Web Dynpro Best Practices:

How to Employ Google Search Functionality in Web Dynpro

Based on SAP NetWeaver™ ’04 Stack 11

Jochen Guertler

Index and Table of Contents

Introduction

Besides the adaptive RFC model, which allows you to access any R/3 backend from your Web Dynpro application, there are alternative model types that offer data access.

One of the most powerful models is the Web Service model, which allows you to access any web service within your Web Dynpro application. It is completely declarative and code-free - at least for simple web services. A good example is the Google web service, which allows you to search the Web using Google.

The following document describes how to create the model and how to realize a neat and usable UI for the Google search results within Web Dynpro.

Prerequisites

The Google Web Service

The first step to use the Google web service is to download the Google Web APIs developer's kit located at http://www.google.com/apis/. You must also create a user account (this is free) to get the license key that entitles you to 1,000 automated queries per day.

The developer's kit contains several examples, some Java wrappers that could be used to access the Google search, and the necessary WSDL file defining the structure of the Google web service. We will use this WSDL file to create our Google web service model later.

Editor's Note: The project file with source is available on SDN at:

https://www.sdn.sap.com/irj/servlet/prt/portal/prtroot/com.sap.km.cm.docs/business_packages/a1-8-4/BestPractices_AccessGoogle.zip

You must be logged in to access the file.

The Web Dynpro Project "`BestPractices_AccessGoogle`"

The attached Web Dynpro project `BestPractices_AccessGoogle` contains the full, running version of the Web Dynpro application presented here. You can directly import it into your SAP NetWeaver Developer Studio, rebuild, deploy, and run it.

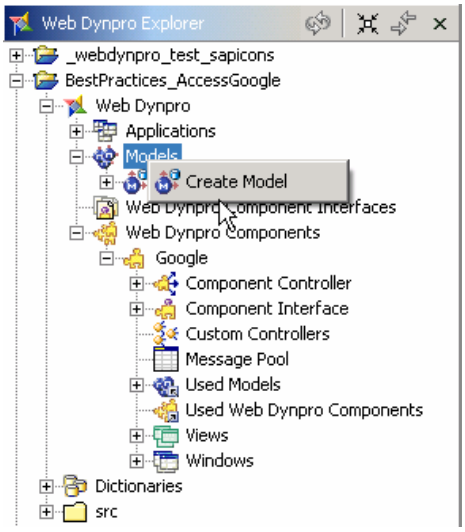
The only thing you must change inside the project is the definition of the license key. Open the implementation tab for the component controller of the `Google` component and browse to the `wdDoInit()` method. You must add your license key at the appropriate code line:

```
//@@begin javadoc:wdDoInit()
/** Hook method called to initialize controller. */
//@@end
public void wdDoInit()
{
  //@@begin wdDoInit()

  // Initialize the search
  Request_GoogleSearchPort_doGoogleSearch searchRequest =
    new Request_GoogleSearchPort_doGoogleSearch();
  searchRequest.setKey("OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO");

  searchRequest.setMaxResults(10);
  searchRequest.setStart(0);

  wdContext.nodeRequest_GoogleSearchPort_doGoogleSearch().bind(
    searchRequest);

  //@@end
}
```

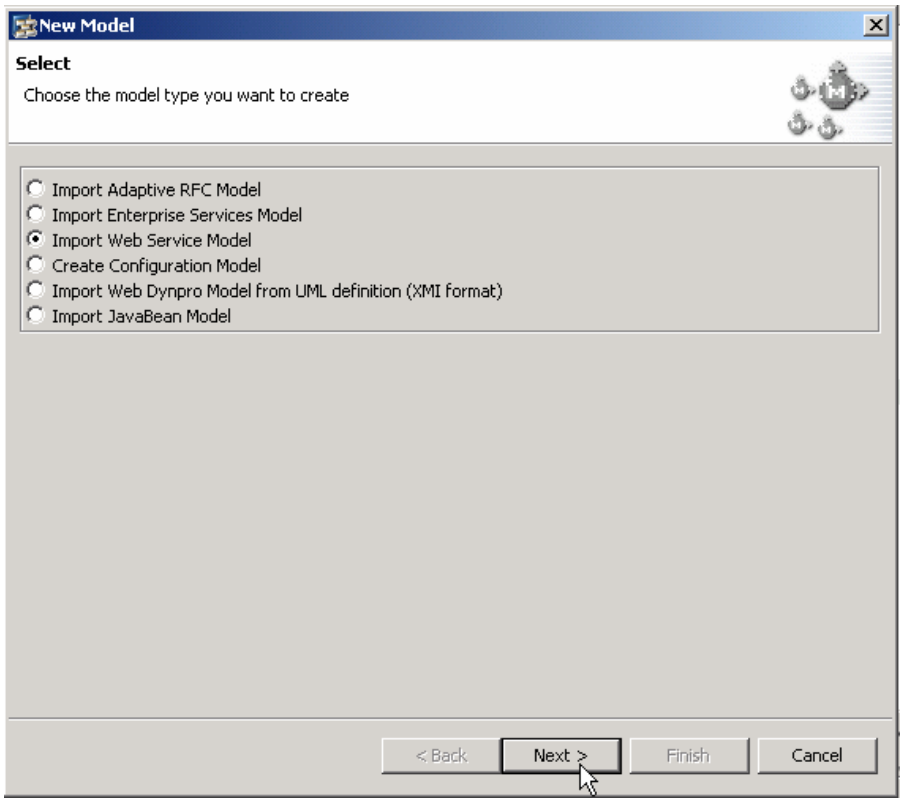
The following document will not describe the steps to create the sample application in detail. Only the important and interesting aspects are covered in the following sections.

