ADF Code Corner

004. Implementing auto suggest functionality in ADF Faces

twitter.com/adfcodecorner

Abstract:

No component represents Ajax and the Web 2.0 idea more than the auto suggest component. Auto suggest, or auto complete as it is referred to as well, shows a list of values in a drop down list that is filtered by the user input in a text text field. But its not the auto suggest component itself that represents the new web, its the behavior that demonstrates partial page updates the best. This articles explains an approach to implement auto suggest in ADF Faces Rich Client of Oracle JDeveloper 11g.

Note: Meanwhile ADF Faces has a suggest component that replaces what is explained in this paper (see sample #62 on ADF Code Corner.

The content is no longer needed but is published for developers who like to read source code – especially JavaScript samples for ADF Faces

Author: Frank Nimphius, Oracle Corporation
twitter.com/fnimphiu
DD-MON-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

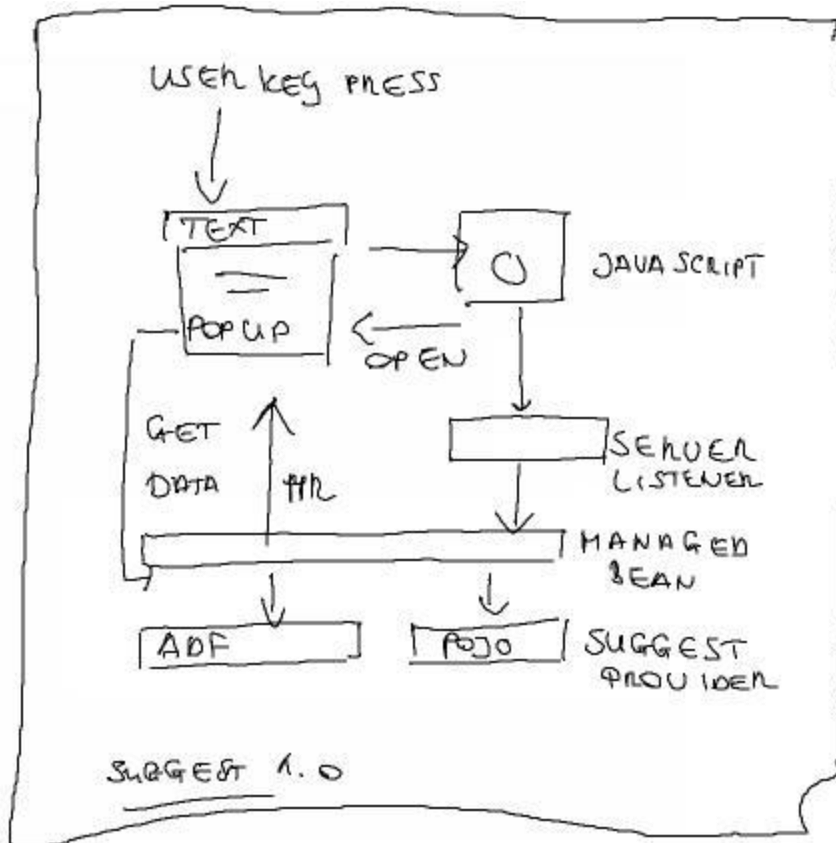
Introduction

A native auto suggest component for ADF Faces Rich Client is on the list of new features planned for a future release of Oracle JDeveloper. However, as mentioned before, its less the component than the functionality that is of interest and the functionality is what can be build with ADF Faces Rich client on-board means today. This articles explains the architecture and the JDeveloper 11*g* workspace of a first prototype that we plan to enhance to a declarative component to a later point in time. We decided to release our prototype with this article to make the suggest functionality available to ADF Faces customers in a guide for manual implementation.

Architecture

The auto suggest functionality requires a client side and a server side implementation that communicate with each other using an asynchronous communication channel. On the server side we use logic to provide the list of values and to filter the list with each keystroke of the user. The list of values is chosen to display strings as labels and values. On the client side, of course, JavaScript is used to listen for the user keyboard input, to display the suggest list and to send the user entered characters as a payload to the server to filter the data.

