Java Web Programming

n + 1, Inc

Feb 2009

Contents

List of Tables

List of Figures

List of Programs

Chapter 1

Java Web Programming Introduction

1.1 Introduction

Java Web Programming provides an in depth look at web based development using the Java programming environment. The course is broken into three sections. The first provides a quick overview of the HTTP protocol and review of basic HTML. The purpose of this section is to provide an understanding of the foundational elements of basic HTML development. The goal is not to provide intimate details of HTML, but to provide a basic background of HTML elements in order to create simple web pages.

The second part of the course focuses on the building blocks for developing a Java web application. The primary concern will be on understanding and using Servlets and Java Server pages to develop web based applications. The class concentrates on the core elements of each technology without delving into every nuance of the interface.

The final section takes the the foundational building blocks and combines them to develop a powerful web framework. One that is both robust and flexible. It provides a tour of the important web design patterns and how they are applied to create a workable web framework.

Chapter 2

HTML and HTTP Protocol

Objectives

- Understand the basics of the HTTP protocol
- Learn URL naming conventions
- Learn basics of how a HTTP Server works
- Understand core HTML tags

2.1 HTTP Protocol

The HTTP protocol provides a mechanism for transmitting MIME encapsulated files from one machine to another. MIME is an acronym for Multipurpose Internet Mail Extensions. MIME provides the format for basic Internet transmission. It doesn't modify basic ASCII text messages, but it provides a scheme for encoding binary data binary data as ASCII characters to simplify transmission of data. Since the discussion on HTTP and HTML only uses the ASCII character sets a further exploration of MIME is unwarranted.

2.1.1 URLs

Uniform Resource Locators, URLs, are used to describe the location of a networked resource. The URL can be broken into four distinct parts for describing a location. The first part defines the protocol necessary to access the resource. The second part defines the address of the service on the network. This information is generally provided as a DNS name but the direct IP address can be used. The third section is optional and is used to tell what port or application needs to be accessed for the resource. Lastly, the URL provides information about retrieving the resource from the server's application.

The last part is known as the uniform resource identifier or URI. Below is an example URL which incorporates each of these elements.

```
http://nplus1.net:8080/some/location/myfile.txt
```

The protocol used to access the resource is the HTTP protocol. The service is located at the DNS name nplus1.net and can be found on port 8080. The resource, myfile.txt, is located at the "/some/location/" path.

It is possible to add parameters to the end of the URL that can be used by the server for processing. Parameters are added to the end of the URL by first appending a "?". Then the parameters are added as a series of name value pairs that are separated by the equal sign. Each name pair is tokenized by using the & character. The series of parameters are known as the query string. Below is an example of a URL with a query string.

```
http://nplus1.net:8080/servlet/myPage.html?name=foo&val=bar
```

In this case, there are two sets of name value pairs after the "?". They are "name = foo" and "val = bar". Most server side programming languages allow the developer to access these variables. Java is no exception. In fact these are integral to web based programming. It is important to note that the URI defined above does not include the query string parameters.

2.1.2 HTTP

HTTP is the protocol used to transfer resources defined by a URL. While the HTTP protocol is used to facilitate transmission it does not actually transfer the resource. The actual transfer of the file is handled by the network protocol such as TCP/IP. HTTP is a protocol that piggy backs on top of the transfer protocols to send the actual data.

Because it uses other transmission protocols there is no way to maintain state between requests. HTTP is at its most basic a simple request/response protocol between a server and a client. The server receives the request and sends a response back to the client.

How A HTTP Server Works

The basics of handling the HTTP protocol are simple. The first step is for a client application to make a HTTP request upon the server. The most common form of client application is the web browser such as Firefox or Internet Explorer. The client will make a network connection to the HTTP server and sends a HTTP request over the connection.

Inside of the HTTP request there are several pieces of information. The name of the resource being requested along with meta data about the request itself. The HTTP server receives the request, locates the resource, and returns it back to the client in a HTTP response. If the resource can't be located the server will send back an error message within the HTTP response. Once the client receives the response, the connection is closed between the client and server.

The client is then responsible for displaying the resource. In the case of HTML, the browser will render the page based upon the tags within the HTML document.

The HTTP Request

The HTTP request that is sent to the server has three distinct sections. The first is the request line. It tells the server what type of request is being made, what resource is requested, and what version of the protocol is being used. The second part is the header information. This contains meta information about the request. The final part is the body. It contains any parameters that are being passed to the server resource.

The best way to understand the request is to take a look at a simple example. Below is a URL where a request is made.

```
http://nplus1.net/foo/myimage.png?name=foo&value=bar
```

The resulting request would look similar to the following:

```
Request Line     GET /foo/myimage.png HTTP 1.0
Header           Connection: Keep-Alive
                 Referrer: http://nplus1.net/foo.html
                 User-Agent: Mozilla/4.0
                 Host: nplus1.net
                 Content-Type: application/x-www-form-urlencoded
                 Content-Length: 17
Body             name=foo&value=bar
```

The GET in the request line tells the HTTP server to retrieve the request resource for the client. There are various request types that will be described in detail later. The URI follows the type and tells the server what and where the resource to retrieve resides. In this instance the resource myimage.png is located in the name space /foo/.

In addition to the request line, the request contains header data. Headers provide meta data about the incoming request. Generally, it contains information specific to the request such as length of message and the MIME encoding used. In addition, it contains information about the client such as browser type and what page linked to it.

The body contains the parameters passed to the resource. These are the name value pairs that show up after the ? in the URL. For a basic web server these values are meaningless. For web servers that can provide dynamic content these values can be parsed and used to affect the resulting resource that is returned.

HTTP Request Types

The GET method introduced last section is a request for a single resource. It is one of seven different request types that are defined by the HTTP protocol. Table 2.1 provides a brief description of each type.

Tool	Description
GET	Retrieves a single resource
POST	Transfers client information to server
PUT	Transfers a resource to be stored on the server
DELETE	Removes resource from specified URL
HEAD	Requests header information only for a resource
OPTIONS	Returns a list of the server's supported HTTP methods
TRACE	Returns the entire network route from client to server and back

Table 2.1: HTTP Request Types

Most of the options shown in the table are either not useful or major security holes in the protocol. Two examples of security problems are the PUT and DELETE types. Neither are supported by servers or browsers because they represent a severe break in security. Clients should not be able to either put files on a web server or delete them. The HEAD, OPTIONS, and TRACE types are supported by web servers, but are not used by modern web browsers. GET and POST are the ones most often used.

GET is a simple request of a resource. When a client enters a URL into a browser a GET is made upon the destination server and the HTML page is returned. If any images are needed to render the HTML page, they are requested using the GET mechanism to complete the assembling of the page.

The POST method is used to submit form data. Whenever a client provides a user with an HTML form for filling out multiple fields, the data is generally submitted as a POST. The POST adds the data to the request without adding it as parameters to the URL. The data becomes embedded in the body of the request instead of in the URL.

HTTP Response

The HTTP response, like the request, has the three distinct parts. The first is the status line which provides the return code. The code can be used to determine if the call succeeded or failed. The code also tells why the request failed. The second part is the header. It contains meta data about the server and the response. The last section is the body of the response. The requested resource is embedded within the body of the returned message.

Below is an example of an HTTP response.

```
Status Line     HTTP/1.0 200 OK
Header          Date: Tue 13 Feb 2003 09:18:43 GMT
                Server: Apache/1.3.23
                Content-Type: text/html
                Content-Length: 1256
                Last-Modified: Fri, 8 Feb 2003 22:05:49 GMT
Body            <html>
                    <header>
                        <title>Test Page</title>
                    </header>
                    <body>
                        <h2>Hello World</h2>
                        This is a simple test page.
                    </body>
                </html>
```

That status line in the example above has a code of 200. 200 is a successful request. Other common return codes are 404 (page not found) and 500 (server error). The header contains information about the server and the response. In this instance the server is Apache v1.3.23. The context type of the body is text/html, the length of the message is 1256 bytes and it was send February 8, 2003. The body of the message contains the simple HTML page.

2.2 HTML

XML, the eXtensible Markup Language, is a generic specification for creating markup languages. It allows developers to define custom tags in order to develop a language for structuring data. HTML is a subset of XML that was created to define the structure for displaying web content. The browser renders the page based upon the tags within the HTML page.

The purpose of this section is to briefly discuss the major elements of the HTML specification and explain how they are rendered within a browser. The section does not provide in depth coverage of the topic of HTML. It is designed to give the student enough information to be able to create simple HTML pages for the purpose of learning Java web programming.

2.2.1 Structure Tags

A web page must follow the basic structure for a well formed page. The initial tag within a page is the <**html**> tag . It is the root tag for all HTML pages. Embedded within the html tag is the <**body**> tag . The information contained within body is what gets rendered by the client's browser. The basic layout for the page is shown below.

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

The first <html> tag is considered the opening tag. According to the XML standard, all open tags must have an associated close tag. A "/" is used to define a closing tag. Thus </html> is the closing tag that matches the opening tag. All content must be within the opening and closing tags. Notice the body tag has its opening and closing tags enclosed within the html tag.

The HTML framework provides a mechanism for defining meta-data about an HTML page. That information can be placed withing a <**head**> tag . The head tag goes before the body tag within the html tag. Common tags within the head are the <**title**> and <**link**> tags. The title tag is used to define the name of the page. The value placed within the tag will show up in the title bar of the browser. The link tag is used for including JavaScript and style sheets.

It is also possible to put comments into a web page. The <!– –> symbols are used to bracket comments. Anything between the two tags are ignored by client browser. Putting all of the elements together, the code below shows the basic structure of a simple web page.

```
<html>
    <head>
        <title>The Page Title</title>
    </head>
    <body>
     <!-- formatting HTML goes here -->
    </body>
</html>
```

The example above demonstrates the basic elements that need to be in place before the formatting elements of HTML can be utilized.

2.2.2 Simple HTML Formatting Tags

HTML provides a collection of tags that can be used to organize, format, and display content within a browser. The simplest tags are stand alone tags that provide simple formatting.

The simplest of these tags are the bold and italic tags. The <**b**> (bold tag) is used to display the current font using its boldface version. The <**i**> (italics) tag is used to provide emphasis to the current font. The text in between the open and close tag will be text effected.

At this point it is important to note another XML rule. Tags can be nested inside one another , but they can not span into each other. Check out the following two examples.

```
Valid:
    This is <b>ok <i>to do</i></b>

Invalid:
    This is <b>not <i>ok</b> to do</i>
```

The first is valid because the italics tag is contained within the bold tag. The second is invalid because the italics tag spans across the close of the bold tag.

White Space And HTML

HTML is white space insensitive. That means any consecutive white space is translated into a single space. Consecutive white space is any order space, tab, or carriage return characters.

Below is a passage from Mark Twain's Tom Sawyer without using any formatting tags.

```
"Tom, it was middling warm in school, warn't it?"


"Yes'm."

"Powerful warm, warn't it?"

"Yes'm."

"Didn't you want to go in a-swimming, Tom?"

A bit of a scare shot through Tom a touch
of uncomfortable suspicion.  He searched
Aunt Polly's face, but it told
him nothing.           So he said:

"No'm well, not very much."
```

It would appear like this on the web page.

```
"Tom, it was middling warm in school, warn't it?" "Yes'm." "Powerful
 warm, warn't it?" "Yes'm." "Didn't you want to go in a-swimming,
 Tom?" A bit of a scare shot through Tom a touch of uncomfortable
 suspicion.  He searched Aunt Polly's face, but it told him nothing.
 So he said: "No'm well, not very much."
```

Not exactly the result that one might have expected.

To handle the issue of proper spacing within a page, HTML provides two simple tags that can help manipulate the spacing of text. The <**p**> (paragraph) tag is used to demarcate a paragraph. Everything between the open and close tag are placed into their own text block that is separated by a double spaced line on either side. The <**br/**> (line break) tag is a standalone tag that inserts a line break at the current point in the text. Text after the line break shows up on the following line.

Below is a passage from Mark Twain's Tom Sawyer using the tags to properly format the page.

```
<p>"Tom, it was middling warm in school, warn't it?"</p>
<p>"Yes'm."</p>
<p>"Powerful warm, warn't it?"</p>
<p>"Yes'm."</p>
<p>"Didn't you want to go in a-swimming, Tom?"</p>
<p>
    A bit of a scare shot through Tom a touch of<br/>
    uncomfortable suspicion.  He searched Aunt Polly's <br/>
    face, but it told him nothing. So he said:
</p>
<p>"No'm well, not very much."</p>
```

The resulting format within the web page is shown below.

"Tom, it was middling warm in school, warn't it?"

"Yes'm."

"Powerful warm, warn't it?"

"Yes'm."

"Didn't you want to go in a-swimming, Tom?"

A bit of a scare shot through Tom a touch of
uncomfortable suspicion. He searched Aunt Polly's
face, but it told him nothing. So he said:

"No'm well, not very much."

Hyper-Links

One of the primary reasons the Internet is useful is the concept of the hyper-link. The hyper-link allows text within a page to be marked in such a way as to be selectable by the end user. The text is linked to another HTML page through the use of URLs. Therefore once the user selects the hyper-linked text, the page specified by the URL is queried and the resulting HTML page is displayed to the user.

The tag used to hyperlink various URLs is the <**a**> tag . The *href* attribute is used to define the URL with which to link. Any text between the opening and closing <a> tags are selectable by the end user and will link them to the new page.

```
Go to <a href="http://cnn.com">CNN</a> to find the latest news.
```

In this example the word CNN would be underlined and clicking on it would take one to CNN's web site.

2.2.3 Lists

HTML provides a simple mechanism for building a list of objects. First the developer must determine what type of list to use. The unordered list, <**ul**> tag, creates a bulleted list of elements. The ordered list, <**ol**> tag, provides a list ordered from numeric 1. After the developer has decided upon a list a series of list elements <**li**> tags are used to define each item to be listed. Each <li> tag set would be nested within the outer tag type.

An example of an ordered list is shown below.

```
<ol>
    <li>First Entry</li>
    <li>Second Entry</li>
    <li>Third Entry</li>
</ol>
```

In this case the ordered list is selected with three items in the list. Each item (First Entry, Second Entry, and Third Entry) are encapsulated within separate list entry tags. The example would result in the following display.

```
1. First Entry
2. Second Entry
3. Third Entry
```

2.2.4 Tables

Tables are one of the more complicated HTML widget structures available to a developer and are also the most flexible. A table is composed of three parts. The table definition, rows in the table, and columns within a row.

The table is defined with the <**table**> tag. It demarcates the start of the table along with providing information about the characteristics of the table. Table 2.2. shows the most common attributes that are associated with the table.

Attribute	Description
border	Determines the thickness of the table's border
width	Determines width of the table
cellspacing	Determines amount of space between cells in the table
cellpadding	Determines how much space to place between cell wall and cell's content

Table 2.2: Common Table Attributes

The *border* attribute is used to determine the thickness of the table's border. The default border value for a table is 1. To remove borders from a table set the border's value to 0.

The *width* attribute determines the width of the table. The most common form of measurement is in pixels. Any numeric value for the attribute will be assumed as screen pixels. A monitor image consists of thousands of little dots which taken together compromise the picture. Each little dot is called a pixel. Therefore on a screen where the resolution is set to 800x600 the table width can be set up to a size of 800 to fit on the screen[1]. Therefore `width="700"` would instruct the table to be displayed within 700 pixels on the screen.

Another option for width is to use a percentage. A percentage value will use that percentage of the available screen space for the size of the table. Thus `width="80%"` would set the table to eighty percent of the available screen size available within the browser.

cellspacing and *cellpadding* are two more options that are used in helping with managing white space within a table. Cellspacing is an attribute that determines the space between cells in a table. Cellpadding is used to determine the space between the edge of the cell and the data within the cell.

[1]This is not technically true. One must take in the pixels used by the browser's border into account when determining maximum displayable width

Below is an example of using the table tab to set a table's attributes.

```
<table border="0" cellspacing="0" cellpadding="2" width="500">

</table>
```

When generating the elements that make up a table it is important to understand how tables populate themselves. There is no way to explicitly state how to populate a cell within a table. Instead the user must build the table one row at a time. For example if one was creating a table with two rows and three columns, the person would have to create the first row and populate the three columns and then go to the next row and populate those columns. Rows are defined with the <**tr**> (table row) tag. Columns within a row are defined using the <**td**> (table data) tag. The row tags are nested inside the table tag and the data tags are nested inside the row tags.

The best way to understand the table is to look at an example. Take the table shown below.

Name	Phone Number
Doe, Jane	555-1233
Smith, Beth	555-1236

The first step is to define the table.

```
<table border="1" cellspacing="0" cellpadding="2">

</table>
```

The next step is to define the first row for the table.

```
<table border="0" cellspacing="0" cellpadding="2">
    <tr>

    </tr>
</table>
```

Now that the row has been created it is possible to fill out each column in the row with the appropriate data.

```
<table border="0" cellspacing="0" cellpadding="2">
    <tr>
        <td>Name</td>
        <td>Phone Number</td>
    </tr>
</table>
```

This process will continue until all 3 rows of data within the table are filled.

```
<table border="0" cellspacing="0" cellpadding="2">
    <tr>
        <td>Name</td>
        <td>Phone Number</td>
    </tr>
    <tr>
        <td>Doe, Jane</td>
        <td>555-1233</td>
    </tr>
    <tr>
        <td>Smith, Beth</td>
        <td>555-1236</td>
    </tr>
</table>
```


One final note about tables. The <td> tag has attributes that can be used to adjust the cell's alignment. The *align* attribute is used to determine justification. The valid values for align are left, right, and center. By default cells are left aligned. The *valign* attribute is used to determine vertical alignment of a cell. If a cell contains multiple lines of data the vertical property might need to be adjusted. The valign property can be either top, bottom, or center. Center is the default value. Below is an example of using both forms of alignment

```
<table border="0" cellspacing="0" cellpadding="0">
    <tr>
        <td align="center">Name</td>
        <td align="center">Phone Number</td>
    </tr>
    <tr>
        <td valign="top">Doe, Jane</td>
        <td>
            555-1233<br/>
            555-1234<br/>
            555-1235<br/>
        </td>
    </tr>
    <tr>
        <td>Smith, Beth</td>
        <td>555-1236</td>
    </tr>
</table>
```

It would create the following table

Name	Phone Number
Doe, Jane	555-1233 555-1234 555-1235
Smith, Beth	555-1236

2.2.5 Form Tag

In HTML, forms provide developers with the means to interact with the end user. It allows the developer to prompt the end user for information. The base tag for a form is the <**form**> tag. It denotes the beginning of the form along with some basic information about how the form will submit data to the server. Specifically it allows the developer to set the type of HTML request used to submit the form and where the form should be submitted.

The form's attribute *method* is used to determine the request type. This can be set to either POST or GET. Case sensitivity is not important for the method. The *action* determines the URL where the form data will be sent. The action can be either a fully qualified URL or a relative URL. Their are two types of relative URLs. One form is a URL that is relative to the current URL. An example is shown below.

```
<form method="post" action="proccess.jsp">

</form>
```

The method is set to post which means it will make an HTTP POST call to the process.jsp URL. The rest of the URL will be determined by the URL that returned the form.

The second type of relative URL is one that begins with a leading "/" character. These are relative to the root URL of either the web application or the domain name.

Input Tag

The <**input**> tag allows for the creation of data entry widgets within the browser. The purpose of these tags are to collect data from the end user. Input widgets come in six forms, a simple text field, a password field, a hidden field, a checkbox, a radio button, or a button. The *type* attribute determines which type of input widget will be displayed within the browser.

Text Field The text widget field creates a single line field that allows the user to input simple text data. Setting the value of the input tag to "text" will create the text box. The size and feel of the text field is controlled by the *size* and *maxlength* attributes.

The size attribute determines the initial size of the control. The size value represents the number of "W" characters that will fit within the textbox. The maxlength attribute determines the maximum number of characters that can be entered inside the textbox. The box will not allow any additional characters to be typed into it.

The textbox can be pre-populated using the *value* attribute. By setting the value, the browser will initialize the textbox widget with the value associated to the value attribute.

The most important attribute is the *name* field. This attribute associates a name with the widget. The name can be referenced through a dynamic web application to retrieve the data entered by the end user.

```
Year: <input type="text" size="5" maxlength="4" name="year" value="2008/>
```

Password Field The password field extends the concept of the text box by providing a special function. Any characters typed into the field will be hidden by "*" characters. It allows end users to enter sensitive data such as a password without it being displayed on the screen.

```
Password: <input type="password" size="10" name="passwd"/>
```

Hidden Field The hidden field is a special field that does not show up to the end user. It allows the developer to embed immutable name value pairs into a form. It is generally used to pass information like an ID field along with form data for processing without displaying it to the end user. It is generally done to ease server processing of data.

```
<input type="hidden" name="sku" value="552-234562"/>
```


Checkbox The checkbox widget is a simple binary selection box. The item is either selected or not. The *name* attribute is the same as other input boxes. It allows a name to be associated with the input. The *value* field is different than the textbox. It assigns the value only if the box is selected by the user. If the box is not selected then no value will be associated with the name. The checked attribute[2] is used to pre-select the checkbox.

On unique feature of the checkbox is the ability to associate the same name with multiple checkboxes. In that case all of the selected values will be associated with the name. Look at the following example.

```
Check Which Pets You Own:<br/>
<input type="checkbox" name="pet" value="cat" checked="true"/> Cat<br/>
<input type="checkbox" name="pet" value="dog"/> Dog<br/>
<input type="checkbox" name="pet" value="goat"/> Goat<br/>
```

In this case the Cat checkbox is selected while the other two are blank. If the user selects Goat and Cat the the value associated with pet will be cat and goat. Being able to work with multiple values will be discussed in the form processing parts of the course.

Radio Button The radio button, like the checkbox, is a binary selection widget. Unlike the checkbox, radio buttons can be grouped into an exclusive OR combination. Therefore only one item in the group may be selected at any single time. The groupings are based on the name of the input line. Thus all radio buttons with the same name are grouped into the same radio group and only one of them can be selected at a time.

```
Check Which Pet You Would Own Next:<br/>
<input type="radio" name="pet" value="cat"/> Cat<br/>
<input type="radio" name="pet" value="dog" checked="true"/> Dog<br/>
<input type="radio" name="pet" value="goat"/> Goat<br/>
```

Submit Button The submit button is used to post the values from the form to the URL specified in the *action* attribute of the form tag. The submit tag creates a button widget on the page. The text on the button is supplied by the *value* attribute. If a name is associated with the tag then the name value pair is submitted with the form. It provides a way to identify between multiple submit buttons on the form. If no value is associated with the widget the value "Submit" will show up on the button.

```
<input type="submit" value="Press To Submit Form"/>
```

Reset Button The reset button is a special widget that resets all of the forms fields to their original values. The *value* attribute determines what text shows up on the button.

```
<input type="reset" value="Reset All Fields"/>
```

[2]The checked attribute does not technically have to be set to checked="true". The word checked is all that is necessary inside the tag for the widget to be selected. checked="true" is used to ensure XML compliance for tag attributes.

Here is a complete example of using input tags within a form.

```
<h2>Update Personal Information</h2>
<form method="post" action="http://nplus1.net/myapp/processForm.html">
    <input type="hidden" name="id" value="5415/>
    <table border="0" cellspacing="3" cellpadding="0">
        <tr>
            <td>First Name:</td>
            <td><input type="text" name="firstName" value="Beth"/></td>
        </tr>
        <tr>
            <td>Last Name:</td>
            <td><input type="text" name="lastName" value="Smith"/></td>
        </tr>
        <tr>
            <td valign="top">Favorite Pet:</td>
            <td>
                <input type="text" name="pet" value="cat"/> Cat<br/>
                <input type="text" name="pet" value="dog"/> Dog<br/>
                <input type="text" name="pet" value="goat"
       checked="true"/> Goat
            </td>
        </tr>
        <tr>
            <td>Over 21?</td>
            <td><input type="checkbox" name="over21" value="yes"/></td>
        </tr>
        <tr>
            <td align="right"><input type="reset" value="Reset Form"/></td>
            <td><input type="submit" value="Update Data"/></td>
        </tr>
    </table>
</form>
```


Select Tag

The select tag allows developers to use drop down boxes within a web form. The <**select**> and <**option**> tags are used in conjunction to create drop down boxes. The select tag determines the name to associate with the drop down. The option tag is used determine both the value to submit with form along with the value to put in the drop down box. Below is a simple example of a drop down box.

```
<select name="pet">
    <option value="cat">Cat</option>
    <option value="dog">Dog</option>
    <option value="goat">Goat</option>
</select>
```

In this instance the name of the drop down box is pet. There are three options that show up in the drop down box. They are Cat, Dog, and Goat. The values submitted for those selections are cat, dog, and goat respectively. Thus if the end user selects Goat from the drop down the name value pair pet=goat will be submitted to the server.

The *selected* attribute[3] can be used to pre-select an option within the select block.

```
<select name="pet">
    <option value="cat">Cat</option>
    <option value="dog" selected="true">Dog</option>
    <option value="goat">Goat</option>
</select>
```

The select block can also be adapted to generate a multi-select box. One where the end user can select multiple entries within the box. The *multiple* attribute is used to modify the select box into allowing multiple selections.

```
<select name="pet" multiple="true">
    <option value="bird">Bird</option>
    <option value="cat">Cat</option>
    <option value="dog" selected="true">Dog</option>
    <option value="goat">Goat</option>
</select>
```

[3]The selected attribute does not technically have to be set to selected="true". The word selected is all that is needed inside the tag for the widget to be pre-selected. The selected="true" is used to ensure XML compliance for tag attributes.

How can one make multiple selections from a simple drop down box? The select box can be expanded beyond a single line drop down into a multi-line selection box. The number of lines displayed by the box is modified using the *size* attribute. By default the value is set to one. Setting this value greater than one will create a selection box instead of a drop down.

```
<select name="pet" size="4" multiple="true">
    <option value="bird">Bird</option>
    <option value="cat" selected="true">Cat</option>
    <option value="dog">Dog</option>
    <option value="duck">Duck</option>
    <option value="goat" selected="true">Goat</option>
    <option value="rabbit">Rabbit</option>
</select>
```

Text Areas

All of the widgets up to this point have dealt with collecting short pieces of information from the end user. What happens if a developer wants to collect paragraphs worth of data? What if they want to be able to collect comments or detailed descriptions from end users? HTML provides a text area widget for collecting multiple lines of data. The <**textarea**> tag is used for this purpose.

```
<textarea></textarea>
```

The *name* attribute is used to associate a name with the widget. The other two important attributes are *rows* and *cols*. These are used to determine the size of the widget. Rows determine how many rows will make up the text box while the columns determine how many "W" characters will fit in each row. Below is an example of text area that uses these attributes.

```
<textarea name="comments" rows="8" cols="60"></textarea>
```

The one attribute that textarea does not have is the value attribute. If a developer wants to pre-populate the text field the information must be placed between the open and close textarea tags.

```
<textarea name="comments" rows="8" cols="60">This is a test</textarea>
```

The data between the textarea tags is space sensitive. Therefore do not place unnecessary spaces or they will show up within the text box. The following is a classic example of what not to do.

```
<textarea name="comments" rows="8" cols="60">
    This is a test
</textarea>
```

In this case the "This is a test" will be placed on the second line of the text box and 4 characters to the right of the left margin.

2.3 Lab Activity

The goal of this lab is to develop an HTML form that allows a user to input data into a web browser.

Part 1.

Develop a form that asks the user to provide the following inputs:

Label	Field	Notes
First Name	Text box	
Last Name	Text box	
City	Text box	
State	Drop down box	Use 4 states
Sex	Use radio buttons	
Pets	Use check boxes	Create 4 options
Comments	Text area	50 columns wide and 6 rows tall
Submit Button	button	Button text should read “Save”

Part 2.

Place the form inside of a table that is 2 columns by 9 rows. The input labels should be in the first column and right aligned. The second column should contain the various input fields and be left aligned. The labels should be aligned to the top of each table row. The submit button should go in the second column of the ninth row.

Part 3.

Preset the value of the city with your current city. Pre-select one of the pets to be checked. In addition make one of the states be selected for the user. Preset the comments field to read ”Enter Comments Here”.

Chapter 3

Servlets

Objectives

- Understand how servlets work
- Understand the life cycle of servlets
- Learn how to code a servlet
- Understand the format of a WAR deployment file
- Learn how to create a WAR file for deployment

3.1 Servlets

The HTTP protocol is a protocol that is used to transfer resources based upon a simple request and response procedure. The client requests a resource from the server and the server responds with the requested resource. The process is known as aptly as the request/response model. The Java Servlet technology is an implementation of this model.

The Servlet technology is bundled within the J2EE environment. All J2EE servers provide a servlet container to handle Servlet services. It is provided as a JAR file by the server. The implementation API for the servlets can be found within the *javax.servlet* package. The central class within the package is the Servlet interface. It provides a set of methods for processing the request/response model. The *javax.servlet.http* package provides a concrete implementation of the Servlet interface. The implementation is one specific to the HTTP protocol.

It is important to note that the *javax.servlet* and *javax.servlet.http* packages are not part of the core J2SE. In order to use classes during development, the classes that form the core of the servlet technology must be added to the CLASSPATH.

They are generally supplied as a JAR file by the J2EE container utilized. In the case of JBoss the jar file is servlet-api.jar and can be found in the client directory of the distribution.

3.1.1 How Servlet Container Works

While servlets are used to handle incoming requests, they must be deployed within a servlet container. The servlet container provides the infrastructure necessary for handling the HTTP protocol. Extending upon the concept of Java's platform independence, servlets can be deployed to any of the available servlet containers[1].

The process is simple. A developer deploys an implementation of the servlet interface to the server. As part of the deployment, the developer maps a URL to the servlet. When a HTTP request comes into the container, the container finds the appropriate servlet based upon the requested URL. The incoming request is converted into a **ServletRequest** object. In addition a **ServletResponse** object which represents the response is created and together the ServletRequest and ServletResponse are passed to the servlet for processing.

The servlet is responsible for handling the incoming request and fill out out the response object to be returned to the client. Once the servlet is done processing, the container takes the information from the ServletResponse and uses it to generate a HTTP response. Which in turn is delivered back to the client application by the container.

[1]Servlet containers include JBoss, IBM's Websphere, BEA's Web Logic

3.1.2 Life Cycle of Servlets

The **Servlet** interface defines all of the methods necessary to handle the entire life cycle for a servlet within a container. The life cycle for the servlet can be broken into three phases: the start of the servlet's life, the handling of HTTP requests, and the end of the servlet's life. The upcoming sections describe each of the cycles in greater detail.

Start Of Servlet Life

The Servlet interface defines an **init()** method that initiates the start of a servlet's life cycle. When an incoming HTTP request for the URL mapped to the servlet comes into the web container, it instantiates an instance of the servlet and calls the *init()* method. The *init()* method is where the developer can add any initialization code for the servlet. Since the it is only called once at creation, it is a perfect candidate for code to initialize variables, connections to a database, or open file streams. Another common activity is to retrieve parameters from various configuration files such as the web.xml[2] file.

Below is an example of using the *init()* method to read application properties into a servlet.

```
private Properties props;

public void init() {
    try {
        FileInputStream fis = new FileInputStream("/tmp/myapp.prop");
        props = new Properties();
        props.load(fis);
        fis.close();
    }
    catch (IOException e) {
        // log error
    }
}
```

[2]The web.xml file will be described in greater detail later in the module.

Handling Servlet Requests

The primary function of a servlet is to be able to handle a HTTP request. The container takes the incoming HTTP request and creates a ServletRequest object that contains header and body information from the request. The servlet container also creates a ServletResponse object that has attributes and methods for creating a response that can be returned by the container. The request and response objects are passed to the servlet's **service()** method for processing.

```
void service(ServletRequest request, ServletResponse response)
```

The servlet API provides a HTTP specific implementation of the servlet interface. It is made to specifically handle the HTTP protocol. The **HttpServlet** class provides this functionality. In addition, the ServletRequest and ServletResponse interfaces are implemented by the **HttpServletRequest** and **HttpServletResponse** classes respectively.

To be able to handle the HTTP protocol, the servlet API provides a HTTP specific implementation of the servlet interfaces. Therefore the Servlet interface is implemented by the HttpServlet class and the ServletResponse and ServletRequest interfaces are implemented by the HttpServletRequest and HttpServletResponse classes.

HttpServlet Methods

The HttpServlet class provides an implementation of the Servlet's *service()* method. It divides incoming calls into separate method calls based upon the type of the request made upon the server. The HttpServlet class has eight methods for each type of HTTP request. They are shown in table 3.1.

Eight HTTP Methods
`public void doGet(HttpServletRequest req, HttpServletResponse resp)`
`public void doPost(HttpServletRequest req, HttpServletResponse resp)`
`public void doPut(HttpServletRequest req, HttpServletResponse resp)`
`public void doDelete(HttpServletRequest req, HttpServletResponse resp)`
`public void doOptions(HttpServletRequest req, HttpServletResponse resp)`
`public void doPut(HttpServletRequest req, HttpServletResponse resp)`
`public void doTrace(HttpServletRequest req, HttpServletResponse resp)`
`public void doHead(HttpServletRequest req, HttpServletResponse resp)`

Table 3.1: Eight Protocol Specific Methods

Each of these eight methods can throw either a **ServletException** or an **IOException**. The implementations provided by the HttpServlet servlet for each of these methods do nothing.

Developers are expected to extend the HttpServlet class and then override the methods they wish to implement. An example of overriding methods to handle GET and POST methods are shown below.

```
    public class MyServlet extends HttpServlet {
       public void doGet(HttpServletRequest req, HttpServletResponse resp)
                  throws ServletException, IOException {
           \\ implementation to handle HTTP GET request goes here
       }

       public void doPost(HttpServletRequest req, HttpServletResponse resp)
                  throws ServletException, IOException {
           \\ implementation to handle HTTP POST request goes here
       }
}
```


End of Servlet Life

When it is time for the servlet engine to shut down, it attempts to unload servlets that have been instantiated. Part of the process is the call the servlet's **destroy()** method.

The user implements the *destroy()* method to close any resources that were opened in the *init()* method. Once the *destroy()* method has been called, the servlet is no longer accessible to the outside world and is marked for garbage collection.

3.1.3 Complete Servlet Example

Program listing 1 and listing **??** provide a complete example of a servlet. The example was broken up into two sections to fit on the page. The sections that follow provides a detailed explanation of how the code works.

