UWL
Connector API

Connecting SAP Office Mail to the Universal Worklist

Table of Contents

1 Scenario

You want to show SAP Business Workflow and other generic object service notifications in the SAP NetWeaver Portal Universal Worklist.

If you experience any problems with this How-To paper, please address your questions to the Business Process Management (BPM) forum on the SAP Developer Network website. You can go directly to the forum via the following link:

https://www.sdn.sap.com/irj/sdn/forum?forumID=146

If you are not yet registered with the SAP Developer Network you can register here: http://www.sdn.sap.com.

2 Introduction

The Universal Worklist (UWL) enables SAP NetWeaver Portal users to manage their work by bringing together notification items from different systems. The UWL delegates the task of connecting to a backend system and retrieving items to a so called *UWL Connector*. The following UWL Connectors are delivered together with the SAP NetWeaver Portal:

- *WebFlow Connector* – fetches work items from SAP Business Workflow.
- *AdHocWorkflow Connector* – fetches collaboration tasks from SAP AdHoc Workflow.
- *Alert Connector*- fetches Alerts and follow-up activities from SAP Alert Management System.
- *ActionInbox Connector*- fetches document related notifications from SAP Knowledge Management.
- *Generic ABAP Connector* – For ABAP based providers, UWL simplifies the connector API by doing much of the plumbing work in this Generic ABAP connector. The ABAP based provider needs to implement the interface IF_UWL_ITEM_PROVIDER to supply items to UWL.

Notification items from other systems can displayed in the UWL as well, provided you have an appropriate connector that knows how to retrieve these items from the respective system. The UWL provides a Java API for the development of such a connector.

This document provides you with a step-by step guide on how to develop an UWL Connector for SAP Office Mail. The connector supports the following features.

Features:

- Display of SAP Office Mail inbox items in the UWL
- Forwarding of SAP Office Mail items
- Retrieval of Attachments

Due to the fact that the connector was developed mainly for training purposes the implementation is not complete and provides room for improvements. The main restrictions of the implementation detailed in this How-To paper are listed below.

Restrictions:

- A mail can only be forwarded once
- Only one attachment item is displayed
- In the item body only RAW & TXT can be displayed
- Only basis system > 4.7 supported due to the use of Web Services

The screenshot below shows the SAP Business Workplace with the SAP Office Mail inbox. Using the connector form this guide you can display the same inbox items in the UWL.

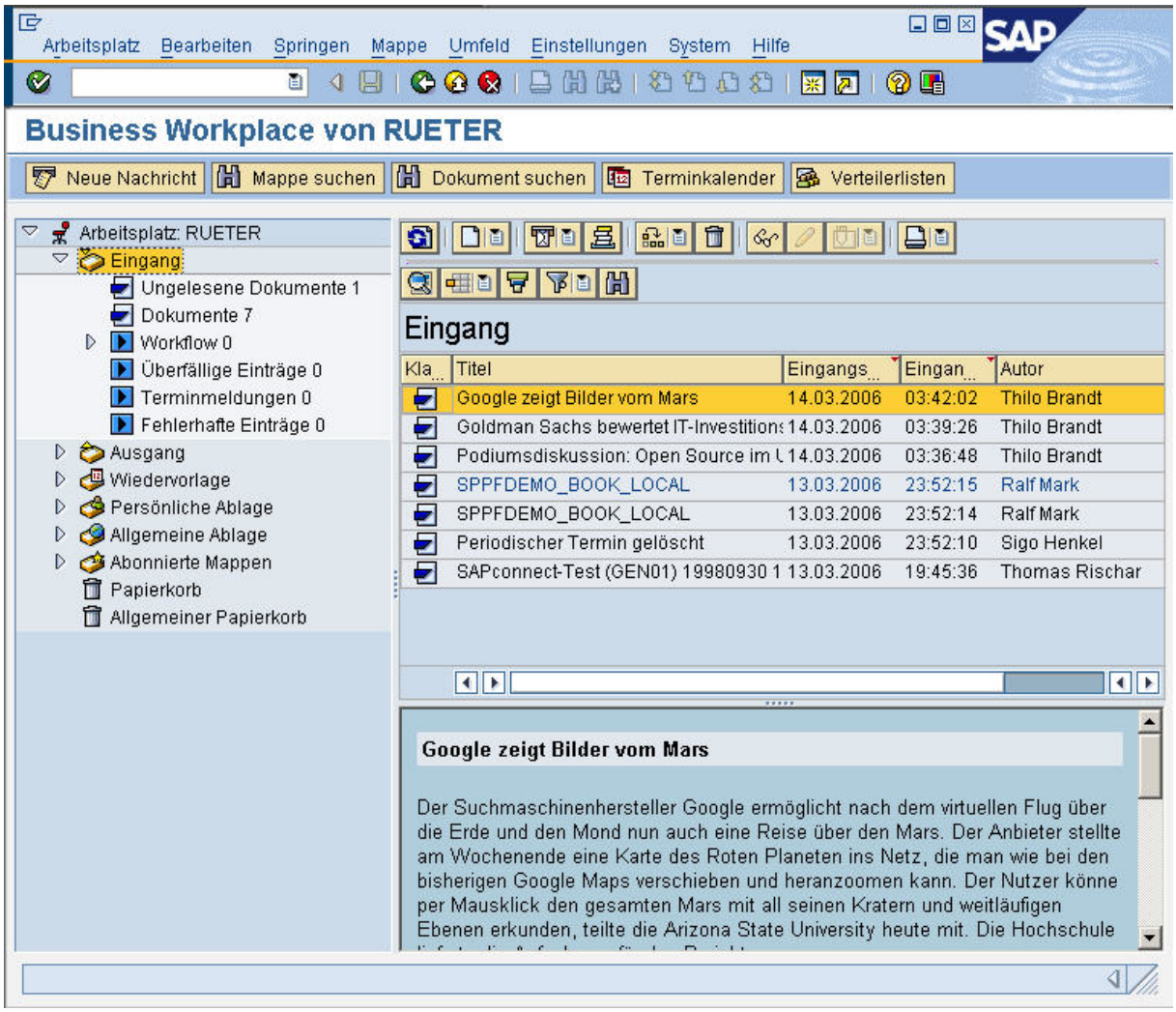

Figure 1: SAP Office Mail

When displaying SAP Office Mail items in the UWL you can also conveniently display and download attachments, forward and delete inbox items.

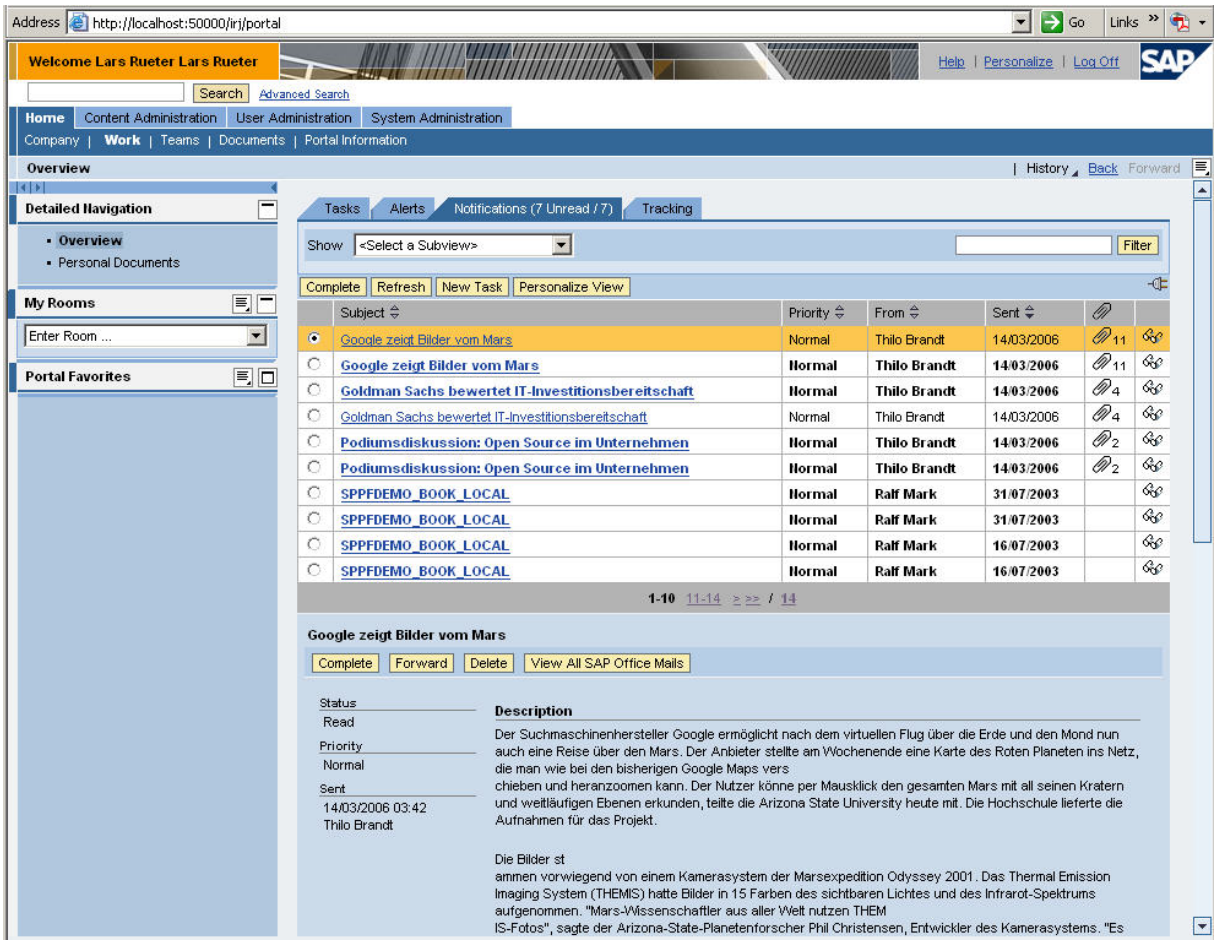

Figure 2: SAP Office Mails displayed in the UWL

The main tasks of a UWL connector are to pull items form a backend system, map these items to the internal format of an UWL item and return the items to the Universal Worklist service. The SAP Office Mail connector also demonstrates how to implement action handlers and an attachment provider. The sequence diagram below shows how the SAP Office Connector calls SAP Office Mail API's to retrieve the inbox for a particular user.

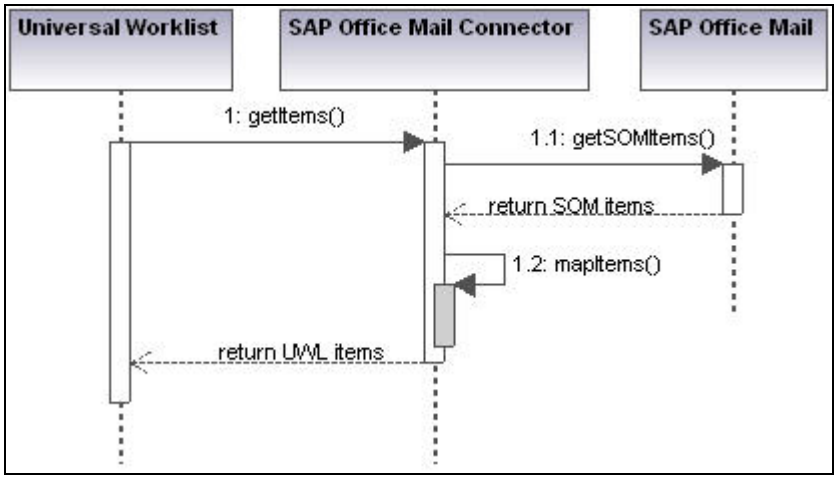

Figure 3: Retrieving SAP Office Mail inbox

The important classes for the implementation of the SAP Office Connector are shown in the class diagram below.