Creating the Google Model

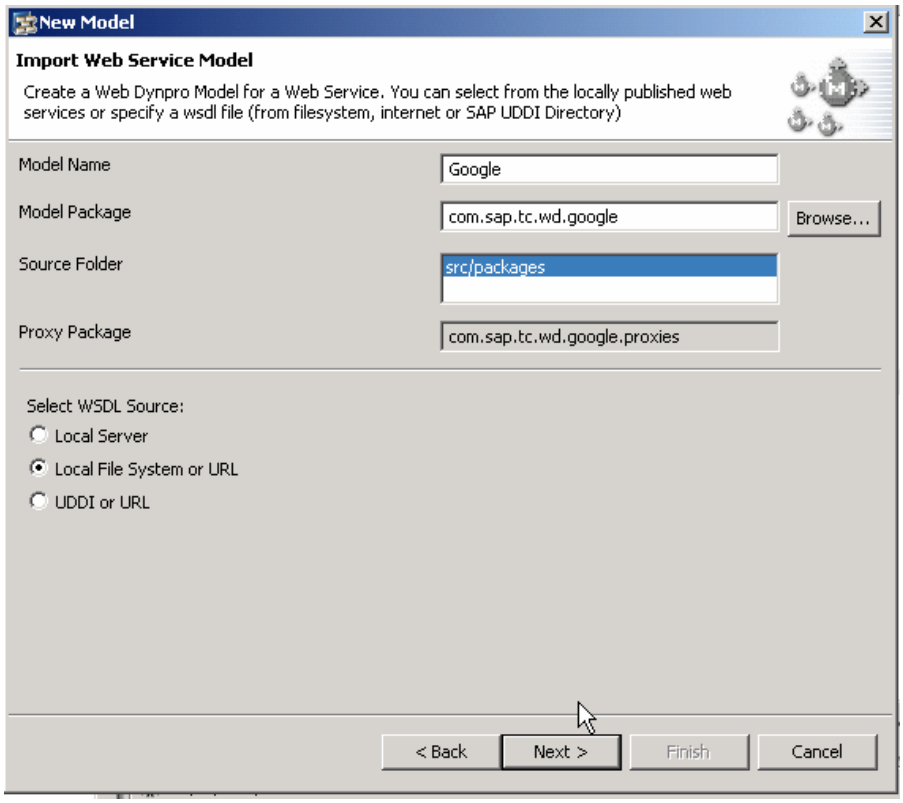
The first step to access Google is to create the necessary web service model. To do that, open the model creation dialog using the context menu of the `Models` node of your Web Dynpro project.



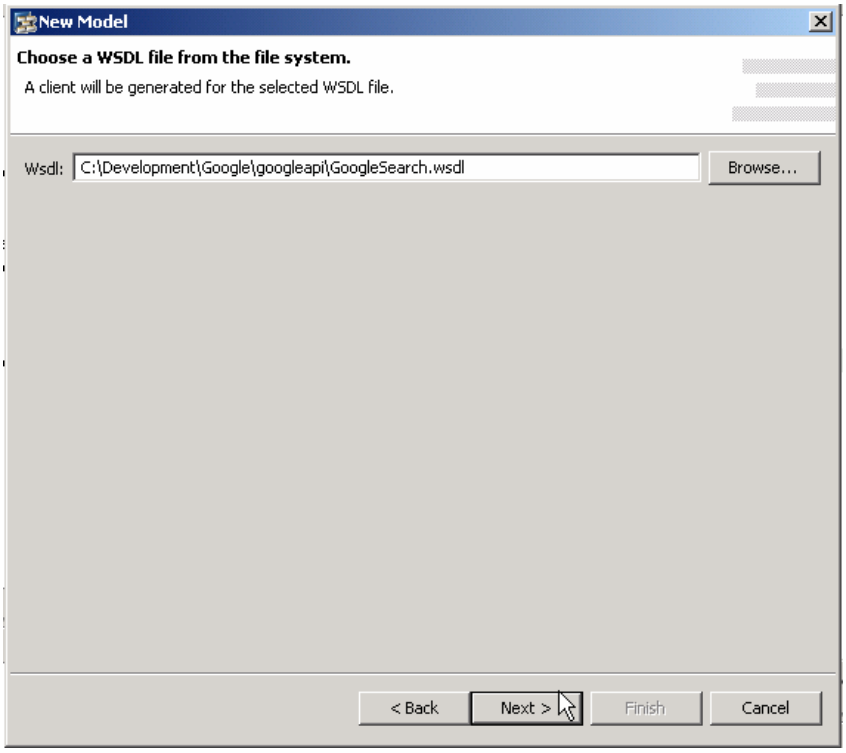
You have to select the `Import Web Service Model` in the first wizard step.



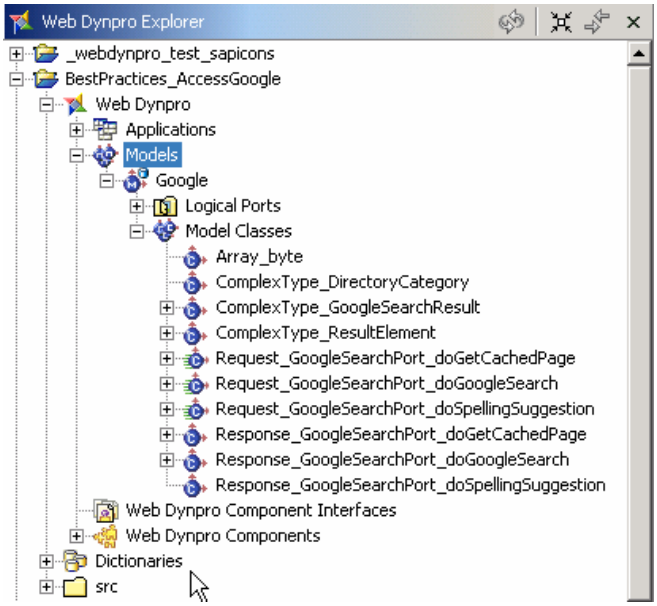
After that, define the name and the package of the model (make sure that the defined package is empty). Furthermore, define the location of your WSDL source. In our example we use the `Local File System or URL` (since we have the downloaded WSDL file in the file system).



After that, select the WSDL file from its location in your file system.



After the next step you can press the Finish button and your Web Dynpro Google model is created automatically based on the defined WSDL file.

