trainologic

JSF

trainologic

UIComponent

- All JSF Components extend from *UIComponent*.
- We will later discuss PrimeFaces.

- However, let's start with examples for the built-in components and get familiar with general attributes.

trainologic

CommandButton

- The <h:commandButton> tag renders an HTML submit button.
- Special JSF related attributes:
  - action – method-binding EL for an action.
  - actionListener – discussed later.
  - image – an image to be displayed instead of text.
  - immediate – discussed later.
  - rendered – an EL which evaluates to boolean.
  - type – ‘submit’ or ‘reset’.
  - value.

trainologic

InputText

- The <h:inputText> tag renders an HTML input text.
- Special JSF related attributes:
  - conveter – converter id.
  - immediate – discussed later.
  - rendered – an EL which evaluates to boolean.
  - required – built-in validation.
  - value.
  - valueChangeListener – discussed later.

trainologic

Messages

- The <h:message> and <h:messages> tags display messages that were registered to the *FacesContext*.
- <h:message for=“”> will only display messages associated to a specific component.

trainologic

Select Boxes

- JSF provides several components for displaying menus, checkboxes, radio buttons and list boxes.
- Before describing those, let's discuss how to specify the values that will appear inside.

- By using: <f:selectItem> and <f:selectItems> tags.

trainologic

SelectItem

- The <f:selectItem> tag provides an option value for the container component.
- Example:

```
<h:selectManyCheckbox id="emails" value="#{mailing.emails}">
   <f:selectItem id="item1" itemLabel="John's mail"
         itemValue="john@x.com" />
   <f:selectItem id="item2" itemLabel="Jane's mail"
         itemValue="jane@x.com" />
   <f:selectItem id="item3" value="#{admin.email}" />
</h:selectManyCheckbox>
```

trainologic

SelectItems

- The <f:selectItems> tag provides the entire set of options for the enclosing component.
- Example:

```
<h:selectManyCheckbox id="emails" value="#{mailing.emails}">
   <f:selectItems value="#{configuration.emails"} />
</h:selectManyCheckbox>
```

trainologic

selectItems

- The EL in the *value* attribute of this tag must evaluate to one of the following types:
  - A SelectItem object.
  - An array of SelectItem objects.
  - A collection of SelectItem objects.
  - A Map where the keys are converted to Strings and serve as labels and the values are converted to Strings and serve as values for the newly created SelectItem objects.

trainologic

dataTable

- The <h:dataTable> tag renders an HTML ‘table’ element.
- Inside you can put the <h:column> element defining a column.

- Let’s see an example…

trainologic

Example

- Example:

```
<h:dataTable id="customers"
         columnClasses="greyColumn,greenColumn"
         headerClass="headerStyle"
         rowClasses="rowStyle" styleClass="backgourndClass"
         value="#{compnay.customers}" var="customer">
  <h:column>
    <f:facet name="header">
      <h:outputText  value="#{msg.customerNameLabel}"/>
    </f:facet>
     <h:outputText value="#{customer.name}"/>
  </h:column>
  <h:column>
    <f:facet name="header">
     Company
    </f:facet>
     <h:outputText value="#{customer.company}"/>
  </h:column>
</h:dataTable>
```

trainologic

Discussion

- A table will be rendered with the number of rows equals to the length of the collection in the value attribute.
- The current element will be stored in the var attribute.
- The EL in the value attribute can be of any type and is converted subject to the following rules:
  - An instance of *DataModel* is used directly.
  - Array of Object, List, ResultSet, jstl.Result are wrapped in an appropriate DataModel object.
  - Everything else is wrapped in a *DataModel* as a single row.

trainologic

Facets

- As you may have noticed, the <h:column> element contained the <f:facet> tag.
- Facets are supported by some JSF components.
- Facets are special nested attributes that do not follow the parent-child relationship.
- A facet always have a name associated with it and the containing component must know what to do with this specific facet.

trainologic

Facets

- For example, the <h:column> component knows about the header and footer facets and will display their values in the table header/footer respectively.
- In order to know which facets are supported and for which component, you'll need to consult the documentation.

