "***web application***".

A ***web application***, as defined in the Java Servlet specification, is a collection of servlets, Java Server Pages (JSPs), JavaServer Faces, HTML documents, images, and other Web resources that are set up in such a way as to be portably deployed across any servlet-enabled Web server.

The goal of having web applications defined as one unit is to standardize a deployment process across different web containers. So, installation of a Web application is simple.

With web applications, the entire application can be contained in a single archive file and deployed by placing the file into a specific directory.

Web applications archive files have the extension **.war**, which stands for **w**eb **a**pplication **a**rchive.

War files are actually jar files (created using the jar utility) saved with an alternate extension. Using the jar format allows jar files to be stored in compressed form and have their contents digitally signed. The .war file extension was chosen over .jar to let people and tools know to treat them differently.

Inside a war file you might find a file listing like this:

- index.html
- myHowTo.jsp
- myRegistration.jsp
- images/banner.gif
- images/another_image.gif
- WEB-INF/web.xml
- WEB-INF/lib/jspbean.jar
- WEB-INF/classes/myServlet1.class
- WEB-INF/classes/myServlet2.class

During installation, a war file can be mapped to any URI prefix path on the server. The war file then handles all requests beginning with that prefix. For example, if the war file above were installed under the prefix /myApp, the server would use it to handle all requests beginning with /myApp. A request for

/myApp/index.html would serve the index.html file from the war file. A request for /myApp/howto.jsp or /myApp/images/banner.gif would also serve content from the war file.

In general, according to the Java Servlet spec and Oracle's WLS implementation, the directory structure layout is as follows:

```
    myWebApp/(publicly available files, such as .jsp, .html, images, style sheets, and other regular
    Web content)
            |                                                              |
            +WEB-INF/
                    |
                    + web.xml (mandatory file)
                    |
                    + weblogic.xml (optional file)       |
                    + classes/ (directory containing Java classes including
                     |            servlets used by the web application)
                     |
                    + lib/ (directory containing jar files used by the web application)
```

Note that we can also deploy web application using **WAR** files instead of regular directories. A WAR file is just a JAR file (which is just a ZIP file) with a *.war* extension. We can create WAR files using jar, or WinZip.

In case of using WAR files, a directory such as *myWebApp* should become *myWebApp.war*, and the top-level directory within the WAR file should be *WEB-INF* (i.e., do not repeat *myWebApp* within the WAR file).

For more information please read Creating and Configuring Web Applications at http://docs.oracle.com/middleware/1221/wls/WBAPP/configurewebapp.htm about web application deployment in the WLS, web.xml and weblogic.xml files.

> **Note**: weblogic.xml file is a proprietary WLS configuration file and is not a mandatory part of standard web application structure. So, we always have to apply its usage with peace of mind.

Web Applications, Deployment Process

So, say you are finished developing your web application code and you are ready to deploy it on the WebLogic Server domain.

The following are the steps for deploying a web application on WebLogic Server.

- Start up your WLS domain.
- Start up the web browser based WLS Admin Console.
- Login to the WLS domain Admin Server from Admin Console.
- Find "Deployments" link on the left hand side panel and click on it.
- Within "Deployments" section of the panel on the right, you should see "Install" button enabled.
- Click on "Install" button.
- Navigate thru your local file system (on the right panel, "Location" section) in order to locate a directory of the web application that you would like to deploy.
- Once you get to see your web application, select a radio button next to it and click on "Next" button.

- On the next screen, under "Choose targeting style" section, leave "Install this deployment as an application" option selected (by default) and click on "Next" button.
- Leave all options as default on the next screen "Optional Settings" and click on "Finish" button.
- After getting "Summary of Deployments" screen back, click on "Checkbox" to the left from the web application entry on "Deployments" panel in order to select it.
- Click on "Start" button on the Deployment panel, and then click on "Servicing all the requests" option.
- You should get "Deployments" panel back with your web application in it listed in "Active" state.

Now your web application is ready to run.

"HelloWorld" Servlet Development, Packaging and Deployment Process

Let's see how we can develop, package, deploy and run the simplest web application, "HelloWorld", using Java Servlet based approach. This example will demonstrate a step-by-step process, including some available alternative options on each step. We are going to start with a bare servlet that accepts a few initialization attributes and then prints out a simple HTML web page back to a client. Then we will demonstrate the same Java Servlet with using Java annotations. Finally, we will modify this servlet to be able to accept and to process a few HTML FORM parameters provided by the user.

In addition to this section of the module, I encourage you to follow description of similar process located at

http://docs.oracle.com/middleware/1221/wls/WBAPP/configureservlet.htm#WBAPP135

1. First off, let's create a directory called "myServletApp" located let's say on the root directory of C drive, and then, within myServletApp directory, we will create the following directory structure:

```
myServletApp/
        |
        +WEB-INF/
                |
                + web.xml
                |
                + classes/
         |
         +src
```

   Note: "src" directory will be used to maintain Java source code, but it will not be included into war file while packaging ready-for-deployment application.

