ID2212, Network Programming with Java
Lecture 11

JavaServer Faces (JSF)
JavaServer Pages (JSP)

Leif Lindbäck
KTH/ICT/SCS
HT 2015

Content

- Overview of JSF and JSP
- JSF Introduction
- JSF tags
- Managed Beans
- Expression language
- JSP Standard Tag Library (JSTL)
- JSF Navigation and validation
- JSP Overview

Design of a Java EE application

- This is covered in the exercise

Two Different View Technologies

JavaServer Faces, JSF

- Newer
- Dynamic views
- Handles non-functional requirements like navigation, validation, composite views and view templates.
- XHTML pages with JSF-specific tags that are converted to XHTML tags.
- Handled by the JSF framework that run inside the Servlet container.

Two Different View Technologies, Cont'd

- JavaServer Pages, JSP
  - Older, left mainly for backwards compatibility
  - Dynamic views
  - Does not handle non-functional requirements.
  - JSP pages with JSP-specific tags. The pages are translated into Servlets.
  - Handled by the Servlet container itself.

Why Use JSF Instead of Plain JSP

- Avoid writing infrastructure code for non-functional requirements like navigation, validation, composite views and view templates.
- Thoroughly tested and proven to work well.
- Lots of documentation, easy to get help.
- Not using a framework means writing new code which means introducing new bugs.

Why Use JSF Instead of Plain JSP? Cont'd

- Non-functional requirements are difficult to code.
- Callback style makes sure all calls to non-functional requirements code are made at the right time.
  - Handled by the framework.

JavaServer Faces, JSF

`javax.faces`

JSF Home page:
http://www.oracle.com/technetwork/java/javaee/overview/index.html

JSF tag library documentation:
http://docs.oracle.com/javaee/7/javaserverfaces/2.2/vdldocs/facelets/

A Simple Example

- The example has two views.

A Simple Example, Cont'd

- The first JSF page, index.xhtml.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">
    <h:head>
        <title>Welcome</title>
    </h:head>
    <h:body>
        <h3>Please enter your name</h3>
        <h:form>
            <p>Name: <h:inputText value="#{user.name}"/></p>
            <p><h:commandButton value="Enter" action="welcome"/></p>
        </h:form>
    </h:body>
</html>
```

A Simple Example, Cont'd

- The second JSF page, welcome.xhtml.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">
    <h:head>
        <title>Welcome</title>
    </h:head>
    <h:body>
        <h3>Welcome here, #{user.name}!</h3>
    </h:body>
</html>
```