A Poor Man's Process Chart



In this version of auto suggest, an *af:inputTextField* is used as the data input component. As soon as the user starts typing into the field, a JavaScript event is captured by the *af:clientListener* component, which passes the event on to a JavaScript function to handle the request. The JavaScript function looks at the incoming key code and if it is a valid character, opens the popup dialog below the text field if it is not open. The list box queries the list data from a managed bean that we added as an abstraction layer between the view and the model to allow list data to be read from ADF, POJO and similar. The truth also is that we need this abstraction layer to keep the suggest functionality generic, because we want the developer to decide about the caching strategy for fetched data. In the sample we use ADF Business Components to query the data and here we have the option to tell it to perform follow up queries from memory instead of a database query. When the list data is refreshed, the list component is added as a

partial target to refresh at the end of the request. The input focus is put back to the input text field for the user to give us the next character.

ADF Business Components

The suggest list is retrieved from a client method that is exposed on the ADF Business Components Application Module. The method takes a string as an input argument to query the Jobs View Object and returns a List of strings back to the client.

Managed Bean

Two managed beans are used in this prototype: SuggestList and DemoAdfBcSuggestModel. The DemoAdfBcSuggestModel abstracts the ADF Business Components query and returns the list of strings to the SuggestList bean to provide the result to the client. If you plan to customize this sample to your needs, then you need to replace the *DemoAdfBcSuggestModel* bean with your implementation. The SuggestList provides two key methods:

public ArrayList<SelectItem> getJobList()

This method populates the suggest list. It is referenced from the *f:selectItems* element, which is a child element of the *af:selectOneListbox* element that resides in an *af:popup* component to suggest the values to the user.

```
public ArrayList<SelectItem> getJobList() {
    //first time query is against database
    if (jobList == null){
      jobList = new ArrayList<SelectItem>();
      List<String> filteredList = null;
      filteredList = suggestProvider.filteredValues(srchString, false);
      jobList = populateList(filteredList);
     }
    return jobList;
  }


 private ArrayList<SelectItem> populateList(List<String> filteredList)
{
    ArrayList<SelectItem> _list = new ArrayList<SelectItem>();
    for(String suggestString : filteredList){
      SelectItem si = new SelectItem();
      si.setValue(suggestString);
      si.setLabel(suggestString);
     _list.add(si);
    }
    return _list;
  }
```

public void doFilterList

On every user keystroke in the suggest input field, the browser client calls the *doFilterList* method through the *af:serverListener* component.

```
public void doFilterList(ClientEvent clientEvent) {
   // set the content for the suggest list
   srchString =
    (String)clientEvent.getParameters().get("filterString");
```

```
   List<String> filteredList =
        suggestProvider.filteredValues(srchString, true);
   jobList = populateList(filteredList);
   RequestContext.getCurrentInstance().addPartialTarget(suggestList);
 }
```

The *af:selectOneListbox* is refreshed at the end of each call to the *doFilterList* method so the new list value is populated

JSF page markup

The suggest field has two *af:clientListener* components assigned that listen for the keyUp and keyDown event. All keyboard events on the suggest field are send to the *handleKeyUpOnSuggestField* JavaScript function, which takes the event object as an input argument. The event object provides information about the event source, the component receiving the key stroke, and the keyCode pressed. Using the ADF Faces RC client JavaScript APIs, we don't need to deal with browser differences, which is big relief.

```
<!-- START Suggest Field -->
<af:inputText id="suggestField"
  clientComponent="true"
  value="#{bindings.JobId.inputValue}"
  label="#{bindings.JobId.hints.label}"
  required="#{bindings.JobId.hints.mandatory}"
  columns="#{bindings.JobId.hints.displayWidth}"
  maximumLength="#{bindings.JobId.hints.precision}"
  shortDesc="#{bindings.JobId.hints.tooltip}">
 <f:validator binding="#{bindings.JobId.validator}"/>
 <af:clientListener method="handleKeyUpOnSuggestField"
                  type="keyUp"/>
 <af:clientListener method="handleKeyDownOnSuggestField"
                  type="keyDown"/>
</af:inputText>
<!-- END Suggest Field -->
The suggest list is defined as follows
<!-- START Suggest Popup -->
<af:popup id="suggestPopup" contentDelivery="immediate" animate="false"
clientComponent="true">
<af:selectOneListbox id="joblist" contentStyle="width: 250px;"
                     size="15" valuePassThru="true"
                     binding="#{SuggestListBean.suggestList}">
  <f:selectItems value="#{SuggestListBean.jobList}"/>
  <af:clientListener method="handleListSelection" type="keyUp"/>
  <af:clientListener method="handleListSelection" type="click"/>
</af:selectOneListbox>
<af:serverListener type="suggestServerListener"
                   method="#{SuggestListBean.doFilterList}"/>
</af:popup>
<!-- END Suggest Popup -->
```

