

## ADF Code Corner

### 017. How-to invoke contextual events from a DVT graph component

**ORACLE®**  
**CODE CORNER**



[twitter.com/adfcodecorner](https://twitter.com/adfcodecorner)

#### Abstract:

The contextual events framework in Oracle ADF defines a publish / subscribe mechanism that leverages the ADF binding layer to notify ADF regions about user interaction performed in other regions, the parent page or a ViewPort (page fragment). Oracle JDeveloper provides declarative help to setup contextual events for input and action components, but not yet for DVT components like graphs. However, only because you don't immediately see a declarative way to do things it doesn't mean that they are not there. This blog article explains how to publish contextual events from a DVT component to update a dependent detail component - a table - in another ADF region.

Author:

Frank Nimphius, Oracle Corporation  
[twitter.com/fnimphiu](https://twitter.com/fnimphiu)  
11-MAR-2010

*Oracle ADF Code Corner is a loose blog-style series of how-to documents that provide solutions to real world coding problems.*

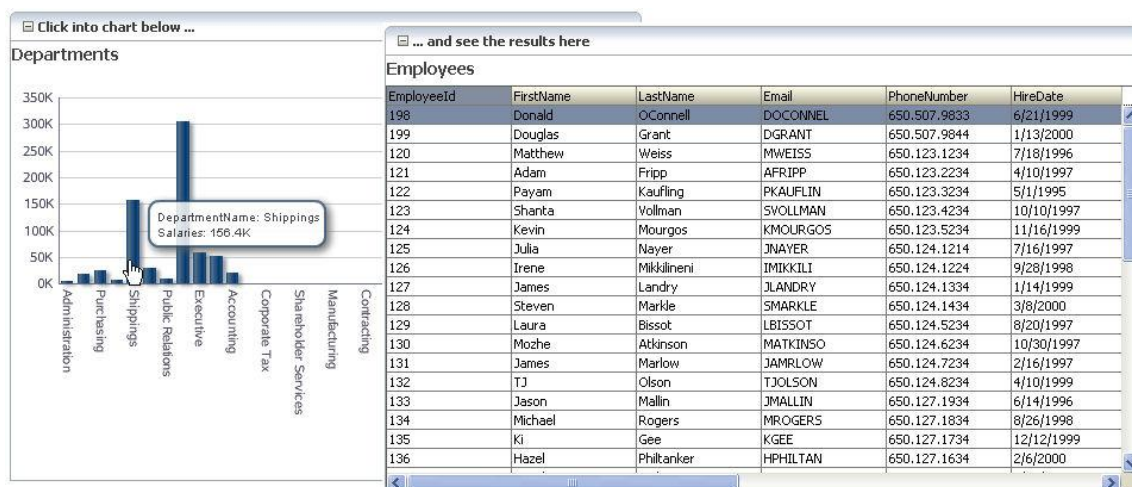
*Disclaimer: All samples are provided as is with no guarantee for future upgrades or error correction. No support can be given through Oracle customer support.*

*Please post questions or report problems related to the samples in this series on the OTN forum for Oracle JDeveloper: <http://forums.oracle.com/forums/forum.jspa?forumID=83>*

## Introduction

Contextual Events in Oracle ADF allow developers to establish communication between a parent ViewPort (page or page fragments) and an ADF regions, or between different regions, without making any assumption about the runtime context. In a previous blog post, I explained how to build contextual events to notify a region about value changes in an input text field, which is a straight forward and completely declarative task to do. Though using a data visualization component (DVT) to publish an event is a bit more challenging, it is a good use case to further explain and understand how contextual events work.

The example in this blog post uses two ADF regions that are located in a jspx document. The first ADF region uses a bounded task flows with a single page fragment to display a list of departments with the overall employee salary for each rendered in a DVT bar graph. The second ADF region uses a bounded task flow with a single page fragment (simplicity is king) to display a list of employees in a table for the selected department in the graph. The communication between the two regions is through contextual events mapped on the shared parent view, the jspx document. The image below shows the runtime view of the sample you can download at the end to this article.



## About the Business Service

The sample uses ADF Business Components as a business service. Note that contextual events are independent of ADF Business Components and can be used with any other business service, including Web Services, POJO and EJB. The only dependency contextual events have is to the ADF binding layer.

The ADF Business Component model exposes two independent collections, departments and employees, where the Employees View Object has a where clause defined that uses a bind variable "deptId" to filter query results to employees that are details of a specific department.

**Note:** Instead of adding a where clause in the View Object query definition, you could use a View Criteria with a bind variable. The View Criteria could be applied to a single View Object instance instead of all instances of a View Object. The programming approach is the same as explained in the following - just the configuration of the ADF Business Component Data Model is different.