A Simple Example, Cont'd

- The managed bean, User.java.

```
package lec11;

import java.io.Serializable;
import javax.inject.Named;
import javax.enterprise.context.SessionScoped;

@Named("user")
@SessionScoped
public class User implements Serializable {
    private static final long serialVersionUID = 0xE9085B8280336BE4L;

    private String name;

    public String getName() {
        return name;
    }

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

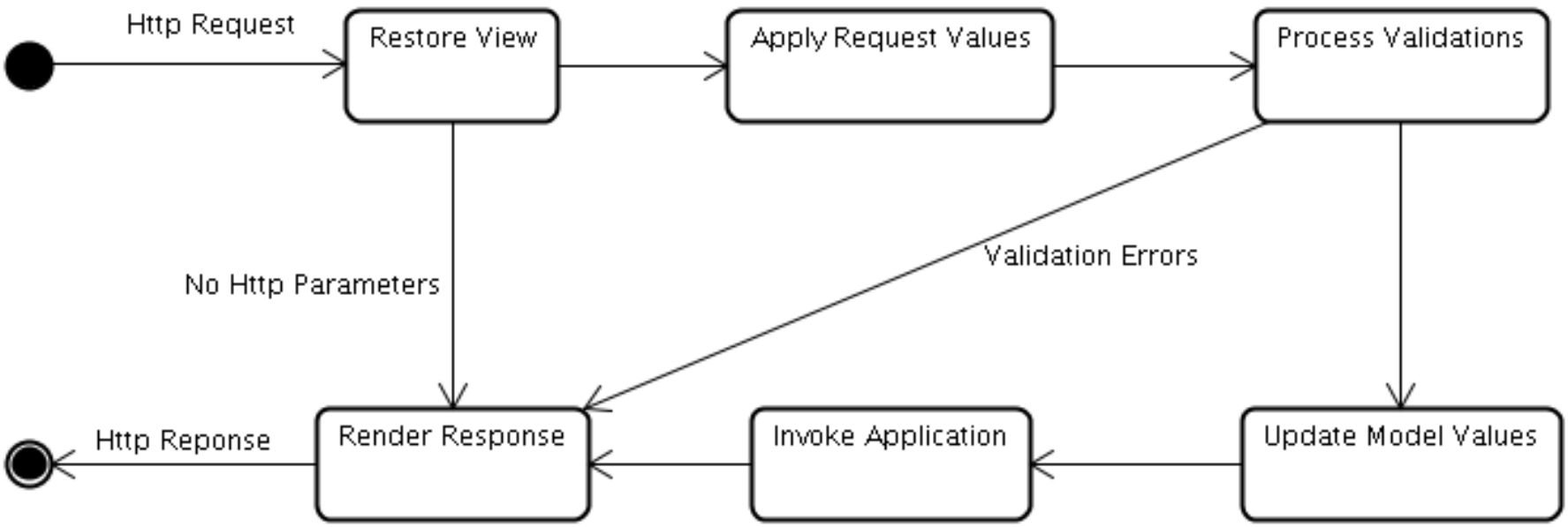
Overview of JSF Architecture

- JSF has a component based architecture
  - Treats view parts as UI components, not as HTML elements.
  - Maintains an internal *component tree*.
  - Think of it as a Swing or AWT UI.
  - `index.xhtml` in the initial example has three components. The first, a form, is the ancestor of the other two, a button and a text field.

Overview of JSF Architecture, Cont'd

- Each tag in a page has an internal associated tag handler class inside JSF.
  - The tag handler classes are organized according to the component tree.
- The internal JSF classes handles translation of JSF tags to HTML tags, interpretation of Http requests, calls to managed beans etc.

The Phases of a JSF Request



The Phases of a JSF Request

- *Restore View* Phase
  - Retrieves the component tree (i.e. tree of internal tag handler classes) for the page if it was displayed previously. It the page is displayed the first time the component tree is instead created.
  - If there are no Http parameters in the request JSF skips directly to the *Render Response* phase.

The Phases of a JSF Request, Cont'd

- *Apply Request Values* Phase
  - The Http request parameters are placed in a hash table that is passed to all objects in the component tree.
  - Each object identifies the parameters belonging to the component it represents and stores those parameter values.
  - Values stored in objects in the component tree are called *local values*.

The Phases of a JSF Request, Cont'd

- *Process Validations* Phase
  - It is possible to attach validators to user editable components (typically text fields) in a JSF page, using JSF tags.
  - Example of validators are that a field is not empty, that a parameter is an integer, that it is a string of a certain length etc.
  - In this phase, the validators are executed to check that the local values are correct.
  - If some validation fails JSF skips to the *Render Response* phase and redisplays the current page with error messages about the failed validations.

The Phases of a JSF Request, Cont'd

- *Update Model Values* Phase
  - The local values are used to update managed beans by invoking setter methods.
  - Managed beans and their properties are identified by their names, in the `index.html` page in the initial example the user enters their name in a text field that has the value `user.name`. This means the name is sent to the method `setName` in the managed bean that is named `user`.

The Phases of a JSF Request, Cont'd

- *Invoke Application* Phase
  - Here the method specified by the `action` attribute of the component that caused the Http request is called.

The Phases of a JSF Request, Cont'd

- *Render Response* Phase
  - Here the next view is created.
  - Everything in the XHTML page except JSF tags is unchanged.
  - JSF tags are transformed to XHTML tags by the objects in the component tree.
  - Getter methods in managed beans are called in order to retrieve values. In the `welcome.xhtml` page in the initial example the value `user.name` is retrieved by a call to the method `getName` in the managed bean that is named `user`.

Tag Libraries in JSF

- JSF tag library documentation is found at `http://docs.oracle.com/javaee/7/javaserverfaces/2.2/vdldocs/facelets/`
- HTML
  - Used to create HTML elements.
  - The recommended prefix is *h:*
  - Some important tags are covered below.

Tag Libraries in JSF

- Core
  - Used to add objects , such as validators, listeners and AJAX support, to HTML elements.
  - The recommended prefix is *f:*
  - Example in the slides explaining validation.

Tag Libraries in JSF, Cont'd

- Facelets
  - Used to create composite views, e.g. views that have common components like header, footer and menu, without using duplicated code.
  - The recommended prefix is *ui:*
  - Not covered in this course.

Tag Libraries in JSF, Cont'd

- Composite Components
  - Used to create custom components.
  - The recommended prefix is *composite:*
  - Not covered in this course.

Tag Libraries in JSF, Cont'd

- JSTL (JSP Standard Tag Library) Core
  - Utility tags managing for example flow control.
  - The recommended prefix is *c:*
  - Some important tags are covered below.
- JSTL (JSP Standard Tag Library) Functions
  - Utility functions mainly for handling strings.
  - The recommended prefix is fn*:*
  - Some example tags are covered below.

Tag Library Declaration

- Tag libraries must be declared in the XHTML page where they are used.
- This is done in the `<HTML>` tag.
- The `index.xhtml` in the initial example uses the HTML tag library. It is declared as follows.

