ADF Code Corner

Oracle JDeveloper OTN Harvest 05 / 2011

twitter.com/adfcodecorner

Abstract:

The Oracle JDeveloper forum is in the Top 5 of the most active forums on the Oracle Technology Network (OTN). The number of questions and answers published on the forum is steadily increasing with the growing interest in and adoption of the Oracle Application Development Framework (ADF).

The ADF Code Corner "Oracle JDeveloper OTN Harvest" series is a monthly summary of selected topics posted on the OTN Oracle JDeveloper forum. It is an effort to turn knowledge exchange into an interesting read for developers who enjoy harvesting little nuggets of wisdom.

http://blogs.oracle.com/jdevotnharvest/

Author: Frank Nimphius, Oracle Corporation
twitter.com/fnimphiu
30-MAY-2011

Oracle ADF Code Corner OTN Harvest is a monthly blog series that publishes how-to tips and information around Oracle JDeveloper and Oracle ADF.

Disclaimer: ADF Code Corner OTN Harvest is a blogging effort according to the Oracle blogging policies. It is not an official Oracle publication. All samples and code snippets are provided "as is" with no guarantee for future upgrades or error correction. No support can be given through Oracle customer support.

If you have questions, please post them to the Oracle OTN JDeveloper forum:
http://forums.oracle.com/forums/forum.jspa?forumID=83

May 2011 Issue – Table of Contents

About the train button bar skip behavior

The **skip** property of a train stop, a view activity or task flow activity, in a bounded task flow has different meanings dependent on the ADF Faces component that is used to handle the train navigation. Using the `af:train` component, the **skip** property disables the train stop, which means users cannot navigate to this view and instead would continue by clicking on another train stop.

Another train stop rendering component in ADF Faces is `af:trainButtonBar and` displays the train with two buttons, one for forward and one for backward navigation. If the **skip** property is set to "true" for a train stop, then navigation stops on the previous train stop with the train button displayed as disabled.

The behavior of the `af:trainButtonBar` appears strange to ADF developers as it prevents users from continuing with the process they started. The component however works as designed. The train button bar is used for sequential trains, which means that each stop is visited in order. Thus, to hold a process, you set the **skip** property of the following stop to "true".

To temporarily hold a train navigation until data entry meets a certain condition; you dynamically set the **skip** property on the next train stop to true using EL When the condition is met, you dynamically set the **skip** property to false and partially refresh the train button bar so that forward navigation is enabled again (Same of course for backward navigation if you changed a previous stop's **skip** property from false to true. For example, if an account name can only be set once in a process, you could use skip to prevent back navigation to this view by setting **skip** to true). A second option for navigating to the next stop in the "skip = true" case is programmatic navigation, in which the program logic follows a control flow case to the next allowed view in a process. This then also enables the train button, assuming the stop following the next has its **skip** property set to false.

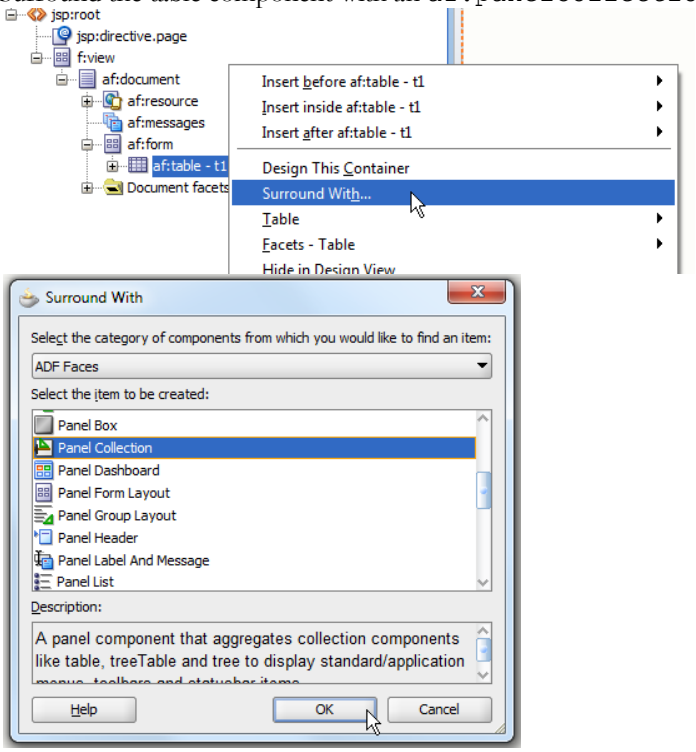
Sample #80 on ADF Code Corner shows an example of how a managed bean extending the `HashMap` class can be used to dynamically set properties on the bounded task flow train stop configurations.

http://www.oracle.com/technetwork/developer-tools/adf/learnmore/index-101235.html

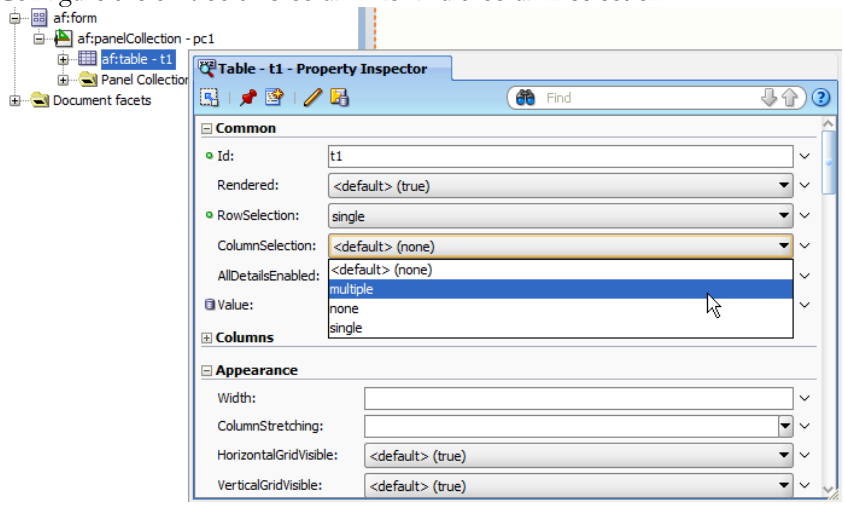
Declarative multi-column sort for ADF bound tables

Enabling sort on an ADF Faces table allows users to sort column data by clicking on an icon in the table header. To sort tables by multiple columns, developers add the `af:panelCollection` component as the parent container of the `af:table` component.