The af:selectOneListbox has two *af:clientListener* defined that listen for the keyUp and click event for the user to select a value (Enter key , click) or to cancel the action (Esc). The *af:serverListener* is called from the JavaScript that is executed when the user enters a new character into the suggest input field. The *af:serverListener* calls the *doFilterList* method in the *SuggestList* bean.

JavaScript

JavaScript has become popular with Ajax and also is an option to use with ADF Faces Rich Client. Unlike previous versions of ADF Faces, ADF Faces Rich Client has a real client side framework that by large is used internally, but also exposes public functions to prevent users from hacking the DOM tree.

> ***Disclaimer***
>
> Before we get into JavaScript programming, **we should mention** that using **JavaScript** in ADF Faces **is the second best solution**.
>
> The best solution always is to use the JavaServer Faces programming APIs because this is what we have good chances to see migrated from one release to the other, which also includes new browser versions that today we don't know are coming.
>
> However, some use cases, like the suggest functionality, require JavaScript to put the logic where it is needed and to save network roundtrips where possible.

In this version of Oracle JDeveloper 11*g*, JavaScript is added to the *metacontainer* facet of the *af:document* element. As soon as we see JDeveloper 11*g* R1 being productive, this part of the code may be changed to the new *af:resource* element.

```
<f:facet name="metaContainer">
  <af:group>
    <![CDATA[
      <script>
   //*******************************************************************
   //PROTOTYPE: AUTO SUGGEST
   //STATUS: 1.0
   //authors: Frank Nimphius, Maroun Imad
   //Functionality
   //=====================================
   // Implemented in v 1.0
   //1 - Suggest window opens JOB_ID field
   //2 - Initial query is to database, follow up queries are in memory
   //3 - Enter list with down arrow
   //4 - Enter key and mouse click to select
   //5 - ESC to close
   //*******************************************************************

   function handleKeyUpOnSuggestField(evt){
     // start the popup aligned to the component that launched it
     suggestPopup = evt.getSource().findComponent("suggestPopup");
     inputField = evt.getSource();

     //don't open when user "tabs" into field
     if (suggestPopup.isShowing() == false &&
       evt.getKeyCode()!= AdfKeyStroke.TAB_KEY){
       //We reference a DOM area by using getClientId()+"::content"
       //to position the suggest list. Shame
       //on use that we are doing this, but there is no other option
```

```
       //we found. Keep in mind that using direct DOM references may
       //break the code in the future if the rendering of the
       //component changes for whatever reason. For now, we want to
       //feel safe and continue using it
       hints = {align:AdfRichPopup.ALIGN_AFTER_START,
                alignId:evt.getSource().getClientId()+"::content"};
       suggestPopup.show(hints);
       //disable popup hide to avoid popup to flicker on key press.
       //Note that we override the default framework behavior of the
       //popup component for this instance only. When hiding the popup,
       //we re-implement the default functionality. By all means, never
       //change the framework functionality on the prototype level as
       //this could have an unpredictable impact on all nstances of
       //this type
       suggestPopup.hide = function(){}
     }

    //suppress server access for the following keys
    //for better performance
    if (evt.getKeyCode() == AdfKeyStroke.ARROWLEFT_KEY ||
      evt.getKeyCode() == AdfKeyStroke.ARROWRIGHT_KEY ||
      evt.getKeyCode() == AdfKeyStroke.ARROWDOWN_KEY ||
      evt.getKeyCode() == AdfKeyStroke.SHIFT_MASK ||