```
<html xmlns="http://www.w3.org/1999/xhtml"
        xmlns:h="http://java.sun.com/jsf/html">
```

Some Tags in the HTML Tag Library

- `head`, renders the head of the page.
- `body`, renders the body of the page.
- `form`, renders an HTML form.
- `inputText`, renders an HTML text field.
- `inputSecret`, renders an HTML password field.
- `outputLabel`, renders a plain text label for another component.
- `outputText`, renders plain text.
- `commandButton`, renders a submit button.

Attributes for The HTML Tags

- All tags mentioned on the preceding page, except `head` and `body`, have the following attributes.
  - `id`, gives a unique name to the component. All components have a unique name. It is assigned by JSF if not stated explicitly with the `id` tag.
  - `value`, specifies the component's currently displayed value. This can be an expression that refers to a property in a managed bean. If so, the value will be read from the bean when the component is displayed and stored to the bean when the component is submitted.
  - `rendered`, a boolean expression that tells whether the component is displayed or not.

Attributes for The HTML Tags, Cont'd

- The `outputLabel` tag also has the `for attribute`.
  - Specifies for which other component this component is a label. The label is normally displayed immediately to the left of that other component.

Attributes for The HTML Tags, Cont'd

- The `commandButton` tag also has the `action` attribute.
  - Tells what to do when the user clicks the button.
  - Can be the name of a XHTML page, without the `.xhtml` extension. In this case the specified page is displayed.
  - Can also be the name of a method in a managed bean, in this case that method is invoked.

Attributes for The HTML Tags, Cont'd

- The `commandButton` tag also has the `action` attribute.
  - Tells what to do when the user clicks the button.
  - Can be the name of a XHTML page, without the `.xhtml` extension. In this case the specified page is displayed.
  - Can also be the name of a method in a managed bean, in this case that method is invoked.

Plain HTML Tags Instead of HTML Tag Library

- It Is allowed to use plain html tags instead of the HTML tag library and the **`h:`** prefix.
- In this case, tags that shall be managed by JSF must have attributes in the **`http://xmlns.jcp.org/jsf`** namespace.
- The following slide illustrates this for the HTML5 **`datalist`** tag, which has no corresponding tag in the JSF HTML tag library.

Plain HTML Tags Instead of HTML Tag Library

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:jsf="http://xmlns.jcp.org/jsf">
    <body>
        <form>
            <input list="browsers"/>
            <datalist id="browsers">
                <option value="Internet Explorer"/>
                <option jsf:id="abc" value="#{data.browser}"/>
            </datalist>
          </form>
       </body>
  </html>
```

JSTL (JSP Standard Tag Library)

Core Tags

- `choose`, an if statement.

```
<c:choose>
  <c:when test="#{condition}">
    The condition was true.
  </c:when>
  <c:otherwise>
    The condition was false.
  </c:otherwise>
</c:choose>
```

- If the boolean condition specified in the `test` attribute is true, the `when` block is executed, if not the `otherwise` block is executed.

JSTL (JSP Standard Tag Library) Core Tags, Cont'd

- **`forEach`**, a loop statement.

```
<c:forEach var="element" items="#{myList}"
           varStatus="status" >
  Element number #{status.count} is #{element}
</c:forEach>
```

- The `var` attribute specifies the name of the variable holding the current element's value. This variable is used when the value shall be displayed.
- The `items` attribute refers to the collection that shall be iterated over.
- The `varStatus` attribute defines a variable that holds information like the current element's index in the collection.

