Building Your Own BAPIs — A Step-by-Step Guide

Ralph Melone

Ralph Melone joined SAP in March 1998 in the Curriculum Development Area, where he is responsible for the integration technology curriculum for BAPI, RFC, Internet, and object-oriented technologies. Prior to joining SAP, Ralph was a Microsoft Certified Instructor, Engineer, and Solution Developer. He had his own consulting firm, and has held management and IT positions with AT&T and ADP.

(complete bio appears on page 72)

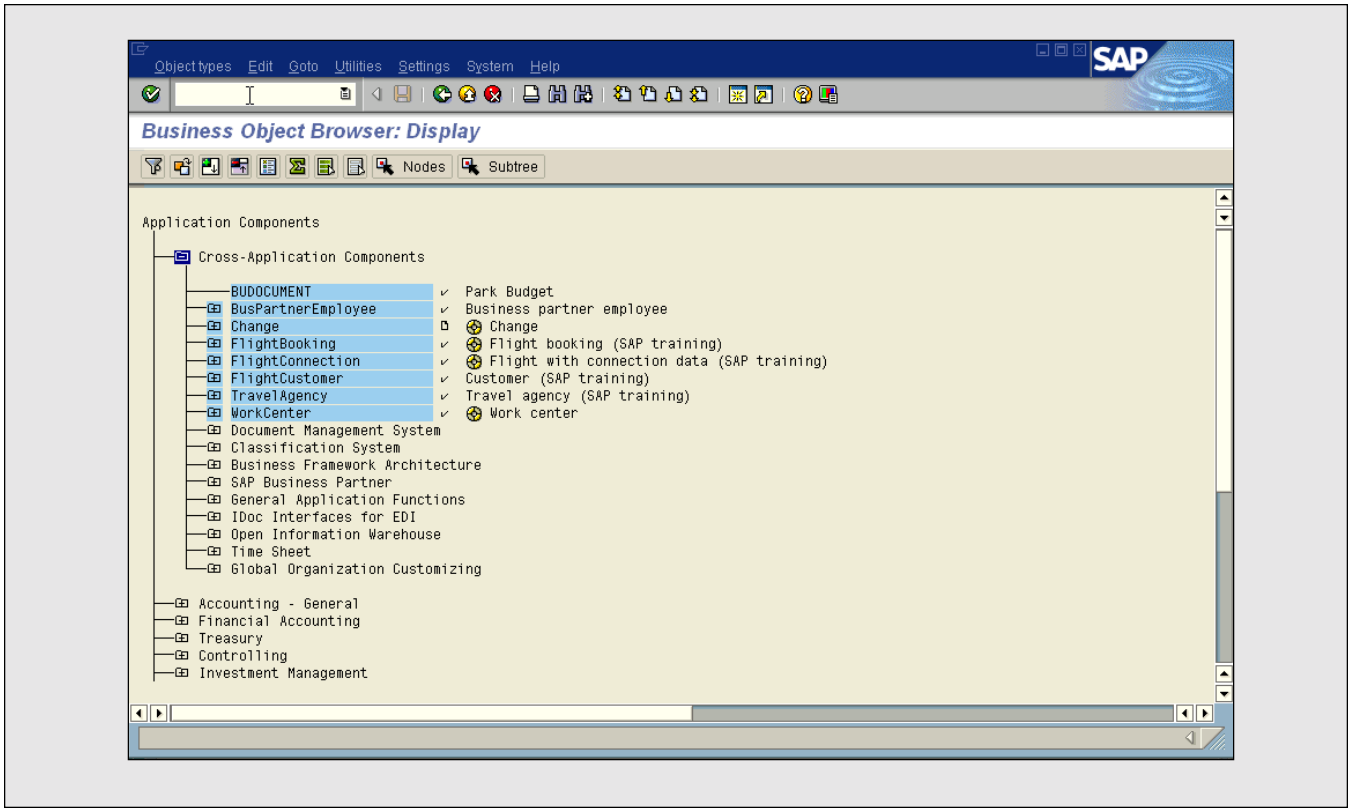
If you are surprised to learn that BAPIs are precisely defined interfaces that provide access to R/3 processes and data, that they are methods of SAP Business Object types, and that these object types and their BAPIs are described and stored in the Business Object Repository, read no further — this article is not for you. If, on the other hand, you have some experience with crafting applications that utilize BAPIs and want to know how to actually create and implement a BAPI in R/3 in the event that an appropriate BAPI does not exist, stick around. I'll show you how.[1]

Before I delve into the details of how to build a BAPI, let's examine when you might want to do so.

Let's say that you have been charged with developing a Web-enabled application that provides sales people with a list and details about customer inquiries. Your company IT guidelines require the use of COM, Visual Basic, and, of course, R/3 to develop this application, and further specify that anyone with authorization needs to be able to access the customer inquiry information via Internet Explorer. It's clear to you that the application needs to employ the CustomerInquiry object type. After a quick search through the Business Object Repository (BOR), you discover that for this object type, the necessary BAPIs — GetList and GetDetail — do not exist. You now set out to build these BAPIs.

[1] Prerequisites for this article are a basic understanding of the ABAP function modules, and some familiarity with the R/3 Function Builder interface of the ABAP Workbench. No Visual Basic, Java, or C++ knowledge is needed.

Figure 1 A View of the BOR Using the Business Object Repository Display Transaction



Assuming you have searched through the BOR and identified the suitable object types for your application, and made certain that the BAPIs needed by your application don't exist, how do you build the BAPIs you want? What do you do? How do you get started?

After offering a brief overview of the infrastructure in which BAPIs operate, I will show you, step by step, how to build a GetList BAPI for the FlightCustomer Business Object type. In so doing, I hope to demonstrate to you that building a BAPI is a simple and straightforward process. Please note that all instructions and screen shots used in this article are from R/3 Release 4.6B.

The BAPI Infrastructure

All SAP Business Object types and their methods are defined and described in the Business Object Repository (BOR),[2] the object-oriented repository in the R/3 system. **Figure 1** shows the BOR hierarchy, which maps to the application hierarchy of R/3. Our focus will be on the FlightCustomer Business Object type.

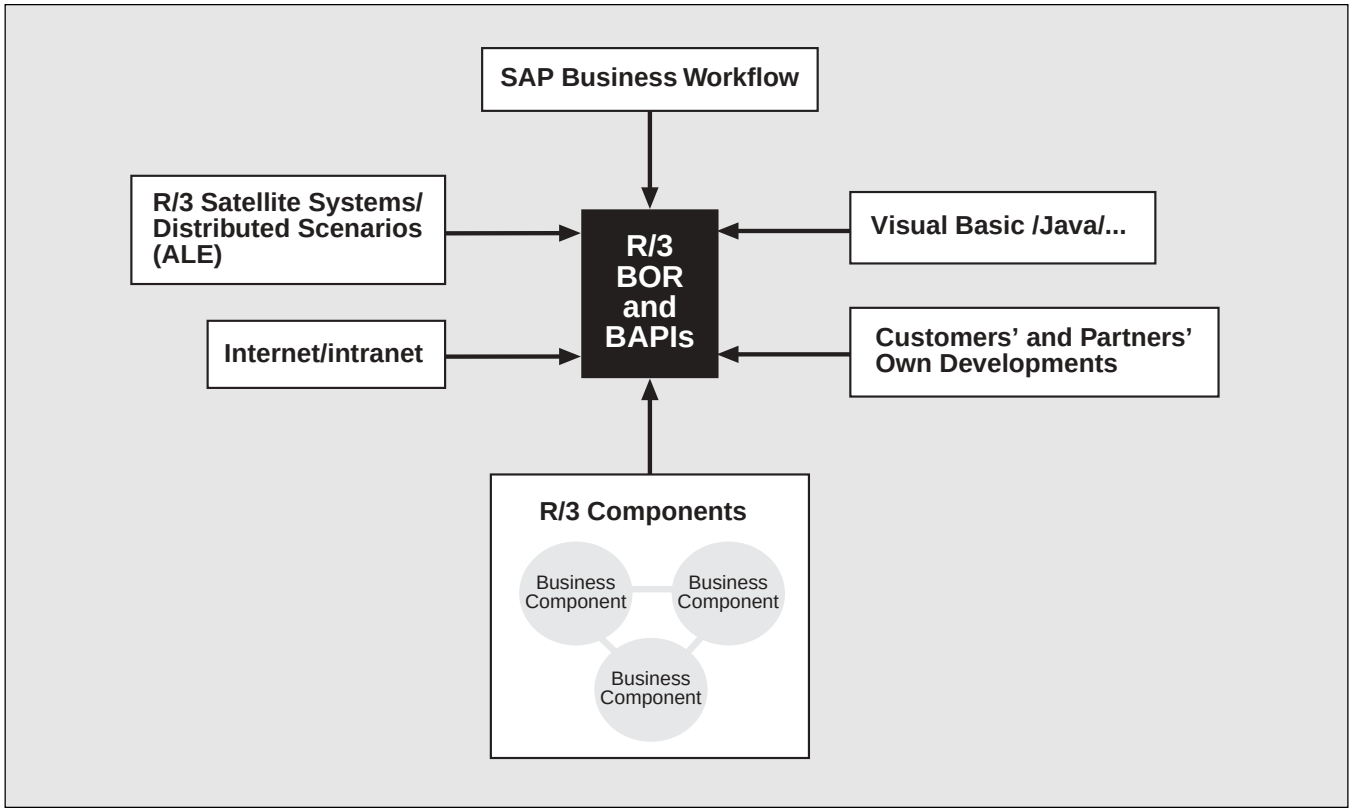
With regard to SAP Business Objects and BAPIs, the BOR provides the following essential functions:

- It provides an object-oriented view of the R/3 system data and processes. R/3 application functions are accessed using methods (BAPIs) of SAP Business Object types. R/3's functionality is simply exposed as a set of Business Objects. The implementation details of these objects are hidden. Only the methods defined as BAPIs can be accessed from the outside.

- It arranges the SAP Business Object types

[2] The BOR was introduced in R/3 Release 3.0, at the same time as SAP Business Objects and SAP Business Workflow. The introduction of BAPIs followed in Release 3.1.

Figure 2 Once Built, Your BAPI Can Be Used by a Variety of Applications and Components



according to the component hierarchy of the SAP R/3 system. It also provides a search mechanism within this hierarchy for finding and returning Business Object types.

- It defines and describes SAP Business Object types, their key fields, and their methods, including the BAPIs.[3]

- It manages the SAP Business Object types and their associated BAPIs in release upgrades.

- It ensures interface stability.

Once you build your own BAPI, it is added to the BOR. (The specifics about how that gets done will be presented later.) Once your BAPI is added to the BOR, just like the other BAPIs with which you are familiar, it can be used by a variety of applications and components across your R/3 landscape. (This concept is illustrated in **Figure 2**.) All that is exposed is the interface specified by you, the BAPI developer.

Our Model Scenario

To build a GetList BAPI for the FlightCustomer[4] object that returns a list of FlightCustomers, we do the following:

1. Identify all the requirements for the retrieval of R/3 data for our application. In general, an SAP Business Object type, whether it be the CustomerInquiry, SalesOrder, or the FlightCustomer object, has a key. This key, which is defined in the BOR, can consist of

[3] SAP Business Object types can have methods that are not BAPIs. These methods are used for SAP Business Workflow.

[4] The FlightCustomer object type was built specifically for SAP training systems. This object type is available in R/3 systems that have an IDES client installed.

several key fields. It is the contents of the key fields that uniquely identify an individual instance of an SAP Business Object type.

2. Define a structure that our BAPI can use to store the returned key fields. In our case, we will be defining a structure to store two parameters: Customer ID and Customer Name.

