ADF Code Corner

79. Strategy for global buttons on a page template

twitter.com/adfcodecorner

Abstract:

Page templates in ADF Faces help designing custom application layout standards for a consistent look and feel throughout an application or applications. Page templates and skinning present the corner stones of the application visual appearance. A productivity aspect of templates is ease of use and maintenance: Template definitions are not compiled into the consuming page but referenced at runtime, which makes it easy to apply layout changes to an existing application.

A frequent requirement for templates is to provide default behavior and global command action items, like global toolbars or buttons. In this article I show how global action and toolbars can be implemented in templates.

Author: Frank Nimphius, Oracle Corporation
twitter.com/fnimphiu
13-APR-2011

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

Page templates can be imported to a project through the use of ADF Library. ADF Libaries allow developers to also deploy JavaServer Faces artifacts like managed beans and ADF controller and `faces-config.xml` configurations used by by ADF reusable components.

From all reusable elements in ADF, including bounded task flows, ADF regions, dynamic and tag library based declarative components, page fragments and page templates, there are two that are candidates for implementing the use case of "templates with behavior": **declarative components** and **page templates**.

Building functional reusable layouts using tag library driven declarative components is subject of sample #24, also here on ADF Code Corner. The remainder of this article therefore focuses on how to add pre-defined functions to page templates:
http://www.oracle.com/technetwork/developer-tools/adf/learnmore/index-101235.html

The challenge with page templates is that, in contrast to declarative components, they don't have method attributes to expose to the consumer page. Method attributes in declarative components can be used to broadcast a component event, like `ActionEvent`, to a property that application developers can assign an action listener to.

To work around the absence of method attributes,in this article, I use a managed bean that is part of the template to dispatch between the action in the template and the function performed in the application. A similar approach, to deploy a managed bean with a template, is also used in the **Dynamic Tabs UI Shell Template** that is available in Oracle JDeveloper 11.1.1.4 onwards to build desktop like web user interfaces with ADF and ADF Faces:

http://www.oracle.com/technetwork/developer-tools/adf/uishell-093084.html

Implementation Outline

Page templates can have Facets, Attributes and ADF bindings defined. Facets are used by the template developer to define areas in which application developers later add ADF Faces components to. Attributes show as template properties in the Oracle JDeveloper Property Inspector.

Read more about page templates in Oracle ADF Faces in the product documentation at

http://download.oracle.com/docs/cd/E17904_01/web.1111/b31974/web_getstarted.htm#BABJEDHG

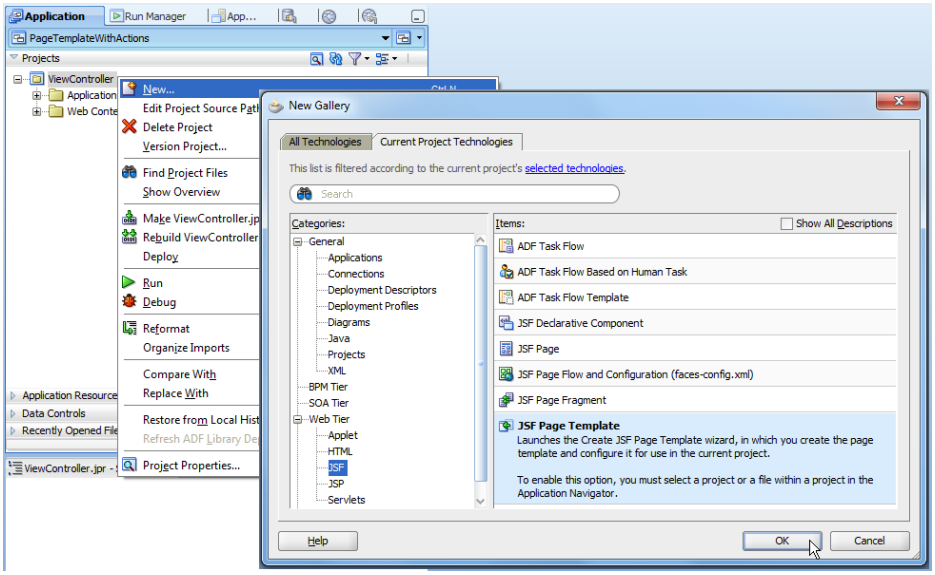
To implement a strategy that allows a template to invoke a method in the consuming page, I use the following artifacts

