Internet Application Project: Java Server Side

420-625-DW

More JSF:

Java Server Facelets, File Uploaders, and Custom Tags

Today

-JSF tags

-Image Uploaders

-Custom tags

Coming up

-Authentication

-Unit testing (Arquillian and Selenium)

-Presentation 3

Java Server Facelets

Officially, the xhtml that we have been using is referred to as *java server facelets*

This is the set of tags, such as <h:body>, etc that we've been using

Tag Libraries Supported by Facelets

- JavaServer Faces Facelets Tag Library
  - Tags for templating

  `xmlns:ui="http://xmlns.jcp.org/jsf/facelets`
- JavaServer Faces HTML Tag Library
  - Component tags for all UIComponent objects

  `xmlns:h="http://xmlns.jcp.org/jsf/core`
- JavaServer Faces Core Tag Library
  - Custom actions

  `xmlns:f="http://xmlns.jcp.org/jsf/html`

Tag Libraries Supported by Facelets

- Pass-through Elements Tag Library
  - Supports HTML5-friendly markup

```
xmlns:jsf="http://xmlns.jcp.org/jsf/jsf
```

- Pass-through Attributes Tag Library
  - Supports HTML5-friendly markup

```
xmlns:p="http://xmlns.jcp.org/jsf/passthrough
```

HTML Tag Library

- `<head />` *becomes* `<h:head />`
- `<link />` *becomes* `<h:outputStylesheet />`
- `<script />` *becomes* `<h:outputScript />`
- `<body />` *becomes* `<h:body />`
- `<form />` *becomes* `<h:form />`
- `<label />` *becomes* `<h:outputLabel />`
- `<textarea />` *becomes* `<h:inputTextarea />`
- `<img />` *becomes* `<h:h:graphicImage />`

HTML Tag Library - <input />

- type="button" makes it an<h:commandButton />
- type="checkbox" makes it an <h:selectBooleanCheckbox />
- type="file" makes it an <h:inputFile />
- type="hidden" makes it an <h:inputHidden />
- type="password" makes it an <h:inputSecret />
- type="reset" makes it an <h:commandButton />
- type="submit" makes it an <h:commandButton />
- Every other type makes it an <h:inputText />
  - Includes color, date, datetime, datetime-local, email, month, number, range, search, time, url, week.

Passing parameters

There are a few ways to pass parameters to your action beans:

1) <h:commandButton
   action="#{user.editAction(delete)}" />

This requires the editAction() method to take as input a String. In this event the String will be assigned the value "delete"

<f:param> tag

2) <h:commandButton action="#{user.editAction}" />
<f:param name="action" value="delete" />

The editAction() method will not take a String. Rather we will parse the parameter

```
Map<String,String> params = FacesContext.getExternalContext().getRequestParameterMap();
String action = params.get("action"); // assigns "delete" to action
```

<f:attribute>

For ActionListeners, Validators, and Converters, use <f:attribute>

3) <h:commandButton action="#{user.editAction}" actionListener="#{user.attrListener}">

<f:attribute name="action" value="delete" />

</h:commandButton>

The action listener can refer to this as:

String action = (String)event.getComponent().getAttributes().get("action"); // assigns "delete" to action

Image Uploading

- To include a file uploader in your page, you need to do two things:

1) Create the Java Sever Facelets code with an <h:inputFile> tag to generate the html for a file uploader. The value of this should be associated with a bean of type *Part*

2) Write a method to write the information to disk.

3) “Optional”: Add a validator to check that the size of the image is not too big, etc

<h:inputFile>

- The inputFile tag is a normal tag, but the value should be connected with a bean containing a *Part* property:

```
<h:inputFile value="#{form.uploadedFile}" />
```

```
@Named
@SessionScoped
public class Form {
        private Part part; // add get/set
}
```

Filtering by extension should be done on the server side. It can also be achieved via a pass-through attribute (html attribute is *accept="jpg,gif"*)

Server response

The bean should look like the following:

```
@Named
@SessionScoped
public class Form {
        private Part part; // add get/set

        // The action done when form is submitted
        public void FormSubmit() {
                // parse the Part object and write it to disk somewhere
                // (the next few slides code is all within this
        }
}
```

Step 1: Convert Part object to bytes

```
InputStream in = part.getInputStream();
  ByteArrayOutputStream os = new ByteArrayOutputStream();
      byte[] buffer = new byte[1024]; // 1024 is buffer size
      int len;
      while ((len = in.read(buffer)) != -1) {
          os.write(buffer, 0, len);
      }

      byte[] allBytes = os.toByteArray();
```

Step 2: Choose the location to save to

Use UUID to ensure it is a new name every time

```
String randomFileName = java.util.UUID.randomUUID().toString();
String extension= file.getSubmittedFileName().substring(part.getSubmittedFileName().lastIndexOf("."));

String outputFile = Config.getFileUploadFolder() + randomFileName + extension;
```

Step 3: Write the bytes!

```
// using java.nio
Files.write(Paths.get(outputFile), allBytes, StandardOpenOption.CREATE);
```

Validators

It is absolutely necessary to add validation on the server side

There are many risks if you don’t do this. The most obvious one would be a DOS attack wherein someone uploads files too big for the space (e.g. several gb large)

Thus we need to add a validator!