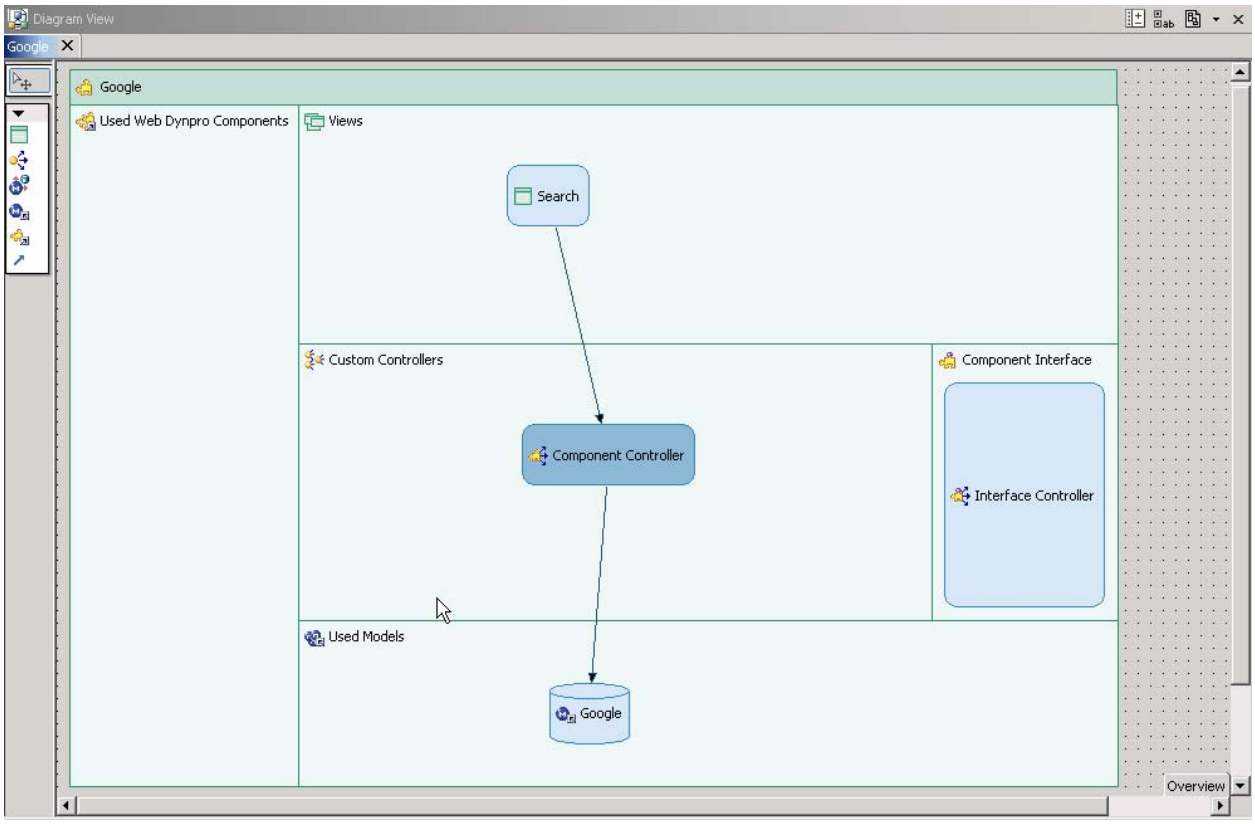


If you have a deeper look at the model created, you recognize that there are three different tasks that you can perform with this model. We will concentrate on the “standard” Google search, which is realized by the `Request_GoogleSearchPort_doGoogleSearch` model class.

The two other model classes (`Request_GoogleSearchPort_doGetCachedPage` and `Request_GoogleSearchPort_doSpellingSuggestion)` are not used in this example - but please feel free to play around with it.

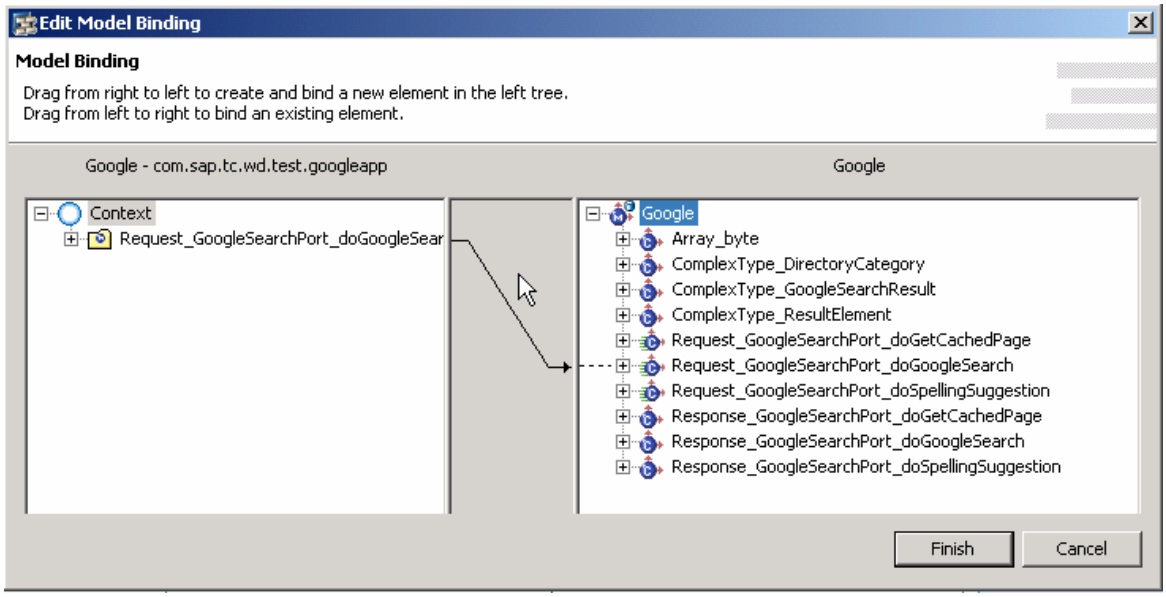
Connecting the Web Dynpro Application to the Google Model

The first step to use the Google model is to connect it to the Web Dynpro application. To do so, start the data modeler of the `Google` component by double-clicking the associated component node in the Web Dynpro explorer.



Use the `Create a model` icon on the left side to add the Google model to the component. After that, create a data link between the Google model and the component controller as displayed in the screenshot. Doing this brings up a dialog to edit the model binding. Drag the `Request_GoogleSearchPort_doGetSearch` node from the right side to the left and drop it under the Context node of the component controller.