- A managed bean owned by the template that all command components are bound to. The managed bean is configured in backing bean scope using the `adfc-config.xml` configuration file. The bean does not need to hold state longer than request and using the backing bean scope allows this template to be used in page fragments as well.
- A Java interface class that describes the methods used by the template. This interface is the contract between the template and the consuming page and is exposed as a required template attribute. Consuming applications must provide a managed bean that implements the interface. All command items in the template that need to execute page specific actions are bound to the managed bean in the template.
- A template attribute that allows developers to use Expression Language to reference a managed bean in their application for the template to call in response to a global button action.

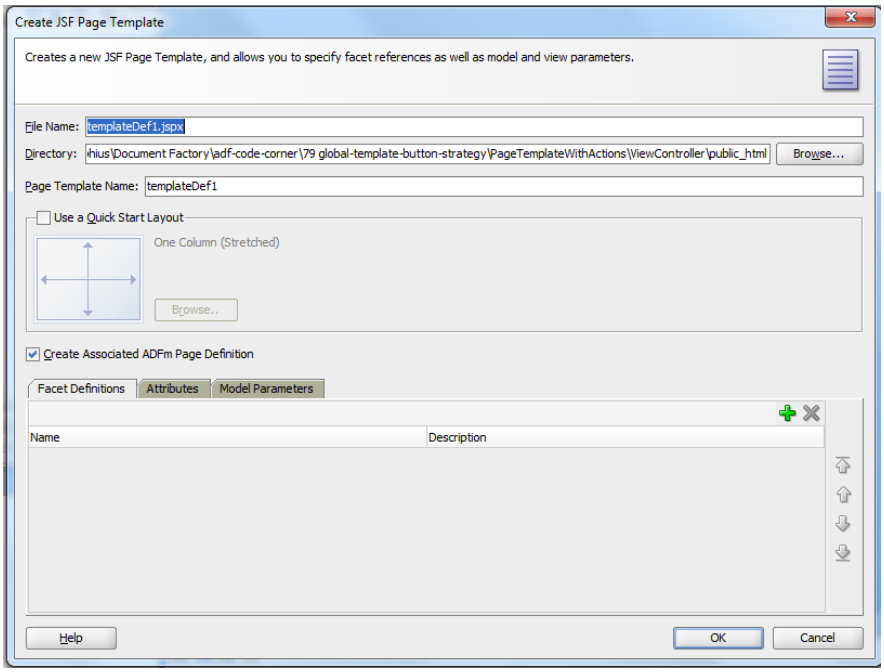
Step-by-step

To build page re-usable templates I recommend building them in their own Oracle JDeveloper project. Ensure the project has the ADF Faces and ADF Page Flow technology scope set.

To create a new template, I chose **File | New** from the JDeveloper menu.

