Internet Application Project: Java Server Side

420-625-DW

Java Server Faces Basics

Last Time: Context Dependency Injection (CDI)

We'll look more closely at CDI as a design pattern for setting up objects

-Three ways to @Inject

-What can be @Injected

-What do to if you're trying to @Inject something that is not a bean?

-A few more miscellaneous things (@Qualifier, @Disposes, @Alternative, @Default)

A couple more things about CDI

-If a field is *static* we cannot mark it for @Inject (so all the concepts we looked at last class are meant for *instance* fields)

-@Dependent is an annotation that can be used to make a sub-bean rely on the same scope as its parent. For example, if you declare class Foo to be @Dependent, and you try to @Inject a Foo from the class Bar, which is SessionScoped, then Foo will also be @Dependent

-Note that @Dependent is required for generic types

Mixing and matching

You can only use @Inject (or @PersistenceContext) from an object that is itself injected.

When *you* call the constructor yourself, none of the @Injects will happen

```
@SessionScoped
public class Foo {
        @Inject private Bar b;
….
}
Foo f = new Foo(); // will not @Inject b! So b is still null!
```

Java Server Faces

JSF is an example of the Model-View-Controller pattern

Model: Managed beans that can store information and connect to other features

View: .xhtml files that can be used to control what the page looks like

Controller: Automatically done within JSF. This is handled by the *FacesServlet* classes.

Recall: web.xml

We have seen web.xml before.

Within web.xml, you specify things such as what the home page will be.

You also specify which *servlet* will handle the request.

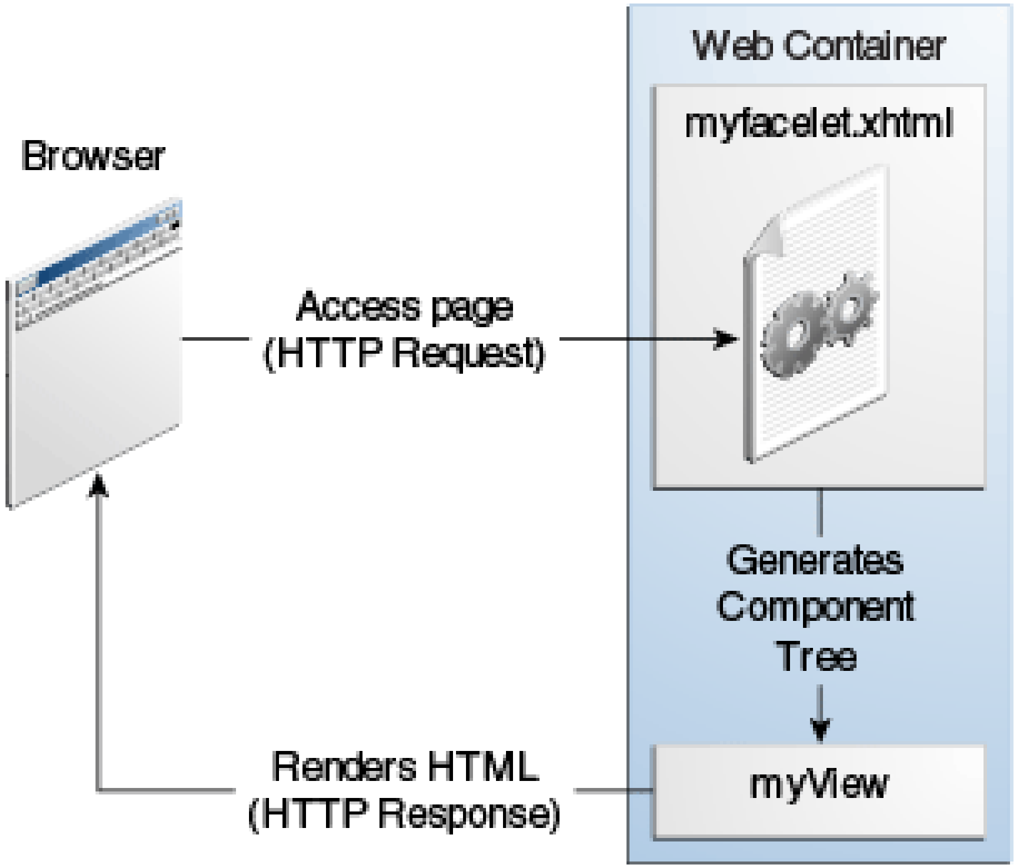
In our case, to make JSF work properly, we need to say that all requests should go through the *FacesServlet*