3. Use the Function Builder to write our ABAP code.

4. Use the BOR/BAPI Wizard to create the BAPI and attach it to the FlightCustomer object.

5. Document our BAPI.

✔ Tip — Let the Standardized BAPIs Be Your Guide

Don't attempt to build a BAPI without first consulting the "BAPI Programming Guide," which can be found on the Open BAPI Network Web site at ***http://www.sap.com/bapi****. This reference provides SAP-prescribed guidelines for BAPI structure and behavior. This is an invaluable reference! The very first things you should familiarize yourself with are the standardized BAPIs. Three of the most popular standardized BAPIs are:*

- *GetList — With this standardized BAPI type you can select a range of object key values, such as company codes or material numbers.*
- *GetDetail — This standardized BAPI type uses a key field (e.g., company code or material number) to retrieve details about a specific occurrence of a Business Object. The data is returned to the calling program.*
- *GetStatus — This standardized BAPI type can be used to query the status of an SAP Business Object, for example, the processing status of a sales order.*

Step 1: Identify the BAPI's Interface Parameters

Clearly, different business scenarios will require different SAP Business Object types and key fields. In our simple scenario, we're using just the FlightCustomer object type. Your real-world applications will most likely require multiple object types.[5]

Once you identify which object types are required by your application, you must identify the relevant import and export parameters for the BAPI. You do this on an object-by-object basis. Why? Well, if the BAPI is going to export parameters, as it is in our simple example, that data has to have a place to go. We will be establishing a structure for this purpose in the next step.

And if key fields are passed to the BAPI by a calling program, the key fields must be set as import parameters in the function module of the BAPI. In this way, a specific instance of an SAP Business Object can be identified. If we were to take our model scenario a step further and create a GetDetail BAPI for our FlightCustomer object type, we would set the CustomerNumber as an import parameter into the GetDetail BAPI's function module, because the CustomerNumber is the key field for the FlightCustomer. For BAPIs that create or generate instances of SAP Business Object types (e.g., Create or CreateFromData), the key fields of the Business Object type must be set as export parameters in the BAPI function module.

In our model scenario, where we are simply building the GetList BAPI for the FlightCustomer object, a key field is *not* required as an import parameter. This BAPI merely *returns* two key fields: Customer ID and Customer Name.

[5] If, for example, you were building a BAPI for retrieving personal information about employees, you would expect to use the EmployeeAbstract, EmployeePrivAddress, and EmployeePersonalData objects.

Step 2: Define a Structure for the BAPI

Now it's time to define a structure that the BAPI can use to store the two returned key fields (Customer ID and Customer Name). This structure parameter contains the values that will be transferred in the ABAP function module. How we name parameters is very important, because these names become the names associated with the method (BAPI) parameters in the Business Object Repository. They are the names exposed to the application programmers outside of R/3.

So, be sure to keep the following guidelines in mind when naming the parameters:

- Names must be in English.
- Avoid abbreviations.
- Names can be up to 30 characters long (20 characters for R/3 Release 4.0 and earlier).
- Parameters that are unique to a BAPI (e.g., those associated with only a Purchase Requisition BAPI) should have names that make sense to developers who work outside the R/3 system. Parameters that are common across all BAPIs (e.g., "standardized parameters" such as "Address" and "Return") should follow the prescribed nomenclature, a practice that is the key to consistent error processing.

The structure we will create is called ZBAPICUST. It is based upon the SCUSTOM table, the customer table used in the sample flight reservation training application we are using in our model scenario. (SCUSTOM contains all the relevant fields about a customer, such as client, customer number, customer name, and so on.) We will name our fields "ID" and "NAME."

✔ Tip
It is important to keep the names of the BAPI parameters that are in the BOR and the names of the parameters in the associated function module identical. As a BAPI developer, you need to empathize with the user — e.g., the Visual Basic or Java programmer. Users of the BAPIs are very dependent upon the names and documentation of the parameters to ensure a successful BAPI call.

The data formats for the ID field will be based upon the data element S_CUSTOMER, which is a data type of NUMC length 8. The NAME field will be based upon the data element S_CUSTNAME, which is a data type of CHAR length 25.

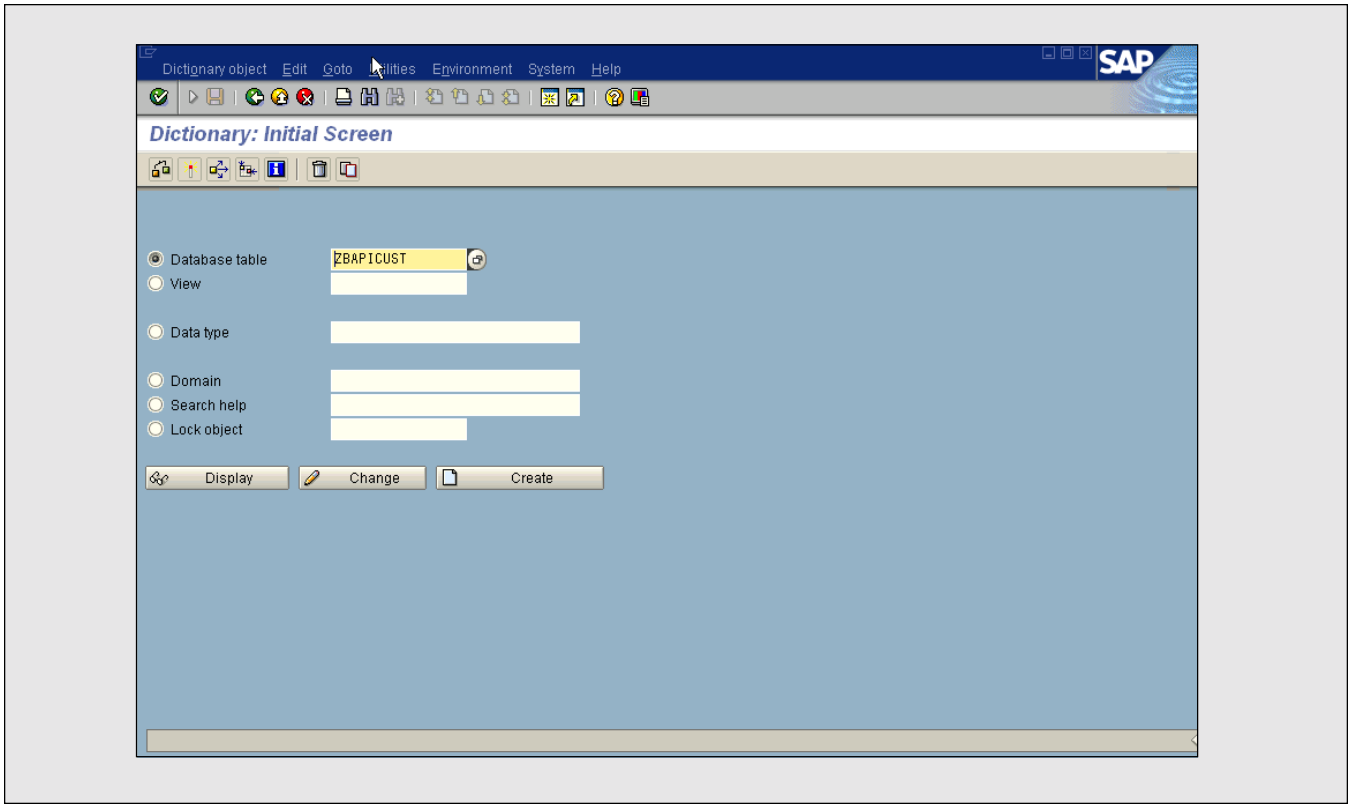
✔ Tip
Define data formats in a "neutral" way. Standard codes (e.g., ISO codes) should be used wherever possible. For fields that contain date information, use ABAP data type D, which uses the YYYYMMDD format, where YYYY is the year, MM is the two-digit month, and DD is the two-digit day.

After we have specified the names, types, and contents of the required parameters, we need to identify and define the required data objects in the Data Dictionary.

Here are the steps you need to take:

1. Go to the Data Dictionary and define a structure for the BAPI. Follow the menu path **Tools → ABAP Workbench → Development** and then select the "ABAP Dictionary" or use transaction SE11 to display the Data Dictionary.

Figure 3 The Initial Screen of the Data Dictionary



2. In the **Database table** field, enter "ZBAPICUST", as shown in **Figure 3**. Click on the **Create** button to display the detail structure screen.

3. In the **Short text** field, enter "Table for FlightCustomer GetList Bapi" (see **Figure 4**).

4. Go to the **Attributes** tab and enter "A" (for Application Table) in the **Delivery class** field.

5. Go to the **Fields** tab and complete the entries for the two fields in the structure as follows:

   - For the field name, enter "ID" under **Fields**.
   - Check the **Key** field checkbox. This indicates that the field is a key field for the table.
   - Check the **Init.** checkbox. This indicates that Not Null is forced for this field.
   - For **Field type**, enter "S_CUSTOMER".
   - Press the Enter key, and the remaining entries for ID will be filled in from the S_CUSTOMER definition automatically.[6]
   - For the next field name, enter "NAME".
   - Check the **Key** field checkbox.
   - Check the **Init.** checkbox.
   - For **Field type**, enter "S_CUSTNAME".
   - Press the Enter key, and the remaining entries for Name will be filled in from the S_CUSTNAME definition automatically.

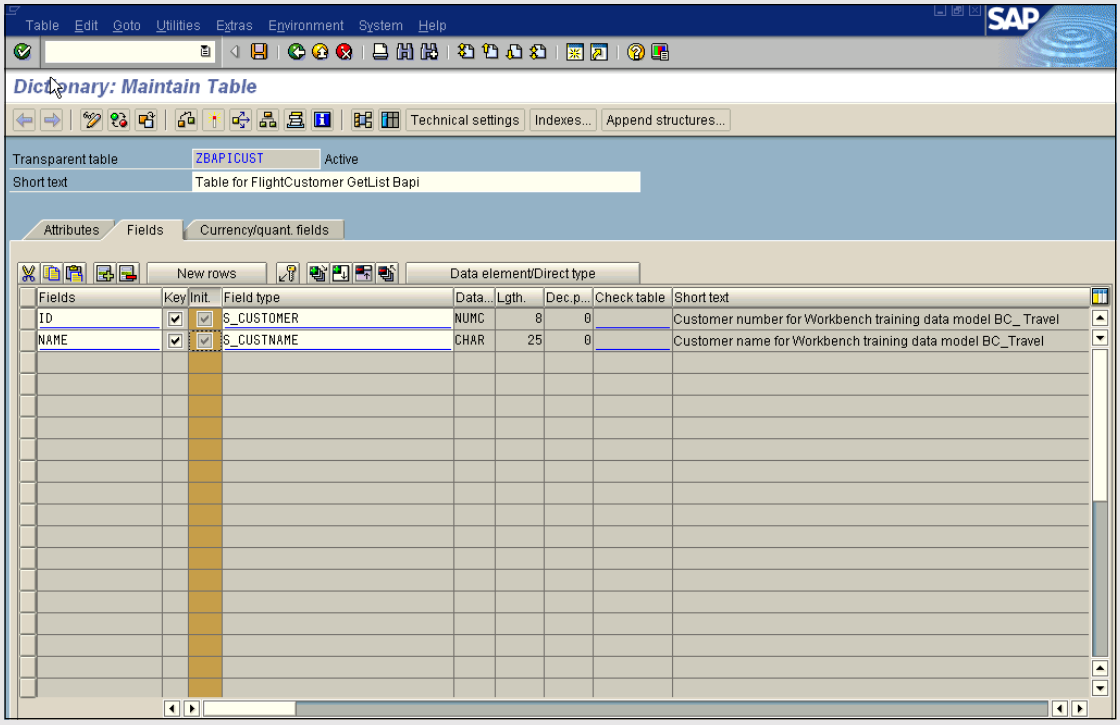
6. Save and activate the structure.