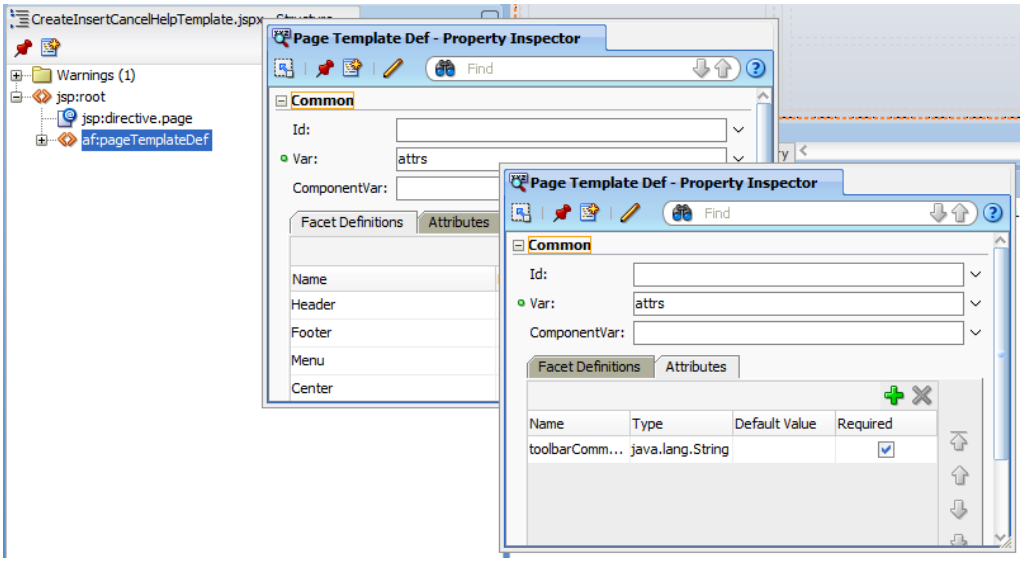


In the template dialog, I defined the template name (display and file name), the facets (areas for the application page content), and an attribute (a property to hold the reference to a managed bean in the application). The attribute – so to say – is my API to implement the contract between the template and the consuming application.

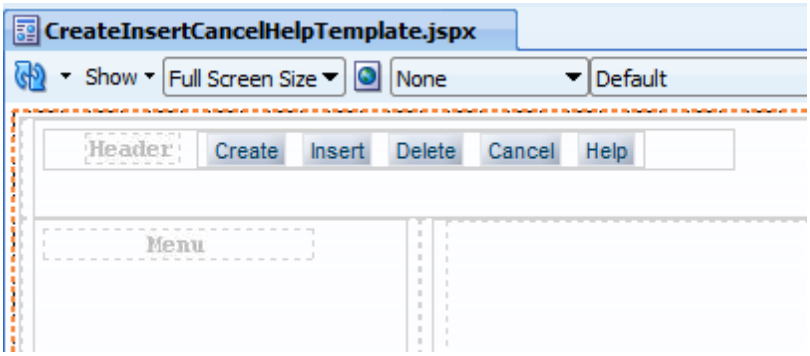


Note: Using the **Quick Start Layout** really turbo boosts the creation of page templates and should always be considered as the starting point

The template definition of the sample in this article is shown in the image below. Note that it defines a single attribute of type **String**. This attribute holds the reference to a managed bean in the consuming application. The **Facet Definitions** are not further explained and most likely will differ in their number and position in your custom templates. Just note that defined facets are added to the template layout using **FacetRef** components from the component palette.

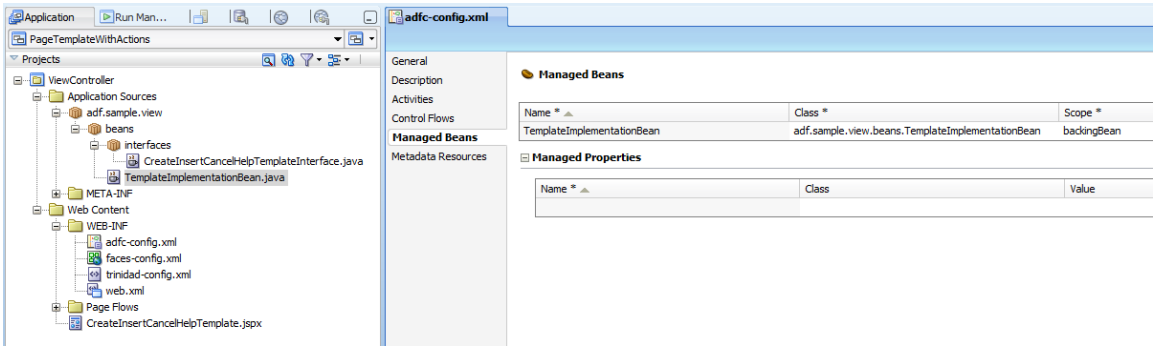


Next I created a **Java Interface** class to define the methods that are invoked by command items in the template. This interface class is the most important detail for the strategy introduced in this paper. You should have a good naming convention for the template and interface names to ensure the name of the interface is predictable for developers using your template in their application development.