TextFile Validator

(src: https://jsflive.wordpress.com/2013/04/23/jsf22-file-upload/)

```
@FacesValidator(“my.project.TextValidator”)
public class TextFileValidator {
public void validateFile(FacesContext ctx,
                    UIComponent comp,
                    Object value) {
  List<FacesMessage> msgs = new ArrayList<FacesMessage>();
  Part file = (Part)value;
  if (file.getSize() > 1024) { // in bytes
    msgs.add(new FacesMessage("file too big"));
  }
  if (!"text/plain".equals(file.getContentType())) {
    msgs.add(new FacesMessage("not a text file"));
  }
  if (!msgs.isEmpty()) {
    throw new ValidatorException(msgs);
  }
}


}
```

In the JSF

```
<h:inputFile value="#{form.uploadedFile}"
validator="my.project.TextValidator" />
```

Or to have AJAX validation:

```
<h:inputFile value="#{form.uploadedFile}"
validator="my.project.TextValidator">
<f:ajax execute="change" render="errorOrThumbnailSection" />
</h:inputFile>
```

Custom Tags

Including content

- **`ui:include`**
  - Includes content from another XHTML page.
- **`ui:composition`**
  - If used with `template` attribute, the specified template is loaded
  - If it's a group of elements, the elements can be inserted into another page
  - All tags outside of `ui:composition` are removed

Including content

- <ui:include src="include.xhtml" />
- The include.xhtml must be inside of a composition tag:

```
<ui:composition
   xmlns="http://www.w3.org/1999/xhtml"
   xmlns:f="http://xmlns.jcp.org/jsf/core"
   xmlns:h="http://xmlns.jcp.org/jsf/html"
   xmlns:ui="http://xmlns.jcp.org/jsf/facelets">
   <h2>Include page</h2>
   <p>Include page blah blah lorem ipsum</p>
</ui:composition>
```

Custom Tags – xhtml method

- Custom tags can be created to insert snippets of xhtml for easy reuse in new pages
- These snippets are considered custom components that you have created
- Can be used in place of or as part of templating

Custom Tags – xhtml method

- To define a custom tag you must:

1. Create an xhtml file and define the contents using the **ui:composition** tag
2. Create a tag library descriptor (.taglib.xml file)
   - Declares the custom tag in it
3. Register the tag library descriptor in the web.xml

Define custom tag contents

```
<ui:composition>
   <h:commandButton type="submit" value="#{okTxt}" />
   <h:commandButton type="reset" value="#{cancelTxt}" />
</ui:composition>
```

- It is not necessary to include the usual tags such as <html> and <body>
- However, NetBeans will declare the file in error for missing these standard tags
- The values will be supplied when the tag is used
- When you write value="#{*parameter*}", it will eventually become an *attribute* of your new tag!

Define a tag library descriptor

- The available tags must be listed in an xml file
- By convention it is named xxxx.taglib.xml where xxxx is the name you give to the library

```
<facelet-taglib>
    <namespace>
        http://kenfogel.com/facelets
    </namespace>
    <tag>
        <tag-name>buttonpanel</tag-name>
        <source>tags/buttonpanel.xhtml</source>
    </tag>
</facelet-taglib>
```

Define a tag library descriptor

- The namespace is an identifier formatted as a url
- It does not need to be, or rarely is, a real url
- You can define one or mare tags in the file
- They all share the same namespace but the xhtml files can be stored in different folders or all in the same one
- The folder is placed in the root folder (i.e. WEB-INF)

Register the tag library

- The tag library must be registered in the web.xml
- It is listed as a context-param so that any servlet, such as the faces servlet, can use it

```
<context-param>
   <param-name>
      javax.faces.FACELETS_LIBRARIES
   </param-name>
   <param-value>
      /WEB-INF/kenfogel.taglib.xml
   </param-value>
</context-param>
```

Using a custom tag

- Include its namespace in the xhtml file and assign it a prefix

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
      xmlns:kf="http://kenfogel.com/facelets" >
```

Using a custom tag

- It’s now ready to use!

```
<h:body>
    <h:form>
        <kf:buttonpanel okTxt="#{msgs.ok}"
                        cancelTxt="#{msgs.cancel}" />
    </h:form>
</h:body>
```