## About Contextual Events

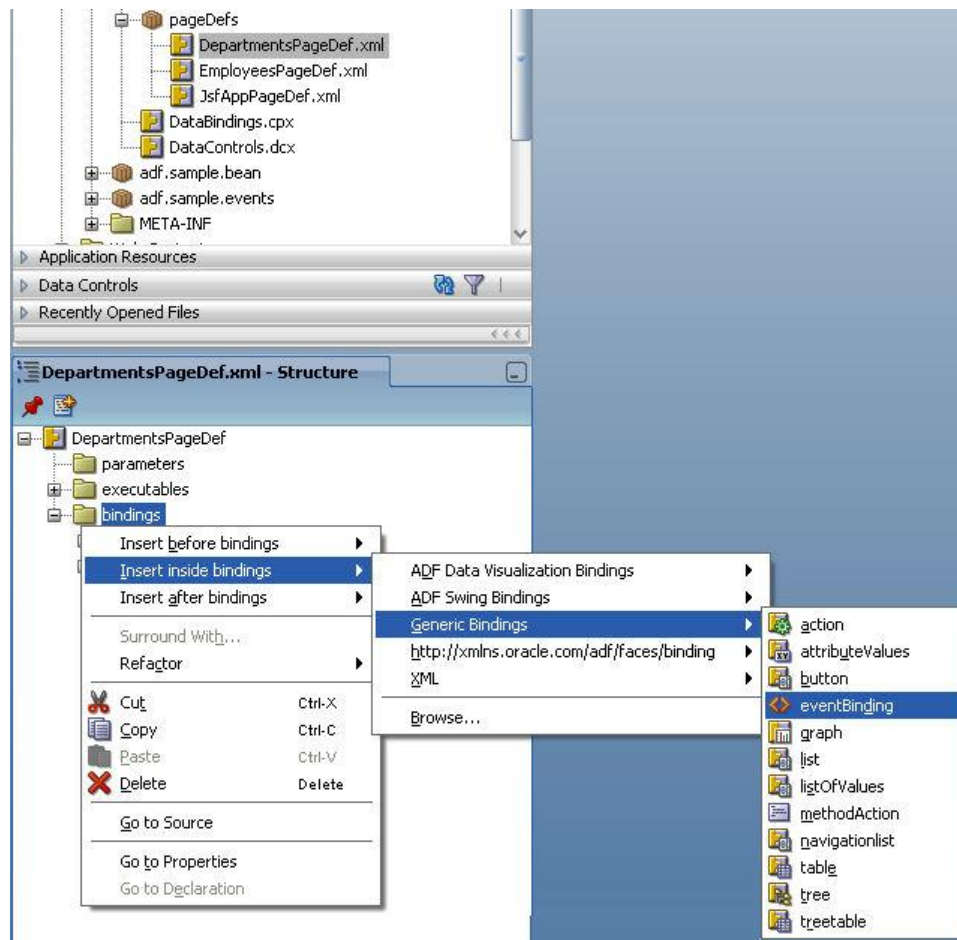
ADF contextual events is a functionality provided by the ADF binding layer and is used to establish loosely coupled communication between ADF regions and their chatting peers, which could be the page or page fragment a region resides in, or another region that passes a message - the payload - in response to a user action. From all possibilities that exist for developers to establish ADF region interaction in their web application, using contextual event is the one that does not require the bounded task flow in an ADF region to be restarted.

- For the DVT example in this blog article, we need to create three artifacts to enable contextual events:
- An ADF event definition in the PageDef file of the page fragment containing the DVT graph
- A method binding in the PageDef file of the page fragment of the detail page fragment to receive and handle the event

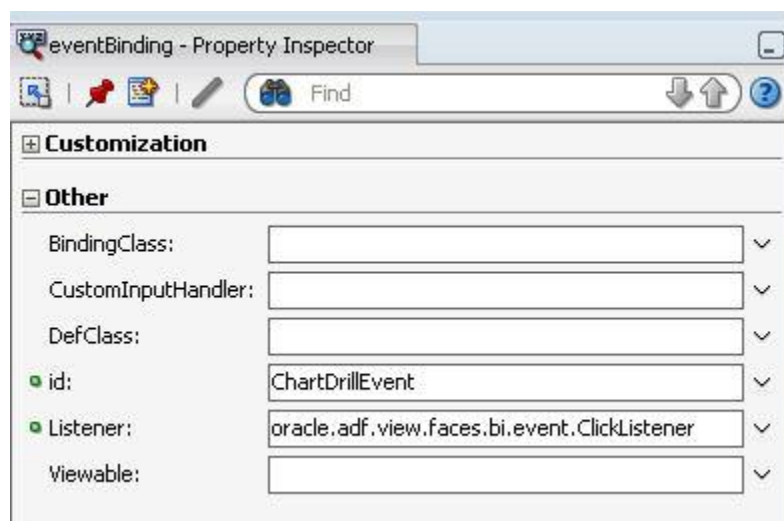
A mapping in the PageDef file of the parent view, the document containing the two regions, to wire up the producer with the event consumer (the event handler).

