Deeper into Java Server Faces & Validation

GÖTEBORGS UNIVERSITET

DAT076/DIT126 Web Applications

Adam Waldenberg

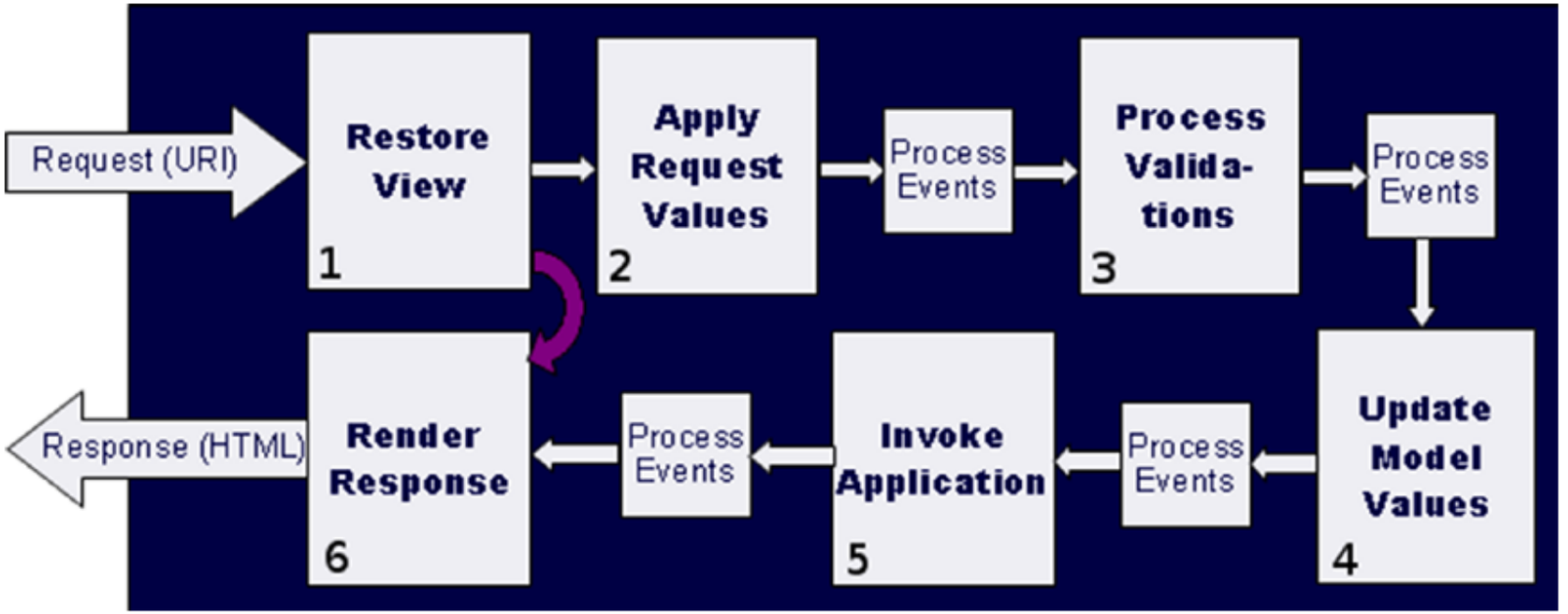
JSF Life cycle

- **Whenever a request occurs, Java Server Faces does a number of things to handle the request**