The following screenshot shows the result. The context of the component controller now contains the complete context structure necessary for the Google search.



You now have a connection from the component controller context to the model. The next step is to create the necessary model objects and to bind it to the context nodes. Doing this makes the context nodes “alive,” and you can use it inside your Web Dynpro application to both define the input parameters and to get the search result items.

The creation of the necessary model objects and the binding to the context node is done in the `wdDoInit()` method of the Google component controller.

```
//@@begin javadoc:wdDoInit()
/** Hook method called to initialize controller. */
//@@end
public void wdDoInit()
{
  //@@begin wdDoInit()

  // Initialize the search
  Request_GoogleSearchPort_doGoogleSearch searchRequest =
    new Request_GoogleSearchPort_doGoogleSearch();
  searchRequest.setKey("00000000000000000000000000000000");

  searchRequest.setMaxResults(10);
  searchRequest.setStart(0);

  wdContext.nodeRequest_GoogleSearchPort_doGoogleSearch().bind(
    searchRequest);

  //@@end
}
```

The first step is to create an instance of the `Request_GoogleSearchPort_doGoogleSearch` model class. Use this model class instance to define all input parameters that should not be visible or changeable by the user later (like the license key).

Furthermore, you can also define default values for all input parameters that could be changed by the user (like the maximum number of search results).

Then bind the model class instance to the context node of the component controller. This defines the connection between the context and the model. This connection is responsible for both passing the input parameters to the model class instance and for reading the results of the Google search.

Defining the Search Query

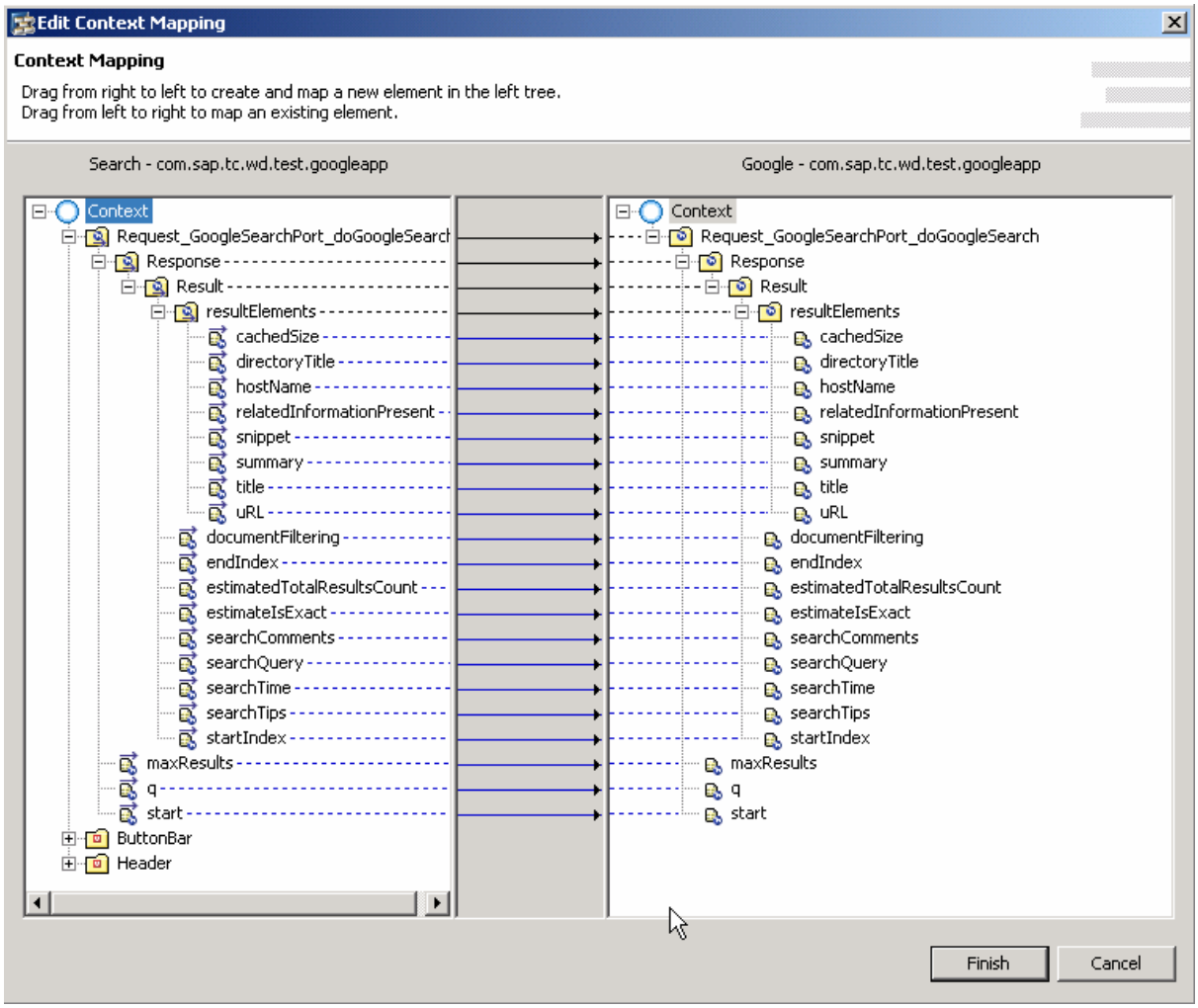
To define the search query, we define a simple UI containing the necessary input field and one button to trigger the Google search. The following screenshot displays this.

From a technical point of view, we must do the following:

- Create the `Search` view with the necessary UI elements.
- Define the context mapping between the context of the Google component controller and the view controller context to make it accessible to the view. This can be done again by

using the data modeler. Create a data link between the component controller and the view controller and define the context mapping.

The following screenshot demonstrates the result.