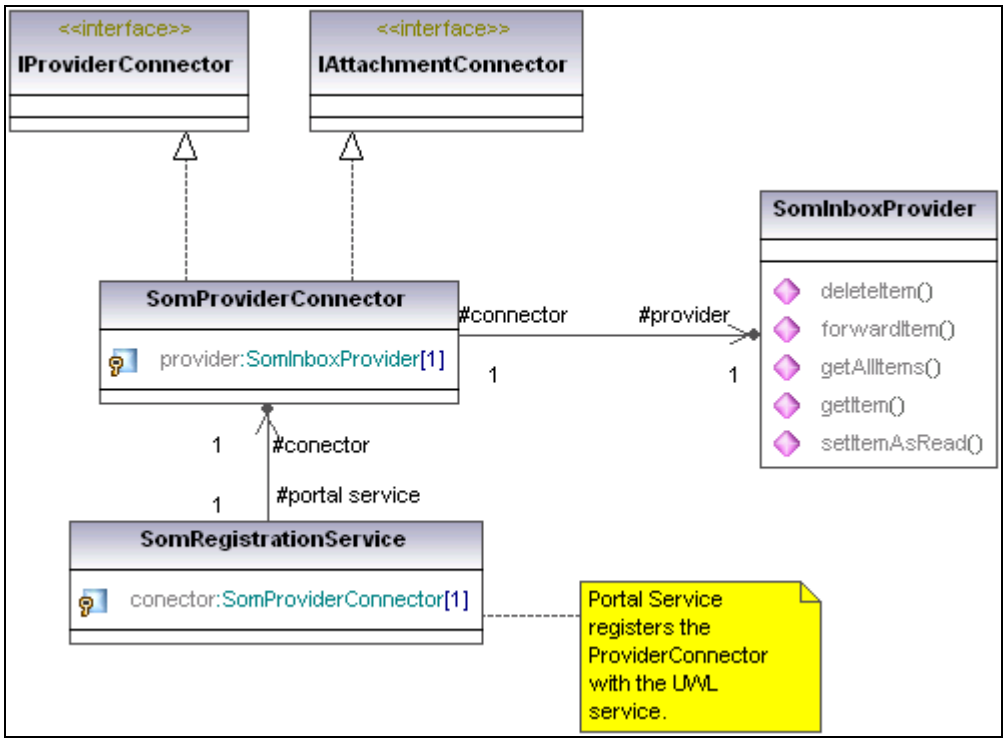

Figure 4: Class Diagram for UWL Connector

- *SomProviderConnector* – Implants the UWL Connector API interfaces IProviderConnector and IAttchmentConnector and dispatches the actual backend communication to the SomInboxProvider

- *SomInboxProvider* – This class (Java Bean) does the main work and communicates with Web Services for getting items from the backend and returning them to the connector. It also takes care of the necessary value conversions.

- *SomRegistrationService* – Registers the SomInboxProvider with the UWL service.

3 Disclaimer

The primary objective of this How-To paper is to demonstrate the capabilities and the handling of the UWL Connector API. The source code for the SAP Office Connector should be treated as an example only. In particular it was not designed to be used in production and SAP does not provide any support for it.

4 Prerequisites

- SAP NetWeaver 2004 SPS14 or higher
- SAP NetWeaver Developer Studio 2.0.14 or higher

- Only basis system > 4.7 supported due to the use of Web Services

5 Coding in details

The solution is based on the latest SAP development concepts using Development Components (DC). For simplicity reasons local DCs were chosen.

The project itself is structured in multiple DCs to separate the backend access from the connector implementation. This implementation fro calling the SAP backend is based on Web Services and is not meant to be the most performant solution. There might be better result for calling the SAP Office Mail RFCs on performance issues using JCo or JCA.

5.1 External libraries

All libraries required in this connector implementation which are not referenced by a standard SAP DC are put into an external library DC project. The External Library DC will contain all Java libraries which are not yet defined in SAPs standard DC delivery. This mainly contains the UWL and portal libraries, but also any 3rd party library which should be used within the connector implementation.

5.2 Provider Connector and Attachment Connector for UWL

For providing item population and attachment support at least two interfaces have to be implemented. The IProviderConnector takes care about single and multiple items retrieving from the provider (Java Bean). The IAttachmentConnector offers methods for attachment support. Both interfaces are implemented in the same class for this SAP Office Mail connector.

The following sequence diagram shows the call stack of the implemented IProviderConnector (SomProviderConnector) while fetching all items of a user's SAP Office Mail inbox:

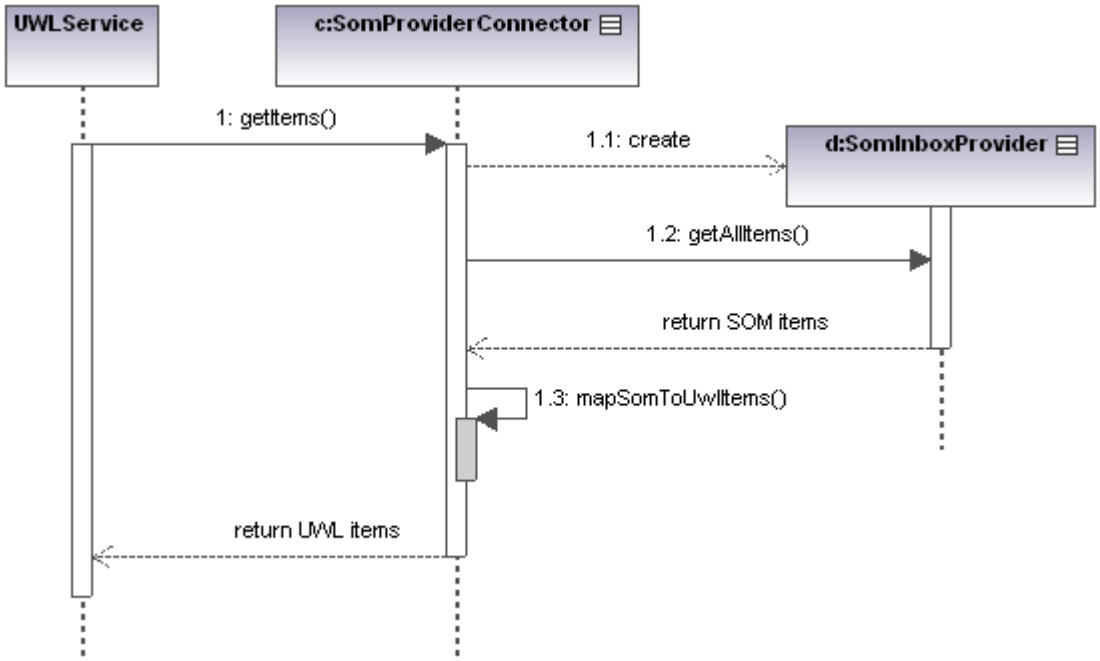

Figure 5: Call stack for retrieving all items

The following sequence diagram shows the call stack of the implemented IAttachmentConnector (SomProviderConnector) while fetching an attachment from a SAP Office Mail:

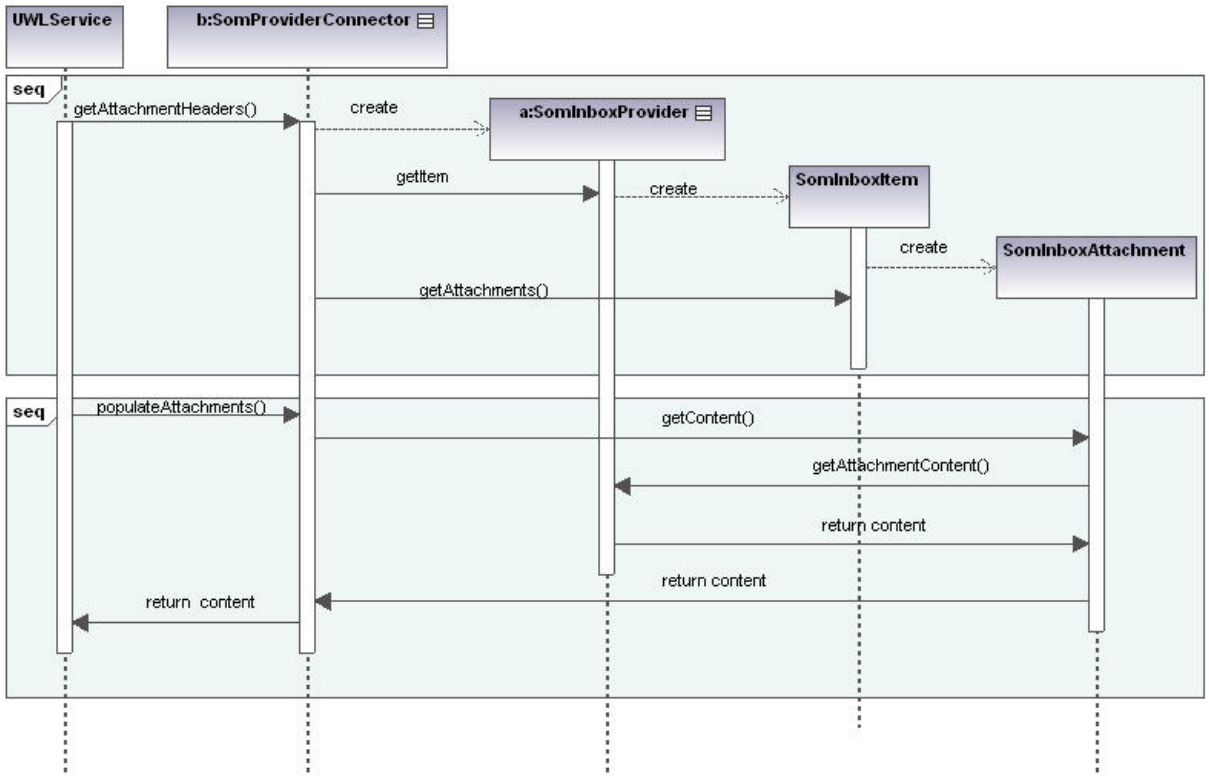

Figure 6: Call stack for fetching an attachment

5.2.1 Setup the IDE

The Provider Connector and Attachment Connector implementations are deployed in a Portal Application Module DC. This Portal Application Module DC will contain the UWL connector implementation classes and the XML configuration file for the UWL.