Program Listing 1 Example Servlet Part 1

```
import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class SimpleCounterServlet extends HttpServlet {
    private int counter;
    private FileWriter fw;
    private PrintWriter pw;

    public void init() throws ServletException {
        try {
            // set counter to 1
            counter = 1;

            // open log to write timestamps
            fw = new FileWriter("/tmp/counter.log");
            pw = new PrintWriter(fw);
        }
        catch (IOException e) {
            System.out.println("Unable to open log file");
        }
    }
}
```


Program Listing 2 Example Servlet Part 2

```
    public void doGet(HttpServlet request, HttpServletResponse response)
                throws ServletException, IOException {
        // log entry
        Date now = new Date();
        this.pw.println("Hit " + counter + " occurred at " + now);

        // send back the number of hits for servlet
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
out.println("<html><body>");
        out.println(counter + " hits");
out.println("</body></html>");

        // increment counter
        counter++;
    }

    public void destroy() {
        // close down log file
        try {
            if ( pw != null ) {
                pw.close();
            }
            if ( fw != null ) {
                fw.close();
            }
        }
        catch (IOException e) {
            System.out.println("Could not close log file");
        }
    }
}
```

Program 1 and 2 provide an implementation of a simple counter servlet that employs all three phases of the servlet's life cycle. The SimpleCounterServlet is created by extending the HttpServlet class and overriding three methods to provide the servlet's functionality.

The first method is the *init()* method. It initializes the counter to 1 and opens a file to write logging messages. The FileWriter opens the file /tmp/counter.log for writing. The PrintWriter wraps the FileWriter to provide a high level writer making it easier to write String data to the log file.

The *doGet()* method provides the business logic of the servlet. The initial thing it does is write out to the log file the hit count and what time it occurred. The rest of the method is used to create a HTML to display the web page and increment the counter. The HttpServletResponse object is used to provide a response to the client.

The PrintWriter is a high level writer that is used to return data to the client. In this instance the PrintWriter is used to return the HTML response to the client.

The *destroy()* method is the final method. Since the method is called at the end of the servlet's life cycle it is used to close the PrintWriter and FileWriter.

Threading And Servlets

Servlet containers are multi-threaded application servers that can handle multiple request at the same time. Servlets on the other hand are only created one time by the servlet engine. Thus the servlet is shared between the various running threads of the servlet container.

Special care must be given to make sure that a developers servlet is thread safe. The previous example shown is not thread safe because nothing guarantees that the counter is only accessed and modified by one servlet request at a time. In theory two requests could return the same result if the threads switch contexts before the counter could be incremented.

The best way to avoid threading issues is to avoid the use of private member variables within a servlet or if they are necessary to make them read only. If the servlet doesn't have an internal state then it is most likely to be thread safe. Future sections will discuss better methods for storing application state.

3.2 Deploying Web Applications

Web applications are the amalgamation of servlets, Java server pages, tag libraries, third party libraries, static content, and images. Java introduced the WAR file as a means of simplify the process of organizing all of the elements and deploying them as web applications.

A WAR file is a standard Java JAR file with a .war extension. The war file is used to group all of the varied elements of the web application into a single distributable unit. What uniquely identifies a WAR file from a normal JAR file is its adherence to a predefined structure. In addition, a WAR file will contain deployment descriptors that are used to initialize the application within the servlet container.

3.2.1 Structure of WAR file

WAR files adhere to a very specific structure. They all must contain a /WEB-INF directory which houses the executable classes and any configuration files. The main configuration file, **web.xml**, goes directly in the WEB-INF directory. It provides the configuration for the web application and all servlets. The web.xml file will be described in greater detail later.

Under the WEB-INF directory are the class and lib sub-directories. The classes directory is where any server side classes are archived. The lib directory is used to store any third party libraries.

All static content and JSP pages can be placed anywhere off the root directory with the exception of the WEB-INF directory. Table 3.2 provides an outline of the WAR structure

Tool	Description
/	Root directory which contains JSPs, images, and other static resources
/WEB-INF	Contains executable classes and configuration files
/WEB-INF/web.xml	Deployment configuration file
/WEB-INF/classes	Server side executable classes are stored in this directory
/WEB-INF/lib	Third party JAR file libraries are stored in this directory

Table 3.2: WAR File Structure

3.2.2 Deployment Descriptor

The web.xml file is the deployment descriptor for Java web applications. It is an XML document that contains information about the web applications configuration and set up information for each servlet.

Below is a an example of a typical web.xml file.

```
<web-app>
    <servlet>
        <servlet-name>Dummy</servlet-name>
        <servlet-class>net.nplus1.acme.DummyServlet</servlet-class>
        <init-param>
            <param-name>filePath</param-name>
            <param-values>/tmp/myFile.txt</param-values>
        </init-param>
    </servlet>
    <servlet>
        <servlet-name>TestServlet</servlet-name>
        <servlet-class>net.nplus1.acme.TestServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>Dummy</servlet-name>
        <url-pattern>/test/dummy.html</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
        <servlet-name>TestServlet</servlet-name>
        <url-pattern>/testing/*</url-pattern>
    </servlet-mapping>
    <session-config>
        <session-timeout>3600</session-timeout>
    </session-config>
    <welcome-file-list>
        <welcome-file>index.html</welcome-file>
        <welcome-file>index.jsp</welcome-file>
    </welcome-file-list>
</web-app>
```

The configuration file in this instance is broken up into 4 distinct sections. The servlet section defines the class path of each servlet along with any initialization information for the servlet. The servlet mapping section maps servlets to a specific URL. The session config is used to define the session time out value which will be described in greater detail later. Lastly, the welcome file list tells the servlet container what file to use when a user requests a directory as a URL.

Servlet Tags

The <**servlet**> tag alerts the servlet container to the availability of a servlet class. Inside the <servlet> tag is the <**servlet-name**> and <**servlet-class**> tags. The servlet name is used to provide a unique identifier for the servlet class that is identified. The name is later referenced in the servlet mapping tag set to map the servlet class to a particular URL. The servlet class must provide the full package name for the servlet. The identified class must be found within the WEB-INF/classses directory or within a JAR file in the WEB-INF/libs directory. If it is not found then the deployment will fail. Every servlet must have its own set of <servlet> tags.

```
<servlet>
    <servlet-name>TestServlet</servlet-name>
    <servlet-class>net.nplus1.acme.TestServlet</servlet-class>
</servlet>
```

In addition the <servlet> tag supports the passing of initial parameter data to the servlet. These values are made available at run time.

```
<servlet>
    <servlet-name>Dummy</servlet-name>
    <servlet-class>net.nplus1.acme.DummyServlet</servlet-class>
    <init-param>
        <param-name>filePath</param-name>
        <param-values>/tmp/myFile.txt</param-values>
    </init-param>
</servlet>
```

The <**init-param**> tag shown above allows the developer to pass parameters to the servlet at run time. The parameters are grouped into name / value pairs that can be accessed by the servlet through the ServletConfig object. The ServletConfig object is created and maintained by the base implementation of HttpServlet. Therefore a local servlet call to *getServletConfig()* allows access to the object. Below is an example of retrieving initial parameters from within a servlet.

```
public class MyServlet extends HttpServlet {
    public void init() throws ServletException {
        ServletConfig config = this.getServletConfig();
        String filePath = config.getInitParameter("filePath");
        ...
    }
    ...
}
```

The initial parameter value is retrieved using the *getInitParameter(String name)* method from the ServletConfig. In the case of the deployment descriptor shown above "filePath" will return the value of "/tmp/myFile.txt".

Servlet Mappings

The <**servlet-mapping**> tag is used to match servlets with a particular URL. The <**servlet-name**> tag contains the name of a servlet defined in the <servlet> tag. The <**url-pattern**> tag provides a specific URL to map to the named servlet. The goal is to map a URL to the servlet's class.

```
<servlet-mapping>
    <servlet-name>Dummy</servlet-name>
    <url-pattern>/test/dummy.html</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>TestServlet</servlet-name>
    <url-pattern>/test/testing.*</url-pattern>
</servlet-mapping>
```

When a web application is deployed, the servlet container loads the URL patterns into a table in memory. When an end user makes a request on the server, it looks in that table to see if the URL requested matches one of the servlet defined URLs. From the table it can retrieve the appropriate servlet class. If the servlet hasn't been previously invoked, the container uses reflection and creates a new instance of the servlet and adds it to the table.

To provide additionally flexibility, the <url-pattern> tag can use "*" as a wild card character. Allowing multiple requests to be directed to the same servlet. Look at the following pattern.

```
<url-pattern>/actions/*</url-pattern>
```

Anything after the actions part of the URL will be redirected to the same servlet. Therefore any of the following partial URLs would match the pattern.

```
/actions/foo.html
/actions/testing/bar.htm
/actions/lookups/foobie
/actions/race/stripes.jsp
/actions/ms.asp
/actions/foo/bar/miss.ht
```

All of these URLs be redirected to the servlet defined in the servlet-mapping.

Note About URLs All web applications have a context path which identifies it from other web applications deployed to the servlet container. On JBoss the default context path is defined by the name of the war file[3]. Therefore if acme.war and test.war are both deployed to the servlet container, one would have a context path of acme while the latter would have a path of test.

```
http::://nplus1.net/acme/shopping/cart.html
http::://nplus1.net/test/test.jsp
```

In this case "acme" and "test" within the URL path are used to determine which web application is being accessed. The context path only defines part of the URL. The servlet path is the section of the URL which goes from the context path to the point it matches a servlet. This is the path which is defined in the url-pattern. Therefore for the first URL the following url-pattern would be necessary.

```
<url-pattern>/shopping/cart.html</url-pattern>
```

If a wild card is used in a <url-pattern tag>, the servlet path only goes to the point where the pattern matches. The rest of the URL is known as the path info. Take the following URL.

```
http::://nplus1.net/acme/shopping/checkout/cart.html
```

Below is the shopping servlet's url-pattern.

```
<url-pattern>/shopping/*</url-pattern>
```

In this instance, the context path is /acme. The servlet path is /shopping and the path info is /checkout/cart.html.

[3]The context path is configurable in JBoss and on all servlet containers

Session Config

As noted before, the HTTP protocol is a stateless protocol. There is no way to tell if any two requests are coming from the same user. To be able to maintain state between multiple client requests, Java containers provides a session object. The session object is mapped to a cookie stored on the client's browser. The concept of sessions will be studied later.

The web.xml deployment descriptor provides a section, <**session-config**>, for configuring the behavior of the session object. It allows a <**session-timeout**> value to be configured.

```
<session-config>
   <session-timeout>3600</session-timeout>
</session-config>
```

The session timeout is used to determine how long the cookie on the browser side is valid. The timer starts at the time the last request was made on the server. When a user makes a future request, the container checks to see if the user has exceeded the allotted time. If so the session is invalidated and resource are released back to the system.

The timeout of a session is experienced by people who use on-line banking solutions. If one doesn't continue using the service frequently, the service logs the user out of the system. In this case the session has expired and the user must log in again to create a new session.

The <session-timeout> in the configuration file is marked in seconds. Therefore a session timeout of 3600 is the equivalent to 1 hour.

Welcome File List

Not all URL requests made by the end user point to an exact resource. Sometimes the URL will point to a directory structure. Since a directory is not a resource which can be returned, the container will search for a default file within the directory.

The file the container searches for is defined in the deployment descriptor within the <**welcome-file-list**> tag.

```
<welcome-file-list>
    <welcome-file>index.html</welcome-file>
    <welcome-file>index.jsp</welcome-file>
</welcome-file-list>
```

Each file is separated by a welcome-file tag. The servlet container will search for each file in the order in which they appear within the file list. In the example above the servlet container will search for the index.html file first. If it doesn't find the file, it searches for the index.jsp file. If neither file is found then the system will return the contents of the directory or an error page depending on the configuration of the servlet container.

3.3 Building War File With Ant

The generation of the WAR file can be automated with a build utility called Ant. Ant will compile the Java classes and package all of the components into a working WAR file.

Traditional build tools like **make** and **nmake** are shell based tools that are dependent upon the platform for which they are written. Ant is different from those tools in that it was written using Java. That makes it a platform independent tool for building applications. It goes beyond a simple build tool by allowing developers to package applications, deploy applications, create javadoc, and even run unit tests.

The Ant build process is driven by a XML build file named build.xml.

3.3.1 Ant Configuration File

The Ant configuration file is an XML file that uses a series of tags to create a build plan. The default name for an Ant build file is build.xml, but ant can be configured to use other build files. Inside the build.xml a project is defined as the root element. Inside the project a series of targets are created. Each target represents a set of tasks to complete a step in the build process.

Below is an example outline of a build.xml file.

```
<project name="My Project" default="deploy" basedir=".">
    <!-- setup stuff goes first -->
    <target name="build">
        ...
    </target>

    <target name="package" depends="build">
        ...
    </target>

    <target name="deploy" depends="package">
        ...
    </target>

    <target name="clean">
        ...
    </target>
</project>
```

The <**project**> tag is the root tag and it represents the project being build. It has three attributes associated with it. The *name* attribute is a marker that serves little purpose for the build, but is a place holder used by IDE's such as eclipse that encorporate Ant into its build environment. The *basedir* attribute tells Ant what directory to run the build. In this case the current directory is used for building. The *default* attribute tells which target to run if none are specified when Ant is invoked.

Each <**target**> contains a series of tasks to perform. The target has a couple of attributes. The *name* attribute is used to uniquely identify the target. The *depends* attribute is an optional attribute that informs Ant of a precondition before running the target.

In the example above, the depends attribute for the "deploy" target is "package". This means that the package target must be executed before deploy can be run. Since the package target depends on the build target, build would have to be run first, package second, and finally the deploy target.

A note about comments. Since the Ant configuration file is XML it uses XML style comments as shown in the example above.

3.3.2 Ant Variables

Ant uses variable to be able to either retrieve values from an external source or to create custom variable for local use. The most common way to create and manage variables is through an Ant property file.

Ant Property File

The Ant property file is used to store platform or deployment specific information about the build. Most of the time it is used to define path information about the layout of the project source files. Property files follow the Java Properties file standard. They are a text file with multiple name/value pairs that exist one per line.

```
jboss.home=/opt/jboss
jboss.server.lib=/opt/jboss/server/default/lib
jboss.deploy=/opt/jboss/server/default/deploy

web.dist=./dist
web.src=./src
web.build=./build
web.conf=./conf/web
web.jsp=./web/jsp
web.static=./web/static

third.party.utils=./third-party/nplus1-utils/lib
```

The standard method for defining property names is to separate words or concepts with periods in a name. The above example uses a number of variables to represent directories within the file system and the project. In this case both absolute and relative paths are defined. Relative paths are based upon the project root directory. The property file can be loaded using the <**property**> tag.

```
<property file="./conf/build.properties"/>
```

Ant variables are referenced using special ${} identifiers. To reference the variable *jboss.server.lib* within an Ant configuration file, the following would be used.

```
${jboss.server.lib}
```

In this case the variable would be replaced by the value "/opt/jboss/server/default/lib" within the property file.

Ant Path Variables

Another use for ant variables is to pre-define a set of "paths" that can be used by the compiler to represent the classpath. Below is an example of defining a path element.

```
<path id="jboss.classpath">
    <fileset dir="${jboss.server.lib}">
        <include name="**/*.jar"/>
    </fileset>
</path>
```

The <**path**> tag has an id attribute that represents the name of the path variable being created. The next step is to add all of the path elements to the path. This is done by adding one or more file sets. Each file set is represented by the <**fileset**> tag. The *dir* attribute tells us which directory to include in the fileset.

The selection of files within the directory can be determined using <**include**> or <**exclude**> tags. The <include> tag is an inclusive set while the <exclude> tag determines what is excluded from the set. The *name* attribute associated with the <include> tag determines what file to include in the fileset.

Wild card character "*" can be used to select multiple files to include or exclude from the set. The * matches any set of characters in the pattern. Table 3.3 provides example of how to use the * wild card character to create patterns.

Pattern	Matches
*	matches all files within a directory
*.jar	matches all files within directory that have .jar extension
jboss*	matches all files in directory that start with jboss
jboss	matches all files in directory that have jboss as part of the name
jboss*.jar	matches all files in directory that start with jboss and have a .jar extension

Table 3.3: Pattern Matches

The special "**" wild card includes the current directory along with any sub-directories when searching for file matches. Therefore "**/*.jar" shown in the example above would mean all jar files in the *${jboss.sever.lib}* directory and any sub directories.

3.3.3 Printing Information To Screen

It is important for the build process to be able to provide feedback to the individual in charge of a project's compilation process. The <**echo**> tag is used to print information to the screen.

One use of the <echo> tag is to create a target that contains a list of the other actionable build targets to run. Below is an example of such a target.

```
<target name="build-targets">
    <echo>My Build System Targets</echo>
    <echo>build        - builds servlets and custom classes</echo>
    <echo>package      - package web app into war file</echo>
    <echo>deploy       - copy war file to deploy directory</echo>
    <echo>clean        - cleans build and dist directories</echo>
</target>
```

The project's default target can be changed to "build-targets" and when the Ant is invoked without a build target the list of possible build targets will be displayed to the end user.

3.3.4 Compiling Java Files

One of the core responsibilities of Ant is to compilation of Java source code. The <**javac**> tag is used to compile source code into byte code. Below is an example of compiling source code.

```
<target name="build">
   <mkdir dir="${web.build}"/>
   <javac srcdir="${web.src}" destdir="${web.build}">
      <classpath>
         <path refid="jboss.classpath"/>
      </classpath>
   </javac>
</target>
```

An important part of the build process is the ability to separate .java files from .class files. Running *javac* from the command line by default will create .class file in the same directory as the .java file. The problem with this is it becomes tricky trying to separate them apart when building an application. The application only needs the .class files. Therefore it is nice to separate them into different directories. A standard that is commonly used is to put the java files in a src directory and the class files in a build directory.

When building an application, the build directory might not exist. Therefore the first step is to make sure the directory exists and if it doesn't create it. The <**mkdir**> tag does just this. The *dir* attribute is used to tell Ant which directory to create.

The next step is the actual compilation of the source code into byte code. Here is where the <javac> tag is used. To help in the separation of source and compiled code, the <javac> tag provides an attribute to define where the source code resides, *srcdir*. It also provides a place to put the bytecode, *destdir*. The <javac> command will compile all .java files in the source directory and all sub-directories.

What happens if the compilation process needs external jar files not included in the core language? When working with web applications, servlets are not part of the core application. The compiler needs access to the HttpServlet and other classes that are part of web development. These need to be added to the compiler's classpath at build time.

The <**classpath**> tag is used to create the necessary classpath. In the <classpath> tag the developer must define a <**path**> tag. This can be done as shown when creating variables or the developer can access a previously defined path element. In the example above the *refid* attribute is used to recall a previously set path variable.

3.3.5 Packaging a WAR File

Once all of the source code is compiled, it needs to be gathered into a distributable WAR file. Ant provides a <**war**> tag for creating the web application's WAR file. The <war> tag has two attributes. The *destfile* attribute defines the name of the WAR file and where to create it. The most common destination for WAR files is into a distribution directory named dist. The <**mkdir**> tag can be used to make sure the destination exists before placing the file there. Below is an example of a target that creates a WAR file.

```
<target name="package-web" depends="build-web">
    <mkdir dir="${web.dist}"/>
    <war destfile="${web.dist}/myapp.war"
                webxml="${web.conf}/web.xml">
        <classes dir="${web.build}"/>
        <lib dir="${third.party.utils}"/>
        <webinf dir="${web.conf}">
            <exclude name="web.xml"/>
        </webinf>
        <zipfileset dir="${web.static}"/>
        <zipfileset dir="${web.jsp}" prefix="dynamic"/>
    </war>
</target>
```

The second attribute of the <war> tag is *webxml*. It tells the task where the web.xml deployment descriptor resides. Ant takes the web.xml file and puts it into the WEB-INF directory of the WAR file.

Four nested tags are available for the <war> tag. The first is the <**classes**> tag. The *dir* attribute tells where to find Java class files to put into the WEB-INF/classes directory of the war file. It will take the directory specified and any sub-directories and place them into the WEB-INF/classes directory of the WAR file. Multiple <classes> tags can be used for multiple class file locations. The <**lib**> tag works in similar fashion to the <classes> tag. Instead of putting class files into a web application it is used to add JAR files to the application. It takes all of the files within the directory specified by the dir attribute and places them into the WEB-INF/lib directory of the WAR file.

Both of these tags work like the <filest> tag and can have <include> and <exclude> tags associated with them.

The <**webinf**> tag is used to add additional files to the WEB-INF directory inside the war file. Again it takes all of the files from the specified directory and puts them into the WEB-INF directory. In the example above the <exclude> tag is used to keep from the web.xml from being added since it is already being placed there by the war tag's webxml attribute.

The last tag is the <**zipfileset**> tag. It is used to put a set of files into the root directory of the WAR file. The *prefix* attribute creates a directory within the root of the war file. It then puts the files from the *dir* directory within that sub-directory. In the example above the files in the *web.static* directory will be placed in the root of the WAR file while the contents of the *web.jsp* directory will be placed in the /dynamic directory of the WAR file.

3.3.6 Deploying a WAR File

Depending on the application server, it could provide a mechanism for deploying a web application to the server. JBoss is a J2EE server where it is possible to have Ant deploy the application. JBoss uses a deploy directory for deployment. Any new WAR file placed into this directory will be loaded by the server.

```
<target name="deploy" depends="package-web">
    <copy file="${dist}/myapp.war" todir="${jboss.server.deploy}"/>
</target>
```

An Ant deployment can be accomplished with the <**copy**> tag. It will copy a file to a specified directory. The *file* attribute determines what file to copy while the *todir* attribute tells where to copy the file.

3.3.7 Cleaning Workspace

The <javac> tag will check to see if a .java file needs to be compiled before going through the compilation process. It checks to determine if the .java file is newer than the already existing .class file. If so then it compiles the file. The <war> tag behaves in the same way. If nothing has changed since the last war file nothing will be created.

To avoid relying on this mechanism, it is useful to clean the development environment so everything can be rebuilt. The easiest way to do this is to delete the build directory that contain the .class files and .war files. Since the build file has isolated the build and deployment directories it is a simple task to delete these files. Below is an example target to perform this task.

```
<target name="clean">
    <delete dir="${base.build}"/>
    <delete dir="${base.dist}"/>
</target>
```

The <**delete**> tag is used to delete a directory. The *dir* attribute tells Ant which directory to delete. Obviously, all the sub-directories will be deleted along with the primary directory.

3.3.8 Example build.xml File

```
<project name="My Project" default="build-targets" basedir=".">
    <property file="./conf/build.properties"/>
    <path id="jboss.classpath">
        <fileset dir="${jboss.server.lib}">
            <include name="**/*.jar"/>
        </fileset>
    </path>

    <target name="build-targets">
        <echo>My Build System Targets</echo>
        <echo>build        - builds servlets and custom classes</echo>
        <echo>package      - package web app into war file</echo>
        <echo>deploy       - copy war file to deploy directory</echo>
        <echo>clean        - cleans build and dist directories</echo>
    </target>

    <target name="build">
        <mkdir dir="${web.build}"/>
        <javac srcdir="${web.src}" destdir="${web.build}">
            <classpath>
                <path refid="jboss.classpath"/>
            </classpath>
        </javac>
    </target>
```



```
    <target name="package-web" depends="build-web">
        <mkdir dir="${web.dist}"/>
        <war destfile="${web.dist}/myapp.war" webxml="${web.conf}/web.xml">
            <classes dir="${web.build}"/>
            <lib dir="${third.party.utils}"/>
            <webinf dir="${web.conf}">
                <exclude name="web.xml"/>
            </webinf>
            <zipfileset dir="${web.static}"/>
            <zipfileset dir="${web.jsp}" prefix="dynamic"/>
        </war>
    </target>

    <target name="deploy" depends="package-web">
        <copy file="${dist}/myapp.war" todir="${jboss.server.deploy}"/>
    </target>

    <target name="clean">
        <delete dir="${base.build}"/>
        <delete dir="${base.dist}"/>
    </target>
</project>
```


3.3.9 Running Ant Build

Ant is run by executing the ant command line utility. The signature for the ant command is the following:

```
$> ant [options] [target [target2 [target3] ..]]
```

The most basic form is to run ant without any arguments.

```
$> ant
```

By default, it will search the current directory for a build.xml file. If one is not found then an error is returned saying build.xml file does not exist. It is possible to specify the name of the build file using the -f option.

```
$> ant -f mybuildfile.xml
```

If a target is not specified on the command line, Ant will run the target named in the default attribute of the <project> tag. Otherwise it will run the targets listed on the command line.

```
$> ant clean
$> ant deploy
```

The example above runs ant using the clean target to clean up the build space and then runs a deploy build to compile all the source, create a WAR file and deploy it to the server.

3.4 Lab Activity

Part 1.

Create a simple servlet that will countdown from 10 to 1. The page should have a heading which reads countdown and display the numbers 10 to 1 on the web page. Each number should be on its own line. Use a for loop to create the numbers.

Part 2.

Create an ant file that will build servlets and place them into a WAR file for deployment. Have the ant file build a WAR file with the servlet from Part 1. Deploy the resulting WAR file to the J2EE container.

Part 3.

Modify the servlet in Part 2 to only display odd numbers in the count down.

Part 4. (Optional)

Create a new servlet that displays a form within a table that queries the user for their first name, last name, and phone number.

Chapter 4

Servlet Request and Response

Objectives

- Understanding the Response Object
- Knowing how to use response error codes
- Learn about response header data
- Learn how to process web form data
- Understand how to get request header data
- Understand the elements of a URL
- Learn about the ServletContext
- Understand how to use forwards and includes

4.1 Servlet Response

Before it is possible to deal with incoming user requests, it is important to understand how responses are handled within servlets. When working with the response it is important to remember that the servlet container handles the actual communication between the servlet and the browser. Incoming requests are converted into concrete implementations of the **ServletRequest** interface and outgoing responses are created from a **ServletResponse** instance.

The servlet container is responsible for converting the incoming HTTP message into a request object that implements the ServletRequest interface. It will generate a **HttpServletRequest** object. The servlet container creates the HTTP response by taking information provided to the concrete implementation of the ServletResponse. In this case, the concrete implementation is the **HttpServletResponse** object.

The goal of this section is to better understand the HttpServletResponse object. the response object enjoys multiple roles. It determines the type of content that is to be transferred both character and binary data. In addition, it can set error codes to return to the client or provide custom header meta-information about the response. The response can also redirect the client to a new URL. It also has a role in the distribution and use of cookies. [1].

[1] Cookies will be discussed in the session management module.

4.1.1 Sending Data To The Client

When creating a response for the client, two things must happen for a successful response. The first step is to set the content type of the data being returned. The task can be accomplished by calling the response object's **setContentType(String type)** method.

```
response.setContentType("text/html");
```

The type value represents the type of data which is being returned. Table 4.1 shows a few common types.

Common Content Types	Description
text/html	HTML content
text/xml	XML content
image/png	PNG image
application/pdf	PDF document

Table 4.1: Response Content Types

The second step is to fill the response with the content to return to the browser. There are two output mechanisms for sending data. One is a writer and the other is an output stream. The writer, being character based, is used to pass text data such as HTML to the client. The output stream is used for binary data such as images or PDF files. Only one or the other can be used for output. Using both will result in an exception. any data passed to one of these output sources will be placed in the body of the outgoing HTTP response. The methods for retrieving the two sources are *getOutputStream()* and *getWriter()*. The signatures are shown below.

```
OutputStream getOutputStream()
PrintWriter getWriter()
```

It is worth repeating that the OutputStream is useful for sending binary data while the PrintWriter is generally used to send string based text such as HTML.

Using Response To Send HTML

What steps would be necessary to send the following HTML back to the requesting client?

```
<html>
<body>
        <h2>Test Output</h2>
        <p>
            Something would go here
        </p>
    </body>
</html>
```

The first step would be to set the content type. Since the data is HTML, the value should be set to "text/html". The next step would be to get an output source and pass data to it. The **PrintWriter** is the best choice because the data being sent is character based. The HTML shown above can be passed to the writer which in turn gets translated into the body of the HTTP response.

Program Listing 3 Example Output Servlet

```
public class MyServlet extends HttpServlet {
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
                      throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("    <body>");
        out.println("        <h3>Test Output</h3>");
        out.println("        <p>;");
        out.println("            Something would go here");
        out.println("        </p>;");
        out.println("    </body>;");
        out.println("</html>");
    }
}
```

Program 3 creates a class that extends the HttpServlet class. It overrides the *doGet(...)* method to send back HTML to the client. The content type is set to "text/html" for the HTML data. The HTML data is passed to the PrintWriter which the container uses to complete the HTTP response.

4.1.2 Error Codes

The HTTP protocol includes a mechanism for returning the status for each server response. HTTP provides a numeric code known as result codes that represents the type of response that is returned to the client. The best known is 404 page not found. But not all of the codes mean an error occurred. A 200 means that the request was able to successfully return the requested resource. Table 4.2 shows the most common result codes[2].

Common Result Codes	Description
200	Successful response
320	Content has been cached on the client
404	Page not found on the server
500	Server error. Request could not be processed

Table 4.2: Result Codes

[2]Full list of result codes can be found in the HTTP RFC http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10

Handling Error Codes

When an error code is either invoked by a servlet or by the server, the servlet engine uses a default error page to return to the user. The page is very generic and not very descriptive of the error. Since a web site designer doesn't necessarily want an ugly generic error page, it is possible to replace the default page with a custom page. The custom page must be defined within the web.xml configuration file. It is used to map the error code with an appropriate HTML/JSP page.

```
<error-page>
    <error-code>404</error-code>
    <location>/page_not_found.html</location>
</error-page>
```

In the example above, an error page is defined for a 404 error code. The path of the location is determined from the root of the web application. If a 404 error occurs trying to reach a resource, the page defined within the web.xml will be returned and not the default 404 error page.

A 404 is an error normally generated by the servlet container. It is possible for servlets to generate their own error codes. The **sendError(...)** methods are used to invoke run time HTTP errors.

```
void sendError(int errorCode)
void sendError(int errorCode, String message)
```

All of the various HTTP error codes have been defined within the servlet API. The HttpServletResponse object has static attributes for each of these error codes. Table 4.3 shows the attributes for a 404 and 500 server error.

Error Code Attributes	Description
404	HttpServletResponse.SC_NOT_FOUND
500	HttpServletResponse.SC_INTERNAL_SERVER_ERROR

Table 4.3: Error Code Attributes

Invoking the *sendError(...)* method will set the error code for the response. In addition it will cause the servlet container to stop processing the current servlet. It will ignore any data that has been added to the HttpServletResponse object. Only the error page is returned to the end user.

Program Listing 4 Example Output Servlet

```
public class MyServlet extends HttpServlet {
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
                      throws ServletException, IOException {
        response.sendError(HttpServletResponse.SC_NOT_FOUND,
"The Request Resource Does Not Exist");
    }

    public void doPost(HttpServletRequest request,
                       HttpServletResponse response)
                       throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("    <body>");
        out.println("        <p>;");
        out.println("            Thanks for submitting the survey");
        out.println("        </p>;");
        out.println("    </body>;");
        out.println("</html>");
    }
}
```

Program listing 4 provides an example of using error codes. The goal of the servlet is to handle POST messages from a web form and return a "thank you" message to the user. Since the servlet is only used to process the POST, it doesn't make any sense to allow GET requests on the servlet. In this example if the user tries to directly access the page directly from a URL, the *doGet(...)* method is called and an error is returned to the user.

4.1.3 Response Header

The HTTP response packet has a header section[3] to send meta data back to the client browser. Meta data can include items such as time stamps, server information, and message size. There are four methods for adding and setting header information.

```
void addHeader(String name, String value)
void setHeader(String name, String value)
void addDateHeader(String name, long time)
void setDateHeader(String name, long time)
```

The **addHeader(...)** and **setHeader(...)** methods are used for setting non date/time based header information. The *name* parameter is used to determine which header is being set and the *value* parameter specifies the string to associate with the header.

What is the difference between add and set methods? The set method will perform one of two actions. If a header is being initially set then it will associate the value with the header. If the header value had already been defined then it will overwrite the previous value with the new value. Add is different in how it treats values that are already set. Instead of overriding the value, it adds a second value associated with the header. It can be used to attach multiple values to a header.

The **addDateHeader(...)** and **setDateHeader(...)** methods are used to set a date/time value for the specified header. The name parameter specifies which header to set. The time parameter is used to specify the time in seconds from epoch time. This value can be retrieved from any *java.util.Date* object by calling its *getTime()* method.

Below is an example of using these methods

```
response.setHeader("Refresh", "30");
response.setDateHeader("Last-Modified", lastDate.getTime());
```

The first line sets the header "Refresh" to the value of 30. This tells the browser to refresh the page after 30 seconds. Every 30 seconds it will request a new version of the current page. The second line sets the "Last-Modified" header to the time stored in the lastDate variable. "Last-Modified" tells the browser the last time the requested resource was modified on the server.

[3]To learn of the different header values read the HTTP RFC2616 Section 14. It can be found at the following URL http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html

4.1.4 Response Forward

One thing a response can do is to tell the receiving browser to request another URL. When this occurs the browser will fetch the resource at the new URL and use it as a replacement for the original requested resource. It serves the function of redirecting the browser to a new URL.

```
response.sendRedirect("http://nplus1.net/foobie.html");
```

This is useful in development when a programmer needs to process POSTed data from a web form but doesn't want the processing page to show any results. Take a simple task list manager web application as an example. The page *addNewTask.html* displays a form that the user can fill out to add a new task. The web form's action POSTs the user input to the page *processNewTask.html*. The process new task page could add the new task and display all of the tasks, but their is already a *listTasks.html* page which displays all of the tasks. Instead of duplicating the logic, the process new task can add the task to the list and then redirect the client to the list tasks page for displaying the contents of the list including the newly added task.

4.2 Servlet Request

With a basic understanding of the servlet response, it is time to turn attention to the handling of incoming request. As noted previously, the incoming HTTP request is encapsulated by the HttpServletRequest object which implements the ServletRequest interface. The request can be used by the developer for any of the following tasks [4].

- Retrieve parameter data from URL or Web Forms
- Retrieve HTTP header information
- Get URL path elements
- Provide medium for communicating between servlets
- Retrieve cookie information
- Retrieve session object

The most common use of the request object is to collect data from a GET or POST request. The next section looks at handling POST data from web forms.

[4] Cookies and Sessions are covered in detail in the Session module

4.2.1 HTML Web Form

HTML provides a set of graphical widgets that can be used to gather information from the end user. These widgets are cobbled together to create web forms that can be filled out by the end user and submitted to the server. Below is a simple example of such a web form.

```
<html>
    <body>
        <h2>Enter Your Pet Information</h2>
        <form method="post" action="myservlet.html">
            <input type="text" name="petName"/><p/>
            <input type="radio" name="petType" value="cat"/> Cat<br/>
            <input type="radio" name="petType" value="dog"/> Dog<br/>
            <input type="radio" name="petType" value="goat"/> Goat<br/>
            <input type="submit" value="Submit Data"/>
        </form>
    </body>
</html>
```

In this example, the web form is going to POST the data to the *myservlet.html* URL. The form itself includes three widgets. One is a pet name text box. Another is the radio button which is used to select the type of pet. The final widget is the submit button for the form. The pet's name typed into the text box is associated with the name *petName*. The pet type selected by the radio buttons is associated with the variable *petType*. When the user submits the form, the data supplied is put into name value pairs based upon widget name and values selected. These name/value pairs are passed to the server for processing.