Step 3: Write the ABAP Code

Let's recap what we have done up to this point. We have identified a scenario for our BAPI, which is to

[6] S_CUSTOMER is a data element that already exists. It represents the customer number for the sample flight reservation training application.

Figure 4 Building the Structure "ZBAPICUST" for the FlightCustomer GetList BAPI



provide a list of FlightCustomers that will display both the ID number and the name of the customer. We then identified ZBAPICUST as the structure we will need to accomplish this. There are two fields within ZBAPICUST — the ID field and the NAME field. We used the Data Dictionary to define this new structure (which will be used in one of our BAPI parameters) and its fields.

We are now ready to write our ABAP code. We will use the Function Builder to write our function module, which will become our GetList BAPI for the FlightCustomer object type. We will also use the Function Builder to define global data that can be accessed by our function module and any other function modules that we might create later on to support this BAPI scenario — e.g., a GetDetail BAPI for the FlightCustomer object. We will also create a form routine to handle messaging.

To write the function module, we perform the following steps:

1. Go to the Function Builder and create a function group. Use the menu path **Tools → ABAP Workbench → Development → Function Builder**.

2. Go to the "Goto" menu and select **Function Groups → Create Group**.

3. Name the function group "ZSBAPI" (see **Figure 5** for the Create Function Group dialog).

Figure 5 Building the Function Group "ZSBAPI"

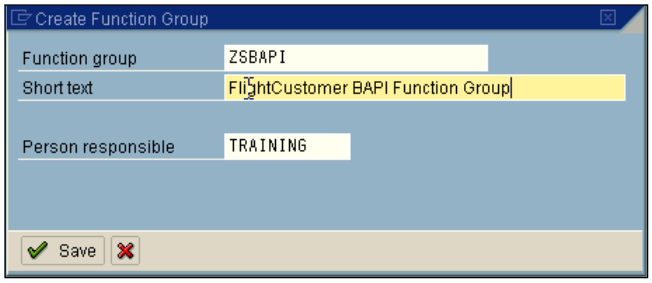
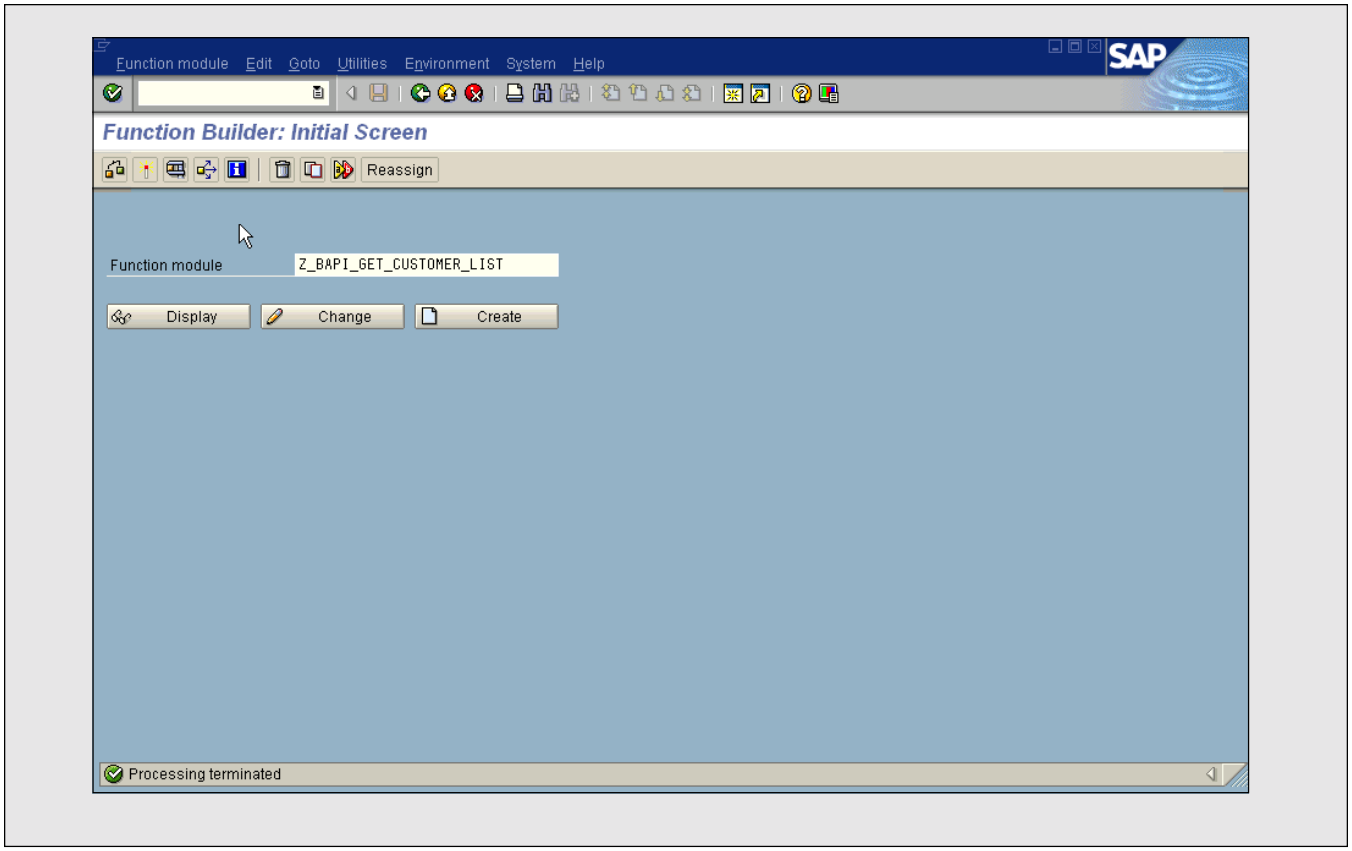


Figure 6 Building the Remote Function Module for the FlightCustomer GetList BAPI

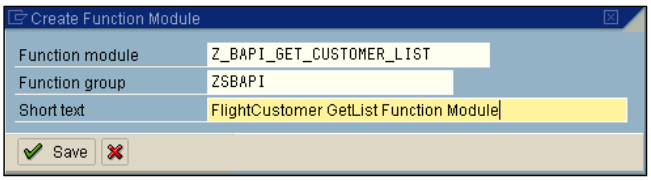


4. Create a function module and name it "Z_BAPI_GET_CUSTOMER_LIST". This is shown in **Figure 6**.

5. Once you have selected the **Create** button, the Create Function Module dialog will display. This is shown in **Figure 7**. Assign the function module to the function group "ZSBAPI" and provide a short text description of the function module.

6. Define the interface for your function module. On the **Attributes** tab (shown in **Figure 8**):
   - Enter the **Short text**. This text can be anything you like. You can see that I opted to enter "Flight Customer BAPI".
   - Under **Processing type** select "Remote-enabled module". Without this option, the function module cannot be invoked via RFC and cannot be used as a BAPI.

Figure 7 The "Create Function Module" Dialog



7. On the **Import** tab, you don't need to make any entries.

8. On the **Export** tab, enter "RETURN" for the **Parameter name** and "BAPIRET2" for the **Reference type** (see **Figure 9**). Press Enter and the **Type spec.** and **Short text** fields are automatically filled in for you by the system.