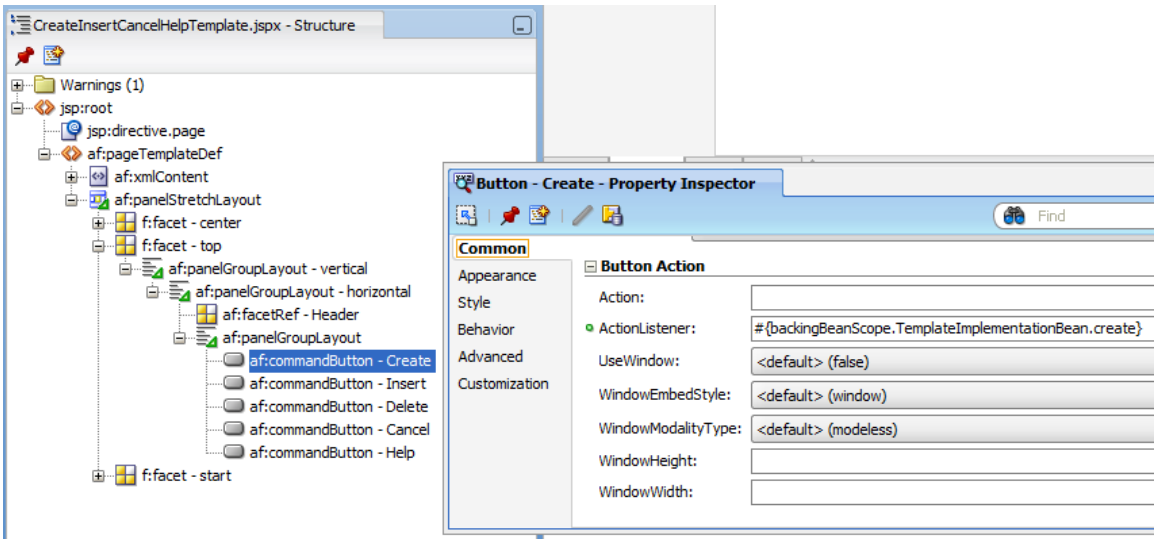

```
import javax.faces.event.ActionEvent;

public interface CreateInsertCancelHelpTemplateInterface {
    public void create(ActionEvent actionEvent);
    public void insert(ActionEvent actionEvent);
    public void cancel(ActionEvent actionEvent);
    public void delete(ActionEvent actionEvent);
    public void help(ActionEvent actionEvent);

 }
```

After creating the interface class, I created a managed bean that I configured in the template's `adfc-config.xml` file (The `adfc-config.xml` file is created automatically for you if the project you develop the template in has the **ADF Page Flow** technology scope set.



Notice that the managed bean scope is set to **backingBean**. Using the **backingBean** scope allows multiple instances of the template to be used on a page, for example when applied to page fragments added to a page through the use of ADF regions.

The managed bean implements the same Java interface that managed beans in the consuming application are expected to implement (the interface created earlier). Using Expression Language, the command buttons in my template all reference one of the methods in the template managed bean in their **Action Listener** property.



Note: If the command buttons are not supposed to refresh the page, make sure their **PartialSubmit** property is set to **true**.

The template managed bean code is shown below. Note that this bean is private to the page template. To indicate that this bean is not for public use, you could (optionally) follow the ADF framework approach and use the word "internal" in the bean package name.

```
import adf.sample.view.beans.interfaces.
       CreateInsertCancelHelpTemplateInterface;
import javax.el.ELContext;
import javax.el.ExpressionFactory;
import javax.el.ValueExpression;
import javax.faces.context.FacesContext;
import javax.faces.event.ActionEvent;

public class TemplateImplementationBean implements
CreateInsertCancelHelpTemplateInterface{
  CreateInsertCancelHelpTemplateInterface templateBean = null;


  public TemplateImplementationBean() {
    super();
  }
  public void create(ActionEvent actionEvent){
    CreateInsertCancelHelpTemplateInterface bean = getBean();
    bean.create(actionEvent);
  };

  public void insert(ActionEvent actionEvent){
    CreateInsertCancelHelpTemplateInterface bean = getBean();
    bean.insert(actionEvent);
  };
  public void cancel(ActionEvent actionEvent){
```

```
    CreateInsertCancelHelpTemplateInterface bean = getBean();
    bean.cancel(actionEvent);
  };

  public void delete(ActionEvent actionEvent){
     CreateInsertCancelHelpTemplateInterface bean = getBean();
      bean.delete(actionEvent);
  };

   public void help(ActionEvent actionEvent){
     CreateInsertCancelHelpTemplateInterface bean = getBean();
     bean.help(actionEvent);
   };

  //Access the template attribute to resolve the application
  //managed bean reference
  private CreateInsertCancelHelpTemplateInterface getBean(){
    if (templateBean == null) {
      FacesContext fctx = FacesContext.getCurrentInstance();
      ELContext elctx = fctx.getELContext();
      ExpressionFactory exprFactory =
                 fctx.getApplication().getExpressionFactory();

      ValueExpression ve = exprFactory.createValueExpression(
                                elctx,
                                "#{attrs.toolbarCommandsBean}",
                                Object.class);
      Object valueObject = ve.getValue(elctx);
      if (valueObject != null) {
        templateBean =
            (CreateInsertCancelHelpTemplateInterface)valueObject;
      } else {
         // log message here
      }
    }

    return templateBean;

  }
}
```