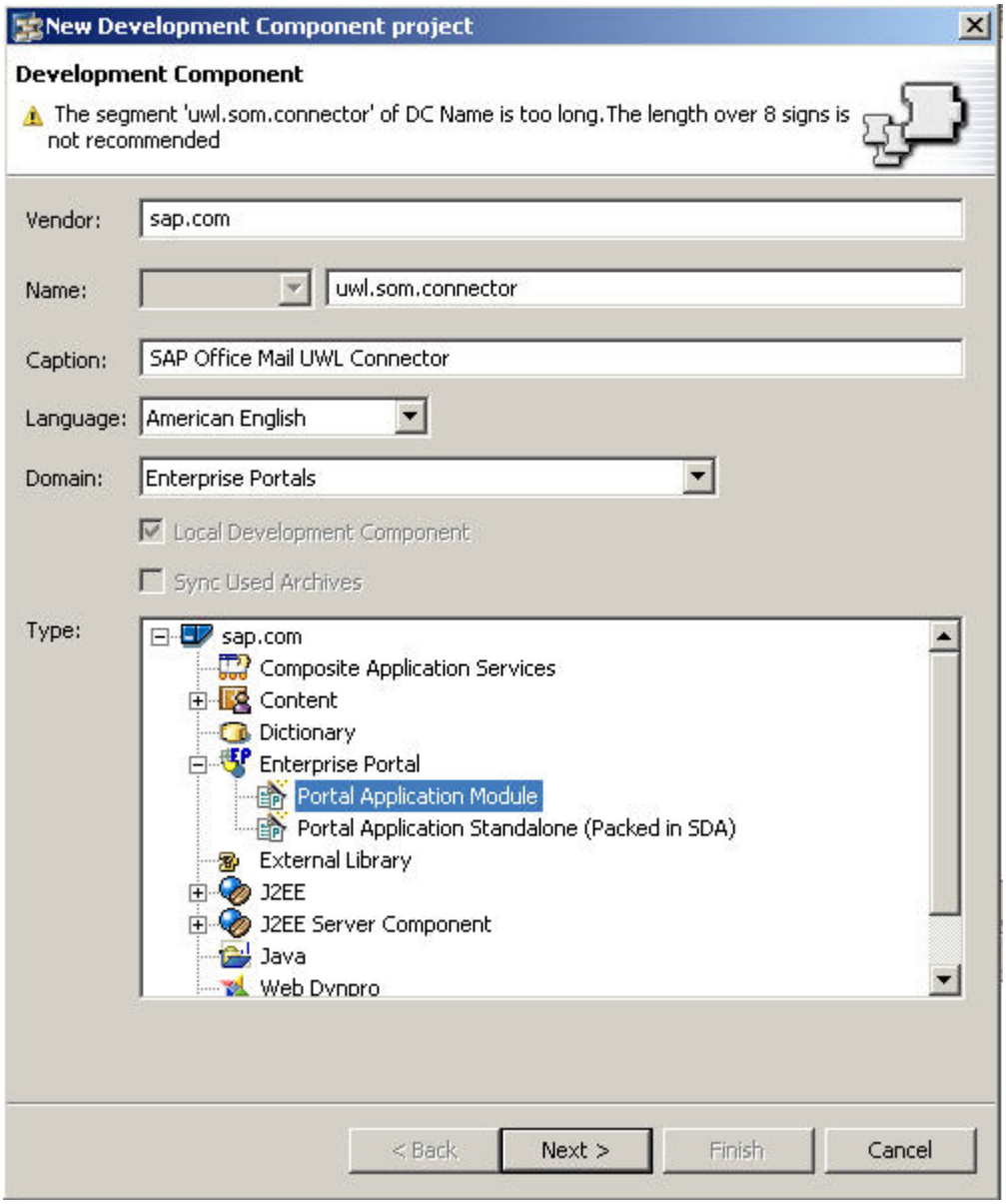

Figure 7: Create a Portal Application Module DC

A portal service implementation is used as registration service to register all UWL components at the UWL service. Make sure that all deployment descriptor references are maintained in the portalapp.xml file. At least a reference to SAPJ2EE::sap.com/uwl.som.provider.application, usermanagement and com.sap.netweaver.bc.uwl must be set. SAPJ2EE::sap.com/uwl.som.provider.application is the reference to the Java Bean implementation.

```
<application-config>
    <property name="SharingReference"
value="SAPJ2EE::sap.com/uwl.som.provider.application"/>
    <property name="ServicesReference"
value="usermanagement,com.sap.netweaver.bc.uwl" />
```

```
    <property name="releasable" value="true"/>
</application-config>
```

During design time and runtime depend DCs must be referenced in the Portal Application Module DC.

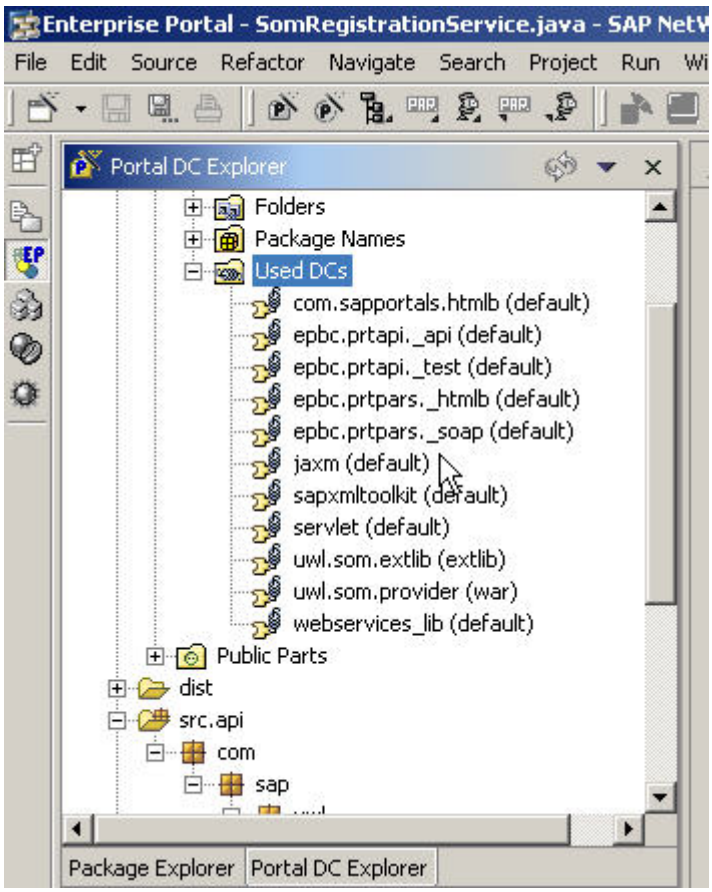

Figure 8: Required DCs for Provider Connector and AttachmentConnector

5.2.2 Implementation

The main method of the implementation is to retrieve all items from the provider. Therefore the getItems() method of the IProviderConnector interface has to be implemented.

```
public ConnectorResult getItems(UWLContext context,
            String itemType, ConnectorFilter connectorFilter,
            String system)
      throws ConnectorException {
```

This method must return a valid list of items from the provider implementation. The data must be mapped from the provider's internal item representation into the UWL item representation.

```
        ConnectorResult result = null;
        List items = null;
```

```
        try {
              SomInboxProvider somProvider =
                    new SomInboxProvider();
              List somItems = somProvider.getAllItems(
                                context.getUser(), system);
              items = mapSomToUwlItems(
                    context, itemType,
                    connectorFilter, system, somItems);
        } catch (SomInboxProviderException e) {
              throw new ConnectorException(e);
        }
   }
```

A valid UWL ItemCollection is created which is returned as a snapshot result by the methods.

```
ProviderStatus  status = new ProviderStatus(true, system,
              SomProviderConnector.SOM_CONNECTOR_ID);

return ConnectorResult.createSnapshotResult(new
                              ItemCollection(items), status);
```

It is also possible to return delta results, which then only should contain modified items from the backend.

To avoid to much backend calls at the same time, the content of a single item is not meant to be pre-fetched by the getItems() method call. Therefore the getDescription() method for a single item is called.

```
public String getDescription(Item item, UWLContext ctx)
        throws ConnectorException {
```

The content always has to be provided as a String representation to the UWL. Other formats have to be converted.

```
  SomInboxProvider somProvider = new SomInboxProvider();
  try {
        SomInboxItem entry =
              somProvider.getItem(ctx.getUser(),
              item.getSystemId(), item.getExternalId());
        return new String(entry.getContent());
```



```
  } catch (SomInboxProviderException e) {
  } catch (UWLException e) {
  }
        return "";
  }
```

Attachments are retrieved in 2 steps. In the first step only the attachment header information is fetched from the provider.

```
public Attachment[] getAttachmentHeaders(UWLContext ctx, Item item)
        throws ConnectorException {
```

The header information should be filled with the required meta information.

```
Attachment(java.lang.String attachmentConnectorID,
  int type,
  java.lang.String title,
  java.lang.String description,
  java.lang.String internalID,
  java.lang.String author,
  java.util.Date timeCreated,
  PriorityEnum priority,
  java.lang.String fileName,
  java.lang.String extension,
  java.lang.String mimeType,
  int fileSize) ;
```

Keep in mind to call the Attachment objects constructor which fits best the header data mapping of the provider's data type.

In the second step the content of the Attachment is retrieved through the IAttachmentConnector's method populateAttachments().

```
public void populateAttachments(UWLContext ctx, Item item,
              Attachment[] attachments) throws ConnectorException {
```

To make the UWL service call the IAttachmentConnector's method, the items should be marked as hollow items. This could be done in the getItems() call of the IProviderConnector.

```
uwlItem.setHollow(true);
```


The UWL service is later on triggered to call the populateHollowItem() method of the IProviderConnector interface.

Optionally the items currently populated can be pushed in to the UWL's pushback channel.

```
uwlService.getPushChannel().updateItem(this, ctx, item);
```

In this connector both interfaces IProviderConnector and IAttachmentConnector are implemented in the same class.

To handle UWL actions it is required to implement also an action handler class, which takes care about the action notifications.

The performAction() method is called by the UWL service if certain actions are fired. To react on these action events an IActionHandler implementation has to be provided.

```
public ProviderStatus performAction(UWLContext context, Item item,
                          Action action, java.util.Map properties)
             throws ConnectorException {
  String actionName = action.getName();
  if(Action.DELETE.equals(actionName)) {
  }
  else if(Action.MARK_AS_READ.equals(actionName)) {
  }
  else if(Action.ACTION_FORWARD.equals(actionName)) {
  }
```

If a certain action event was detected the corresponding action in the provider could be triggered.

5.2.2.1 Implementing a registration service (Portal Serivce)

To register the IProviderConnector and IAttachmentConnector implementation at the UWL service a portal service, called registration service, is implemented which calls the UWL service register methods during the Portal Runtime starts up.

The registration service is based on the IService interface of the portal runtime. Its task is to register the IProviderConnector and IAttachmentConnector implementation at the UWL service in its afterInit() method. Keep in mind that the portal service key is unique.

```
  public void afterInit()
  {
```



```
IUWLService uwlService =
(IUWLService)PortalRuntime.getRuntimeResources().
                  getService(IUWLService.ALIAS_KEY);
```

In this connectors case only one instance has to be created, since this class implements the IProviderConnector and IAttachmentConnector interface.

```
connector = new SomProviderConnector(uwlService);

//register with UWL service
uwlService.registerProviderConnector(connector);
uwlService.registerAttachmentConnector(connector.getId(),
                                                    connector);
}
```

The service must implement the afterInit() method, which is called by the portal runtime after the portal application is started.