Since the form is a POST the name value pairs will be placed in the body of the HTTP packet.

4.2.2 Processing Web Form Data

The HttpServletRequest object provides multiple methods for retrieving parameter data from a submitted form. The most commonly uses method is the **getParameter(...)** method.

```
String getParameter(String name)
```

The getParamter method retrieves the value entered by the client. The value retrieved is based upon the widget's name. Once the values from the form are retrieved the data can be consumed by the developer. Program 5 is an example of processing the web form from the previous section and creating a dynamic HTML response.

Program Listing 5 Program To Process A Web Form

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

  // retrieve the name value pairs from request object
  String name = request.getParameter("petName");
  String type = request.getParameter("petType");

  // create an HTML response
  response.setContentType("text/html");
  PrintWriter out = response.getWriter();
  out.println("<html><body>");
  out.println("<h3>Pet Stuff</h3>");
  out.println("<p>;");
  out.println(name + " is a pretty name for a " + type");
  out.println("</p>;");
  out.println("</body>;</html>");
}
```

In this example the pet name and pet type are retrieved from the text widget and radio widget respectively. Notice that the value passed into the *getParameter(...)* matches the name of the individual widgets. These values are used as part of the HTML output.

Instead of retrieving each value from the request object, a *java.util.Map* can be used to contain all of the name/value pairs. The map can be retrieved using the **getParameterMap()** method.

```
Map getParameterMap()
```

All of the name value pairs are stored within the map and can be retrieved from it one at a time. The example above can be modified to use the map object.

```
Map nameValues = request.getParameterMap()
```



```
String name = (String)nameValues.get("petName");
String type = (String)nameValues.get("petType");
```

The *getParamter(...)* method is perfect for working with widgets that provide a single piece of data. The text box, text area, and radio buttons are examples of single value data. Some of the HTML widgets, the multi-select box and checkbox widget, will support multiple values for a selection. Below is an example of a checkbox form.

```
What news letters would you like to receive:<br/>
<input type="checkbox" name="news" value="Developer"/> Developer Tips<br/>
<input type="checkbox" name="news" value="SysAdmin"/> SysAdmin Tips<br/>
<input type="checkbox" name="news" value="Management"/> Management Tips<br/>
```

Since each element is a checkbox, it is possible to select none, one, two, or all of the different checkboxes when a form is submitted. It is possible to check any combination of the three boxes, but there is only one variable associated with all of the values. The question is how does one retrieve all of the values. The request object has a **getParameterValues(...)** method for these instances where multiple values are possible.

```
String [] getParameterValues(String name)
```

The *getParameterValues()* method returns all of the values associated with the name as a String array. The developer can iterate over the array to get all of the values associated with the name. Below is an example of using the method to process the previous checkboxes.

```
String [] newsletters = request.getParameter("news");

// create an HTML response
response.setContentType("text/html");
PrintWriter out = response.getWriter();
out.println("<html><body>");
out.println("<h3>Thank You</h3>");
out.println("<p>;");
out.println("Thank you for selecting the following newsletters: ");
for ( int x = 0; x < newsletters.length; x++ ) {
    out.print(newsletters[x] + " Tips ";
}
out.println("</p>;");
out.println("</body>;</html>");
```

In addition to the various ways for retrieving values, the request object provides a method, **getParameterNames(...)**, for retrieving the names from the name/value pairs.

```
Enumeration getParameterNames()
```


It returns an *java.util.Enumeration* with all of the widget names. The enumeration can be traversed to return each name.

4.2.3 Handling GET Data

POSTing isn't the only way to send data to the server for processing. The GET method also provides a means for sending parameter data. All parameter data for GET method get tacked to the end of the URL as a series of simple name value pairs.

```
http://nplus1.net/myapp/editItem.html?primaryId=5256&category=Sports
```

The name value pairs are preceded by a question mark. It marks the end of the URL and the beginning of the parameters. Each parameter is separated by an equal sign and each set of parameters are separated by an ampersand. In the example above there are two sets of parameters. The first is the *primaryId* which has a value of 5256. The second is the *category* which has a value of "Sports". These values can be processed like any web form data. Below is an example of processing these parameters.

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

// note that number data is capture as a string
String idData = request.getParameter("id");

// convert data to numerical primitive
int id = Integer.parseInt(idData);
```

It is important to remember that all parameters are Strings. If a developer needs for the value of a parameter to be a primitive data type, then they must convert from String to that type.

4.2.4 HTTP Header Data

The HTTP request header contains meta data about the browser which submitted the request. The meta data is generally known as the request headers. There are two methods for retrieving request header data.

```
String getHeader(String name)
long getDateHeader(String name)
```

The first method, **getHeader(...)** returns a String representation of the header. The second method, **getDateHeader(...)**, returns the date value as an integer which can easily be converted into a *java.util.Date object*. There are a wide number of possible headers available[5]. The example below retrieves two of these headers, the "User-Agent" and "Date".

```
String browser = request.getHeader("User-Agent");

int dateValue = request.getDateHeader("Date");
java.util.Date date = new java.util.Date(dateValue);
```

The "User-Agent" header returns the type of browser which submitted the request while the "Date" header returns the timestamp of when the request was sent by the client. The example above converts the timestamp integer into a *java.util.Date* object.

[5]More information on available headers can be found in a quick guide to HTTP headers http://www.cs.tut.fi/ jkorpela/http.html

A list of available headers can be retrieved programmatically by using the **getHeaderNames()** method.

```
Enumeration getHeaderNames()
```

The method returns an *java.util.Enumeration* of all the headers that the client sent to the server. Program listing 6 shows how to create a servlet that will display all of the HTTP's header names and associated values.

Program Listing 6 Display List of Headers

```
public class HeaderServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response)
                throws ServletException, IOException {

        // get list of header names
        Enumeration headers = request.getHeaderNames();

        // create an HTML response
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html><body>");
        out.println("<h3>Header Information</h3>");
        out.println("<p>;");

        // loop over names and print out the name / value pairs
        while ( headers.hasMoreElements() ) {
            String headerName = headers.nextElement();
            String headerValue = request.getHeader(headerName);
            out.println(headerName + "=" + headerValue + "<br/>;");
        }

        out.println("</p>;");
        out.println("</body>;</html>");
    }
}
```

4.2.5 URL Elements

The request object allows developers to retrieve information about the URL's uniform resource identifier. The URI is the section of the URL that appears after the server destination The URI for a Java web application is made up of three parts, the context , the servlet , and the path information.

The context path defines which web application is being accessed. The servlet path comes after the context path and identifies what servlet or JSP page is being accessed. The remaining portion of the URI is considered the path information. The best way to understand the parts of the URI is to review a few examples. The first example is a web application called "acme". It has the following servlet mapping within the web.xml.

```
<servlet-mapping>
    <servlet-name>Dummy</servlet-name>
    <url-pattern>/test/dummy.html</url-pattern>
</servlet-mapping>
```

Next look at the following URL which access the servlet

```
http://www.nplus1.net/acme/test/dummy.html?dummyId=369

Non URI: http://www.nplus1.net
Context Path:       /acme
Servlet Path:       /test/dummy.html
Path Information:   ?dummyId=369
```

In this example the URL before the URI is the address of the server. The context path is the path that maps to the web application. The servlet path is the information from the context to the desired resource. In this instance the servlet path is /test/dummy.html. The path information is everything after the path. In this case it is the URL parameters "?dummyId=369". The next example is a servlet mapping that uses a wildcard character.

```
<servlet-mapping>
    <servlet-name>My Servlet</servlet-name>
    <url-pattern>/test/myservlet/*</url-pattern>
</servlet-mapping>
```

The following example reviews the URL and breaks the URI into its three components.

```
http://www.nplus1.net/acme/test/myservlet/magic/foobie.html"

Non URI: http://www.nplus1.net
Context Path:       /acme
Servlet Path:       /test/myservlet
Path Information:   /magic/foobie.html
```


In this instance the servlet path only covers the portion of the URL up til it matches the wild card character. Everything after the wild card match is the path information. All three elements of the URI can be retrieved from the request object using the following methods.

```
String getContextPath()
String getServletPath()
String getPathInfo()
```

Program 7 is an example of a simple servlet that displays URI information.

Program Listing 7 Displays Different Segments of URI

```
public class URIServlet extends HttpServlet {
    public void doGet(HttpServletRequest request,
            HttpServletResponse response)
            throws ServletException, IOException {

        // create an HTML response
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html><body>");
        out.println("<h3>Header Information</h3>");
        out.println("<p>;");

        out.println("Context Path = " + request.getContextPath() + "<br/>");
        out.println("Servlet Path = " + request.getServletPath() + "<br/>");
        out.println("Path Info = " + request.getPathInfo() + "<br/>");

        out.println("</p>;");
        out.println("</body>;</html>");
    }
}
```


4.2.6 Servlet Context

The **ServletContext** is a mechanism by which developers can communicate with the servlet container. Each web application is assigned a ServletContext and can be retrieved from the HttpServlet by calling the **getServletContext()** method.

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    ServletContext context = this.getServletContext();
}
```

Since only one ServletContext is created by the application, it can be used as warehouse for global data within an application. It provides two methods for this purpose. One for setting a name value pair. Another for retrieving the value based upon the name.

```
void setAttribute(String name, Object value)
Object getAttribute(String name)
```

Below is a contrived example for getting and setting attributes within a context.

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
    ServletContext context = this.getServletContext();
    context.setAttribute("tempPath", "/var/tmp/myapp");
    ...
    String path = (String)context.getAttribute("tempPath");
}
```


4.2.7 Forwards And Includes

Another use for the ServletContext is the ability to retrieve and utilize the **RequestDispatcher**. The dispatcher provides a wrapper for resources on the server and can be used to direct a request to the specified resource.

The RequestDispatcher is retrieved from the context by passing the URI of the resource to be wrapped.

```
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        ServletContext context = this.getServletContext();
        RequestDispatcher dispatch =
            context.getRequestDispatcher("/anotherplace.html");
}
```

In this example the dispatcher is created to wrap the "/anotherplace.html" resource. The path starts with the root of the web application. The developer has two options for working with the resource. It can be included within the current request or control can be forwarded to the resource specified.

Includes A call to an **include(...)** takes the output from the wrapped resource and merges the results into the current servlet's output. The resource is literally dropped into the current location within the servlet. Traditionally includes have been used for things like web page headers and footers. Things that are used on multiple web pages. Rather than duplicate the sections of the page, it is possible to just include the common code into the current servlet.

Below is an example of including an outside resource. Notice that the include call must pass in the request and response objects. These objects can be accessed and modified by the included resource.

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

        PrintWriter out = response.getWriter();

        ServletContext context = this.getServletContext();
        RequestDispatcher dispatch =
                context.getRequestDispatcher("/anotherservlet.html");

        out.println("line before include<br/>");
        dispatch.include(request, response);
        out.println("line after include<br/>");
}
```

In this example a dispatch wrapper to another servlet is created. After the line "line before include" is created. Control leaves the current servlet and is passed to *anotherservlet.html*. Output from *anotherservlet.html* is inserted into the main servlet's output stream. After anotherservlet runs, control is returned to the main servlet. The main servlet finishes by printing "line after include". In effect, the *anotherservlet.html's* content is bracketed by the two lines from the primary servlet

Forwards A call to **forward(...)** takes control from the current servlet and passes it to the wrapped resource. In the process, any output that was created by the current servlet is flushed and no longer used. Traditionally forwards are used to pass control to a UI servlet after processing data from a web form. Below is an example of forwarding to another resource. Like the *include(...)* call, forwards must include the request and response object.

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

    String data = request.getParameter("someData");

    // do something with form data and send user to a results page

    ServletContext context = this.getServletContext();
    RequestDispatcher dispatch =
                context.getRequestDispatcher("/results.html");
    dispatch.forward(request, response);
}
```

This provides an excellent example of separating responsibilities. The servlet above is used to process the data while another servlet is used to manage the user interface.

Communicating Between Servlets

When doing a forward or include the request and response objects are included in the call. As already noted, these objects become available to the forwarded or included resource. This fact can be leveraged to provide a mechanism for transferring data between servlets. The request object has two methods to set name/value pairs that are identical to the ones from the ServletContext object.

```
void setAttribute(String name, Object value)
Object getAttribute(String name)
```

These can be used by the developer to set and retrieve name value pairs. Thus, he can set a request attribute in *servlet A* and retrieve the attribute in *servlet B* using the *include(...)* or *forward(...)* method. Below is an example of *servlet A*.

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

    String data = request.getParameter("someData");

    // store the data as a request attribute
    request.setAttribute("magicalData", data);

    ServletContext context = this.getServletContext();
    RequestDispatcher dispatch =
            context.getRequestDispatcher("/servletB.html");
    dispatch.forward(request, response);
}
```

In this case the request's attribute "magicalData" is filled with parameter data from the incoming request. Since the attribute is associated with the request object, it is passed to *servlet B* when the dispatch is called to forward processing. Therefore, the attribute "magicalData" can be accessed within *servlet B*. Below is an example of *servlet B* retrieving the data.

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

    String data = (String)request.getAttribute("magicalData");

    PrintWriter out = response.getWriter();
    out.println("<h2>The magical data is " + data + "</h2>");
}
```


4.3 Lab Activity

Part 1.

Create a servlet that creates a HTML form that prompts the user for the following information: first name, last name, and phone number.

Part 2.

Create a second servlet to handle the POST of the form created in the part 1. It should thank the user for filling out the form and then display the user's name and phone number.

Part 3.

Modify the servlet from part 2 to perform error handling. If the user doesn't enter a valid name or a phone number that is not 10 digits then forward the client to the form servlet created in part 1. The servlet should be modified to display an error message telling the user what fields were filled out incorrectly. This should be done with inter-servlet communication.

Part 4.

Modify the servlet from part 2 modify how the errors are handled. The servlet should redirect the user to the URL of the form servlet. Create a parameter to pass on the URL to tell the first servlet what error occurred. Modify the first servlet to display the appropriate error based upon this error code.

Notice how the URLs the client sees differs between part 3 and part 4.

Part 5. (Optional)

If the client tries to enter the URL of the processing servlet created in part 2 using a GET request have the servlet send a 404 error.

Part 6. (Optional)

Create a custom page not found HTML error page to display when a 404 error is returned by the servlet container.

Chapter 5

Session Management

Objectives

- Understand basics of session state management
- Learn how to retrieve and use the HttpSession object
- Understand how cookies work
- Learn how to set and retrieve cookie values

5.1 Session Management

As noted previously, the HTTP protocol is a stateless protocol. Each request is handled independently of any other request. The communication protocol doesn't provide any mechanism for tracking a user across multiple requests.

Since the protocol doesn't provide state management, it is up to the web container to create a mechanism from the tools available. The meta information transferred in the header of a HTTP message can be manipulated to create a state system allowing users to be tracked across multiple requests. The key is to force a unique identifier into the header information of the client's request.

If the same identifier is used with every request, then it is possible for the servlet engine to track a particular client. The way state is managed by the container is simple. The engine creates a HttpSession object and associates it with a unique identifier. The unique identifier is passed to the client and told to use it for every request. Since each session object is associated with a unique identifier, it is guaranteed to be a unique object which can be associated with a user. Therefore, the session object can be utilized to maintain state information between HTTP requests.

How does the client learn about the unique identifier? How does it know to place it within the HTTP header. The process is done with cookies. Cookies are domain specific name/value pairs. Which means that the name/value pair is associated with a particular domain. When the client makes a request upon that particular domain, it will place all of the cookie information associated with the domain name into the HTTP header.

Cookies were originally meant as a mechanism for storing state about the client. The problem is the state information would have to be sent every time a request on the server is made. A lot of cookie information could create significant overhead when making a request upon the server. The session object relieves this problem by storing all of the information associated with the client on the server side. All that is needed to get the session information is the unique identifier.

5.1.1 HttpSession Object

The **HttpSession** object is created and managed by the servlet container. The container is in charge of creating the unique identifier and sending the cookie to the server. The name of the cookie is JSSESSIONID and it is guaranteed to be unique for the servlet container. In addition the container manages the mappings of session id's to the actual HttpSession object simplifying the means in which developers retrieve the object. All the developer has to do is retrieve the session object from the HttpServletRequest object.

```
HttpSession getSession(boolean value)
```

The HttpSession object is not created until specifically told by the developer. The programmer must tell the request object when to create the session. The boolean argument in the **getSession(...)** method call is used to determine if the session should be created. If the session already exists it will be returned regardless of the boolean value. Otherwise if the value is "true" then a new session will be created. If the value is "false" then the call will return "null".

Once created, the HttpSession object allows the developer to track a user across multiple HTTP requests. The object maintains state and allows for a fluid session with the end user. One note, the server will not store these session objects indefinitely. The server will release the session object if a client doesn't make a new request with an allotted amount of time. The time is defined within the web.xml configuration using the session tags.

5.1.2 HttpSession Methods

Once the HttpSession object is established, the developer has a number of methods to manipulate the session data. The biggest use for the session object is to store data associated with the end user. The HttpSession object has three methods for storing and retrieving data.

```
Object getAttribute(String name)
void setAttribute(String name, Object value)
void removeAttribute(String name)
```

Program 8 is an example of creating a session object and populating it with a value.

Program Listing 8 Servlet A

```
public class ServletA extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response)
                throws ServletException, IOException {

        String data = (String)request.getAttribute("magicalData");
        int value = Integer.parseInt(data);

        HttpSession session = request.getSession(true);
        session.setAttribute("magicData", new Integer(value));
    }
}
```

The value stored in the session can be retrieved in a later call to a different servlet. Program 9 shows how to retrieve a session value.

Program Listing 9 Servlet B

```
public class ServletB extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response)
                throws ServletException, IOException {

        HttpSession session = request.getSession(true);
        int magicNumber = (Integer)session.getAttribute("magicData");

        PrintWriter out = response.getWriter();
        out.println("<h2>The magical data is " + data + "</h2>");
    }
}
```


In addition the developer can force the invalidation of a session with a call to the **invalidate()** method.

```
HttpSession session = request.getSession(true);
session.invalidate();
```

This function is mostly used in the real world to log a person off of a system so someone using the computer after him doesn't use the same session variable.

5.1.3 Guessing Game Example

A guessing game is a simple game. The computer randomly picks a number between 1 and 100. The client makes a guess in that range. If the number they guesses is lower than the computer's number then the machine returns a message with those sentiments. If the number guessed is higher than the computer's number an appropriate message is given back to the client. This continues until the client guesses the correct number.

When he or she guess the computer's random number, the program returns a message saying the user guessed it and how many attempts it took for the correct guess. The computer can then ask the end user if he or she wants to play again. The computer generates another random number and the game is played again.

For this game to be developed as a web application, it is important to be able to maintain state for each user. Somehow the random number needs to be retained as well as the number of guesses that are attempted. The HttpSession object is perfect for maintaining this type of state. The next couple of sections walk through the development of a web based guessing game.

Servlet Overview

The game can be broken down into two components, the user interface and the business layer. The business logic must manage the random number, determine if the guess is correct, and keep track of the number of guesses made by the client. The goal of the application will be to separate these elements within a single servlet.

The first step is deciding how to handle the business logic. There are two methods available for handling requests, the *doGet(...)* and *doPost(...)* methods. The *doGet(...)* method is executed whenever a GET request is made upon the servlet. Since that only occurs when the user is first connecting to the servlet, it provides an excellent opportunity to initialize the random number, reset the counter, and greet the user.

The *doPost(...)* method is called when a POST request is made. POSTs are made when forms are submitted. This method can be used to handle guesses by the end user. If the guess is not equal to the random number a hint is provide to the client telling them if the guess was too high or too low. If the number is guessed then a message needs to be displayed informing the end user of their correct guess and the number of attempts it took. The user should also be given the chance to play again.

Now that the business logic has been divided, it is time to turn the focus to the user interface. The UI functionality can be isolated by creating a *showPage(...)* method.

To make the *showPage(...)* method work it will need a some parameters passed to it. First, a PrintWriter from the HttpServletResponse is needed to generate the HTML. A message parameter would be useful as well. It can be used to greet the user and tell him about the results of his guess. Another useful parameter would be a boolean that would display a link to give the end user a chance to play again.

Below is an outline of the GuessingGame servlet.

```
public class GuessingGame extends HttpServlet {
    public void doGet(HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, IOException {
        // code for get goes here
    }

    public void doPost(HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, IOException {
        // code for post goes here
    }

    private void showPage(PrintWriter out, String message, boolean again) {
        // code for show page goes here
    }
}
```


doGet Method

The *doGet(...)* method is responsible for three things. Creating a random number, resetting the guess counter, and providing an initial welcoming message. Below is an example of the *doGet(...)* method.

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

    HttpSession session = req.getSession(true);
    int randomNumber = (int)Math.round(Math.random() * 100);
    session.setAttribute("randomNumber", randomNumber);
    session.setAttribute("guessNumber", 1);

    PrintWriter out = resp.getWriter();
    this.showPage(out, "Guess a number between 1 and 100", false);
}
```

The first thing that is necessary is to get a session object so state can be stored. A random number is generated using the *java.util.Math* functions and stored in the session object . In addition, a counter "guessNumber" is added to the session.

The writer is retrieved from the response object and passed to the *showPage(...)* method along with a greeting message. The last element tells the *showPage(...)* method that a link to replay the game is not necessary.

showPage Method

The *showPage(...)* method is responsible for creating the HTML used to generate the client's user interface.

```
private void showPage(PrintWriter out, String message, boolean again) {
    out.println("<html><body>");
    out.println("<h2>Guessing Game</h2>");
    out.println("<b>" + message + "</b>");
    if ( again ) {
        out.println("<p/><a href=\"\">Play Again?</a>");
    }
    else {
        out.println("<form method=\"post\" action=\"game.html\">");
        out.println("Guess: <input type=\"text\" name=\"guess\"/>");
        out.println("<input type=\"submit\" value=\"Submit Guess\"/>");
        out.println("</form>");
    }
    out.println("</body></html>");
}
```

The first thing it does is generate the necessary HTML elements to display a web page. The next step is to show the message passed into the method followed by a link to play again depending on the boolean value *again*. If the user isn't asked to play again, a HTML form is displayed to allow the client to guess a value and submit it.

doPost Method

The doPost(...) method is in charge of handling user guesses.

```
    public void doPost(HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, IOException {

        HttpSession session = req.getSession();
        int randomNumber = (Integer)session.getAttribute("randomNumber");
        int guessNumber = (Integer)session.getAttribute("guessNumber");
        String guessValue = req.getParameter("guess");
        int guess = Integer.parseInt(guessValue);
        PrintWriter out = resp.getWriter();

        if ( guess < randomNumber ) {
            this.showPage(out, "Your guess was too low", false);
        }
        else if ( guess > randomNumber ) {
            this.showPage(out, "Your guess was too high", false);
        }
        else {
            String msg = "You guessed right in " + guessNumber + " trys";
            this.showPage(out, msg, true);
        }
        session.setAttribute("guessNumber", ++guessNumber);
    }
```

The first step is to get the random number from the session. The user's input is converted to an int and compared to the random number. If the guess is less than the random number then a message is returned to the user that the guess was too low. If the guess is higher than the random number then a messages is returned that the guess was too high. If the numbers are equal then the user is told they had guessed correctly along with the number of guesses it took them. The boolean *again* is set to "true" so a link will be included to allow the user an opportunity to play again.

Complete Example

Program Listing 10 Guessing Game Part 1

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

public class GuessingGame extends HttpServlet {
   public void doGet(HttpServletRequest req, HttpServletResponse resp)
         throws ServletException, IOException {

      HttpSession session = req.getSession();
      int randomNumber = (int)Math.round(Math.random() * 100);
      session.setAttribute("randomNumber", randomNumber);
      session.setAttribute("guessNumber", 1);

      PrintWriter out = resp.getWriter();
      this.showPage(out, "Guess a number between 1 and 100", false);
  }
```


Program Listing 11 Guessing Game Part 2

```
  public void doPost(HttpServletRequest req, HttpServletResponse resp)
          throws ServletException, IOException {

      HttpSession session = req.getSession();
      int randomNumber = (Integer)session.getAttribute("randomNumber");
      int guessNumber = (Integer)session.getAttribute("guessNumber");
      String guessValue = req.getParameter("guess");
      int guess = Integer.parseInt(guessValue);
      PrintWriter out = resp.getWriter();

      if ( guess < randomNumber ) {
         this.showPage(out, "Your guess was too low", false);
      }
      else if ( guess > randomNumber ) {
         this.showPage(out, "Your guess was too high", false);
      }
      else {
         String msg = "You guessed right in " + guessNumber + " trys";
         this.showPage(out, msg, true);
      }
      session.setAttribute("guessNumber", ++guessNumber);
  }

  private void showPage(PrintWriter out, String message, boolean again) {
      out.println("<html><body>");
      out.println("<h2>Guessing Game</h2>");
      out.println("<b>" + message + "</b>");
      if ( again ) {
         out.println("<p/><a href=\"\">Play Again?</a>");
      }
      else {
         out.println("<form method=\"post\" action=\"game.html\">");
         out.println("Guess: <input type=\"text\" name=\"guess\"/>");
         out.println("<input type=\"submit\" value=\"Submit Guess\"/>");
         out.println("</form>");
      }
      out.println("</body></html>");
  }
}
```

5.2 Cookies

The HttpSession object is a powerful tool for create web applications. The session can be used to maintain conversational state with a client browser. It allows the developer to associate any Java object with a particular user. Even though the session object is vital for maintaining conversational state, it does have its limitations.

For one thing the state is maintained for a very limited period of time. The web.xml defines a timeout value. Once this value is exceeded between request the session is invalidated and no longer available. Because the cookie value for the SESSIONID is stored within the browser's memory, the session dies when the user closes down the browser.

Cookies are a means of associating information with the client for extended periods of time. The information is stored as name value pairs on the file system. Cookies are organized by domain name. When a request is made upon a particular domain, any cookies associated with the domain are passed along with the request.

Since cookies are stored on the client's machine they can last well beyond the lifespan of the browser. In Java, cookies are represented by the Cookie object. The management of cookies in the web container is simple. Cookies are created by passing in the name/value pair to the constructor.

```
Cookie cookie = new Cookie("userId", "johnd");
```

The cookie provides a number of methods for manipulating the cookie. Table 5.1 provides a list of cookie methods.

Tool	Description
String getName()	Retrieves the unique identifier associated with cookie
String getValue()	Retrieves the cookie's value
int getMaxAge()	Returns the maximum time a cookie is valid in seconds
void setValue(String value)	Sets the cookie's value
void setMaxAge(int seconds)	Sets the maximum time cookie is valid in seconds

Table 5.1: Cookie Methods

Notice there is no *setName(...)* method in the Table 5.1. That is because once the name is set in the constructor it can not be changed after. If a new name is desired, a new cookie must be created.

5.2.1 Creating New Cookie

Once cookies are created and defined they have to be sent back to the client's browser for storage. This process is done by adding the cookie to the response object using the **addCookie(...)** method defined below.

```
void addCookie(Cookie cookie)
```

Below is an example of creating a cookie and associating it with the response object.

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

    String userId = "johnd";
    Cookie userCookie = new Cookie("userId", userId);
    Cookie ageCookie = new Cookie("age", "29");

    response.addCookie(userCookie);
    response.addCookie(ageCookie);
}
```

5.2.2 Retrieving Cookie Values

After cookies are associated with the client browser, every request made by the browser will include those cookie name value pairs. That means the cookies should be available from the request object. They are made available by the **getCookies()** method defined below.

```
Cookie [] getCookies()
```

All of the cookies sent by the browser are included in the array of cookies returned by the call. It is up to the developer to find the proper cookie based upon the cookie's name attribute. The value attribute can then be retrieved from the cookie. The example below shows how this can be done to retrieve the values set in the previous section.

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

    String userId = "";
    int age = 0;

    Cookie [] cookies = request.getCookies();
    for ( Cookie cookie : cookies ) {
        if ( cookie.getName().equals("userId") ) {
            userId = cookie.getValue();
        }
        else if { cookie.getName().equals("age") ) {
            String value = cookie.getValue();
            age = Integer.parseInt(age);
        }
    }
}
```

Note that the user must loop through all of the cookies to find the one based on the name attribute of the cookie.

5.3 Lab Activity

Part 1.

The goal of the lab is to create a simple task list application. The first step is the creation of a servlet which will display the contents of the task list. The task list should be stored in the HttpSession object. After the task list, display a HTML form that allows the person to add a new task. A task should be composed of a priority, a due date, and text describing the task.

Part 2.

Modify the servlet in part 1 to process the submission of the add task form to add a new task to the task list. The new task should be displayed at the end of the task list.

Part 3.

Modify the servlet in part 1 to display check boxes next to each task item. A button should be added to the bottom of the list that allows the user to delete the tasks.

Part 4.

Implement the delete process by removing the selected items from the task list. Redisplay the list without the deleted task items.

Part 5. (Optional)

Modify the servlet in part 1 to add a link to the text of the task. The link should go to a new servlet that will display the task and allow the user to edit the task.

Part 6. (Optional)

Implement the process to edit the task and the redirect the user to the task list so it can display the list with the newly modified task in it.

Chapter 6

JavaServer Pages

Objectives

- Understand mechanics of JSP
- Learn how to use JSP scriptlets and expressions
- Know how to use implicit JSP variable
- Understand the various JSP directives
- Learn how to use JSP error handling
- Understand the JSP expression language

6.1 JavaServer Pages

JavaServer Pages, known as JSP, is a simplification of the Servlet platform. It allows developers an environment to quickly generate dynamic content without the overhead of developing with Servlets. JSP is a set of special tags which allow server based Java code to be embedded within an HTML page. The code is executed on the server side and the resulting HTML is returned to the client.

JSPs have the ability to communicate to the entire range of Java technologies, including interaction with Servlets, Enterprise Java Beans, Databases, and other server based resources. These resources can be accessed at run time to create dynamic HTML content for the end user. While it is possible to develop entire web application with JSP, the technology is generally used only to provide the presentation layer for the web application.

6.1.1 Mechanics of JSP

With Servlets, the URL of the dynamic page is mapped in a web.xml file. The servlet container is responsible for making sure the request is submitted to the appropriate Servlet. JSPs work differently. The URL for a JSP is not defined in the web.xml file. It is based upon the the directory structure within the WAR file. The servlet container looks within the structure of the WAR file for the specific JSP file.

Once the file is discovered for the first time, it is loaded into memory and converted into a Servlet. The generated Servlet is compiled on the fly and the JSP URL is mapped to the newly generated Servlet. Once the Servlet class is compiled an instance of the resulting class is loaded into memory. The incoming request is passed to the newly created Servlet for processing.

The second time the URL is called, the container checks to see if the time stamp on the JSP file is newer than the mapped URL within the container. If the time stamp is unchanged, then the previously created Servlet is used to handle the new request.

With a JSP, the first time the page is accessed it will take longer for the server to respond because the underlying servlet must be created and loaded into memory. All subsequent requests on the JSP will be much faster because the servlet has already been generated and loaded into memory. The process can be sped up by pre-compilation of the JSP pages. A process which converts all of the JSP pages into servlets and compiles them into class files. A process which will greatly increase the speed of first access.

6.1.2 JSP Basics

A JSP page is a mixture of server side coding along with HTML mixed into a single web page. When a request for a JSP is made, the server side Java within the page is executed on the server and the resulting HTML is returned to the end user. There are three basic tags used to denote JSP code. They are the scriptlet, expression tags, and directive tags. The following sections will focus on the first two types of tags.

JSP Scriptlet Tags

Scriptlets allow developers to embed Java commands within a JSP page. These commands are executed on the server and are used to create the resulting dynamic HTML page. The scriptlet tags are specified using the <% and %> symbols. The Java commands are embedded within these two tags.

Any valid Java statements can be placed with the scriptlet tags. Typically, the Java code is used to create conditionals, looping constructs, manipulate HTML header data, connect to databases, and interface with Enterprise Java Beans. Anything that can be done within the Java language can be placed within these tags.

Java code is generally used to create conditionals and looping constructs, manipulate HTML header data, connect to databases, or interface with Enterprise Java Beans. Anything that can be done within the Java language can be done in between these tags. Below is an example of Java code embedded within an HTML page.

```
<html>
    <%
        String data = "foobie";
        int x = 2;
        if ( x < 5 ) {
            data = "more foobies";
        }
    %>
<body>
<h2>This is a test</h2>
</body>
</html>
```

The example above is not very interesting because the code doesn't interact with the HTML. What makes scriptlets interesting is how they can be interspersed between HTML tags to create dynamic HTML. The concept is most prevalent with looping and conditional constructs. The following example uses looping to generate dynamic HTML.

```
<ul>
<%
    for ( int x = 0; x < 5; x++ ) {
%>
<li>this is a bullet point</li>
<%
    }
%>
</ul>
```


Here the for loop is split between two scriptlet tags. This is not only legal , but is in fact desired. It allows the HTML in between the scriptlets to be repeated multiple times. The output from this code is shown below.

```
<ul>
    <li>this is a bullet point</li>
    <li>this is a bullet point</li>
    <li>this is a bullet point</li>
    <li>this is a bullet point</li>
    <li>this is a bullet point</li>
</ul>
```

Study this example until it makes sense. It is important to grasp the concept of code spread across multiple scriptlet tags intermingled within HTML. The previous example can be expanded upon to use conditionals for determining the HTML to display.

```
<ul>
<%
    for ( int x = 0; x < 5; x++ ) {
        if ( x % 2 == 0 ) {
%>
<li>this is an even bullet point</li>
<%
        }
        else {
%>
<li>this is an odd bullet point</li>
<%
        }
    }
%>
</ul>
```

In this case the code iterates the value of x between 0 and 4. The value of x during each loop determines what HTML value is generated. In this example the resulting output is shown below.

```
<ul>
    <li>this is an even bullet point</li>
    <li>this is an odd bullet point</li>
    <li>this is an even bullet point</li>
    <li>this is a odd bullet point</li>
    <li>this is an even bullet point</li>
</ul>
```

Why does the even bullet point come first? Looking at the code above the value of x is first set to 0. When the first conditional is evaluated it is determined that 0 is an even value and thus the even bullet point is embedded into the resulting HTML.