The **getBean()** method in the sample code above is the implementation of the strategy introduced in this article. Using the "*attrs*" template variable handle, the managed bean attempts to access the application managed bean that is referenced from the template. It then passes the command action from the global buttons to the application bean to handle the job.

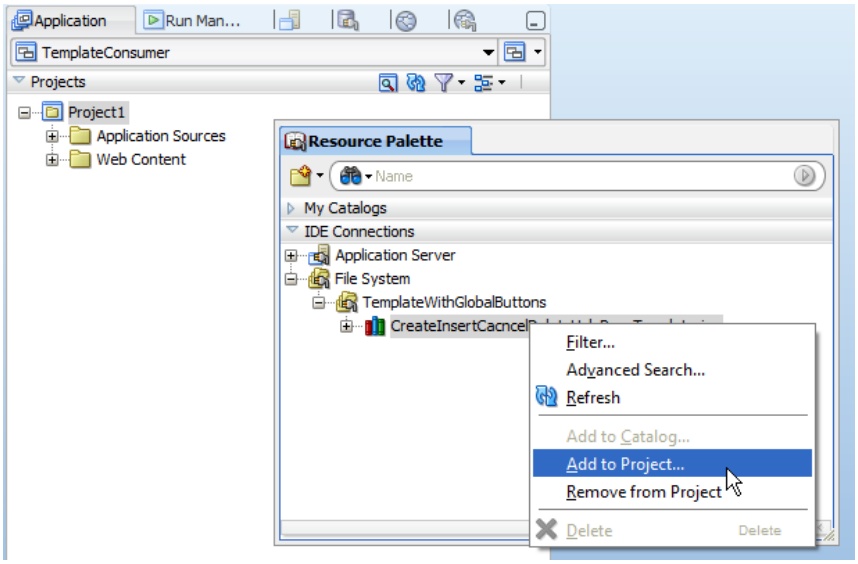
To deploy the template, create a new deployment profile for an ADF library. This ADF library can then be imported into the consuming application project, which also makes the interface class available.

Read more about ADF Libraries:

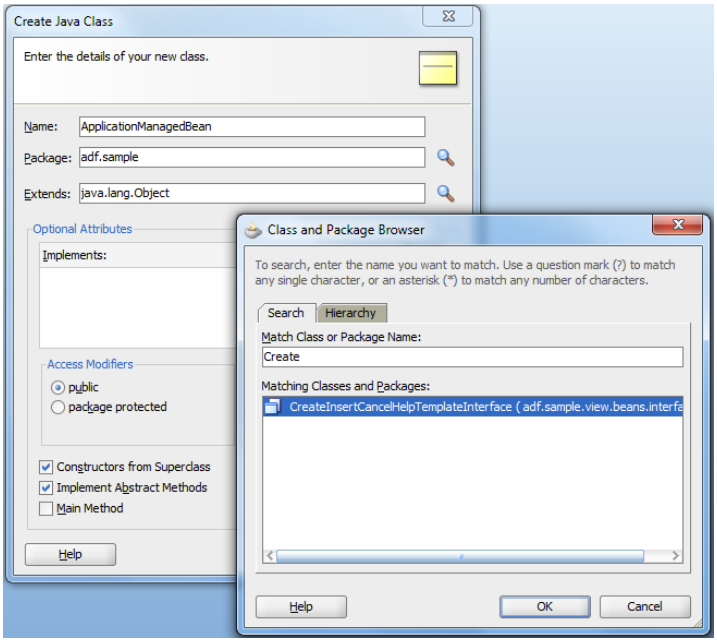
http://download.oracle.com/docs/cd/E17904_01/web.1111/b31974/reusing_components.htm#BEIGHHCG

Using the Template

To use the template in an application, I referenced the ADF Library in the Oracle JDeveloper **Resource Palette**:



I then added the template library to an ADF Faces project so the template definition, managed bean and interface class become available. A managed bean is created that implements the template interface class to handle the method invocation of the template. The Java source file creation dialog of the managed bean I created is shown below.