Sample web.xml

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

<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
                    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
                    version="3.1">
   <session-config>       <session-timeout> 30       </session-timeout>    </session-config>
      <welcome-file-list>    <welcome-file>faces/home.xhtml</welcome-file></welcome-file-list>
   <!--     FacesServlet is main servlet responsible to handle all request.
   It acts as central controller.  This servlet initializes the JSF components before the JSP is displayed. -->
   <servlet>
      <servlet-name>Faces Servlet</servlet-name>
      <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
      <load-on-startup>1</load-on-startup>
   </servlet>
   <servlet-mapping>
      <servlet-name>Faces Servlet</servlet-name>
      <url-pattern>/faces/*</url-pattern>
   </servlet-mapping>
   <servlet-mapping>
      <servlet-name>Faces Servlet</servlet-name>
      <url-pattern>*.jsf</url-pattern>
   </servlet-mapping>
   <servlet-mapping>
      <servlet-name>Faces Servlet</servlet-name>
      <url-pattern>*.faces</url-pattern>
   </servlet-mapping>
   <servlet-mapping>
      <servlet-name>Faces Servlet</servlet-name>
      <url-pattern>*.xhtml</url-pattern>
   </servlet-mapping>
</web-app>
```

Responding to a Client Request for a JavaServer Faces Page



Idea

The idea is that JSF will build a tree for you of all the UI components. This is done based on the xhtml code that we write.

This tree can then get rendered as html.

Most of the work behind the scenes is in building and maintaining this tree

From xhtml to a tree

- Each tag represents a component in the tree
- Each component has a *renderer* that produces HTML output that reflects the component state
- Code that is not part of a specific tag (e.g. plain text) is passed along "as is" and then rendered

Step 1: Build view tree (note: not the same as html!)

-The first time a user goes to a page, the program needs to build a *component tree*. This is a list of all the UI pieces that the view needs. It includes *references* to the managed beans that are within individual UI components.

-At this time, event handlers and validators (if we are using them) are added to the tree as well.

If it is a return to the page, known as a "postback request" then this needs to be rebuilt. It can do this because the previous built trees are stored within the *FacesContext*

Step 2: Request values

Next, any request values / parameters that are part of the request (either via get or set) are applied to the view to adjust it.

If there are no request values, then no validation is necessary (nor model updating), so we will skip to the render() phase

This is also when any events / listeners will happen. These will occur based on what we insert into the view code:

```
<h:commandButton id = "submitButton"
  value = "Submit" action = "#{userData.showResult}"
  actionListener = "#{userData.updateData}" />
</h:commandButton>
```

updateData() must take as input an ActionEvent object

Step 3: Do validation

Next, we will do validation. This is based on any validators you've added to your code.

If the new values submitted are not good, the request will terminate

Example validator:

```
<h:inputText id = "nameInput" value = "#{userData.name}"
      label = "name" >
      <f:validateLength minimum = "5" maximum = "8" />
   </h:inputText>
```

Ensures (on the server side) that the String has the correct length.

Note: When we use *Ajax*, we can treat a form *as if* it was submitted, so that the user can do server side validation that appears as if it were client side

Custom Validator : Method 1

You can also create custom validators with your own logic in them. This allows you to have your own custom logic in it.

```
<h:inputText id="urlInput" value="#{userData.data}"
label="URL" >
   <f:validator validatorId="com.tutorialspoint.test.UrlValidator"
/>
</h:inputText>
```

In this case, UrlValidator is a class that should implement the interface *Validator.* The validatorId is just any unique ID that you will give to your validator (similar to @Named())

Validation errors

If there are any validation errors, you can display then using the tag *message*

```
<h:message for="userid" id="userIdError" style="color:
red"/>
```

Custom Validator : Method 2

It is also possible to define the validation within the bean itself

```
<h:inputText id="mno" value="#{mobile.mno}" required="true"
size="4" disabled="#{mobile.mno}"
validator="#{mobile.validateModelNo}">
```

In this case, we must have a method:

```
public void validateModelNo(FacesContext context,
UIComponent comp, Object value)
```

which throws an exception if it's invalid. The message thrown is what will be shown in the ui (e.g. within the h:message tag)

Preview: With Ajax

```
<h:inputText id="userid" value="#{registration.userId}"
validator="#{registration.validateUserId}">
    <f:ajax event="blur" execute="@this"
render="userIdError"/>
    </h:inputText>

  <h:message for="userid"
id="userIdError" style="color: red"/>
```

Step 4: Update Model

Next, we will update the *model*, that is, the beans.

Note that in phase 2, we updated only the *ui view*. That is, we updated the view with whatever extra information.

This is essentially synchronizing the model / beans with the view:

```
bean.setValue(view.getValue())
```

Step 5: Invoke Application

At this point, we handle any form actions, links, etc.

Step 6: Render

Finally, we render the tree.

This is left up to the server (e.g. glassfish, tomcat, IIS, etc) to figure out exactly what the html should be

Example "life cycle"

Suppose we have a login page with elements:

-username field

-password field

-submit button

First view

The first time we go to the page, JSF builds the component tree of what to render.

The tree shows that there is an html page, which contains a form which has 3 elements in it.

The 3 elements are linked to the managed bean that they are connected with.

Enter a bad user name

Suppose we enter a bad user name (and hit submit)

Then the request is sent. The new updates are applied to the *view*, but not the model.

The validation fails, so the model is not updated. We skip to the render() step which is translated using Glassfish (or whatever server you're using). At this step, the tag <h:message> will display the error message

Enter a good name

If we enter a good name, then:

-view is updated

-the validation is successful

-the backing bean is synced with the view

-we do the action triggered by the form (i.e. the method, and if it returns something, then we show that view– which involves creating this tree again!)

Other configuration information

There are several other important configuration files in addition to the web.xml file

We'll go through them now

glassfish-web.xml

- Should be located within WEB-INF (same folder as web.xml)
- Contains server configuration information specific to glassfish. (If you choose to use a different server e.g. TomCat, you would need to find those specific configurations)
- One use out of this is to map local folders to the request. For example, for pictures

Using Pictures in JSF

You can insert a picture into JSF using the <h:graphicImage> tag

```
<h:graphicImage value = “resources/images/foo.jpg"/>
```

If you use a relative path for the value (which you should do whenever referring to a page that is part of your app!), then it is expected that within your project you have webapp/resources/images/foo.jpg

Pictures as resources

- Everything within the resources folder will get automatically replaced whenever you redeploy
- If you have a static picture on your page, it is fine—and even recommended-- to include it within the resources folder
- Since it is part of the application, it makes sense that every time you deploy it, it should get updated
- For non-static images, such as uploaded data, user avatars, etc, this doesn't make sense
  - Use local disk system (what we'll use)
  - Use blob storage or some sort of distributed server (for distributed systems, backup, etc)

How to:

- Normally though, a “random” folder on the hard drive is not visible to the server
- We can include this by adding a line in the glassfish-web.xml file:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE glassfish-web-app PUBLIC "-//GlassFish.org//DTD GlassFish Application Server 3.1 Servlet
3.0//EN" "http://glassfish.org/dtds/glassfish-web-app_3_0-1.dtd">
<glassfish-web-app error-url="">
  <class-loader delegate="true"/>
  <jsp-config>
    <property name="keepgenerated" value="true">
      <description>Keep a copy of the generated servlet class' java code.</description>
    </property>
  </jsp-config>
  <property name="alternatedocroot_1" value="from=/Downloads/* dir=C:/Users/danie" />
</glassfish-web-app>
```

This means that from now on, we can write Downloads/whatever and it will automatically get mapped to the folder C:/Users/danie on the server’s computer

So you could store the images there, and yet refer to them on your page!

```
 <h:graphicImage value = "Downloads/car.jpg"/>
```

faces-config.xml

- Some additional configuration can be put into faces-config.xml
- Also goes into WEB-INF folder
- Much of this is now deprecated, because it is handled by the annotations
- The main thing used today still is for navigation rules as well as resource bundles, which are useful for multi language support

Coming up:

-What to do with images (how to receive them)

-Internationalization

-More on Facelets (the syntax for .xhtml)

-Validators / Ajax / Listeners / Events