JSP Expressions Tags

The JSP scriptlet tags are nice for embedding Java logic within a web page, but what if a developer wants to embed a dynamic value in the HTML? The JSP expression tags provides the developer with a mechanism to accomplish the task. Expression tags take any valid Java expression and evaluates the expression to find its string value. Primitives are automatically converted to strings. The resulting String replaces the expression tag within the final HTML returned to the client.

The expression tag is opened with <%= and closed with a %>. In between, is where the Java expression is placed. Below are two simple examples of using the expression tag.

```
<p>This is a <%= "simple string" %></p>
<p>The value of 45 + 23 = <%= 45 + 23 %></p>
```

The first line embeds the String "simple string" within the HTML and the second line evaluates 45 + 23 and converts the resulting value into a string and embeds it. Below is the output from the example.

```
<p>This is a simple string</p>
<p>The value of 45 + 23 = 68</p>
```

One important note about expressions is the lack of semicolon. Semicolons in Java are used to define a statement. A statement can be composed of an expression, but an expression is not a statement. An expression must evaluate to a value and is not a statement of code. Thus the semicolon can not be used within an expression tag. The reason for this becomes apparent later when the process of converting JSPs to Java Servlets is explained.

On top of simple expressions, the JSP expression tag can use previously defined variables. The variables are normally defined within previous JSP scriptlets. Below is an example of a using values defined in a scriptlet within a series of expressions.

```
<%
    int x = 7;
    int y = 2;
%>
<html>
    <body>
        <h2>Example of Expressions</h2>
        <%= x %><br/>
        <%= y %><br/>
        <%= x * y %><br/>
        <%= x - 18 + y %><br/>
        <%= "x = " + x %><br/>
    </body>
</html>
```

In this example a scriptlet tag is used to define the values of x and y. The rest of the page uses various expression tags to generate various outputs using these variables. The resulting HTML is shown below.

```
<html>
    <body>
        <h2>Example of Expressions</h2>
        7<br/>
        2<br/>
        14<br/>
        -9<br/>
        x = 7<br/>
    </body>
</html>
```


In addition, expression tags can be embedded within scriptlet conditionals and loops. The following example extends the previous section's looping example. It puts the value of "odd" or "even" next to the value of the loop.

```
<ul>
<%
    String oddEven = "";
    for ( int x = 0; x < 5; x++ ) {
        if ( x % 2 == 0 ) {
            oddEven = "even";
        }
        else {
            oddEven = "odd"
        }
%>
    <li><%= x %> is an <%= value %> number</li>
<%
    }
%>
</ul>
```

In this instance the loop goes from zero to four. Each time through the loop the value of loop variable is evaluated to see if it is an odd or even value. The String value of odd or even is placed within String variable *oddEven*. The value of x and the value of *oddEven* are then placed within the resulting HTML The process is repeated until the loop is exhausted. The resulting HTML is shown below.

```
<ul>
    <li>0 is an even number</li>
    <li>1 is an odd number</li>
    <li>2 is an even number</li>
    <li>3 is an odd number</li>
    <li>4 is an even number</li>
</ul>
```


Comments

This is a good point to talk about embedding comments within a JSP page. The <%- - and - -%> tags are used to indicate comments.

```
<%-- Example of a JSP comment --%>
<%--
Example of a multi line JSP comment
--%>
```

Comments are notes that can be attached to a JSP page to provide information about the dynamic portions of the page, but do not show up within the HTML. JSP comments are removed at the server level and thus never appear in the resulting HTML. They are only available within the JSP source code.

6.1.3 Implicit Variables

When working with Servlets all of the business logic for the page was placed inside either the *doGet(...)* or *doPost(...)* methods. Each method had a HttpServletRequest and HttpServletResponse object passed into it. As studied previously, these objects are used to get data from the incoming request and place information into the outgoing request. They are an integral part of development of web applications.

These objects are necessary within a JSP page. How else could one get parameter data from the incoming request without it? How else can one access the HttpSession object? The problem is solved in JSP by creating a set of implicit or predefined variables. Table 6.1 provides a list of implicit variables that are available within a JSP page. The table maps each variable to the object type they represent.

Implicit Variable	Object Class
request	HttpServletRequest
response	HttpServletResponse
out	PrintWriter from HttpServletResponse
application	ServletContext
session	HttpSession from HttpServletRequest
config	ServletConfig
page	this
exception	JspException

Table 6.1: Implicit Variables And Their Corresponding Classes

Two of the variables **out** and **session** are special in that they are derived from the HttpServletRequest and HttpServletResponse objects. The *out* variable is created from a call to the response's *getWriter()* method. The *session* variables is created from a call to the request's *getSession()* method.

```
PrintWriter out = response.getWriter();
HttpSession session = request.getSession();
```

Each of these variables are available within any JSP scriptlet or JSP expression tag. These variables are created and initialized by the JSP engine.

Below is an example of using implicit variables within a JSP page.

```
<%
    response.setContentType("text/html");
    String iters = reqeust.getParameter("iterations");
    String name = (String)session.getAttribute("name");
%>
<html>
    <body>
        <h3>Say Hello</h3>
        <%
            int iterations = Integer.parseInt(iters);
            for ( int x = 0; x < iterations; x++ ) {
                out.println("Hello " + name + "<br>");
            }
        %>
    </body>
</html>
```

The example above uses the **response**, **request**, *session*, and *out* objects. The *response* object is used to set the content type of the page. The *request* object is used to get the submitted parameter "iterations" and uses the value to set the number of times to loop. The *session* object is used to retrieve the name of the user from the client's session. The *out* variable is used to place the "Hello <name>" string into the HTML output stream. Generally, the expression tags are used for including data within HTML rather than the out variable.

The next example removes the *out* variable from the previous sample and replaces it with an expression tag.

```
<%
    response.setContentType("text/html");
    String iters = reqeust.getParameter("iterations");
%>
<html>
<body>
<h3>Say Hello</h3>
<%
    int iterations = Integer.parseInt(iters);
    for ( int x = 0; x < iterations; x++ ) {
%>
<p>Hello <%= session.getAttribute("name") %></p>
<%
    }
%>
</body>
</html>
```


It is worth noting that the *session.getAttribute(...)* method returns the value of "name" as an object and not a String. The reason the expression works is that the Object's *toString()* method is called to find the String value to place within the HTML.

6.1.4 Include Directive

Java provides a number of directives for performing special operations upon the JSP page. Each directive uses the <%@ %> tags to inform the JSP page it is a special operation. The **include** directive allows the developer to incorporate external files into the current JSP. The syntax for the imperative tag is shown below.

```
<%@ include file="include/header.html" %>
<%@ include file="/mystuff/include/stuff.jsp" %>
```

Files are inserted into the JSP page at the point where the directive is defined. In fact, they are imported before server processing begins. Thus any JSP page that is included will be processed as if they were part of the original JSP page.

In the example above, there are two different path types used for the included file. The first, without a leading "/", is known as a relative path. It is found in relationship to the calling JSP. The header.html would be within the include directory. The include directory must be in the same directory as the JSP page which calls the include.

The second example, the one with the leading "/" is a full path. The *stuff.jsp* file is found off the root of the web application in the "mystuff" and "include" directories.

A common practice for using the include directive is separate web page creation. A header and footer elements can be designed to provide a framework for the web page. These elements can be included in each JSP to provide a consistent look and feel for the web application. Since most of the look and feel is controlled in these two files, it is easier to maintain and update the site's appearance.

Below is an example of a *header.jsp* page.

```
<%
String header = "Guest Account"
String user = (String) user.getAttribute("user");
if ( user != null ) {
        header = "User " + user;
}
%>
<html>
   <body>
      <h2><%= header %></h2>
```

And here is an example of a *footer.html* page.

```
    &copy; n + 1, Inc.
    </body>
</html>
```


These can now be included into any page on the site to combine all of the elements into a page with a consistent look and feel.

```
<%@ include="header.jsp" @>

<b>Counter Stuff</b><br/>
<%
    for ( int x = 0 ; x < 11; x++ ) {
%>
    <%= x %>^3 = <%= x * x * x %><br/>
<%
    }
%>

<%@ include="footer.html" @>
```

This is the equivalent of creating the following JSP page.

```
<%
    String header = "Guest Account"
    String user = (String) user.getAttribute("user");
    if ( user != null ) {
        header = "User " + user;
    }
%>
<html>
    <body>
        <h2><%= header %></h2>

        <b>Counter Stuff</b><br/>
        <%
            for ( int x = 0 ; x < 11; x++ ) {
        %>
            <%= x %>^3 = <%= x * x * x %><br/>
        <%
            }
        %>

        &copy; n + 1, Inc.
    </body>
</html>
```

What makes the include useful is the ability to remove duplicated code between JSP pages.

6.1.5 Page Directive

The **page** directive is used to apply attributes to a JSP page. It has the following format.

```
<%@ page [attributes] %>
```

There are a number of different attributes that can be associated with a JSP page. Two of the most common are the *import* and *contentType* attributes. *contentType* is used to set type of content returned by the JSP page. If the content is HTML then the type should be set to "text/html". Below is an example of setting it for a page.

```
<%@ page contentType="text/html" %>
```

Another common page element is the *import* option. It allows elements in any Java package be imported into the current page. It performs the same function as the *import* statement in a Java class file. The only difference is multiple imports can be set at one time. Below is an example of using the import attribute.

```
<%@ page import="java.util.LinkedList, java.sql.*" %>
```

What follows is an example of using these elements inside a JSP page.

```
<%@page contentType="text/html"%>
<%@page import="java.util.LinkedList, java.util.Iterator"%>
<%
    LinkedList<String> myList = new LinkedList<String>();
    myList.add("one");
    myList.add("two");
    myList.add("three");
%>
<html>
    <body>
        <h2>List of elements</h2>
        <%
            Iterator<String> iterator = myList.iterator();
            while ( iterator.hasNext() ) {
        %>
            <%= iterator.next() %><br/>
        <%
            }
        %>
    </body>
</html>
```


6.1.6 JSP Error Handling

It is possible to use try / catch blocks to handle errors within JSP scriptlet tags. But what happens if an exception occurs that isn't within a try / catch block? The JSP engine can't allow an exception to go uncaught. Otherwise the servlet engine would crash. Therefore the JSP engine must catch all possible exceptions. In fact all exceptions are caught and the thrown exception is wrapped within a **JspException**.

The creator's of Java Server Pages added a mechanism to work with these unhandled exceptions. The *page* directive has an *errorPage* attribute that allows the developer to specify a JSP page to designate as an error page. If an uncaught exception occurs then the JSP engine will redirect to the specified error page. Below is an example of telling the JSP where to direct errors.

```
<%@page errorPage="my_error.jsp"%>
<html>
    <body>
    <h2>List of elements</h2>
    <%
        int [] nums = { 5, 2, 4 };
        for ( int y = 0 ; y < 5; y++ ) {
    %>
        <%= nums[y] %><br/>
    <%
        }
    %>
    </body>
</html>
```

Not any page can be designated as an error page. A JSP that is going to be used as an error page must define itself as an error page. That is accomplished by using the page directive with the *isErrorPage* attribute. In the previous example, *my_error.jsp* was defined as the error page. That means that the top of the *my_error.jsp* must contain the page directive with *isErrorPage* set to "true" otherwise an error will occur and the engine will return a 500 server error. Below is an example outline of an error page.

```
<%@page isErrorPage="true"%>
<html>
    <body>
        ...
    </body>
</html>
```


Once a page is defined as an error page, it gains the use of the implicit variable **exception**. The *exception* variable is only available on error pages. It represents the JspException that occurred within the main page. Below is an example of creating an error page that shows what type of exception and shows the elements of the stack trace.

```
<%@ page isErrorPage="true" %>
<html>
    <body>
        <h2>Error Page</h2>
        An error has occurred with the system. <br>
        <hr>
        <h3><%= exception.getMessage() %></h3>
        <b>Debugging Information</b> -
                Time Stamp (<%= new java.util.Date() %>)
        <% StackTraceElement [ ] ste = exception.getStackTrace(); %>
        <table border="1" cellspacing="0" cellpadding="4">
            <tr>
                <th>Class</th>
                <th>Method</th>
            </tr>
            <%
                for ( int i = 0; i < ste.length ; i++ ) {
            %>
                <tr>
                    <td><%= ste[i].getClassName() %></td>
                    <td><%= ste[i].getMethodName() %></td>
                </tr>
            <% } %>
        </table>
    </body>
</html>
```

While the example above will display the stack trace for the error, it is not a good idea to have an error page in production that shows the stack trace. It provides hackers with information they can use to potentially penetrate the system. A developer should log the error and return a friendly error message to the user.

6.1.7 How JSPs Are Converted To Servlets

One thing that can provide better insight into how a JSP page works is by understanding how JSP pages are converted into Servlets. Take the following JSP page used in a previous example.

```
<%
    response.setContentType("text/html");
    String iters = request.getParameter("iterations");
%>
<html>
    <body>
        <h3>Say Hello</h3>
        <%
            int iterations = Integer.parseInt(iters);
            for ( int x = 0; x < iterations; x++ ) {
        %>
            <p>Hello <%= session.getAttribute("name") %></p>
        <%
            }
        %>
    </body>
</html>
```

The web container takes the page and converts it into a Servlet. It starts by using a template for a Servlet and merging the content of the JSP into the template. Rather than supply a *doGet(...)* and *doPost(...)* implementation, the JSP implements the servlet interface's *service(...)* method.

```
public void service(HttpServlet request, HttpServletResponse response) {
    HttpSession session = request.getHttpSession();
    PrintWriter out = response.getWriter();

    // JSP page is converted here
}
```

Notice how implicit variable are "created" for the JSP page. The *request* and *response* variables are provided by the method signature. The *session* and *out* implicit variables are generated at the top of the method.

The conversion process is simple. Each line of HTML is placed within an *out.println()*. The lines with scriptlet tags are just copied into the servlet as is. The expression tags are placed within *out.print()* lines. Therefore the example above would be converted into the following servlet.

```
public void service(HttpServletRequest request,
                    HttpServletResponse response) {
    HttpSession session = request.getHttpSession();
    PrintWriter out = response.getWriter();

    // JSP page is converted here
    response.setContentType("text/html");
    String iters = request.getParameter("iterations");
    out.println("<html>");
    out.println("<body>");
    out.println("<h3>Say Hello</h3>");
    int iterations = Integer.parseInt(iters);
    for ( int x = 0; x < iterations; x++ ) {
        out.print("<p>Hello ");
        out.print( session.getAttribute("name") );
        out.println("</p>");
    }
    out.println("</body>");
    out.println("</html>");
}
```

The resulting servlet is compiled, deployed and mapped to the original JSP page.

6.2 Expression Language

Now that the reader has a basic understanding of JSP pages, it is time to turn attention to a quirky feature that has become an integral part of managing user interfaces. The feature is known as the expression language or EL for short.

The expression language was introduced with the Java Standard Template Library, JSTL. It was meant as an easy method for accessing data from objects known as Java Beans. The hope was to reduce the amount of scriptlet and expression tags used within a web page. The primary goals of EL was to create a simple language that was quick to write, simple to learn, and easy to maintain (especially for non-Java programmers).

As time went on, the benefits of EL delivered and it turned out to be a very popular tool. So popular that it was added to JSP 2.0 specification.

6.2.1 EL Basics

A section of EL within a JSP page is designated by using a dollar sign and braces `${ }`. Inside of the braces, a simple expression is defined for evaluation. Any variable used within the expression is known as an identifier. An identifier can be either an implicit object or a variable.

If it is a variable, a special series of steps are preformed to find the value of the variable. The expression manager searches different scopes within the JSP page to find the value to associate with the variable. It searches in order the page object, request object, session object, and application context to find a value for the variable. It does this by performing a request upon each object's *getAttribute(...)* method. It returns the first value that it finds.

For example, take the variable *foobie*.

```
${foobie}
```

JSP would make a call to the page object's *getAttribute(...)* method and see if a value exists. If it doesn't it would move on to the request object and so on until the application level variable is reached.

```
page.getAttribute("foobie");
request.getAttribute("foobie");
session.getAttribute("foobie");
```

Once the object is found, it returns the object to the expression for evaluation. The results of the expression's evaluation is then displayed within the JSP page. If the returned value is a primitive or a String the value can be displayed directly. Complex objects that hold multiple values provide special nomenclature for accessing the various values.

6.2.2 Accessing Data From Complex Variables

The EL searches through the various scopes to find an object associated with the name. The data contained in the object can then be accessed directly from EL. The "." operator allows EL to evaluate a property of the object. Properties are accessed using the rules for creating JavaBeans.

JavaBeans

JavaBeans are Java objects that conform to a certain specification. First they must all have a default constructor. Secondly, the class must access its properties using get/set methods. Lastly the object must be serializable. EL really only cares about the second rule. The "." operator is used to access properties that use the get/set methods.

Below is a simple example of a JavaBean like object.

```
public class Person {
    private String firstName;
    private String lastName;
    private int age;

    public String getFirstName() {
        return this.firstName;
    }
    public String getLastName() {
        return this.lastName;
    }
    public int getAge() {
        return this.age;
    }
    public void setFirstName(String name) {
        this.firstName = name;
    }
    public void setLastName(String name) {
        this.lastName = name;
    }
    public void setAge(int age) {
        this.age = age;
    }
}
```


Accessing Data

As previously noted, the properties are accessed using the "." operator. Below is an example of accessing values from a Person object within a JSP. In the example the Person is stored as a variable *person* within a session object.

```
<html>
    <body>
        <h2>Personal Info</h2>
            ${person.firstName} ${person.lastName}
            is ${person.age} years old.<br/>
    </body>
</html>
```

The same code using scriptlet and expression tags is shown below.

```
<%
    Person person = (Person)session.getAttribute("person");
%>
<html>
    <body>
        <h2>Personal Info</h2>
            <%= person.getFirstName() %> <%= person.getLastName() %> is
            <%= person.getAge() %> years old.<br/>
    </body>
</html>
```

Which version is cleaner? The ease of understanding is the primary reason EL has made its way into the JSP standard.

Accessing Objects With Objects

It is possible to access objects with objects using the "." operator. Each "." accesses properties of the previous object. The best way to understand this concept is through an example. What if the Person object was used to define the president of a professional group object. The professional group might look like the following.

```
public class ProfessionalGroup {
    private String name;
    private Person president;

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

Below is an example of a JSP that takes the group and displays the name of the group's president.

```
<html>
    <body>
        <h2>${group.name} Directory</h2>
        President: ${group.president.firstName}
        ${group.president.lastName}<br/>
    </body>
</html>
```

In this example, group represents a ProfessionalGroup object. Take the expression *group.president.firstName*: The first "." operator accesses the *president* property which is a Person object. Since the result is an object, it is possible to access its properties. The second "." operator does exactly this and accesses the person's *firstName* property.

6.2.3 Variables as Maps, Lists, and Arrays

Some of the common data types a developer might use are Maps, Lists, and Arrays. Unfortunately, none of these data types use JavaBean standards for storing and retrieving data. Fortunately, EL provides a mechanism for accessing these data types. Each are treated in similar fashion to accessing an array in Java.

Lists and Arrays are treated identically by EL. They are both accessed as if they were Java arrays. Which means they use indexes within brackets to access individual elements of the list. The index is zero based. The example below shows how EL accesses a list.

```
<%@ page import="java.util.*" %>
<%
    ArrayList<Person> peeps = new ArrayList<Person>();

    Person person1 = new Person();
    person1.setFirstName("John");
    person1.setLastName("Doe");

    Person person2 = new Person();
    person2.setFirstName("Beth");
    person2.setLastName("Smith");

    peeps.add(person1);
    peeps.add(person2);
    request.setAttribute("somePeeps", peeps);
%>
<html>
    <body>
        <h2>List of People</h2>
        <p>
            Person 1: ${somePeeps[0].firstName}
                      ${somePeeps[0].lastName}<br/>
            Person 2: ${somePeeps[1].firstName}
                      ${somePeeps[1].lastName}<br/>
        </p>
    </body>
</html>
```

Like embedded JavaBeans, it is possible to access properties from the list. The example above retrieves each element from the list and accesses the properties from the element using a ".".

The other data type used frequently is the map. Since a map is not a list, elements within the map can't be accessed using a simple integer based index. Instead the user can use the "key" value in the map within the brackets to get back the value associated with the key.

Below is a simple example that populates a map and then displays it using EL. It is worth noting that string values in EL are usually surrounded with single quotes.


```
<%@ page import="java.util.*" %>
<%
    HashMap<String,String> person = new HashMap<String,String>();
    person.put("first name", "Beth");
    person.put("last name", "Smith");
    request.setAttribute("person", person);
%>
<html>
    <body>
        <h2>List of People</h2>
        <p>
            Person: ${person['first name']} ${person['last name']}<br/>
        </p>
    </body>
</html>
```

Of course it is possible to embed one data types within another. The internal object would be accessed using the brackets as if it where a single entity. Thus the result would have the appearance of a multi-dimensional array. The final examples shows embedding a map within a list.

```
<%@ page import="java.util.*" %>
<%
    ArrayList<Person> peeps = new ArrayList<Person>();

    HashMap<String,String> person = new HashMap<String,String>();
    person.put("first", "John");
    person.put("last", "Doe");
    peeps.add(person);

    person = new HashMap<String,String>();
    person.put("first", "Beth");
    person.put("last", "Smith");
    peeps.add(person);

    request.setAttribute("somePeeps", peeps);
%>
<html>
    <body>
        <h2>List of People</h2>
        <p>
            Person 1: ${somePeeps[0]['first']} ${somePeeps[0]['last']}<br/>
            Person 2: ${somePeeps[1]['first']} ${somePeeps[1]['last']}<br/>
        </p>
    </body>
</html>
```


6.2.4 Implicit Objects

EL has the ability to access the implicit objects of a JSP page. Table 6.2 provides a list of implicit objects along with their JSP equivalents.

Implicit Variable	JSP Equivalent
`${param.name}`	request.getParameter(”name”)
`${paramValues}`	request.getParameters()
`${header.name}`	request.getHeader(”name”)
`${headerValues}`	request.getHeaders()
`${cookie.name}`	request.getCookie(”name”)
`${initParam.name}`	config.getInitiParameter(”name”)

Table 6.2: Implicit EL Variables

Below is an example of a simple HTML form that allows the user to input a first name, last name, and phone number. The data is sent to the *results.jsp* page.

```
<form method="post" action="results.jsp">
    First Name: <input type="text" name="firstName"/><br/>
    Last Name: <input type="text" name="lastName"/><br/>
    Phone Number: <input type="text" name="phone"/><br/>
    <input type="submit" value="Show Results"/>
</form>
```

The *result.jsp* page below shows how to use the EL implicit object to display the first name, last name, and phone number.

```
<h2>Form Information</h2>
<ul>
    <li>First Name: ${param.firstName}</li>
    <li>Last Name: ${param.lastName}</li>
    <li>Phone Number: ${param.phone}</li>
</ul>
```


6.2.5 EL Operators

EL can do more than just retrieve data from an object or a variable. It can be used to perform simple calculations and comparisons. First off, it can perform all of the math operators supported by Java. These include addition, subtraction, multiplication, division, and the modulus operator. Below are three example of using math operations within an EL statement.

```
<p>5 + 8 = ${5 + 8}</p>
<p>x - y = ${x + y}</p>
<p>x + 3 = ${x * 3}</p>
```

On top of simple math operations, EL can be used to perform comparison operations. All of the basic Java comparison operators are available in EL. The <, >, <=, >=, ==, !=, &&, — —, ! call all be used. An additional **empty** operator is available. It is used to check to see if the variable exists. It is similar to checking to see if an object is *null* or an empty string.

At this point there is not much use for EL comparison operators. The result of the operation is "true" or "false" which isn't useful to place within a web page. Comparative expressions show their use when dealing with the Java Standard Template Library or JSTL. It is used for conditional comparisons within a web page. JSTL is covered in a future module. Another thing to note. The == and != comparison can be used to compare String values.

Another conditional operator that can be used is the tertiary operator.

```
a = <comparison> ? x : y
```

The tertiary operator can be used to assign a value based upon the results of the comparison. Below is an example of a select box the allows the user to select "Dog", "Cat", and "Fish".

```
<select name="pet" size="1">
    <option value="none">-- None --</option>
    <option value="dog">Dog</option>
    <option value="cat" selected>Cat</option>
    <option value="fish">Fish</option>
</select>
```


In this example the "cat" option is preselected. The selection can also be made based upon a simple comparison using the tertiary operator. In the example below the parameter value is used to determine which option are selected.

```
<select name="pet" size="1">
    <option value="none" ${param.petType == 'none' ?
                           'selected' : ''}>-- None --</option>
    <option value="dog" ${param.petType == 'dog' ?
                           'selected' : ''}>Dog</option>
    <option value="cat" ${param.petType == 'cat' ?
                           'selected' : ''}>Cat</option>
    <option value="fish" ${param.petType == 'fish' ?
                           'selected' : ''}>Fish</option>
</select>
```

In the first option line the EL makes a call to request.getParameter("petType") to get the value from the header. If the value is true then the String value 'selected' is placed into the option tag. If it false then an empty string is placed into the option tag. The comparison is made for each parameter type.

6.3 Lab Activity

Part 1.

The goal of the lab is to create a simple task list application using JSP pages. The first step is the creation of a JSP which will display the contents of the task list. The task list should be stored in the HttpSession object. Use EL to display the contents of the list.

After the task list, display a HTML form that allows the person to add a new task. A task should be composed of a priority, a due date, and text describing the task. Use a task object bean to store this data.

Part 2.

Modify the JSP from part 1 to use an include for the header and footer of the page. Create a header.jsp and footer.jsp to provide the header and footer respectively.

Part 3.

Create a add task JSP page that handles the submission of the add task form. The JSP should take the data from the form and create a new task to add to the task list. Redirect back to the task list JSP to display the modified task list.

Part 4.

Modify the JSP in part 1 to display check boxes next to each task item. A button should be added to the bottom of the list that allows the user to delete the tasks.

Part 5.

Create a delete tasks JSP that removes the selected items from the task list. Redirect the user back to the task list JSP to display the task list without the deleted task items.

Part 6. (Optional)

Modify the JSP in part 1 to add a link to the text of the task. The link should take the user to a new JSP that will display the task and allow the user to edit the task's information.

Part 7. (Optional)

Implement a process task JSP that edits the entry in the task list and redirects the user back to the task list which should display the list with the freshly modified task within it.

Chapter 7

Tag Libraries

Objectives

- Understand what a tag library is
- Understand JSTL
- Learn how to use core tag libraries
- Leran how to use format tag libraries
- Learn how to create custom tag libraries

7.1 Tag Libraries

What is a Tag Library? Technically it is a set of tags that are logically grouped by functionality to form a library of tags. Functionally, a tag is an extension of the JSP concept. It allows a user to define a JSP XML element that is embedded within a JSP page. The element provides a container for programming logic to be applied to the page. The tag will not only simplify the creation of the JSP pages, but if used properly will remove the need for JSP scriptlets within a page.

Since tags are XML elements, they take on all of the characteristics of any XML tag. A namespace is matched with the tag to tell which tag library the tag is associated. Attributes can also be assigned to the tags to allow for the passing of dynamic values for processing by the tag. In addition to attributes, the tags can access data from both the implicit and explicit objects defined within a JSP page.

Generally, tags are used for two functions. First and foremost they are used to display data to the end user. The second function is the ability for tags to intercommunicate with each other so that more complex solutions can be developed.

The basic format of a Java Tag follows.

```
<ns:tagName attr1="value1" attr2="value2"> </ns:tagName>
```

In this format, *ns* represents the namespace of the tag. The *tagName* is the name associated with the tag's functionality. *attr1* and *attr2* are two attributes associated with the tag. Below is an example of a simple tag.

```
<c:out name="count" value="3"></c:out>
```

In this example, the namespace is *c*. The tag's name is *out* and it has two attributes, *name* and *value* whose values are count and 3 respectively. Since the tag is a stand alone tag, the closing tag can be incorporated into the body of the start tag.

```
<c:out name="count" value="3"/>
```


7.2 JSTL

The first version of JSP made it difficult to format and display data easily within a web page. The JSTL, Java Standard Template Library, was created to make the process of formatting output easier. In addition JSTL added libraries that would simplify working with databases and XML documents. It was the first technology to deploy the EL, expression language. In JSP 2.0 EL was added to the JSP standard. The JSTL was soon to follow. JSTL was renamed JSP Standard Tag Library and is now part of the JSP standard.

The JSTL is composed of 4 sets of libraries that perform different tasks. They are Core, Format, XML, and SQL.

The Core library is used to simplify the output of data within a web page. It provides mechanisms for flow control, outputting data, and URL management.

The Format library is used for internationalization. It makes it easy to output data based upon locality. In addition, it provides an easy mechanism for formatting dates and numbers.

The XML library provides methods for reading and manipulating XML documents. The SQL library is used to query the database to update the database and pull results.

The Core and Format libraries are the ones that will be the focus of the class. They are the ones used to simplify the output of data to the end user. The other libraries have uses but they are beyond the scope of this course.

7.2.1 Including JSTL Libraries

The first step in using JSTL within a JSP page is to use the taglib directive to include the library within the page. Below are the taglib statements for each of the four JSTLs.

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

The taglib directive takes two arguments, the uri and the prefix. The URI for a custom taglib is defined within the web.xml file. It is used to map back to a particular library. Since the JSTL is part of the core language, the URI shown above will point to the correct library. Table 7.1 shows the URI to use for each JSTL.

URI	Library
http://java.sun.com/jsp/jstl/core	Core Library
http://java.sun.com/jsp/jstl/fmt	Format Library
http://java.sun.com/jsp/jstl/xml	XML Library
http://java.sun.com/jsp/jstl/sql	SQL Library

Table 7.1: URIs For JSTL

The prefix's shown above can be set to whatever value desired, but it is standard practice to define the core library as **c**, format library as **fmt**, etc.

7.2.2 Core Tags

The core library provides a set of mechanisms for displaying data and controlling the flow of the user interface. They introduce the ability to provide conditionals and loops into the web application within a tag set. The following sections provides a tour of the most commonly used core tags.

Set Tag

The set tag is used to define a variable and associate a value with it. The tag extends the concept of a variable by allowing it to be associated with any of the web scopes: page, request, session, or application. Below is a simple example of setting a variable within a JSP page.

```
<c:set var="counter" value="10" scope="request"/>
```

The tag has three attributes available. The var attribute is the name of the variable to create. The value attribute associates the value with the variable name. It can be a constant as shown in the example above or it can be the result of an EL evaluation. The scope determines what object will hold the variable. In this case the HttpServletReqeust object. That does it by calling the *setAttribute(...)* method on the request object.

The page scope associates the variable with the current page object. request scope associates with the HTTP request and is the same as calling *request.setAttribute(...)*. The session scope associates the variable with the HttpSession object. It is the same as calling *session.setAttribute()*. The application scope associates the variable with the application's context. The most common values for the scope is request and session. The page scope can be used for local variables that are not passed to include tags.

The nice thing about the set tag is the variable can be accessed by a JSP EL statement. Below is an example of displaying the content from a set.

```
<c:set var="counter" value="10" scope="page"/>
<p>
    The value of the counter is ${counter}.
</p>
```

In this example the variable is access and displayed on the web page using standard EL constructs. It will generate the following HTML.

```
<p>
    The value of the counter is 10.
</p>
```

Set can be used multiple times for a single variable. A new set will replace the current value associated with the variable to the new one. It works in a similar fashion to the assignment operator in Java. Below is an example of using EL to set a variable to a new value.

```
<c:set var="counter" value="10" scope="page"/>
<c:set var="counter" value="${counter + 2}" scope="page"/>
<p>
   The value of the counter is ${counter}.
</p>
```


forEach Tag

The **forEach** tag provides a simple tag based looping mechanism for a JSP page. It works by repeating the content between the open and closing tags a defined set of iterations. The iterations can be defined as a set number of iterations or it can be told to loop through all the elements of any class that implements the Collection interface or a Java array.

The ability of the *forEach* tag to loop through the elements of any class that implements the Collection interface makes it ideal for looping through Lists, Sets, and Enumerations. It makes the tag invaluable for displaying a list of data within a JSP. It removes the need for JSP scriptlets to manage looping which can be unwieldy.

Table 7.2 shows the list of attributes that are used by the forEach tag.

Attribute	Description
var	Name associated with the object that is currently the focus of the iteration
items	The Collection or array of items to be iterated over
begin	Alternate to items that defines the start number for a known loop size
end	The end number for a known loop size
step	The size of increment for a known loop size

Table 7.2: URIs For JSTL

When dealing with the forEach tag there are really two different looping mechanisms. One is for loops with a known size and one for iterating over the contents of an entire collection. The known loop uses the *begin* and *end* attributes to set up the boundaries of the loop with the step function capability provided by the *step* attribute. In the case of Collections or arrays, the *items* attribute is used to define what list is going to be traversed. In both cases the *var* attribute assigns a variable to hold the value from the current loop iteration.

The easier to understand is the *forEach* that uses a known loop size. Below is an example of a simple looping mechanism that iterates between 2 and 7.

```
<c:forEach var="counter" begin="2" end="7">
    The counter is at ${counter}<br/>
</c:forEach>
```


This example would result in the following HTML.

```
The counter is at 2<br/>
The counter is at 3<br/>
The counter is at 4<br/>
The counter is at 5<br/>
The counter is at 6<br/>
The counter is at 7<br/>
```

Compare this example to the same solution using JSP scriptlets.

```
<%
    for ( int x = 2; x <= 7; x++ ) {
%>
    The counter is at <%= x %><br/>
<%
    }
%>
```

Which one is easier to read? Which one can be managed without knowing anything about Java? The answer is obvious. The real purpose for the JSTL is to make it easier to manage the user interface using JSPs.

The mechanism for looping over collections of data is a bit more complicated to explain but easy to use. The items attribute is used to define the collection using EL. This means that the collection must be visible from EL. It must be an attribute of either the page, request, session, or application objects.

The best way to understand the process is to go through an example.

```
<%
    java.util.ArrayList list = new java.util.ArrayList();
    list.add("Test 1");
    list.add("Test 2");
    list.add("Test 3");
    list.add("Test Over");
    request.setAttribute("myList", list);
%>

<c:forEach var="entry" items="${myList}">
    ${entry}<br/>
</c:forEach>
```

The first step is the construction of the list. It needs to be accessible from EL. This is accomplished by adding it as an attribute to the request object with the name *myList*. The name can be used by the EL to reference the collection.

When the code gets to the *forEach* tag, it begins a loop with the first element in the list. Each item in the collection is traversed. The value associated to the *entry* variable will be different for each loop. It will hold the current iteration of the loop. In this instance, the following HTML would be generated.

```
Test 1<br/>
Test 2<br/>
Test 3<br/>
Test Over<br/>
```

The real power of using the forEach loop with EL is in the ability to loop through and display data from a list of objects. The final example of the section uses the following Employee object.