5.2.3 Configuration

To create a new instance of a connector, the connector has to be configured in the UWL configuration. A new system entry has to be defined under Universal Worklist Systems. The system alias defined in this configuration is taken for identifying the System Landscape system.

5.2.3.1 Modifying the standard UWL configArchive (only SAP NetWeaver 2004)

In NW04 it is necessary to modify the standard UWL configArchive and redeploy it with your DC deployment. This modification takes care about the registration of the configuration files for the connector. In SAP NetWeaver 2004s and later releases this can be done by the standard UWL configuration UI.

The connector name will not show up in the drop down box in the UWL Systems Configuration UI until the following steps are done:

1. Locate the following file under your portal installation directory: com.sap.netweaver.bc.uwl.configarchive (it is a zip file). It is located at <WAS Install directory>\<SID>\JC<InstanceNumber>\j2ee\cluster\server0\apps\sap.com\irj\servlet_jsp\irj\root\WEB-INF\portal\portalapps\com.sap.netweaver.bc.uwl\config.
2. Copy this config archive to a temp directory.
3. Extract the install\meta\lib\com.sap.netweaver.bc.uwl.configmeta file (it is a zip file) from the configarchive file, and then extract the com.sap.netweaver.bc.uwl\systems\UWLSystems.cc.xml from the configmeta file.
4. Edit the UWLSystems.cc.xml file and add your connector name in the list of connectors.

5. Now update the configmeta file with the modified UWLSystems.cc.xml file and then update the configarchive file with this modified configmeta file. Make sure to maintain the relative paths of the files in the zip archives.
6. Copy this modified configarchive to your portal project location under the PORTAL-INF\config directory. This ensures that when you build your portal project, the resulting par file contains the configarchive. (When you create a portal project in the Netweaver Developer Studio, the PORTAL-INF directory is created under the “dist” directory.

Keep in mind to zip and extract the data with a JAR compliant zip tool.

5.2.3.2 Defining item types, views and actions

To show the required information in the UWL you can provide an XML based descriptor which influences the view of the UWL iView. In this connector implementation the SAP Office Mail items should occur as Notifications with an individual sub view.

The XML file must be located in the Connectors DC under the path ./dist/PORTAL_INF/classes and must be deployed with the DC.

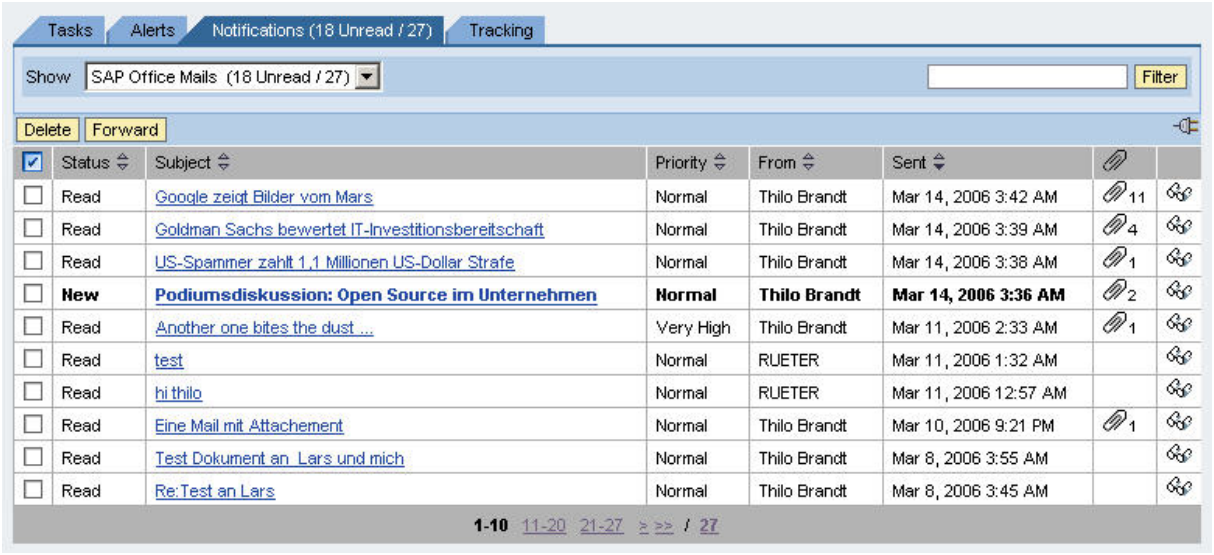

Figure 9: SAP Office Mail sub view

To define a sub view for SAP Office Mails the uwl_<connector>config.xml file has to be modified. Define the item types provides by the connector. This connector’s item type is uwl.notification.som

```
<ItemTypes>
  <ItemType name="uwl.notification.som"
     connector="SomProviderConnector"
     defaultView="SomView"
     defaultAction="showDetails">
  <Actions>
    <Action reference="delete"/>
    <Action reference="forward"/>
```

```
  </Actions>
   </ItemType>
</ItemTypes>
```

Define the sub view and assigned actions of the SAP Office Mails. Refer to the online help to get information which fields are supported.

```
<Views>
<View name="SomView" width="98%"
  supportedItemTypes="uwl.notification.som" columnOrder="status,
  subject, priority, creatorId, createdDate, attachmentCount,
  detailIcon" sortby="createdDate:descend, priority:descend"
emphasizedItems="unread" selectionMode="MULTISELECT"
tableDesign="STANDARD" visibleRowCount="10" headerVisible="yes"
tableNavigationFooterVisible="yes" tableNavigationType="CUSTOMNAV"
actionRef="">

<Descriptions default="SAP Office Mails"/>
<Actions>
  <Action reference="delete"/>
  <Action reference="forward"/>
</Actions>
</View>
</Views>
```

5.2.3.3 Creating a new instance of the connector

The connector has to be registered in the UWL system configuration. A new UWL system for the connector has to be created (System Administration -> System Configuration -> Universal Work -> Universal Work System)

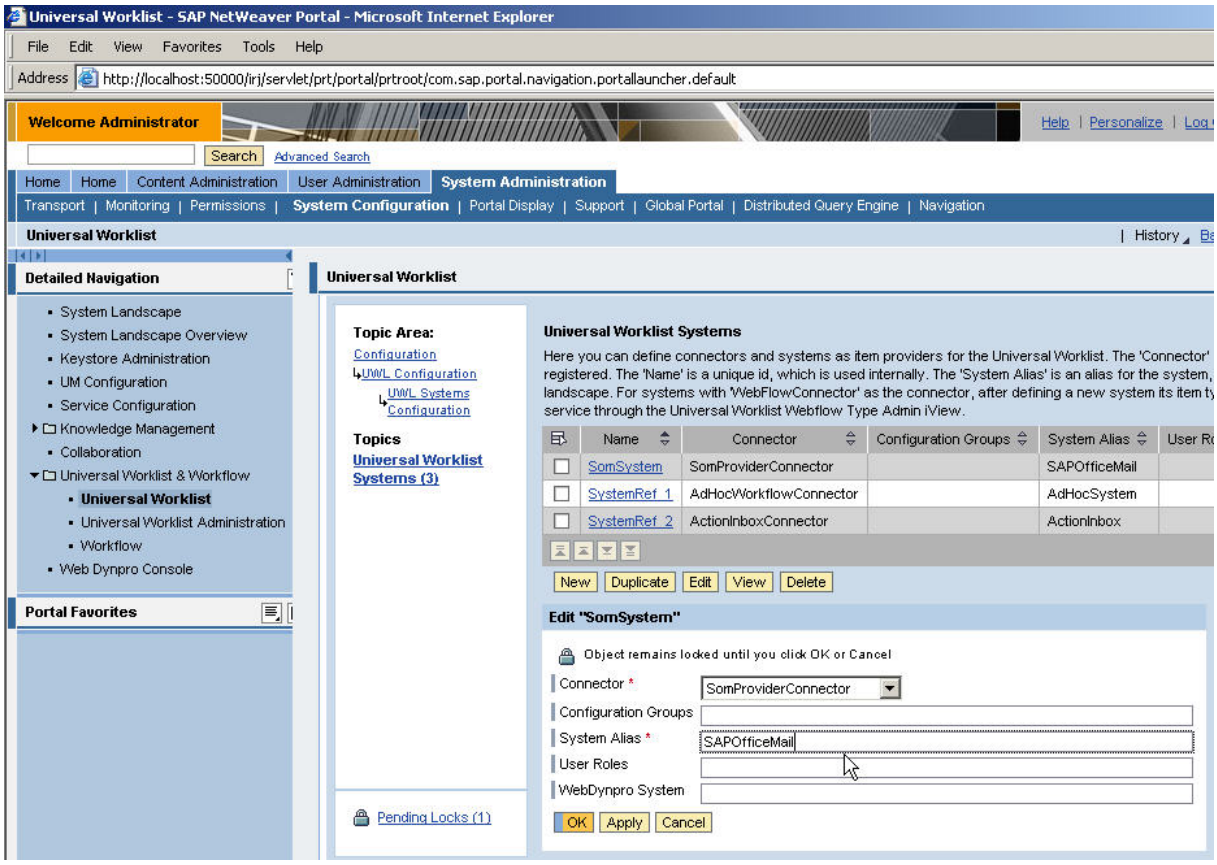

Figure 10: Create a new UWL system for the Connector implementation

Select the Connector for SAP Office Mail and specify a System Alias which is used for mapping the Portal System landscape information and the user mapping data. Keep in mind that this alias is later used for the user mapping determination. It must match in letter case-sensitive to the System landscape alias.

5.3 Provider Bean Implementation

- The Web Services are wrapping the required SAP Office Mail RFC calls to the SAP system. The Java Bean SomInboxProvider offers convenient access to the Web Service layer for the connector.

5.3.1 Setup the IDE

The Java Bean classes have to be build and deployed within in an Enterprise Application DC. The Enterprise Application DC is the deployment unit for the Web Module DC. It simply will contain the generated classes of the Web Module.

The Web Module DC contains the Java class of the Bean (SomInboxProvider). All Web Service exposed Public Parts have to be added as Used DC in the Web Module DC.