trainologic

panelGrid

- The <h:panelGrid> tag renders an HTML <table> element.
- Unlike <h:dataTable> the panelGrid component is used to specify the content directly on the page and not through a managed bean.
- By default, each nested component is displayed in a separated column.
- You can alter the behavior by using the <h:panelGroup> element.
- Example…

trainologic

Example


```
<h:panelGrid id="panel" columns="2">
  <f:facet name="header">
    <h:outputText value="#{msgs.productsSearchTitle}"/>
  </f:facet>
  <h:outputLabel for="productName" value="#{msgs.productName}" />
  <h:inputText id="productName" value="#{query.productName}" />
  <h:outputLabel for="filter" value="#{msgs.filter}" />
  <h:inputText id="filter" value="#{query.filter}" />
  <f:facet name="footer">
    <h:panelGroup>
      <h:commandButton id="submit" value="#{msgs.submit}" />
    </h:panelGroup>
  </f:facet>
</h:panelGrid>
```

trainologic

Bean Validation

- A specification represented by JSR 303.
- Provides a way to describe validations as meta-data (annotations or XML).
- We will focus on annotations.
- The reference implementation is the Hibernate Validator project.

trainologic

Hello World

- A simple example for Bean Validation:

```
public class Person {

    @NotNull
    private String name;

    @Min(18)
    @Max(120)
    private int age;

    @Past
    private Calendar birthDate;
```

trainologic

Performing the Validation

- Ok, so what do these annotations do?
- Well, as always, annotations do nothing in Java. They are just meta-data.
- However, you can use the Bean Validation API to perform validations on annotated bean.
- We will also discuss when it will happen automatically in JSF.

trainologic

Performing the Validation

- Example:

```
Person p = new Person();
// name is null
p.setAge(70);
Calendar nowPlus5Years = Calendar.getInstance();
nowPlus5Years.add(Calendar.YEAR, 5);
// birthDate is in the future
p.setBirthDate(nowPlus5Years);

ValidatorFactory factory =
        Validation.buildDefaultValidatorFactory();
Validator validator = factory.getValidator();
Set<ConstraintViolation<Person>> results = validator.validate(p);
for (ConstraintViolation<Person> constraintViolation : results) {
        System.out.println(constraintViolation);
}
```

trainologic

Discussion

- As you can see, the validation API returns all of the constraint violations.
- Contain the error message, message key, the invalid bean and the cause.

- Let's now go through the standard supported constraints:

trainologic

Built-in Constraints

- @AssertFalse – checks that the field/property is false.
- @AssertTrue.
- @DecimalMax – checks that the field/property (must be one of: *BigDecimal, BigInteger, String, byte, short, int, long* or wrappers) is lower or equal to the annotation's attribute.
- @DecimalMin.
- @Digits – Works on the same types @DecimalMax and checks for a defined maximum number of digits and fractional digits.

trainologic

Built-in Constraints

- @Future – checks whether the field/property is in the future.
- @Past.
- @Max – like @DecimalMax, only that the field must be a number and not a representation (e.g., String).
- @Min.
- @NotNull.
- @Null.
- @Pattern – checks that the value matches a regular expression.

trainologic

Built-in Constraints

- @Size – checks whether the element's size (*String*, collection, *Map* or array) is between a specified range.
- @Valid – recursively performs validations on the annotated field/property.

trainologic

Creating Custom Constraints

- You can create custom constraints.
- Let's create a constraint to validate that a *String* field/property is n IP.
- First, we create the annotation:

```
// supports fields/properties
@Target( { ANNOTATION_TYPE, FIELD, METHOD })
@Retention(RetentionPolicy.RUNTIME)
@Constraint(validatedBy = CheckIP.class)
@Documented
public @interface IP {
   String message() default "{com.trainologic.checkip}";
   Class<?>[] groups() default {};
   Class<? extends Payload>[] payload() default {};
   // should support localhost (127.0.0.1)
   boolean value() default true;
}
```

trainologic

The Validating Class

- Now, the class containing the validation logic:

```
public class CheckIP implements ConstraintValidator<IP, String>
{

   private boolean supportsLocalHost;
   private static final String LOCALHOST = "127.0.0.1";

   @Override
   public void initialize(IP constraintAnnotation) {
         supportsLocalHost = constraintAnnotation.value();
   }

   @Override
   public boolean isValid(String value,
     ConstraintValidatorContext constraintValidatorContext) {
        if (value == null)
           return false;
```

trainologic

The Validating Class

```
            String[] ipSegments = value.split("\\.");
            if (ipSegments.length != 4) {
               return false;
            }
            for (String string : ipSegments) {
               try {
                     int val = Integer.parseInt(string);
                     if (val < 0 || val > 255) {
                        return false;
                     }
               } catch (NumberFormatException e) {
                     return false;
               }
            }
            if (supportsLocalHost == false &&
                     LOCALHOST.equals(value)) {
               return false;
            }
            return true;
      }
}
```

trainologic

Using Groups

- By default, when you perform validation, all the constraints are being checked.
- You can control this behavior by introducing validation groups.
- All you need to do, is create a “marker” interface and bind the constraint annotations to it.
- Let’s see an example…