```
public class Employee {
    private int id;
    private String firstName;
    private String lastName;

    public Employee(int id, String firstName, String lastName) {
        this.id = id;
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public void setId(int id) {
        this.id = id;
    }
    public void setFirstName(String name) {
        this.firstName = name;
    }
    public void setLastName(String name) {
        this.lastName = name;
    }

    public int getId() {
        return id;
    }
    public String getFirstName() {
        return firstName;
    }
    public String getLastName() {
        return lastName;
    }
}
```


This time the example will create a list of objects and iterate over the objects using the *forEach* tag. Each element will be accessed using EL to pull data directly from the bean / data object.

```
<%
    Employee one = new Employee(1, "Jane", "Doe");
    Employee two = new Employee(13, "Beth", "Smith");
    Employee three = new Employee(7, "Sarah", "Johnson");
    java.util.LinkedList list = new java.util.LinkedList();
    list.add(one);
    list.add(two);
    list.add(three);
    request.setAttribute("employees", list);
%>

<table border="1" cellspacing="0" cellpadding="4">
    <tr>
        <td>Employee ID</td>
        <td>First Name</td>
        <td>Last Name</td>
    </tr>
    <c:forEach var="employee" items="${employees}">
        <tr>
            <td>${employee.id}</td>
            <td>${employee.firstName}</td>
            <td>${employee.lastName}</td>
        </tr>
    </c:forEach>
</table>
```


In this case each iteration of the loop grabs individual objects from the list and uses the "." in EL to access the attributes of each object. The resulting HTML is shown below.

```
<table border="1" cellspacing="0" cellpadding="4">
    <tr>
        <td>Employee ID</td>
        <td>First Name</td>
        <td>Last Name</td>
    </tr>
    <tr>
        <td>1</td>
        <td>Jane</td>
        <td>Doe</td>
    </tr>
    <tr>
        <td>13</td>
        <td>Beth</td>
        <td>Smith</td>
    </tr>
    <tr>
        <td>7</td>
        <td>Sarah</td>
        <td>Johnson</td>
    </tr>
</table>
```


if Tag

The **if** tag can be used for simple conditional tests. It has one attribute, *test*, which is used to perform the expression test using EL. The results of the expression must be "true" or "false" otherwise an error will occur.

Below is an example using the *if* tag.

```
<c:forEach var="counter" begin="1" end="10">
    ${counter}</br>
    <c:if test="${counter == 5}">
        We are half way!</br>
    </c:if>
</c:forEach>
```

It would result in the following HTML

```
1</br>
2</br>
3</br>
4</br>
5</br>
We are half way!</br>
6</br>
7</br>
8</br>
9</br>
10</br>
```

The only drawback to using the *if* tag is the lack of an else or else if tag. That means the *if* tag can only be used in a situation where a single comparison is needed. If the more common multi-conditional implementation is needed then the *choose* tag is used.

choose Tag

The **choose** tag is the conditional tag for developer wanting to do multiple choice conditionals. The main *choose* tag is a wrapper tag for the multiple conditionals and has no attributes. The two inner tags that are used to perform each of the conditionals are **when** and **otherwise** tags. The *when* tag has a *test* attribute for providing an expression test. It is possible to use multiple *when* tags within the *choose*. The code will use the first conditional that evaluates to true. The *otherwise* tag is the equivalent of the else in an if/else command. It will be selected for execution if none of the other *choose* tags evaluate to true.

Below is a simple example to provide a better understanding of how to use the *choose* tag.

```
<%
    Person person = new Person();
    person.setName("Beth Smith");
    person.setAge(23);
    request.setAttribute("beth", person);
%>

<c:choose>
    <c:when test="${beth.age > 20}">
        Come on in ${beth.name}, you are old enough to enter the club.
    </c:when>
    <c:when test="${beth.age < 1}">
        That is not a real age.
    </c:when>
    <c:otherwise>
        Sorry ${beth.name}, I can't allow you into the club. You are
        not old enough.
    </c:otherwise>
</c:choose>
```

The *choose* tag denotes that a conditional decision is being made. Inside the tag, the developer must have at least one *when* tag. In this example, two *when* tags are used to provide different conditionals. The first test to make sure the user is 21 or older. The second checks to see if the age is zero or a negative value. If neither tag's conditions are met then the value from the default *otherwise* tag will be chosen.

url Tag

The **url** tag is a handy tag that can be used to properly encode URLs. It can be used as a stand along tag to create a URL or it can be used in conjunction with the **param** tag to build more complex URLs.

The basic *url* tag has two attributes associated with it. The *var* attribute is used to assign the URL to a variable that can be displayed at a later time. The second is the *value* attribute. It allows the user to define the URL to create . It can be either a full URL, a relative URL, or one based upon the application root. A full URL is one that contains the "http://" moniker at the beginning. A root based URL is one that begins with a leading "/" character. Relative paths are ones that do no begin with either a "/" or an "http://" and are based upon the path of the currently accessed URL.

The following example shows how the three different types are used. For the purpose of the example, the URL of the current page is *http://nplus1.net/foobie/employee/benefits.html*. In this instance the application root is foobie. The first example uses the full URL type.

```
<a href="<c:url value="http://cnn.com"/>">Test URL 1</a>
```

It would result in the following HTML.

```
<a href="http://cnn.com">Test URL 1</a>
```

The full URL is never used within a URL tag because it is an unnecessary step. The next example uses the application root as a starting position for creating the URL.

```
<a href="<c:url value="/work/timesheet.html"/>">Test URL 1</a>
```

It would result in the following HTML.

```
<a href="http://nplus1.net/foobie/work/timesheet.html">Test URL 1</a>
```

In this case the root application is foobie therefore the URL in the example would be expanded from the application root. The final example is relative paths.

```
<a href="<c:url value="healthplan.html"/>">Test URL 1</a>
```

Since it is a relative path, the value is based upon the path of the previous URL. It would generate the following HTML.

```
<a href="http://nplus1.net/foobie/employee/healthplan.html">Test URL 1</a>
```


Notice that the path used "http://nplus1.net/foobie/employee/" is the same path for the current URL.

The *url* tag does not have to be used solely for simple URL generation. It can be used for URLs that include request parameters. For example:

```
http://nplus1.net/foobie/work/timesheet.html?id=42&day=13&month=January
```

In this case three parameters are being passed along with the URL. The first is the id whose value is 42. The second and third are used to pass the value for the day and month respectively.

It is possible to build this type of complex URL using a *url* tag. But it can only be accomplished with the help of the *param* tag. The *param* tag has two attributes for this function, *name* and *value*. The name/value pair represents a parameter formed at the end of the URL. The previous URL could be constructed with the *url* tag is shown below.

```
<c:url value="/work/timesheet.html" var="timeUrl">
    <c:param name="id" value="42"/>
    <c:param name="day" value="13"/>
    <c:param name="month" value="January"/>
</c:url>
```

In this case it is associated with the variable *timeUrl*. The value can be placed within a HREF tag using EL.

```
<a href="${timeUrl}">Edit timesheet for Jan 13th</a>
```

The following URL would result.

```
<a href="http://nplus1.net/foobie/work/timesheet.html?id=42&
    day=13&month=January">Edit timesheet for Jan 13th</a>
```

This concept can be extended by using EL in the value portion of the *param* tag. An action which makes it easy to generate dynamic URLs. Below is an example using the *forEach* loop to generate multiple dynamic URLs.

```
<%
    ArrayList<Employee> employees = new ArrayList<Employee>();
    employees.add(new Employee(101, "John", "Doe"));
    employees.add(new Employee(102, "Jane", "Doe"));
    employees.add(new Employee(103, "Beth", "Smith"));
    employees.add(new Employee(104, "Sarah", "Johnson"));

    request.setAttribute("employeeList", employees);
```



```
%>
<table border="0" cellspacing="0" cellpadding="4">
    <tr>
        <td>Last Name</td>
        <td>First Name</td>
        <td> </td>
    </tr>
    <c:forEach var="employee" items="${employeeList}">
        <c:url value="editName.html" var="editUrl">
            <c:param name="employeeId" value="${employee.id}"/>
        </c:url>
        <tr>
            <td>${employee.lastName}</td>
            <td>${employee.firstName}</td>
            <td> <a href="$editUrl">Edit Name</a> </td>
        </tr>
    </c:forEach>
</table>
```

In this example, four Employee objects are added to an ArrayList named *employees*. It in turn is added to the request as an attribute named *employeeList*. The forEach tag within the page loops over each element from the ArrayList and assigns it to the variable *employee*. Each *employee's* id is associated with the *editUrl* variable and then used within a HREF tag to dynamically create a URL based upon the *employee's* id. The example above would generate the following HTML.

```
<table border="0" cellspacing="0" cellpadding="4">
    <tr>
        <td>Last Name</td>
        <td>First Name</td>
        <td> </td>
    </tr>
    <tr>
        <td>Doe</td>
        <td>John</td>
        <td>
            <a href="http://nplus1.net/foobie/editName.html?employeeId=101">
            Edit Name</a>
        </td>
    </tr>
    <tr>
        <td>Doe</td>
        <td>Jane</td>
        <td>
            <a href="http://nplus1.net/foobie/editName.html?employeeId=102">
            Edit Name</a>
        </td>
    </tr>
    <tr>
        <td>Smith</td>
        <td>Beth</td>
        <td>
            <a href="http://nplus1.net/foobie/editName.html?employeeId=103">
            Edit Name</a>
        </td>
    </tr>
    <tr>
        <td>Johnson</td>
        <td>Sarah</td>
        <td>
            <a href="http://nplus1.net/foobie/editName.html?employeeId=104">
            Edit Name</a>
        </td>
    </tr>
</table>
```


7.2.3 Format Tags

The Format library was meant to handle internationalization issues that arise when developing web applications. While this course doesn't have time to delve into the concepts behind internationalization, it can look at two useful tags for displaying numeric and date data.

formatNumber Tag

The format number tag can be a relief for people who have had to display formatted numbers on a web page. The biggest pain in working with Java numbers is the proper display of floating point numbers. By default Java will print out all of the decimal places it has for a number. That could be zero to many. The **formatNumber** tag has two attributes to deal with this problem. The *maxFractionDigits* and the *minFractionDigits*.

What if a developer needed to display 5 floating point numbers on the page and only wanted to show them to three decimal places? How could they use the *formatNumber* tag to accomplish this task. The example below is one possible solution.

```
<%
    double [] numbers = { 4.2521, 4.2567, 4.0, 4.2, 4.13 };
    request.setAttribute("results", numbers);
%>
<table border="0" cellspacing="0" cellpadding="4">
    <tr>
        <td>Year</td>
        <td>Results</td>
        <c:set var="year" value="2003"/>
    </tr>
    <c:forEach var="result" items="${results}">
        <tr>
            <td>${year}</td>
            <td align="right">
                <fmt:formatNumber maxFractionDigits="3" value="${result}"/>
            </td>
        </tr>
        <c:set var="year" value="${year + 1}"/>
    </c:forEach>
</table>
```


This would result in the following HTML

```
<table border="0" cellspacing="0" cellpadding="4">
    <tr>
        <td>Year</td>
        <td>Results</td>
    </tr>
    <tr>
        <td>2003</td>
        <td align="right">4.252</td>
    </tr>
    <tr>
        <td>2004</td>
        <td align="right">4.257</td>
    </tr>
    <tr>
        <td>2005</td>
        <td align="right">4.0</td>
    </tr>
    <tr>
        <td>2006</td>
        <td align="right">4.2</td>
    </tr>
    <tr>
        <td>2007</td>
        <td align="right">4.13</td>
    </tr>
</table>
```

The only problem with the output is that the numbers will not line up on the page by decimal place. This problem can be resolved by changing the *formatNumber* tag to add a value for the *minFractionDigits*. It will ensure that the defined number of decimal places will be displayed.

```
<fmt:formatNumber minFractionDigits="3" maxFractionDigits="3"
    value="${result}"/>
```


It would generate the revised HTML where all the numbers have the same number of decimal places and would line up neatly on the screen.

```
<table border="0" cellspacing="0" cellpadding="4">
    <tr>
        <td>Year</td>
        <td>Results</td>
    </tr>
    <tr>
        <td>2003</td>
        <td align="right">4.252</td>
    </tr>
    <tr>
        <td>2004</td>
        <td align="right">4.257</td>
    </tr>
    <tr>
        <td>2005</td>
        <td align="right">4.000</td>
    </tr>
    <tr>
        <td>2006</td>
        <td align="right">4.200</td>
    </tr>
    <tr>
        <td>2007</td>
        <td align="right">4.130</td>
    </tr>
</table>
```

The attributes *maxIntegerDigits* and *minIntegerDigits* work in the same fashion, but they deal with the number of digits to the left of the decimal place or whole numbers. Another attribute that is useful for dealing with large whole number is the *groupingsUsed* attribute. This will break up large numbers into 3 digit groupings. In the US those groups are separated by a ,. For example the number 10000000000.12 can be displayed as 10,000,000,000.12.

```
<fmt:formatNumber value="10000000000.12" groupingsUsed="true"/>
```

Here is where the internationalization part comes into play. If the locale of the server is somewhere in Europe the previous example would be shown using European standards and the value would be shown as 10.000.000.000,12.

Another useful feature of the *formatNumber* tag is the ability to assign a type value to a number. The type attribute allows the developer to set the type. By default all numbers are treated as the type "number". Two other types are available the "percent" and "currency" types. Both of which use the locale for determining what to display. Below is an example of using the currency type.

```
<fmt:formatNumber type="currency" value="12342.12"/>
```

It would display the value as $12,342.12.

formatDate Tag

The **formatDate** tag can be used to format both dates and times. The tag supports five different attributes. The *value* attribute holds the date or time to be formatted. The *type* attribute is used to determine what time representation to used. The valid entries are "date", "time" or "both". Date and time are used to show either the date or the time respectively while both will display the time and the date in a combined format. The next attribute to define is the *dateStyle* and the *timeStyle*. These are predefined output formats for each date/time type. Valid values include "short", "medium", "long" or "full". The output for each type is the same as those defined by the *java.text.DateFormat* object within the Java API. Table 7.3 shows example formating for each type.

Type	Date Example	Time Example
short	1.13.72	5:08am
medium	Jan 13, 1972	5:08am
long	January 13, 1972	5:08:12am
full	Thursday, January 13, 1972 AD	5:08:12am EDT

Table 7.3: Date / Time Types

Below are three examples of using the *formatDate* followed by the HTML output that would be produced.

```
<%
    Date date = new Date();
    request.setAttribute("rightNow", date);
%>
<fmt:formatDate type="date" value="${rightNow}" dateStyle="medium"/><br/>
<fmt:formatDate type="time" value="${rightNow}" timeStyle="long"/><br/>
<fmt:formatDate type="both" value="${rightNow}" dateStyle="short"
      timeStyle="short""/>
```

The HTML:

```
Jan 23, 2009<br/>
2:18:03pm<br/>
1.23.2009 2:18pm
```


While these generic types are handy in many cases, it doesn't come close to handling all of the needs for a developer. Instead of using the *dateStyle* and *timeStyle* the developer can create a custom formatting pattern. The allowed patterns are defined in the *java.text.SimpleDateFormat* object.

These symbols can be combined in various combinations to create a wide variety of date and time types for display. Table 7.4 provides a list of common formatting options.

Character	Description
y	year
M	month in year
d	day in month
E	day in week (ex Monday, Tue)
H	Hour in day (24 hour)
h	hour in AM / PM
m	minute in hour
s	second in minute
a	AM / PM
z	time zone (ex EDT, CST)

Table 7.4: Date Pattern Symbols

Below are three examples of using the patterns from Table 7.4 to create custom date output.

```
<fmt:formatDate value="${rightNow}" patten="M/d/yyyy"/><br/>
<fmt:formatDate value="${rightNow}" patten="HH:mm:ss"/><br/>
<fmt:formatDate value="${rightNow}" patten="MMM d, yyyy hh:mm:ss a"/><br/>
```

The results of these patterns are shown below.

```
1/25/2009<br/>
14:25:42<br/>
Jan 25, 2009 2:25:42 pm<br/>
```


7.3 Custom Tag Libraries

The JSTL is an excellent tool for assisting the developer in the creation of the visual elements inside of a JSP page. Unfortunately, it doesn't easily meet all of the needs for a developer. In those instances, a more custom solution to the user interface is appropriate. This is where the creation of custom tag libraries prove useful.

Tags can be lumped into two categories when discussing the creation of a custom tag library. There are simple tags that do not contain a body. They are self contained tags that perform a singular operation. Below is an example of a body-less tag.

```
Multiplying 5 and 10 leads to a result of <tag:multiply x="5" y="10"/><br/>
```

The more complicated group of tags are those that have a body. Tags that allow data to be encompassed within the start and end tags. Below is an example of a customized tag that has content between the tags.

```
<tag:loop start="2" iterations="5">
    ${loopValue}<br/>
</tag:loop>
```

The section that follows goes through the basic steps necessary for the construction of a simple body-less tag file and then expands upon it to encompass more complicated tags which contain content.

7.3.1 Simple Tag Files

The first step in creating a custom tag is to the generation of the business logic for the tag. It is accomplished by creating a *.tag* file which contains all of the coding associated with the tag. A *.tag* file is nothing more than a JSP fragment. Since it is JSP based, it has all of the properties available to a JSP page. This includes the implicit objects. The only thing that differentiates it from a JSP page is that it must be marked with a JSP tag directive at the top of the page.

Once the tag is coded, the tag stub must be made available to the web application. This is done by sticking the stub into the war files's WEB-INF/tag directory. At deployment time, the application server will sift through the war file and find all of the custom tags. All that remains to do is to invoke the custom tag from within other JSP pages.

Directives

There are two directives that are specifically assigned to the creation of tags. The **tag** directive informs the container that the fragment is a custom tag. The *body-content* attribute of the directive is used to tell the container which type of tag it is. A value of "empty" tells the container that the tag is body-less. If the tag is a complex tag with a body then the value of "scriptless" is used.

```
<%@ tag body-content="empty" %>
<%@ tag body-content="scriptless" %>
```

It is important to note that the *tag* directive must be the first line within the tag fragment.

The second directive reserved for tags is the *attribute* directive. It defines the attributes that can be associated with a tag. In the example previously, the multiply tag had two attributes associated with it. They were the values of *x* and *y*. For these attributes to be available to the tag they must be defined within the tag fragment using the *attribute* directive.

Attributes must define three properties. First and foremost they must have a name to associate to the tag. This is the *name* of the tag's attribute. The attribute must also have a data *type* associated with it. This value can be either a String or any of the primitive wrapper classes. Lastly, a required value is used to determine if the tag's attribute is a *required* attribute or not. This value can be either "true" or "false". Below are some example of defining tag attributes.

```
<%@ attribute name="firstName" required="true" type="java.lang.String" %>
<%@ attribute name="age" required="true" type="java.lang.Integer" %>
<%@ attribute name="weight" required="false" type="java.lang.Double" %>
```

The initial attribute, *firstName*, is a required attribute whose value will be a String. The second attribute, *age*, is also required and the value must be an integer. The final attribute, *weight*, is not required but if it is it must be a double value.

The fragment below brings the two directives together into a cohesive example.

```
<%@ tag body-content="empty" %>
<%@ attribute name="firstName" required="true" type="java.lang.String" %>
<%@ attribute name="age" required="true" type="java.lang.Integer" %>
<%@ attribute name="weight" required="false" type="java.lang.Double" %>
```


If this fragment is placed within a *person.tag* file, then the following examples would be valid invocations of the tag file. Notice how the attributes defined above translate into the invocation of the tag.

```
<tag:person firstName="Beth" age="23"/>
<tag:person firstName="John" age="24" weight="172.4"/>
```

The next set of tag invocations will fail. Why?

```
<tag:person firstName="Beth" weight="101.4"/>
<tag:person age="24" weight="172.4"/>
<tag:person firstName="Beth"/>
```

The first and third examples would fail because the required field age was not supplied. The second would fail because the required field firstName was not defined as an attribute of the tag.

One final note before moving on to tag creation is the additional attribute *rtexprvalue*. *rtexprvalue* determines whether or not a attribute can include a run time expression. By default attributes allow *rtexprvalue*. That means that a JSP expression scriptlet can be used for the assignment of an attribute. Below is an example of using a JSP expression scriptlet when defining an attribute for a custom tag.

```
<tag:person firstName="<%= session.getAttribute("name") %>"/>
```


Creating The Tag's Business Logic

Once the tag is defined, it is time to embed the tag logic. A tag can contain almost any element which can be found in a normal JSP page. It can use the *taglib* directive, *page* directive, and any of the implicit variables defined by JSP.

Below is a solution of the multiply tag used in previous examples.

```
<%@ tag body-content="empty" %>
<%@ attribute name="x" required="true" type="java.lang.Integer" %>
<%@ attribute name="y" required="true" type="java.lang.Integer" %>
${x * y}
```

In this case the tag is being defined along with two attributes, *x* and *y*. The values from the attributes are multiplied in the EL statement on the last line to show the results of the operation.

While the simple example is effective, it is not very illustrative of what can be done within a tag. Let's embellish the example a bit and add a third variable which is optional. In addition, the example will get the user information from the session object and display it along with the results.

```
<%@ tag body-content="empty" %>
<%@ attribute name="x" required="true" type="java.lang.Integer" %>
<%@ attribute name="y" required="true" type="java.lang.Integer" %>
<%@ attribute name="z" required="false" type="java.lang.Integer" %>

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

<c:set var="results" value="${x * y}"/>
<c:if test="${!empty z}">
    <c:set var="results" value="${results * z}"/>
</c:if>

<%
    String name = (String)session.getAttribute("userName");
    out.print(name + ", the answer to the problem ");
%>

${x} * ${y}
<c:if test="${!empty z}">
    * ${z}
</c:if>
is ${results}
```

In this example the results are first calculated for *x* times *y*. A test is next done to see if the *z* attribute was supplied. If it is then the multiplication includes the *z* value to calculate the results. Once the result is tabulated, the code retrieves the name from the session object and uses the *out* variable

to display the user's name. This was done to provide an example of using scriptlets and implicit variables within the tag. In the real world the JSP scriptlet would have been replaced with an EL expression shown below.

```
${userName}, the answer to the problem
```

Once the *userName* is outputted, a test is done to see if the *z* attribute was supplied. If *z* is defined then the display of the formula and results will include the value of *z*.

Adding Tag Files To WAR With Ant

Now that the simple tag fragment has been coded, how does a developer add the tag into the WAR file generated at compile time? The task can be accomplished through a simple modification to the ANT build.xml file.

The .tag file needs to be added to the WEB-INF/tags directory within the war file. The *zipfileset* tag that was used to put all of the JSP, HTML, and other resources into the root of the war file can be used to accomplish the task of putting the tag fragments within a tag directory within the WEB-INF.

Below is a modified version the ANT file to include tags.

```
<war destfile="${dist}/course.war" webxml="${conf.web}/web.xml">
    <classes dir="${build}"/>
    <lib dir="${web.lib}"/>
    <zipfileset dir="${web.jsp}"/>
    <zipfileset dir="${web.tags}" prefix="WEB-INF/tags/"/>
</war>
```

The line below is what was added to the build.xml file for the inclusion of tags.

```
<zipfileset dir="${web.tags}" prefix="WEB-INF/tags/"/>
```

The *zipfileset* task takes all of the tags that reside within the *web.tags* directory and zip them into a single directory. The *prefix* attribute is defines what directory within the WAR file in which they will be placed. In this case they are included in the WEB-INF/tags directory.

Invoking Tag Files

The next step after adding the tag files to the WAR file is to import them into JSP pages. The steps for using custom tag libraries are the same for using JSTL. All a developer needs is the proper *taglib* directive and the custom tag will be available. Below is an example of importing a tag library.

```
<%@ taglib prefix="bs" tagdir="/WEB/INF/tags" %>
```

The directive does two things. It points to the WEB-INF/tags directory to look for the appropriate tag information. Secondly, It uses the XML prefix "bs" to provide a unique namespace for tag calls. It associates the namespace with the set of custom libraries. The name of the tag to invoke must match the name defined in the JSP tag fragment. Below is an example of invoking the multiply tag defined earlier.

```
<%@ taglib prefix="bs" tagdir="/WEB/INF/tags" %>

<html>
     <body>
          <h2>Multiply Numbers</h2>
          <p>
               5 * 4 = <bs:multiply x="5" y="4"/><br/>
               8 * 7 = <bs:multiply x="8" y="7"/>
          </p>
          <h2>Squares</h2>
          <p>
               <c:forEach var="num" begin="1" end="10">
                    Square of ${num} is
                    <bs:multiply x="${num}" y="${num}"/><br/>
               </c:forEach>
          </p>
     </body>
</html>
```


7.3.2 Tag With Body

It is possible to extend the basic functionality of the body-less tag to include working with the information placed between the opening and closing of the custom tag. Below is an example of an XML tag that contains a body.

```
<b>This is a test</b>
```

In this case the XML tag is a HTML bold tag. The body of the bold tag is the text in between the opening and closing of the bold tag. In this instance the data is "This is a test". In HTML, the browser would take the data between these tags and use a bold font for the display of the text.

When working with custom tags that contain a body, the body-content of the tab must be changed from "empty" to "scriptless". Otherwise the definition of the tag doesn't change.

The next step would be to add the internal logic for the tag. In this case the <jsp:doBody/> tag can be used to represent the content between the tags.

The simplest example of using the body is a simple looping mechanism. The sample tag is set to show the body of the tag "x" number of times. The number of iterations is defined by the required attribute iterations. In this instance the tag will add the text "Iteration x:" before the body.

```
<%@ tag body-content="scriptless"%>
<%@ attribute name="iterations" required="true" type="java.lang.Integer"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<c:forEach var="iter" begin="1" end="${iterations}">
    Iteration ${iter}: <jsp:doBody />
</c:forEach>
```

Once the tag is coded, it uses the same process as defined for the body-less tag for using the custom tag within a JSP page.

```
<%@ taglib prefix="n" tagdir="/WEB-INF/tags" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

<html>
    <body>
        <h2>Looping</h2>
        <n:loop iterations="5">
            My Loop!
        </n:loop>
    </body>
</html>
```

The example would generate the following HTML.


```
<html>
    <body>
        <h2>Looping</h2>
        Iteration 1: My Loop!
        Iteration 2: My Loop!
        Iteration 3: My Loop!
        Iteration 4: My Loop!
        Iteration 5: My Loop!
    </body>
</html>
```

Every time the <jsp:doBody/> tag is referenced the content within the tag is executed. Therefore it is possible to perform other tag functions within the custom tags. Let's change the previous example to execute some commands within the custom tag.

```
<%@ taglib prefix="n" tagdir="/WEB-INF/tags" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

<html>
    <body>
        <h2>Looping</h2>
        <c:set var="x" value="10"/>
        <n:loop iterations="5">
            My Loop value is ${x}!
            <c:set var="x" value="${x + 1}"/>
        </n:loop>
    </body>
</html>
```

It would generate the following HTML. In this case the value of *x* is displayed and incremented each time the <jsp:doBody/> tag is called by the parent loop tag.

```
<html>
    <body>
        <h2>Looping</h2>
        Iteration 1: My Loop value is 10!
        Iteration 2: My Loop value is 11!
        Iteration 3: My Loop value is 12!
        Iteration 4: My Loop value is 13!
        Iteration 5: My Loop value is 14!
    </body>
</html>
```


What about the possibility of naming conflicts. What if the looping tag used the variable *x* as well?

```
<%@ tag body-content="scriptless"%>
<%@ attribute name="iterations" required="true" type="java.lang.Integer"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<c:forEach var="x" begin="1" end="${iterations}">
    Iteration ${x}: <jsp:doBody />
</c:forEach>
```

In this case, we would get the same results as before. Why? Because the variable *x* within the tag is a local variable and therefore it doesn't affect the content of "*x*" within the body of the tag. There are no naming conflicts.

Tag Variables

One final aspect of custom tags is the ability to share internal tag variables outside of the tag. The variable directive is used to define tag level variables that are accessible to the outside world. There are three variable scopes that can be applied to the tag. Each one will be explained in a moment. For now look at the format for the variable directive.

```
<%@ variable name-given="mySpiffyVariable" scope="AT_BEGIN" %>
```

The name-given attribute defines the name of the variable while the scope defines where the variable is accessible. There three scopes available are AT_BEGIN, NESTED, and AT_END. The list below defines each of these scopes.

- AT_BEGIN – The variable is set before the body of the tag is evaluated. The variable used within the body will be set to that new value. The final value of the variable will be visible for the entire page.
- NESTED – The variable is set before the body of the tag and can be used within the body of the tag. The variable is reset once tag scope has passed.
- AT_END – The variable is not set before the body of the tag is evaluated. The variable maintains it's previous value during the body of the tag. Once the tag is processed the value set by the tag will be visible for the entire page.

Normally this is not a good idea because variable names inside the tag are exposed to the external calling page. Such functionality makes it very difficult to re-use the tag because one must have an intimate knowledge of the tag's internals before using it.

The only exception to the rule is when you pass in the name of the variable to create from the tag. This allows the JSP developer to define the name of the variable to be created and used by the page. Two new attributes to the variable directive are necessary to make this happen. The first is called *name-from-attribute*. It associates a variable to the value from one of the tag's attributes. In addition the *alias* attribute is used to give the variable a local value to use within the tag. It may sound confusing, but is easily understood from an example.

```
<%@ tag body-content="empty" %>
<%@ attribute name="squareVar" required="true" type="java.lang.String"
            rtexprvalue="false" %>
<%@ attribute name="x" required="true" type="java.lang.Integer" %>
<%@ variable name-from-attribute="squareVar"
            variable-class="java.lang.Integer"
            alias="myvar" scope="AT_END" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<c:set var="myvar" value="${x * x}"/>
```


In the example above the tag is given two attributes, *squareVar* and *x*. The *squareVar* attribute is going to be the name of the variable to create. For it to be used as a variable, the type must be a String and the *rtexprvalue* must be set to false. Next comes the variable definition. The *squareVar* attribute is assigned to a local variable with an alias of *myvar*.

The tag then defines *myvar* to be the square of the value of the attribute *x*.

The JSP page to use the new square tag would look like the following.

```
<%@ taglib prefix="n" tagdir="/WEB-INF/tags" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

<html>
    <body>
        <h2>Squaring Stuff</h2>
        <c:set var="myX" value="100"/>
        <n:square squareVar="var1" x="10"/>
        <n:square squareVar="var2" x="${myX}"/>
        The number 10 squared is ${var1}<br/>
        The number ${myX} squared is ${var2}<br/>
    </body>
</html>
```

Here the square tag is called and the value passed into *squareVar* is *var1* and the value is 10. The tag associates the value of 10 squared to the variable *var1* and makes it available to the rest of the page. Thus when `${var1}` is accessed later the value supplied is 100. The same principle holds for the second square tag, but this time it uses a predefined variable to supply the *x* value.

It would result in the following HTML

```
<%@ taglib prefix="n" tagdir="/WEB-INF/tags" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

<html>
    <body>
        <h2>Squaring Stuff</h2>
        The number 10 squared is 100<br/>
        The number 100 squared is 10000<br/>
    </body>
</html>
```


7.4 Lab Activity

Part 1.

The goal of the lab is to create an application that will list employee data. An employee has an employee number, first name, last name, city, state, and a salary. The employee list will be stored in the HttpSession object.

The first step is to create a JSP page to display the contents of the employee list. Use JSTL core tags and EL to populate the page. There should be no JSP scriptlets used in the creation of the page.

Provide a link at the bottom of the page to add a new employee.

Part 2.

Create a JSP page that only contains HTML widgets necessary to enter an employee's data. It should have a widget for id, first name, last name, city, state, salary, and a submit button. Do not define the form tag within this page.

Part 3.

Next create an add employee JSP. The page should inform the user to add a new employee and provide a form for the user to fill out the employee's information. Use the import function of the core library to import the HTML widgets created in part 2. The form tag should be defined in the add employee JSP.

Part 4.

Create a JSP that processes the addition of a new user to the list and redirects the user to the listing JSP from part 1.

Part 5.

Modify the JSP in part 1 to display the salary as a monetary value using the format tag. Provide a href around the employee number that links to an edit employee page. Use core url tag to create edit link tag.

Part 6.

Create an edit employee JSP that informs the user to edit data for an employee. Two things must be done to make this work. First edit the JSP created in part 2 to prepopulate the forms. Use the JSP in creating the edit task.

Secondly, use the core tags to identify the employee to edit. There should be no scriptlet tags within the edit employee JSP.

Part 7.

Create a JSP that processes the editing of the existing user and redirects the user to the listing JSP from part 1.

Part 8. (Optional)

Create a custom tag that generates a drop down box for displaying states. It should have a parameter that allows the developer to preset a particular state as selected. Populate the state box with 4 states.

Chapter 8

Java Web Programming Best Practices

Objectives

- Understand the MVC pattern
- Learn how to implement a Front Controller
- Learn how to implement an Application Controller

8.1 Best Practices

After learning the basic building blocks of Java web programming, it is now time to explore how to use the basic elements of web programming to develop high quality applications. One of the earliest design methodologies most considered good practice was the three tier development model. In this case the logic of an application is broken down into three separate tiers, presentation, business, and data.

The presentation tier is dedicated to the user interfaces experience. It is this tier were all of the coding logic is dedicated towards displaying information to or collecting data from the end user. This includes things like input validation, user authentication and authorization, and state management. The presentation tier is a combination of the attractive user interface interwoven with the security and stubs that talk to the application's business tier.

The business tier is where all of the business logic in an application resides. This is where a programmer would implement the business rules for the application. It is the layer where data gets manipulated as it flows between the user interface and the data store. It bridges the presentation tier and data tier. The data tier is where the data is stored and maintained. Usually it takes the form of a database.

In the case of web applications, Servlets and JSP represent the presentation tier of an application. They can be used in conjunction to provide an attractive user interface, basic security, and a bridge for talking with the business tier. The rest of the chapter outlines and develops a re-usable framework for building a presentation tier.

8.1.1 Goals For Web Framework

When creating a framework for web development, it is important to keep in mind a few goals for development. The first step is to look for known solutions to the problem to see if they can be applied to the web framework. The best thing to do is look through known design patterns to see if any are applicable to the development of a web framework.

Secondly, it is important that any solution properly implements the n-tier solution. It must separate the user interface from the business rules in the middle tier. It is important that presentation tier data objects are not used within the business tier. Doing so would tightly couple the presentation tier to the business tier.

The third thing to keep in mind is the necessity of avoiding unmanageable code blocks. The framework being developed should encourage smaller block of code. Classes with over two hundred lines of code can prove unwieldy and difficult to manage over a long time. Part of the framework solutions is to isolate responsibilities.

If each role is separated then two good things happen. The code is more manageable and more importantly it reduces dependency problems in development. Dependency problems occur when two pieces of code are tightly coupled. Thus a change in one code block automatically affects the other block. Tightly coupled code makes it very difficult to deal with functional requirement changes in an orderly way. Because changes to requirements make changes to tightly coupled systems propagate throughout the project. If the framework is properly decoupled then changes to project requirements are not as big of an issue. The inherit flexibility of the system makes it easier to handle the constant forces of change that inflict all projects.

8.1.2 MVC Pattern

The first patten that is applicable to the web framework is the model/view/controller pattern or MVC for short. It is a pattern that has been used for developing user interfaces for many years. The model/view/controller pattern gets its name from the three components that make up the pattern. The model represents the data to be displayed to the end user. The view represents the user interface that the client sees. The controller is used to take user interface interactions to update the model or to view the state of the model. In short the controller is the middle man between the data and the client.

It utilizes a known pattern to separate the user view from the business rules and data. So how do Servlets and JSP elements fit into this model? The view is represented by the HTML that is delivered to the client's browser. The controller is going to be the web framework being developed and the model is represented by the business tier of the application. In effect the web framework is the controller between the HTML delivered to end user and the business layer of the application.

The framework outlined in this courseware breaks the controller into two parts. The front controller for managing the view and the application controller that is used to manage interactions with the business tier.

8.2 Front Controller Servlet Pattern

HTML is a request based protocol, which means that all requests on a web application consist of a series of URLs. If each URL is treated as a unique access point into the system it would be very difficult to maintain consistency in the management of these requests. Think about security for a second. If each URL was unique then each URL would have to force some sort of authorization. That means the system could introduce multiple forms of authorization or worse could accidentally leave out authorization.

The idea behind the front controller servlet is to centralize where all incoming URLs are directed. Remembering back, serlvets could define wild cards for their servlet-pattern in the web.xml. At the time it was pointed out that the wild card allowed a servlet to be able to handle multiple URLs. This function can be exploited to create the front servlet. It would provide a central location for all incoming URL requests.

The pattern reduces the need for duplicate control logic and begins the process of separating the business tier from the view. The front controller takes incoming URL requests and passes the value to the action controller. The action controller is responsible for the request's action logic. The action returns the view to display. The front controller is responsible for managing the display and returning it to the end user's browser.

Below is a graphical illustration of this concept.

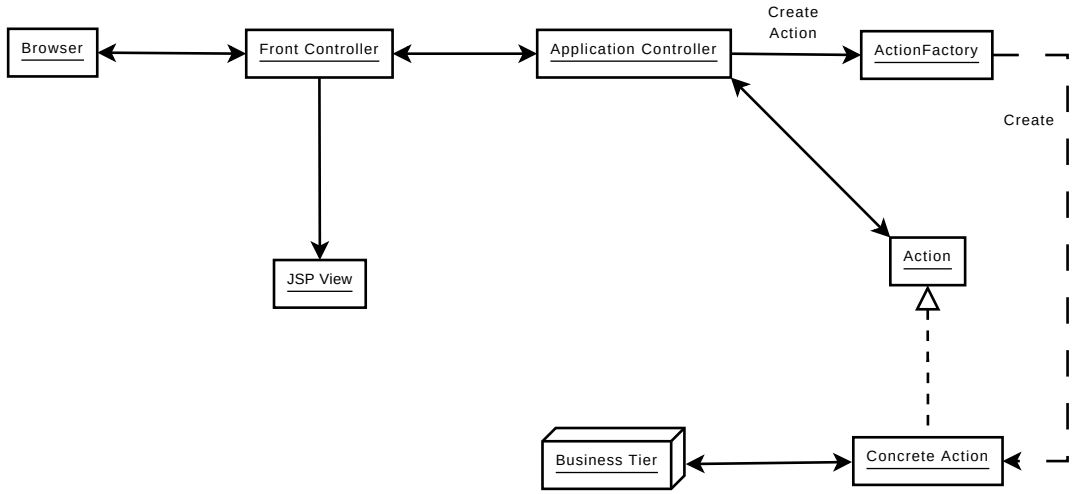

Figure 8.1: Basic MVC Framework

The HTML request comes into the front controller. Responsibility is handed to the application controller to handle the request. The application controller returns the view in the form of a URL to a JSP. The front controller is then responsible for forwarding control the JSP which creates the HTML returned to the client browser.

8.2.1 Creating The Front Controller Servlet

The creating of a front controller servlet is a two step process. Start by creating the the servlet class and then create an entry in the web.xml file to associate a URL with the class. The example below shows what needs to be added to the web.xml file to create a servlet that will handle multiple requests.