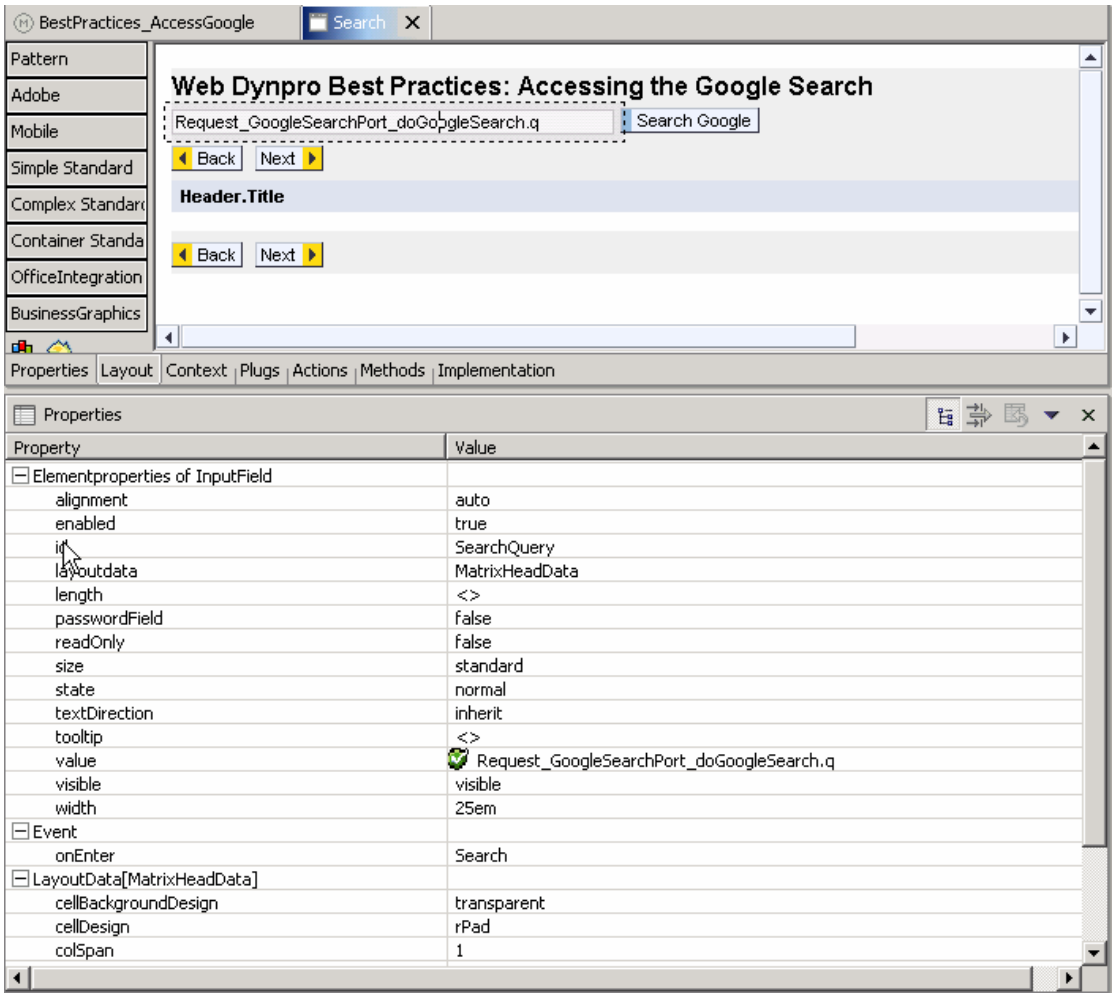
As you can see, the complete view controller context is mapped to the component controller context.

Tip: You should never map a view controller context directly to a model class.

In doing this, you lose the possibility of exchanging the model class later without changing the view controller. Instead use a custom controller context or a component controller context to wrap the model class used.

It is always a good idea to create a special "**model component**" (i.e., a Web Dynpro component defining no UI (no interface view), but providing access to the underlying model via the interface controller context) to wrap the model class used.

For the definition of the query we have to fill the `q` attribute of the `Request_GoogleSearchPort_doGoogleSearch` context node. Therefore, we just bind the input field `SearchQuery` to this context attribute.


BestPractices_AccessGoogle
Search
Pattern
Adobe
Mobile
Simple Standard
Complex Standard
Container Standa
OfficeIntegration
BusinessGraphics
Web Dynpro Best Practices: Accessing the Google Search
Request_GoogleSearchPort_doGoogleSearch.q
Search Google
Back
Next
Header.Title
Back
Next
Properties
Layout
Context
Plugs
Actions
Methods
Implementation
Properties
Property
Value
Elementproperties of InputField
alignment
auto
enabled
true
id
SearchQuery
layoutdata
MatrixHeadData
length
<>
passwordField
false
readOnly
false
size
standard
state
normal
textDirection
inherit
tooltip
<>
value
Request_GoogleSearchPort_doGoogleSearch.q
visible
visible
width
25em
Event
onEnter
Search
LayoutData[MatrixHeadData]
cellBackgroundDesign
transparent
cellDesign
rPad
colSpan
1

Triggering the Google Search

To trigger the Google search, the only thing that has to be done is to create the `SearchGoogle` action and to bind this action to the `onAction` event of the button.

The action event handler looks like this:

```
//@@begin javadoc:onActionSearch(ServerEvent)
/** Declared validating event handler. */
//@@end
public void onActionSearch(com.sap.tc.webdynpro.progmodel.api.IWDCustomEvent wdEvent )
{
  //@@begin onActionSearch(ServerEvent)

  // Reset the page index for the new search ...
  updateSearchIndex(1);

  // ... and start the search
  wdThis.wdGetGoogleController().doGoogleSearch();

  if (wdContext.currentResultElementsElement() == null) {
    wdComponentAPI.getMessageManager().reportException(
      "No search results found for '"
        + wdContext.currentRequest_GoogleSearchPort_doGoogleSearchElement().getQ()
        + "'",
      true);
  }
  //@@end
}
```

As you can see, the search is triggered by calling the `doGoogleSearch()` method of the Google component controller. The implementation of this method looks like this:

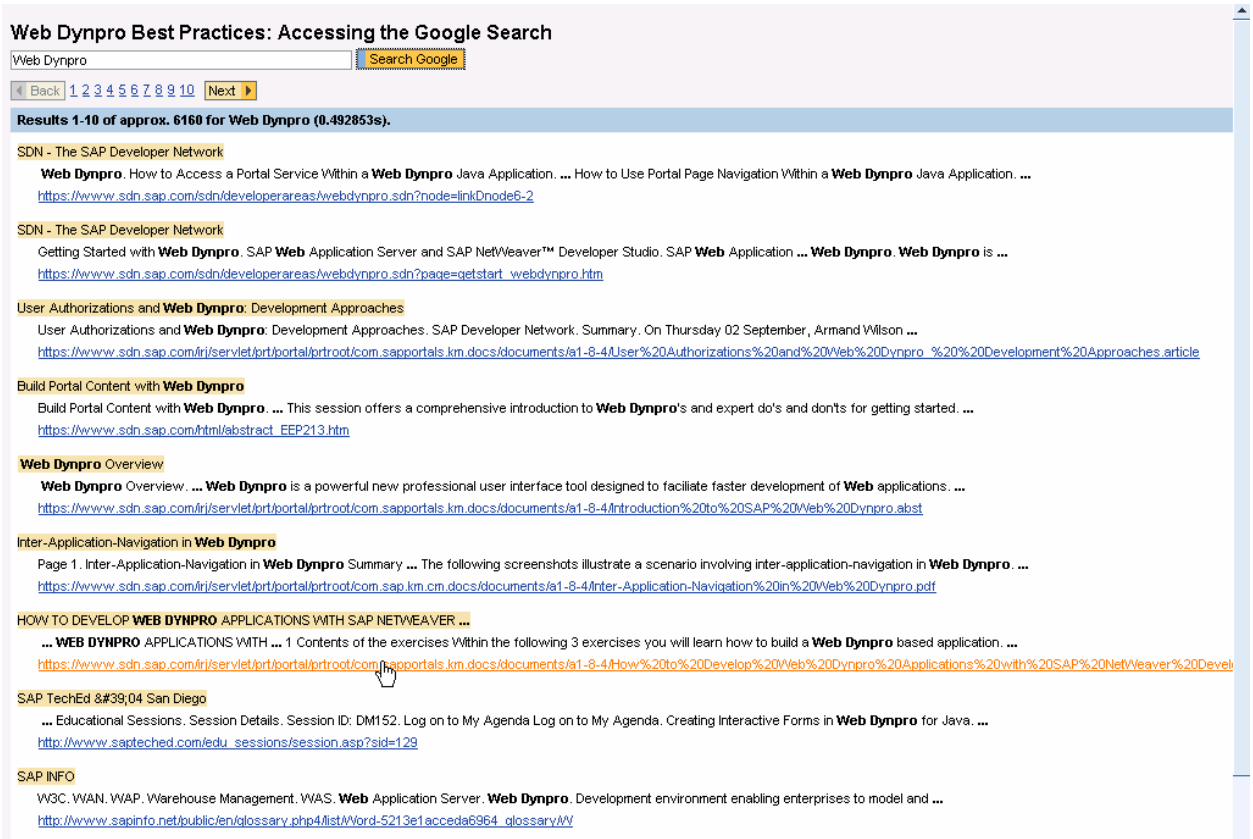
```
//@@begin javadoc:doGoogleSearch()
/** Declared method. */
//@@end
public void doGoogleSearch() {
  //@@begin doGoogleSearch()
  try {
    wdContext.currentRequest_GoogleSearchPort_doGoogleSearchElement().modelObject().execute();
    wdContext.nodeResponse().invalidate();
  } catch (Exception e) {
    wdComponentAPI.getMessageManager().reportException(e.getLocalizedMessage(), true);
  }
  //@@end
}
```

First, the model object is executed. The context node is used to gain access to the necessary model object instance. This is possible because we bound this model class instance to the context node in the `wdDoInit()`. Second, the `Response` context node is invalidated to make sure that the new response data of the model execution is filled into the context node.

Then, the Google search is successfully executed and the resulting data is automatically filled into the `Response` context node. You can now use this context node to display the search results.

Displaying the Google Search Results

The following screenshot shows the final display of Google search results:



To create this look, we need to apply some tricks:

- The Google search result strings are HTML-compliant. Therefore, we need a way to render an HTML-compliant string using the standard Web Dynpro UI elements.
- Because the list of search result items is dynamic, we must create the display of each result item dynamically.

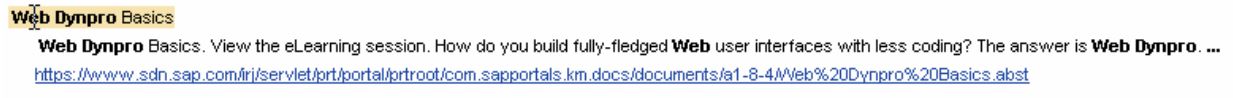
- To browse between the different search results pages, we must create a navigation bar that allows us to access all available search result items using the navigation buttons or the page index links.

The next two sections describe all these topics.

Displaying the HTML Results Strings

Let's first look at the display of a single item from the search results.

First, we have the search item title containing the searched phrase (as emphasized text). Second, there is a short section of the document. The search phrase is again emphasized. Third, there is the URL pointing to the document. Clicking this link opens a new browser window with the page.



To render both the title and the short section of the document, we need a way to render an HTML-compliant string.

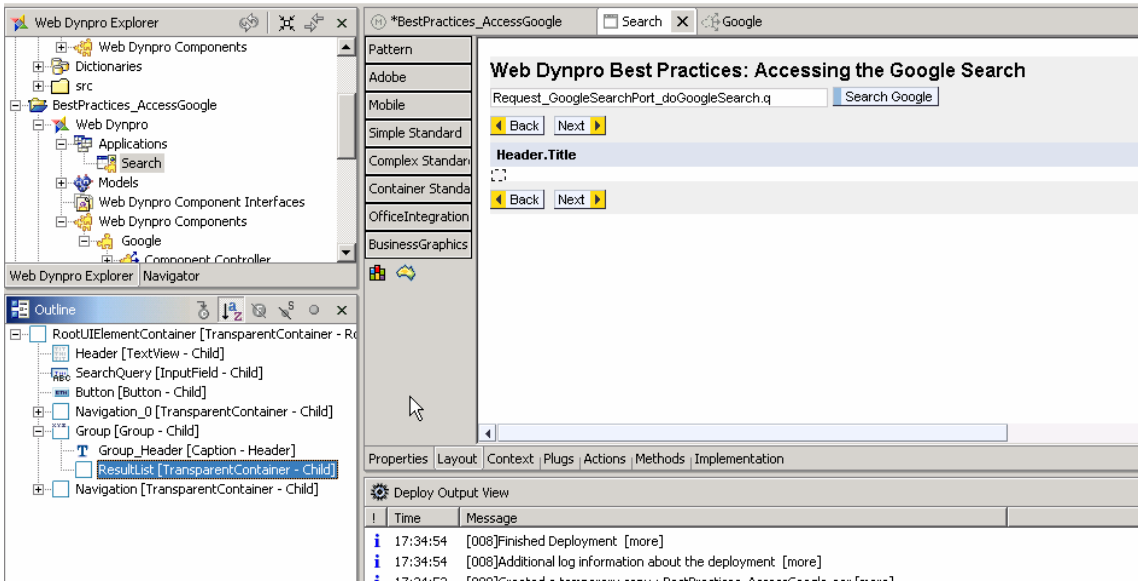
In SAP NetWeaver '04, there is no such Web Dynpro UI element available. The only thing we have is the simple `TextView` UI element that allows us to render text with a specific layout and color.