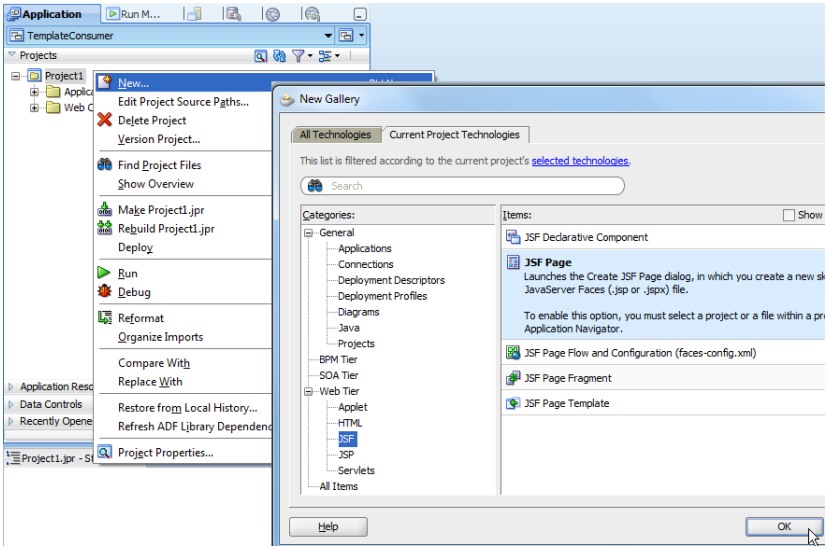


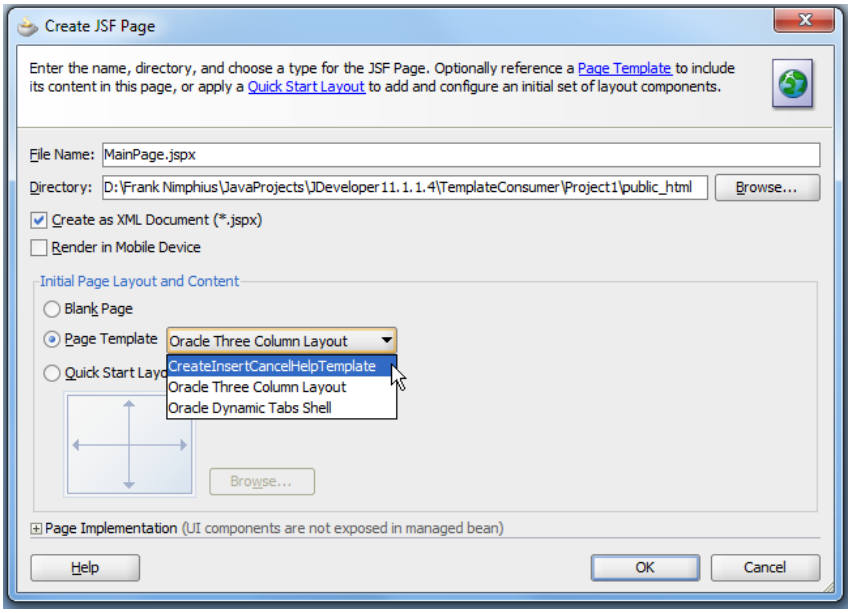
In my test application, I configured the application managed bean in the `adfc-config.xml` file. The managed bean class contains the methods that are called from the template.

All I need to do in the consuming application is to provide the implementation code for the template methods. As a start, I let the consuming application print statements about the methods called by the template button to verify it is working. In a later step, I would add the real application code to execute.

Note: If the template is applied to a page fragment in a bounded task flow, you configure the bean in the task flow definition file.

