Java EE Enterprise Architecture

The Web Tier

JSF validators and converters

Introduction

- Last lecture we…
  - introduced Java EE
  - discussed Java EE components and containers
  - looked at a simple JSF example

- In this lecture we will…
  - Give a bit more detail about JSF
  - Review JSF validators and converters

JavaServer Faces (JSF)

- JSF is a server-side component framework for building Java-based web applications

- JSF technology consists of the following:
  - An API for…
    - representing components and managing their state
    - handling events
    - server-side validation
    - server-side data conversion
    - defining page navigation
    - supporting internationalization and accessibility
  - Tag libraries for adding components to web pages and for connecting components to server-side objects

JSF components

- A typical JavaServer Faces application includes:
  - Web pages
    - in which UI components are laid out
    - XHTML files called Facelets
    - use the Expression Language (EL) to link to managed beans
  - Tag libraries
    - for adding UI components to web pages
  - Managed beans
    - lightweight container-managed objects (POJOs)
    - act as backing beans with properties and functions for UI components on page

JSF components

- A typical JavaServer Faces application includes:
  - A web deployment descriptor (web.xml file)
  - One or more application configuration resource files
    - e.g. faces-config.xml file, which can be used to
      - define page navigation rules
      - configure beans and custom components
  - Custom objects
    - created by the application developer and can include:
      - custom components
      - validators
      - converters
      - listeners

JSF benefits

- Clean separation of behaviour from presentation for web applications
  - finer-grained separation than is traditionally offered by client-side UI architectures
- Separation of logic from presentation
  - also allows each member of a web application development team to focus on a single piece of the development process
  - provides a simple programming model to link the components

The Expression Language (EL)

- Use simple expressions to dynamically access data from JavaBeans components
  - e.g. an `<h2>` tag can surround an EL expression linking to a session-scoped managed bean named `Hello:`
    - `<h2>Hello, #{hello.name}!</h2>`

- The EL is used for the following functions:
  - Deferring or immediately evaluating expressions
  - Setting and getting data in managed beans
  - Invoking methods in managed beans
  - Dynamically perform arithmetic operations

The Expression Language (EL)

- Expressions for immediate evaluation
  - Use the `${}` syntax
  - Evaluated without passing through the usual life-cycle
- Expressions for deferred evaluation
  - Use the `#{}` syntax
  - Evaluated after passing through the usual life-cycle
- Because of JSF's multiphase lifecycle, most EL expressions are deferred
  - We usually want expressions to be validated and converted before being used in the application
  - Therefore, defer evaluation of expressions until the appropriate point in the life-cycle

Managed Beans

- A typical JSF application has at least one managed bean, each of which can be associated with the components used in a particular page

- A managed bean
  - is created with a default constructor
  - has a set of properties
  - has a set of methods that perform functions for a component

- Like all JavaBeans components, a property consists of:
  - a private data field
  - accessor (get) and mutator (set) methods

Managed Beans

- The most common functions that managed bean methods perform include:
  - Validating the data a UI input component
  - Handling an event fired by a UI component
  - Generating values for UI output components
  - Determining the outcome of an action
    - to identify the next page to which the application must navigate

Developing a JSF application

- Stages include
  - Write the managed beans
  - Create web pages using UI component tags
  - Define page navigation

- In what order should these be done?

Conversion and validation in JSF life-cycle

- When submitted form is received…
- Generate a tree of UI components in submitted form
  - Populate tree with request parameter & previous state
  - Perform conversions, then validations
- If valid, update managed bean components
  - Call set methods in the bean, using values from component tree
- Render response
  - if valid
    UI component tree for next view, populate with values from the managed bean, send HTML to browser
    else
    output original view with error messages
  - Save state (in session scope)

JSF standard converters

- Form data in HTML pages are passed as string values in the HTTP request
- Converters change the string values to strongly-typed Java objects (and vice-versa)
  - e.g. int to Integer, double to Double, Boolean to Boolean, etc.
  - See Oracle (2014), Table 11-1, Section 11-1
- Automatic conversion of primitive data types
  - Determined from the property type in the managed bean that is bound to the UI component
- JSF provides a set of converters
  - Gives the developer some flexibility in the conversion