2. Now, let's create a Java Servlet called HelloWorldServlet with the following code.

```
package edu.jhu.jee.lf;

import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
```

```
7   import javax.servlet.http.HttpServletResponse;
8   import java.io.IOException;
9   import java.io.PrintWriter;
10
11  public class HelloWorldServlet extends HttpServlet {
12    private String l_name_;
13    private String f_name_;
14
15    public void init(ServletConfig servletConfig) throws ServletException {
16      super.init(servletConfig);
17      l_name_ = getInitParameter("last_name");
18      f_name_ = getInitParameter("first_name");
19    }
20
21    public void doPost(HttpServletRequest req, HttpServletResponse res)
22        throws ServletException, IOException {
23      this.doGet(req, res);
24    }
25
26    public void doGet(HttpServletRequest req, HttpServletResponse res)
27        throws ServletException, IOException {
28
29      // set content type and other response header fields first
30      res.setContentType("text/html");
31
32      // then write the data of the response
33      PrintWriter out = res.getWriter();
34      out.println("<br>Hello world! This example is provided by: " + l_name_ + " " + f_name_);
35    }
36
37    public String getServletInfo() {
38      return "A simple servlet";
39    }
40  }
```

Notice that HelloWorldServlet extends HttpServlet.

We can also see (lines 15-19) how this servlet accesses the values of two initialization attributes, “last_name” and “first_name”, during execution of its **init** method:

```
l_name_ = getInitParameter("last_name");
f_name_ = getInitParameter("first_name");
```

Later, in its doGet I() method (lines 26-35), the servlet prints out a message that includes previously obtained values of two initailization attributes.

3. Next step is to compile this servlet and to save its compiled class in the myServletApp/WEB-INF/classes directory.

   Note: In order to successfully compile Java EE classes, we need to make sure to set up CLASSPATH to include all necessary Java EE libraries (jar files). WebLogic Server conveniently provides a script called setDomainEnv.cmd, located in C:\Oracle\Middleware\user_projects\domains\lfDomain\bin directory. So, simple execute this script and you will get CLASSPATH set up.

   Once you have CLASSPATH set up, you can compile the HelloWorldServlet by executing the following command from C:\myServletApp\src directory:

   ```
   javac -d ./WEB-INF/classes *.java
   ```

4. Next step is to define **web.xml** configuration file as follows.

```
<?xml version="1.0" encoding="UTF-8"?>

<web-app version="2.4" 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">

<servlet>
    <servlet-name>HelloWorld</servlet-name>
    <servlet-class>edu.jhu.jee.lf.HelloWorldServlet</servlet-class>

    <init-param>
       <param-name>first_name</param-name>
       <param-value>LEONID</param-value>
    </init-param>

    <init-param>
       <param-name>last_name</param-name>
       <param-value>FELIKSON</param-value>
    </init-param>

</servlet>

<servlet-mapping>
      <servlet-name>HelloWorld</servlet-name>
      <url-pattern>/jhu/*</url-pattern>
</servlet-mapping>

</web-app>
```

In this file, we define initialization attributes for HelloWorldServlet in the init-param element of the servlet element, using **param-name** and **param-value** tags**.**

Once again, it is required that the web.xml file has to be located in the WEB-INF directory of the web application.

Also, be sure to understand how this servlet is identified for execution. The URL to be used to call a servlet is determined by:

- The name of the Web application containing the servlet and
- The name of the servlet as mapped in the deployment descriptor of the Web application. Request parameters can also be included in the URL used to call a servlet.

Generally the URL for a servlet conforms to the following format:

http://host:port/webApplicationName/mappedServletName?parameter

The components of the URL are defined as follows:

- ***host*** is the name of the machine running WebLogic Server.
- ***port*** is the port at which the above machine is listening for HTTP requests.
- ***webApplicationName*** is the name of the Web application containing the servlet.
- ***parameters*** are one or more name-value pairs containing information sent from the browser that can be used in your servlet.

For example, to use a Web browser to call the HelloWorldServlet, enter the following URL:

http://localhost:7001/myServletApp/jhu

5. Now we can package entire web application into war file, although it is permissible to deploy web application in “as-is”, exploded form, without packaging it into war format. So, from C:\myServletApp directory, simply execute the following command:

   ```
   jar cfv myServletApp.war WEB_INF
   ```

6. Deploy myServletApp web application on WebLogic Server, using Admin web browser utility.

7. Execute it by invoking it from web browser with the following URL:

   http://localhost:7001/myServletApp/jhu

8. Update HelloWorldServlet with use of Java annotations.

- The Java Servlet 3.0 specification has offered several new features that are aimed at easing the development of servlet applications and will benefit both servlet developers and framework developers.
- Specifically, The Java Servlet 3.0 API focuses on ease of development by making use of JSR 175 *annotations* to enable declarative-style programming. This means that we can swiftly develop a servlet or a filter class by simply annotating the class with appropriate annotations like @Servlet or @ServletFilter.
- Annotations not only make the coding of servlet, filter, and listener classes easier, but also make deployment descriptors optional for a web application, even though the application archive may have servlet, filter, or context listener classes.

- The web container is responsible for processing the annotations located in classes in the *WEB-INF/classes* directory, in a .jar file located in the *WEB-INF/lib* directory, or in classes found elsewhere in the application's CLASSPATH.
- With Java EE metadata annotations, the standard web.xml deployment descriptor is optional.
- While comparing use of annotations and use of "legacy" XML based deployment descriptors, it is interesting to note that the deployment descriptor takes precedence over annotations. In other words, the deployment descriptor overrides configuration information specified through the annotation mechanism.
- Java Servlet 3.0 contains a new attribute called "**metadata-complete**" on the web-app element of the web deployment descriptor. This attribute defines whether the web descriptor is complete, or whether the class files of the web application should be examined for annotations that specify deployment information. If the attribute is set to true, the deployment tool must ignore any servlet annotations present in the class files and use only the configuration details mentioned in the descriptor. Otherwise, if the value is not specified or set to false, the container must scan all class files of the application for annotations. This provides a way to enable or disable scanning of the annotation and its processing during the startup of the application.
- All annotations introduced in Servlet 3.0 can be found under the packages
    - javax.servlet.http.annotation
    - javax.servlet.http.annotation.jaxrs
- The servlet specification states annotations can be defined on certain Web components, such as servlets, filters, listeners, and tag handlers. The annotations are used to declare dependencies on external resources. The container will detect annotations on such components and inject necessary dependencies before the component's life cycle methods are invoked.
- See WebLogic Annotation for Web Components document at