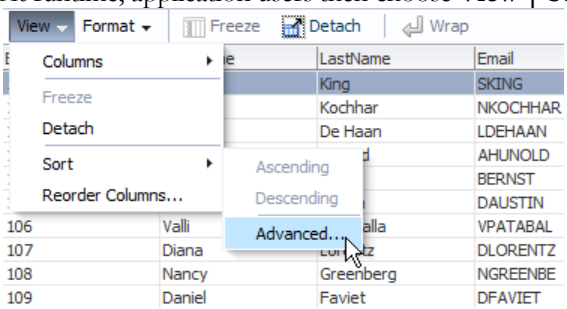
1. Surround the table component with an `af:panelCollection` component



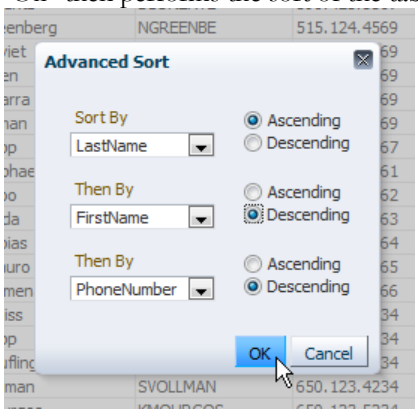
2. Configure the `af:table` column for multi column selection



3. At runtime, application users then choose **View | Sort | Advanced**



4. In the dialog, users select the columns to include in the sort and also the sort direction. Pressing "Ok" then performs the sort of the table data



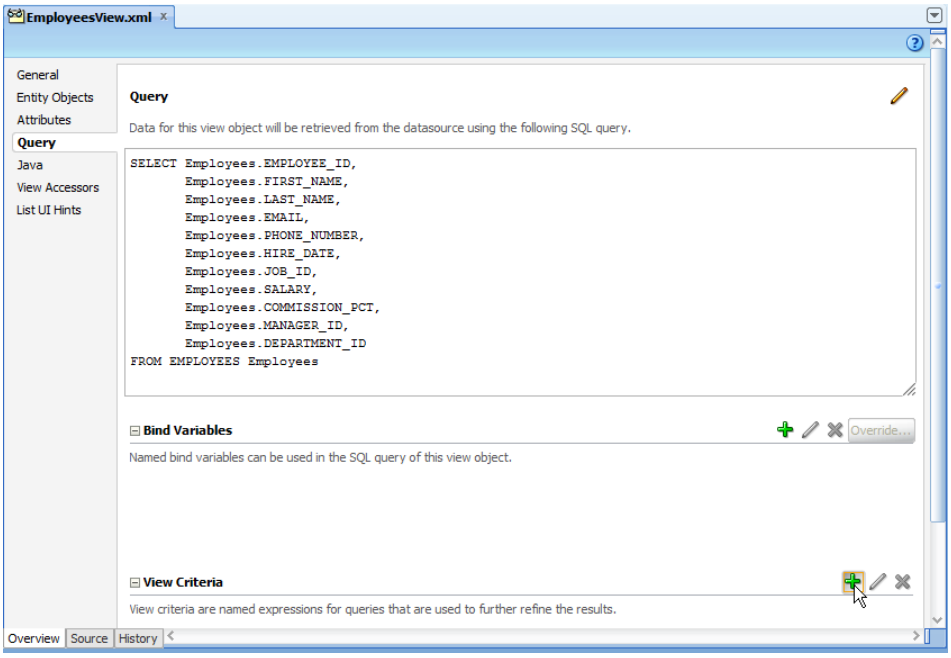
Building a search form from bind variables in a View Criteria

Using the ADF Faces `af:query` component, you can easily build search forms based on **View Criteria** defined on a View Object in ADF Business Components.

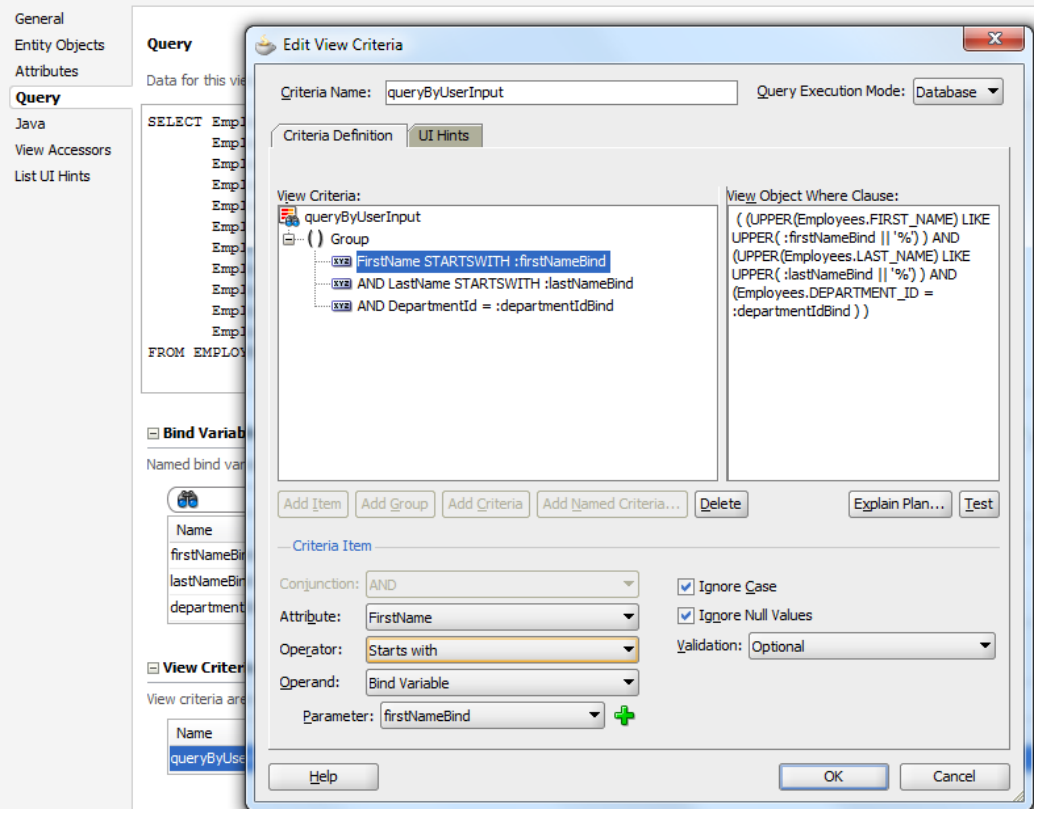
Just for you to recall how to do this: Expand the View Object node that contains the View Criteria in the **Data Controls panel.** Drag and drop the named criteria located in the **Named Criteria** folder to a JSF page and choose an entry of the **Query** or **Quick Query** context menu.