Figure 8 Defining the Interface for the Remote Function Module

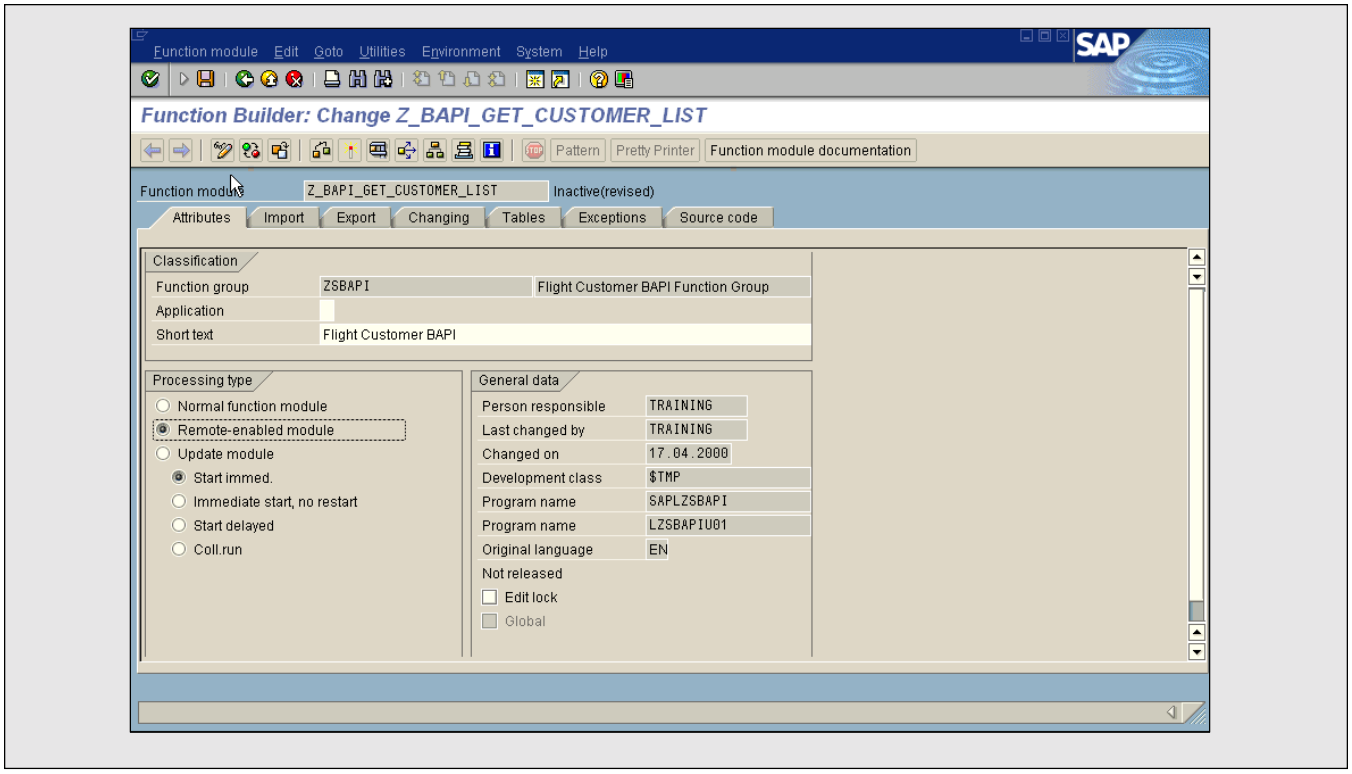


Figure 9 Setting the Export Parameters for the Remote Function Module

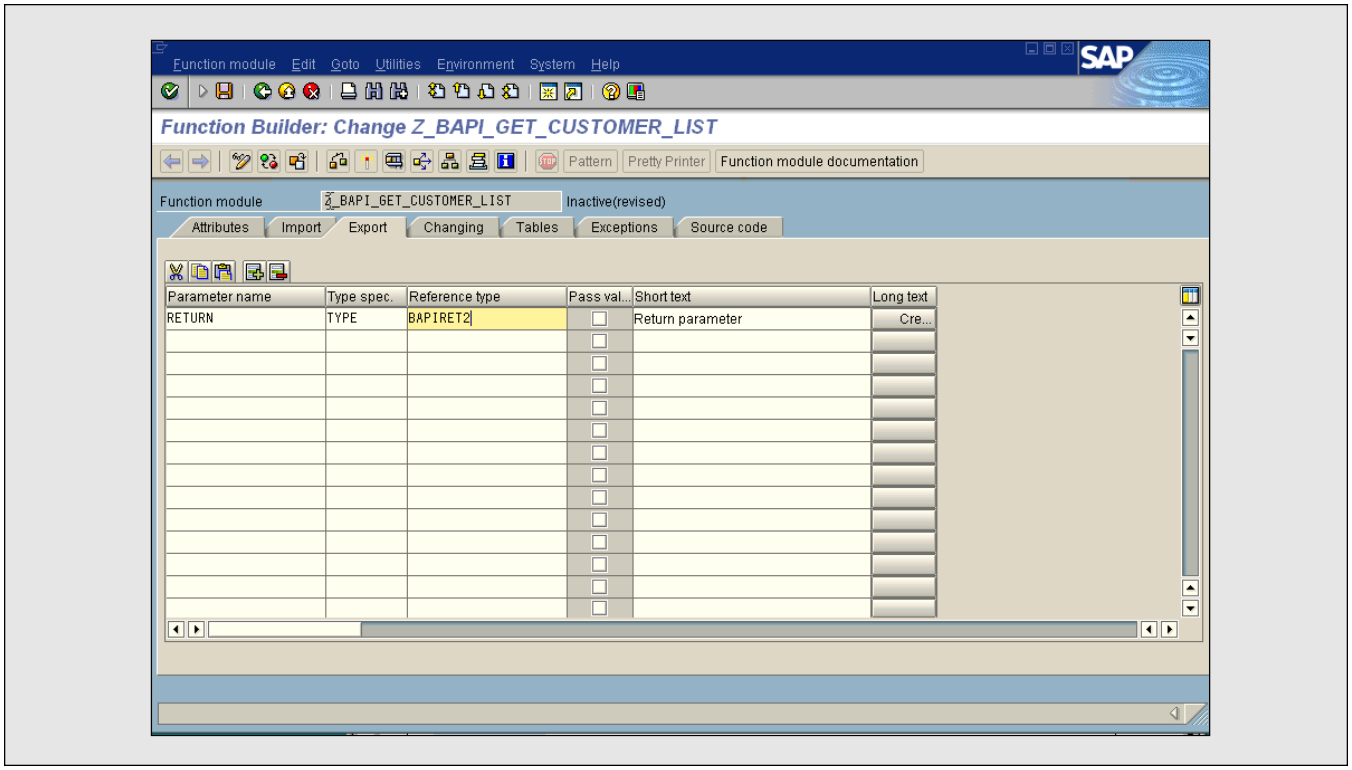
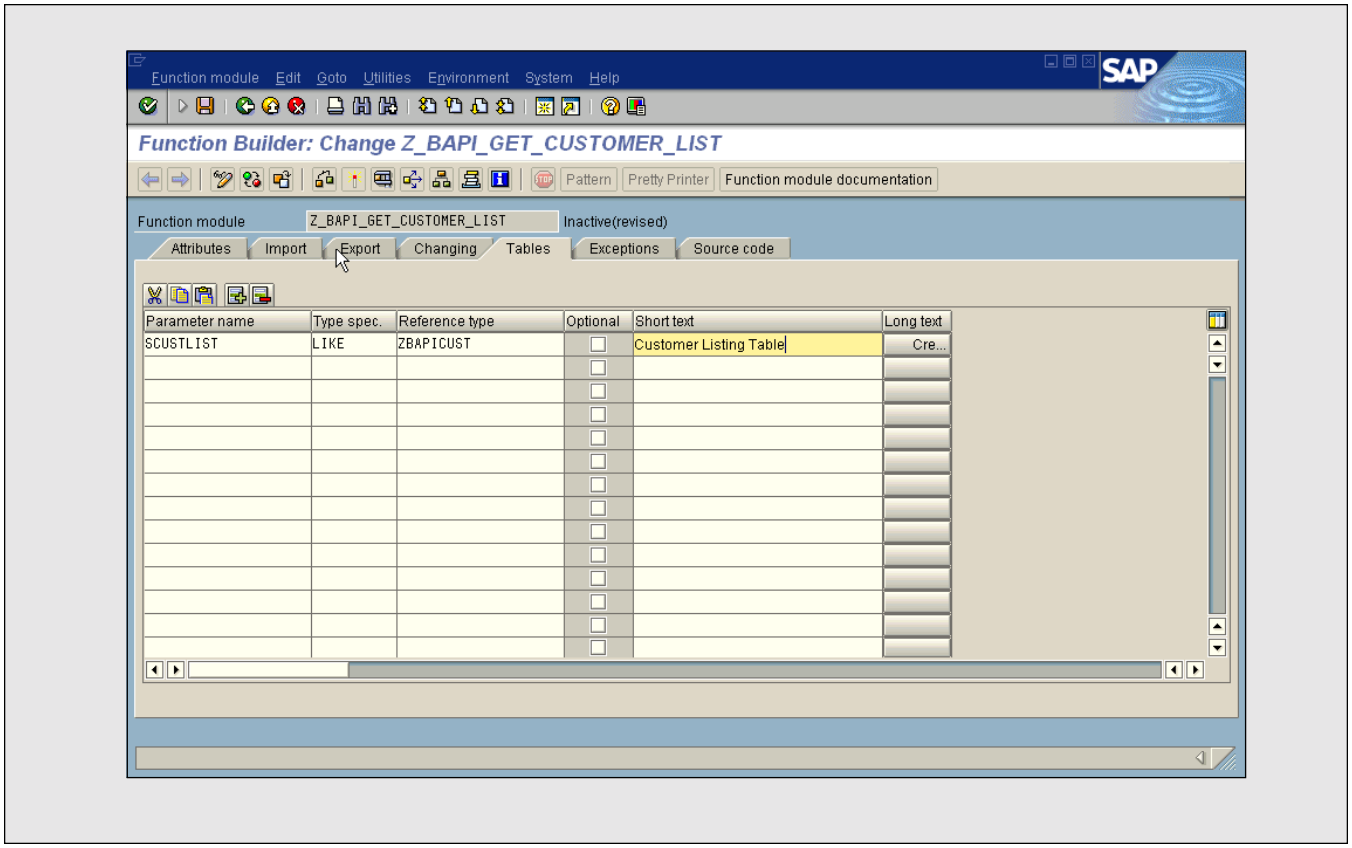


Figure 10 Setting the Table Parameters for the Remote Function Module



The **Type spec.** field indicates the type assignment for the reference type. "Return" is a standardized parameter name. We use BAPIRET2 for the structure name because it is also used as a reference structure in many other BAPIs.

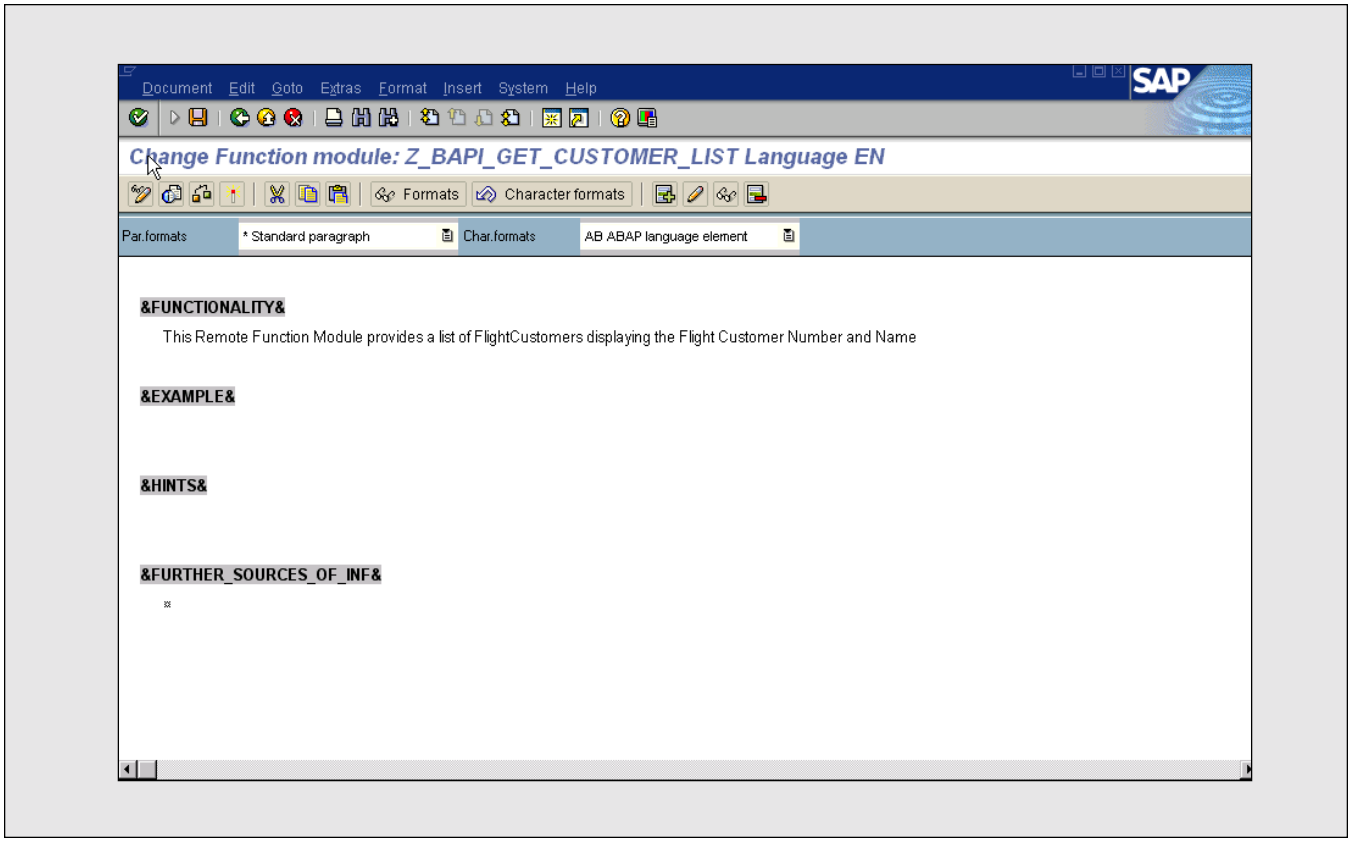
9. On the **Changing** tab, you do not need to make any entries either. Please note that changing parameter types are not supported in BAPI development.

10. On the **Tables** tab, enter "SCUSTLIST" for the **Parameter name**, and enter "ZBAPICUST" for the **Reference type**, as shown in **Figure 10**. As before, the **Type spec.** and **Short text** field are filled in automatically — in this case a "LIKE" type assignment is filled in by the system. The parameter name SCUSTLIST is unique to our GetList BAPI for the FlightCustomer object.

11. Do not make any entries on the **Exceptions** tab. Exceptions are not recommended for BAPIs.

12. In the Function Builder, you will find a **Function module documentation** button. Use this button to enter documentation for the function module. A documentation window provides a text editor with several predefined sections for you to fill in (see **Figure 11**). The **Functionality** section can provide documentation about what the function module does, as shown in Figure 11, as well as the data elements that are used, and information about the parameters that may help the user of the function module. The **Example** section may contain illustrative examples of using the function module, and the **Hints** section can be valuable for documenting any tricky and subtle functionality that might help the developer. The **Further Sources**

Figure 11 The Documentation Window for the Remote Function Module



of Inf section is provided for references to other documentation in the system relevant to the function module. (More on documentation toward the end of the article.)