trainologic

Example

- Our marker interface:

```
public interface NullChecks {
}
```

- Usage:

```
public class Person {
    @NotNull(groups = NullChecks.class)
    private String name;
```

trainologic

Discussion

- Now the default validation will not validate the *name* for *null.*
- In order to use the group, you need to specify explicitly:

```
Person p = new Person();
// name is null

ValidatorFactory factory =
        Validation.buildDefaultValidatorFactory();
Validator validator = factory.getValidator();
Set<ConstraintViolation<Person>> results = validator.validate(p,
        NullChecks.class, Default.class);
for (ConstraintViolation<Person> constraintViolation : results) {
        System.out.println(constraintViolation);
}
```

trainologic

Listeners

- There are 3 types of listeners in JSF:
  - Phase listeners – we've already discussed those.
  - Action listeners – listeners that receive *ActionEvent.*
  - Value change listeners – listeners that receive *ValueChangeEvent.*

trainologic

Action Listeners

- Some UI components (e.g., CommandButton) have action and actionListerner attributes.
- Both are used for binding providing Action Listeners.
- Although there is no restriction upon the name of the method bound for these attributes, there is a subtle difference between them:
- An actionListener bound method must have a return type of void (i.e., it does not affect the page flow) while the action bound method can return a String type.

trainologic

Action Listeners

- If both attributes are specified, the actionListener bounded method will be invoked first.
- You can also bind more action listeners by using the <f:actionListener> tag.
- The only argument an action listener method can have must be of type: ActionEvent.
- Of course, those methods are invoked in the ‘Invoke Applications’ phase.

trainologic

Value Change Listeners

- Value-change event listeners can be defined for UIInputs either by attribute or nested tag.
- Methods which are bound by the attributes need to return void and accept a single argument of type: ValueChangeEvent.
- Note that this event is fired in the ‘Process Validations’ phase.
- Remember that at this phase, the managed beans do not contain the current request’s values.

trainologic

The Immediate Attribute

- Some JSF components have the ‘immediate’ property.
- When this property is set to true, the component has a slightly different lifecycle.
- This property appears in 2 JSF interfaces: *EditableValueHolder* and *ActionSource/2.*
- For example, *UIInput* implements *EditableValueHolder* and *UICommand* implements *ActionSource2*.

trainologic

Immediate UIInput

- Components that implement the *EditableValueHolder* and have their 'immediate' property computed to *true,* behave as follow:
- In the 'Apply Request Values' phase, these components will also perform their validations (instead of in the 'Process Validations' phase).

- Let's see this in the source code…

trainologic

Immediate UIInput

- *UIInput*'s *processDecodes()* method, which is invoked during the 'Apply Request Values' phase:

```
public void processDecodes(FacesContext context) {
        if (context == null) {
            throw new NullPointerException();
        }

        // Skip processing if our rendered flag is false
        if (!isRendered()) {
            return;
        }
        super.processDecodes(context);
        if (isImmediate()) {
            executeValidate(context);
        }
}
```

trainologic

Immediate UICommand

- Components that implement the *ActionSource/2* interface and have their ‘immediate’ property set to *true* behave as follow:
- They will register their actions to the end of the ‘Apply Request Values’ phase instead of the ‘Invoke Applications’ phase.

- Again, let’s see it in the source code…

trainologic

Immediate UICommand

- *UICommand's queueEvent()* method:

```
public void queueEvent(FacesEvent e) {
        UIComponent c = e.getComponent();
        if (e instanceof ActionEvent && c instanceof ActionSource) {
            if (((ActionSource) c).isImmediate()) {
                e.setPhaseId(PhaseId.APPLY_REQUEST_VALUES);
            } else {
                e.setPhaseId(PhaseId.INVOKE_APPLICATION);
            }
        }
        super.queueEvent(e);
    }
```

trainologic

Flash Scope

- Flash scope is a scope that includes the current request and the next.
- Can be accessed by *FacesContext.getExternalContext().getFlash().*
- From the EL can be accessed with the *flash* implicit object.
- Can be used in a Post-Redirect-Get pattern.

trainologic

Ids

- You can specify a unique id for the component by using the id attribute.
- Note that this is a server-side component id.
- It must be unique within its naming container (e.g., form).
- The rendered HTML will display a client-side ids which are globally unique in the page.
- You can retrieve the client side id by invoking the getClientId() method on the UIComponent.

trainologic

Component Binding

- All the JSF component tags allow you to specify an EL expression in their binding attribute.
- When you specify an EL in this attribute the component itself will be placed in the specified managed bean property.