Though configurable to some degree, the search form rendered by the `af:query` component is not as customizable as you need it to be. In this case you need to look at alternative solutions:

One of these alternative solutions is to apply a View Criteria that uses bind variables to filter the View Object query to the View Object instance. The bind variables then are exposed in the **DataControls Panel** and can be set by executing the **ExecuteWithParams** operation.

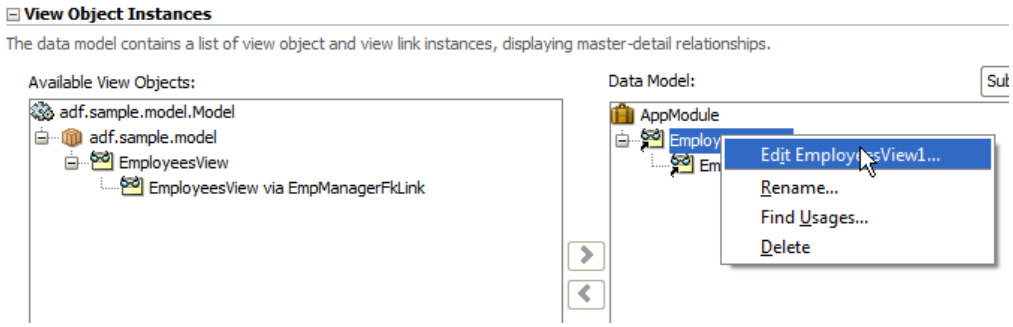


To implement this solution, start by creating a View Criteria for the View Object. The View Criteria will have all attributes to query added and associated to a bind variable.

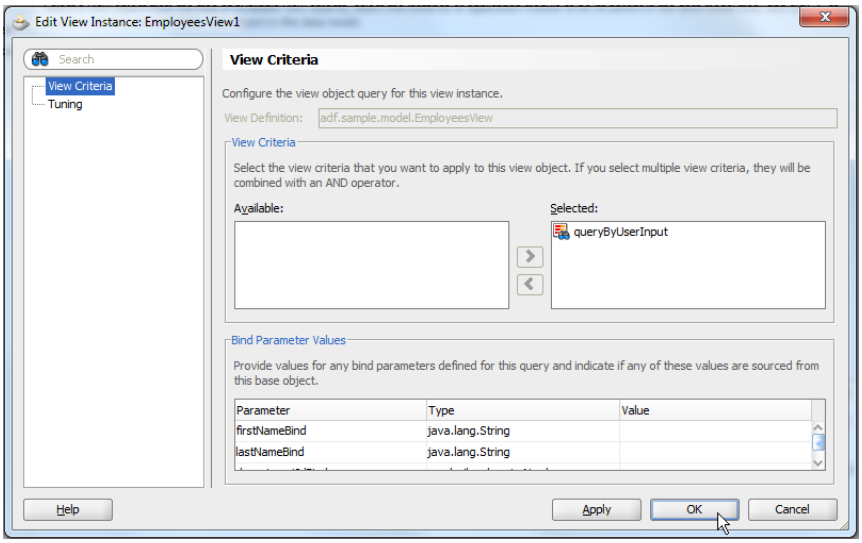


To create an attribute in the View Criteria, select the **Group** node and press the **Add Item** button. Select the attribute to add and set the **Operand** field to **Bind Variable.** Create new bind variables by pressing the green plus icon next to the **Parameter** field and make sure you assign the correct data type for it. Also note that for string parameters you want to set the **Operator** field to **contains**, **endsWith** or **startWith**.

Note that bid variables can have UI hints defined which allows you to set the labels accordingly.

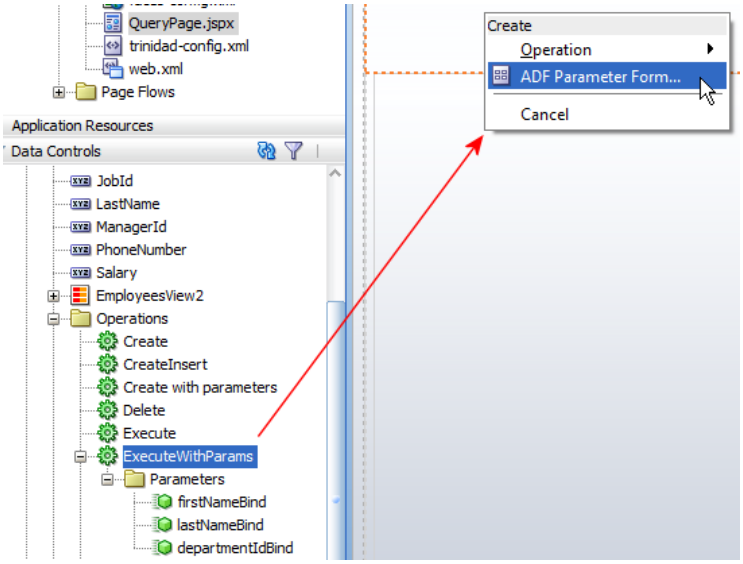


To associate the View Criteria with a View Object instance in the Application Module. Double click the Application Module (**AppModule** in the sample) object in the Application Navigator to open the editor window for it. Choose the **Data Model** category to see View Object instance. Select the View Object instance you want to apply the View criteria to and choose **Edit <VO Name>** from the right mouse context menu.