The idea now is simple (as most great ideas are): We use the `TextView` UI element to render the HTML-compliant strings by splitting the whole HTML string into single fragments (defined by the different HTML tags). All the fragments are rendered by different `TextView` UI elements, which are put together in one `TransparentContainer` UI element to render the whole HTML string. All this is done dynamically. That's it.

So, here are the details.

First we have to add the necessary `TransparentContainer` UI element to the search results. We do NOT do this dynamically, although it would be possible to do so. You should declare your UI whenever possible to avoid performance problems during rendering.

As you can see on the following screenshot we declare more or less the complete search result UI. Only the really dynamic parts, i.e. the list of the single search results are added later on using dynamic programming.



Tip: Declare as much as possible parts of the UI. Use dynamic programming only for the truly "dynamic" parts.

After declaring the necessary `TransparentContainer` UI element, we have to fill it with the search results items. As discussed earlier, we must display HTML-compliant strings. We do this in the following method `createHTMLViewer`.

```
private static void createHTMLViewer(
  String htmlfragment,
  IWDTransparentContainer container,
  IWDView view,
  WDTextViewSemanticColor color) {

  // Reset the container
  container.removeAllChildren();

  // Delete the line breaks
  htmlfragment = StringUtil.searchAndReplace(htmlfragment, "<br>", "");

  // Divide the HTML string
  String[] parts = StringUtil.divide(htmlfragment, '<');

  WDTextViewDesign design = null;
  String part = null;

  // Loop the list of fragments and display the bold fragments as EMPHASIZED
  for (int i = 0; i < parts.length; i++) {

    if (parts[i].startsWith("b>")) {
      design = WDTextViewDesign.EMPHASIZED;
    } else {
      design = WDTextViewDesign.STANDARD;
    }

    part = StringUtil.searchAndReplace(parts[i], "/b>", "");
    part = StringUtil.searchAndReplace(part, "b>", " ");
    part = StringUtil.searchAndReplace(part, "&quot;", "'");

    // Create the needed TextView UI element
    addHTMLPart(part, container, view, design, color);
  }
}
```

We must pass the HTML-compliant string and the `TransparentContainer` UI element instance storing the different `TextView` UI element instances. Furthermore, we need the Web Dynpro `View` instance to create new `TextView` UI element instances and the color that should be used for the text.

The method first prepares the HTML-compliant string by deleting any line breaks. After that, we split the whole string into different fragments. We use the "<" character to get string fragments based on the HTML tags. Loop this list of fragments and check for fragments that should be displayed as emphasized text. The last step is to delete the HTML end tags.

At the moment, this method obviously does not cover all HTML tags – but feel free to extend this method to handle more.

The last step is to call the `addHTMLPart` method for each HTML fragment to create the necessary `TextView` UI elements. This method looks like this:

```
private static void addHTMLPart(
  String htmlPart,
  IWDTransparentContainer container,
  IWDView view,
  WDTextViewDesign design,
  WDTextViewSemanticColor color) {

  IWDTextView textView =
    (IWDTextView) view.createElement(IWDTextView.class, String.valueOf(textViewId++));
  textView.setText(htmlPart);
  textView.setDesign(design);
  textView.setSemanticColor(color);
  container.addChild(textView);
}
```

There is nothing special in it – we just create a new `TextView` UI element instance and define the text, the design, and the color. In the last step we add this new instance to the passed `TransparentContainer` UI element instance.

Provide a Search Navigation Bar

The following screenshot displays the navigation bar.

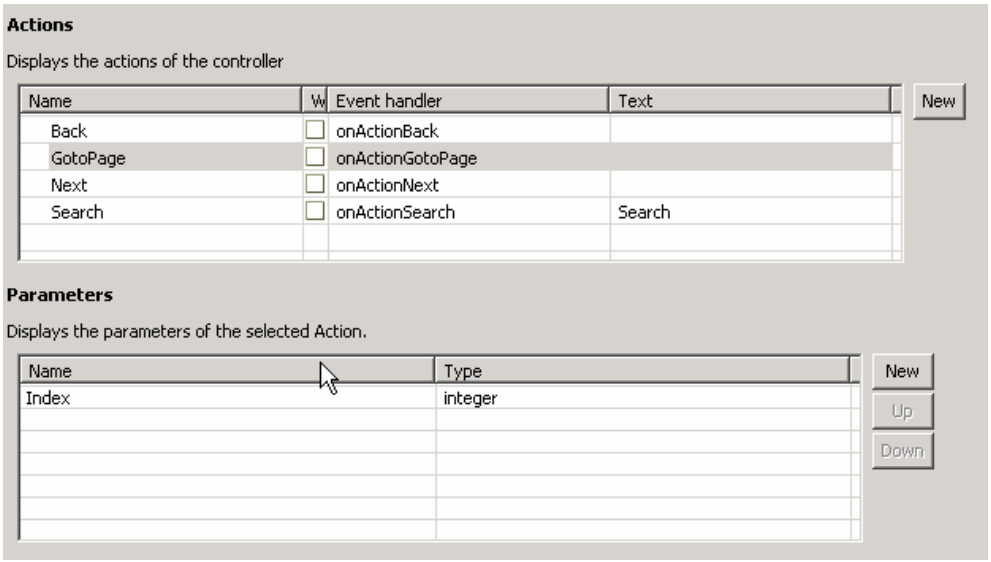
It consists of two main parts. There is a page index list that allows you to jump to a specific search results page. Besides that, there are two buttons to navigate back and forward through the list of search results pages.