JSF converter messages

- Each of the standard converters has a standard error message
- If the conversion fails, the standard error message is displayed
  - e.g. `j_idt11:numsibs: 'r' must be a number between -2147483648 and 2147483647 Example: 9346`

Overriding error messages

- The standard error message can be overridden:

```
<h:inputText id="numsibs"
             title="Number of siblings: "
             value="#{user.numberOfSiblings}"
             required="true"
             requiredMessage="My required message"
             converterMessage="My converter message"
             maxlength="2" >
</h:inputText>
```

Invoking JSF converters

- To use a converter, it must be registered to a component using one of the following:
  - Bind the component's value to a backing bean property (most common technique)
    - e.g. `value=#{item.quantity}`
  - Nest the standard converter tag inside the component's tag
    - `f:convertDateTime` or `f:convertNumber`
  - Nest an `f:converter` tag inside the component's tag
  - Use the component's `converter` attribute to refer to the ID of the converter class

(Oracle, 2014: section 11-2)

f:convertDateTime tag

- Nest the tag inside the component's tag

- An example:

```
<h:inputText id="dob"
             title="Date of birth: "
             value="#{user.dateOfBirth}"
             required="true"
             requiredMessage="My required message"
             converterMessage="My converter message"
             maxlength="10" >
    <f:convertDateTime type="date"
                       pattern="dd/MM/yyyy" />
</h:inputText>
```

f:converter tag

- Nest the tag inside the component's tag
- Used for types where there is no standard converter
  - We must write the custom converter class
- An example:

```
<h:inputText id="dob"
             title="Date of birth: "
             value="#{user.dateOfBirth}"
             required="true"
             requiredMessage="My required message"
             converterMessage="My converter message"
             maxlength="10" >
    <f:converter converterId="calendarConverter" />
</h:inputText>
```

converter attribute

- Use the component tag's converter attribute
- Used for types where there is no standard converter
  - We must write the custom converter class
- An example:

```
<h:inputText id="dob"
             title="Date of birth: "
             value="#{user.dateOfBirth}"
             required="true"
             requiredMessage="My required message"
             converter="calendarConverter"
             converterMessage="My converter message"
             maxlength="10" />
```

Custom converter example

- Make your own notes about the custom converter code
  - Download wk2-02_lecProj.zip
  - Class `CalendarConverter`

JSF standard validators

- The JSF core tag library has a set of standard validators, some of which are:
  - `f:validateDoubleRange`
  - `f:validateLongRange` (also used for `int` values)
  - `f:validateLength`
  - `f:validateRegex`
- Each of the standard validators has a standard error message, which can be overridden

```
<h:inputText id="numsibs"
             value="#{user.numberOfSiblings}"
             validatorMessage="My validator message"
             maxlength="2" >
       <f:validateLongRange minimum="0" maximum="0" />
</h:inputText>
```

Custom validators

- The developer can write custom validators
  - Validator method
    - Invoked using the validator attribute
  - Validator bean
    - Invoked using the validator tag

`validator` attribute

- Use the component tag's `validator` attribute

- An example:

```
<h:inputText id="dob"
             title="Date of birth: "
             value="#{user.dateOfBirth}"
             required="true"
             requiredMessage="My required message"
             validator="#{user.validateFirstName}"
             validatorMessage="My validator message"
             maxlength="10" />
```

f:validator tag

- Nest the tag inside the component's tag

- An example:

```
<h:inputText id="dob"
             title="Date of birth: "
             value="#{user.dateOfBirth}"
             required="true"
             requiredMessage="My required message"
             validatorMessage="My validator message"
             maxlength="10" >
    <f:validator validatorId="userValidator" />
</h:inputText>
```

Custom validators

- Make your own notes about the custom validators
  - Download wk2-02_lecProj.zip
  - See `index.xhtml` and `UserValidator` class

References

- Oracle (2014) *Java Platform, Enterprise Edition: The Java EE Tutorial* [Online] Available from http://docs.oracle.com/javaee/7/tutorial/ [Accessed: 17 Jan 2017]