```
<servlet>
    <servlet-name>FrontController</servlet-name>
    <servlet-class>net.nplus1.web.FrontControllerServlet</servlet-class
</servlet>
<servlet-mapping>
    <servlet-name>FrontController</servlet-name>
    <url-pattern>/action/*</url-pattern>
</servlet-mapping>
```

When talking about the web.xml it is important to remember that there are two parts to defining a servlet. The servlet section which associates a name with the class file and the mapping which maps the class to a particular url pattern. In this case the url pattern utilizes the "*" wild card to indicate any URL to the web application that begins with a /action/. If the web applications root context is foobie then all of the following URLs would match the pattern and be redirected to the servlet.

```
http://foobie.nplus1.net/foobie/action/myurl.html
http://foobie.nplus1.net/foobie/action/play/test.htm
http://foobie.nplus1.net/foobie/action/foo/bar/mine.jsp
http://foobie.nplus1.net/foobie/action/test/fun
```

Now that the web.xml file is defined, the next step is to create the class file. In this case, the class **net.nplus1.web.FrontControllerServlet** needs to be defined.

```
package net.nplus1.web;

import javax.servlet.http.HttpServlet;

public class FrontControllerServlet extends HttpServlet {

}
```


Since this servlet is going to handle all requests for multiple URLs it has to implement both the *doGet(...)* and *doPost(...)* methods of the servlet.

```
package net.nplus1.web;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class FrontControllerServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse
                        response) throws ServletException, IOException {
    }

    public void doPost(HttpServletRequest request, HttpServletResponse
                        response) throws ServletException, IOException {
    }
}
```

Because it is going to treat all requests the same, it only make sense to create a method for processing the request and having both the doGet and doPost call that singular method.

```
package net.nplus1.web;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class FrontControllerServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse
                        response) throws ServletException, IOException {
        this.process(request, response);
    }
    public void doPost(HttpServletRequest request, HttpServletResponse
                    response) throws ServletException, IOException {
        this.process(request, response);
    }

    public void process(HttpServletRequest request,
                        HttpServletResponse response) {
       /*
        * front controller logic goes here
        */
    }
}
```


8.2.2 Handling Incoming Requests and Displaying The View

An incoming request is sent to the Front Controller servlet's *doGet(...)* or *doPost(...)* method and the call is forward to the *process(...)* method. Since any number of possible URLs are directed to this method, the first task is to find out what URL the user has requested. This can be retrieved from the path information.

```
public void process(HttpServletRequest request,
                    HttpServletResponse response) {
    String url = request.getPathInfo();
}
```

It is at this point that the Application Controller mentioned earlier comes into play. It takes the url and request information and processes the logic for the incoming request. It returns a view back to the Front Controller for display. Rather than creating a new Application Controller on every request this is one of those times where it is acceptable to have a private member variable within a servlet. The Application Controller doesn't hold any state so it won't run into any problems running in a threaded environment.

The Application Controller should be made a private member variable and initialized in the servlet's *init()* method.

```
private ApplicationController appController;

public void init() {
    this.appController = new ApplicationController();
}
```

Now the *url* and *request* can be passed to the Application Controller and the view can be received back from the controller.

```
String view = appController.processRequest(url, request);
```

What view is returned? It returns the path to a particular JSP page that holds the displayable HTML. Once the view is returned, it is the job of the Front Controller to pass control to the designated JSP. This can be accomplished using the RequestDispatcher and the *forward(...)* method.

```
RequestDispatcher dispatcher =
        this.getServletContext().getRequestDispatcher(view);
dispatcher.forward(request, response);
```

The JSP is responsible for the HTML view that is sent back the the browser. This makes sense because JSP pages are better than servlets for managing HTML information. At this point, the view

should only contain display logic and should contain no control logic. The control logic should have been handled by the Application Controller. Any data returned from the business tier is placed into the HttpServletRequest object by the Application Controller. Thus it is accessible from within the JSP view.

At this point it is time to think about error handling. What if the requested URL doesn't have an action associated with it? No view could be created and the value returned would be null. It is easy to check to see if the view is null. If it is then a 404 error can be returned to the user.

```
if ( view == null ) {
    response.sendError(HttpServletResponse.SC_NOT_FOUND);
}
else {
    RequestDispatcher dispatcher =
             this.getServletContext().getRequestDispatcher(view);
    dispatcher.forward(request, response);
}
```

Program 12 provides the code for the complete Front Controller.

Program Listing 12 Front Controller Servlet

```
package net.nplus1.web;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class FrontControllerServlet extends HttpServlet {

    private ApplicationController appController;

    public void init() {
        this.appController = new ApplicationController();
    }

    public void doGet(HttpServletRequest request, HttpServletResponse
                response) throws ServletException, IOException {
        this.process(request, response);
    }

    public void doPost(HttpServletRequest request, HttpServletResponse
                response) throws ServletException, IOException {
        this.process(request, response);
    }

    public void process(HttpServletRequest request,
                        HttpServletResponse response) {
        String url = request.getPathInfo();
        String view = appController.processRequest(url, request);

        if ( view == null ) {
            response.sendError(HttpServletResponse.SC_NOT_FOUND);
        }
        else {
            RequestDispatcher dispatcher =
                     this.getServletContext().getRequestDispatcher(view);
            dispatcher.forward(request, response);
        }
    }
}
```

The code accomplishes something very important. It decouples the control logic from the view logic in the web application. The control logic is handled by the Application Controller while the view logic is handled by the JSP page. All the while the Front Controller knows nothing about

how the Application Controller or JSP view operates. It is just a simple yet elegant mechanism for bridging the gap between the view and the controller.

8.3 Application Controller Pattern

Now it is time to shift focus to the Application Controller. The goal of the Application Controller is to centralize and modularize action and view management. The first step is to take the incoming request which is associated with a particular URL and redirect it to an isolated action. The purpose of which is to make each URL action discrete from other actions. Each action is executed and return a view for display. The view is returned to he front controller for dispatch.

By breaking down each URL request into separate actions, the system allows incremental development based on use case functionality. Which means that each use case will have its own discrete segment of the code that is decoupled from the rest of the system. A system which will make it easier to test individual functionality without fear of affecting other parts of the system. The diagram below provides a visual representation of the web framework including the Application Controller.

8.3.1 Actions

The command pattern can be used to provide the action functionality for the Application Controller. The command pattern allows the developer to create discrete functionality that is decoupled from the Application Controller framework. The first step in implementing the command pattern is to create an interface for handling request from the server. Program 13 provides the code for an action interface that accomplishes this task.

Program Listing 13 Action Interface

```
package net.nplus1.web;

import javax.servlet.http.HttpServletRequest;

public interface WebAction {
    public String process(HttpServletRequest request) throws Exception;
}
```

The first thing to notice is that the method call used is similar to the *doGet(...)* and *doPost(...)* methods of the servlet. It is done on purpose because the framework must allow generic processing of incoming requests. The only difference between the methods is the action interface does not need access to the HttpServletResponse object because the action is not responsible for the user interface output.

Since all of the actions implement the same interface, it is easy to build a generic mechanism for processing WebActions. The Application Controller essentially becomes that generic mechanism. But what should be done about the creation of all of the concrete action classes?

8.3.2 Action Factory

The factory pattern is a perfect fit for this task. It can be used to create the concrete implementation of the WebAction interface based upon the requesting URL. The factory is in charge of mapping the URL to the concrete implementation. Once the appropriate action is found it passes it back by the interface.

The easiest way to map URLs to instances is by using an if block of code within the controller. If URL A then create B. Once B is created it can be sent to the controller for processing;

Program Listing 14 Action Factory

```
package net.nplus1.web;

import net.nplus1.web.actions.TodoList;
import net.nplus1.web.actions.AddTask;
import net.nplus1.web.actions.ProcessTask;

public class WebActionFactory {
    public static WebAction createAction(String url) {
        WebAction action = null;

        if ( url.equals("/todo.html") {
            action = new TodoList();
        }
        else if ( url.equals("/add_task.html") {
            action = new AddTask();
        }
        else if ( url.equals("/process_task.html") {
            action = new ProcessTask();
        }

        return action;
    }
}
```

Notice two things about program listing 14. The concrete implementation of the WebAction interface is based upon the request URL. If the URL is not found then action returned is null. Secondly, notice that each implementation class is associated with its interface. This means that the Application Controller only has to deal with WebAction interface and not have to worry about knowing about each concrete implementation. Thus it is easy to generate a simple generic WebAction handler within the Application Controller.

Before looking at the implementation of the Application Controller it is worth noting at this point that the use of an if/else block for the factory is not a good idea for two reasons. First, it is necessary to modify and recompile the factory every time a new action is added. Secondly, the size of the if/else block would become unwieldy for larger applications. A better approach would be to use either a property or XML file for the lookup conversion and then uses reflection to create an instance

of the concrete class. That way all of the mappings are handled outside of the code base. The Struts framework works in a similar fashion and uses XML files for the mappings.

8.3.3 Application Controller Code

```
package net.nplus1.web;

import javax.servlet.http.HttpServletRequest;

public class ApplicationController {
    public String processRequest(String url, HttpServletRequest request) {
        String view = null;

        try {
            WebAction action = WebActionFactory.createAction(url);
            String view = action.process(request);
        }
        catch (Exception e) {
            view = "/error.jsp";
        }

        return view;
    }
}
```

The code is very simple and straight forward. It uses the *WebActionFactory* to create a concrete action and processes the action to get back a view. The action is responsible for returning the JSP to be used for viewing. If an error occurs in either the creating or processing of the action, then an error.jsp page is returned for viewing.

8.3.4 Lookup The View

The Application Controller's functionality can be extended beyond what is shown here. One of the problems with having the action return the JSP page is that it provides little flexibility for where the JSP pages reside. If they need to be moved then all of the actions would have to be modified to handle the change. Another option is to return a name for the view and then have a lookup mechanism for mapping the name to a particular JSP URL.

Program Listing 15 View Lookup

```
package net.nplus1.web;

public class LookupJSP {
    public static String lookup(String view) {

        String jsp = null;

        if ( view.equals("todo") {
            jsp = "/todo_list.jsp";
        }
        else if ( view.equals("addForm") {
            jsp = "/add_entry.jsp";
        }

        return jsp;
    }
}
```

Once again, the if block used in this example is not ideal. It would be better to use an outside resource such as a property or XML file to lookup these values. Regardless of the mechanism used, it is easy to update the controller to include the lookup step. In this case if no view is found then the returning view is null and the front controller will send back a page not found return code. Listing 16 shows the final Application Controller.

Program Listing 16 Application Controller

```
package net.nplus1.web;

import javax.servlet.http.HttpServletRequest;

public class ApplicationController {
    public String processRequest(String url, HttpServletRequest request) {

        String view = null;

        try {
            WebAction action = WebActionFactory.createAction(url);
            if ( action != null ) {
                String viewName = action.process(request);
                view = LookupJSP.lookup(viewName);
            }
        }
        catch (Exception e) {
            view = "jsp/error.jsp";
        }

        return view;
    }
}
```

8.4 Concrete Example Using The Framework

Once the framework is developed, it is time to go through the creation of a simple application using the framework. The section below uses the framework to generate a simple task list application. One that will display the task list and allow users to add new tasks.

For the application to work the developer must handle three different URL requests. One to display the task list, *todo.html*, one to show the add task form, *add_task.html*, and one to add the new task to the task list, *process_task.html*. The application will store the task list in the session variable.

8.4.1 Mapping Todo URL

These URLs must be mapped to actions by the ActionFactory. The first URL to map is the todo.html page.

```
package net.nplus1.web;

import net.nplus1.web.actions.TodoList;

public class WebActionFactory {
    public static WebAction createAction(String url) {
        WebAction action = null;

        if ( url.equals("/todo.html") {
            action = new TodoList();
        }

        return action;
    }
}
```


8.4.2 Creating TodoList Action

The next step is to create the *TodoList* WebAction to associate with the *todo.html* URL. The action begins by retrieving the task list from the session object. If the list doesn't exist then it needs to be instantiated. Lastly, the view must be returned by the action resulting in the following class.

```
package net.nplus1.web;

import java.util.List;
import java.util.LinkedList;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;

public class TodoList implements WebAction {
    public String process(HttpServletRequest request) throws Exception {
        HttpSession session = request.getSession();
        List<String> values = (List<String>)session.getAttribute("list");
        if ( values == null ) {
            values = LinkedList<String>();
            session.setAttribute("list",values);
        }
        return "todo";
    }
}
```


8.4.3 Mapping Todo View

The view *todo* needs to be mapped to a JSP view in the LookupJSP.

```
package net.nplus1.web;

public class LookupJSP {
    public static String lookup(String view) {
        String jsp = null;

        if ( view.equals("todo") {
            jsp = "/todo_list.jsp";
        }

        return jsp;
    }
}
```


8.4.4 Todo View

The JSP view, *todo_list.jsp* needs to be created to display the items in the task list. Since the task list is stored in the HttpSession under the name "list" it is available to the view using EL. Therefore it is simple to create a *forEach* loop to show all of the tasks. A link for adding a new task is also necessary.

```
<html>
    <body>
        <h2>Todo List</h2>
        <ul>
            <c:forEach var="task" items="${list}">
                <li>${task}</li>
            </c:forEach>
        </ul>
        <a href="add_task.html">Add New Task</a>
    </body>
</html>
```


8.4.5 Adding A Task

The next step is to create the add task action. First the URL mapping must be added to the action factory.

```
package net.nplus1.web;

import net.nplus1.web.actions.TodoList;
import net.nplus1.web.actions.AddTask;

public class WebActionFactory {
    public static WebAction createAction(String url) {
        WebAction action = null;

        if ( url.equals("/todo.html") {
            action = new TodoList();
        }
        else if ( url.equals("/add_task.html") {
            action = new AddTask();
        }

        return action;
    }
}
```


8.4.6 Creating AddTask

Now the *AddTask* action can be created. It only has two purposes. The first is to reset the error message mechanism and the second is to return the JSP view to display. The error message is being associated with the request so it can be accessible from the JSP using EL.

```
package net.nplus1.web;

import javax.servlet.http.HttpServletRequest;

public class AddTask implements WebAction {
    public String process(HttpServletRequest request) throws Exception {
        request.setAttribute("errorMessage", "");  // set value to empty
        return "addForm";
    }
}
```


8.4.7 View Lookup

The view, *addForm*, needs to be mapped to the JSP view *add_entry.jsp* in the LookupJSP.

```
package net.nplus1.web;

public class LookupJSP {
    public static String lookup(String view) {
        String jsp = null;

        if ( view.equals("todo") {
            jsp = "/todo_list.jsp";
        }
        else if ( view.equals("addForm") {
            jsp = "/add_entry.jsp";
        }

        return jsp;
    }
}
```


8.4.8 Add Task View

The JSP must be created to display an entry form allowing the end user to enter a new task. The form should allow the user to add a task into an entry box and submit it for processing. A secondary purpose is to provide a mechanism for displaying an error message if necessary. If add task JSP is called from the *AddTask* the variable *errorMessage* will be blank.

```
<html>
    <body>
        <form method="post" action="process_task.html">
            <h2>Add Todo Entry</h2>
            <p>${errorMessage}</p>
            <p>
                <input type="text" name="entry/><br/>
                <input type="submit" value="Add Task"/>
            </p>
        </form>
    </body>
</html>
```


8.4.9 Processing New Tasks

The final step in the creation of the simple application is an action for processing the data from the add task form. Before working on the processing of the task, the URL must be mapped in the *WebActionFactory*.

```
package net.nplus1.web;

import net.nplus1.web.actions.TodoList;
import net.nplus1.web.actions.AddTask;
import net.nplus1.web.actions.ProcessTask;

public class WebActionFactory {
    public static WebAction createAction(String url) {
        WebAction action = null;

        if ( url.equals("/todo.html") {
            action = new TodoList();
        }
        else if ( url.equals("/add_task.html") {
            action = new AddTask();
        }
        else if ( url.equals("/process_task.html") {
            action = new ProcessTask();
        }

        return action;
    }
}
```


8.4.10 Creating ProcessTask

Below is an implementation of the *ProcessTask* action. The action has two jobs. The first is to make sure the end user entered a value into the form. If not then the form needs to be re-displayed along with an error message telling the user to fill out the form. If the user entered data then the list must be retrieved from the HttpSession object and the new task appended to the list. The view should be directed to the JSP which displays the list of tasks.

```
package net.nplus1.web;

import javax.servlet.http.HttpServletRequest;

public class ProcessTask implements WebAction {
    public String process(HttpServletRequest request) throws Exception {
        String value = request.getParameter("entry");
        if ( value == null || value.trim().isEmpty() ) {
            request.setAttribute("errorMessage","You must enter a value");
            return "addForm";
        }

        HttpSession session = request.getSession();
        List values = (List)session.getAttribute("list");
        if ( values != null) {
            values.add(value);
        }
        return "todo";
    }
}
```

Two things are going on in this action. If the value of the form is *null* or is empty then the error message is set and the user is taken to the *addForm* view. This view will show the add task form again along with the newly defined error message. If the value is not null then the list is retrieved form the session. If the list exists then the value entered by the client is appended to the list. The view returned will be the todo view which was created previously. It will show the list of tasks. The new task will show up at the end of the list.

8.4.11 Final Thoughts

There are four steps required for processing a new URL request. The first is to map the URL to an action. Then the action can be created. The action determines the view that must be mapped back to a JSP page. The final step is to create the JSP view. If the mappings occur in a XML document, then all that is necessary is one class, one JSP and two quick modifications to XML documents.

Why go through the process of developing this particular framework? The first reason is that the framework separates the views from the control logic. There are no longer any need for scriptlets inside JSP pages. Everything can be handled by tags and EL. Secondly, it creates the possibility of re-usable views. In the example above the todo view is used by the TodoList and the ProcessTask tasks. Meanwhile the tasks are an independent bridge to the business layer of the application and have no direct tie-ins to the view.

Secondly, the framework creates a series of discrete actions. Each is independent of the other. That means modifying one does not affect the behavior of another. Therefore modifications to each action is guaranteed not to affect any other code. Making it easier to modify and test code generation. Not to mention the fact that request can be mapped directly to use cases in a development document. In the end the solution is more flexible allowing for better change management and code maintenance.

8.5 Lab Activity

Part 1.

The goal of the lab is to create a simple task list application using the framework developed in this chapter.

The task list should be stored in the HttpSession object. A task should consist of a priority, a due date, and text describing the task. Use a task object bean to store this data.

The first step is to create the web framework.

Part 2.

Create a view task list action and an associated JSP view. The JSP view should have no JSP scriptlets. It should only use JSTL and EL.

Part 3.

Create action to add task along with a view to add tasks.

Part 4.

Create action to process the adding of a task and then display the task list JSP view.

Part 5.

Modify list view to provide link to edit task. Create an edit task action. Use the same view that was created in add task to display data to be edited.

Part 6.

Create action to process the editing of a task and then display the task list JSP view.

Chapter 9

Best Practices: Intercepting Filters

Objectives

- Learn how to modify framework to include filters
- Understand how to use the RequestContext filter
- Understand how to implement a security filter

9.1 Intercepting Filters

The framework developed in the section on best practices is a starting point for a usable web application framework. This module will extend the framework enabling it to intercept and manipulate requests before and after they are processed by the web actions.

Since all controller logic goes through the application controller it a key place where data comes into and out of the system. Since everything flows through the controller it is possible for the developer to filter the content on the way in or out.

One task which can utilize this single point of entry is system security. The choke point in the application controller makes it an ideal candidate for providing authentication and authorization services. Logging services and HTML form processing are additional services that can take advantage of filtering.

The developer is not limited to only one filter. It is possible to chain multiple filters together and provide a filtering pipeline for incoming and outgoing requests.

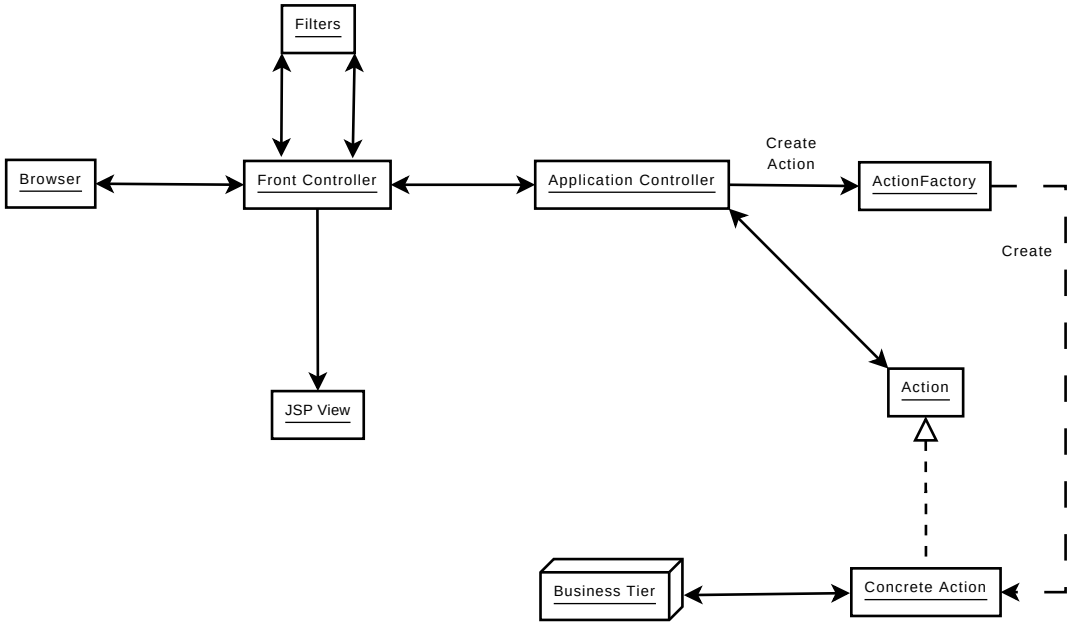

Figure 9.1: Intercepting Filters

9.2 Context Filter

The initial framework that has been developed does an excellent job separating the view from the controller logic. While the controller action is separated from the view logic, the action is still tied to the web platform. Examine the web action interface again.

```
package net.nplus1.web;

import javax.servlet.http.HttpServletRequest;

public interface WebAction {
    public String process(HttpServletRequest request) throws Exception;
}
```

Notice that anything implementing the WebAction interface has to use the HttpServletRequest object. The HttpServletRequest object is tied directly to the web container. It means the action can not be used independently of the web server. It is therefore directly tied to HTTP and the Servlet platform. One of the goals when developing the web framework was to isolate web tier data objects from the business tier. It makes no sense to use Servlet constructs within a WebAction. What the developer needs is a mechanism to convert objects from the platform specific data type to an independent data type which can be used by the business layer.

An interception filter can be used to provide a clean break from the web platform. The name for such a filter is the context filter. The goal of the context filter is to decouple the protocol dependent data structure (HTTP specific) from the action layer. It will translate to and from the HttpServletRequest object to a neutral context object usable in the action layer.

9.2.1 ContextFilter Class

The **ContextFilter** will be an extension of the web framework to provide the context filtering. There will be two filters associated with the context. The first filter must take information in the request and transfer it to the context object for processing by the action. The second filter will take data from the context and use it to populate the request object after processing by the action. This way any data from the action will be available by the view for display.

For this to happen, the filter must be broken into two distinct phases. A pre action task to create the context and a post action task to deconstruct the context. The first step is the construction of the protocol independent data type named the **RequestContext**. In the pre action filter, the ContextFilter is used to create the context from the platform specific class HttpServletRequest. The post action filter's job is to take data from the context and update the HttpServletRequest object's attributes. These two methods are outlined in the code shown below.

```
package net.nplus1.web.filter;

import javax.servlet.http.HttpServletRequest;

public class ContextFilter {
    public static RequestContext buildContext(HttpServletRequest request) {
        ...
    }

    public static void updateRequest(RequestContext context,
            HttpServletRequest request) {
        ...
    }
}
```

The *buildContext(...)* method is used to create the request context while the *updateRequest(...)* method takes the context and updates the HttpServletRequest object.

9.2.2 RequestContext

The RequestContext object must be a platform independent data type for handling requests. For it to be useful for a web application it must be able to hold data from the HttpServletRequest. It must be able to handle the request parameters and session data from the HttpServletRequest object. Of secondary importance the context can be made to handle the request's header information. In addition, the context must be able to store data being transferred from the business layer to the user interface.

All of the parameter and session data[1] used by the HttpServletRequest stores the data inside of maps. Since maps are already used for these two data types it only makes sense to store the data

[1] Header data also is stored as a name value pair even though it isn't used in this example

using the same method. It is possible to store data from the business layer using a map construct as well. Therefore we need three different maps to store all of the necessary data. The only thing left is to create accessor and mutator methods to store and retrieve the context's data. Program 17 creates the three maps and provides a way to access and changes the values of the map.

Program Listing 17 Request Context

```
package net.nplus1.web.filter;

import java.util.HashMap;

public class RequestContext {
   private HashMap<String,Object> parameterMap;
   private HashMap<String,Object> sessionMap;
   private HashMap<String,Object> dataMap;

   public RequestContext() {
       this.parameterMap = new HashMap<String,Object>();
       this.sessionMap = new HashMap<String,Object>();
       this.dataMap = new HashMap<String,Object>();
   }

   public void setParameter(String key, Object value) {
      this.parameterMap.put(key, value);
   }

   public Object getParameter(String key) {
      return this.parameterMap.get(key);
   }

   public void setAttribute(String key, Object value) {
      this.dataMap.put(key, value);
   }

   public Object getAttribute(String key) {
      return this.dataMap.get(key);
   }

   public void setSessionAttribute(String key, Object value) {
      this.sessionMap.put(key, value);
   }

   public Object getSessionAttribute(String key) {
      return this.sessionMap.get(key);
   }
}
```

Notice that the solution shown above uses no classes that are not already part of the base Java lan-

guage. Using the RequestContext as the data transfer object between the user interface and business layer removes the protocol dependency problem that existed when using the HttpServletRequest object.

9.2.3 Incoming Filter

When a HttpServletRequest comes into the Application Controller the ContextFilter is applied to the request. The *buildContext()* method is called to perform the filtering. The method's job is to retrieve the request parameters and session attributes. These values are added to the context. The header information could be retrieved at this point as well, but will be skipped to simplify the solution.

Converting Request Parameters

The most common way for developers to access request parameters is to use the *getParameter(...)* method. It returns the String value based upon the supplied key. This method works for all non multi-value parameters. Most forget the availability of the *getParameterValues(...)* method which will return a String array from the supplied key. The method will return a single entry array for normal parameters and will return all the values from a multi-value form in the HTML.

Since the single value fields can be accessed with the *getParameterValues(...)* method, the filter solution should use it for processing request elements. The first step before accessing the values is to determine the name of the parameters. The *getParameterNames()* method provides a *java.util.Enumeration* of the available parameter names. Between using *getParameterNames()* and *getParameterValues(...)* it is possible to populate the RequestContext.

```
RequestContext context = new RequestContext();
Enumeration list = request.getParameterNames();

while ( list.hasMoreElements() ) {
    String name = (String)list.nextElement();
    String [] values = request.getParameterValues(name);
    if ( values == null ) {
        context.setParameter(name, null);
    }
    if (values.length == 1) {
        context.setParameter(name, values[0]);
    }
    else {
        context.setParameter(name, values);
    }
}
```

In this case the RequestContext is instantiated and the Enumeration is obtained from the request object. The name is retrieved from the enumeration and a call is made to *getParamteterValues(...)*. If the array length is 1 then it is a single value and should be set as a single String. If the value is larger than 1 then it is a multi-value field and should be added to the context as a multi-value array. The details of storing and retrieving data from the context will be explored next.

Handling Request Parameters

The next step is to re-work the context to better take advantage of the multi-value parameters. The storing of data will not change, it is only how the data is retrieved that should matter. Therefore the *setParameter(...)* method won't change.

```
public void setParameter(String key, Object value) {
    this.parameterMap.put(key, value);
}
```

Instead, there needs to be two new methods for getting parameter data from the request. The best way to do this is to mimic the behavior of the request object itself and provide two methods. One for single value parameters and one for multi-value parameters.

First, look at the single *getParameter(...)* method. While the method returns a single value, it has no idea what data type is stored in the map. It could be a String or a String array. Therefore it must be able to handle both types of data in a singular fashion.

```
public String getParameter(String key) {
    String value = null;
    Object data = this.parameterMap.get(key);
    if ( data != null && data instanceof String ) {
        value = (String)data;
    }
    else if ( data != null && data instanceof String [] ) {
        String [] stringData = (String [])data;
        value = stringData[0];
    }
    return value;
}
```

It solves the problem with a simple conditional. If the element in the map is a single String then return the value. If the element is an array what is the best way to return a single String? The most reasonable method is to either return the first element from the array or throw an error. In the example above the method returns the first element from the array.

Handling Multi-Valued Parameters

If the developer is retrieving data from a multi-value form field then a method must be available to collect the data from the context's map. It can be accomplished by adding a *getParameterValues(...)* method. The method would take the key and return the array associated in with the key.

```
public String [] getParameterValues(String key)
```

Again the logic is simple. If the value stored is an array then return the array. If the value isn't an array then it must be a single String object. The best way to resolve this problem is to convert the String into an array with a single element.

```
public String [] getParameterValues(String key) {
    String [] values = null;
    Object data = this.parameterMap.get(key);
    if ( data != null && data instanceof String [] ) {
        values = (String [])data;
    }
    else if (data != null ) {
        values = new String[1];
        values[0] = (String)data;
    }
    return values;
}
```


Converting Session Data

Up to this point, the conversion filter has been focused on the request parameters. Another source for conversion is the session data. Data stored in the client's session object must be transposed in the same way as parameter data. The session is simpler because there is not need to worry about the data type being used. Since all session attributes are Objects, they can be directly copied to the context.

```
HttpSession session = request.getSession(true);
Enumeration list = session.getAttributeNames();
while ( list.hasMoreElements() ) {
    String name = (String)list.nextElement();
    context.setSessionAttribute(name, session.getAttribute(name));
}
```


The code above gets added to the ContextFilter to complete the *buildContext(...)* method.