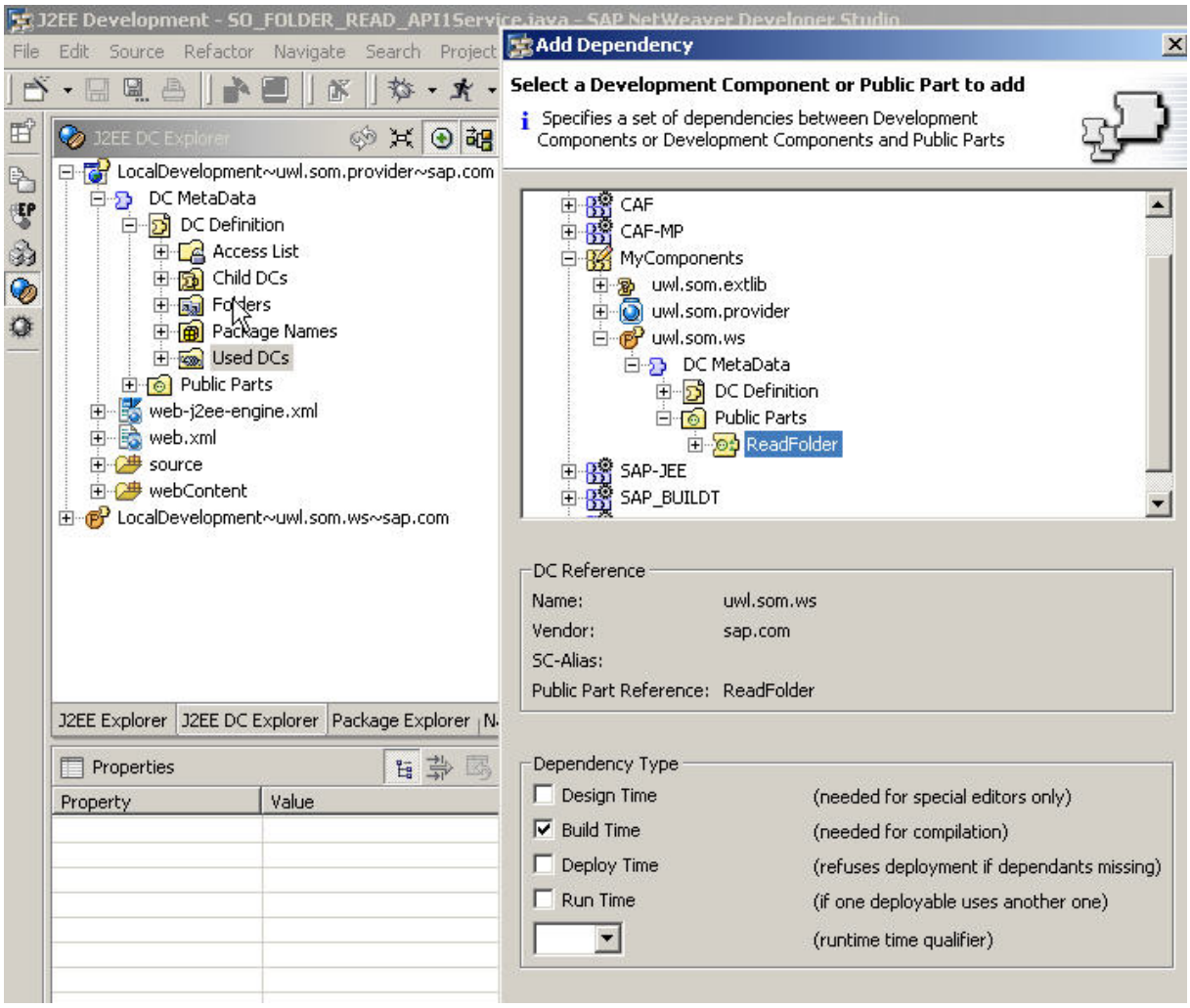

Figure 11: Add Web Services Public Part to Used DCs

Add all required DCs to the Used DC part of the Web Module DC. Keep in mind also to add the standard DCs jaxrpc, servlet and webservices_lib.

The Enterprise Application DC is the deployment unit for the Web Module DC. It simply will contain the generated classes of the Web Module. Add the already existent Web Module DC to the Enterprise Application DC.

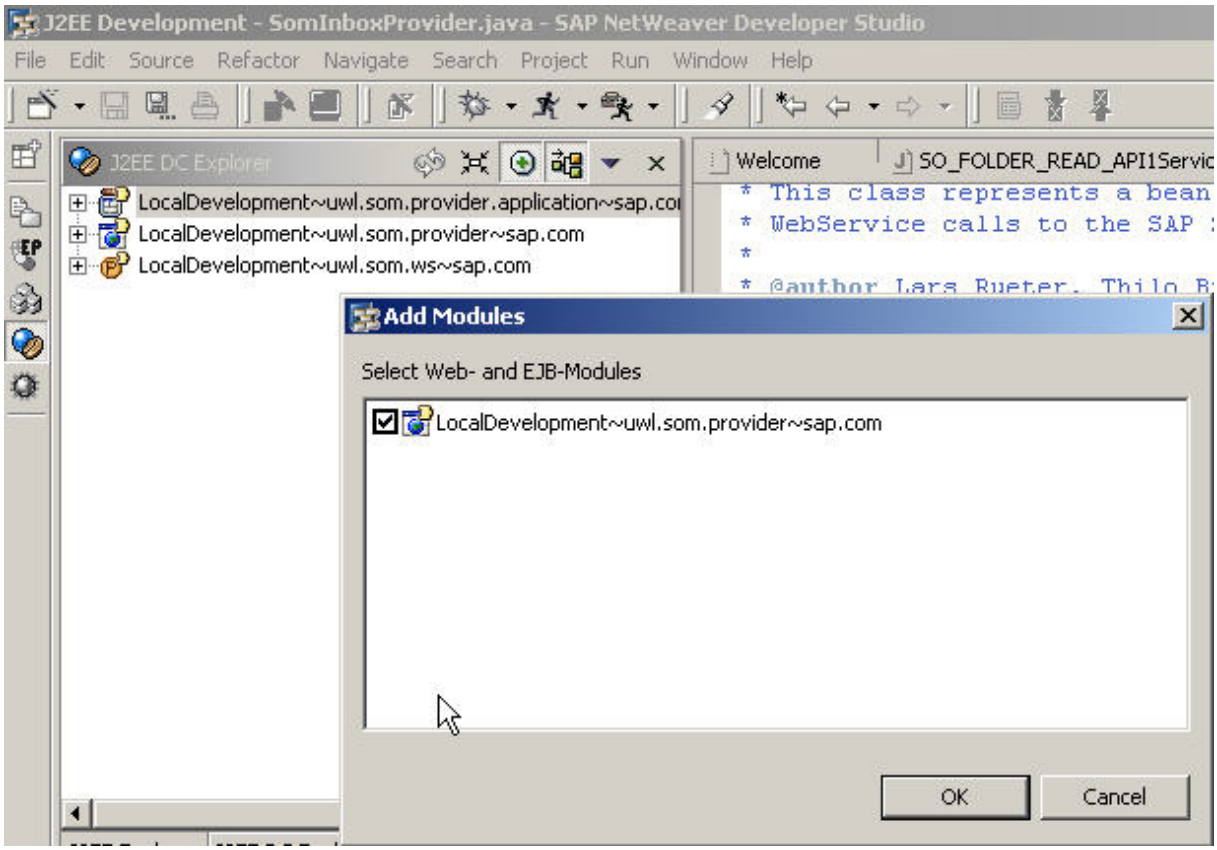

Figure 12: Add Web Module DC to Enterprise Application DC

Additional web descriptor attributes can be set optionally after the Web Module DC assignment.

5.3.2 Implementation

The Java Bean (Provider Bean) is offering access to convenient methods which wraps the Web Service call. The methods are chosen in order to be equivalent with IProviderConnector and IAttachmentConnector interfaces.

The following methods are introduced:

- getAllItems(): Calls all items of the SAP Office Mail inbox for a specific user
- getItem(): Calls a single item
- deleteItem(): Deletes a single item in SAP Office Mail
- forwardItem(): Forwards a single item to a set of receivers
- setItemAsRead(): Sets the READ flag of an item in SAP Office Mail
- getAttachmentContent(): Gets the content of a single attachment

Some of these method call may result in multiple Web Service calls.

5.3.2.1 General issues for Web Service calls

There are some general issues you have to keep in mind when calling a certain Web Service from within the Java Bean:

Looking up the required Web Service in the Name Service (JNDI lookup) . The lookup string is build in general

```
wsclients/proxies/sap.com/<WebServiceDC>/<completePathToWebservice>
```

Determine the user mapping data to call the backend with the correct backend credentials.

```
IUserMappingData umd = this.getMapping(user, system);

port._setProperty(
  SO_OLD_DOCUMENT_SEND_API1PortType.USERNAME_PROPERTY,
  this.getMappedUser(umd));

port._setProperty(
  SO_OLD_DOCUMENT_SEND_API1PortType.PASSWORD_PROPERTY,
  this.getMappedPassword(umd));
```

Create a new requesting object e.g. a document and set all required parameters. All structures/tables have to be at least initialized with values not equal null.

```
SO_DOCUMENT_READ_API1 document =
                    new SO_DOCUMENT_READ_API1();
document.setDOCUMENT_ID(itemId);

SOATTLSTI1[] attachments = new SOATTLSTI1[0];
document.setATTACHMENT_LIST(attachments);

SOLIX[] contents = new SOLIX[0];
document.setCONTENTS_HEX(contents);
:
```

Date conversion is required for all SAP dates provided as String objects.

```
Date sendDate = new Date(System.currentTimeMillis());
DateFormat df =
        new SimpleDateFormat("yyyy-MM-dd hh:mm:ss");
sendDate = df.parse(dt);
```

5.3.2.2 Mapping metadata (item attributes)

The SomInboxItem keeps track of the mapping from SAP Office Mail attributes to UWL item attributes. Several information units are mapped. Some of the SAP Office Mail attributes could not be mapped. They will not be available through the UWL access.

The provider is mapping the backend data to a proxy object (SomInboxItem). The SomInboxItem class keeps track of all UWL related information which can be mapped as attributes.

```
private void fillMetadata(SomInboxItem item, Object somItem) {
  item.setDOCID(((com.sap.uwl.som.folder.types.SOFOLENTI1)
              somItem).getDOC_ID());
  item.setDocSize(((com.sap.uwl.som.folder.types.SOFOLENTI1)
              somItem).getDOC_SIZE());
:
```

5.3.2.3 Fetching content (item data)

The item's data (SomInboxItem) is stored in a structure parameter called SOLISTI1, which can be retrieved by calling the getContent () method from the SomInboxItem. The data has to be transformed into a Java byte array (byte[]) to be convenient with the UWL API. Due to the fact that there are binary types of SAP Office Mails, these types are ignored currently. This implementation only takes care on SAP Office Mails of type RAW and TXT.

```
// this is just a work-a-round for ASCII content
boolean isLinebreak = item.getObjType().equalsIgnoreCase("TXT");
StringBuffer b = new StringBuffer((int)item.getDocSize());
for (int i=0;i<contents.length;i++) {
  b.append(contents[i].getLINE());

  if (isLinebreak)                    b.append(
  System.getProperty("line.separator"));
}
item.setContent(b.toString().getBytes());
```

The provider is transforming the SAP SOLISTI1 structure into a byte[] object.

5.3.2.4 Fetching attachment data

The attachment (SomInboxAttachment) information is gathered in 2 steps. The first step only collects the attachment metadata from the SO_DOCUMENT_READ_API1 Web Service. In the second step SO_ATTACHMENT_READ_API1 is called in a separate method call. This way was chosen to avoid too many calls to the backend and to be convenient with the UWL API. The content of the attachment is called via a Callback mechanism in the getAttachementContent() method implementation. The

content of an attachment is currently always called from the backend system. There is no caching used so far.

```
SomInboxAttachment[] somAttachments = new
SomInboxAttachment[attachments.length];
for (int i=0;i<attachments.length;i++) {
  somAttachments[i] = new SomInboxAttachment();
  somAttachments[i].setUser(
        currentItem.getUser());
  somAttachments[i].setSystem(
        currentItem.getSystem());
              somAttachments[i].setAttachmentId(
        attachments[i].getATTACH_ID());
              somAttachments[i].setAttachmentTitle(
        attachments[i].getATT_DESCR());
              somAttachments[i].setAttachmentType(
        attachments[i].getATTACH_TYP());
              somAttachments[i].setAttachementSize(
        Integer.parseInt(
  attachments[i].getATT_SIZE().trim()));

}
currentItem.setAttachments(somAttachments);
```

The provider is storing the backends attachment data in the proxy object of the Java bean.