Functions In the JSTL (JSP Standard Tag Library) Functions Library

- Note that these are functions, not tags.
- `contains(str, substr)`, returns `true` if `str` contains `substr`.
- `startsWith(str, substr`, returns `true` if `str` starts with `substr`.
- `length(str)`, returns the length of `str`.
- And many more.

JSTL (JSP Standard Tag Library), Cont'd

- Example:

```
<c:choose>
    <c:when test="#{fn:containsIgnoreCase(user.name, 'Leif')}">
        <h3>Sorry, you are banned!</h3>
    </c:when>
    <c:otherwise>
        <h3>Welcome here, #{user.name}!</h3>
    </c:otherwise>
</c:choose>
```

Managed Beans

- Managed beans are plain Java classes.
  - Must have a public no-arg constructor.
  - Must have a scope annotation, e.g. `@SessionScoped`, see next slide for more examples.
  - Annotated `@Named("myName")`, where `myName` becomes the name of the bean.
- The beans are managed by the *CDI* (*Context and Dependency Injection*) container.
  - Part of Java EE
  - Creates and connects objects according to specifications in annotations.
  - Powerful framework, but not covered in this course.

Managed Beans, Cont'd

- All managed beans have a scope which defines their life time. Some scope annotations are:
  - `ApplicationScoped`, the object will exist for the entire application life time.
  - `SessionScoped`, the object will be discarded when the current Http session ends.
  - `ConversationScoped`, a conversation can be started and stopped manually in the code. If it is not, it has the life time of a Http request. Unlike sessions, conversations are unique for each browser tab and therefore thread safe.
  - `RequestScoped`, the object will be discarded when the current Http request is handled.

Managed Beans, Example

```java
package lec11;

import java.io.Serializable;
import javax.inject.Named;
import javax.enterprise.context.SessionScoped;

@Named("user")
@SessionScoped
public class User implements Serializable {
    private static final long serialVersionUID = 0xE9085B8280336BE4L;

    private String name;

    public String getName() {
        return name;
    }

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

Expression Language

- The *expression language* is used in dynamic expressions in JSF (and JSP) pages.
  - Stateless, variables can not be declared.
  - Statements are written between #{ and }
  - The result of an EL statement is a string.

Expression Language, Cont'd

- The syntax is `#{something.somethingElse}`, where `something` is for example one of the following:
  - The name of a managed bean.
  - `param`, which is a `java.util.Map` containing all HTTP request parameters. If there are more parameters with the same name the first is returned.
  - `paramValues`, which is a `java.util.Map` containing all HTTP request parameters. No matter how many parameters there are with the specified name a `java.util.List` with all of them is returned.

Expression Language, Cont'd

- `header`, which is a `java.util.Map` containing all HTTP headers. If there are more headers with the same name the first is returned.
- `headerValues`, which is a `java.util.Map` containing all HTTP headers. No matter how many headers there are with the specified name a `java.util.List` with all of them is returned.
- `cookie`, which is a `java.util.Map` containing all HTTP cookies.

EL, The Operators . and []

- If the operator `.` is used `(#{something.somethingElse})` the following must be true.
  - `something` is a `java.util.Map` or a managed bean.
  - `somethingElse` is a key in a `java.util.Map` or a property in a managed bean or a method in a managed bean.

EL, The Operators . and [], Cont'd

- If the operator `[]` is used (`#{something["somethingElse"]}`) the following must be true.
  - `something` is a `java.util.Map`, a managed bean, an array or a `java.util.List`.
  - If `somethingElse` is a string (surrounded by double quotes, "") it must be a key in a `java.util.Map`, a property in a managed bean, an index to an array or an index to a `java.util.List`.
  - If `somethingElse` is not surrounded by double quotes it must be a valid EL statement.

EL Examples

- If these are managed beans:

```
@Named(“person”)
public class PersonBean {
    @Inject private DogBean dog;

    public DogBean getDog() {
        return dog;
    }
}
```

```
@Named(“dog”)
public class DogBean {
    private String name;

    public String getName() {
        return name;
    }
}
```

- Then it is allowed to write `#{person.dog.name}` or `#{person[dog["name"]]}`.

EL Examples, Cont'd

- Input from an HTML form:

```
<form>
  Address: <input type="text" name="address">
  Phone1: <input type="text" name="phone">
  Phone2: <input type="text" name="phone">
</form>
```

- Can be read like this:

```
The address is #{param.address}
Phone1 is #{param.phone}
Phone1 is #{paramValues.phone[0]}
Phone2 is #{paramValues.phone[1]}
```

- However, there is seldom any need for this since request parameters are normally handled by managed beans.

The EL Operators

- Remember that JSF/JSP pages are views and thus not the place for a lot of calculations.
- Arithmetic
  - *addition:* `+`
  - *subtraction:* `-`
  - *multiplication:* `*`
  - *division:* `/` *or* `div`
  - *remainder:* `%` *or* `mod`
- Logical
  - and: `&&` *or* `and`
  - or: `||` *or* `or`
  - not: `!` *or* `not`

The EL Operators, Cont'd

- Relational
  - *equals:* == *or* `eq`
  - *not equals:* != *or* `ne`
  - *less than:* < *or* `lt`
  - *greater than:* > *or* `gt`
  - *less than or equal to:* <= *or* `le`
  - *greater than or equal to:* >= *or* `ge`

EL, Null Values

- Since EL is used for user interfaces it produces the most user friendly output.
- This means that (like HTML) it tries to silently ignore errors.
- Null values does not generate any output at all, no error messages are produced.

Navigation

- What calls should be made to the model and which is the next view, provided the user has clicked YYY in view ZZZ.
- The next view may differ depending on the outcome of the call to the model.
- Answers to the above should be stated as a set of navigation rules that are easy to change.
  - The value of the action attribute of the button or link the user invoked is called the outcome. The navigation handling depends on the outcome.

Static Navigation

- If the outcome is the name of a XHTML page then that page is displayed.
  - This is called *static navigation*. The outcome is always the same.

Dynamic Navigation

- A user action can often have different outcomes, for example a login attempt might succeed or fail.
- In this case *dynamic navigation* must be used.

Dynamic Navigation, Cont'd

- Using dynamic navigation the value of the action attribute must be an expression identifying a method, for example `#{loginManager.validateUser}`, assuming that there is a managed bean named `loginManager` that has a method called `validateUser`.
- The outcome will be the value that is returned by this method. If the return value is not a `String` it will be converted to a `String` by calling its `toString` method.
- The outcome could be the name of a XHTML page, just like with static navigation. If so this page will be displayed.

Dynamic Navigation, Cont'd

- It is not a good design that methods in the model knows names of XHTML files.
- Therefore we want to have the action handling method return a logical view name that is mapped to a XHTML file name.

Dynamic Navigation, Cont'd

- This is achieved by adding a navigation rule to the `faces-config.xml` file

```
<navigation-rule>
    <from-view-id>/login.xhtml</from-view-id>
    <navigation-case>
        <from-outcome>success</from-outcome>
        <to-view-id>/welcome.xhtml</to-view-id>
    </navigation-case>
    <navigation-case>
        <from-outcome>failure</from-outcome>
        <to-view-id>/login.xhtml</to-view-id>
    </navigation-case>
</navigation-rule>
```

- The above means that if an action handling method specified on the `login.xhtml` page returns `success` the `welcome.xhtml` page is displayed next. If on the other hand the method returns `failure` the `login.xhtml` page is displayed again.

Dynamic Navigation, Cont'd

- Even though the action handling method now returns a logical outcome, one could argue that we still have some amount of mixture of business logic and view handling.
- Consider for example a method `withdraw` in a bank application. Such a method would normally be `void`, but would now instead have to return the `String success` only to indicate to JSF that the withdrawal was successful.

Dynamic Navigation, Cont'd

- To avoid this problem we can let the `withdraw` method remain `void`, and instead add another method, `success`, that returns `true` only if the last transaction was successful. `faces-config.xml` would then look as follows.