```
package net.nplus1.web.filter;

import java.util.Enumeration;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;

public class ContextFilter {
    public static RequestContext buildContext(HttpServletRequest request) {
        RequestContext context = new RequestContext();
        Enumeration list = request.getParameterNames();

        while ( list.hasMoreElements() ) {
            String name = (String)list.nextElement();
            String [] values = request.getParameterValues(name);
            if ( values == null ) {
                context.setParameter(name, null);
            }
            else if (values.length == 1) {
                context.setParameter(name, values[0]);
            }
            else {
                context.setParameter(name, values);
            }
        }

        HttpSession session = request.getSession(true);
        list = session.getAttributeNames();
        while ( list.hasMoreElements() ) {
            String name = (String)list.nextElement();
            context.setSessionAttribute(name, session.getAttribute(name));
        }
    }

    public static void updateRequest(RequestContext context,
            HttpServletRequest request) {
        ...
    }
}
```


9.2.4 Outgoing Filter

Once the framework's action class has finished, the data stored within the context must be returned to the request object. It allows the JSP view access to the data stored in the context. It is the role of the outgoing filter to provide a way to refresh the request and session data.

Two tasks need to be accomplished by the outgoing filter. Any updates to the session need to be transferred to the HttpSession object. Any data added to the context must be transferred to the request object. The request parameters aren't affected because they can not be modified.

When the web action executes it has the ability to add data to the context using the *setAttribute()* method. The framework needs to make this data accessible to the JSP view. Ideally it would be accessible from the EL language. It can be accomplished by taking the attributes added to the context and add them to the request object as attributes. By adding them there, they will be accessible from EL within the JSP view.

For the developer to be able to transfer the data from the context to the request object, the context must be modified to return the set of keys associated with the attributes.

```
public Set<String> keySet() {
    return dataMap.keySet();
}
```

The key set can be used to find all of the values associated with the context and copy them over to the request.

```
Set keySet = context.keySet();
Iterator iter = keySet.iterator();
while (iter.hasNext()) {
    String key = (String)iter.next();
    request.setAttribute(key, context.getAttribute(key));
}
```

The same process can be used for refreshing the session object. The key set of the session data stored in the context must be made available to the filter.

```
public Set<String> sessionKeySet() {
    return sessionMap.keySet();
}
```


The session object can be retrieved from the request and the key set can be used to update the session with data from the context.

```
HttpSession session = request.getSession(true);
Enumeration keySet = context.sessionKeySet();
iter = keySet.iterator();
while (iter.hasNext()) {
    String key = (String)iter.next();
    session.setAttribute(key,
    context.getSessionAttribute(key));
}
```

Since all of the session data is copied to the context during the incoming filter, all of the session data is available to the web action. During the process of applying the outgoing filter the data in the context is copied back to the session object. All changes made to the session data or any new attributes will be reflected in the session object.

Listing 18 and 19 provides the code for the final context object that incorporates all of the changes discussed.

Program Listing 18 Final Request Context Part 1

```
package net.nplus1.web.filter;

import java.util.HashMap;

public class RequestContext {
    private HashMap<String,Object> parameterMap;
    private HashMap<String,Object> dataMap;
    private HashMap<String,Object> sessionMap;


    public void setParameter(String key, Object value) {
        this.parameterMap.put(key, value);
    }

    public String getParameter(String key) {
        String value = null;
        Object data = this.parameterMap.get(key);
        if ( data != null && data instanceof String ) {
            value = (String)data;
        }
        else if ( data != null && data instanceof String [] ) {
            String [] stringData = (String [])data;
            value = stringData[0];
        }
        return value;
    }

    public String [] getParameterValues(String key) {
        String [] values = null;
        Object data = this.parameterMap.get(key);
        if ( data != null && data instanceof String [] ) {
            values = (String [])data;
        }
        else if (data != null ) {
            values = new String[1];
            values[0] = (String)data;
        }
        return values;
    }
```

Program Listing 19 Final Request Context Part 2

```
    public void setAttribute(String key, Object value) {
        this.dataMap.put(key, value);
    }

    public Object getAttribute(String key) {
        return this.dataMap.get(key);
    }

    public void setSessionAttribute(String key, Object value) {
        this.sessionMap.put(key, value);
    }

    public Object getSessionAttribute(String key) {
        return this.sessionMap.get(key);
    }

    public Set<String> keySet() {
        return dataMap.keySet();
    }

    public Set<String> sessionKeySet() {
        return sessionMap.keySet();
    }
}
```


Lastly, a look at the complete ContextFilter in Program listing 20 and 21.

Program Listing 20 Context Filter Part 1

```
package net.nplus1.web.filter;

import java.util.Enumeration;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;

public class ContextFilter {
    public static RequestContext buildContext(HttpServletRequest request)  {
        RequestContext context = new RequestContext();
        Enumeration<String> list = request.getParameterNames();

        while ( list.hasMoreElements() ) {
            String name = (String)list.nextElement();
            String [] values = request.getParameterValues(name);
            if ( values == null ) {
                context.setParameter(name, null);
            }
            else if (values.length == 1) {
                context.setParameter(name, values[0]);
            }
            else {
                context.setParameter(name, values);
            }
        }

        HttpSession session = request.getSession(true);
        list = session.getAttributeNames();
        while ( list.hasMoreElements() ) {
            String name = (String)list.nextElement();
            context.setSessionAttribute(name, session.getAttribute(name));
        }
    }
```

Program Listing 21 Context Filter Part 2

```
    public static void updateRequest(RequestContext context,
                                     HttpServletRequest request) {
        Set<String> keySet = context.keySet();
        Iterator iter = keySet.iterator();
        for ( String key : keySet ) {
            request.setAttribute(key, context.getAttribute(key));
        }

        HttpSession session = request.getSession(true);
        Set<String> sessionKeySet = context.sessionKeySet();
        iter = keySet.iterator();
        for ( String key : sessionKeySet ) {
            session.setAttribute(key, context.getSessionAttribute(key));
        }
    }
}
```

9.2.5 Modifying Framework

Now that the ContextFilter has been designed, it is easy to modify the framework to utilize the filtering. The first step is the modification of the WebAction interface. The interface must be decoupled from the protocol specific HttpServletRequest object. The RequestContext is used to replace the HttpServletRequest. The new interface would have the following signature.

```
public interface WebAction {
    public String process(RequestContext request) throws Exception;
}
```

The ApplicationController has to be modified as well. This time to support the new filters. The first step is to create the RequestContext before the action is processed. The second step is to update the HttpServletRequest object with data from the RequestContext after the action is processed. In the code modification below the incoming and outgoing filter section is marked with comments.

```
package net.nplus1.web;

import javax.servlet.http.HttpServletRequest;

public class ApplicationController {
    public String processRequest(String url, HttpServletRequest request) {
        String view = null;

        try {
            // incoming filters
            RequestContext context = ContextFilter.buildContext(request);

            // process action
            WebAction action = WebActionFactory.createAction(url);
            if ( action != null ) {
                String viewName = action.process(request, context);
                view = LookupJSP.lookup(viewName);
            }

            // outgoing filters
            ContextFilter.updateRequest(context, request);
        }
        catch (Exception e) {
            view = "/error.jsp";
        }

        return view;
    }
}
```

Applying the filters finishes the work necessary for the creation of a context filtering system.

9.3 Security Filtering

Since all requests go through the application controller it becomes the perfect location to provide security for the system. This singular location can act as a traffic cop to determine authorization and authentication.

Authentication is the first priority. If the user is not authenticated to access the system the traffic cop can redirect the user to a login page to challenge their credentials. Authorization is the second priority. If the user is not authorized to access the web action then the traffic cop can log the violation and redirect the user to an illegal access page.

A custom SecurityFilter can fill the role of the traffic cop. It can be added to the framework to provide authentication and authorization services. The centrality of the security code makes it easier to maintain and makes it easier to apply it consistently across the system. It could even be integrated with the JAAS API to provide a more solid security model.

At this point, it is important to note that the discussion and examples that follow provide a rudimentary security system. The goal is to show how to implement a security mechanism within the web framework. There isn't time to go into all of the security concerns with web applications. The system outline below should be considered a minimal security measure. In a production environment a more robust and tested security system should be implemented. Integrating JAAS would be a step in the right direction.

9.3.1 Authentication Filter

The first step in securing a web application is to provide an authentication mechanism. Authentication is the process of verifying a user's identification. Authentication usually involves a challenge process where the system will ask the user to confirm who he or she is. The most common form of challenge is through the use of a user identifier and password.

Before looking at the process of challenging users, it is important to define a data structure to be used in the authentication and authorization process. In the JAAS standards this object is known as a Principal. For the purposes of the class, the courseware will create a custom principal like object. The object will have a unique identifier, properties of a user such as name, and a list of roles that can be tested for authorization. To ensure that other developers don't accidentally change security information, the roles and unique identifiers should be immutable.

```
package net.nplus1.web.filter;

import java.util.List;

public class SecureUser {
    private String userId;
    private String name;
    private List<String> roles;

    public SecureUser(String userId, List<String> roles) {
        this.userId = userId;
        this.roles = roles;

        // accessors
        public String getName() { return name; }
        public String getUserId() { return userId; }
        public List<String> getRoles() { return roles; }

        // mutators
        public void setName(String name) { this.name = name; }
    }
}
```

The SecureUser user listed above will be used as a token for determining authentication.

Challenging The User

When the user successfully passes the challenge stage, the authenticated token can be added to the session object. Generally a mechanism should be employed to make sure the token is not accidentally overridden by the application. One means of accomplishing this is by using a special signature for framework calls. In this case a double _ is used to mark the token.

```
public static boolean challengeUser(HttpServletRequest request) {
    String userId = request.getParameter("userId");
    String password = request.getParameter("password");

    boolean authenticated = false;
    SecureUser user = null;

    // code to check authentication against a user store
    // create new SecureUser based upon that data

    if ( authenticated ) {
        HttpSession session = request.getSession();
        session.setAttribute("__secureUser__", user);
    }

    return authenticated;
}
```

The code above omits the process of checking the user's credentials against some form of user store for a simple reason. Most user stores are either stored in databases or LDAP like mechanisms. The code for attaching to either of these destinations is beyond the scope of this class.

Authentication Filter

The next step in developing an authentication filter is creating a method that will determine if the user has been authenticated.

```
public static boolean userAuthenticated(HttpServletRequest request) {
    boolean authenticated = false;
    HttpSession session = request.getSession();
    Object token = session.getAttribute("__secureUser__");
    if ( token != null && token instanceof SecureUser) {
        authenticated = true;
    }

    return authenticated;
}
```

The code checks to make sure the user has a valid authenticated SecureUser token. If the client does then the user is successfully authenticated and can be allowed to pass beyond the filter.If the user does not have a token, then the controller can redirect the user to a challenge view.

The complete SecurityFilter is shown in Listing 22.

Program Listing 22 Security Filter

```
package net.nplus1.web.filter;

import javax.servlet.http.HttpSession;
import javax.servlet.http.HttpServletRequest;

public class SecurityFilter {
    public static boolean challengeUser(HttpServletRequest request) {
        String userId = request.getParameter("userId");
        String password = request.getParameter("password");
        boolean authenticated = false;
        SecureUser user = null;

        // code to check authentication against a user store
        // create new SecureUser based upon that data

        if ( authenticated ) {
            HttpSession session = request.getSession();
            session.setAttribute("__secureUser__", user);
        }

        return authenticated;
    }

    public static boolean userAuthenticated(HttpServletRequest request) {
        boolean authenticated = false;
        HttpSession session = request.getSession();
        Object token = session.getAttribute("__secureUser__");

        if ( token != null && token instanceof SecureUser) {
            authenticated = true;
        }

        return authenticated;
    }
}
```

Modifying The Framework

The next step is to integrate the SecruityFilter into the framework. The filter needs to address two issues. First it needs to check to see if the user is submitting a challenge. This can be done by looking at the incoming URL. If the challenge is not successful the user needs to be redirected to the login along with an error message.

```
if ( url.equals("process_login.html") ) {
    if ( !SecurityFilter.challengeUser(request) ) {
        request.setAttribute("error", "Invalid Login");
        return LookupJSP.lookup("login");
    }
}
```

The second task guarantees the authentication of the user. If the user is not authenticated then re-direct the user to the login page.

```
if ( !SecurityFilter.userAuthenticated(request) ) {
    return LookupJSP.lookup("login");
}
```


The new controller code is shown below.

```
package net.nplus1.web;

import net.nplus1.web.filter.RequestContext;
import net.nplus1.web.filter.ContextFilter;
import net.nplus1.web.filter.SecurityFilter;
import javax.servlet.http.HttpServletRequest;

public class ApplicationController {
    public String processRequest(String url, HttpServletRequest request) {

        String view = null;

        try {
            // security filter
            if ( url.equals("process_login.html") ) {
                if ( !SecurityFilter.challengeUser(request) ) {
                    request.setAttribute("error", "Invalid Login");
                    return LookupJSP.lookup("login");
                }
            }
            if ( !SecurityFilter.userAuthenticated(request) ) {
                return LookupJSP.lookup("login");
            }

            // context filter
            RequestContext context = ContextFilter.buildContext(request);

            // process action
            WebAction action = WebActionFactory.createAction(url);
            if ( action != null ) {
                String viewName = action.process(context);
                view = LookupJSP.lookup(viewName);
            }

            // outgoing filters
            ContextFilter.updateRequest(context, request);
        }
        catch (Exception e) {
            view = "jsp/error.jsp";
        }

        return view;
    }
}
```


9.3.2 Authorization Filter

Once the user's credentials have been established, a system for providing authorization can be created. In this instance, the courseware will develop a role based access for the system. A mechanism that allows users to be added to roles and the roles are granted access to resources.

There are a number of different ways to implement role based authorization. The best way to control access is to do it at the web action level. Since each action is an independent entity within the system, it makes sense to provide access control at this level. One method for accomplishing this task is to map actions to roles using an XML file. Below is an example of a partial XML file that performs this mapping.

```
<role-security>
    <action>AddTodoList</action>
    <roles>
        <role>client</role>
        <role>admin</role>
    </roles>
</role-security>
```

Another way to handle role to action mapping is to associate the role information with the actual web action. It can be done in one of two ways. One is to associated a role annotation to the action class. Annotations are a way to associated meta data with code. Roles would just be a form of meta information to associate with a class.

```
public class AddTodoList implements WebAction {
    @SecureWebaction(roles="client,admin")
    public String process(RequestContext context) throws Exception {
        ...
    }
}
```

While annotations are an elegant way to accomplish this task, the solution involves the processing of annotations which goes beyond the scope of this class. Instead a simpler mechanism will be used to perform the same function. It will be done with the use of tagging interfaces. Where each role has an associated tagging interface. The action would then implement the tagging interfaces for the roles that have access to the system.

```
public class AddTodoList implements WebAction, ClientRole, AdminRole {
    public String process(RequestContext context) throws Exception {
        ...
    }
}
```


Tagging Interfaces

What is a tagging interface? It is an interface that has no methods associated with it.

```
public interface Client {}
public interface Admin {}
```

When associating an interface with a class two things happen. The class must implement the methods of the interface and the class can be polymorphically treated as the interface. Since the tagging interface doesn't have any methods, the class doesn't have to do anything to implement the interface. In addition the class can be treated as the interface. Since the interface doesn't have any methods this feature can have limited value.

The one thing the tagging interface provides is a way to mark a class. It is possible using the *instanceof* operator to determine if a class implements a particular interface. Thus it can be used to determine if the web action implements one of the role interfaces.

```
WebAction action = new (WebAction)AddTodoList();

if ( action instanceof ClientRole ) {
    System.out.println("Client has access to this web action");
}
```


It becomes easy to create a manager class that takes a web action and determine what roles have been associated with the action through tagging interfaces.

```
package net.nplus1.web.filter;

import net.nplus1.web.WebAction;
import java.util.List;
import java.util.ArrayList;

public class RoleManager {
    public static List<String> getActionRoles(WebAction action) {
        ArrayList<String> roles = new ArrayList<String>();
        if ( action instanceof ClientRole ) {
            roles.add("client");
        }
        if ( action instanceof AdminRole ) {
            roles.add("admin");
        }
        if ( action instanceof VisitorRole ) {
            roles.add("visitor");
        }

        return roles;
    }
}
```

In this case a string of roles are generated based upon the types of tagging interfaces that the web action class implements.

Authorization Code

Once a method is created to determine the roles associated with an action, the next step is to compare the user's roles with the action's roles. If they overlap then access to the action is granted. If not then the user should be directed to a message informing them that access has been denied an their attempt was logged.

A method can be added to the SecurityFilter class to test to see if the user is authorized to access an action.

```
package net.nplus1.web.filter;

import net.nplus1.web.WebAction;

import java.util.List;
import javax.servlet.http.HttpServletRequest;

public class SecurityFilter {
    public static boolean authorizeUser(WebAction action,
                            HttpServletRequest request) {
        boolean validUser = false;

        // get action roles
        List<String> actionRoles = RoleManager.getActionRoles(action);

        // get user roles
        HttpSession session = request.getSession();
        SecureUser user = (SecureUser)session.getAttribute("__secureUser");

        if ( user != null ) {
            List<String> userRoles = user.getRoles();
            for ( String userRole : userRoles ) {
                for ( String actionRole : actionRoles ) {
                    if ( userRole.equalsIgnoreCase(actionRole) ) {
                        validUser = true;
                        break;
                    }
                }
                if ( validUser ) {
                    break;
                }
            }
        }
    }
}
```

In this case the list of roles associated with the action are retrieved along with the user's list of roles.

They are compared to see if there is any match of roles. If so then the user is authorized.

Adding User Information To Context

Web developers frequently need access to information about the user who has been authorized to use the system. Rather than force programmers to retrieve the SecureUser directly from the HttpSession object, it is easier to add the secure user to the context.

This can be accomplished with two quick changes to the filtering process. The first step is to modify the context to allow the addition of a SecureUser.

```
public class RequestContext {
    private SecureUser user;

    public void setSecureUser(SecureUser user) {
        this.user = user;
    }

    public SecureUser getSecureUser() {
        return user;
    }
}
```

Once the context is retro-fitted for the SecureUser, a method can be added to the SecurityFitler to put the secure user into the context.

```
public static void setUserContext(HttpServletRequest, request,
                                  RequestContext context) {
    HttpSession session = request.getSession();
    SecureUser user = (SecureUser)session.getAttribute("__secureUser__");
    context.setSecureUser(user);
}
```


Since the SecureUser is being added to the RequestContext, the SecurityFilter can be re-written to use the RequestContext instead of the HttpSession object. The final security manager looks like the following.

```
package net.nplus1.web.filter;

import net.nplus1.web.WebAction;
import java.util.List;
import javax.servlet.http.HttpSession;
import javax.servlet.http.HttpServletRequest;

public class SecurityFilter {
    /*
    * AUTHORIZATION CODE
    */
    public static boolean userAuthorized(WebAction action,
                                         RequestContext context) {
        boolean validUser = false;

        // get action roles
        List<String> actionRoles = RoleManager.getActionRoles(action);

        // get user roles
        SecureUser user = context.getSecureUser();

        if ( user != null ) {
            List<String> userRoles = user.getRoles();
            for ( String userRole : userRoles ) {
                for ( String actionRole : actionRoles ) {
                    if ( userRole.equalsIgnoreCase(actionRole) ) {
                        validUser = true;
                        break;
                    }
                }
                if ( validUser ) {
                    break;
                }
            }
        }
    }
```



```
    public static void setUserContext(HttpServletRequest, request,
                                RequestContext context) {
        HttpSession session = request.getSession();
        SecureUser user = (SecureUser)session.getAttribute("__secureUser__");
        context.setSecureUser(user);
    }

    /*
    * AUTHENTICATION CODE
    */
    public static boolean challengeUser(HttpServletRequest request) {
        String userId = request.getParameter("userId");
        String password = request.getParameter("password");

        boolean authenticated = false;
        SecureUser user = null;

        // code to check authentication against a user store
        // create new SecureUser based upon that data

        if ( authenticated ) {
            HttpSession session = request.getSession();
            session.setAttribute("__secureUser__", user);
        }

        return authenticated;
    }

    public static boolean userAuthenticated(HttpServletRequest request) {
        boolean authenticated = false;
        HttpSession session = request.getSession();
        Object token = session.getAttribute("__secureUser__");

        if ( token != null && token instanceof SecureUser) {
            authenticated = true;
        }

        return authenticated;
    }
}
```


Modifying The Framework

The last step is modifying the framework to use the SecurityFilter. The first thing the framework must do is add the secure user to the request context after authentication. Once the user is added to the context and the web action is created, the security filter can be applied to determine if the user is authorized to execute the requested action.

If the user is not authorized to use the system, then the framework must provide an invalid access error message to the end user. This is where any security logging might take place. In the example below the user is shown the "invalid access" page.


```
package net.nplus1.web;

import net.nplus1.web.filter.RequestContext;
import net.nplus1.web.filter.ContextFilter;
import net.nplus1.web.filter.SecurityFilter;
import javax.servlet.http.HttpServletRequest;

public class ApplicationController {
    public String processRequest(String url, HttpServletRequest request) {
        String view = null;

        try {
            // security filter
            if ( url.equals("process_login.html") ) {
                if ( !SecurityFilter.challengeUser(request) ) {
                    request.setAttribute("error", "Invalid Login");
                    return LookupJSP.lookup("login");
                }
            }
            if ( !SecurityFilter.userAuthenticated(request) ) {
                return LookupJSP.lookup("login");
            }

            // context filter creation
            RequestContext context = ContextFilter.buildContext(request);

            // add secure user to context
            SecurityFilter.setUserContext(request, context);

            // create action
            WebAction action = WebActionFactory.createAction(url);

            if ( action != null ) {
                // authorization filter
                if ( !SecurityFilter.userAuthorized(action, request) ) {
                    return LookupJSP.lookup("invalid access");
                }

                // process action
                String viewName = action.process(context);
                view = LookupJSP.lookup(viewName);
            }

            // outgoing filters
            ContextFilter.updateRequest(context, request);
        }
        catch (Exception e) {
            view = "jsp/error.jsp";
```



```
            }

            return view;
        }
    }
```


Final Thoughts On Authorization

The tagging interface mechanism was chosen because of its simplicity. It was the easiest mechanism for showing how to apply authorization for a user. In a production system either the XML or annotation method should be utilized depending on needs. If a developer wants to centralize role access then the XML file should be used. If the user wants a method for associating roles with action in a decentralized manner the annotation method should be used. Annotations are preferable to tagging interfaces because they are not compiled as the source code. The role data is meta data that needs to be associated with the class. A perfect fit for annotations.

9.4 Lab Activity

Part 1.

Modify the framework developed in the best practices lab to use the Context Filter.

Part 2.

Modify the actions from solution in the best practices lab to use the context instead of the HttpServletRequest object.

Part 3.

Add the ability to delete tasks from the task list.

Appendices

A Lab Results

The results from the labs are shown below.

A.1 Lab 2

Part 1

```
<html>
    <body>
        <h2>Enter Data</h2>
        <form method="post" action="">
            First Name: <input type="text" name="firstName"/></br>
            Last Name: <input type="text" name="lastName"/></br>
            City: <input type="text" name="city"/></br>
            State:
                <select name="state" size="1">
                    <option value="IN">IN</option>
                    <option value="KY">KY</option>
                    <option value="OH">OH</option>
                    <option value="TN">TN</option>
                </select><br/>
            Sex:
                <input type="radio" name="sex" value="m"/> Male
                 <input type="radio" name="sex" value="f"/> Female<br/>
            Pets: <input type="checkbox" name="pet" value="cat"/> Cat
                <input type="checkbox" name="pet" value="cat"/> Dog
                <input type="checkbox" name="pet" value="cat"/> Goat
                <input type="checkbox" name="pet" value="cat"/> Llama
                </br>
            Comments:
            <textarea name="comments" cols="50" rows="6"></textarea></br>
            <input type="submit" value="Save"/></br>
        </form>
    </body>
</html>
```


Part 2

```
<html>
    <body>
        <h2>Enter Data</h2>
        <form method="post" action="">
        <table border="0" cellspacing="0" cellpadding="4">
            <tr>
                <td align="right"> First Name: </td>
                <td> <input type="text" name="firstName"/> </td>
            </tr>
            <tr>
                <td align="right"> Last Name: </td>
                <td> <input type="text" name="lastName"/> </td>
            </tr>
            <tr>
                <td align="right"> City: </td>
                <td> <input type="text" name="city"/> </td>
            </tr>
            <tr>
                <td align="right"> State: </td>
                <td>
                    <select name="state" size="1">
                        <option value="IN">IN</option>
                        <option value="KY">KY</option>
                        <option value="OH">OH</option>
                        <option value="TN">TN</option>
                    </select>
                </td>
            </tr>
            <tr>
                <td align="right"> Sex: </td>
                <td>
                    <input type="radio" name="sex" value="m"/> Male
                    <input type="radio" name="sex" value="f"/> Female
                </td>
            </tr>
            <tr>
                <td align="right"> Pets: </td>
                <td>
                    <input type="checkbox" name="pet" value="cat"/> Cat
                    <input type="checkbox" name="pet" value="cat"/> Dog
                    <input type="checkbox" name="pet" value="cat"/> Goat
                    <input type="checkbox" name="pet" value="cat"/> Llama
                </td>
            </tr>
            <tr>
                <td align="right" valign="top"> Comments: </td>
```



```
                <td>
                    <textarea name="comments" cols="50" rows="6"></textarea>
                </td>
            </tr>
            <tr>
                <td></td>
                <td align="right"> <input type="submit" value="Save"/> </td>
            </tr>
        </table>
        </form>
    </body>
</html>
```

Part 3

```
<html>
    <body>
        <h2>Enter Data</h2>
        <form method="post" action="">
        <table border="0" cellspacing="0" cellpadding="4">
            <tr>
                <td align="right"> First Name: </td>
                <td> <input type="text" name="firstName"/> </td>
            </tr>
            <tr>
                <td align="right"> Last Name: </td>
                <td> <input type="text" name="lastName"/> </td>
            </tr>
            <tr>
                <td align="right"> City: </td>
                <td> <input type="text" name="city" value="Waddy"/> </td>
            </tr>
            <tr>
                <td align="right"> State: </td>
                <td>
                    <select name="state" size="1">
                        <option value="IN">IN</option>
                        <option value="KY" selected="true">KY</option>
                        <option value="OH">OH</option>
                        <option value="TN">TN</option>
                    </select>
                </td>
            </tr>
            <tr>
                <td align="right"> Sex: </td>
                <td>
                    <input type="radio" name="sex" value="m"/> Male
                    <input type="radio" name="sex" value="f"/> Female
                </td>
            </tr>
            <tr>
                <td align="right"> Pets: </td>
                <td>
                    <input type="checkbox" checked="true"
                                           name="pet" value="cat"/> Cat
                    <input type="checkbox" name="pet" value="cat"/> Dog
                    <input type="checkbox" name="pet" value="cat"/> Goat
                    <input type="checkbox" name="pet" value="cat"/> Llama
                </td>
            </tr>
            <tr>
```



```
                <td align="right" valign="top"> Comments: </td>
                <td>
                     <textarea name="comments" cols="50"
                               rows="6">Enter Comments Here</textarea>
                </td>
            </tr>
            <tr>
                <td></td>
                <td align="right"> <input type="submit" value="Save"/> </td>
            </tr>
        </table>
        </form>
    </body>
</html>
```

A.2 Lab 3

Part 1 - Countdown.java and web.xml

```
package net.nplus1.javaweb.lab3;

import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class Countdown extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response)
                            throws IOException, ServletException {
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<body>");
        out.println("<h2>Countdown</h2>");
        for ( int x = 10; x > 0; x-- ) {
            out.println(x + "<br/>");
        }
        out.println("</body>");
        out.println("</html>");
    }
}

<?xml version="1.0" encoding="UTF-8"?>
<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/web-app_2_4.xsd"
       version="2.4">
    <servlet>
        <servlet-name>Countdown</servlet-name>
        <servlet-class>net.nplus1.javaweb.lab3.Countdown</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>Countdown</servlet-name>
        <url-pattern>/lab3/countdown.html</url-pattern>
    </servlet-mapping>
</web-app>
```


Part 2 - build.xml

```
<project name="courseware" default="deploy-web" basedir=".">

    <!-- import property file -->
    <property file="./build.properties"/>

    <!-- set path information -->
    <path id="jboss.server.classpath">
        <fileset dir="${jboss.server.lib}">
            <include name="**/*.jar"/>
        </fileset>
    </path>
    <path id="jboss.client.classpath">
        <fileset dir="${jboss.client.lib}">
            <include name="**/*.jar"/>
        </fileset>
    </path>

    <target name="build">
        <mkdir dir="${build}"/>
        <javac srcdir="${src}" destdir="${build}">
            <classpath>
                <path refid="jboss.server.classpath"/>
                <path refid="jboss.client.classpath"/>
            </classpath>
        </javac>
    </target>

    <target name="package-web" depends="build">
        <mkdir dir="${dist}"/>
        <war destfile="${dist}/course.war" webxml="${conf.web}/web.xml">
            <classes dir="${build}"/>
            <zipfileset dir="${web.tags}" prefix="WEB-INF/tags"/>
            <zipfileset dir="${web.jsp}"/>
        </war>
    </target>

    <target name="deploy-web" depends="package-web">
        <copy todir="${jboss.server.deploy}" file="${dist}/course.war"/>
    </target>

    <target name="remove-web">
        <delete file="${jboss.server.deploy}/course.war"/>
    </target>

    <target name="clean">
        <delete dir="${build}"/>
```



```
        <delete dir="${dist}"/>
    </target>

</project>
```

Part 3 - OddCountdown.java

```
public class OddCountdown extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response)
                      throws IOException, ServletException {
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<body>");
        out.println("<h2>Countdown</h2>");
        for ( int x = 10; x > 0; x-- ) {
            if ( x % 2 == 1 ) {
                out.println(x + "<br/>");
            }
        }
        out.println("</body>");
        out.println("</html>");
    }
}
```

Part 4 - PhoneForm.java

```
package net.nplus1.javaweb.lab3;

import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class PhoneForm extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response)
                      throws IOException, ServletException {
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<body>");
        out.println("<h2>Phone Numbers</h2>");
        out.println("<form method=\"post\" action=\"\">");
        out.println("<table border=\"0\" cellpadding=\"4\" cellspacing=\"0\">");
        out.println("<tr>");
        out.println("<td>First Name:</td>");
        out.println("<td><input type=\"text\" name=\"firstName\"/></td>");
        out.println("</tr>");
        out.println("<tr>");
        out.println("<td>Last Name:</td>");
        out.println("<td><input type=\"text\" name=\"lastName\"/></td>");
        out.println("</tr>");
        out.println("<tr>");
        out.println("<td>Phone Number:</td>");
        out.println("<td><input type=\"text\" name=\"phone\"/></td>");
        out.println("</tr>");
        out.println("<tr>");
        out.println("<td></td>");
        out.println("<td><input type=\"submit\" value=\"Enter\"/></td>");
        out.println("</tr>");
        out.println("</table>");
        out.println("</form>");
        out.println("</body>");
        out.println("</html>");
    }
}
```

A.3 Lab 4

Part 1 - PhoneEntry.java

```
package net.nplus1.javaweb.lab4;

import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class PhoneEntry extends HttpServlet {
    public void doPost(HttpServletRequest request, HttpServletResponse response)
                        throws IOException, ServletException {
        this.doGet(request, response);
    }

    public void doGet(HttpServletRequest request, HttpServletResponse response)
                        throws IOException, ServletException {
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<body>");
        out.println("<h2>Enter Phone Number</h2>");

        String error = (String)request.getAttribute("errorMessage");
        if ( error != null && !error.isEmpty() ) {
            out.println("<p>Error occured:<br/> " + error + " </p>");
        }

        out.println("<form method=\"post\" action=\"process_phone.html\">");
        out.println("<table border=\"0\" cellpadding=\"4\" cellspacing=\"0\">");
        out.println("<tr>");
        out.println("<td>First Name:</td>");
        out.println("<td><input type=\"text\" name=\"firstName\"/></td>");
        out.println("</tr>");
        out.println("<tr>");
        out.println("<td>Last Name:</td>");
        out.println("<td><input type=\"text\" name=\"lastName\"/></td>");
        out.println("</tr>");
        out.println("<tr>");
        out.println("<td>Phone Number:</td>");
        out.println("<td><input type=\"text\" name=\"phone\"/></td>");
        out.println("</tr>");
        out.println("<tr>");
```



```
        out.println("<td></td>");
        out.println("<td><input type=\"submit\" value=\"Enter\"/></td>");
        out.println("</tr>");
        out.println("</table>");
        out.println("</form>");
        out.println("</body>");
        out.println("</html>");
    }
}
```

Part 2 - ProcessPhone.java

```
package net.nplus1.javaweb.lab4;

import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class ProcessPhone extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
                      throws IOException, ServletException {
        response.sendError(HttpServletResponse.SC_NOT_FOUND);
    }

    public void doPost(HttpServletRequest request, HttpServletResponse response)
                       throws IOException, ServletException {
        String firstName = request.getParameter("firstName");
        String lastName = request.getParameter("lastName");
        String phone = request.getParameter("phone");

        String errorMessage = "";
        if ( firstName.trim().isEmpty() ) {
            errorMessage += "First name is missing<br/>";
        }
        if ( lastName.trim().isEmpty() ) {
            errorMessage += "Last name is missing<br/>";
        }

        String testPhone = phone.replaceAll("-", "");

        if (testPhone.length() != 10) {
            errorMessage += "Invalid number of digits";
        }
        else {
            try {
                Long.parseLong(testPhone);
            }
            catch (NumberFormatException e) {
                errorMessage += "Invalid phone number";
            }
        }
```



```
        if ( !errorMessage.isEmpty() ) {
            request.setAttribute("errorMessage", errorMessage);
            RequestDispatcher dispatch =
                this.getServletContext().getRequestDispatcher(
                                                    "/lab4/phone_entry.html");
            dispatch.forward(request, response);
        }

        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<body>");
        out.println("<h2>Survey Completed</h2>");
        out.println("Thank you " + firstName + " " + lastName + " for " +
                    "filling out the survey.<br/>");
        out.println("We have recorded your phone number as " + phone);
        out.println("</body>");
        out.println("</html>");
    }
}
```

Part 3, 4, 5 - PhoneEntryTwo.java and ProcessPhoneTwo.java

PhoneEntryTwo.java

```
package net.nplus1.javaweb.lab4;

import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class PhoneEntryTwo extends HttpServlet {
    public void doPost(HttpServletRequest request, HttpServletResponse response)
                      throws IOException, ServletException {
        this.doGet(request, response);
    }

    public void doGet(HttpServletRequest request, HttpServletResponse response)
                     throws IOException, ServletException {
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<body>");
        out.println("<h2>Enter Phone Number</h2>");