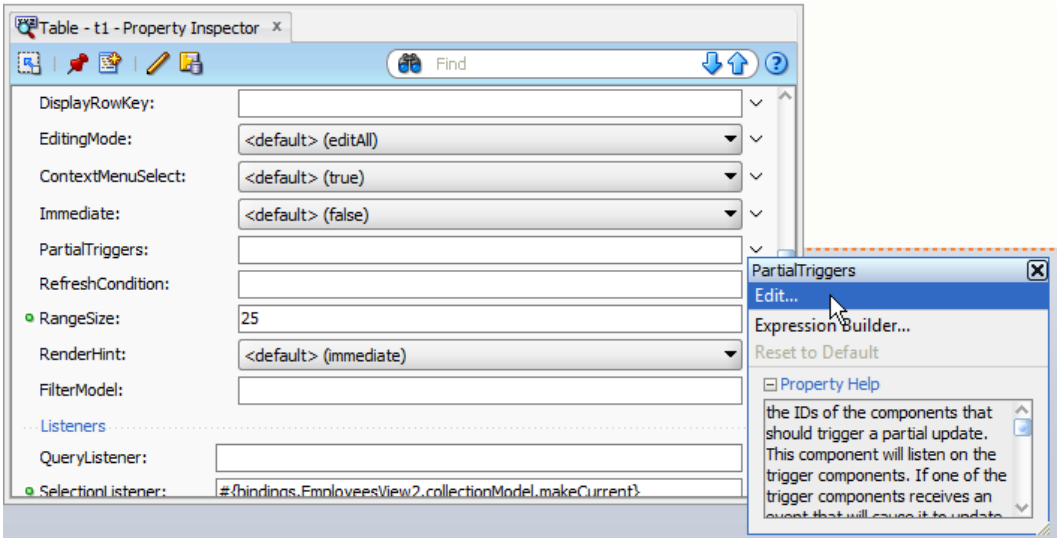
Select the View Criteria and press Ok. You don't need to provide default values for the bind variables.

Open the ADF Faces page that should have the search form defined. In the DataControls panel, expand the **Operations** node of the View Object instance and drag and drop the **ExecuteWithParams** entry as an **ADF Parameter Form**.

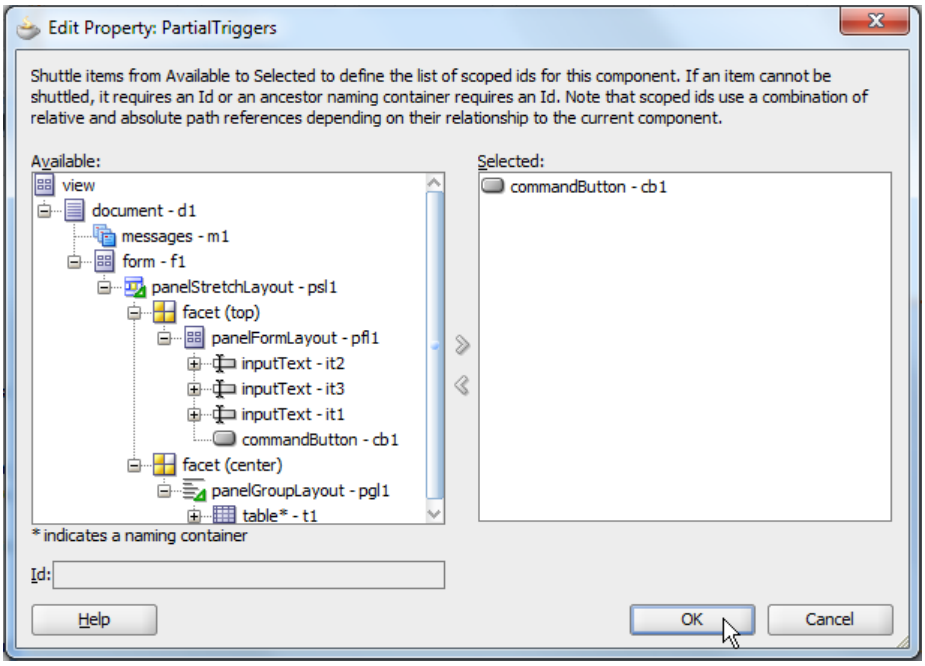


After you created the parameter form, drag and drop the View Object instance itself as a table to the same page.

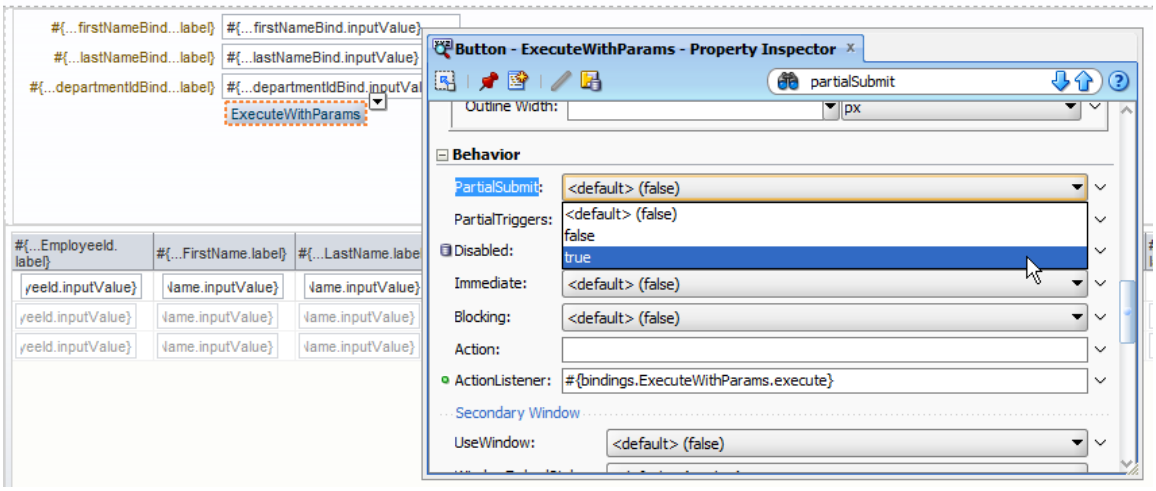
Select the table on the page and open the Property Inspector (ctrl+shift+I) and browse to the **Partial Triggers** property. Use the **Edit** option of the context menu to search for the command button created for the search form.