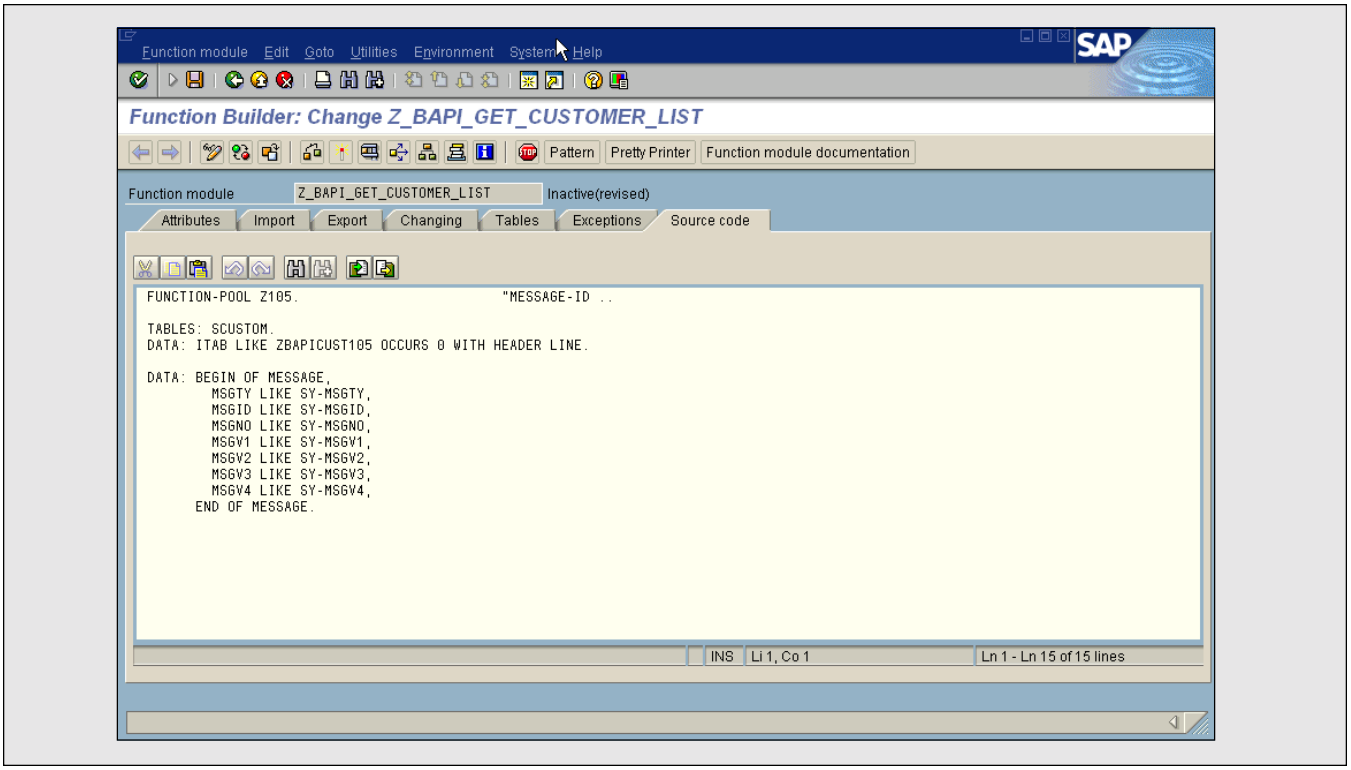
The Global Data

We now need to add global data for the function group — specifically declarations for the SCUSTOM table, data for a messages structure, and data for the internal table. The internal table will contain the application data that the GetList BAPI returns to the client program. We are creating global data so that additional BAPIs we create in this function pool[7] will be able to share it. We are only creating one BAPI in our particular example, but you could take this a bit further on your own and create a GetDetail BAPI. In this case, the GetDetail BAPI would be able to access the global data we have created here, and you would not need to repeat these declarations.

We are creating global data so that additional BAPIs we create in this function pool will be able to share it. We are only creating one BAPI in our particular example, but if you created an additional BAPI, it would be able to access the global data we have created here, and you would not need to repeat these declarations.

[7] A pool is a group of functions that have related tasks or functionality.

Figure 12 Global Data ABAP Program Code in the ABAP Editor



To add the global data, we follow these steps:

1. While in the ABAP Editor for the function module, go to the menu path **GoTo** → **Global Data** to access the global data area from the **Source code** tab, as shown in **Figure 12**.

2. Code the function module. The "SELECT" statement is shown in **Figure 13**. This statement moves all FlightCustomer data from the database to the ID and NAME fields of the structure we created in the Export tab of the Function Builder interface.

   In the function module code (Figure 13) you see the use of a messaging routine named SET_RETURN_MESSAGE. In this routine, we see some strange code: "IF 1 = 2". Why include a line of code that will never evaluate to "True"? The code in the message routine is designed to leverage the "Where Used" list utility. Allow me to explain…. This line of code, which obviously will not evaluate to "True," eliminates the possibility of ever executing the ABAP "MESSAGE" statement, which is crucial since BAPIs are not supposed to directly communicate with the end user. On the other hand, the same table (T100) that is used to store online user messages also contains the BAPI error message that will be returned to the client in the Return parameter. Using the MESSAGE statement in an ABAP program, even one that is never executed, automatically creates a reference so that you can find out where a particular message is being used. In short, to benefit from this Where Used capability in a BAPI, we include a MESSAGE statement, but never execute it. Comparing 1 to 2 for equality does the trick!

3. We will need to create a form routine that handles messages for the function module. To access the form routine (subroutine) area in the ABAP Editor, double-click on the name of the form routine being called in the function module code — i.e., SET_RETURN_MESSAGE in Figure 13. The SET_RETURN_MESSAGE form routine is shown in **Figure 14**. This is a standard routine

Figure 13 The ABAP Function Module Program Code for the FlightCustomer GetList BAPI

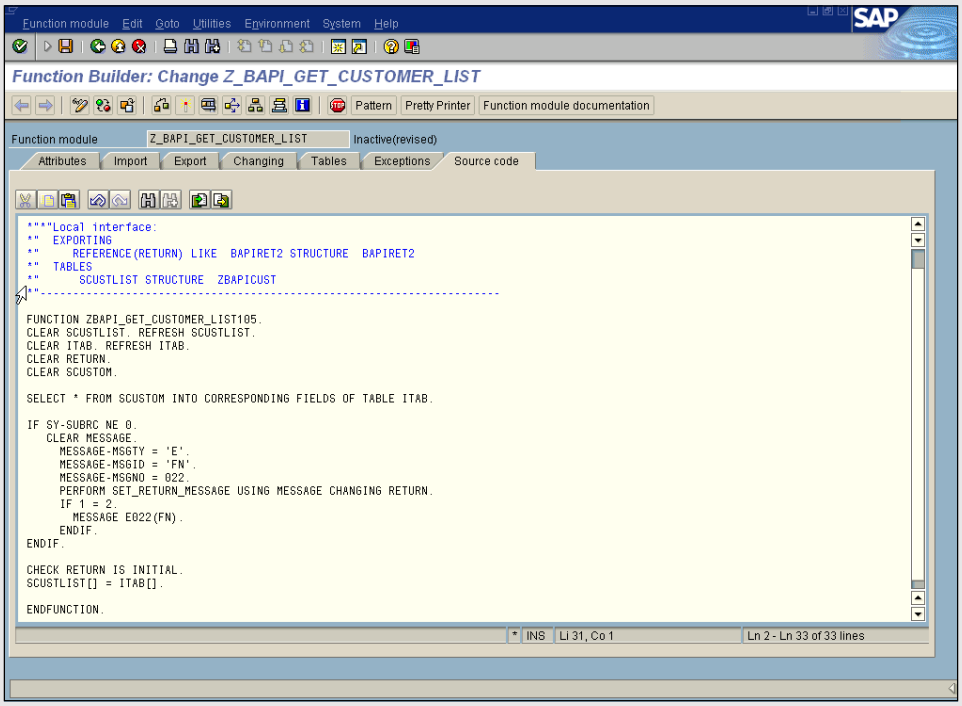


Figure 14 The ABAP Form Routine Program Code for the FlightCustomer GetList BAPI

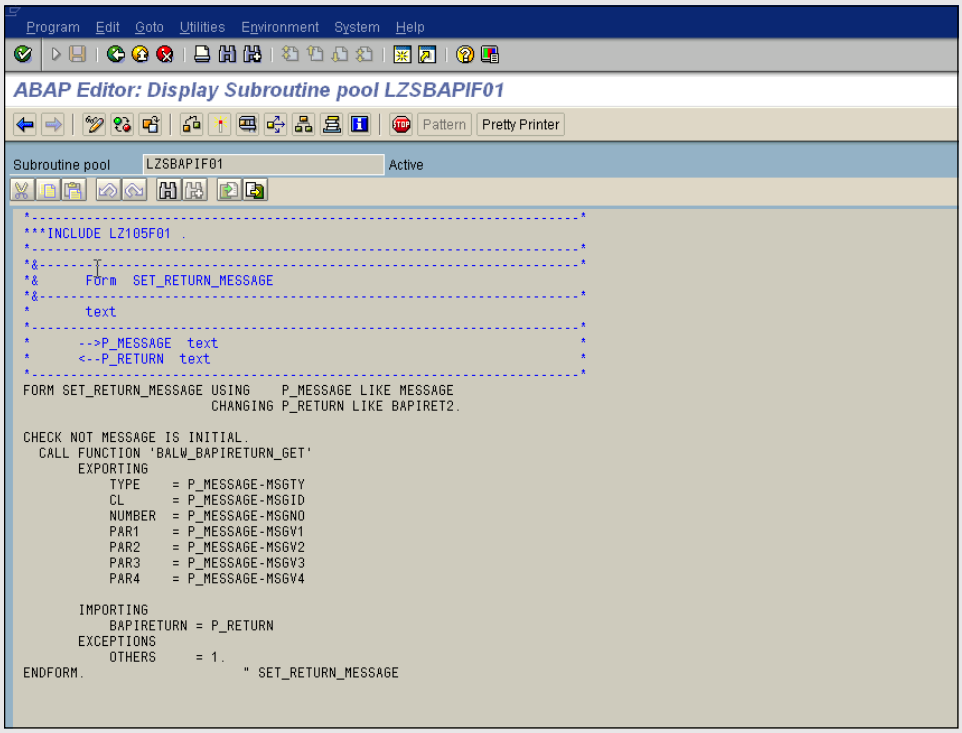
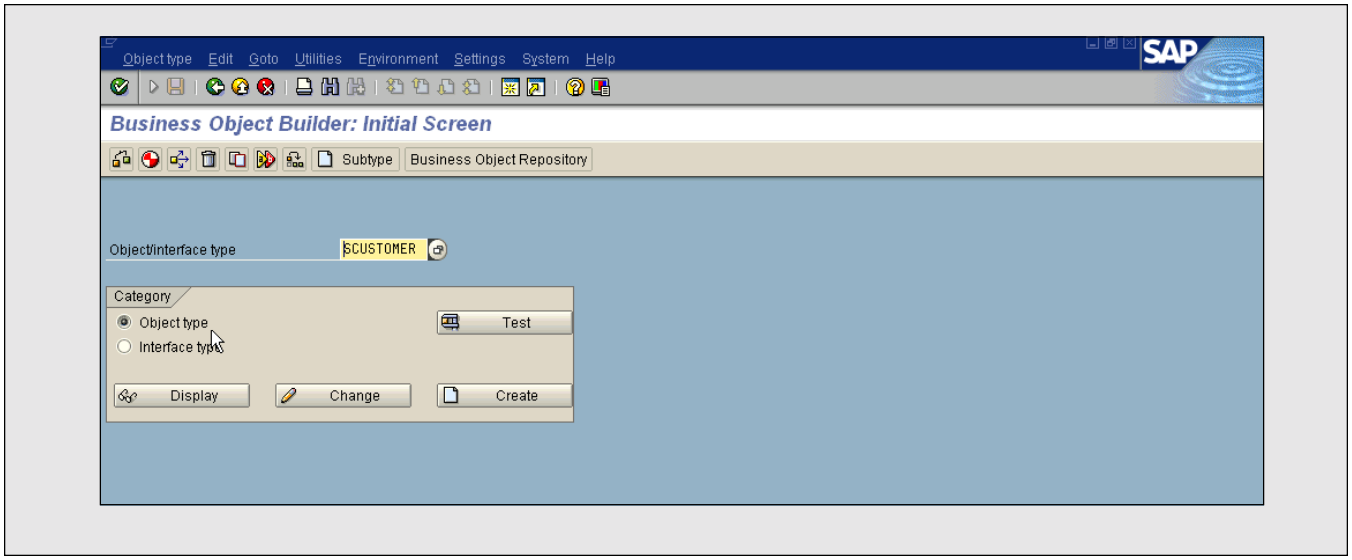


Figure 15 Business Object Builder Initial Screen



for BAPIs that handles the messaging structure for the BAPIRET2 structure that we defined earlier. This structure is one of several structures used in BAPIs to inform the external program — the program that calls the BAPI — on whether or not the BAPI was successful. Please note that we are creating this form routine for illustrative purposes only. There are standard functions we can use to accomplish this messaging functionality.