  http://docs.oracle.com/middleware/1221/wls/WBAPP/annotateservlet.htm

- Settings in web.xml, can also be used to override the settings provided by the annotations. The web.xml takes precedence. Again, we can also set metadata-complete to true in web.xml to tell the container only to consider configuration in web.xml and skip annotations altogether.
- Let's update our HelloWorldServlet to start using Java annotations.

```
package edu.jhu.jee.lf;

import weblogic.servlet.annotation.WLInitParam;
import weblogic.servlet.annotation.WLServlet;
import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.io.PrintWriter;

@WLServlet(name = "HelloWorldServlet",
    mapping = "/jhu",
    initParams = {
```

```
                    @WLInitParam(name = "last_name", value = "Felikson"),
                    @WLInitParam(name = "first_name", value = "Leonid")
            }
  )
public class HelloWorldServlet extends HttpServlet {
  private String l_name_;
  private String f_name_;

  public void init(ServletConfig servletConfig) throws ServletException {
    super.init(servletConfig);
    l_name_ = getInitParameter("last_name");
    f_name_ = getInitParameter("first_name");
  }

  public void doPost(HttpServletRequest req, HttpServletResponse res)
            throws ServletException, IOException {
            this.doGet(req, res);
  }

  public void doGet(HttpServletRequest req, HttpServletResponse res)
            throws ServletException, IOException {

    // set content type and other response header fields first
            res.setContentType("text/html");

    // then write the data of the response
            PrintWriter out = res.getWriter();
            out.println("<br>Hello world! This example is provided by: " + l_name_ + " " + f_name_);
  }

  public String getServletInfo() {
            return "A simple servlet";
  }
}
```

Here you can see use of @WLServlet annotation (weblogic.servlet.annotation.WLServlet), as follows:

```
@WLServlet (name = "HelloWorldServlet",
       mapping = "/jhu",
       initParams = {
              @WLInitParam(name = "last_name", value = "Felikson"),
                @WLInitParam(name = "first_name", value = "Leonid")
        }
  )
```

This annotation declares HelloWorldServlet class as Java Servlet. It also provides URL mapping rules, as well as it includes two initialization attributes with names ‘last_name” and “first_name” and their respective values. Of course, in order to successfully compile this annotation, we need to import both annotation classes, **weblogic.servlet.annotation.WLInitParam** and **weblogic.servlet.annotation.WLServlet.**

@WLServlet annotation defines various attributes for declaring parameters for the servlet. All attributes on this annotation are optional. The following is a list of all attributes of @WLServlet annotation.

Name	**Description**	**Data Type**	**Required?**
displayName	Display name for the servlet after deployment	String	No
Description	Servlet description	String	No
Icon	Icon location	String	No
Name	Servlet name	String	No
initParams	Initialization parameters for the servlet	WLInitParam[]	No
loadOnStartup	Whether the servlet should load on startup	int	No
runAs	The run-as user for the servlet	String	No
Mapping	The url-pattern for the servlet	String[]	No

- Since we are using Java annotations here, there is no need to keep web.xml configuration file. So, when you are ready to package this application into war file, you can safely remove web.xml from WEB-INF directory. However, you still need to keep WEB-INF/classes directory for compiled HelloWorldServlet class.

- **Note:** Be aware that starting from WebLogic Server version 12c, Oracle has deprecated WebLogic Server-specific annotations:

  @WLServlet, @WLFilter, @WLInitParam

  in favor of the standard annotations defined in the servlet 3.0 specification. We can still use WebLogic Servlet-specific annotations (located in **weblogic.servlet.annotation** package), or we can use Java Servlet 3.0 standard annotations (located in **javax.servlet.annotation** package).

  @WebServlet annotation is used to register a Servlet with a container. Below is the complete list of attributes that annotation encapsulates.

  - name
  - description
  - value
  - urlPatterns
  - initParams

- loadOnStartup
- asyncSupported
- smallIcon
- largeIcon

The attributes are simply bits of data that prior to Servlet 3.0 would go into <servlet> or <servlet-mapping> tags of web.xml. Now they can be specified in Java code.

The **@WebInitParam** annotation provides a means of specifying initialization parameters for servlets and filters. The annotation defines the following attributes:

- name
- value
- description

The **@WebFilter** annotation has the following attributes

- filterName
- description
- displayName
- initParams
- servletNames
- value
- urlPatterns
- dispatcherTypes
- asyncSupported

9. Now, let's review an example of using HTTM FORM parameters, provided as user input to HelloWorldServlet.

   - First off, let's create HTML page called **myFORM.html** that will using FORM tags as follows.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<body>
        <form action="jhu/hello" method="POST">
                <h1>Input Form</h1>
                        <table>
                                <tr>
                                <td>First Name:</td>
                                <td><input type="text" name="first_name"/></td>
                                </tr>
                                <tr>
                                <td>Last Name:</td>
                                <td><input type="text" name="last_name"/></td>
                                </tr>
                                <tr>
                                <td class="field"><input type="submit" name="submit"
                                value="Continue"/></td>
                                <td></td>
                                </tr>
                        </table>
        </form>
```