```
<navigation-rule>
    <from-view-id>/withdraw.xhtml</from-view-id>
    <navigation-case>
        <if>#{bankManager.success}</if>
        <to-view-id>/success.xhtml</to-view-id>
    </navigation-case>
</navigation-rule>
```

No Matching Navigation Case

- If there is an outcome that does not correspond to a XHTML file and that has no matching navigation case, the last page is displayed again.

Validation

- Check data entered by the user.
- If validation fails the same view is shown again together with an error message.
- Which validations are to be made on what and which error messages to show if they fail is specified by attaching validators to user input components.

Validation, Cont'd

- Validation concerns only checks that can be done without understanding the meaning of the input. The check should not include business logic.
  - For example that a field is not empty, that it contains an integer or that it is an email address.
- Since some validation checks, like those mentioned above, occur frequently they are predefined in JSF.

Validation Example

```
<h:inputText id="name" label="Name"
value="#{user.name}">
    <f:validateRequired/>
</h:inputText>
<h:message for="name"/>
```

- The `validateRequired` tag checks that the text field is not empty.
- The `message` tag displays the error message if the validation failed.
- It is possible to customize the error message, but that is outside this course.

Composite Views

- Views often consist of several parts like header, footer, navigation menus, main content etc.
- Many of these parts are common for different views.
- In order to avoid duplicated code it must be possible to reuse both page fragments (html) and page layout (html tables or css).
- Handled by the facelets tag library, but not covered in this course.

Internationalization (i18n) and localization (l10n)

- Internationalization means to make it possible to switch language. To add the possibility to show the user interface in a new language should only require to write the words in the new language, not any additional coding.
- Localization means to add support for a new language.
- Handled by the JSF core tag library, but not covered in this course.

JSP: JavaServer Pages

`javax.servlet.jsp`

JSP Home page:
http://www.oracle.com/technetwork/java/javaee/tech/index.html

What Is JSP?

- Framework used before JSF.
- A JSP page is written in HTML and translated to a Servlet by the Servlet container.
- Dynamically-generated web content.
- Does not handle non-functional requirements like navigation and validation.

The Life Cycle of a JSP

- At the first call:
  - The container translates the JSP to a servlet (translation time)
  - The container compiles the servlet (compile time)
  - The container instantiates the servlet the same way it instantiates any servlet.
  - The container calls `jspInit()`.

The Life Cycle of a JSP, Cont'd

- At all calls:
  - The container calls `_jspService()` of the servlet that was generated at the first call (request time).
- If the JSP is unloaded from the container:
  - The container calls `jspDestroy()`.

A Translated JSP

- The JSP:

```
<html>
   <head>
      <title>Hello World!</title>
   </head>

   <body>
      <h1>Hello World!</h1>
   </body>
</html>
```

A Translated JSP, Cont'd

- The generated servlet (Tomcat 5.5.15):

```
package org.apache.jsp;

import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;

public final class hw_jsp extends org.apache.jasper.runtime.HttpJspBase
    implements org.apache.jasper.runtime.JspSourceDependent {

  private static java.util.List _jspx_dependants;

  public Object getDependants() {
    return _jspx_dependants;
  }

  public void _jspService(HttpServletRequest request,
                HttpServletResponse response)
                throws java.io.IOException, ServletException {
```

- The JSP's response is generated in `_jspService()`

A Translated JSP, Cont'd

- The generated servlet (cont):

```
JspFactory _jspxFactory = null;
   PageContext pageContext = null;
   HttpSession session = null;
   ServletContext application = null;
   ServletConfig config = null;
   JspWriter out = null;
   Object page = this;
   JspWriter _jspx_out = null;
   PageContext _jspx_page_context = null;


   try {
     _jspxFactory = JspFactory.getDefaultFactory();
     response.setContentType("text/html");
     pageContext = _jspxFactory.getPageContext(this, request, response,
                       null, true, 8192, true);
     _jspx_page_context = pageContext;
     application = pageContext.getServletContext();
     config = pageContext.getServletConfig();
     session = pageContext.getSession();
     out = pageContext.getOut();
     _jspx_out = out;
```

A Translated JSP, Cont'd

- The generated servlet (cont):