Referencing the button from the table ensures the table to be refreshed when the command is pressed.

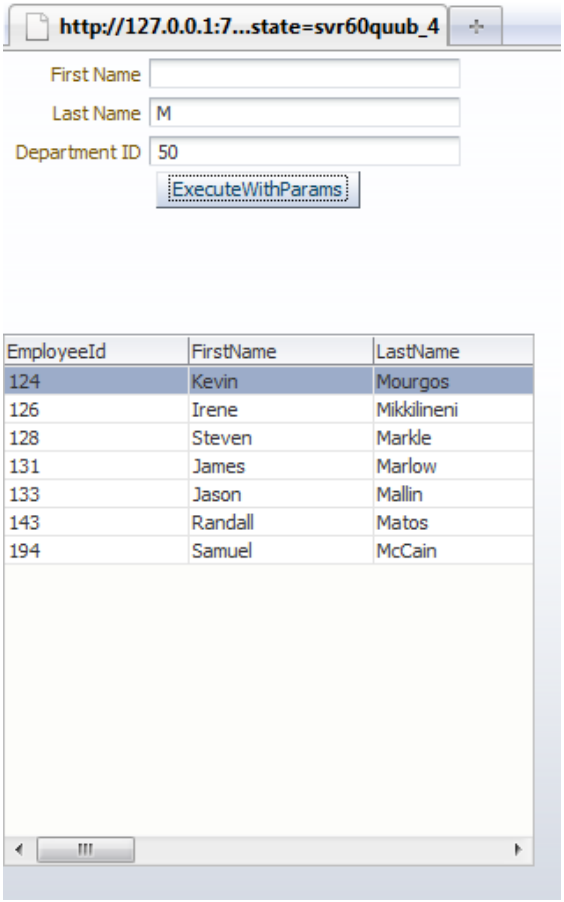


Next, select the search command button of the search form and browse to its **PartialSubmit** property. Set the property to **true** so that when pressing the button, the page is not refreshed as a whole and instead an Ajax request is send to the server.



At runtime you can now add search conditions into the search form and press the search button to see the table queried similar to when using the `af:query` component.

Note that in contrast to the `af:query` component, the search form in this example is fully customizable by you.



Useful extra steps for you to do

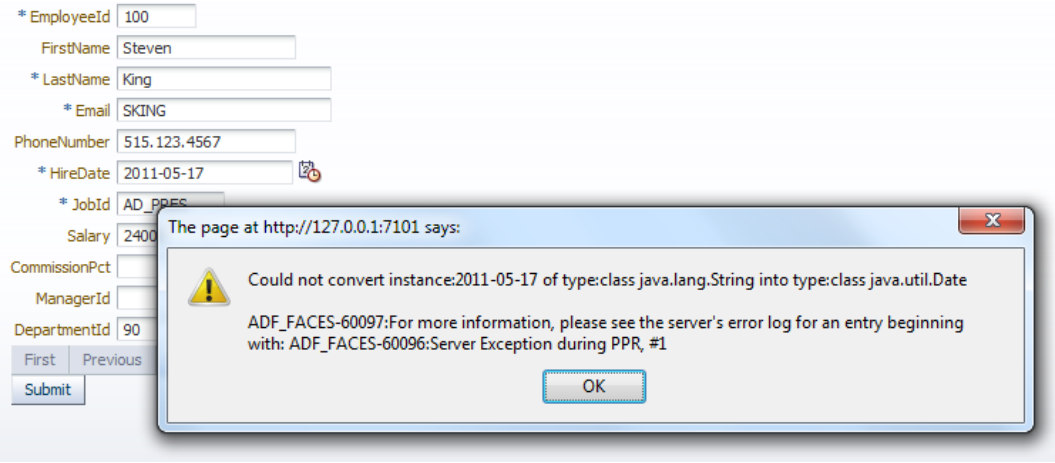
- Rename the ExecutesWithParams method binding in the PageDef file to something meaningful to you and your peers. Make also sure all EL references (e.g. the command button) are updated accordingly
- Provide UI hints for the bind variables for user friendly (non technical) prompts

TypeConversionException using af:validateDateTimeRange

ADF Faces provides the `af:validateDateTimeRange` tag in the **Operations** category of the **Component Palette** for applications to enforce valid date and time ranges.

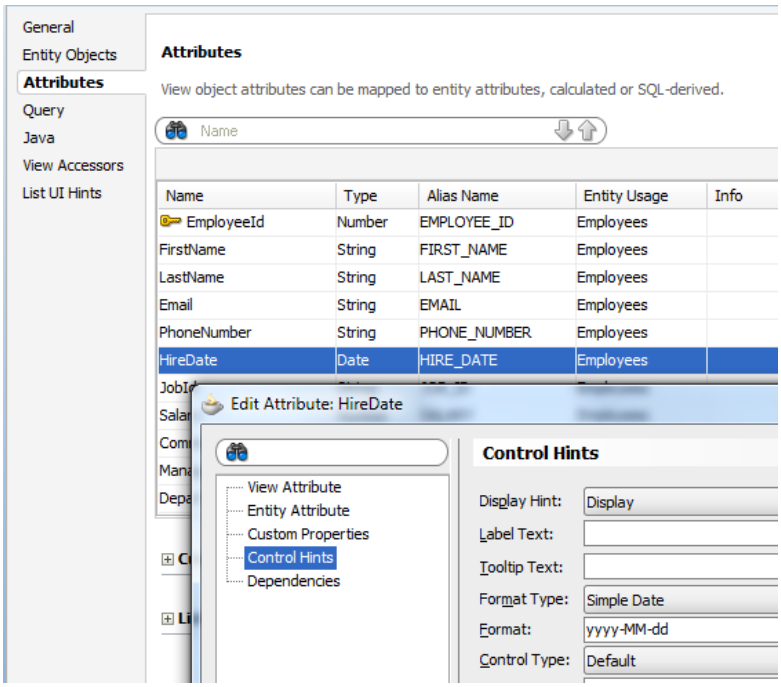
However adding the `af:validateDateTimeRange` to an ADF Business Components bound date field may cause a type conversion exception at runtime as shown below:

```
<LifecycleImpl> <_handleException> ADF_FACES-60098:Faces lifecycle receives
unhandled exceptions in phase PROCESS_VALIDATIONS 3
org.apache.myfaces.trinidadinternal.convert.TypeConversionException: Could not
convert instance:2011-05-17 of type:class java.lang.String into type:class
java.util.Date
at org.apache.myfaces.trinidadinternal.convert.GenericConverterFactory.convert
(GenericConverterFactory.java:289)
at org.apache.myfaces.trinidadinternal.validator.DateTimeRangeValidator.
validate(DateTimeRangeValidator.java:63)
```



The exception is thrown when a user updates the date field and submits the form. The reason for this exception is that the date field in ADF Business Components has a simple date format configured in its UI Control hints. To avoid this error, you need to unset the simple date format configuration on the View Object or Entity Object.