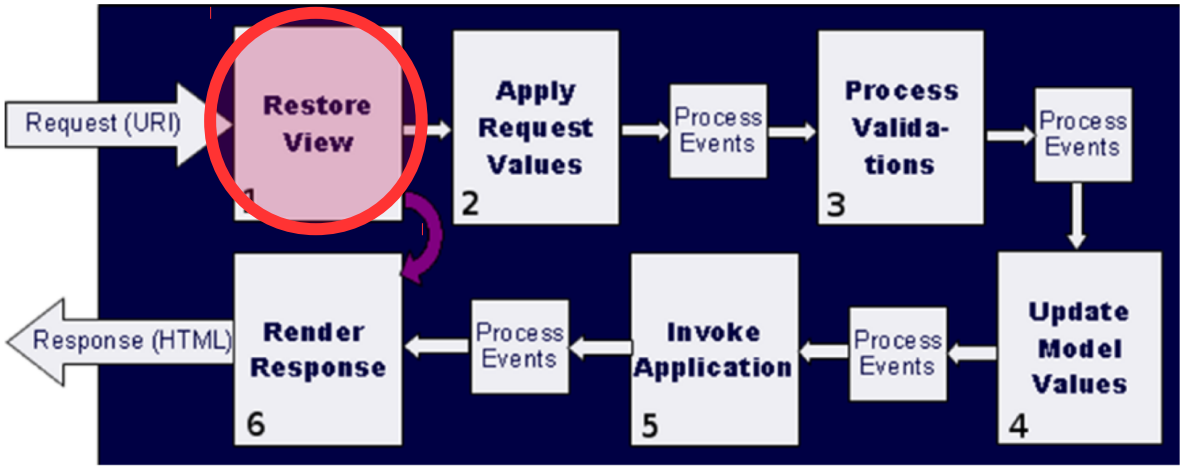
- Create and populates the view
- Fill in the state of the components
- Validate all input that was requested for validation
- Call all the backing values
- Call any invoked events
- Render the view!

JSF Life cycle

The JSF lifecycle consists of six separate phases



JSF Life cycle – Restore View



Step 1:

Build the view and wire event handlers and validators to UI components

- Begins as soon as a link or button is clicked and JSF receives a request
- Tries to identify the view and connects all logic to each component
- Save everything in ***FacesContext***

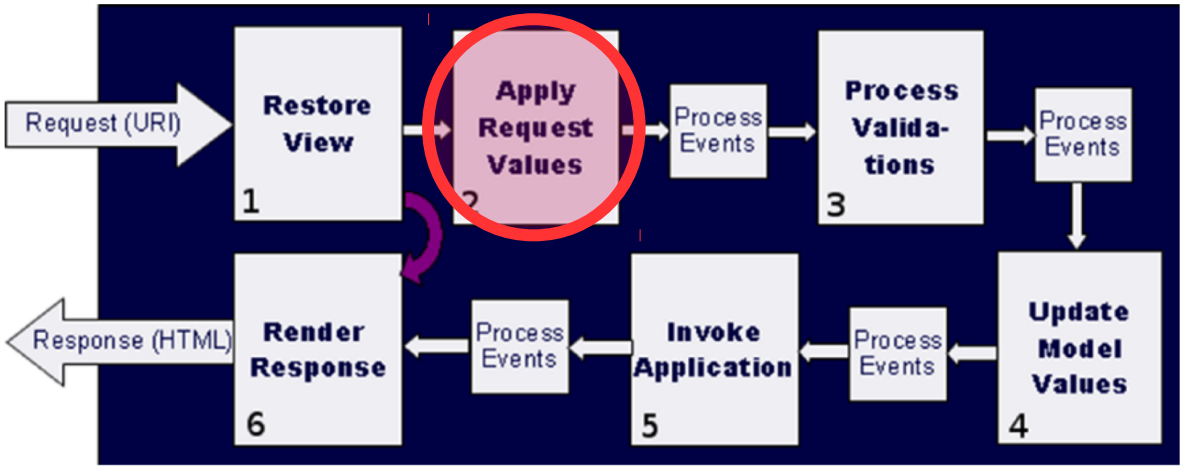
Restore View side effects

The way JSF works can cause a very annoying (but pretty harmless) exception

```
javax.faces.application.ViewExpiredException: View could not be restored
```

- The `<o:enableRestorableView>` tag of OmniFaces can work around this
- Instructs the view handler to recreate the entire view whenever the view has been expired