Step 4: Create the API Method Using the BAPI Wizard

In order to expose the remote function module as an API method (a.k.a. a BAPI), we need to use the BAPI Wizard. This tool will generate some additional code that is required so that our function module is a valid method of a Business Object type in the BOR. This code allows our BAPI to be called as a Workflow method within R/3 in addition to being able to be invoked by an outside client program. The steps to use the BAPI Wizard are outlined below:

1. Go to the Business Object Builder — you can use the main menu path:

   Tools → ABAP Workbench → Development → Business Object Builder

2. In the Business Object Builder initial screen (shown in **Figure 15**), enter "SCUSTOMER" as the object type. Then select the **Change** command button. This will launch the Change Object Type screen (**Figure 16**). In our example, we use SCUSTOMER as the external object type identification, and FlightCustomer as the "official" object type name. SAP Objects can be identified by both an object type and an object name. SCUSTOMER is the object type for the object named FlightCustomer. The SCUSTOMER object type serves as a unique internal R/3 object identifier. The object name serves as an alias that developers can use to identify an object through some meaningful name. **Figure 17** shows the **General** tab of the object type SCUSTOMER. Note that the object name is FlightCustomer.

3. Once we have displayed the Change Object Type screen by selecting the **Change** command button, follow the menu path:

   Utilities → API methods → Add method

   This will display the Create API Method Properties dialog, where you enter the function module name you created in the previous steps — e.g.,

Figure 16 *Change Object Type Screen*

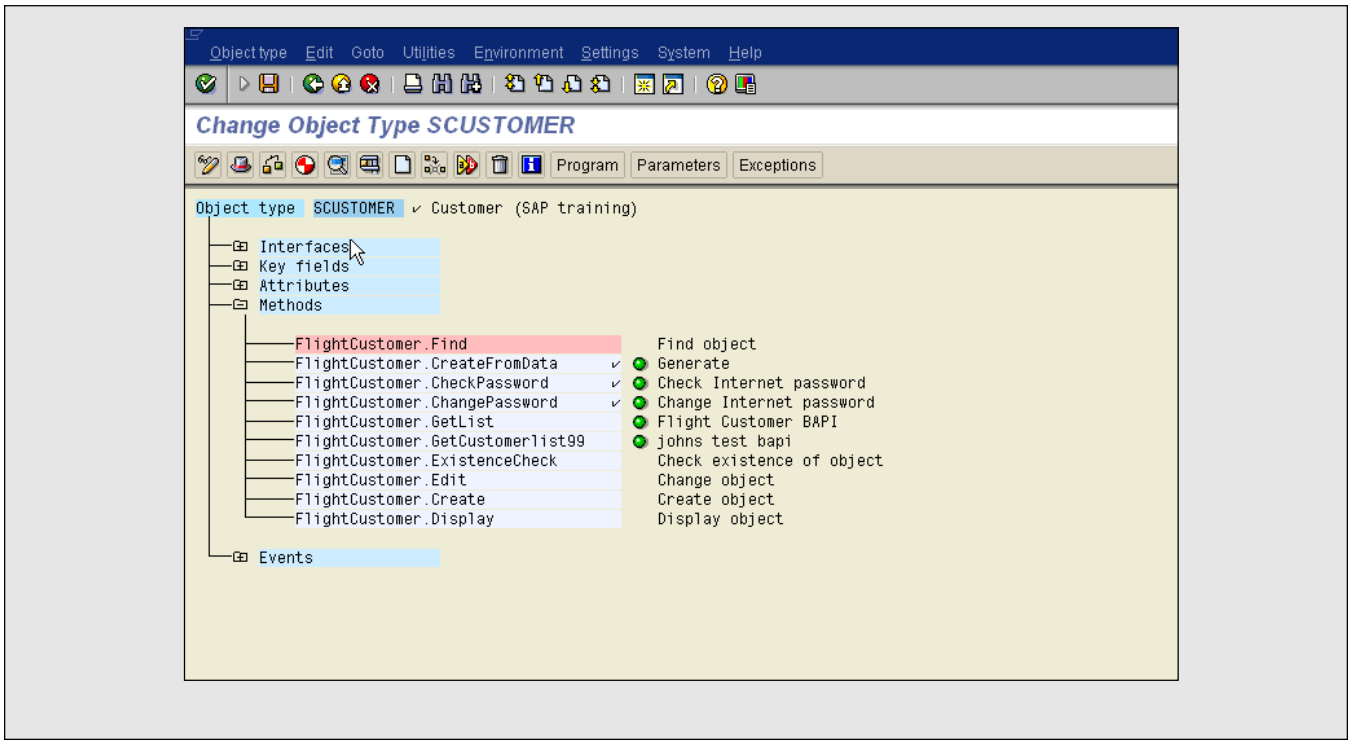
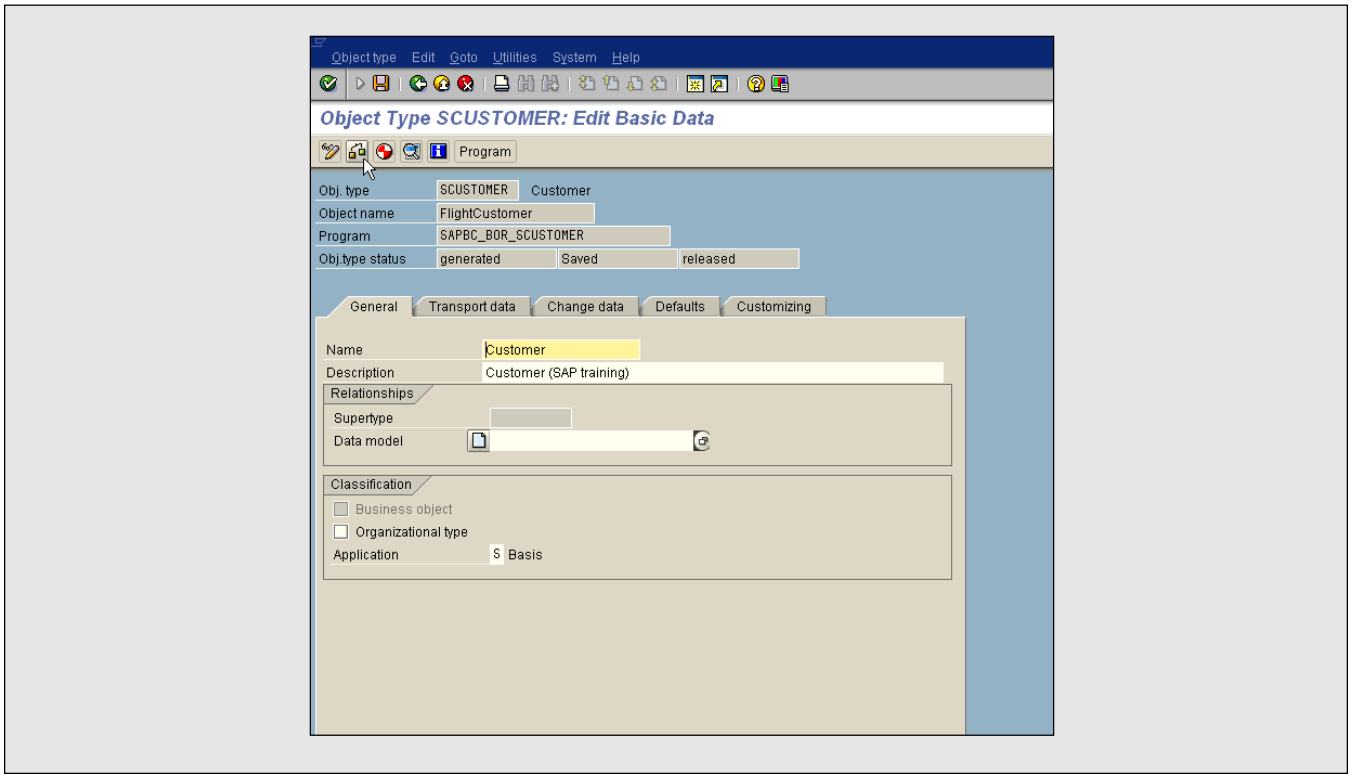


Figure 17 *General Tab of Object Type "SCUSTOMER"*



Z_BAPI_GET_CUSTOMER_LIST. You can see this in **Figure 18**. Select the checkmark button (✔) at the lower left of the dialog. This will invoke the BAPI Wizard and begin to walk you through the process of generating the BAPI in the Business Object Repository. You can see the first step of the BAPI Wizard, which takes us to the Create API Method Features dialog, in **Figure 19**.

4. Enter the method name, using the name "GetList", as shown in **Figure 20**. In the text boxes, enter an appropriate name and description for documentation purposes. In the **Characteristics** section, make sure the **Synchronous** and **Instance-independent** checkboxes are checked. An instance-independent BAPI means that R/3 does not expect a key field to be passed to it for instantiating a specific object instance.

5. Select the middle button (▶) at the lower left of the dialog. This will take us to the next step in the BOR/BAPI Wizard, the Create Parameters dialog, shown in **Figure 21**.