```
      out.write("<html>\n");
      out.write("   <head>\n");
      out.write("      <title>Hello World!</title>\n");
      out.write("   </head>\n");
      out.write("\n");
      out.write("   <body>\n");
      out.write("      <h1>Hello World!</h1>\n");
      out.write("   </body>\n");
      out.write("</html>\n");
      out.write("\n");
    } catch (Throwable t) {
      if (!(t instanceof SkipPageException)){
        out = _jspx_out;
        if (out != null && out.getBufferSize() != 0)
          out.clearBuffer();
        if (_jspx_page_context != null) _jspx_page_context.handlePageException(t);
      }
    } finally {
      if (_jspxFactory != null) _jspxFactory.releasePageContext(_jspx_page_context);
    }
  }
}
```

- The response is sent.

Where Can I Find the Translated JSPs?

- Like the translation itself, that is server dependant. Tomcat (5.5.15) places them in `TOMCAT_HOME/work/Catalina/localhost/jsp/org/apache/jsp` in a file called `<name of the jsp>_jsp.java`, that is `xyz_jsp.java` if the jsp is called `xyz.jsp`.

Actions and Directives

- A directive is an instruction to the container about the translation of a JSP.
  - Does not exist in the translated Java code.
  - There are three directives: *page*, *taglib* and *include*.
  - Written between `<%@` and `%>` (for example `<%@page ... %>`).

Actions and Directives, Cont'd

- An action is translated into Java code and executed at request time.
  - Syntax: `<prefix:action name/>`
  - Standard actions are defined in the specification and have the prefix `jsp`.
  - Custom tags are defined by the developer and may have any prefix (except reserved prefixes like `jsp`).

To Include Other Files

- The `include` directive:
  - `<%@include file="header.jsp"%>`
  - The directive is replaced with the content of the specified file (`header.jsp`) at translation time.
  - Both static (for example HTML files) and dynamic content (for example other JSP files) can be included.
  - The path to the included file is specified relative to the file with the `include` directive.

To Include Other Files, Cont'd

- The `jsp:include` standard action:
  - Syntax: `<jsp:include page="header.jsp"/>`
  - The included page is translated to a servlet that is called (that is, its `_jspService()` method is called) by the including servlet at request time.
  - Only JSPs can be included.
  - The output of the included Servlet is inserted in the output of the including Servlet.
  - The path to the included page is a URL. It is either relative to the URL of the including page or absolute starting with the context root of the web application.

To Include Other Files, Cont'd

- It is possible to pass parameters to the included page if the `jsp:include` action is used.
  - The following code should be placed in the including page:

```
<jsp:include page="header.jsp">
  <jsp:param name=subTitle" value="A dynamic subtitle"/>
</jsp:include>
```

  - The parameter `subTitle` will be available as an HTTP request parameter in the included page.
  - It can be output like this:

```
<h3>${param.subTitle}</h3>
```

Error handling

- It is possible to define error pages. If an exception occurs in a JSP (or servlet) the container forwards the call to the error page.
  - Error pages are defined like this in the deployment descriptor:

```
<!-- An error page for a Java exception. The call is
forwarded to the error page if the specified exception or a
subclass of it is thrown. -->
<error-page>
  <exception-type>java.lang.Throwable</exception-type>
  <location>/errorpage.jsp</location>
</error-page>

<!-- An error page for an HTTP error -->
<error-page>
  <error-code>404</error-code>
  <location>/errorpage.jsp</location>
</error-page>
```

Error handling, Cont'd

- An example of an error page:

```
<%@ page isErrorPage="true" %>
<html>
  <head>
    <title>This page handles exceptions</title>
  </head>

  <body>
    <h1>This page handles exceptions</h1>
    <p>An ${pageContext.exception} was thrown. Its message was:
       ${pageContext.exception.message}</p>
  </body>
</html>
```

- This must always be written in an error page.
- This is the exception object that was thrown.

NEVER EVER Write Java code in a JSP

- There are ways to insert Java code directly in a JSP.
  - Possible only for backwards compatibility
- NEVER EVER do that!
  - Gives high coupling and low cohesion.
  - Makes the code inflexible, difficult to understand and hard to maintain.
  - Forces web page designers to learn Java programming.
- Use EL and custom tags instead.

The <%@taglib%> directive

- Used to declare custom tags that are used in a JSP
- Syntax:

```
<%@ taglib prefix="myTags" uri="uri/of/my/tld" %>
```

  - The prefix attribute specifies the prefix part of the tag (`<myTags:someTag/>`).
  - The `uri` attribute tells the name of a TLD describing the tags.

To Write Your Own Tags

- There are many useful tags in JSTL.
- There are many third party taglibs, for example *Jakarta Tagblibs*.
- Sometimes we still have to write new tags, there are two types:
  - *Tag files* look like ordinary JSPs but are called like custom tags.
  - *Tag handlers* are written in Java code.
  - Only the latter are covered here.

A Simple Tag

- The Java tag handler:

```
package se.kth.timetags;

import java.io.IOException;
import java.util.Date;
import javax.servlet.jsp.JspWriter;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.tagext.SimpleTagSupport;