### Creating the Event Producer

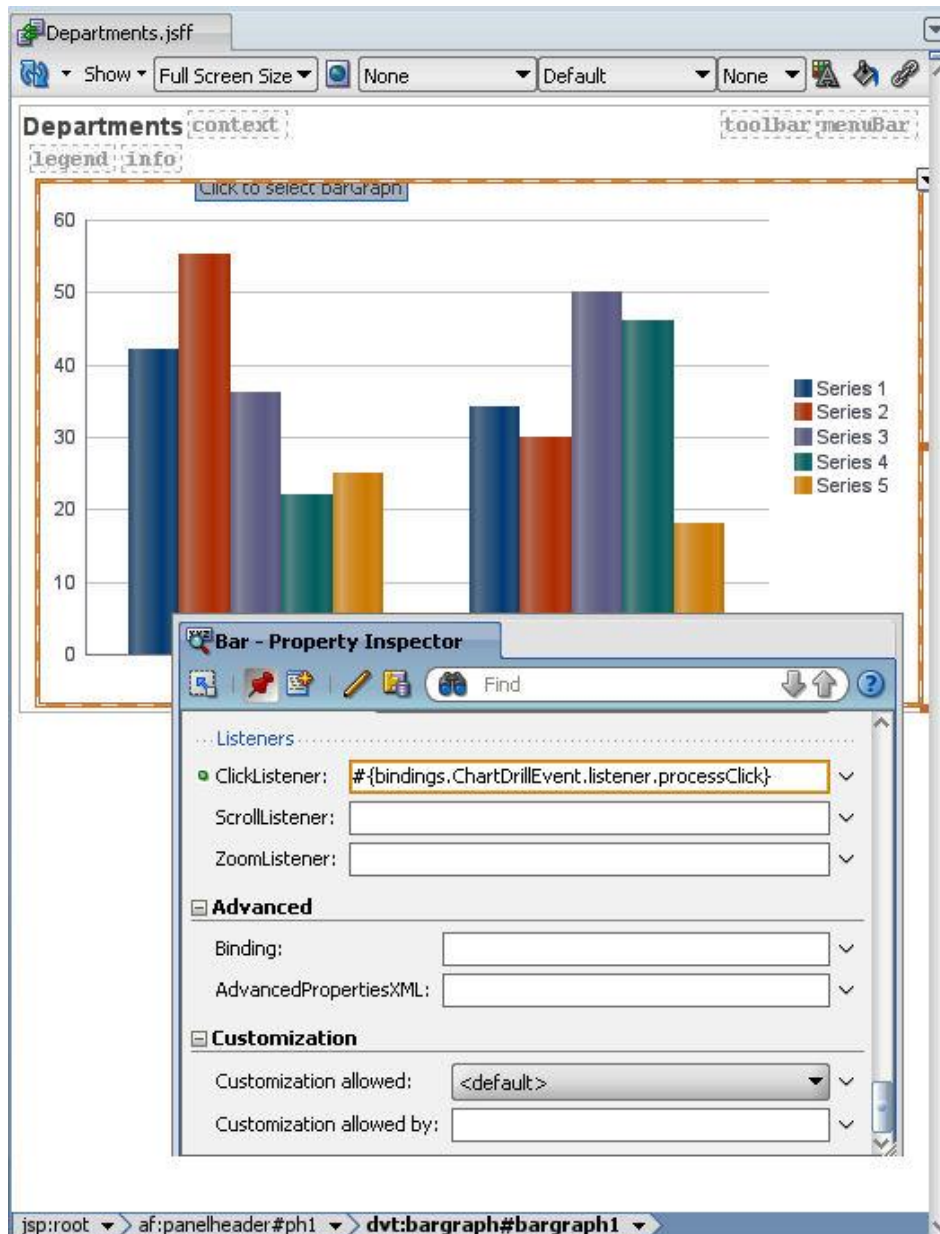
The event producer is an **eventBinding** entry in the PageDef file (the ADF binding file) of the page or page fragment that contains the DVT graph component. To create the event definition, open the PageDef file in the Oracle JDeveloper Structure Window and use the right mouse context menu on the **bindings** node. From the context menu, choose **Insert inside bindings | Generic Bindings | eventBinding** to create the entry for the producer in the ADF binding.



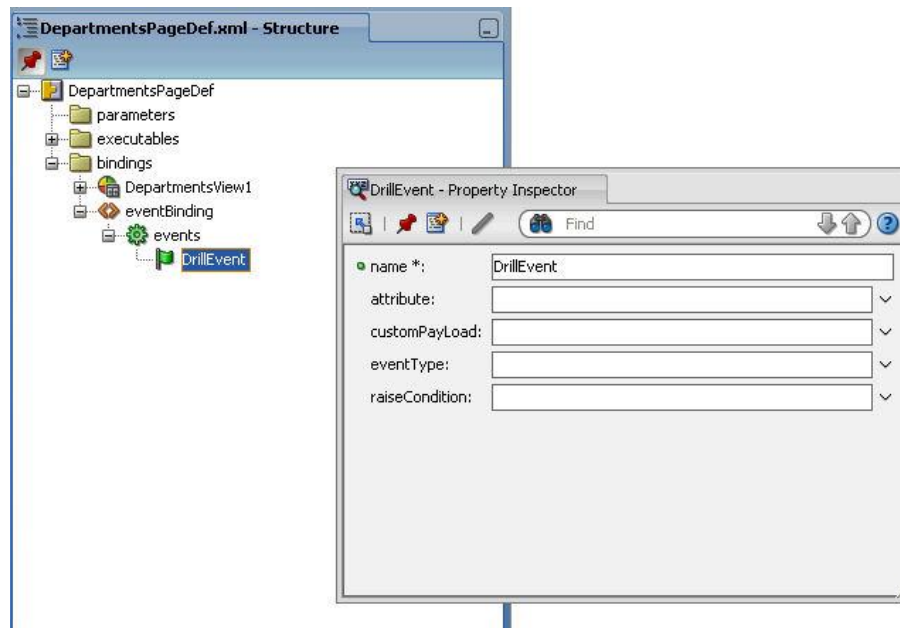
Open the Property Inspector for the created eventBinding and configure the binding to work with DVT click events as shown below



The event binding "id" property is a name developers freely chose for their producer. Later, when invoking the producer event, the producer name is used i) in the DVT component click listener property and ii) the mapping that binds the event receiver to the producer. The "Listener" property is set to the **ClickListener** type, which allows us to invoke this event binding from a DVT component.