Figure 18 The "Create API Method Properties" Dialog

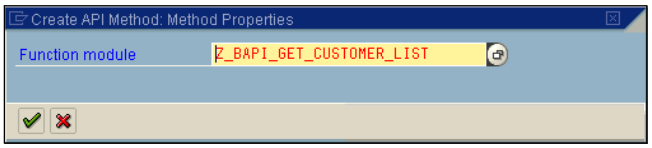


Figure 19 The "Create API Method Features" Dialog Before the Name Change

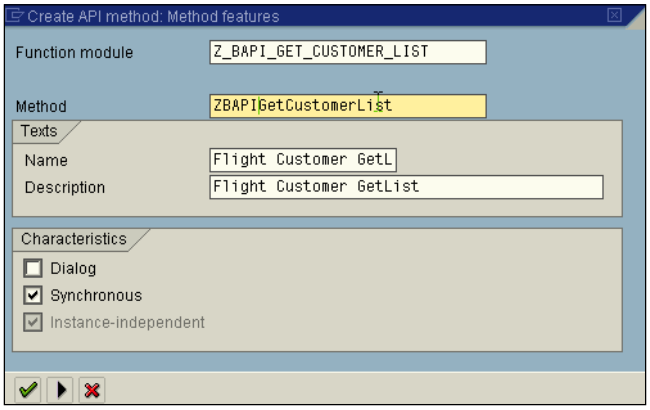
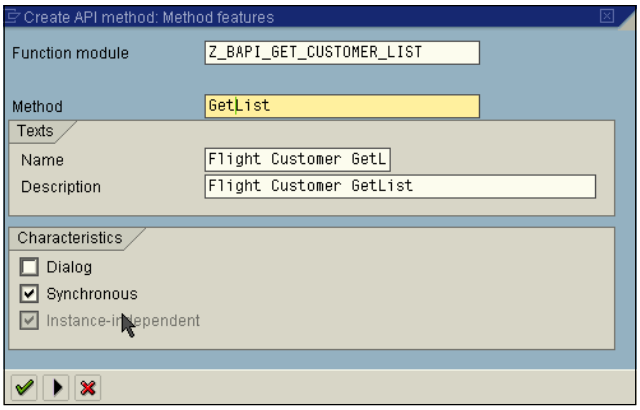


Figure 20 The "Create API Method Features" Dialog After the Name Change



In the Create Parameters dialog, a list of parameters and default names is displayed, which we need to edit as required. We will modify the parameter names as follows:

- Each new word in the parameter name must start with a capital letter — for example, "Scustlist".

- The end result of the BAPI Wizard is a program that is shown in **Figure 22**. The program starts with the statement "BEGIN_METHOD GETLIST". The BAPI Wizard generated this code for us. The program associates our GetList method with our object.

- Specify whether the individual table parameters are used for data import or data export. Our Return parameter should be export-only. Table parameters such as Scustlist can serve as both an Import and Export parameter. Table parameters are marked with a check in column "MLine" (multiple lines). Select the next button (▶) at the lower left of the dialog to bring us to the next

Figure 21 *The BAPI Wizard's "Create Parameters" Dialog*

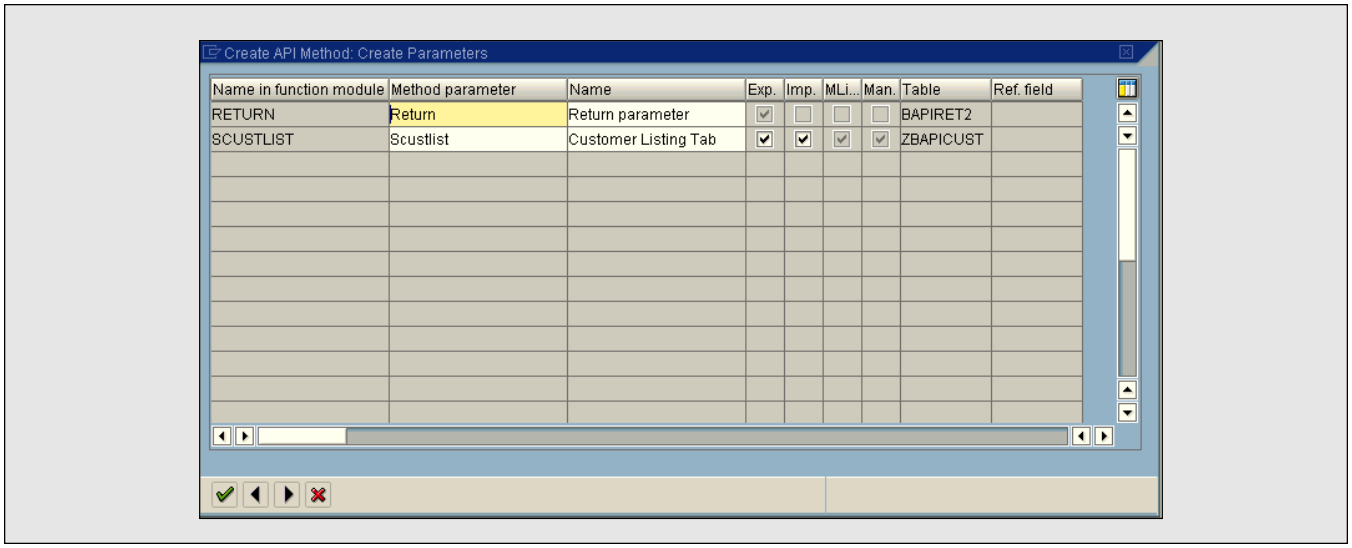


Figure 22 *Program "SAPBC_BOR_SCUSTOMER"*

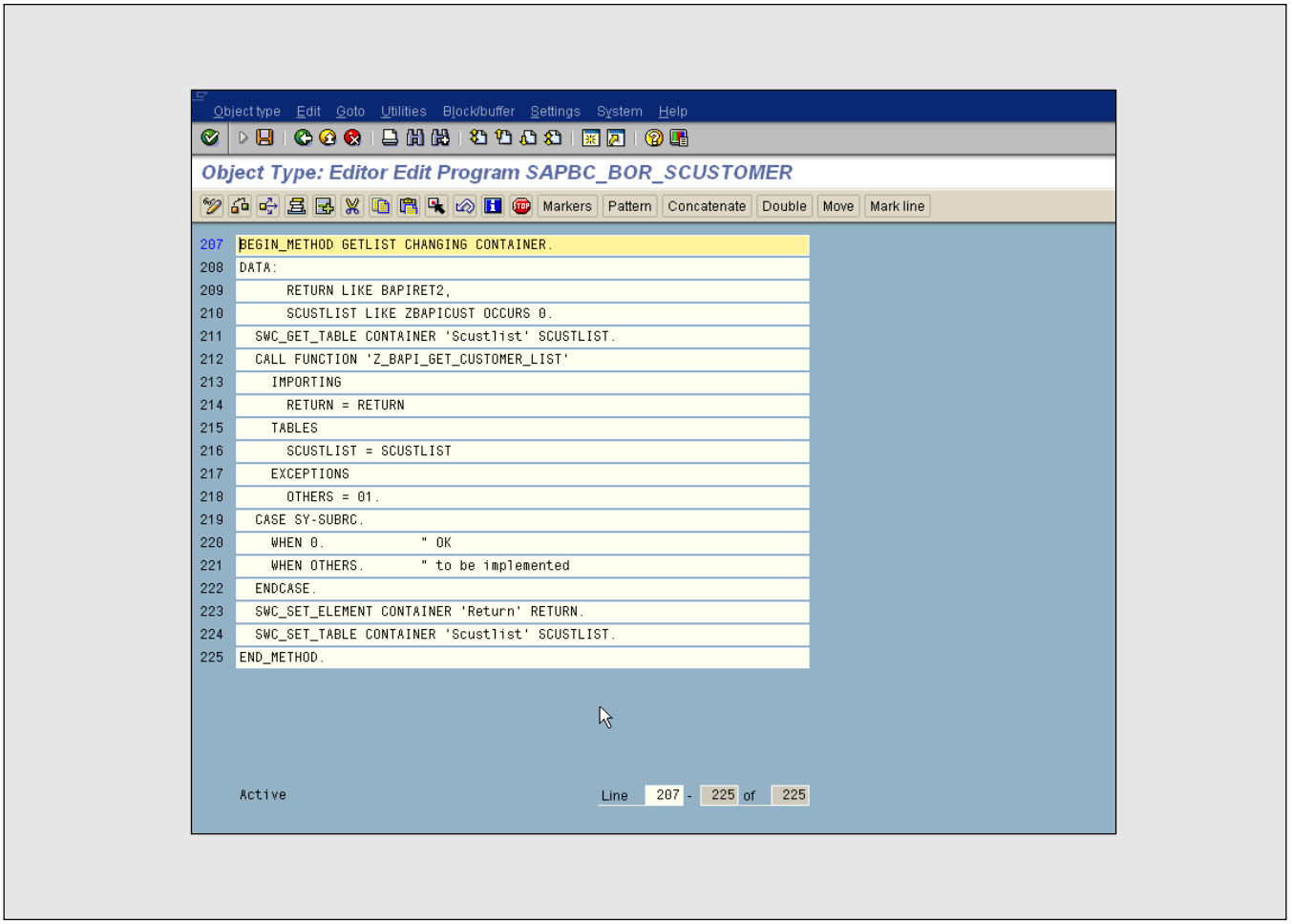
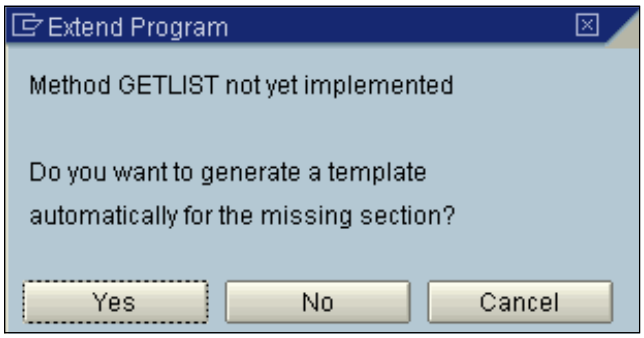


Figure 23 The BAPI Wizard's Extend Program Step



step in the BOR/BAPI wizard, the Extend Program step, shown in **Figure 23**.

6. The Extend Program dialog warns us that the BAPI is not yet implemented. Select "Yes" to generate the template that will add the BAPI as a method of the SCUSTOMER object type. This step adds "wrapper object-oriented" code to our function module, converting it to a method of an SAP Business Object.

7. The next step is to generate the BAPI entry. The BAPI Wizard has returned us to the Change Object Type screen. Select the plus (+) sign next to Methods to expand and display all the methods of SCUSTOMER. Our BAPI should now be listed with a green circle icon (green light). Select the BAPI once, and then select the Generate icon in the toolbar (fourth icon from the left). This is illustrated in **Figure 24**.

8. Test the BAPI by selecting your BAPI and then selecting the test tool in the toolbar (the sixth icon from the left), or you can press the F8 key. You will reach the screen shown in **Figure 25**. The results of the test are shown in **Figure 26**.

Figure 24 The FlightCustomer Object (SCUSTOMER) with the GetList BAPI Now Displayed