JSF Life cycle – Apply Request

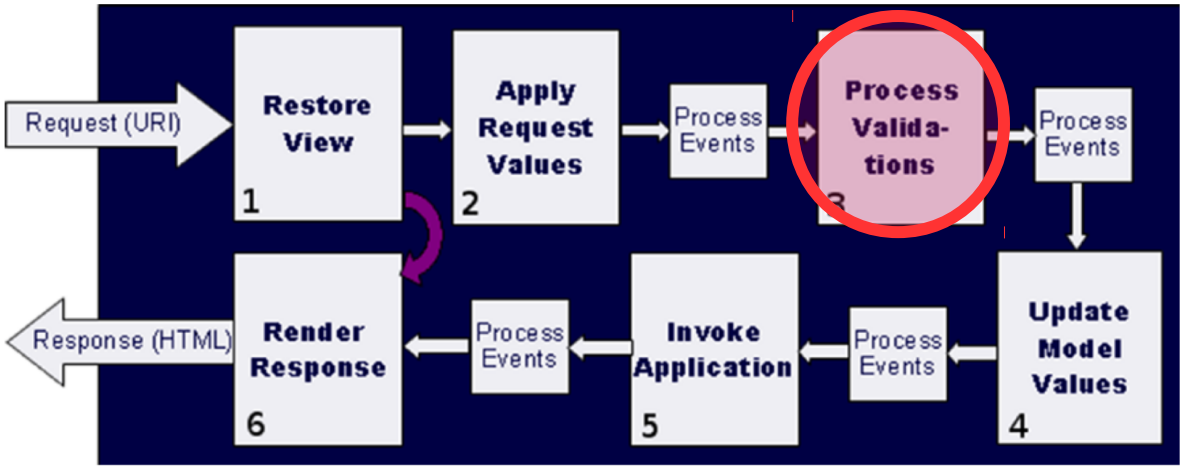


Step 2:

Apply values to components in the view

- For example apply text fields and selections to the components in the view
- With **`immediate=true`** you can fire action events (which are otherwise fired in the *Invoke Application phase*) at the end of this phase, in order to skip validation.
- Upon failures we then jump to the *Render Response phase*

JSF Life cycle – Process Validations

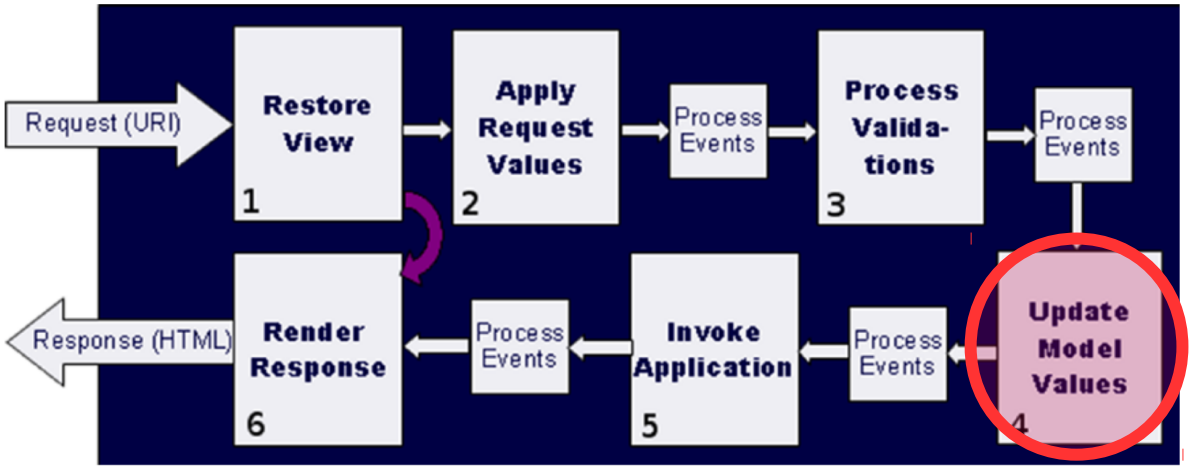


Step 3:

Execute conversion plus JSF and Bean Validation on the component properties

- Validators only happen if **`rendered=true`** is set on a component
- Conversions happen before validators are called
- Upon failures we then jump to the *Render Response phase*