The image below show the completed eventBinding definition in the PageDef file of the event producer page, the Departments page fragment in this sample. You create the **events** and the **event** nodes using the right mouse context menu on the **eventBinding** node, choosing **Insert inside ...** from the menu options.



After you configured the DVT component's ClickListener property to reference the new event binding entry, any selection in the graph is passed on to the ADF binding layer. The payload that is contained in the event notification is **ClickEvent** and allows you to determine the department the user selected to get detail information about.

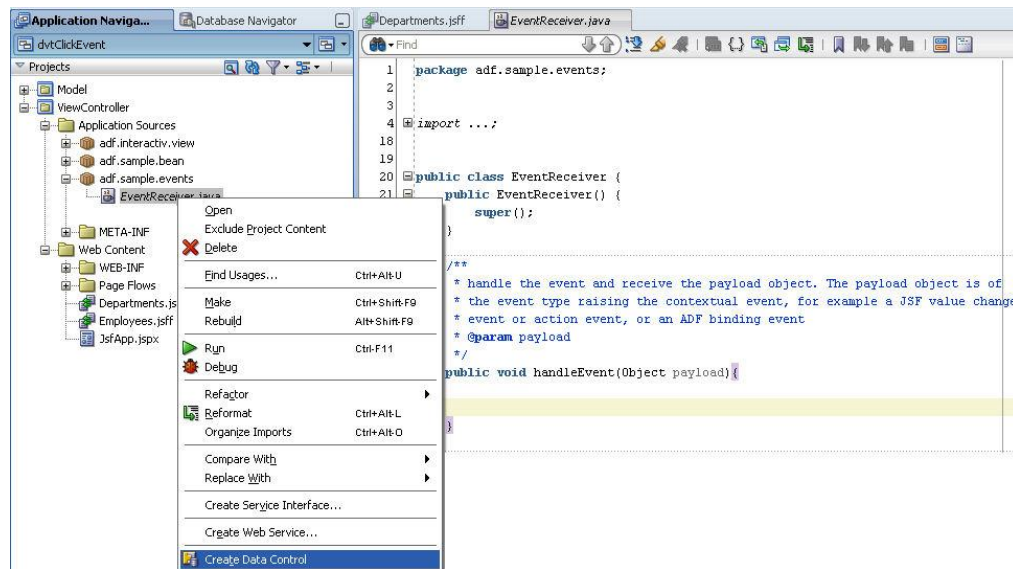
Using the *customPayload* property in the dialog shown above, developers can define a custom payload - for example a value or object read from the binding layer or a managed bean - that is not the event object of the click event. The *raiseCondition* property allows you to conditionally raise the event.

### Creating the Event Receiver

The event receiver needs to be defined on the Data Control and is exposed on the ADF binding of the view that hosts the detail UI component. The event handler can be an operation that is exposed on the Data Control, or a public method that is defined and exposed on the Data Control. In this example, a POJO bean **EventReceiver** is created in the ViewLayer project that exposes a single method `public void handleEvent(Object payload)`. The name of the POJO and the name of the method that handles the event is up to you - the developer. To create a Data Control from the POJO, select it in the Oracle JDeveloper Application Navigator and choose "Create DataControl" from the context menu.

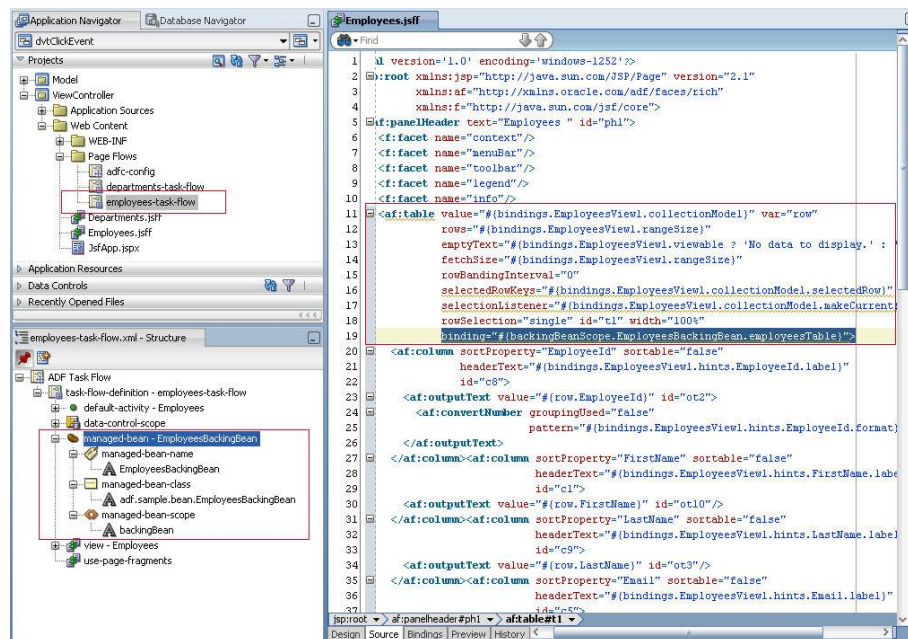
As you see from the image below, you don't need to code the full method implementation first before you can configure contextual events. All you need to create is the DataControl with a public method exposing the event handler method and signature. In the example shown above, the method expects an object type as an argument. Of course, the method can be more specific on the type of argument and the number of arguments it has. If there is more than a single input argument, then this needs to be reflected when mapping the event producer and its payload to the event receiver.





