CS 110
Introduction to Internet Programming

Java Server Faces

Dr. Williams
Central Connecticut State University

Model View Controller

- The purpose of object oriented programming is to encapsulate details of implementation
- The purpose of the model view controller architecture is to provide similar encapsulation at a higher level
  - Business/Data Model
  - View/Presentation
  - Control of application flow

MVC cont.

- Model
  - Objects that represent business entities
  - Encapsulate business logic
  - Encapsulate how data is stored
- View
  - Logic for how model representation is displayed
  - Change in presentation of data can be made without changes to object model
- Controller
  - Control flow between view elements
  - Control when data should be persisted

Java Server Faces

- Purpose of Java Server Faces (JSF) is to provide a framework for MVC pattern
- JSF page similar structure as JSP – flexibility for rich display components but
  - easier interaction with data model
  - easier interaction with controller flow
- JSF code resides in an XHTML file referred to as a facelet

JSF tags

- JSF tags are similar to the JSP tag libraries
- Set of predefined tags to make form layout, validation and integration with controller and model easier
- Like JSP, JSF pages can consist of a mixture of HTML code along with dynamic code

JSF tags cont.

- Similar to JSP tag libraries, the first thing that must be done is mapping a prefix to a tag library
- Most common ones are:
- xmlns:h="http://java.sun.com/jsf/html"
  - Contains tags related to rendering HTML code
- xmlns:f="http://java.sun.com/jsf/core"
  - Core JSF functionality for things like validation

Syntax of import

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

- This is followed by tags which are similar to their HTML counterparts
  - <h:head>, <h:body>
- Reason plain HTML is not used is these serve as markers for where generated code should placed within the page

First JSF page

- firstJSF.xhtml

Key JSF tags for constructing the page

- <h:messages> (explained in detail later)
- <h:form> creates form information allows attributes to be dynamically filled based on page contents (method is always POST)
- <h:outputText> similar to JSP <c:out> - used to output while safely escaping values
- <h:inputText> input textbox – *label* specifies field name that should be displayed for validation errors
- <f:validateLength> covered later
- <h:commandButton> submits form

JSF resources

- Importing a style sheet or javascript file
  - <h:outputStylesheet library="css" name="styles.css"/>
  - <h:outputScript library="js" name="common.js" />
- To ease maintaining consistency throughout the site JSF standardizes where resources of style sheets and javascript files should be placed
  - All CSS and javascript placed under resources directory
  - Referencing file specified by library=“path”

Navigation

- Navigation of a page is defined via the
- <h:commandButton action="confirmation" value="submit"> tag
- The action attribute specifies a name of the page to navigate to
  - If it is another JSF page only the basename needs to be specified
  - Or controller method (discussed later)
- Value is the text to display on the button

Layout extensions

- As with JSPs, HTML elements like tables can be used to define layout, but JSF also defines a couple of tags to make common layout's easier
- <h:panelGrid columns="2" columnClasses="rightAlign,leftAlign“>
- Constructs a table where each JSF tag between the panel grid tags is placed into cells across the table until the end of the row is reached then the next element goes on the next row.
- The columnClasses attribute allows the user to specify a CSS class to be applied to each of the different columns

PanelGrid/PanelGroup example

- <h:panelGroup> Allows more than one element to appear in a cell of the panel grid – it can also be used to create an empty cell
- Example:

panelGridExample.xhtml

Managed beans

- The crux of how JSF works is driven by **managed beans**
- A managed bean is just like any other Java bean but has the annotation @ManagedBean before the declaration of the class
- Managed beans mean they can be used within JSF pages and bound to fields, functionality, events

Managed bean scopes

- The scopes of managed beans works similar to the way beans in JSPs were used.
- Underneath where the @ManagedBean annotation appears the scope is defined
  - @ApplicationScoped
  - @SessionScoped
  - @RequestScoped
  - @ViewScoped
  - @NoneScoped – instantiated when accessed by another bean
  - @CustomScoped – user defined map for when the bean should be within scope
- If not specified default is request scoped

Binding values

- A JSF page is made dynamic by adding **value binding expression**s
- A binding expression is of the form:

```
<h:inputText label="email"
  value="#{customer.email}">
```

- This syntax says that the managed bean with class name Customer will have its value email tied to this input text field. Note when referencing bean first letter is lower case

Binding example

- panelGridExample.xhtml
- confirmation.xhtml

Create a basic JSF

- Create a managed bean named Car
  - Make
  - Year
  - StateOwned (state is a key word)
- Create JSF pages
  - Enter data
  - Display data

Validation

- One of the most common task with forms is validating the entered data
- JSF provides framework for specifying common validation
  - Length
  - Validity of number field
  - Required fields entered
  - Matches regular expression
  - Capability to extend and create custom validation

Validation framework

- Upon form submit, JSF's validation framework executes checking validity of fields
  - If a field value is invalid the framework writes an error message to the page and prompts the user to fix their data
  - Form will not redirect or process until the validation criteria has been met
- Result: provides the ability for the developer to specify allowed values and after that the developer can assume valid values received

Creating validation

- Validation can be created in three different ways
  - Built in validation
  - Custom validator class (implements Validator interface)
  - Custom validation function
- Once developed and added to the code, framework takes care of enforcing validation

Using validation

- Whenever validation is processed a list of validation errors are collected to be displayed back to the user to display these errors, the messages tag must be added to the page

```
<h:messages></h:messages>
```