JSF Life cycle – Update Model Values

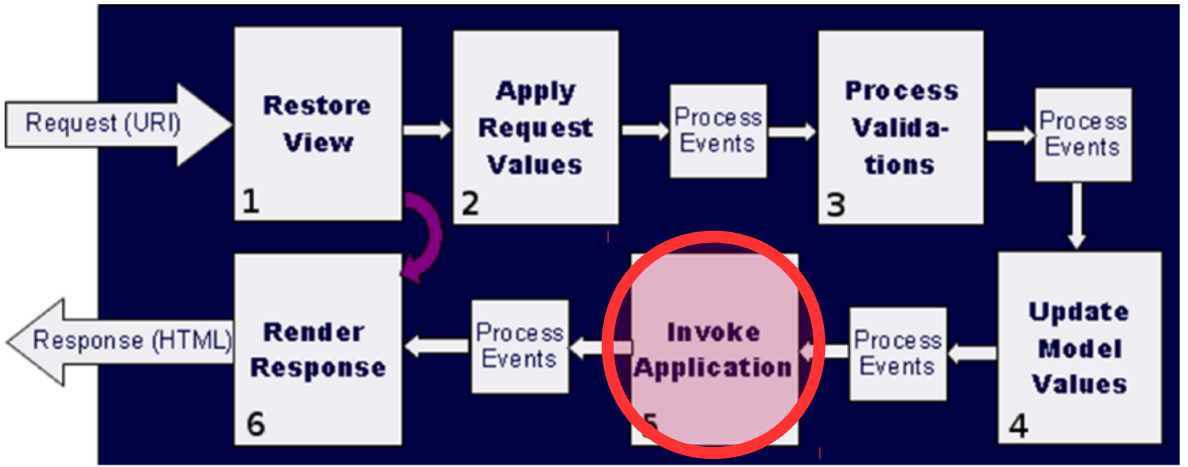


Step 4:

Update the view model and beans with the value set on the component properties

- If we reach this phase it means user input is syntactically and logically correct
- Calls the setter of every component property and updates the model

JSF Life cycle – Invoke Application

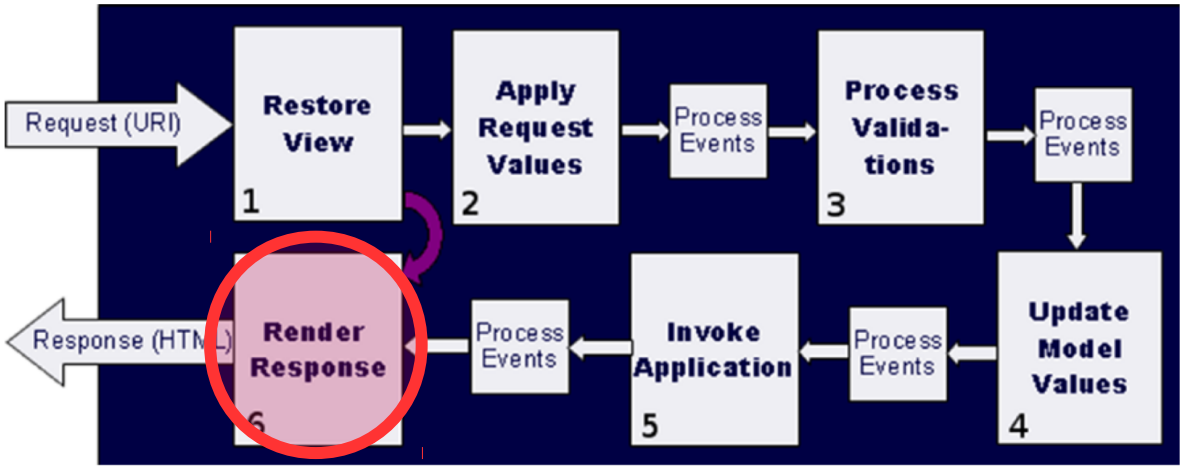


Step 5:

Call all the registered event listeners

- Action listeners get fired first, with an option to modify the response
- Next, actions with the business logic get called and determine the location of the next page
- Use the action listener to prepare for the action call (a component can have an arbitrary amount of listeners)

JSF Life cycle – Render Response



Step 6:

Render the response

- Store the state of the view before rendering the response
- Render the response back the client

Using the JSF Life cycle

JSF offers methods to allow you to plug into the different phases of the life cycle

```
<f:event type="eventType" listener="#{bean.onEvent}" />
```

- **postAddToView** Runs right after the component is added to view during the restore view phase or render response phase
- **preValidate** Runs right before the component is to be validated
- **postValidate** Runs right after the component has been validated
- **preRenderView** Runs right before the view is rendered (render response phase)
- **preRenderComponent** Runs right before the component is rendered (render response phase)

Using the JSF Life cycle continued

There is also an alternate method to allow you to listen to all the different phases directly

```
<f:view beforePhase=”#{bean.onEvent}”/>
<f:view afterPhase=”#{bean.onEvent}”/>
```

```
@Data
@Named
@ViewScoped
public class Bean implements Serializable {
    public void onEvent(PhaseEvent event) {
        if (event.getPhaseId() == PhaseId.RENDER_RESPONSE) {
            /* Do something… */
        }
    }
}
```

Using life cycle events

An example of an event re-shuffling the component tree prior to the rendering phase of a component

```
<?xml version="1.0" encoding="UTF-8"?>
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:h="http://java.sun.com/jsf/html"
      xmlns:p="http://primefaces.org/ui" xmlns:f="http://java.sun.com/jsf/core">
    <h:head><title>Shuffle test</title></h:head>
    <h:body>
        <h:panelGroup>
            <p:panel header="First"/>
            <p:panel header="Second"/>
            <p:panel header="Third"/>
            <p:panel header="Fourth"/>
            <f:event listener="#{shuffleBackingBean.onShuffle}"
                     type="preRenderComponent" />
        </h:panelGroup>
    </h:body>
</html>
```

Using life cycle events

The implementation of the shuffle event

```
@Named
@ViewScoped
public class ShuffleBackingBean implements Serializable {
    public void onShuffle(ComponentSystemEvent event) {

        final List<UIComponent> components = new
           ArrayList<>(event.getComponent().getChildren());

        Collections.shuffle(components);
        event.getComponent().getChildren().clear();
        event.getComponent().getChildren().addAll(components);
    }
}
```

Using life cycle events

Let’s test it and look at this life cycle event in action!

Validating data

EE and JSF primarily focuses on server-side validation

- While there are JSF tags for validation such as `<f:validateLength>` and `<f:validateRegex>` etc, bean validation is the recommended method
- Bean validation offers a rich set of validators by default with many external libraries providing many useful validators. Implementing your own validator is also very easy
- ***javaee.github.io/javaee-spec/javadocs/javax/validation/constraints/package-frame.html***
  for an exhaustive list of the validators in EE8
- ***Examples:*** *@Assert, @Digits, @Email, @Future, @Min, @Max, @Pattern*

What about client side validation?

Use a component framework that natively supports client-side validation

- In PrimeFaces you can set `validateClient="true" on command components` to enable client-side validation
- PrimeFaces even handles the standard bean validators and make them work client-side!
- The API can be extended with custom validators; ***primefaces.github.io/primefaces/8_0/#/core/clientsidevalidation***
- There is also a component `<p:clientValidator>` that allows you to add dynamic on-page validation

PrimeFaces examples

```
<p:inputText id="integer" value="#{validationBean.integer}">
    <p:clientValidator event="keyup"/>
</p:inputText>
```

```
<h:form>
    <p:messages />
    <p:inputText required="true" />
    <p:inputTextarea required="true" />
    <p:commandButton value="Save" validateClient="true" ajax="false" />
</h:form>
```

- Bean validatiors become client-side!
- Notice how we are able to fire validation events in real time despite the connection to the server-side

HTML5 example

Preferred way when using HTML5 validation is to use the previously mentioned *pass-through* library

```
<label for="startDate" >Trip Start:</label>
<h:inputText pt:type="date"  id="startDate"
             value="#{...tripStartDate}" >
    <f:convertDateTime pattern="YYYY-MM-dd" />
</h:inputText>
```

```
<h:inputText id="number" pt:type="number"
             pt:min="1" pt:max="10"  value="..."/>
```

Programatic validation and attributes

You can also nest *f:passThroughAttribute* inside a component

```
<h:inputText value="#{html5Bean.text}" required="true">
    <f:passThroughAttributes value="#{html5Bean.attrs}"/>
</h:inputText>
```

```
@PostConstruct
public void init() {
    attrs = new HashMap<String, String>();
    attrs.put("type", "range");
    attrs.put("min", "1");
    attrs.put("max", "10");
    attrs.put("step", "2");
}
```

Using validation

Let’s take a look at some validation examples!