Create the Page Index List

To create the page index list we call the `createPageIndexList` method displayed below.

```
private static void createPageIndexList(
  IPrivateSearch wdThis,
  int startIndex,
  IWDTransparentContainer indexList,
  IWDView view) {

  IWDLinkToAction link = null;

  // Create ten page indexes
  for (int i = 0; i < 10; i++) {

    // Create the LinkToAction UI element
    link = (IWDLinkToAction)
      view.createElement(IWDLinkToAction.class, "PageIndexLink" + globalIndex++);

    // Use parameter mapping to define the index of the page
    link.setOnAction(wdThis.wdGetGotoPageAction());
    link.mappingOfOnAction().addParameter("Index", startIndex + i);

    link.setText(String.valueOf(startIndex + i));
    link.createLayoutData(IWDMatrixData.class);

    indexList.addChild(link);
  }
}
```

It is important to use dynamic parameter mapping so the `index` parameter uses the same Web Dynpro action `GotoPage` for all displayed `LinkToAction` UI element instances. The following screenshot shows the definition of the `GotoPage` action.



The `onActionGotoPage` event handler of the `GotoPage` action calls the `updateSearchIndex` method passing the requested page index.

```
private void updateSearchIndex(int pageIndex) {
  wdContext.currentButtonBarElement().setPageIndex(pageIndex);
  wdContext.currentRequest_GoogleSearchPort_doGoogleSearchElement().setStart((pageIndex - 1) * 10);
}
```

The `updateSearchIndex` method updates the start index for the next Google search.

Create the Navigation Buttons

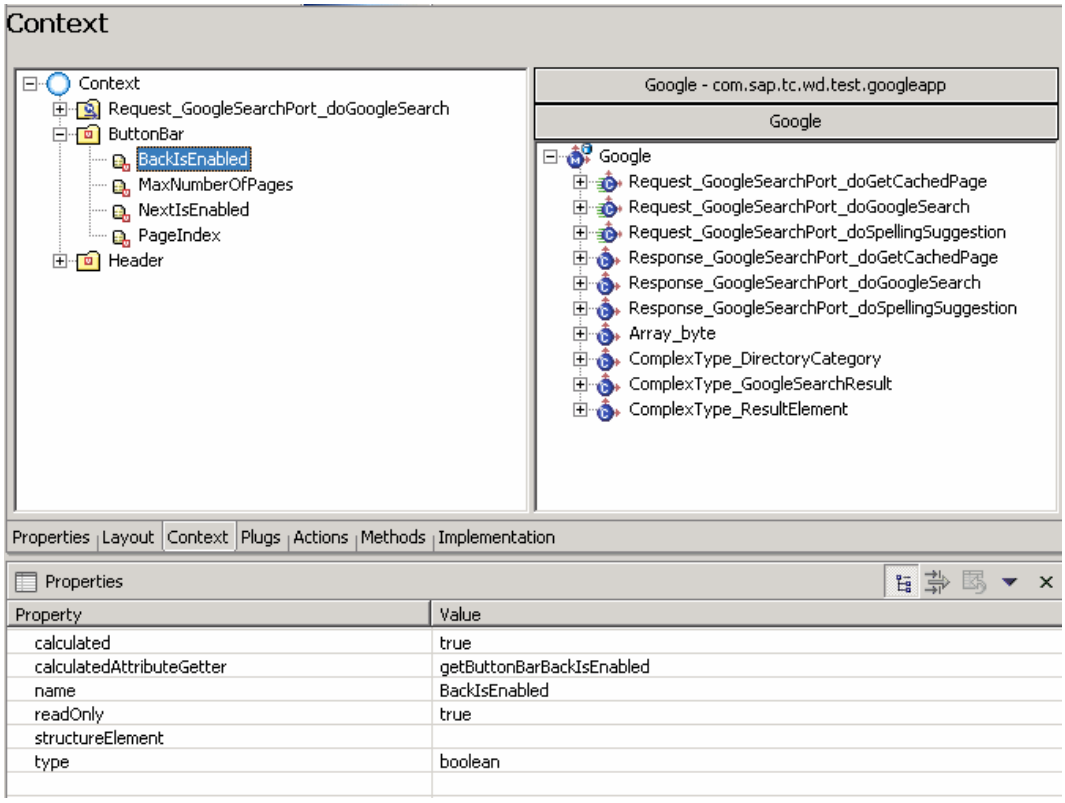
To ensure that the back and forward navigation buttons are correctly enabled, we define the `ButtonBar` context node in the Search component controller context. There are different context nodes attributes defining whether or not the buttons are enabled. These attributes are attributes.

For example, the context node attribute `BackIsEnabled` is calculated by the method `getButtonBarBackIsEnabled`.

```
//@@begin javadoc:getButtonBarBackIsEnabled(IPrivateSearch.IButtonBarElement)
/**
 *  Declared getter method for attribute BackIsEnabled of node ButtonBar
 *  @param element the element requested for the value
 *  @return the calculated value for attribute BackIsEnabled
 */
//@@end
public boolean getButtonBarBackIsEnabled(IPrivateSearch.IButtonBarElement element)
{
  //@@begin getButtonBarBackIsEnabled(IPrivateSearch.IButtonBarElement)
  return (wdContext.currentButtonBarElement().getPageIndex() > 1);
  //@@end
}
```

Tip: Use calculated attributes to define the values of context attributes that depend on some other "data," like other context attributes values.

The following screenshot shows the definition of the `ButtonBar` context node.



Summary

This document presented an example of using the Web Dynpro Web Service Model. We executed a Google search and displayed the search results in a tidy manner.

Keep in mind the following:

- Do not bind any view controller context directly to a model class. Use model components or custom controllers to wrap the model used.
- Only create the truly dynamic parts of your UI using dynamic programming. Declare as much as possible.
- Use calculated attributes to define context attributes with values that are depend on other data.