**Note:** Only changes that change the signature of the event handler method require recreating the Data Control.

The event receiver in this example reads the payload, an instance of **ClickEvent**, determines the selected department and queries the employees table based on this information. To query and refresh the employee table, the **handleEvent** method in this example accesses a managed bean that has a setter/getter binding to the table on the employees page fragment. The managed bean is configured in the bounded task flow definition that contains the employees view and is set to **backingBeanScope**.



Through its **af:table** binding reference (shown in the image above), the managed bean can access the table CollectionModel and the ADF binding providing this information. The latter is important to set the

where clause of the ViewObject query so the employees are filtered for the selected department. The managed bean exposes a public method - `reQueryEmployeeestable(oracle.jbo.domain.Number deptId)` - for the contextual event handler to access and execute using EL.

```
import oracle.adf.view.rich.component.rich.data.RichTable;
import oracle.adf.view.rich.context.AdfFacesContext;
import oracle.jbo.VariableValueManager;
import oracle.jbo.ViewObject;
import oracle.jbo.uicli.binding.JUCtrlHierBinding;
import org.apache.myfaces.trinidad.model.CollectionModel;

public class EmployeesBackingBean {
    private RichTable employeesTable;

    public EmployeesBackingBean() {
        super();
    }

    public void reQueryEmployeeestable(oracle.jbo.domain.Number deptId){
        //get access to the ViewObject through the table's ADF tree binding.
        //Note that this approach assumes ADF BC as the business service
        //because View Objects don't exist elsewhere
        CollectionModel model = (CollectionModel)
            this.getEmployeesTable().getValue();
        JUCtrlHierBinding adfBinding =
            (JUCtrlHierBinding) model.getWrappedData();
        ViewObject employeesViewObject = adfBinding.getViewObject();
        //set the bind variable in the query to the value of the department
        VariableValueManager vmanager =
            employeesViewObject.getVariableManager();
        vmanager.setVariableValue("deptId", deptId);
        employeesViewObject.executeQuery();
        //refresh the table to show the result of the query
        AdfFacesContext adfFacesCtx = AdfFacesContext.getCurrentInstance();
        adfFacesCtx.addPartialTarget(this.getEmployeesTable());
    }

    //table reference
    public void setEmployeesTable(RichTable employeesTable) {
        this.employeesTable = employeesTable;
    }

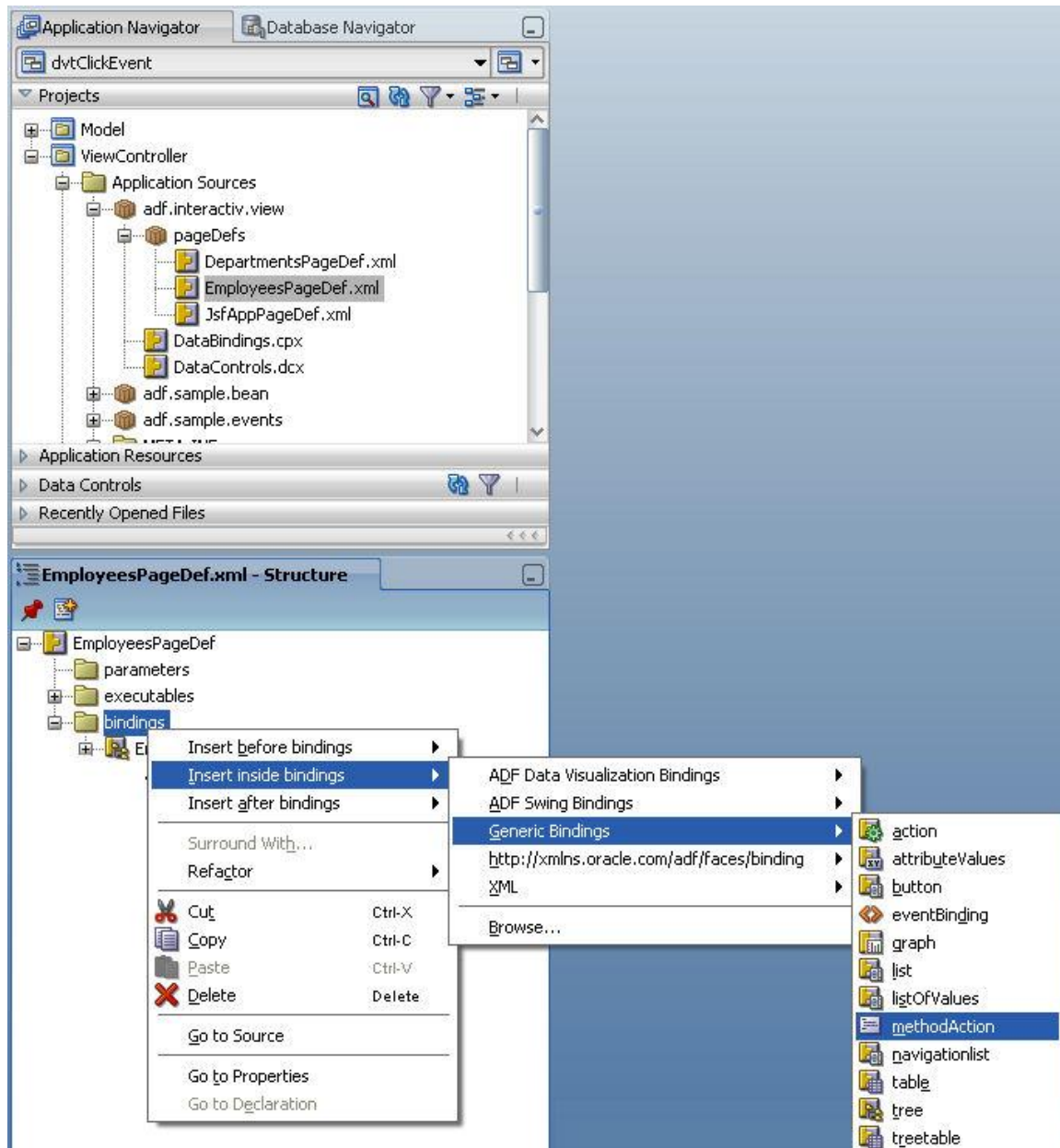
    public RichTable getEmployeesTable() {
        return employeesTable;
    }
}
```