Navigation in Java Server Faces

In JSF, the action callback decides the navigation outcome and to which location redirection occurs

```
<h:commandButton action=”#{myControllerBean.onPressed}”/>
```

```
public String onPressed() {
    if (condition) {
        /* do stuff */
        return “success”;
    }
    return “error”:
}
```

- Navigation in JSF is either ***implicit*** or ***rule-based***
- Both methods have their advantage and disadvantage

Defining navigation rules

Navigation rules are defined in the *faces-config.xml* file of your application

```
<navigation-rule>
    <from-view-id>page1.xhtml</from-view-id>
    <navigation-case>
        <from-outcome>success</from-outcome>
        <to-view-id>/page2.xhtml</to-view-id>
    </navigation-case>
    <navigation-case>
        <from-outcome>error</from-outcome>
        <to-view-id>/error.xhtml</to-view-id>
    </navigation-case>
</navigation-rule>
```

- Notice how these navigation rules plug into the outcomes from the action event in the last example

Sending and reading query parameters

Several approaches are available. JSF makes it very easy and OmniFaces improves it further

- You can bind the a query parameter to a backing bean value – allowing you access to it in your controller layer

```
<f:metadata>
    <f:viewParam name="parameterOne" value="#{bean.parameterOne}"/>
</f:metadata>
```

- OmniFaces makes it ridiculously easy and even removes the binding from the XHTML page

```
@Inject @Param
private String foo;
```

More on parameters

With OmniFaces we also have easy support for path parameters and hash parameters

```
http://example.com/mypage/john.smith
```

```
@Inject @Param(pathIndex=0)
private String name;
```

- This would inject the string *“john.smith”* into the name variable on page load

```
http://example.com/page.xhtml#foo=baz
```

```
<f:metadata>
    <o:hashParam name="foo" value="#{bean.foo}" />
    <o:hashParam name="bar" value="#{bean.bar}" default="kaz" />
</f:metadata>
```

- This would place the string *“baz”* into the backing property *foo*

Listening on hash URL changes

OmniFaces also allows us to dynamically listen to hash-changes in the URL

- Could be in reaction to pressing a link
- Could also be in rection to the user manually entering a new hash link
- Fires the event ***HashChangeEvent*** which can be observer using CDI in the followin way;

```
public void onHashChange(@Observes HashChangeEvent event) {
    String oldHashString = event.getOldValue();
    String newHashString = event.getNewValue();
    // ...
}
```

Using cookies

Using cookies in JSF was not very seamless in the past, but OmniFaces makes it very simple

```
@Inject @Cookie
private String foo;
```

- The cookie name is taken from the variable
- Can be overridden with `@Cookie(name="foo")`
- It really is that easy!

Some upcoming features

Yours truly has been working on @JSParam for OmniFaces which allows for simple injection of JavaScript expressions

```
@JSParam("window.screen.width")
private String screenWidth;
```

```
@Data
@JsonIgnoreProperties(ignoreUnknown = true)
public static class Navigator {
    @JsonProperty private String vendor;
    @JsonProperty private String userAgent;
    @JsonProperty private String language;
}

@JSParam("navigator")
private Navigator navigator;
```

- Pull request here: ***github.com/omnifaces/omnifaces/pull/506***

Basic web.xml settings needed for JSF

A couple of settings need to be done in your web deployment descriptor (web.xml)

```
<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>*.xhtml</url-pattern>
</servlet-mapping>
<welcome-file-list>
    <welcome-file>index.xhtml</welcome-file>
</welcome-file-list>
```