```
</body>
</html>
```

- In this myFORM.html page, we define FORM tag with **action** attribute that will call on execution of **jhu/hello** web resource. This resource is mapped to HelloWorldServlet (either in web.xml configuration file, or by using Java annotation):

```
<?xml version="1.0" encoding="UTF-8"?>

<web-app version="2.4" 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">

<servlet>
    <servlet-name>HelloWorld</servlet-name>
    <servlet-class>edu.jhu.jee.lf.HelloWorldServlet</servlet-class>
</servlet>

<servlet-mapping>
      <servlet-name>HelloWorld</servlet-name>
      <url-pattern>/jhu/hello</url-pattern>
</servlet-mapping>

</web-app>
```

- We are going to save myFORM.html file in the myServletApp web application root directory as follows:

```
myServletApp/
             |
             myFORM.html
             |
             +WEB-INF/
                     |
                     + web.xml
                     |
                    + classes/
             |
             +src
```

- Now, we can modify HelloWorldServlet code to include accessing FORM based input parameters. Input parameters are always sent in *name=value* pairs, and are accessed through the HttpServletRequest object. You can obtain an Enumeration of all parameter names in a query, and fetch each parameter value by using its parameter name. A parameter usually has only one value, but it can also hold an array of values. Parameter values are always interpreted as Strings, so you may need to cast them to a more appropriate type.

```
package edu.jhu.jee.lf;

import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.io.PrintWriter;

import java.util.Enumeration;

public class HelloWorldServlet extends HttpServlet {
  private String l_name_;
  private String f_name_;

  public void init(ServletConfig servletConfig) throws ServletException {
    super.init(servletConfig);
  }

  public void doPost(HttpServletRequest req, HttpServletResponse res)
      throws ServletException, IOException {
    this.doGet(req, res);
  }

  public void doGet(HttpServletRequest req, HttpServletResponse res)
      throws ServletException, IOException {

    // set content type and other response header fields first
    res.setContentType("text/html");

    // then write the data of the response
    PrintWriter out = res.getWriter();
    Enumeration params = req.getParameterNames();
    String paramName = null;
    String[] paramValues = null;

    while (params.hasMoreElements()) {
        paramName = (String) params.nextElement();
        paramValues = req.getParameterValues(paramName);
        out.println("\nParameter name is " + paramName);
        for (int i = 0; i < paramValues.length; i++) {
            out.println(", value " + i + " is " +
            paramValues[i].toString());
         }
     }
}
```

```
   public String getServletInfo() {
     return "A simple servlet";
   }
}
```

- After compiling HelloWorldServlet class and packaging it (along with web.xml) into war file, we can deploy it on the WebLogic Server.

Note: Be aware that in order to redeploy it, all we need to do is to update existing application. Simple go to WebLogic Admin console, check myServletApp on the check box, and click on "Update" button, and then on "Finish" button.

- Now, we are ready to execute myServletApp application. For that, type in the following URL on your web browser window:

http://localhost:7001/myServletApp/myForm.html

- Now, simply fill in the HTML form and click on Continue button.
- Next page will be generated by HellowWorldServlet and it will look like:

```
Parameter name is submit , value 0 is Continue
Parameter name is first_name , value 0 is LEONID
Parameter name is last_name , value 0 is FELIKSON
```

10. So, in previously described HelloWorldServlet example, we have demonstrated step-by-step process of developing, compiling, packaging, deploying and executing the web application. We have seen ways of how to accept Java Servlet initialization attributes, how to use Java annotations, and finally, how to accept and to process a few HTML FORM parameters provided by the user.

    Now, let's talk about JavaServer Pages technology.

JavaServer Pages (JSPs)

- As it was stated in the "Overview" for this session, **JavaServer Pages** - JSP, for short - is a Java-based technology that simplifies the process of developing the dynamic content of a response before sending it back to the client.
- We can think of JSP as a type of server-side scripting language although it operates quite differently behind the scenes. JavaServer Pages are text files, usually with extension ".jsp", that take the place of traditional HTML pages. JSP files contain traditional HTML along with embedded code that allows the page developer to access data from Java code running on the server.
- Specifically, when the page is requested by a user and processed by the HTTP server, the HTML portion of it is passed straight through. However, the code portions of it are executed at the time the request is received and the dynamic content generated by this code is spliced into the page before it is sent to the user. This provides for a separation of the HTML presentation aspects of the page from the programming logic contained in the code, a unique benefit we'll consider in detail.
- Benefits of JSPs as a system for dynamic content generation:
  - Since this is a Java-based technology, it inherits all of the advantages that the Java language provides with respect to development and deployment. As an object-oriented

language with strong typing, encapsulation, exception handling and automatic memory management, use of Java leads to increased programming productivity and more robust code. In addition, as Java-based code, it becomes portable across all platforms that support a JVM.

- Because JSP is a Java standard (and a part of the Java EE Platform), it enjoys the Java EE server's vendor-neutral nature. You can literally move the same JSP code from one web container to another and run it "as is" with no changes.
- It has an ability to leverage the underlying Java platform functionality, such as database access, directory services, distributed computing (especially critical in the Java EE environment), and security elements.
- JSP is designed to promote software reusability, another critical element in the modern software development process. Entire JSP-based files or individual elements of them can be reused (using custom tags and supporting Java-based classes).
- An important side effect of decoupling of presentation and implementation through JSP is that it promotes a clear division of labor in the development and maintenance of the web applications for dynamic content generation.

How does JSP work?

The web server is needed to use and run the JSPs. More specifically, we are going to need a web container that will be hosting JSPs. In our course we are using the WebLogic Server as a web container.

Usually, to be recognized as a JSP document, the JSP file has to have an extension jsp. It has to be allocated and then found by a web container in a specific file directory.

When a JSP request comes in and gets processed by a web server the processing is done on the JSP tags present on the page in order to generate content dynamically, and the output of that processing in combination with the page's static HTML will be returned to the web browser.

The JSPs tags are basically the instructions to the web container to process them and convert them into the Java servlet code. There is a special servlet, JSP page compiler that is called by the container to process JSP pages. This JSP page compiler compiles JSP into a page-specific servlet and then runs this JSP servlet class. The important rule is that before compiling, the JSP page compiler will check to see if the requested JSP page has been compiled already. If it has been, the JSP page compiler will check to see if the compiled JSP page is current. It will compile it again only if the JSP source code is newer than its compiled counterpart.

To compile a JSP page, the JSP page compiler parses it looking for JSP tags.

While parsing, it translates the JSP code into the equivalent Java source code which, when executed, will generate the output according to the Java code within JSP tags.

Static HTML gets translated into Java strings, which will be sent unmodified in their original sequence into an output stream. The JSP tags are translated into Java code.

If JSP code refers to a Java Bean, the call to a corresponding bean object will be constructed, according to the JSP-to-Bean property reference. Scripting elements are transferred as is.

The overall Java code will be mixed in an output stream with static HTML content at an appropriate location.

The Java code then is used to create a service method for a servlet. After creating the Java source the page compiler calls the Java compiler. That is why it is good to inform the JSP container about the location of a Java compiler and some compiling parameters.

The resulting Java class is then placed in a specified location to be executed as a servlet.

Obviously, the action described above needs to take place once per each JSP. For the next request the page compiler is not going to repeat this process unless the JSP source code has been changed (a time stamp is used to verify that).

JSP Tag Conventions

The JSP tags are referred to as HTML-like. They begin and end with angle brackets (the < and > characters). They fall into two basic categories:

- scripting-oriented tags (this idea came from ASP technology); and
- full set of tags based on the Extended Markup Language, XML.

Scripting-oriented Tags

The scripting-oriented tags start with <% and end with %>. The following is an example of scripting-oriented tag:

```
<%! Double radius = 7.5; %>
<%= 2 * Math.PI * radius %>
<% if (radius > 10.0)
{
out.println("Exceeds recommended maximum. Please analyze the value.");
} %>
<%@ include file="myFooter.html" %>
```

XML-based Tags

The XML-based tags are case sensitive. The XML tags can be with or without a body. The ones with a body start with < to open and /> to close it.

```
<jsp:forward page="admin.jsp"/>
```

The ones with a body follow an HTML convention. The opening tag uses the "<" character, and the ">" character is the closing tag. The closing tag uses as its closing delimiter.

In this example, the body content is another JSP tag. The tag in the body does not itself have a body, so it follows the previous convention.