The UI Control hint is not automatically set when creating ADF Business Components models, which helps to control the setting of simple date patterns to date fields that don't require enforcing a specific date and time range



Task Flow transactions and Application Modules

On the Enterprise Methodology Group (EMG) and OTN forum, Chris Muir fired up a discussion about the bounded task flow transaction settings and its impact to the ADF Business Components Application Modules. In two blog entries, Chris summarizes his findings, which is a good – though advanced – read I recommend:

"JDev 11g, Task Flows & ADF BC the Always use Existing Transaction option it's not what it seems"

http://one-size-doesnt-fit-all.blogspot.com/2011/05/jdev-11g-task-flows-adf-bc-always-use.html

"JDev 11g, Task Flows & ADF BC one root Application Module to rule them all?"

http://one-size-doesnt-fit-all.blogspot.com/2011/05/jdev-11g-task-flows-adf-bc-one-root.html

Using check boxes for table row selection(s)

Two questions are re-occuring on the OTN forum. The first question is about how to add checkboxes in front of ADF table rows for the user to select them. The native implementation of table row selection is to select multiple table rows (assuming the table is configured for multi row selection) by holding the *ctrl* key and clicking into the cell. For using checkboxes, there is a bit of coding required, which is explained in this blog post by Sameh Nassar

http://sameh-nassar.blogspot.com/2009/12/use-checkbox-for-selecting-multiple.html

The second question that comes up quite often is how to quickly select all rows in a table, using a check box in the column header of the first table column or a keyboard shortcut. Luc Bors from AMIS answers this question in a blog entry for using the *ctrl+a* key combination and for using a command button. The

change from using a command button to using a check box is minimal and should present no problem for ADF developers.

http://technology.amis.nl/blog/8269/adf-11g-select-all-rows-in-an-adf-table

Determine the arguments of an operation binding at runtime

On the OTN forum, Navaneetha Krishnan answered the question of how to access an Operation Binding at runtime and determine the arguments it expects. Background of the question is that if you call `getParamsMap()` on the Operation Binding, it does not contain the keys of the method arguments.

The example code shown below accesses an `OperationBinding` that is defined in the PageDef file for a method. It then dynamically parses the binding definition for configured method argument names.

For example, a method to relocate employees may be exposed on the ADF Business Component model and exposed in the Data Control panel for drag and drop UI binding in ADF. The method signature of the sample may be as shown below

```
relocateEmployee(Number departmentId, Number employeeId,
                 Boolean withSalaryRaise, Long salaryRaiseInPercent)
```

The easiest way for you to create a method like the one below, is to drag and drop the method from the DataControls panel to the page and have it rendered as a button or link. A binding dialog opens for you to define default values or reference objects that provide the argument values at runtime.

Double click the command item to create a managed bean method that contains generated code for the operation to invoke ("relocateEmployee" in the sample). The following managed bean method was built this way and then extended with the code posted by Navaneetha.

The code sample has a place holder line for where your application would look up the argument values to pass to the method. For example, if there is a HashMap available (e.g. exposed by a managed bean, or passed as an input parameter to a bounded task flow) then you could check if it contains values for the operation argument names you read from the `OperationBinding`.

```
import java.util.Map;
import oracle.adf.model.BindingContext;
import oracle.adf.model.OperationParameter;
import oracle.adf.model.binding.DCInvokeMethod;
import oracle.adf.model.binding.DCInvokeMethodDef;
import oracle.binding.BindingContainer;
import oracle.binding.OperationBinding;
…
//method called from a command item in ADF Faces
public String onRelocateEmployee() {
  //generated binding access code
  BindingContainer bindings = getBindings();
  //access the method binding in the PageDef file
  OperationBinding operationBinding =
      bindings.getOperationBinding("relocateEmployee");
  //to pass arguments to an operation binding, a Map is used.
```

```
//the Map is retrieved by a call to getParamsMap on the operation
//binding
Map operationParamsMap = operationBinding.getParamsMap();
//get access to the operation definition
DCInvokeMethod method =
         (DCInvokeMethod)operationBinding.getOperationInfo();
if (method != null) {
  DCInvokeMethodDef methodDef = method.getDef();
    if (methodDef != null) {
      OperationParameter[] operationParameters = null;
      operationParameters  = methodDef.getParameters();
      if (operationParameters != null) {
         for (OperationParameter operationParameter :
              operationParameters) {
            String argumentName = operationParameter.getName();
            Object argumentType = operationParameter.getTypeName();
            Object defaultValue = operationParameter.getValue();
            if (argumentName != null) {
              Object value = <determine value for argumentNname>;
              operationParamsMap.put(argumentName, value != null ?
                                     value : defaultValue);
             }
           }

         }

       }
    }
    //the operation arguments are provided. Now it is time to
    //execute it
    Object result = operationBinding.execute();
    if (!operationBinding.getErrors().isEmpty()) {
         //TODO log error
         //TODO handle error
         return null;
     }
     return null;
  }
```

Invoking af:exportCollectionActionListener from Java

The `af:exportCollectionListener` tag is a client behavior tag in ADF Faces. Client behavior tags implement functionality that requires JavaScript on the client side if coded manually by application developer. In fact, client behavior tags wrap the JavaScript that is required for a solution and expose a JSF tag for declarative use instead. The `af:exportCollectionListener` exports the visible content of a collection, like exposed in an ADF Faces table, to Excel. The tag is dragged on top of a command item,

like a toolbar button or a command button, and then configured with the table id for which rows should be exported, the name of the file to produce, whether you want to export all rows or just those that are selected, and an optional title string. For more information, see the tag documentation:

http://download.oracle.com/docs/cd/E17904_01/apirefs.1111/e12419/tagdoc/af_exportCollectionActionListener.html