5.4 AttachmentProviderService

This section describes how to implement a Portal Service to retrieve SAP Office Mail attachments. The service connects to the backend using the J2EE Java Resource Adapter. It will also be exposed as a portal Web Service to be called by the provider.

First you need to create a new DC and add the required library references. In the next step you create the Portal Service and finally expose it as a web service.

5.4.1 Setup the IDE

In the NWDS create a new DC of the type Portal Application Standalone (Packed in SDA) with the name uwl.som.provider.attachments. In this DC you can now create a Portal Service as shown in the figure below.

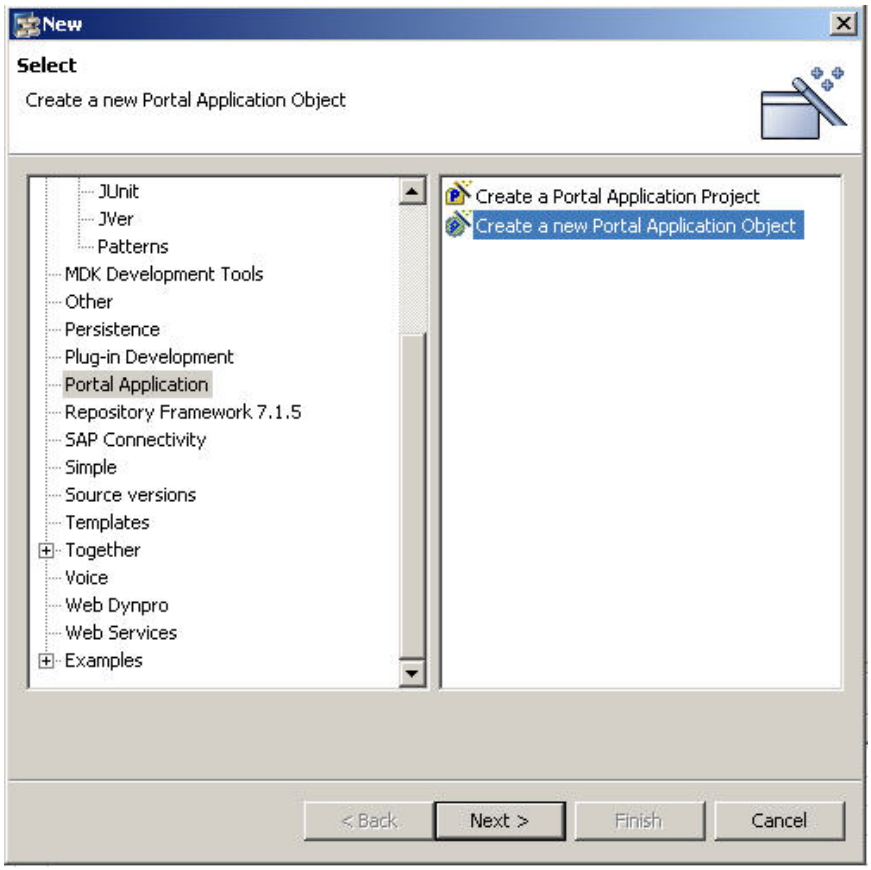

Figure 13: Creating a Portal Service

In addition to the portal service also create a Java Class called SimpleTransaction in the same package. Your package should look similar to the figure below.

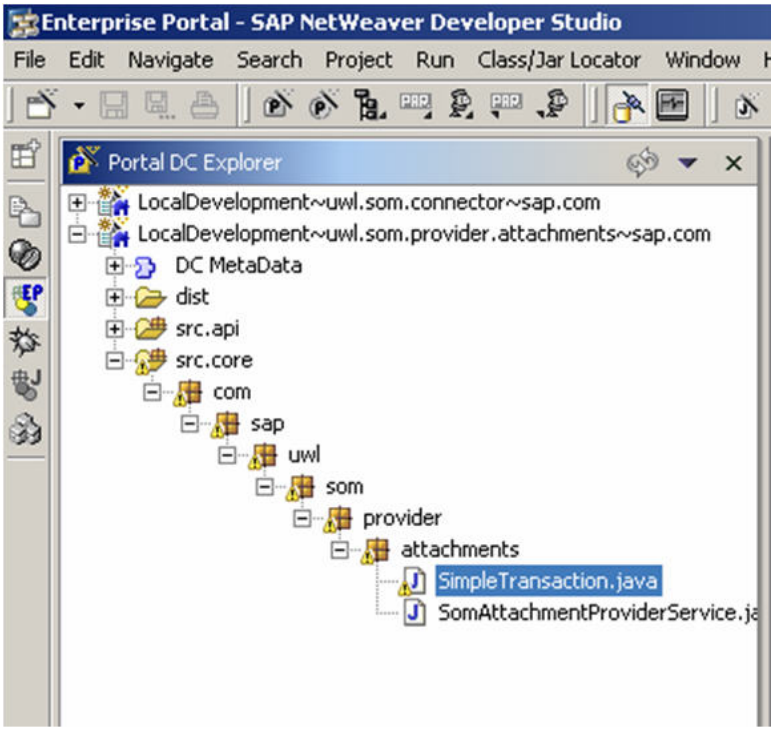

Figure 14: Package contents of the attachment portal service

To use the J2EE Java Resource Adapter from your portal service you have to reference a number of libraries. Locate the following libraries by doing a search under \usr\sap\<SID>\<INSTANCE ID>\j2ee

- com.sap.portal.ivs.connectorservice_api.jar
- connector.jar
- exception.jar
- portal_services_api_lib.jar

These libraries need to be copied to the libraries directory in your external library DC. The figure below shows how to add the libraries to the public part of the library DC. This needs to be done for each library that you have added.

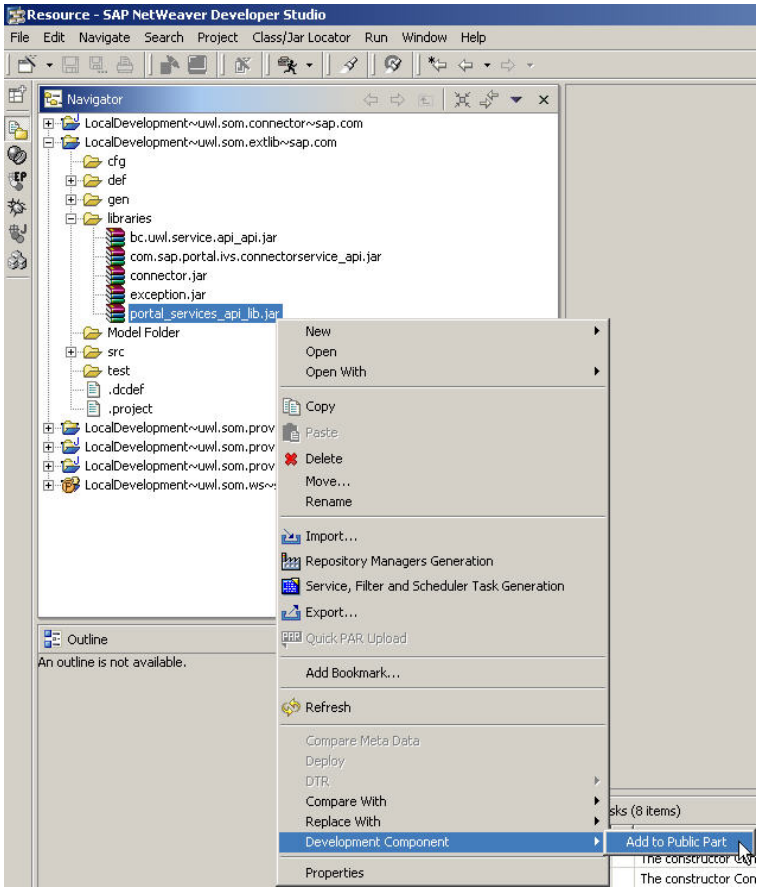

Figure 15: Add libraries to the DC public part

Now that you have added the libraries to the public part of the library DC you can add a reference to the library DC. You also have to add a reference to the SAP_JTECHS tc/conn/connectorframework.

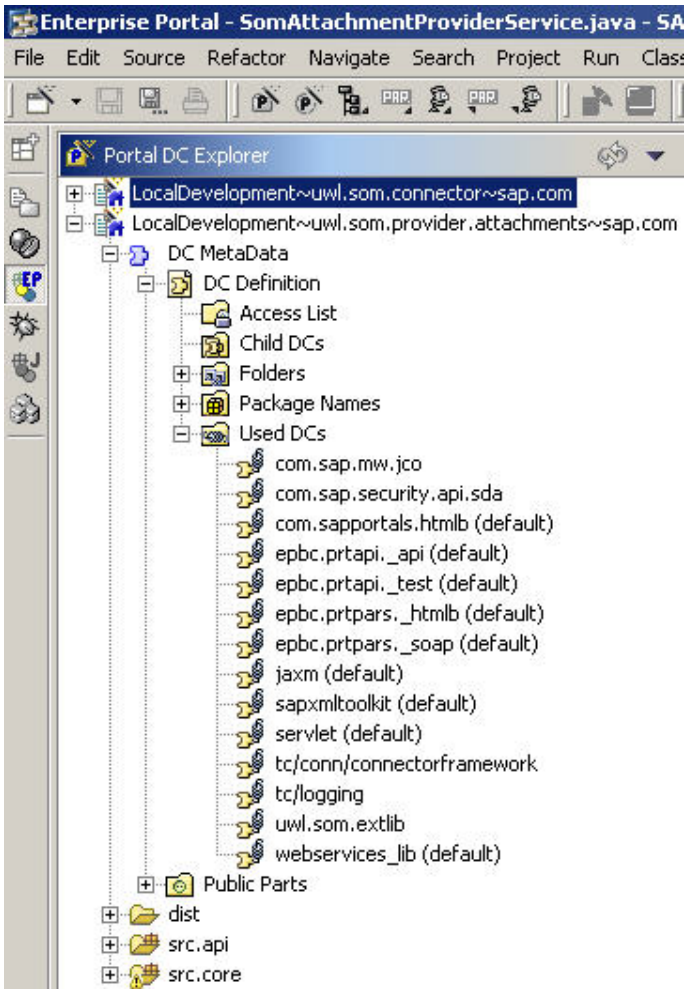

Figure 16: Define used DCs for the attachment DC

Make sure that you add the following ServicesReferences to your portalapp.xml.

```
<application-config>
  <property name="releasable" value="true"></property>
  <property name="ServicesReference"
              value="com.sap.portal.runtime.system.inqmy,
                     com.sap.portal.ivs.connectorservice,
                     com.sap.portal.ivs.systemlandscapeservice,
                     urlgenerator,
                     com.sap.portal.usermapping,
                     usermanagement,
                     com.sap.portal.runtime.application.rfcengine,
                     com.sap.portal.runtime.config">
  </property>
</application-config>
```

Now you can start implementing the attachment service. Details of the implementation are discussed in the next section. To make your service available as a web service only a few additional steps are required in NWDS.