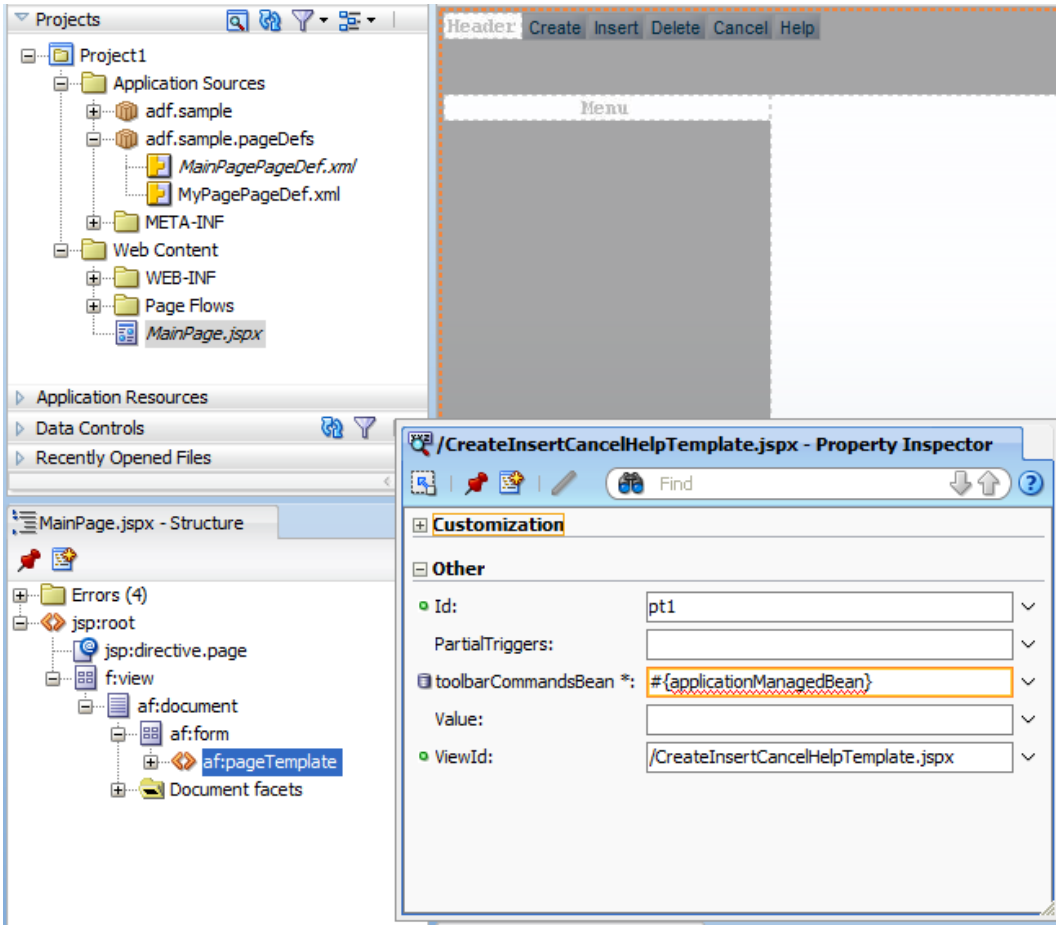
After importing the ADF Library that contains the template to the view controller project, when creating a new page, the template shows in the list of available templates:





With the template referenced from the page, I configured the **toolBarCommandBean** property to point to the managed bean in the application that implemented the interface defined by the template. This way, whenever a button in the template is pressed, the corresponding managed bean method is invoked.

This strategy allows developers to build templates with global function buttons, but also templates that have command items with a page or view specific functionality because in theory, each template usage can be supported by a different managed bean in the application.



Note: If the application managed bean is in a scope other than request, session and application, you need to prefix it with the scope. For example, if the application managed bean is configured in view scope, you would use #{viewScope.applicationManagedBean} as the value of the "toolbarCommandBean" property. As an advice for best practices: managed beans should be configured to be in the shortest possible scope.

The less state variables of a managed bean need to remember between subsequent requests the shorter the lifespan of the scope can be. If a bean only executes functionality, the scope to use is **none**, **request**, or **backingBean**. Only backing bean scope references need a prefix of "backingBeanScope".

Download

You can download the sample template I built for this article as sample #79 from the ADF Code Corner website.

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

The **Deploy** folder contains an ADF Library JAR file that I generated from the project and that you can use to try the integration of this template in a sample application. The workspace is for Oracle JDeveloper 11.1.1.4, the strategy though will work with any existing Oracle JDeveloper 11g release.