        String error = (String)request.getParameter("error");
        if ( error != null && !error.isEmpty() ) {
            int errorMessage = Integer.parseInt(error);
            out.println("<p>Error occured:<br/> ");
            if ( errorMessage > 7 ) {
                out.println("No first name entered<br/>");
                errorMessage -= 8;
            }
            if ( errorMessage > 3 ) {
                out.println("No last name entered<br/>");
                errorMessage -= 4;
            }
            if ( errorMessage > 1 ) {
                out.println("Not enough numbers in phone number<br/>");
                errorMessage -= 2;
            }
            if ( errorMessage == 1 ) {
                out.println("Invalid phone number<br/>");
            }
            out.println("</p>");
        }
```



```
        out.println("<form method=\"post\" action=\"process_phone2.html\">");
        out.println("<table border=\"0\" cellpadding=\"4\" cellspacing=\"0\">");
        out.println("<tr>");
        out.println("<td>First Name:</td>");
        out.println("<td><input type=\"text\" name=\"firstName\"/></td>");
        out.println("</tr>");
        out.println("<tr>");
        out.println("<td>Last Name:</td>");
        out.println("<td><input type=\"text\" name=\"lastName\"/></td>");
        out.println("</tr>");
        out.println("<tr>");
        out.println("<td>Phone Number:</td>");
        out.println("<td><input type=\"text\" name=\"phone\"/></td>");
        out.println("</tr>");
        out.println("<tr>");
        out.println("<td></td>");
        out.println("<td><input type=\"submit\" value=\"Enter\"/></td>");
        out.println("</tr>");
        out.println("</table>");
        out.println("</form>");
        out.println("</body>");
        out.println("</html>");
    }
}
```

ProcessPhoneTwo.java

```
package net.nplus1.javaweb.lab4;

import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class ProcessPhoneTwo extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response)
                      throws IOException, ServletException {
        response.sendError(HttpServletResponse.SC_NOT_FOUND);
    }

    public void doPost(HttpServletRequest request, HttpServletResponse response)
                       throws IOException, ServletException {
        String firstName = request.getParameter("firstName");
        String lastName = request.getParameter("lastName");
        String phone = request.getParameter("phone");

        int errorMessage = 0;
        if ( firstName.trim().isEmpty() ) {
            errorMessage += 8;
        }
        if ( lastName.trim().isEmpty() ) {
            errorMessage += 4;
        }

        String testPhone = phone.replaceAll("-", "");

        if (testPhone.length() != 10) {
            errorMessage += 2;
        }
        else {
            try {
                Long.parseLong(testPhone);
            }
            catch (NumberFormatException e) {
                errorMessage += 1;
            }
        }

        if ( errorMessage > 0 ) {
            response.sendRedirect("phone_entry2.html?error=" + errorMessage);
```



```
        }

        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<body>");
        out.println("<h2>Survey Completed</h2>");
        out.println("Thank you " + firstName + " " + lastName +
                    " for filling out the survey.<br/>");
        out.println("We have recorded your phone number as " + phone);
        out.println("</body>");
        out.println("</html>");
    }
}
```

Part 6 - web.xml and error.html

web.xml

```
<error-page>
    <error-code>404</error-code>
    <location>/lab4/error.html</location>
</error-page>
```

error.html

```
<html>
    <body>
        <h2>The Page You Requested is Not Available</h2>
    </body>
</html>
```


A.4 Lab 5

Task.java

```
package net.nplus1.javaweb.lab5;

public class Task {
    private int priority;
    private String dueDate;
    private String task;

    public int getPriority() {
        return priority;
    }
    public void setPriority(int priority) {
        this.priority = priority;
    }
    public String getDueDate() {
        return dueDate;
    }
    public void setDueDate(String dueDate) {
        this.dueDate = dueDate;
    }
    public String getTask() {
        return task;
    }
    public void setTask(String task) {
        this.task = task;
    }
}
```

ViewTaskList.java

```
package net.nplus1.javaweb.lab5;

import java.io.IOException;
import java.io.PrintWriter;
import java.util.ArrayList;
import java.util.List;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

public class ViewTaskList extends HttpServlet {
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
                        throws ServletException, IOException {
        HttpSession session = req.getSession();

        List<Task> tasks = (List<Task>)session.getAttribute("taskList");
        if ( tasks == null ) {
            tasks = new ArrayList<Task>();
            session.setAttribute("taskList", tasks);
        }

        PrintWriter out = resp.getWriter();
        out.println("<html><body>");
        out.println("<h2>Task List</h2>");
        out.println("<form method=\"post\" action=\"task_list.html\">");
        out.println("<input type=\"submit\" value=\"Delete Tasks\"/>");
        out.println("<input type=\"hidden\" name=\"action\" value=\"delete\"/>");
        out.println("<p><table border=\"1\" cellpadding=\"4\">");
        out.println("<tr><td>x</td><td>Pri</td><td>Due</td><td>Task</td></tr>");
        for ( int x = 0; x < tasks.size(); x++ ) {
            Task task = tasks.get(x);
            out.println("<tr>");
            out.println("<td><input type=\"checkbox\" name=\"id\" value=\""
                        + x + "\"></td>");
            out.println("<td>" + task.getPriority() + "</td>");
            out.println("<td>" + task.getDueDate() + "</td>");
            out.print("<td>");
            out.print("<a href=\"edit_task.html?id=" + x + "\">"
                      + task.getTask() + "</a>");
            out.println("</td>");
            out.println("</tr>");
        }
        out.println("</table></p>");
```



```
        out.println("</form>");
        out.println("<form method=\"post\" action=\"task_list.html\">");
        out.println("<input type=\"hidden\" name=\"action\" value=\"add\"/>");
        out.println("Priority: <input type=\"text\" name=\"priority\"/><br/>");
        out.println("Due Date: <input type=\"text\" name=\"dueDate\"/><br/>");
        out.println("Task: <input type=\"text\" name=\"task\"/><br/>");
        out.println("<input type=\"submit\" name=\"Add Task\"/><br/>");
        out.println("</form>");
        out.println("</body></html>");
    }

    protected void doPost(HttpServletRequest req, HttpServletResponse resp)
                          throws ServletException, IOException {
        String action = req.getParameter("action");
        HttpSession session = req.getSession();
        List<Task> tasks = (List<Task>)session.getAttribute("taskList");

        if ( action.equals("add") ) {
            String priority = req.getParameter("priority");
            String dueDate = req.getParameter("dueDate");
            String task = req.getParameter("task");
            int pri = Integer.parseInt(priority);

            Task newTask = new Task();
            newTask.setDueDate(dueDate);
            newTask.setPriority(pri);
            newTask.setTask(task);

            tasks.add(newTask);
        }
        if ( action.equals("delete") ) {
            String [] indexes = req.getParameterValues("id");
            for ( int x = indexes.length - 1; x >= 0; x-- ) {
                int index = Integer.parseInt(indexes[x]);
                tasks.remove(index);
            }
        }

        this.doGet(req, resp);
    }

}
```

EditTask.java

```
package net.nplus1.javaweb.lab5;

import java.io.IOException;
import java.io.PrintWriter;
import java.util.List;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

public class EditTask extends HttpServlet {
    public void doGet(HttpServletRequest req, HttpServletResponse resp)
                        throws ServletException, IOException {
        HttpSession session = req.getSession();
        List<Task> tasks = (List<Task>)session.getAttribute("taskList");

        String id = req.getParameter("id");
        int index = Integer.parseInt(id);
        Task task = tasks.get(index);

        PrintWriter out = resp.getWriter();
        out.println("<html><body>");
        out.println("<h2>Edit Task</h2>");
        out.println("<form method=\"post\" action=\"edit_task.html\">");
        out.println("<input type=\"hidden\" name=\"id\" value=\"" + id
                    + "\"/>");
        out.println("Priority: <input type=\"text\" name=\"priority\" value=\""
                    + task.getPriority() + "\"/><br/>");
        out.println("Due Date: <input type=\"text\" name=\"dueDate\" value=\""
                    + task.getDueDate() + "\"/><br/>");
        out.println("Task: <input type=\"text\" name=\"task\" value=\""
                    + task.getTask() + "\"/><br/>");
        out.println("<input type=\"submit\" value=\"Edit Task\"/><br/>");
        out.println("</form>");
        out.println("</body></html>");
    }

    public void doPost(HttpServletRequest req, HttpServletResponse resp)
                        throws ServletException, IOException {
        HttpSession session = req.getSession();
        List<Task> tasks = (List<Task>)session.getAttribute("taskList");

        String id = req.getParameter("id");
        int index = Integer.parseInt(id);
```



```
        String priority = req.getParameter("priority");
        String dueDate = req.getParameter("dueDate");
        String task = req.getParameter("task");
        int pri = Integer.parseInt(priority);

        Task editTask = tasks.get(index);
        editTask.setDueDate(dueDate);
        editTask.setPriority(pri);
        editTask.setTask(task);

        resp.sendRedirect("task_list.html");
    }
}
```

A.5 Lab 6

header.jsp

```
<html>
    <body>
        <h2>Todo List</h2>
        <p>
```

footer.jsp

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


todo_list.jsp

```
<%@ page import="net.nplus1.javaweb.lab5.Task,java.util.*" %>
<%
    List<Task> tasks = (List<Task>)session.getAttribute("taskList");
    if ( tasks == null ) {
        tasks = new ArrayList<Task>();
        session.setAttribute("taskList", tasks);
    }

%>
<%@ include file="header.jsp" %>
<p>
<form method="post" action="delete_tasks.jsp">
    <table border="1" cellspacing="0" cellpadding="4">
        <tr>
            <td>x</td>
            <td>Pri</td>
            <td>Due</td>
            <td>Task</td>
        </tr>
        <%
            for ( int x = 0; x < tasks.size(); x++ ) {
                Task task = tasks.get(x);
        %>
            <tr>
                <td>
                    <input type="checkbox" name="id" value="<%= x %>"/>
                </td>
                <td><%= task.getPriority() %></td>
                <td><%= task.getDueDate() %></td>
                <td>
                    <a href="edit_task.jsp?id=<%= x %>">
                        <%= task.getTask() %>
                    </a>
                </td>
            </tr>
        <%
            }
        %>
</table>
<input type="submit" value="Delete Selected Tasks"/>
</form>
</p>
<p>
<form method="post" action="add_task.jsp">
    <table border="0" cellspacing="0" cellpadding="4">
        <tr>
```



```
            <td>Priority:</td>
            <td><input type="text" name="priority"/></td>
        </tr>
        <tr>
            <td>Due Date:</td>
            <td><input type="text" name="dueDate"/></td>
        </tr>
        <tr>
            <td>Task:</td>
            <td><input type="text" name="task"/></td>
        </tr>
        <tr>
            <td> </td>
            <td><input type="submit" value="Add Task"/></td>
        </tr>
    </table>
</form>
</p>

<%@ include file="footer.jsp" %>
```


add_task.jsp

```
<%@ page import="net.nplus1.javaweb.lab5.Task,java.util.*" %>
<%
    List<Task> tasks = (List<Task>)session.getAttribute("taskList");

    String priority = request.getParameter("priority");
    String dueDate = request.getParameter("dueDate");
    String task = request.getParameter("task");
    int pri = Integer.parseInt(priority);

    Task newTask = new Task();
    newTask.setDueDate(dueDate);
    newTask.setPriority(pri);
    newTask.setTask(task);

    tasks.add(newTask);

    response.sendRedirect("todo_list.jsp");
%>
```

delete_task.jsp

```
<%@ page import="net.nplus1.javaweb.lab5.Task,java.util.*" %>
<%
    List<Task> tasks = (List<Task>)session.getAttribute("taskList");

    String [] indexes = request.getParameterValues("id");
    for ( int x = indexes.length - 1; x >= 0; x-- ) {
        int index = Integer.parseInt(indexes[x]);
        tasks.remove(index);
    }

    response.sendRedirect("todo_list.jsp");
%>
```


edit_task.jsp

```
<%@ page import="net.nplus1.javaweb.lab5.Task,java.util.*" %>
<%
    List<Task> tasks = (List<Task>)session.getAttribute("taskList");
    String id = request.getParameter("id");
    int index = Integer.parseInt(id);
    Task task = tasks.get(index);
%>
<%@ include file="header.jsp" %>
<p>
<form method="post" action="process_task.jsp">
    <input type="hidden" name="id" value="<%= id %>"/>
    <table border="0" cellspacing="0" cellpadding="4">
        <tr>
            <td>Priority:</td>
            <td><input type="text" name="priority"
                       value="<%= task.getPriority() %>"/></td>
        </tr>
        <tr>
            <td>Due Date:</td>
            <td><input type="text" name="dueDate"
                       value="<%= task.getDueDate() %>"/></td>
        </tr>
        <tr>
            <td>Task:</td>
            <td><input type="text" name="task"
                       value="<%= task.getTask() %>"/></td>
        </tr>
        <tr>
            <td> </td>
            <td><input type="submit" value="Update Task"/></td>
        </tr>
    </table>
</form>
</p>

<%@ include file="footer.jsp" %>
```

process_task.jsp

```
<%@ page import="net.nplus1.javaweb.lab5.Task,java.util.*" %>
<%
    List<Task> tasks = (List<Task>)session.getAttribute("taskList");

    String id = request.getParameter("id");
    int index = Integer.parseInt(id);

    String priority = request.getParameter("priority");
    String dueDate = request.getParameter("dueDate");
    String task = request.getParameter("task");
    int pri = Integer.parseInt(priority);

    Task editTask = tasks.get(index);
    editTask.setDueDate(dueDate);
    editTask.setPriority(pri);
    editTask.setTask(task);

    response.sendRedirect("todo_list.jsp");
%>
```


A.6 Lab 7

Part 1, 4 - Employee.java and employee_list.jsp

```
package net.nplus1.javaweb.lab7;

public class Employee {
    private int id;
    private String firstName;
    private String lastName;
    private String city;
    private String state;
    private double salary;

    public int getId() {
        return id;
    }
    public String getFirstName() {
        return firstName;
    }
    public String getLastName() {
        return lastName;
    }
    public String getCity() {
        return city;
    }
    public String getState() {
        return state;
    }
    public double getSalary() {
        return salary;
    }
    public void setId(int id) {
        this.id = id;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
    public void setCity(String city) {
        this.city = city;
    }
    public void setState(String state) {
        this.state = state;
    }
```



```
    public void setSalary(double salary) {
        this.salary = salary;
    }
}
```


employee_list.jsp

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

<html>
    <body>
        <h2>Employee List</h2>
        <table border="0" cellspacing="0" cellpadding="4">
            <tr>
                <td>Id</td>
                <td>First Name</td>
                <td>Last Name</td>
                <td>City</td>
                <td>State</td>
                <td>Salary</td>
            </tr>
            <c:forEach var="employee" items="${employeeList}">
                <tr>
                    <td>
                        <c:url value="edit_employee.jsp" var="editUrl">
                            <c:param name="id" value="${employee.id}"/>
                        </c:url>
                        <a href="${editUrl}"> ${employee.id} </a>
                    </td>
                    <td>${employee.firstName}</td>
                    <td>${employee.lastName}</td>
                    <td>${employee.city}</td>
                    <td>${employee.state}</td>
                    <td>
                        <fmt:formatNumber value="${employee.salary}"
                                          type="currency"/>
                    </td>
                </tr>
            </c:forEach>
        </table>
        <p>
            <a href="add_employee.jsp">Add New Employee</a>
        </p>
    </body>
</html>
```


Part 2, 6 - employee_widget.jsp , states.tag

employee_widget.jsp

```
<%@ taglib prefix="bs" tagdir="/WEB-INF/tags" %>

<table border="0" cellspacing="0" cellpadding="4">
    <tr>
        <td>Employee ID</td>
        <td><input type="text" name="id" value="${employee.id}"/></td>
    </tr>
    <tr>
        <td>First Name</td>
        <td>
            <input type="text" name="firstName" value="${employee.firstName}"/>
        </td>
    </tr>
    <tr>
        <td>Last Name</td>
        <td>
            <input type="text" name="lastName" value="${employee.lastName}"/>
        </td>
    </tr>
    <tr>
        <td>City</td>
        <td><input type="text" name="city" value="${employee.city}"/></td>
    </tr>
    <tr>
        <td>State</td>
        <td>
            <bs:states name="state" state="${employee.state}"/>
        </td>
    </tr>
    <tr>
        <td>Salary</td>
        <td><input type="text" name="salary" value="${employee.salary}"/></td>
    </tr>
    <tr>
        <td> </td>
        <td align="right"><input type="submit" value="Save Employee"/></td>
    </tr>
</table>
```


states.tag

```
<%@ tag body-content="scriptless"%>

<%@ attribute name="state" required="false" type="java.lang.String" %>
<%@ attribute name="name" required="true" type="java.lang.String" %>

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

<select name="${name}" size="1">
    <option value="IN" ${state == 'IN' ? 'selected="true"' : ''}>IN</option>
    <option value="KY" ${state == 'KY' ? 'selected="true"' : ''}>KY</option>
    <option value="OH" ${state == 'OH' ? 'selected="true"' : ''}>OH</option>
    <option value="TN" ${state == 'TN' ? 'selected="true"' : ''}>TN</option>
</select>
```

Part 3 - add_employee.jsp and process_add.jsp

add_employee.jsp

```
<html>
    <body>
        <h2>Add Employee</h2>
        <form method="post" action="process_add.jsp">
            <%@ include file="employee_widgets.jsp" %>
        </form>
    </body>
</html>
```

process_add.jsp

```
<%@ page import="net.nplus1.javaweb.lab7.Employee,java.util.*" %>
<%
    List<Employee> employees = (List<Employee>)session.getAttribute("employeeList");
    if ( employees == null ) {
        employees = new ArrayList<Employee>();
        session.setAttribute("employeeList", employees);
    }

    String id = request.getParameter("id");
    int employeeId = Integer.parseInt(id);
    String firstName = request.getParameter("firstName");
    String lastName = request.getParameter("lastName");
    String city = request.getParameter("city");
    String state = request.getParameter("state");
    String salary = request.getParameter("salary");
    double employeeSalary = Double.parseDouble(salary);

    Employee emp = new Employee();
    emp.setId(employeeId);
    emp.setFirstName(firstName);
    emp.setLastName(lastName);
    emp.setCity(city);
    emp.setState(state);
    emp.setSalary(employeeSalary);
    employees.add(emp);

    response.sendRedirect("employee_list.jsp");
%>
```


Part 5

edit_employee.jsp

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

<html>
    <body>
        <h2>Edit Employee</h2>
        <c:forEach var="employee" items="${employeeList}">
            <c:if test="${employee.id == param.id}">
                <form method="post" action="process_edit.jsp">
                    <%@ include file="employee_widgets.jsp" %>
                </form>
            </c:if>
        </c:forEach>
    </body>
</html>
```

process_edit.jsp

```
<%@ page import="net.nplus1.javaweb.lab7.Employee,java.util.*" %>
<%
    List<Employee> employees = (List<Employee>)session.getAttribute("employeeList");

    String id = request.getParameter("id");
    int employeeId = Integer.parseInt(id);
    String firstName = request.getParameter("firstName");
    String lastName = request.getParameter("lastName");
    String city = request.getParameter("city");
    String state = request.getParameter("state");
    String salary = request.getParameter("salary");
    double employeeSalary = Double.parseDouble(salary);

    for ( Employee employee : employees ) {
        if ( employee.getId() == employeeId ) {
            employee.setId(employeeId);
            employee.setFirstName(firstName);
            employee.setLastName(lastName);
            employee.setCity(city);
            employee.setState(state);
            employee.setSalary(employeeSalary);
        }
    }

    response.sendRedirect("employee_list.jsp");
%>
```


A.7 Lab 8

FrontControllerServlet.java

```
package net.nplus1.javaweb.lab8;

import java.io.IOException;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class FrontControllerServlet extends HttpServlet {
    private ApplicationController appController;

    public void init() {
       this.appController = new ApplicationController();
    }

    public void doGet(HttpServletRequest request, HttpServletResponse response)
                      throws ServletException, IOException {
        this.process(request, response);
    }

    public void doPost(HttpServletRequest request, HttpServletResponse response)
                       throws ServletException, IOException {
        this.process(request, response);
    }

    private void process(HttpServletRequest request,
                         HttpServletResponse response)
                         throws ServletException, IOException {
        String url = request.getPathInfo();
        String view = appController.processRequest(url, request);

        if ( view == null ) {
            response.sendError(HttpServletResponse.SC_NOT_FOUND);
        }
        else {
            RequestDispatcher dispatcher =
                     this.getServletContext().getRequestDispatcher(view);
            dispatcher.forward(request, response);
        }
    }
}
```


ApplicationController.java

```
package net.nplus1.javaweb.lab8;

import javax.servlet.http.HttpServletRequest;

public class ApplicationController {
    public String processRequest(String url, HttpServletRequest request) {
        String view = null;
        try {
            WebAction action = WebActionFactory.createAction(url);
            if ( action != null )
            {
                String viewName = action.process(request);
                view = LookupJSP.lookup(viewName);
            }
        }
        catch (Exception e) {
            view = "/lab8/error.jsp";
        }

        return view;
    }
}
```

WebAction.java

```
package net.nplus1.javaweb.lab8;

import javax.servlet.http.HttpServletRequest;

public interface WebAction {
    public String process(HttpServletRequest request) throws Exception;
}
```

WebActionFactory.java

```
package net.nplus1.javaweb.lab8;

public class WebActionFactory {
    public static WebAction createAction(String url) {
        WebAction action = null;

        if ( url.equals("/todo.html") ) {
            action = new TodoList();
        }
        else if ( url.equals("/add_task.html") ) {
            action = new AddTask();
        }
        else if ( url.equals("/process_task.html") ) {
            action = new ProcessTask();
        }
        else if ( url.equals("/edit_task.html") ) {
            action = new EditTask();
        }
        else if ( url.equals("/delete_tasks.html") ) {
            action = new DeleteTasks();
        }

        return action;
    }
}
```


LookupJsp.java

```
package net.nplus1.javaweb.lab8;

public class LookupJSP {
    public static String lookup(String view) {
        String jsp = null;

        if ( view.equals("todo list") ) {
            jsp = "/lab8/todo_list.jsp";
        }
        else if ( view.equals("task form") ) {
            jsp = "/lab8/task_form.jsp";
        }

        return jsp;
    }
}
```

TodoList.java

```
package net.nplus1.javaweb.lab8;

import java.util.LinkedList;
import java.util.List;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;

import net.nplus1.javaweb.lab5.Task;

public class TodoList implements WebAction {
    public String process(HttpServletRequest request) throws Exception {
        HttpSession session = request.getSession();
        List<Task> tasks = (List<Task>)session.getAttribute("todoList");

        if ( tasks == null )
        {
            tasks = new LinkedList<Task>();
            session.setAttribute("todoList", tasks);
        }

        return "todo list";
    }
}
```


AddTask.java

```
package net.nplus1.javaweb.lab8;

import javax.servlet.http.HttpServletRequest;

import net.nplus1.javaweb.lab5.Task;

public class AddTask implements WebAction {
    public String process(HttpServletRequest request) throws Exception {
        Task task = new Task();
        request.setAttribute("action", "Add");
        request.setAttribute("task", task);

        return "task form";
    }
}
```

EditTask.java

```
package net.nplus1.javaweb.lab8;

import java.util.List;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;

import net.nplus1.javaweb.lab5.Task;

public class EditTask implements WebAction {
    public String process(HttpServletRequest request) throws Exception {
        String id = request.getParameter("id");
        int index = Integer.parseInt(id);

        HttpSession session = request.getSession();
        List<Task> tasks = (List<Task>)session.getAttribute("todoList");

        Task task = tasks.get(index);
        request.setAttribute("action", "Edit");
        request.setAttribute("task", task);

        return "task form";
    }
}
```


ProcessTask.java

```
package net.nplus1.javaweb.lab8;

import java.util.List;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;

import net.nplus1.javaweb.lab5.Task;

public class ProcessTask implements WebAction {
    public String process(HttpServletRequest request) throws Exception {
        String action = request.getParameter("action");
        HttpSession session = request.getSession();
        List<Task> tasks = (List<Task>)session.getAttribute("todoList");

        Task task = null;

        if ( action.equals("Add") )
        {
            task = new Task();
            tasks.add(task);
        }
        else if ( action.equals("Edit") )
        {
            String id = request.getParameter("id");
            int index = Integer.parseInt(id);
            task = tasks.get(index);
        }

        String priority = request.getParameter("priority");
        task.setPriority(Integer.parseInt(priority));
        task.setDueDate(request.getParameter("dueDate"));
        task.setTask(request.getParameter("task"));

        return "todo list";
    }
}
```

DeleteTasks.java

```
package net.nplus1.javaweb.lab8;

import java.util.List;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;

import net.nplus1.javaweb.lab5.Task;

public class DeleteTasks implements WebAction {
    public String process(HttpServletRequest request) throws Exception {
        HttpSession session = request.getSession();
        List<Task> tasks = (List<Task>)session.getAttribute("todoList");

        String [] indexes = request.getParameterValues("id");
        for ( int x = indexes.length - 1; x >= 0; x-- ) {
            int index = Integer.parseInt(indexes[x]);
            tasks.remove(index);
        }

        return "todo list";
    }
}
```


todo_list.jsp

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
    <body>
        <h2>Todo List</h2>
        <form method="post" action="delete_tasks.html">
        <table border="1" cellspacing="0" cellpadding="4">
            <tr>
                <td>x</td>
                <td>Pri</td>
                <td>Due</td>
                <td>Task</td>
            </tr>
            <c:set var="id" value="0"/>
            <c:forEach var="task" items="${todoList}">
                <tr>
                    <td><input type="checkbox" name="id" value="${id}"/></td>
                    <td>${task.priority}</td>
                    <td>${task.dueDate}</td>
                    <td>
                        <c:url var="editUrl" value="edit_task.html">
                            <c:param name="id" value="${id}"/>
                        </c:url>
                        <a href="${editUrl}">${task.task}</a>
                    </td>
                </tr>
                <c:set var="id" value="${id + 1}"/>
            </c:forEach>
        </table>
        <input type="submit" value="Delete Tasks"/>
        </form>
        <p>
        <a href="add_task.html">Add New Task</a>
        </p>
    </body>
</html>
```


error.jsp

```
<html>
    <body>
        <h2>Error Occurred</h2>
        <p>
        Error occurred trying to access the page.
        </p>
    </body>
</html>
```

task_form.jsp

```
<html>
    <body>
    <h2>${action} Task</h2>
<form method="post" action="process_task.html">
    <input type="hidden" name="action" value="${action}"/>
    <input type="hidden" name="id" value="${param.id}"/>
    <table border="0" cellspacing="0" cellpadding="4">
        <tr>
            <td>Priority:</td>
            <td><input type="text" name="priority" value="${task.priority}"/></td>
        </tr>
        <tr>
            <td>Due Date:</td>
            <td><input type="text" name="dueDate" value="${task.dueDate}"/></td>
        </tr>
        <tr>
            <td>Task:</td>
            <td><input type="text" name="task" value="${task.task}"/></td>
        </tr>
        <tr>
            <td> </td>
            <td><input type="submit" value="Save"/></td>
        </tr>
    </table>
</form>
    </body>
</html>
```


A.8 Lab 9

ApplicationController.java

```
package net.nplus1.javaweb.lab9;

import javax.servlet.http.HttpServletRequest;

public class ApplicationController {
    public String processRequest(String url, HttpServletRequest request) {
        String view = null;
        try {
            // incoming filters here
            RequestContext context = ContextFilter.buildContext(request);
            WebAction action = WebActionFactory.createAction(url);
            if ( action != null )
            {
                String viewName = action.process(context);
                view = LookupJSP.lookup(viewName);
            }

            // outgoing filters here
            ContextFilter.updateRequest(context, request);
        }
        catch (Exception e) {
            view = "/lab8/error.jsp";
        }

        return view;
    }
}
```

RequestContext.java

```
package net.nplus1.javaweb.lab9;

import java.util.HashMap;
import java.util.Set;

public class RequestContext {
    private HashMap<String,Object> parameterMap;
    private HashMap<String,Object> dataMap;
    private HashMap<String,Object> sessionMap;

    public RequestContext() {
        this.parameterMap = new HashMap<String, Object>();
        this.dataMap = new HashMap<String, Object>();
        this.sessionMap = new HashMap<String, Object>();
    }

    public void setParameter(String key, Object value) {
        this.parameterMap.put(key, value);
    }

    public String getParameter(String key) {
        String value = null;
        Object data = this.parameterMap.get(key);
        if ( data != null && data instanceof String) {
            value = (String)data;
        }
        else if ( data != null && data instanceof String [] ) {
            String [] stringData = (String[])data;
            value = stringData[0];
        }

        return value;
    }

    public String [] getParameterValues(String key) {
        String [] values = null;
        Object data = this.parameterMap.get(key);
        if ( data != null && data instanceof String [] ) {
            values = (String [])data;
        }
        else if (data != null ) {
            values = new String[1];
            values[0] = (String)data;
        }
        return values;
    }

    public void setAttribute(String key, Object value) {
```



```
        this.dataMap.put(key, value);
    }

    public Object getAttribute(String key) {
        return this.dataMap.get(key);
    }

    public void setSessionAttribute(String key, Object value) {
        this.sessionMap.put(key, value);
    }

    public Object getSessionAttribute(String key) {
        return this.sessionMap.get(key);
    }

    public Set<String> keySet() {
        return dataMap.keySet();
    }

    public Set<String> sessionKeySet() {
        return sessionMap.keySet();
    }
}
```

ContextFilter

```
package net.nplus1.javaweb.lab9;

import java.util.Enumeration;
import java.util.Set;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;

public class ContextFilter {
    public static RequestContext buildContext(HttpServletRequest request)  {
        RequestContext context = new RequestContext();
        Enumeration<String> list = request.getParameterNames();

        while ( list.hasMoreElements() ) {
            String name = (String)list.nextElement();
            String [] values = request.getParameterValues(name);
            if ( values == null ) {
                context.setParameter(name, null);
            }
            else if (values.length == 1) {
                context.setParameter(name, values[0]);
            }
            else {
                context.setParameter(name, values);
            }
        }

        HttpSession session = request.getSession(true);
        list = session.getAttributeNames();
        while ( list.hasMoreElements() ) {
            String name = (String)list.nextElement();
            context.setSessionAttribute(name, session.getAttribute(name));
        }

        return context;
    }

    public static void updateRequest(RequestContext context,
                                     HttpServletRequest request) {
        Set<String> keySet = context.keySet();
        for ( String key : keySet ) {
            request.setAttribute(key, context.getAttribute(key));
        }

        HttpSession session = request.getSession(true);
        Set<String> sessionKeySet = context.sessionKeySet();
```



```
        for ( String key : sessionKeySet ) {
            session.setAttribute(key, context.getSessionAttribute(key));
        }
    }
}
```

WebActionFilter.java

```
package net.nplus1.javaweb.lab9;

public class WebActionFactory {
    public static WebAction createAction(String url) {
        WebAction action = null;

        if ( url.equals("/todo.html") ) {
            action = new TodoList();
        }
        else if ( url.equals("/add_task.html") ) {
            action = new AddTask();
        }
        else if ( url.equals("/process_task.html") ) {
            action = new ProcessTask();
        }
        else if ( url.equals("/edit_task.html") ) {
            action = new EditTask();
        }
        else if ( url.equals("/delete_tasks.html") ) {
            action = new DeleteTasks();
        }

        return action;
    }
}
```

TodoList.java

```
package net.nplus1.javaweb.lab9;

import java.util.LinkedList;
import java.util.List;

import net.nplus1.javaweb.lab5.Task;

public class TodoList implements WebAction {
    public String process(RequestContext context) throws Exception {
        List<Task> tasks = (List<Task>)context.getSessionAttribute("todoList");

        if ( tasks == null )
        {
            tasks = new LinkedList<Task>();
            context.setSessionAttribute("todoList", tasks);
        }

        return "todo list";
    }
}
```

AddTask.java

```
package net.nplus1.javaweb.lab9;

import net.nplus1.javaweb.lab5.Task;
import net.nplus1.javaweb.lab9.WebAction;

public class AddTask implements WebAction {
    public String process(RequestContext context) throws Exception {
        Task task = new Task();
        context.setAttribute("action", "Add");
        context.setAttribute("task", task);

        return "task form";
    }
}
```

EditTask.java

```
package net.nplus1.javaweb.lab9;

import java.util.List;

import net.nplus1.javaweb.lab5.Task;

public class EditTask implements WebAction {
    public String process(RequestContext context) throws Exception {
        String id = context.getParameter("id");
        int index = Integer.parseInt(id);

        List<Task> tasks = (List<Task>)context.getSessionAttribute("todoList");

        Task task = tasks.get(index);
        context.setAttribute("action", "Edit");
        context.setAttribute("task", task);

        return "task form";
    }
}
```

DeleteTasks.java

```
package net.nplus1.javaweb.lab9;

import java.util.List;

import net.nplus1.javaweb.lab5.Task;

public class DeleteTasks implements WebAction {
    public String process(RequestContext context) throws Exception {
        List<Task> tasks = (List<Task>)context.getSessionAttribute("todoList");

        String [] indexes = context.getParameterValues("id");
        for ( int x = indexes.length - 1; x >= 0; x-- ) {
            int index = Integer.parseInt(indexes[x]);
            tasks.remove(index);
        }

        return "todo list";
    }
}
```


ProcessTask.java

```
package net.nplus1.javaweb.lab9;

import java.util.List;

import net.nplus1.javaweb.lab5.Task;

public class ProcessTask implements WebAction {
    public String process(RequestContext context) throws Exception {
        String action = context.getParameter("action");
        List<Task> tasks = (List<Task>)context.getSessionAttribute("todoList");

        Task task = null;

        if ( action.equals("Add") )
        {
            task = new Task();
            tasks.add(task);
        }
        else if ( action.equals("Edit") )
        {
            String id = context.getParameter("id");
            int index = Integer.parseInt(id);
            task = tasks.get(index);
        }

        String priority = context.getParameter("priority");
        task.setPriority(Integer.parseInt(priority));
        task.setDueDate(context.getParameter("dueDate"));
        task.setTask(context.getParameter("task"));

        return "todo list";
    }
}
```

B GNU Free Documentation License

Version 1.3, 3 November 2008

<http://fsf.org/>

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "**Document**", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "**you**". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "**Modified Version**" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "**Secondary Section**" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall

subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "**Invariant Sections**" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "**Cover Texts**" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "**Transparent**" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "**Opaque**".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "**Title Page**" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "**publisher**" means any person or entity that distributes copies of the Document to the public.

A section "**Entitled XYZ**" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "**Acknowledgements**", "**Dedications**", "**Endorsements**", or "**History**".) To "**Preserve the Title**" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty

Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with

the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with . . . Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Index