```
<jsp:useBean id="login" class="MyLoginBean">
<jsp:setProperty name="login" property="group" value="admin"/>
</jsp:useBean>
```

Buffered Output

- The HTTP response that the server sends back to the client involves existence of an output buffer. The JSP architecture is designed in such a way that the page's content is not automatically sent to the browser as it is generated. Instead, all page content is stored temporarily in an output buffer. The physical transmission occurs only after the entire page has been processed.

- Due to the buffering step, it is possible to produce and send the header information for the response at any time during the process of generating a body for the HTTP response. A JSP could include code that conditionally sets a cookie, and that code could appear anywhere on the JSP page.
- Although HTTP response cannot be retracted, the use of the buffered output means that the response can be postponed until all of the content information has been generated. Programming logic can drive this.
- The output buffer is finite in size. When it gets full, its content is automatically flushed and sent to the client. In this case it is possible that the body and the header of the response will be transferred to the client mixed up.

Session Management and Scalability

HTTP Session Management

Session management is a process of maintaining state across multiple HTTP requests. Just like the Java servlets, JSPs have the capability to maintain session information. For example, the session object (as an implicit object) can be used to store data like login information or the content across multiple user screens. The session object gets removed from the memory once the user's session times out.

Scalability

- Scalability is one of the most critical issues in web architecture today. The failure to provide scalability leads to user frustration and refusal to utilize a given site or application.
- The scalability of a JSP container is completely based on the scalability of servlets. The memory management to control servlet class loading and maintenance in conjunction with concurrent multi-users requests, as well as buffering of the output response, becomes critical for JSP container vendors.

JSP Markup Tags

There are four categories of JSP markup tags:

1. directives (for the JSP container to construct a servlet);
2. scripting elements (to provide programming instructions to be executed to process each request);
3. comments (for documentation purposes);
4. actions (based on a specific tag, but usually to get some request processing done).

Directives

Directives are orders for the JSP container to provide special information about the JSP page to be processed. They don't produce any output to the client; instead, they generate side effects that change the way the JSP container processes the page.

1. The **page directive** has the following syntax:

   <%@ page attribute1="value1" attribute2="value2" attribute3=... %>

   or in XML-based format

<jsp:directive.page attribute1="value1" attribute2="value2" attribute3=... />

The eleven page directive attributes are: info, language, contentType, extends, import, session, buffer, autoFlush, isThreadSafe, errorPage, isErrorPage.

Each JSP page can have multiple page directives. However, with the exception of the **import** attribute, no individual page attribute may be specified multiple times on the same page.

Perhaps the most common directives are the **import** attribute and the **session** attribute.

For complete coverage of syntax and meaning of each attribute please read the JSP documentation.

2. The **include directive** enables a developer of a JSP page to include the content of one file into another. The file to be included is identified via a local URL. As a result of the action, the directive will be replaced with the contents of the indicated file. The syntax of the include directive is as follows:

   <%@ include file="localURL" %>

   or in XML-based format

   <jsp:directive.include file="localURL" />

   There is no limit on number of the include directives within one JSP page and there are no restrictions on nesting the include directives.

   The include directive has the effect of substituting the contents of the included file before the page is translated into source code and compiled into a servlet.

3. The **taglib directive** notifies the JSP container that a page relies on one or more custom tag libraries. We will cover custom tag creating and use in next section.

Scripting Elements

There are three types of scripting elements: declarations, expressions, and scriptlets.

1. **Declarations** are used to define variables and methods specific to a JSP page in order to reference them later on the same page. The following is a syntax:

   <%! declaration(s) %>

   or in XML-based format

   <jsp:declaration> declaration(s) </jsp:declaration>

   We can declare variables and methods, both class members and instance members.

2. **Expressions** are elements to enforce the JSP container to produce a result from evaluating the expression, convert it into a string, and send as a response. The syntax for this type of element is:

<%= expression %>

or in XML-based format

<jsp:expression> expression </jsp:expression>

3. **Scriptlets** are a code written in Java (default scripting language) or some other scripting language, to be inserted into a servlet, compiled, and executed to produce the output as a response for a client. The syntax for this type of element is:

   <% scriptlet %>

   or in XML-based format

   <jsp:scriptlet> scriptlet </jsp:scriptlet>

   JSP's scriptlets will run for each request received by the page.

For detailed information you might look at our textbook or in Sun's JSP documentation.

Implicit Objects

In addition to custom objects that a JSP developer can build and have complete control over, the JSP container provides a number of predefined internal objects or **implicit objects**. They are implicit because they are automatically available via scripting elements.

There are nine available implicit objects:

Category	**Object**	**Class or Interface**	**Description**
Input/Output	request	javax.servlet.http.HttpServletRequest	Request data, including parameters
--"--	response	javax.servlet.http.HttpServletResponse	Response data
--"--	out	javax.servlet.jsp.JspWriter	Output stream for page content
Contextual	session	javax.servlet.http.HttpSession	User-specific session data
--"--	application	javax.servlet.ServletContext	Data shared by all application pages
--"--	pageContext	javax.servlet.jsp.PageContext	Context data for page execution
Servlet-related	page	javax.servlet.jsp.HttpJspPage	Page's servlet instance
--"--	config	javax.servlet.ServletConfig	Servlet configuration data
Error handling	exception	java.lang.Throwable	The uncaught Throwable that resulted in the error page being invoked