When the command item that has the `af:exportCollectionListener` tag added is invoked by a user, the tag is executed as well and produces the desired output. However, applications usually are a lot more dynamic than to wait for a user to press a button. Instead developers may want to invoke the client behavior programmatically, which in case of a client behavior you can do using JavaScript

```
var button = AdfPage.PAGE.findComponentByAbsoluteId('pc1:ctb1');
if (button != null)
{
  AdfActionEvent.queue(button);
}
```

The JavaScript code above looks an ADF Faces component up in a search from the page root. The component is identified by its absolute component Id ('pc1:ctb1' in the example as the button is in a toolbar facet of a panel collection, which is a naming container). If the component is found, the client event – AdfActionEvent – is queued for the button, which virtually presses it.

However, what if you don't want JavaScript in your application code? In this case, you can use Java to invoke JavaScript from a managed bean. The following code sample provides a method that is called from Java logic in your managed bean to composes the JavaScript String and to invoke it on the client:

```
import javax.faces.component.UIComponent;
import javax.faces.component.UIViewRoot;
import javax.faces.context.FacesContext;
import org.apache.myfaces.trinidad.render.ExtendedRenderKitService;
import org.apache.myfaces.trinidad.util.Service;

public void magicallyPressToolbarCommand(String id) {
  FacesContext fctx = FacesContext.getCurrentInstance();
  UIViewRoot uiViewRoot = fctx.getViewRoot();
  UIComponent comp = uiViewRoot.findComponent(id);
  //check if the component is a command component. For the
  //exportCollectionListener this is important as we need to
  //queue a client side action event
  if (comp != null && (comp instanceof UIXCommand)) {
    String clientId = comp.getClientId(fctx);
    StringBuffer scriptBuffer = new StringBuffer();
    scriptBuffer.append("var button =
                        AdfPage.PAGE.findComponentByAbsoluteId('");
    scriptBuffer.append(clientId + "'); ");
    scriptBuffer.append("if (button != null){");
    scriptBuffer.append("AdfActionEvent.queue(button);}");
    writeJavaScriptToClient(scriptBuffer.toString());
```

```
  }
  else {
    //ID probably not found. Log the issue
     System.out.println("Component with ID " + id +
                                " not Found !!!");
    //TODO - use proper ADFLogger print message
  }
}

//universal helper method for invoking scripts on the client
private void writeJavaScriptToClient(String script) {
  FacesContext fctx = FacesContext.getCurrentInstance();
  //Trinidad classes
  ExtendedRenderKitService erks = null;
  erks = Service.getRenderKitService(fctx,
  ExtendedRenderKitService.class);
  erks.addScript(fctx, script);
}
```

What you need to pay extra attention to is the ID of the component (a toolbar component in my example). If a component is within a naming container like PageTemplate, Table, PanelCollection, Region, etc. then the ID(s) of the surrounding naming containers must be added as a prefix to the component ID. The form is *namingContainerId:componentId*.

Disadvantage of using JavaScript in Java: The script is harder to debug on the client.

One more challenge!

Okay, now that I showed how to invoke a command item on a client using JavaScript from Java, what if you don't want to have a button showing in the UI? In this case, it should be the application logic only that invokes the `af:exportCollectionAtionListener`. In this case you still add a command item to the page but set its **Visible** property to **false**. This way the command item is hidden in the user interface, but the functionality is still available for the application to use.

Note: Don't however try the same setting the "Rendered" property to false. If a component is not rendered then it cannot be found and invoked on the client.

About default value in SelectOneChoice component

A common requirement when using an `af:selectOneChoice` component is to set its default value to a value in the list. Some developers try to hack into the `af:selectOneChoice` component rendering to set the default to e.g. the first value in the list. This however, doesn't work and if it would, would have a side effect, which is that the underlying model wouldn't know if the default list value is meant to be set as a selection by the user or just a proposed value by the developer. Therefore, the proper way for selecting a default list value in an `af:selectOneChoice` component is to define the default value on the attribute of the object to update. Using ADF Business Components as a business service, the default value would be defined on the Entity Object attribute or the View Object attribute. If you use JPA, you set the default

value on the JPA entity attribute. POJO is similar to JPA. So don't look for manipulating the list rendering but add a default value to the attribute populated by the list.

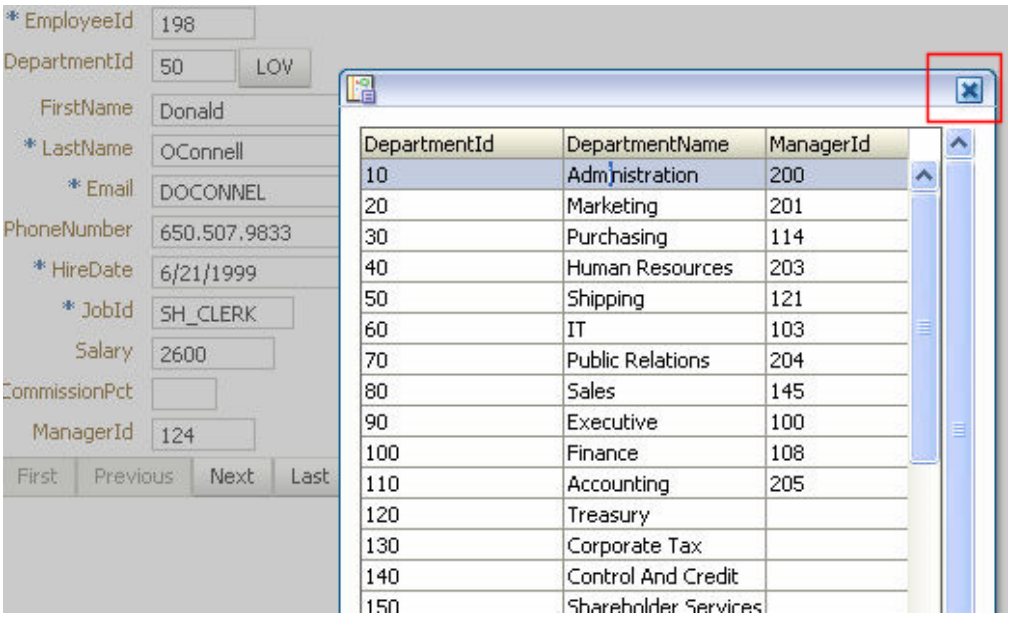
Handling the close icon on a task flow opened as a dialog

Calls to bounded task flows can be configured to open the task flow in an inline popup dialog. To do so, you set the task flow call activity property **Run as Dialog** to true and the **Display Type** property to inline-popup launches the called bounded task flow in an inline popup.