With this set up, the contextual event handler - the receiver - can get access to the table instance and pass the selected departments Id in. Because there is no guarantee that the employees table is displayed when the click event is published (though in my example here this guarantee is given), make sure the event handler checks for possible NPE and handles them gracefully when accessing the backing bean instance.

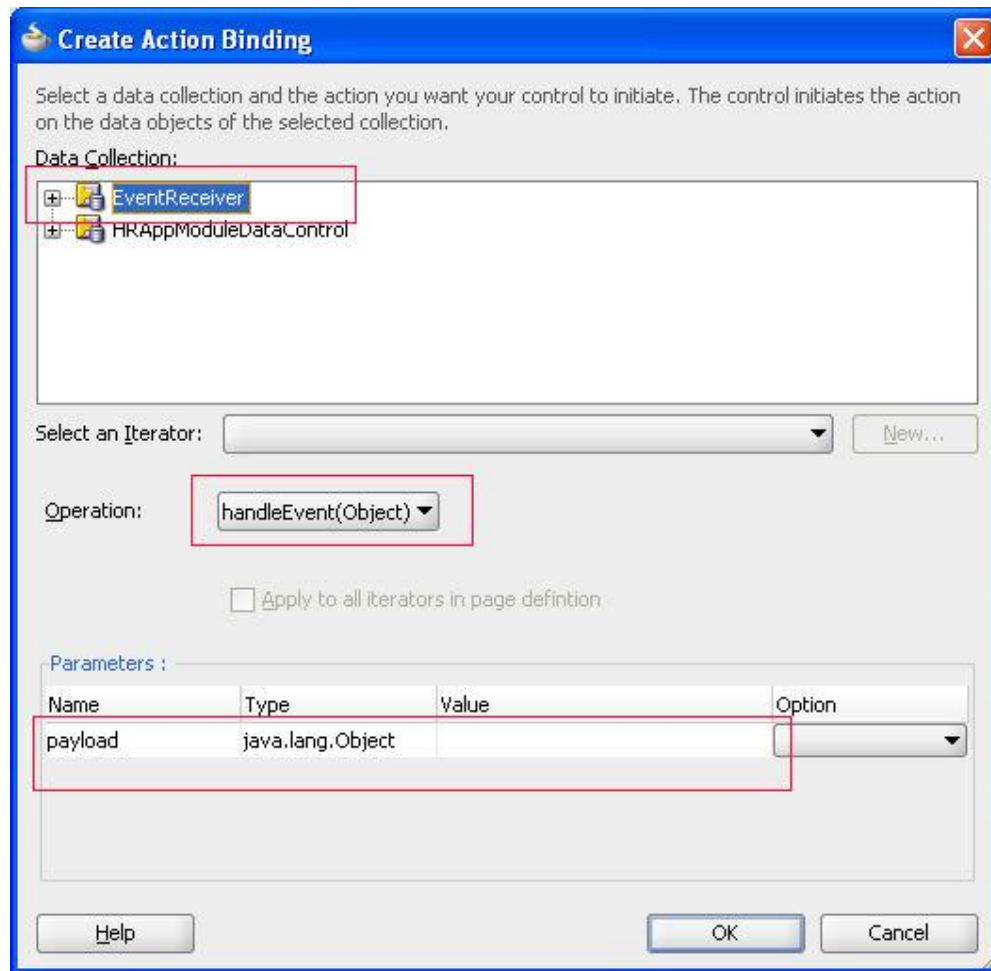
To create the event handler - the receiver - in the ADF binding of the employees page fragment, open the PageDef file in the Structure Window, or in the binding editor that opens through a double click on the



PageDef entry. In the PageDef file, create a method binding for the *handleEvent* method exposed on the POJO DataControl.



In the opened dialog, select the POJO DataControl and choose the event handler method from the available Operations. Because the method argument is provided by the contextual event mapping, you don't need to reference a value in this dialog. Of course, if you wanted, then you could reference a value and ignore the event payload, in which case you only use the event notification mechanism of the contextual event framework.



With this, we configured the event producer and the event handler in their respective ADF binding files. All that is needed to get ADF contextual events working is to create the event mapping, which is the last step to do in this little tutorial.