      evt.getKeyCode() == AdfKeyStroke.END_KEY ||
      evt.getKeyCode() == AdfKeyStroke.ALT_KEY ||
      evt.getKeyCode() == AdfKeyStroke.HOME_KEY) {
      return false;
   }
   //close the popup when teh ESC key is pressed
   if (evt.getKeyCode() == AdfKeyStroke.ESC_KEY){
     hidePopup(evt);
     return false;
   }

   // get the user typed values
   valueStr = inputField.getSubmittedValue();
   // query suggest list on the server. The custom event references the
   //af:serverListener component which calls a server side managed bean
   //method. The payload we send is the current string entered
   // in the suggest field
   AdfCustomEvent.queue(suggestPopup,"suggestServerListener",
     // Send single parameter
     {filterString:valueStr},true);
   // put focus back to the input text field. We set the timout to 400
   //ms, which is not the final answer but a value that worked for us.
   //In a next version, we may improve this
   setTimeout("inputField.focus();",400);
 }

 //TAB and ARROW DOWN keys navigate to the suggest popup we need to
 //handle this in the key down event as otherwise the TAB doesn't work.
 //Note that a TAB key cannot be used to navigate in the suggest list.
 //Use the arrow keys for this. The TAB key can be done to navigate the
 //list, but this requires some more JavaScript that we didn't complete
 //in time
 function handleKeyDownOnSuggestField(evt){
  if (evt.getKeyCode() == AdfKeyStroke.ARROWDOWN_KEY) {
```

```
      selectList   = evt.getSource().findComponent("joblist");
      selectList.focus();
      return false;
    }
    else{
      return false;
    }
  }

//method called when pressing a key or a mouse button on the list. The
//ENTER key or a mosue click select the list value and write it back to
//the input text field (the suggest field)
 function handleListSelection(evt){
  if(evt.getKeyCode() == AdfKeyStroke.ENTER_KEY ||
      evt.getType() == AdfUIInputEvent.CLICK_EVENT_TYPE){
      var list = evt.getSource();
      evt.cancel();
      var listValue = list.getProperty("value");
      hidePopup(evt);
      inputField = evt.getSource().findComponent("suggestField");
      inputField.setValue(listValue);
    }
    //cancel dialog
    else if (evt.getKeyCode() == AdfKeyStroke.ESC_KEY){
      hidePopup(evt);
    }
  }

//function that re-implements the node functionality for the popup
//to then call it Please do as if you've never seen this code ;-)
 function hidePopup(evt){
  var suggestPopup = evt.getSource().findComponent("suggestPopup");
    //re-implement close functionality
    suggestPopup.hide = AdfRichPopup.prototype.hide;
    suggestPopup.hide();
  }
  </script>
  ]]>
  </af:group>
</f:facet>
```

What's Next

This is a first proofpoint that austosuggest functionality can be done with ADF Faces Rich Client. We are planning to continue improving this prototype and a little road map is available in the page source code of the sample. The most important improvement is to create a declarative component from this sample that is parameterizable and that allows developer to use multiple suggest instances on a single page without any java or JavaScript conflict. To this time we are open for feedback regarding new features and bug reports, but also do appreciate receiving code samples that have improvements implemented and bugs fixed.

Download Sample

The Oracle JDeveloper 11*g* sample works against the HR database schema and can be downloaded from the ADF Code Corner website:

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

Note: Meanwhile ADF Faces has a suggest component that replaces what is explained in this paper. The content is no longer needed but is published for developers who like to read source code – especially JavaScript samples for ADF Faces

RELATED DOCOMENTATION

☒	af:autoSuggestBehavior component - http://download.oracle.com/docs/cd/E15523_01/apirefs.1111/e12419/tagdoc/af_autoSuggestBehavior.html
☒	
☒	