ADF Code Corner

003. Advanced Expression Language Techniques

twitter.com/adfcodecorner

Abstract:

Expression Language is a dot notated scripting format that allows you to access in memory and Java objects. In JavaServer Faces, you use Expression Language to to bind UI components to managed beans or to the Oracle ADF binding layer. Using EL from a page or page fragment in Oracle JDeveloper 11*g* is as easy as opening the Expression Builder dialog to select the object to reference. However, usecases exist that need you to work with Expression Language in Java. For example, developers that create JavaServer Faces component instances dynamically at runtime need to apply Expression Language to the component to access the binding layer to read or write data. This blog article focuses on how to work with Expression Language in Java.

Author: Frank Nimphius, Oracle Corporation
twitter.com/fnimphiu
21-MAY-2009

Introduction

Before getting started with the topic, using Expression Language in Java should be an exception. There are use cases in ADF that require EL to be used in Java. For exmple, the "row" variable that is used to populate a table with data is only accessible in EL. So to implement conditional row formatting when the table renders, you must use EL to access the current row values. Another usecase that requires EL in Java is to programmatically create a method binding on a component.

Accessing the ADF binding layer from Expression Language also is ..... **not** a good usecase. You can access the binding layer from the BindingContext object, which provides a native Java access.

This blog entry is a take out from the Oracle Press "Oracle Fusion Developer Guide: Building Rich Internet Applications with Oracle ADF Business Components and ADF Faces" book that Lynn Munsinger and I wrote for McGraw Hill.

Advanced EL techniques

There exist two types of expressions in EL value expressions and method expressions. Value expressions address a bean property, expecting the provided expression to resolve to a valid pair of getter and setter methods. Method expressions reference methods that are exposed on an EL accessible object.

The following classes are used when working with expressions from Java in JavaServer Faces

ELContext The ELContext class is used as an argument to the ExpressionFactory class methods to create value and method bindings. It also exposes the ELResolver instance, which can be used to resolve model properties.

JSF uses the ELResolver instance to resolve references like #{managedBean.firstname}. To access an instance of ELContext from a managed bean in JSF, you use

```
FacesContext fctx = FacesContext.getCurrentInstance();
ELContext elctx = fctx.getELContext();
```

ExpressionFactory The ExpressionFactory class exposes the createValueExpression method to create a Java handle to a value expression and the createMethodExpression method that creates a Java handle to a method. The ExpressionFactory instance is accessible in JavaServer Faces as follows

```
FacesContext fctx = FacesContext.getCurrentInstance();
ELContext elctx = fctx.getELContext();
Application jsfApp = fctx.getApplication();
ExpressionFactory exprFactory = jsfApp.getExpressionFactory();
```

How to programmatically create a component value reference

ADF Faces Rich Client components can be created dynamically in Java and added to a page. To bind the component to the ADF binding layer, or to a managed bean property, a ValueExpression reference is created for the *value* attribute of the component. The example below creates a new InputText field and binds it to the "DepartmentName" attribute in the ADF binding layer. The Java import statements for this example are listed for completeness

```
//imports
import javax.el.ELContext;
import javax.el.ExpressionFactory;
import javax.el.ValueExpression;
import javax.faces.application.Application;
import javax.faces.context.FacesContext;
import oracle.adf.view.rich.component.rich.input.RichInputText;
…
//create a new ADF Faces input text component instance
RichInputText inputText = new RichInputText();
inputText.setLabel("Department Name");
FacesContext fctx = FacesContext.getCurrentInstance();
ELContext elctx = fctx.getELContext();
Application jsfApp = fctx.getApplication();
//create a ValueExpression that points to the ADF binding layer
ExpressionFactory exprFactory = jsfApp.getExpressionFactory();
ValueExpression valueExpr = exprFactory.createValueExpression(
                             elctx,
                             "#{bindings.DepartmentName}",
                              Object.class
                             );
//add the expression to input text value property
inputText.setValueExpression("value",valueExpr);
//add the component to the PanelFormLayout to show in the page
getPanelForm().getChildren().add(inputText);
//refresh the container to show text field
AdfFacesContext.getCurrentInstance().addPartialTarget(getPanelForm());
```

How to programmatically access ADF binding values

Expression Language can also be used to access the ADF binding to read and write values, like it can be used to access any other managed bean property. The example below accesses the "DepartmentId" attribute binding of the ADF binding layer to read the current department id value.

```
FacesContext fctx = FacesContext.getCurrentInstance();
ELContext elctx = fctx.getELContext();
Application application = fctx.getApplication();
ExpressionFactory exprFactory = application.getExpressionFactory();
ValueExpression valueExpr = exprFactory.createValueExpression(
                              elctx,
                              "#{bindings.DepartmentId.inputValue}",
                              Object.class);
oracle.jbo.domain.Number departmentId = null;
departmentId = (oracle.jbo.domain.Number) valueExpr.getValue(elctx);
```

Using ADF Business Components as a business service, the ADF binding reference returns oracle.jbo.domain object types for attributes that use ADF Business Component special types like Number, Date and DBSequence.