For your reference, below is the complete **EventReceiver** class code with the **handleEvent** event handler method used in this example.

```
import adf.sample.bean.EmployeesBackingBean;
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.faces.bi.event.ClickEvent;
import oracle.dss.dataView.DataComponentHandle;
import oracle.jbo.Key;

public class EventReceiver {
    public EventReceiver() {
        super();
    }
}
```

```

    }

    /**
     * handle the event and receive the payload object. The payload
     * object is of the event type raising the contextual event, for
     * example a JSF value change event or action event, or an ADF
     * binding event
     * @param payload
     */
    public void handleEvent(Object payload){
        //cast the payload to the ClickEvent. Note that the ctx event
        //handler could have used a typed argument instead of "Object",
        //in which case a) an error would be raised if the value in the
        //payload is of a wrong type and b) a casting in the method is
        //not required.
        //There are good arguments for the use of both approaches. The one
        //using an Object argument for example allows you to gracefully
        //handle possible type cast exceptions instead of handling it
        //using the configured ADF or ADFc exception handlers.

        ClickEvent clickEvent = (ClickEvent) payload;
        //determine the area that the user clicked on
        if (clickEvent.getComponentHandle() instanceof
            DataComponentHandle){
            DataComponentHandle dch =
                (DataComponentHandle) clickEvent.getComponentHandle();
            //get the selected row key from the graph (click event)
            Key rwKey = (Key) dch.getValue(DataComponentHandle.ROW_KEY);
            EmployeesBackingBean employeesBean = getEmployeesBean();
            //the ADF BC key is an array of PK attribute. In this case
            //we have a single PK attribute defined

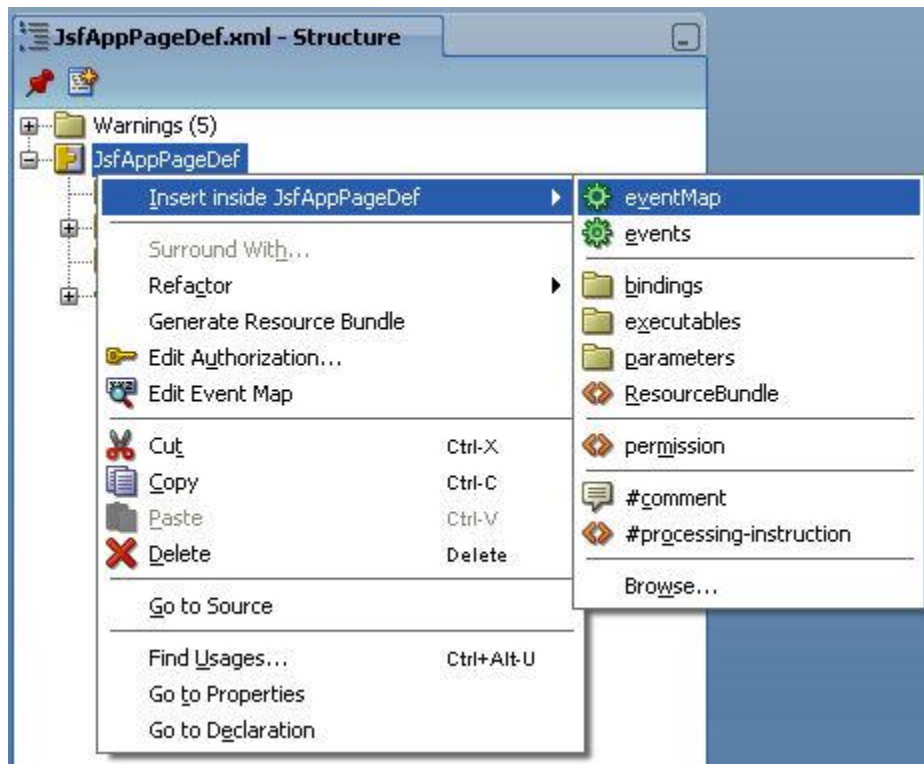
            employeesBean.reQueryEmployeestable((oracle.jbo.domain.Number)
                rwKey.getKeyValues()[0]);
        }
    }

    /** private method that resolves the handle to the employee backing
        bean using EL */
    private EmployeesBackingBean getEmployeesBean() {
        FacesContext fctx = FacesContext.getCurrentInstance();
        ELContext elctx = fctx.getELContext();
        Application jsfApp = fctx.getApplication();
        ExpressionFactory exprFactory = jsfApp.getExpressionFactory();
        ValueExpression ve = null;
        //access the backing bean, which is configured in the employees
        //task flow. Its configured to be in backing bean scope
        ve = exprFactory.createValueExpression(
            elctx,
            "#{backingBeanScope.EmployeesBackingBean}",
            Object.class);
        return (EmployeesBackingBean) ve.getValue(elctx);
    }
}

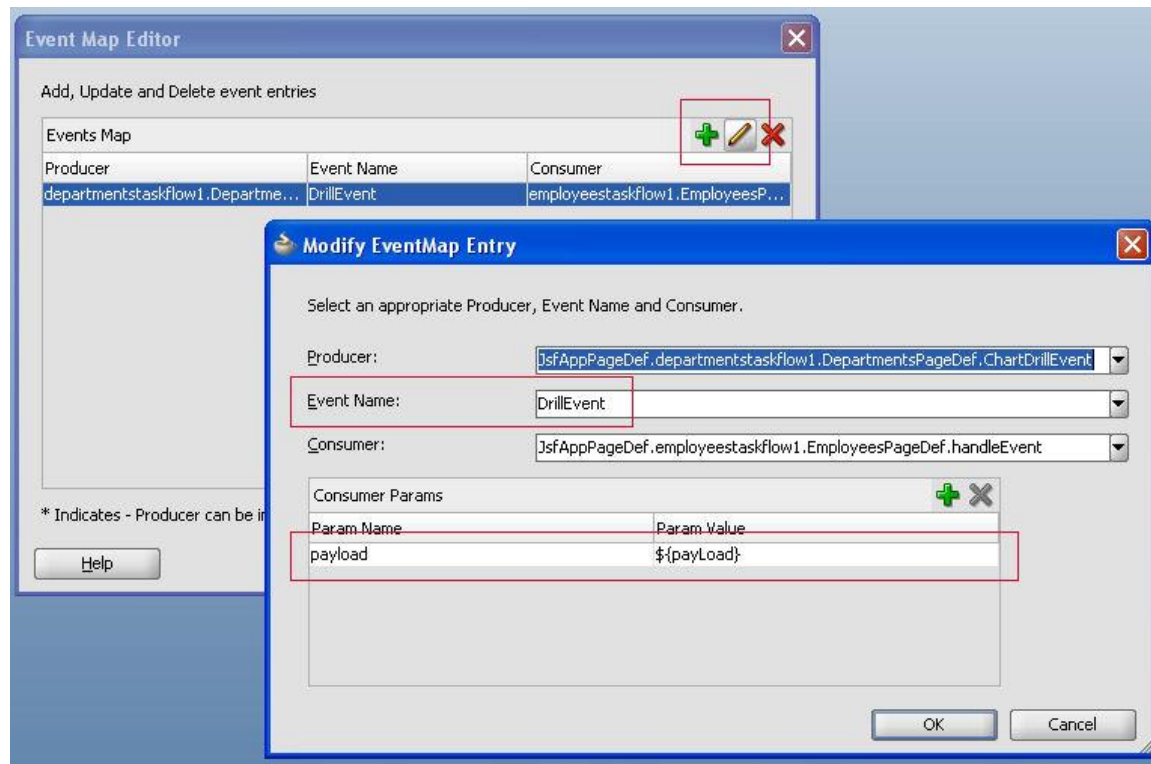
```