The following is a brief description of all nine objects broken into four categories:

Input/Output

- **Request** object is required to implement the javax.servlet.ServletRequest interface associated with the request or, in case of an HTTP request; it must implement a subclass of this interface, javax.servlet.http.HttpServleREquest. The request object gets either all the request parameters via getParameterNames or getParameterValues methods, or an individual request parameter via getParameter(name) method. It also allows for retrieval of the request header information. In addition, it provides miscellaneous functionality such as access to the request's URL, user, and session.
- **Response** object implements the javax.servlet.http.ServletResponse interface or, in case of HTTP response, is a subclass of this interface, the javax.servlet.http.HttpServletResponse. You can set content types, get into response header, add cookies, check status code, etc.
- **Out** object is an instance of the PrintWriter used to send output to the client. But the JSP spec requires it to be even more specific and to become an instance of javax.servlet.jsp.JspWriter class. Using the out object, you can control the buffering process. The most practical use of out object is in scriptlets, because you can generate output from within the body of a scriptlet without having to close the scriptlet to insert static page content or JSP expression.

Contextual Objects

- **Session** object helps keep track of user's current session. You can add application-specific information to the session object by means of attributes and their values. By doing this you can transmit this information between pages within a given session. The information about the session itself is available through the methods of the javax.servlet.http.HttpSession interface. The session object is an instance of this interface. Since the sessions are created automatically, this object is bound even if there was no incoming session reference. The one exception is if the session attribute of the page directive is set to false. Then any attempt to refer to the session object causes compilation errors.K
- **Application** object is an instance of the javax.servlet.ServletContext interface. Through this interface's methods you can obtain some information about the servlet container, archive logging, and associate attributes with an application.
- **PageContext** object is an instance of the javax.servlet.jsp.PageContext class. This very powerful object has variety of features. It provides programmatic access to all other implicit objects. It provides methods for accessing attributes created by other objects. Finally, it can tranfser control from the current page to another page, or dispatch the request from one JSP to another. As we know, four different implicit objects are capable of storing attributes: the pageContext object, the request object, the session object and the application object. How long the particular attribute will live depends on the scope of the object which stores it:
    - Page attributes, stored by pageContext, last only within the duration of a single page.
    - Request attributes may be passed between pages as control is transferred.
    - Session attributes persist for the duration of the user's session.
    - Application attributes are retained for the duration of a single application's pages.

Servlet-related Objects

- **Page** object is simply a synonym for Java keyword t h i s, often called "this pointer". It represents the JSP page itself and sooner or later the servlet class into which the page has been translated. It's rarely used if the scripting language is Java.
- **Config** object is the javax.servlet.ServletConfig interface implementation for this JSP page and can be used to interact with initialization parameters. The practical use of it is very limited.

Error Handling

- **Exception** object is only available on pages that have been designated as error pages using the isErrorPage attribute of the page directive. This object is an instance of the java.lang.Throwable class.

Comments

There are two styles of comments:

1. the comments that are in a JSP page for documentation purposes only,
2. the comments that are transmitted back to the browser.

There are three ways to have comments in a JSP page:

1. **The content comments**
   The content comments are the only style that is going to be shipped back to browser (obviously, since they are only comments, the browser is not going to visualize them). The following is a syntax that will be included in output of a JSP page (notice the similarity with HTML syntax):

   <!- - comments [<%= expression %>]- ->

   Notice that since these comments are part of dynamic content of the page, they [comments] by themselves could be dynamically generated (for instance, when you use expressions within comments).

2. **The JSP comments**
   The JSP comments can be seen only on a JSP source page and do not show on output being sent to the browser. The following is a syntax:

   <% - - comments - - %>

3. **The scripting language comments**
   Those comments are just regular Java language comments embedded into a Java scriptlet and are completely ignored by JSP container. They will appear in the servlet source code but not in generated output being sent to the browser. The following is a syntax:

   <% some-Java-scriplet-code /* comments */ %>

   or

   <%
   some-Java-scriplet-code // comments some-Java-scriplet-code // comments some-Java-scriplet-code // comments %>

Actions

JSP actions (supported only in a single XML based format) control the behavior of the servlet engine along with other directives and scriptlets. They allow for the transfer of control from one page to another, inclusion of a file, interaction with JavaBeans components residing on the server side, or finally generation of HTML for the Java plugin. In addition, the JSP developer can create a custom JSP tag via tag libraries to extend the functionality of standard JSP tags.

Available actions include:

- jsp:forward - Forward the requester to a new page.
- jsp:include - Include a file at the time the page is requested.
- JavaBeans tags:
    - jsp:useBean - Find or instantiate a JavaBean.
    - jsp:setProperty - Set the property of a JavaBean.
    - jsp:getProperty - Insert the property of a JavaBean into the output.
- jsp:plugin - Generate browser-specific code that makes an OBJECT or EMBED tag for the Java plugin.

This week we will focus only on forward and include actions and leave the JavaBeans tags for the next week.

The jsp:forward action

Using this action you can permanently transfer control from a JSP page to a different URL location on the server. It has a single attribute, page, which should consist of a relative URL.

The following is syntax:

```
<jsp:forward page="localURL" />
```

This could be a static value, or could be computed at request time, as in the two examples below:

```
<jsp:forward page="/mysubdir/myError.jsp" />
<jsp:forward page="<%= someJavaExpression %>" />
```

someJavaExpression will be evaluated by the JSP container at run-time and the result will be interpreted as the URL of the forwarded page. So, this is what happens at run-time on the server:

browser –> request –> original page –> forwarding request –> forwarded page –> response –> browser

This transfer of control is completely transparent to the browser.

The JSP container will assign a new pageContext object to the forwarded page, however it will keep the request object and session object the same. So, as a result:

- page attributes are not shared.
- request and session attributes are shared.
- application attributes may or may not be shared.

Since the request object is common to both the original page and the forwarded page, any request parameters that were available on the original page will be also available to the forwarded page.

It is also possible to add new request parameters passing control this way by means of a tag within the body of the action:

```
<jsp:forward page="local URL">
<jsp:param name="parameterName1"
value="parameterValue1"/>
...
<jsp:param name="parameterNameN"
```

value="parameterValueN"/>
</jsp:forward>

The action terminates the processing of the current page and passes control to the forwarded page. Therefore, this action tag can be particularly useful in conditional logic. For example, you might check if a client successfully logged in to forward a request to the "Success" page, whereas in case of an unsuccessful login, to forward the same request to an "Error" page.

Note: Be aware of the output buffering issue. If the output buffer has already been flushed by the time the request is forwarded, it means that a portion of the response has already been sent to the client's browser. In this case, it becomes impossible to disregard that output. The JSP container will check this condition and will throw an IllegelStateException exception. To avoid this problem, make sure to have a big enough buffer size!

The jsp:include action

With this action you can **temporarily** transfer control to another document to incorporate the content generated by this "included" document (perhaps, a JSP page or servlet or even static HTML document).

The following is syntax:

<jsp:include page="localURL" />

Unlike the include directive, such as:

<%@ include file="MyFile" %>

which inserts the file at the time the JSP page is translated into a servlet. This action inserts the file at the time the page is requested. This gives the JSP developer a great deal of power to determine what has to be included at run-time vs. compile time. The downside is that the include action has some performance implications because the JSP container has to dispatch the request to the included page, and then to incorporate the response into the original output.

As with forward action, the included JSP page will be assigned a new pageContext object, but will share the same request and session objects, and may or may not share the same application object.

It is also possible to add new request parameters by means of a tag within the body of the action:

<jsp:include page="local URL" flush="true">
<jsp:param name="parameterName1 value="parameterValue1"/> ... <jsp:param name="parameterNameN value="parameterValueN"/> </jsp:include>

Note: Again, as with forward action, we have to be aware of the issue of flushing the output buffer.

Design Considerations

The most challenging step in a process of developing a web-based application is obviously a design.

The **objective** of design is to decide what type of software components our application will consist of and how those components will interoperate.

The **constraints** include such issues as amount of development effort, ease of debugging and testing, quick implementation of future enhancements, performance and scalability, and others.

This is a time when we have to architect our application components in a such a way that will gain the most by utilizing the Java EE programming model and services, and advocating the software use and reuse paradigm.

There are many different ways of designing the application. Usually we approach the design stage by presenting the required functionality of the application as the:

- presentation layer,
- control layer,
- application logic layer.

We have to consider many different, sometimes contradictive factors. The expected application behavior has to be sustained no matter how irregular or spontaneous the behavior our client will impose.

- What if a client decides to leave a session without logging out and gracefully notifying about his intention?
- What if a client somehow decides not to follow the native flow of application control, but instead to ask the desired URL to go directly to the middle of application state?

Please study one of the most popular presentation layer design patterns called Model-View-Controller (MVC) presented in Chapter 4.4, "Web-Tier Application Framework Design" of the *Designing Enterprise Applications with the J2EE™ Platform, Second Edition* book, see
http://www.adobe.com/support/jrun/documentation/pdf/j2ee_ent_app_design.pdf

JSP Tag Extensions

Let's look into another very important capability of JSPs called JSP tag extensions (or custom tags).

At this time, I would like you to study the following chapters from Oracle's Java EE 5 Tutorial:

Chapter 7, JSP Standard Tag Library, see
http://docs.oracle.com/javaee/5/tutorial/doc/bnakc.html

and

Chapter 8, Custom Tags in JSP Pages, see
http://docs.oracle.com/javaee/5/tutorial/doc/bnalj.html

In addition, please read:

The JSP Custom Tags Tutorial, see http://www.tutorialspoint.com/jsp/jsp_custom_tags.htm

Now, I want to emphasize a few critical key points:

- JSP custom tags are designed based on a combination of two technologies: XML and Java. XML is used to provide the **TLD**, a **T**ag **L**ibrary **D**escriptor. Java is used to implement **Tag Handlers**. A Tag Handler is a Java class that implements an interface, *javax.servlet.jsp.tagext*. This interface has six methods; the three most used methods are:
    - **int doStartTag ( ) throws JspException**

  - **int doEndTag ( ) throws JspException**
  - **void release ( )**
- Web application containers invoke the Tag methods listed above in the order they are listed.
- The JSP file uses the *taglib* directive to directly access a tag library descriptor. For example:

  < taglib uri="**counters**" prefix='util' %>

- A taglib directive in the web application's deployment descriptor specifies the actual location of the TLD for a tag library identified by a URI of counters. For example:

  <taglib>
  <taglib-uri>**counters**</taglib-uri>
  <taglib=location>**/WEB-INF/tlds/counter.tld**</taglib-location>
  </taglib>

- Implement a tag handler that extends the TagSupport class [a helper class provided by a Java package with standard implementation of two methods, doStartTag ( ) and doEndTag ( )], to overwrite those methods.
- Be aware of the tag life cycle management.
- Be aware of JSP 1.2 tag extension enhancements.
- Pay special attention to deployment and packaging of tag libraries. As with other Java EE application components, this process (deployment and packaging) becomes a "popular" source of errors.

JSPs with Java Beans Support

Java beans are a good way to take out code from JSP files. They can be used to minimize the coding and maintenance in JSP and leave all the complexity in the JavaBean (good practice). JavaBean components are well suited to the task of capturing business logic into reusable software components. These reusable components are potentially large Java programs.

Just like for any JSP tag, there are two formats for the JSP action directive:

<jsp:useBean id="name" scope="scopeName" bean_details />

or

< jsp:useBean id="name" scope="scopeName" bean_details >
Body
</jsp:useBean>

where **bean_details** is either **class="classname"** or **class="classname" type="typeName"** or **beanName="beanName" type="typeName"** or **type="typeName"**

The **<jsp:usebean> </jsp:usebean>** tag attempts to locate an instance of the Bean class. If no such object exists, the tag causes a Bean object to be created. Creation can take either of two forms:

1. Instantiation from the Bean class.
2. Reconstruction from a previously serialized Bean object.

The attributes for the JSP tag above are described below:

- **id="beanInstanceName"**

  The purpose of this attribute is to name a variable that identifies the Bean. You can reference this variable later in expressions or scriptlets in the same JSP file. This variable name is accessible within the scope that you identify as explained below.

- **scope="page | request | sessions | application"**

  The purpose of this attribute is to define the scope in which the Bean object exists. This also defines the scope in which the **id** described above is accessible.

  The choices are page, request, session, or application.

  - **page** scope means that an object [Java bean] is bound to the *javax.servlet.jsp.PageContext*. This object is placed in the *PageContext* object and can be accessed by invoking the *getAttribute ( )* methods on the implicit *pageContext* object.
  - **request** scope means that the object is bound to the *javax.servlet.ServletRequest* and can be accessed by invoking the *getAttrubute ( )* methods on the implicit request object.
  - **session** scope means that the object is bound to the *javax.servlet.jsp.PageContext* and can be accessed by invoking the *getValue ( )* methods on the implicit session object.
  - **application** scope means that the object is bound to the *javax.servlet.ServletContext* and can be accessed by invoking the *getAttribute ( )* methods on the implicit application object. This scope is the most persistent.

  The default is **page**.

- **class="package.class" type="package.class"**

  This attribute causes a Bean to be instantiated from a class, using the new keyword and the class constructor.

  The class must not be abstract, and must have a public constructor that takes no arguments. Both the package and class name are case sensitive.