duckcreek technologies

Exercise: Product Definition
TDF Validations

This lab builds on a previous lab where we only added fieldElements to the display. Now we'll enforce some rules using the validations framework.

Java-Based Labs

This exercises the ability to add a custom Java-based field Validation.

The APDataCollection APIs and IdIterator class as described in the earlier AXE module are also used as part of this lab; if necessary, refer back to that module for details on how to use these components.

Specification for the new field Validator

For this exercise we're going to create a custom class which only allows the "Prior Carrier Information/Loss Information" > "Add Losses" > "Number of Claims" flag to be greater than 3 if there's at least one Location for the state of Massachusetts (StateProvCd=MA); if there are no MA locations, the Number of Claims value must be <= 3.

Use the Eclipse wizards for creating new Java packages and classes (File > New > Other > Java …)

If you are not familiar with these tools, your instructor will demonstrate their usage.

1. Create a new Validator class
    - Package (create it):

      `com.sdktraining.workerscomp.validators`

    - Class:

      `NumClaimsValidator`

        - Be sure you extend the recommended Base class for a custom Validator, as described in the presentation

Provide implementations for the required (abstract) methods as defined in the base class…

- The `getType()` method indicates what type of validator this class implements

- This corresponds to the type="custom" attribute in the TDF (XML) field `validator` element
- For custom validators, the appropriate return value is literally "custom"
- The SDK provides a symbol for this value, so we'll use that symbol instead of the hard-coded String:

```
return IValidator.CUSTOM_VALIDATOR_TYPE;
```

- The `getDefaultErrorMessageTemplate()` defines the message text (String) that will be displayed to the user when an invalid data value is detected
  - It is a "template" because it is expected to include placeholders for context-specific information in the form `${<variable>}`
  - The return value should *always* use `${field_label}` to reference the field with invalid data

```
return "${field_label} is not valid at this time.";
```

- The `getVariableMapForErrorMessageTemplate()` method defines the context-specific values for `${<variable>}` references in the message template
  - The base class provides a method that defines the current `${field_label}` value

```
return
createStartingMapWithFieldLabelVariable(validationContext);
```

  - We'll see how to add custom values to this map in a later exercise
- The `appendJavaScriptValidationBody()` method defines the client-side javascript expression used to validate the field in-page
  - As described in the presentation, client-side validations occur *immediately** after a user updates a field value
    - vs. server-side validations, which occur as part of the page submit processing (after the Continue action)

    * NOTE for text fields, client-side validation occurs when the user tabs out of the field or the field loses focus (waiting until all of the text has presumably been entered in the field)

- Initially, we will (in effect) have no client-side validation- the "real" validation will only occur server-side

```
jsmi.getMethodBodyWriter().addContent("return true;");
```

- The `isValid()` method is where we do the work of validating the Number of Claims field value
  - The `ValidationContext` parameter provides a reference to the field that this validation is associated with and is useful in obtaining information about the calling field, such as it's label and current value.
  - As the method name implies, a return value of `true` is used to indicate the field has a valid value; a return value of `false` is used to indicate the field has an invalid value
  - Access the field's value using the `ValidationContext` parameter `getValue()` method
  - Access the `APDataCollection` via the `ValidationContext` parameter `getApData()` method
  - Check the workitem data for any Location in MA (Massachusetts)
    - Invoke `createIndexTraversalBasis()` on the APDataCollection
    - The XPath to the State abbreviation is: `Location.Addr.StateProvCd`
    - *Note: The* `APDataCollection` *has the data that has already been saved and committed; if we needed data from the current page we'd use the* `HTMLDataContainer`
  - Return the appropriate validity indicator based on the combination of 1) the Number of Claims field value and 2) the presence or absence of any MA Location as described in the Specification (above)

2. Configure the "Number of Claims" field in the TDF to use this new custom Validator
3. Test your validator to insure it conforms to the specification
   - Your test cases *must* include multiple Locations to be complete
   - Test with no MA Locations
   - Test with at least one MA Location…
     - …in the first Location

- …in any of Locations 2..n, but *not* in the first

NOTE the first & 2..n are references to the ordinal position of the Locations within the workitem XML, not (necessarily) their order on the page

Don't forget to test your code by doing an "Add" or "Save" when the Number of Claims field is blank. Since it's an optional field the empty value must be accounted for (allowed as valid) in your isValid implementation.

Advanced Exercise #1

- Declare class-scoped variables to store the conditions used to test for validity

```
… stateProvCdWithNoLimit="MA";
… numberOfClaimsNormalLimit=3;
```

  - This simulates the concept of the "variable" nature of how things actually work

- Modify your `isValid()` implementation to use these variables instead of the hard-coded values

- Modify your `getDefaultErrorMessageTemplate()` implementation to define placeholders for the (simulated) variable values

```
return "The ${field_label} field cannot exceed ${max_claims} when
there is not at least one Location for ${state}.";
```

- Modify your `getVariableMapForErrorMessageTemplate()` implementation to provide the context-specific values for the new message template variables

  - The base class's `createStartingMapWithFieldLabelVariable()` will fill in the field_label variable value, but the rest is up to you

  - If you're not familiar with the Java SE `Properties` class, it's just a glorified `Map`

Retest to see the new message.

Advanced Exercise #2

- There are at least three other ways to execute the check to see if there's at least one MA `Location`. Implement one or more of them for the practice:

  - Using the `IdIterator` / `ElementPathExpression` classes

  - Using the `getCount()` method

- Using the `exists()` method

Advanced Exercise #3

- As a purely academic exercise, change the `appendJavaScriptValidationBody()` (client-side validation) implementation to

```
jsmi.getMethodBodyWriter().addContent("return false;");
```

  - You'll need to navigate to a different page and then come back in order to get the updated javascript expression to the browser
  - Update the Number of Claims field value to invoke the new client-side validation
  - What happens? After seeing what happens can you think of a wellimplemented version of this method could be helpful?