To write a value back to the binding layer, use the setValue method of the ValueExpression.

```
valueExpr.setValue(elctx, <object>);
```

Note: You can access the ADF binding layer without using EL. For this call BindingContext.getCurrent() and getCurrentBindingsEntry(). You can then access all bindings by their ID. Personally, I prefer using the BindingContext object as it means less code to write.

How-to programmatically create a command action reference

The example below shows how to dynamically create a command component in Java and reference a method exposed on the ADF binding layer. The command button that is added executes the "Delete" operation that exist on a View Object and that is exposed on the ADF binding layer of the current JSF page. The binding layer definition, used in this example shows as follows

```
<action IterBinding="EmployeesView3Iterator" id="DeleteEmployee"
  InstanceName="HRAppModuleDataControl.EmployeesView3"
  DataControl="HRAppModuleDataControl" RequiresUpdateModel="false"
  Action="removeCurrentRow"/>
```

The action binding is defined with its *id* attribute set to "DeleteEmployee". The "DeleteEmployee" string is used to access the operation in the binding layer from Expression Language and execute it. The managed bean Java code that creates and configures the command button shows as follows

```
//Create the ADF Faces command button
RichCommandButton deleteButton = new RichCommandButton();
deleteButton.setText("Delete");
//Use partial submit so the page is not fully reloaded
//when pressing the button
deleteButton.setPartialSubmit(true);
//Obtain handle to ExpressioFactory
FacesContext fctx = FacesContext.getCurrentInstance();
ELContext elctx = fctx.getELContext();
Application jsfApp = fctx.getApplication();
ExpressionFactory exprFactory = jsfApp.getExpressionFactory();
//Create and add method expression that references the ADF
//binding layer to execute the operation
```

```
MethodExpression methodExpr = null;
//create method expression that doesn't expect a return type (null),
//and that has no parameters to pass (new Class[]{})
methodExpr = exprFactory.createMethodExpression(
               elctx,
               "#{bindings.DeleteEmployee.execute}",
               null,
               new Class[]{});
//add the expression to the button's action attribute
deleteButton.setActionExpression(methodExpr);
panelForm.getChildren().add(deleteButton);
AdfFacesContext.getCurrentInstance().addPartialTarget(getPanelForm());
```

How-to programmatically create component listeners

Beside of using a method expression to define a reference to a command button *action* attribute, the same can be used to reference listener implementations, like ActionListener or ValueChangeListener. Listener methods that are configured in a managed bean take a single argument, which is an instance of the event object, like ActionEvent and ValueChangeEvent, for the developer to work with. An ActionListener method in a managed bean looks as follows

```
 public void handleButtonPressed(ActionEvent event){
  //add code here
}
```

To dynamically reference this action from an existing command component, or a newly created component, use the following code below, assuming the handle to the command component instance is "commandButton"

```
FacesContext fctx = FacesContext.getCurrentInstance();
ELContext elctx = fctx.getELContext();
Application application = fctx.getApplication();
ExpressionFactory exprFactory = application.getExpressionFactory();
//Create method expression that expects an ActionEvent argument to
//be passed in
MethodExpression methodExpr =  null;
methodExpr  = exprFactory.createMethodExpression(
                elctx,
                "#{CustomBean.handleButtonPressed}",
                null,
                new Class[] {ActionEvent.class});
//Create a new ActionListener based on a method expression
MethodExpressionActionListener actionListener = null;
actionListener = new MethodExpressionActionListener(methodExpr);
//add listener to the commandButton instance
commandButton.addActionListener(actionListener);
```

Similar, to add a ValueChangeListener to an input component, like InputText, that invokes the "handleValueChange" method in a managed bean to handle the value change event

```
MethodExpression methodExpr = null;
```

```
methodExpr = exprFactory.createMethodExpression(
              elctx,
             "#{ CustomBean.handleValueChange}",
              null,
              new Class[] {ValueChangeEvent.class});
MethodExpressionValueChangeListener changeListener = null;
changeListener = new MethodExpressionValueChangeListener(methodExpr);
inputText.addValueChangeListener(changeListener);
```

The “handleValueChange” method signature in the managed bean looks as follows

```
public void handleValueChangeEvent(ValueChangeEvent event){
  //add code here
}
```

How-to access managed beans in Java using EL

If a managed bean method requires access to an instance of another managed bean then one option is to reference the managed bean in a managed bean property. Another option is to reference the managed bean instance using Expression Language, as shown below

```
FacesContext fctx = FacesContext.getCurrentInstance();
ELContext elctx = fctx.getELContext();
Application jsfApp = fctx.getApplication();
ExpressionFactory exprFactory = jsApp.getExpressionFactory();
ValueExpression valueExpr = exprFactory.createValueExpression(
                             elctx,
                            "#{EmployeesBean}",
                             Object.class);
EmployeesBean employees = (EmployeesBean) vExpr.getValue(elctx);
```