### Building the Event Map

The event map is created on an ADF binding file that "sees" the PageDef files for the event producer and receiver. In this example, its the PageDef file of the page that contains both ADF regions. Note that ADF also supports wild card events, in which case the mapping does not directly "see" the producer and thus uses an asterisk "\*" instead. To create the event map, select the PageDef file of the parent view ( a page of page fragment) and choose **Insert inside | eventMap** from the context menu.



Then, select the PageDef root node again and choose **"Edit Event Map"** from the context menu to create the contextual event mapping. This step now is completely declarative as shown in the image below. Just press the green plus icon or the pencil depending on whether you want to create or edit a mapping



The event name "DrillEvent" is what we defined when creating the producer. It automatically shows when selecting the "ChartDrillEvent" eventBinding as the producer. You then select the Consumer (the event handler) and use the green plus icon to define one or more argument mappings. In this example, the event handler has a single argument, that I pass the payload object to. The payload of the event is accessible from the `${payload}` EL string.

**Note:** The dialog refers to the event handler as **Consumer**, which is a popular though misleading term for it. Events are not consumed but listened for. If they were consumed then an event could only be handled by a single event handler and stop immediately after. As a real life example: you can consume a bottle of beer, but you can only listen to a live concert of your favorite band. Similar in contextual events. The event receiving end(s) listen for it, handle it but don't consume it.

### Invoking contextual events from Java

Everything you can with EL you can also do in Java. So instead of using EL from the `ClickListener` property to directly access the contextual event producer eventBinding - "ChartDrillEvent" in this sample - in the ADF binding layer, you can use a managed bean. This way developers can add custom code before and after the click event event processing. An example for such a managed bean is shown below.

```
import oracle.adf.model.BindingContext;
import oracle.adf.view.faces.bi.event.ClickEvent;
import oracle.adf.view.faces.bi.event.ClickListener;
import oracle.binding.BindingContainer;
...
// optional - instead of
```

```
// #{bindings.ChartDrillEvent.listener.processClick}

public void onClick(ClickEvent clickEvent) {
    BindingContext bctx = BindingContext.getCurrent();
    BindingContainer bindings = bctx.getCurrentBindingsEntry();
    //access the eventBinding object of the producer
    JUEventBinding eventBinding =
        (JUEventBinding) bindings.get("ChartDrillEvent");
    //the eventBinding is configured as a ClickListener so we
    //can invoke it from Java
    ClickListener ddl = (ClickListener)eventBinding.getListener();
    ddl.processClick(clickEvent);
}
```

...

## Download the sample

You can download the sample shown above from **ADF Code Corer**:



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

It is a JDeveloper 11.g R1 PS1 workspace that is configured to work with the HR database schema. Before you can run it, please change the database connection to a database accessible to you. Run the JSPX file and wait for both ADF regions to render. Then click on one of the graph bars to see the detail table refreshing

---

### RELATED DOCUMENTATION

---

	Contextual events in ADF product documentation <a href="http://download.oracle.com/docs/cd/E15523_01/web.1111/b31974/web_adv.htm#CACJBFGI">http://download.oracle.com/docs/cd/E15523_01/web.1111/b31974/web_adv.htm#CACJBFGI</a>
	Oracle Fusion Developer Guide – McGraw Hill Oracle Press, Frank Nimphius, Lynn Munsinger <a href="http://www.mhprofessional.com/product.php?cat=112&amp;isbn=0071622543">http://www.mhprofessional.com/product.php?cat=112&amp;isbn=0071622543</a>