You can create a web service form a portal service using a wizard in NWDS. Just go to New -> Portal Application -> Create Portal Application Object on the next screen you select your attachment DC. When you select next you should see the dialog box shown in the figure below.

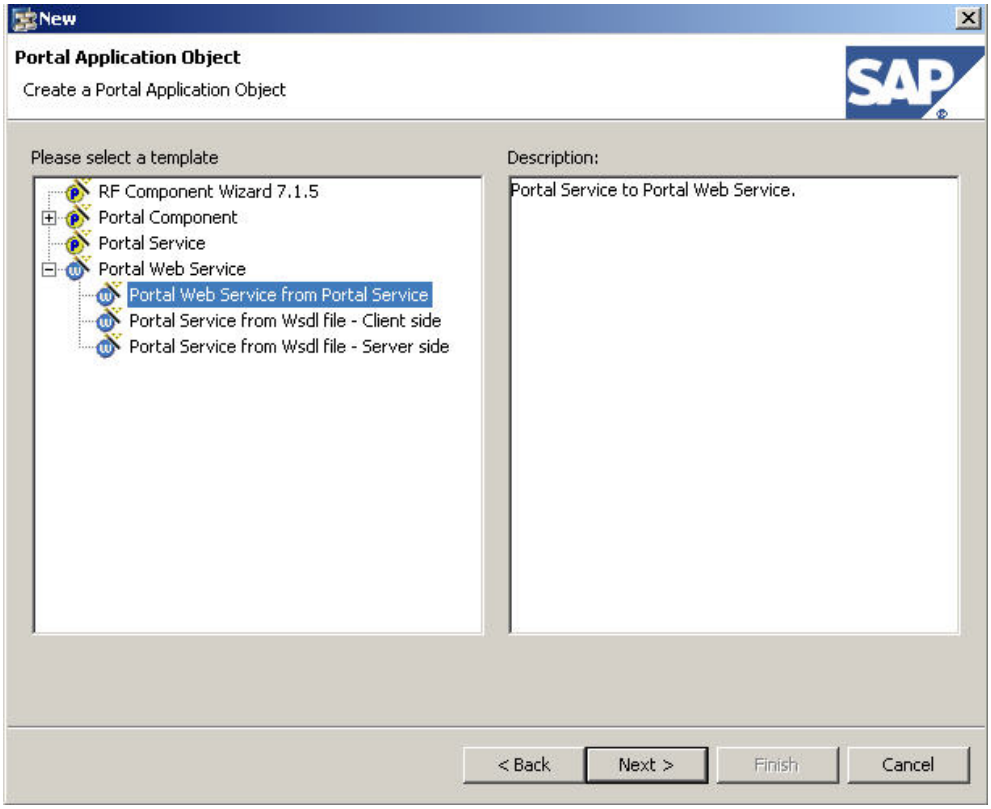

Figure 17: Create a Web Service from a Portal Service

Select Portal Web Service form Portal Service and continue to select your interface method *getAttachemt* and finish.

To test your Web Service you can use the Portal Web Services Checker that is part of the NWDS.

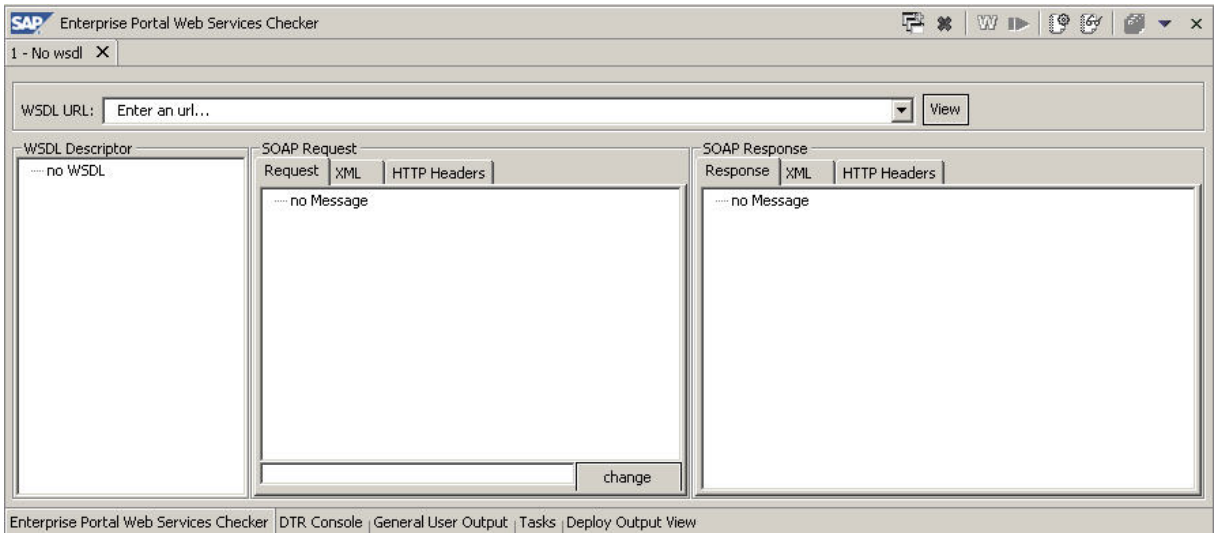

Figure 18: Portal Web Services Checker

5.4.2 Implementation

The implementation has three parts. First you define the interface of your Portal Service. You will need a getAttachment method as shown in the code below.

```
public interface ISomAttachmentProviderService extends IService
{
      public static final String KEY = "attachment_provider";

      public byte[] getAttachment( String user,
                                   String system,
                                   String attachmentId );

}
```

In the Portal Service itself you can leave the generated code untouched and just implement the getAttachement method. In five steps you can get the attachment content from the backend. To do this there is a helper class called SimpleTransaction that handles the details of the connection.

```
public byte[] getAttachment(String usr, String system,
                            String attachmentId) {
      [ ... ]
      IUser user = UMFactory.getUserFactory().getUser(usr);

      // 1. Prepare the RFC connection
      SimpleTransaction t = new SimpleTransaction(system,
            new ConnectionProperties(Locale.getDefault(), user));

      // 2. Set the attachment ID in the INPUT structure
      MappedRecord ipList = t.createInput();
      ipList.put(INPUT_ATTACHMENT_ID, attachmentId);

      // 3. Execute the RCF call
      MappedRecord opList =
                  t.executeFunction(FM_SO_ATTACHMENT_READ);

      // 4. Read the doc size and type from the OUTPUT structure
      IRecord header = (IRecord) opList.get(OUT_ATTACHMENT_DATA);
      int docSize =
      Integer.parseInt(header.getString(OUT_ATT_SIZE).trim());

      String docType = header.getString(OUT_ATTACH_TYP).trim();

      // 5. Get the content
```

```
        IRecordSet content = null;
        if (docType.equals(ATTACH_XML) ||
             docType.equals(ATTACH_TXT)) {
        content = (IRecordSet) opList.get(OUT_ATTACHMENT_CONTENT);
        } else {
              content = (IRecordSet) opList.get(OUT_CONTENTS_HEX);
        }
        [ ... ]
  }
```

As you can see in the code above it is not sufficient to only get the content, but you also need to get some information from the attachment header as well. From the document type you can usually tell in which output field the content will be transmitted. For text documents the content is stored in the field ATTACHMENT_CONTENT while for binary data it will be in CONTENTS_HEX.

To convert the attachment content into a byte array you can use the following source code that reads the output record set, writes it to an output stream which is then converted into a byte array.

```
ByteArrayOutputStream contentByte =
                              new ByteArrayOutputStream(docSize);
content.beforeFirst();
while (content.next()) {
      contentByte.write(content.getBytes(OUT_LINE));
}
contentByte.flush();

byte[] result = contentByte.toByteArray();
contentByte.close();
```

Below you find six basic steps to connect to the backend taken from the SimpleTransaction class. The class contains additional code that deals with error handling and is slightly more complex then the code below.

```
// 1. Get the connector portal service
IConnectorGatewayService cgService =(IConnectorGatewayService)
      PortalRuntime.getRuntimeResources().
                  getService(IConnectorService.KEY);

// 2. Get a connection from the connector service
IConnection connection =
      cgService.getConnection(system, connectionProperties);

// 3. Create interaction objects
```

```
IInteraction ix = connection.createInteractionEx();
IInteractionSpec ixspec = ix.getInteractionSpec();

// 4. Create INPUT structure
RecordFactory rf = ix.getRecordFactory();
MappedRecord input = rf.createMappedRecord("input");

// 5. Execute function and get OUTPUT structure
ixspec.setPropertyValue("Name", "SO_ATTACHMENT_READ_API1");
MappedRecord output = (MappedRecord) ix.execute(ixspec, input);

// 6. Close connection
ix.close();
connection.close();
```

In step 2 above the system and connectionProperties are passed as parameters. The connectionProperties contain the IUser object for the user that is logged in to the portal and wants to retrieve his SAP Office inbox. The IUser and the system are required to get the user mapping information for the backend system.

Once you have implemented the attachment service is might be helpful to test it by calling it from a PortalComponent. The portal component below calls the service directly, but does not return the attachment. It could be used for debugging the service. Alternatively you may wish to use the NWDS Portal Web Service tester to call the service.

```
public void doContent(IPortalComponentRequest request,
                              IPortalComponentResponse response)
{
      String system = request.getParameter("SYS");
      String attachmentId = request.getParameter("AID");

      ISomAttachmentProviderService myService =
            (ISomAttachmentProviderService) PortalRuntime.
                  getRuntimeResources().getService(service_name);

      StringBuffer s = new StringBuffer();
      s.append("<html><body>");
      s.append("Getting attachment ...");

      IUser user = (IUser)request.getUser();

      ConnectionProperties cp =
```

```
            new ConnectionProperties(request.getLocale(),user);

      myService.getAttachment(user.getUniqueID(),
                              system, attachmentId);

      s.append("... successful.");
      s.append("</body></html>");
      response.write(s.toString());

  }
  }
```

5.4.3 Configuration

You need to create a SAP system in the portal landscape and provide a user mapping to run this service. Both of these tasks are a prerequisite to run the SAP Office Connector and are described in this How-to guide in section 5.6.2.

5.5 Web Services for SAP Office Mail RFCs

The Web Services are encapsulating the SAP Office Mail RFCs from the SAP system. They are offering all functions which are used in this connector implementation. All Web Services will be bundled in a Web Services DC project. This Web Service DC will contain the generated Java Web Services stubs of the used SAP Office Mail RFC modules. The function group used in this example is SOI1 containing the following RFCs:

- SO_USER_READ_API1: Reading user's inbox information
- SO_FOLDER_READ_API1: Reading the user's folder data (items)
- SO_DOCUMENT_READ_API1: Reading an single item
- SO_DOCUMENT_DELETE_API1: Deleting a single item
- SO_OLD_DOCUMENT_SEND_API1: Forwarding an item
- SO_ATTACHMENT_READ_API1: Reading an attachment of an item
- SO_DOCUMENT_SET_STATUS_API1: Sets the documents status, e.g. to read

5.5.1 Setup the IDE

To make these Web Services available later in the connector implementation they have to be put into a deployable proxy project.