To launch the bounded task flow in the dialog, a command item, like `af:commandButton` is used that, in its **Action** property, references the control flow case that exist between the current view and the task flow call activity

```
<af:commandButton text="Lookup" id="cb6"
    windowEmbedStyle="inlineDocument" useWindow="true"
    windowHeight="300" windowWidth="300"
    action="control_flow_ name" partialSubmit="true"/>
```

Note that it is the command item launching the dialog that defines the size of the opened dialog.



By default, the dialog that contains the task flow has a close icon added in its right upper corner that, if clicked, closes the dialog, abandons the bounded task flow and returns to the calling page. However, using this approach of cancelling the dialog does not produce a return event, which means that the calling view does not get any feedback of what the user did and how he, or she, exit the dialog. To avoid this "loss of control" (there may be use cases where you don't care, but often you do), all you can do is to hide the icon from the dialog. There are two approaches for you to take:

1. Configure a managed bean as shown below to hide the close icons from all dialogs in the application (note that the "dailog" used in the managed bean is not a misspelling in this post but a misspelling the internal framework code.

```
<managed-bean>
  <managed-bean-name>
    oracle$adfinternal$view$rich$dailogInlineDocument
  </managed-bean-name>
  <managed-bean-class>java.util.TreeMap</managed-bean-class>
  <managed-bean-scope>application</managed-bean-scope>
    <map-entries>
      <key-class>java.lang.String</key-class>
      <value-class>java.lang.String</value-class>
      <map-entry>
        <key>MODE</key>
        <value>withoutCancel</value>
      </map-entry>
    </map-entries>
  </managed-bean>
```

2. Use skinning to hide the icon

   af|panelWindow::close-icon-style{ display:none; }

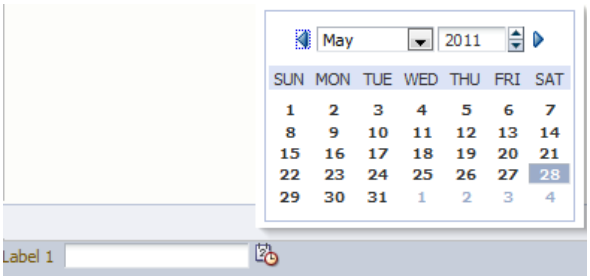
The question of how to hide the close icon in the bounded task flow dialog is reoccurring and I documented this in two blog articles you can access from the links below:

1. Managed Bean approach (JDeveloper 11.1.1.4)
2. Skinning Approach

Note: The full URLs are provided at the end of this OTN Harvest document too.

Select month and year only using af:inputDate

The `af:inputDate` component is an input field with date picker that users use to select an input date.



The value returned by the `af:inputDate` component by default contains the selected day, month and year of the chosen date. To customize the date format or the information applied to the input field, you use the f:convertDateTime tag.

For example, the following inputDate definition …

```
<af:inputDate label="Label 1" id="id1">
    <f:convertDateTime pattern="MM/yyyy"/>
</af:inputDate>
```

… produces the following date input at runtime

OTN forum: Good questions produce better answers

If you are a new developer with Oracle ADF, you for sure have more questions to ask than answers to give. Good to know that the Oracle JDeveloper forum on OTN is a place to go for help. Unfortunately some questions don't contain enough information or lack a use case, making it hard for more experienced ADF developers to help. Therefore, I like to encourage all new members of our forum on OTN to read the post **"Announcement: Please read this before you post"** published by Shay Shmeltzer:

http://forums.oracle.com/forums/ann.jspa?annID=56

To re-iterate some of his points (copying from the doc)

Have a meaningful subject for your thread
Start your threads subject with the technology your question relates to follow by a clear and short description of your problem.
Bad example: Urgent! can you help me Good example: JSF: How to add custom components

Mark Bug and ER thread clearly
If you are reporting a bug start your subject with BUG. If you are requesting a new feature start your subject with ER.

Specify which version of the product you are using
Let us know the specific version of JDeveloper (and other products) you are using.
Bad Example - "How do I switch the projects JDK in JDeveloper?"
Good Example - "How do I switch the projects JDK in JDeveloper 10.1.2.1811?"

Specify any error messages you get
Specific error messages help to identify specific problems
Bad Example - I tried to connect to the database and it doesn't work
Good Example - I tried to connect to the database and got the error message: "Io exception: The Network Adapter could not establish the connection."

Post code when it can help
If you have a small piece of code that can illustrate the problem post it. You can enclose it in [c ode] tags [/code] tags (remove spaces from the tags)

Once you find a solution
Please mark your question as answered.
You should also recognize responses to your thread as helpful or correct using the buttons when appropriate.

One suggestion I like to add is: Don't start with "Adam and Eve" when asking a question. Start with the use case you want to implement (just plain English) and then what you tried and what obviously didn't work. Don't lose yourself in constructs like "VO A1 calls VO A2, which has attributes VO A2 AT1 and VOA2 AT2, linked by a View Link VLV1V2. I dag V1 A2 AT2 as a popup populated by VO A3, which is linked from VO A3 …. Hoe you see this is getting nowhere. Don't make me learn your business before I

can understand the question. Break down your use case so it translates to a common database schema like HR, FOD, or OE.

RELATED DOCOMENTATION

☒	Hide bounded task flow dialog close icon: 1. http://blogs.oracle.com/jdevotnharvest/entry/how-to_remove_the_close_icon_from_task_flows_opened_in_dialogs_11114 2. http://blogs.oracle.com/jdevotnharvest/entry/how-to_hide_the_close_icon_for_task_flows_opened_in_dialogs
☒	
☒	