/**
 * A tag that displays the current date and time.
 */
public class DateTimeTag extends SimpleTagSupport {

    public void doTag() throws JspException, IOException {
        JspWriter out = getJspContext().getOut();
        out.print(new Date());
    }
}
```

  - The tag handler class must extend `javax.servlet.jsp.tagext.SimpleTagSupport`

A Simple Tag, Cont'd

- The Java tag handler:

```
package se.kth.timetags;

import java.io.IOException;
import java.util.Date;
import javax.servlet.jsp.JspWriter;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.tagext.SimpleTagSupport;

/**
 * A tag that displays the current date and time.
 */
public class DateTimeTag extends SimpleTagSupport {

    public void doTag() throws JspException, IOException {
        JspWriter out = getJspContext().getOut();
        out.print(new Date());
    }
}
```

  - The output of `public void doTag()` will be inserted in the JSP's response.

A Simple Tag, Cont'd

- The tag library descriptor:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<taglib 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
jsptaglibrary_2_0.xsd" version="2.0"
>
  <tlib-version>1.0</tlib-version>
  <uri>timeTags</uri>
  <tag>
    <name>date-time</name>
    <tag-class>course6b4056.timetags.DateTimeTag</tag-class>
    <body-content>empty</body-content>
  </tag>
</taglib>
```

  - Simply copy this part.
  - The version of the tag. Any value can be used.

A Simple Tag, Cont'd

- The tag library descriptor:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<taglib 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
jsptaglibrary_2_0.xsd" version="2.0"
>
  <tlib-version>1.0</tlib-version>
  <uri>timeTags</uri>
  <tag>
    <name>date-time</name>
    <tag-class>course6b4056.timetags.DateTimeTag</tag-class>
    <body-content>empty</body-content>
  </tag>
</taglib>
```

  - The name of the taglib. Must be the same as the uri in the taglib directive in the JSP. The uri is only a string, it is not interpreted in any way.

A Simple Tag, Cont'd

- The tag library descriptor:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<taglib 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
jsptaglibrary_2_0.xsd" version="2.0"
>
  <tlib-version>1.0</tlib-version>
  <uri>timeTags</uri>
  <tag>
    <name>date-time</name>
    <tag-class>course6b4056.timetags.DateTimeTag</tag-class>
    <body-content>empty</body-content>
  </tag>
</taglib>
```

  - The name of the tag.

A Simple Tag, Cont'd

- The tag library descriptor:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<taglib 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
jsptaglibrary_2_0.xsd" version="2.0"
>
  <tlib-version>1.0</tlib-version>
  <uri>timeTags</uri>
  <tag>
    <name>date-time</name>
    <tag-class>course6b4056.timetags.DateTimeTag</tag-class>
    <body-content>empty</body-content>
  </tag>
</taglib>
```

  - The name of the Java tag handler.
  - The allowed content of the tags body.

A Simple Tag, Cont'd

- Possible values of `body-content` in the tld:
  - *empty* means the tag must not have a body.
  - *tagdependent* means the body content is not interpreted by the JSP. Its content is handled as plain text.
  - *scriptless* means the body may contain EL but not Java code. EL statements are interpreted.

A Simple Tag, Cont'd

- The JSP:

```
<%@ taglib uri="timeTags" prefix="time" %>
<html>
  <head>
    <title>Clock</title>
  </head>
  <body>
    <h1>Clock</h1>
    <h3><time:date-time/>
  </body>
</html>
```

  - The name of the tld. Must be the same as in `uri` in the TLD. The uri is not interpreted in any way.
  - The prefix in the JSP.
  - The name of the tag. Must be the same as in `name` in the TLD.

A Simple Tag, Cont'd

- The compiled Java class should be placed under `WEB-INF/classes` in a directory matching the package name. The example class is in the package `se.kth.timetags` and should thus be placed in the directory `WEB-INF/classes/se/kth/timetags`
- The TLD should be placed in `WEB-INF`.

Where do the container look for tag libraries?

- A tag library and its tag handlers are defined by the TLD.
- To execute a tag the container must find a TLD with the same uri as in the taglib directive in the JSP with the tag.

Where do the container look for tag libraries? Cont'd

- The container looks for TLDs in these places:
  - `WEB-INF`
  - Directories under `WEB-INF`
  - `META-INF` in jar files in `WEB-INF/lib`
  - Directories under `META-INF` in jar files in `WEB-INF/lib`