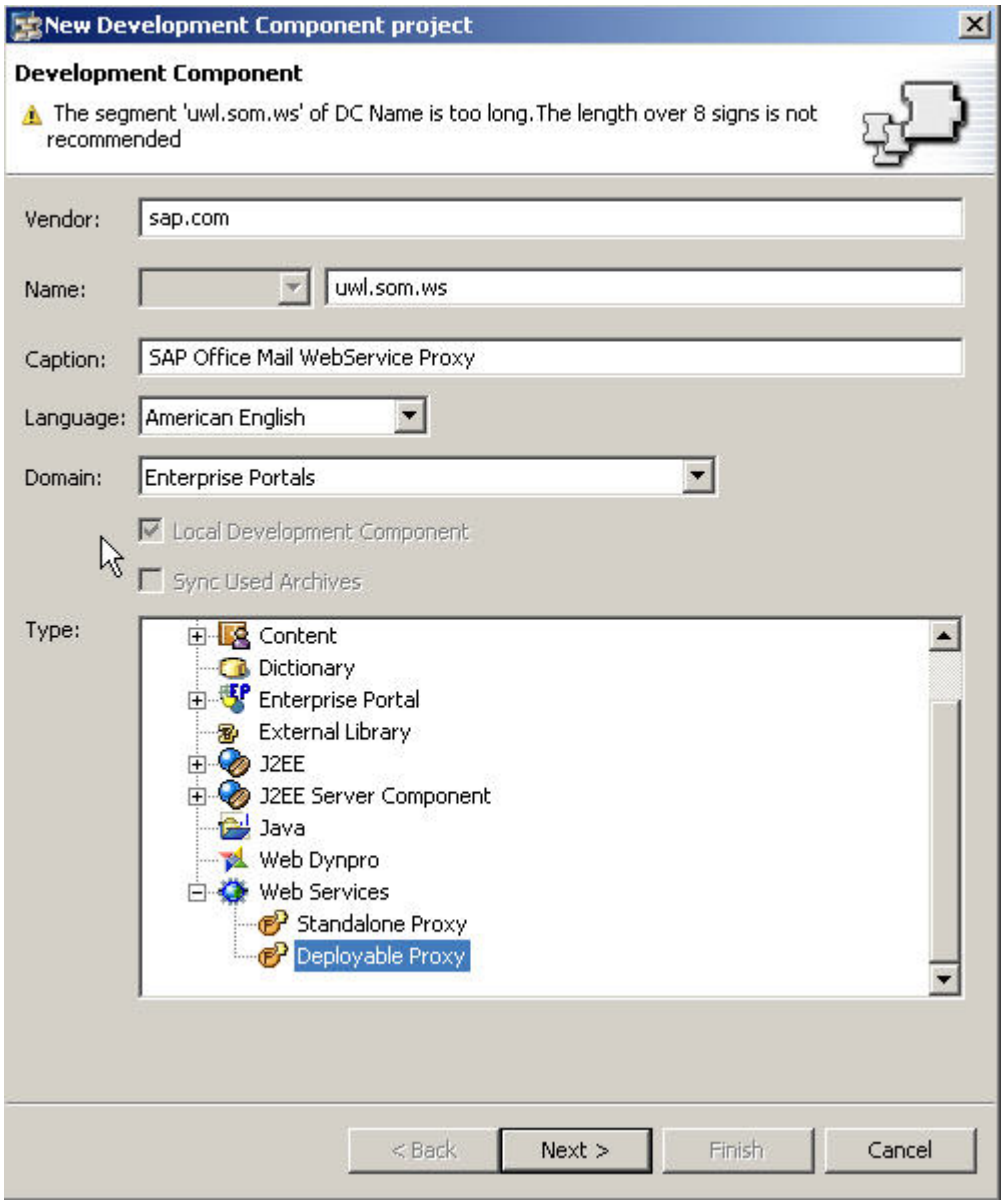

Figure 19: Create a deployable proxy project

Each SAP Office Mail RFC module has to be wrapped with a Web Service stub. To get the corresponding WSDL description for each RFC, simply call the Web Services Browser in the SAP System.

```
http://<hostname>:<port>/sap/bc/bsp/sap/WebServiceBrowser/search.html?sap-client=<relevant_client>
```

The WSDL files have to be imported by the Web Service Wizard for creating the proxy classes. A bunch of classes get created for each RFC module. These classes should not be changed by the developer.

After the Web Service stubs are generated in the Web Service DC, each Web Service has to be published as Public Part of the DC.

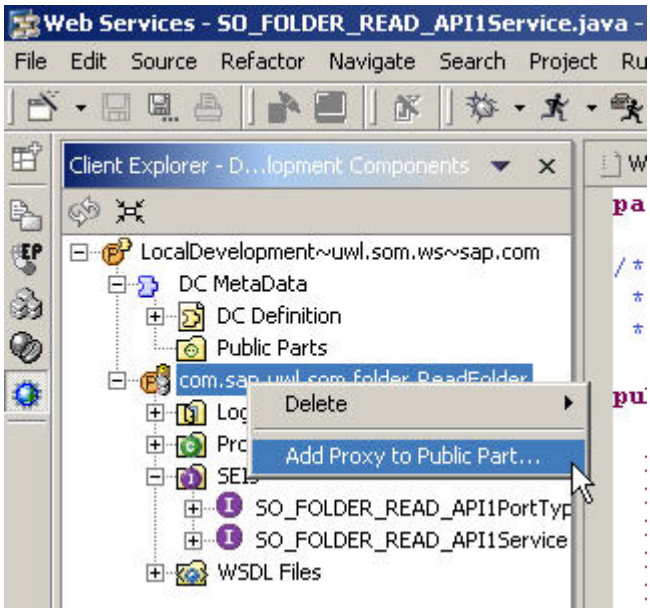

Figure 20: Add Web Service to Public Part of DC

Define a name for each Public Part. The Public Part name can later also be used for looking up the Web Service in the Java Naming Service (JNDI).

5.6 Configurating the UWL and Systemlandscape

5.6.1 Adding a system definition to System Landscape service

A new system has to be created in the Portal System landscape service. The system is used to resolve the user mapping for the Web Service calls in the Java Bean implementation. Create a new SAP system in the system landscape of the portal and set Logon Method to UIDPW (or any other method if supported by your backend system). A System alias for the SAP Office Mail has to be defined which is used for the user mapping. This alias must match the System alias of the UWL system.

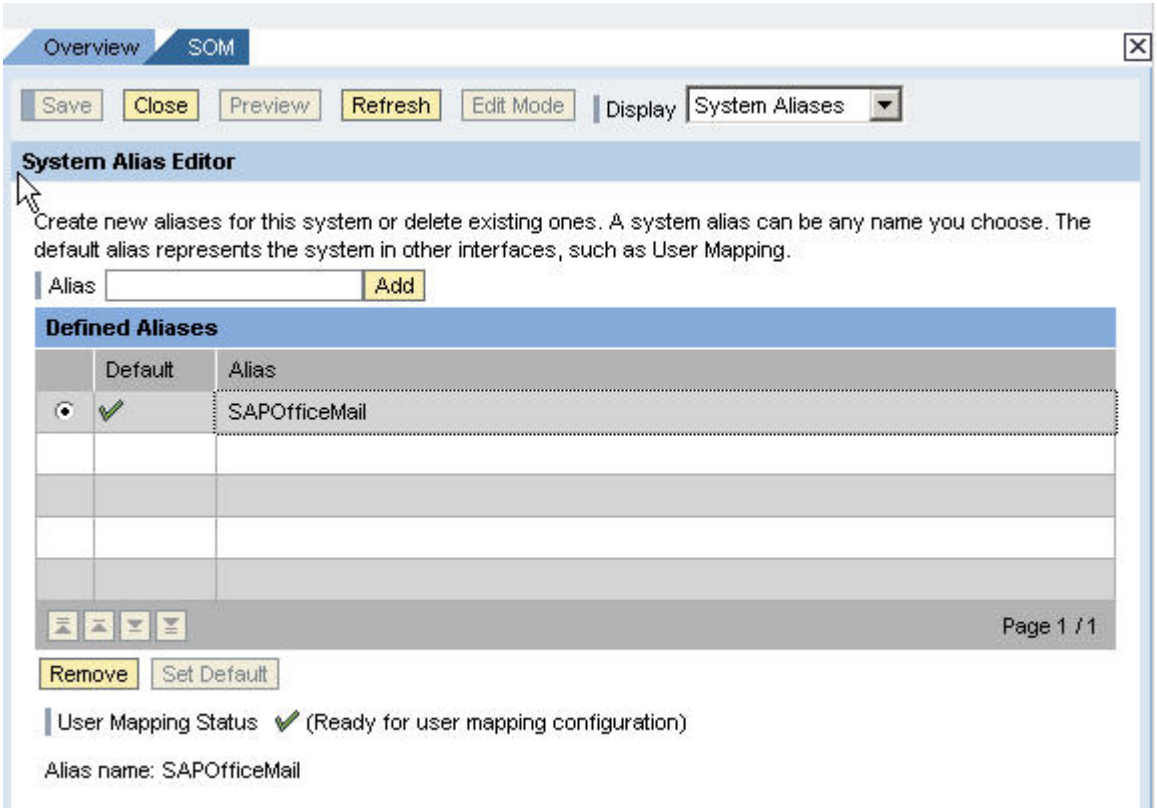

Figure 21: Create a system alias for SAP Office Mail

Keep in mind that only one system alias is defined for one connector instance in the UWL system configuration.

5.6.2 User mapping

A user mapping has to be setup by every user which wants to access the backend system. The mapped user and password are taken by the Java Bean implementation to connect to the backend system. If you have SSO enabled the user mapping is not required. Therefore chose the corresponding logon method in the system landscape system's definition.

6 FAQs

The FAQ section offers answers to open issues already known by the authors.

6.1 Why do I have to pre-fill structure elements/tables to call the Web Services stubs?

The structures and tabled passed to the SAP system are filled with the data of the backend system. If you don't pass them or even do not initialize them, you will get an NullPointerException from the Web Services call.

6.2 Do I have to parse the date field from the SAP Office Mail system?

It is recommend to parse the dates and convert them into java.util.Date objects. Date objects are normally used by the UWL to sort. This is only possible if the Dates are in the Java Date object format. A simple transforming code could be look like this:

```
DateFormat df = new SimpleDateFormat("yyyy-MM-dd hh:mm:ss");
Date myDate = df.parse(dt);
```

6.3 Why do I need a separate Attachment Provider Service?

Attachment's conent is retrieved as an array of bytes (byte[]). Unfortunately the SO_ATTACHMENT_READ_API1 RFC is not able to deliver Web Service convenient return values for byte[]. For this reason a separate provider was introduced which calls the data via JCA from the Backend. Of course this is only an work-a-round.

6.4 Deployment of the Portal Application DCs failed. Error message: Config archives could not be updated?

Keep in mind to pack and extract the configArchives within your Portal Application DCs, e.g. com.sap.netweaver.bc.uwl.configArchive, with a JAR tool instead with WinRAR or WinZip. Tools other then JAR packer are known to produce a different archive Checksum than the JAR packer.

7 References

	Source
[EX2006]	RSS Connector example and tutorial, by UWL Development team
[JR2006]	JRA Online Documentation, help.sap.com
[BP2006]	SDN BPM Support-Forum
[WS2006]	Web Services Online Documentation, help.sap.com