- When validation errors occur they will be added to the page in the location of this tag
- Without it the user will be unaware as to why the form is not submitting

Standard validation

Validation tag	Description
<f:validateLength minimum="2" maximum="5">	Verifies input length is between the tag's minimum and maximum values
<f:validateDoubleRange minimum="0.2" maximum="75.5">	Verifies input is valid Double within the specified range inclusive
<f:validateLongRange minimum="-56" maximum="1234">	Verifies input is valid Long within the specified range inclusive
<f:validateRequired>	Validates that the tag is not empty (same as adding required="true" in the input text tag)
<f:validateRegex pattern="[a-z]+">	Validates that the value is in the format specified by the regular expression in the pattern attribute
<f:validateBean>	Allows validation to be done by annotations on the managed bean without having to add validators to the JSF tags

Example of basic validation

```
<h:inputText label="First Name"
 value="#{customer.firstName}" required="true">
 <f:validateLength minimum="2"
      maximum="30">
 </f:validateLength>
</h:inputText>
```

Add validation

- To your input form add the following conditions:
- Make must between a minimum of 3 characters and max of 60
- Year is required and must be between 1960-2013

Custom validation – Validator class

- One of the ways custom validation can be implemented is by creating a class that implements javax.faces.validator.Validator
- The class also must be annotated with @FacesValidator(value="emailValidator")
- This interface has one method validate which throws the ValidatorException
- Essentially what your class does is check the passed input, if it is fine nothing is done otherwise a ValidatorException is thrown

Validator class example

- `EmailValidator.java`

Using a faces validator

- A faces validator is used by adding:

```
<h:inputText label="Email"
    value="#{customer.email}" >
  <f:validator
  validatorId="emailValidator" />
</h:inputText>
```

- Upon submitting the form, the contents of the Email textbox will then be sent to the faces validator we registered with the name “emailValidator” with our annotation
- The JSF framework will then take over to ensure the validation criteria is met before the form is submitted, messaging the user if it is not

Custom validation - Validator methods (sometimes logic part of MVC-Model)

- Another way validation can be created is with validator methods
- Validator methods can be added to any managed bean
- To support validation 1-many functions may be added to the class that accept the same parameters as the validate method on the validator interface
- Useful for adding custom bean logic to the managed entity bean or for creating utility classes that contain many forms of validation

Validator methods ex.

- ValidationUtils.java

Using validator methods

```
<h:inputText label="Last Name"
  value="#{customer.lastName}" required="true"
  validator="#{validationUtils.validateAlphaSpace}">
  <f:validateLength minimum="2"
  maximum="30"></f:validateLength>
</h:inputText>
```

Add validation

- Make must between a minimum of 3 characters and max of 60
- Year is required and must be between 1960-2013
- Develop state abbreviation validator method on ValidationUtils
- Your validator should check that the entered state is a valid state abbreviation (You don't need to enter all 50 just pick 3)
- Add the validation to your state owned field

MVC – Model
Integration with JPA

- One of the main benefits of JSF is its seamless integration with the data model and the controller logic
- Since there is nothing special about Managed beans other than the @ManagedBean annotation a managed bean can also be an entity bean
- Binding values on page easy persistence and reading from DB

MVC - Controller

- The last component of MVC is the controller which is responsible for the flow of the application
- The JSF framework does this by being able to direct control to managed beans
- Managed bean provides method that command button (<h:command>) can bind to its action

Controller cont.

Controller is responsible for

- Enhancing object model values if needed
- Persisting data if needed
- Directing the application to the next page

Controller setup

- Like the other components in the JSF framework, the controller is also a managed bean
- Functions that can be bound to the pages action need to take no parameters and return a String
- String returned will be the page the application should be directed to

Controller and data model

- For the controller to access a bean an attribute for that bean must be added as a property of the bean
- Property is then marked as

```
@ManagedProperty(value = "#{customer}")
private Customer customer;
```

Which binds the customer bean to the customer attribute

- The controller can then persist the bean by creating an instance of the entityManager and calling persist on the bean

MVC demo

- customerDataEntry.xhtml
- CustomerController.java
- confirmationSaved.xhtml

Creating your own MVC

- Extend the pages/bean you developed for the car
- Add a controller to control submission of the car info page that persists the car before directing the browser to the display page you created

Project stages

- One of the features of JSF is for the application and application server to behave differently whether in development or production
  - SystemTest
  - UnitTest
  - Development
  - Production
- Specifically the application server will log differently depending on production stage – ie. More information debugging in development, no informational logging in production

Project stages cont.

- In addition to having the application server automatically adjust its logging, your code can execute based on the property as well:

```
FacesContext facesContext =
    FacesContext.getCurrentInstance();
Application application =
        facesContext.getApplication();
if (application.getProjectStage().equals(
   ProjectStage.Production){
   //do something
}else if
  (application.getProjectStage().equals(
        ProjectStage.Development){
…
```

Setting the project stage

- To set the project stage a custom JNDI resource must be defined
- In the GlassFish admin console
  - Go to JNDI|Custom resources
  - Click new
  - Enter:

  JNDI Name: java.faces.PROJECT_STAGE

  Resource type: java.lang.String

  Factory Class:
  com.sun.faces.application.ProjectStageJndiFactory
  - Add a new property “stage” set its value to “Development” and click Save