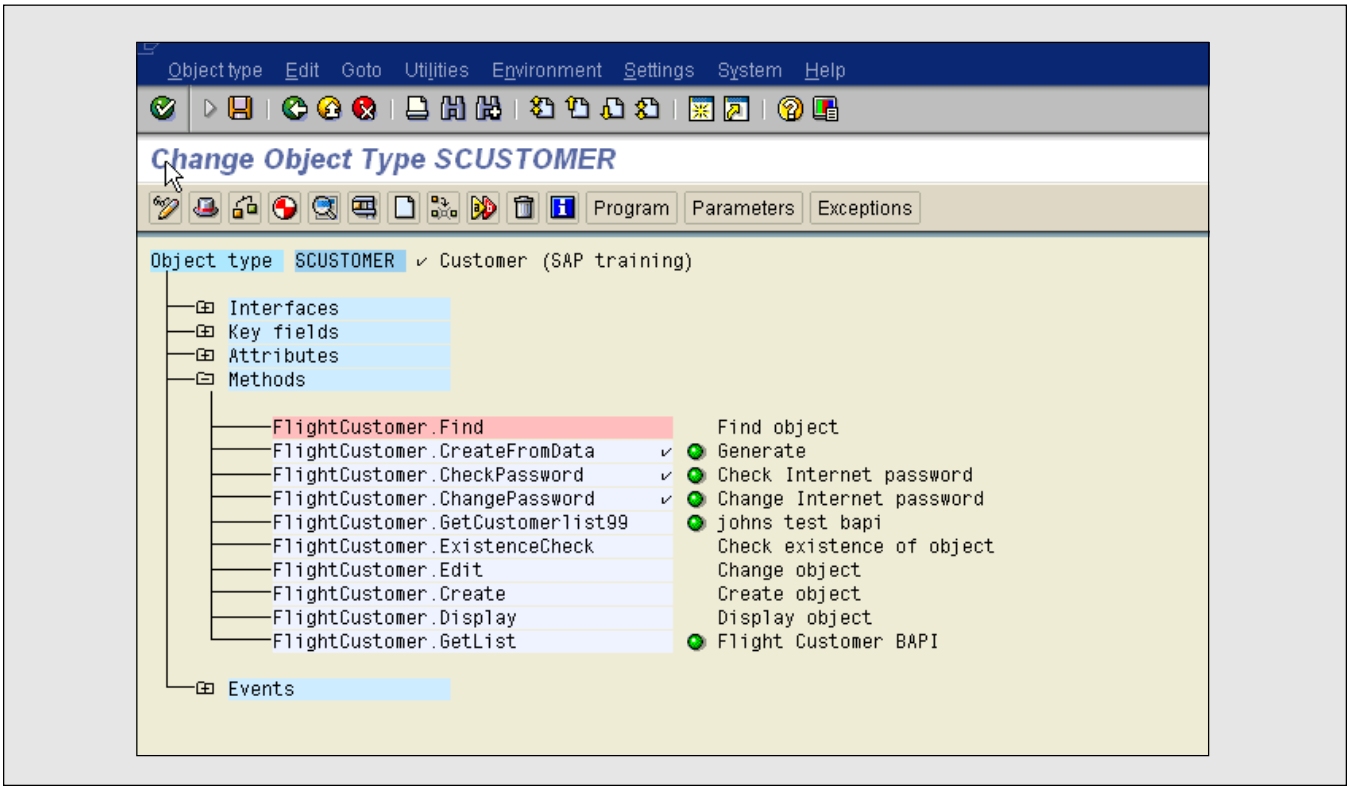


Figure 25 Testing the FlightCustomer GetList BAPI

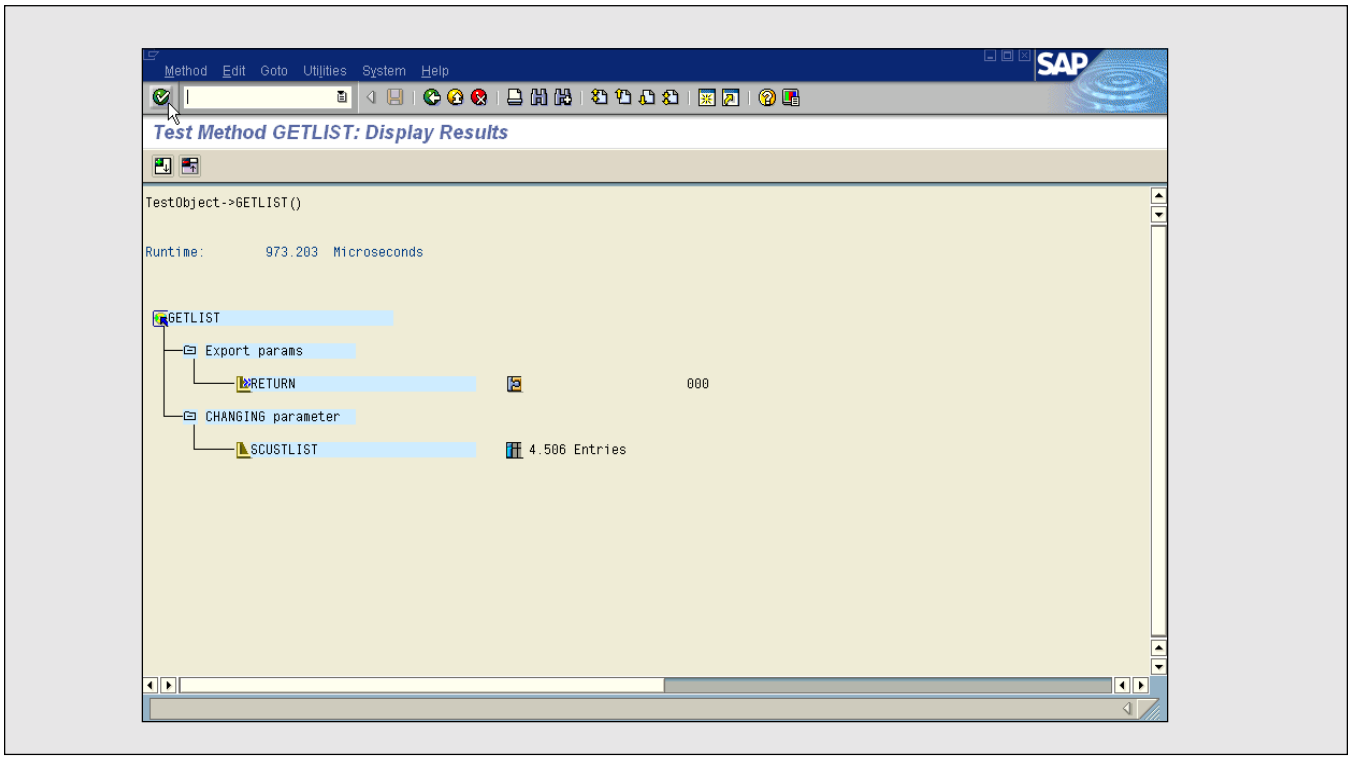
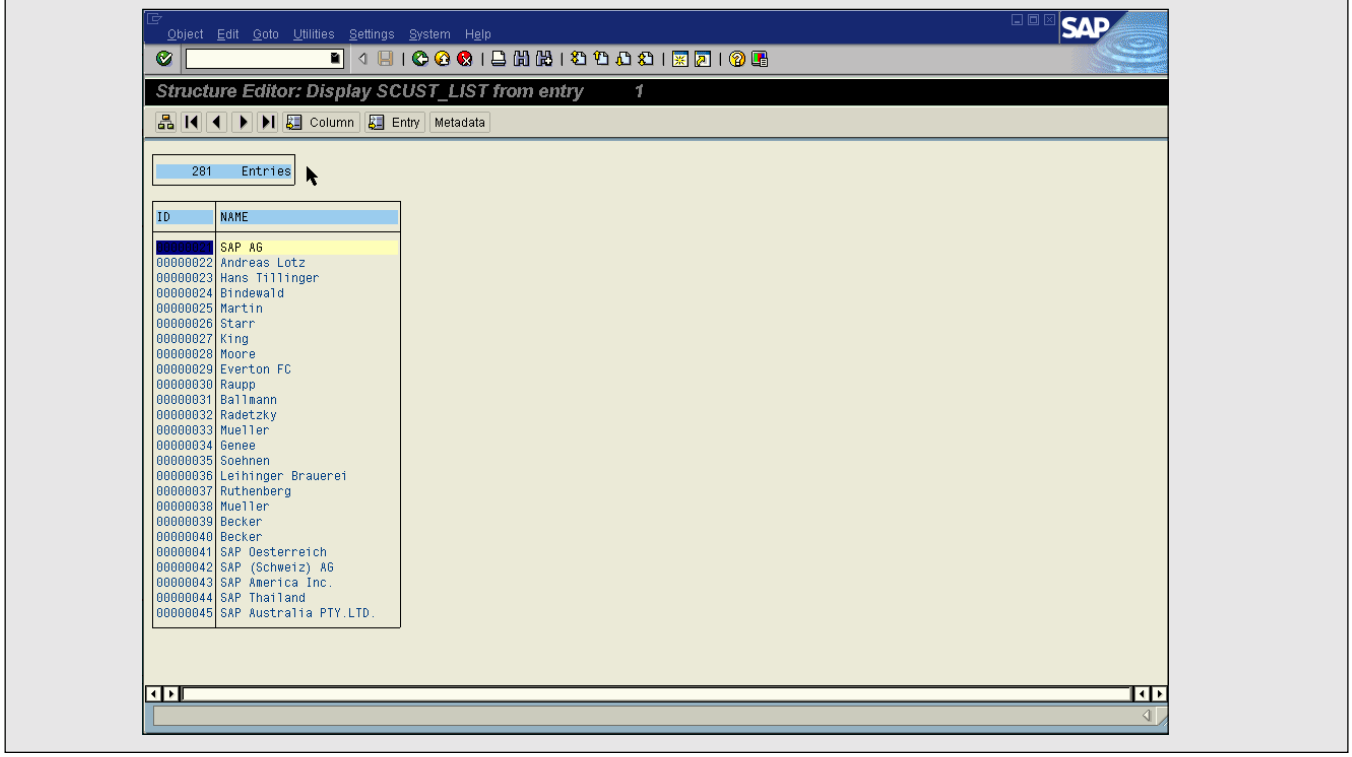


Figure 26 FlightCustomer GetList BAPI Test Results



Step 5: Document Your BAPI

The documentation on the BAPI and the interface parameters are created in the Function Builder for the function module that the BAPI is based upon. Among the components that must be documented are:

- The function module
- The parameters of the function module
- The fields of the parameters
- The key fields

The purpose of this documentation is to describe exactly what the BAPI can do and how external programs can use the BAPI. For example, for a GetDetail type BAPI, we would document what key fields must be passed as import parameters from an external program. We would also specify the exact detail that our BAPI sends back (exports) to the calling application, and if there are special authorizations required to use the BAPI. Other special areas should also be documented, for example, if the BAPI uses a COMMIT WORK or if there are any important limitations (functionality that the BAPI does not support). The documentation on Return parameters must describe all possible return values and messages.

The documentation on the BAPI and the interface parameters are created in the Function Builder for the function module that the BAPI is based upon. Among the components that must be documented are the function module, the parameters of the function module, the fields of the parameters, and the key fields. The purpose of this documentation is to describe exactly what the BAPI can do and how external programs can use the BAPI.

Helpful Hints

In order to maintain consistent access to R/3 data and processes, you must be sure to follow certain guidelines:

✔ **BAPIs must not contain certain commands.** These commands are CALL TRANSACTION, SUBMIT REPORT, and SUBMIT REPORT AND RETURN. The call of a BAPI must not trigger further Logical Units of Work (LUWs) that are independent of the BAPI. This is the reason these ABAP keywords are not permitted.

✔ **BAPIs must not invoke a COMMIT WORK.** In Release 4.0, those BAPIs that cause database changes must not use the COMMIT WORK statement. Instead, a special BAPI — TransactionCommit — of the BAPIService object type is used. This BAPI executes the COMMIT WORK. The program flow in this case would be the following:

Call BAPI to change data →
Call BAPI Service.TransactionCommit

✔ **BAPI structures must not use Include structures.** Enhancements to the Include structures generally lead to incompatible changes to the BAPI structure.

✔ **There should be no functional dependencies between two BAPIs.** A BAPI call must not be adversely impacted by an earlier call to a BAPI. A follow-up call must not assume an earlier call. For this reason, avoid using Set or Get parameters in the BAPI or global memory. A repeated call of one BAPI must produce the same result.

✔ **BAPIs must perform their own authorization checks.**

✔ **BAPIs must not be dialog dependent.** While Release 4.6 does away with this restriction, it's still a good idea *not* to allow your BAPIs to have dialogs.

They must not produce any screen output, including any function modules that the BAPI might call.

An external program — for example, a Java applet — calling a BAPI would halt execution if the BAPI called a dialog box. The Java program would stop execution and wait for some response back from R/3, which in turn is waiting for some action to be taken on the dialog.

✔ **BAPIs must not cause the program to abort or terminate.** Relevant messages must be communicated through the Return parameter. In this way, a standard for external developers is established. They can always check the Return parameters of BAPIs to determine success or failure of the BAPI.

✔ **For BAPIs that read large amounts of data — e.g., GetList BAPIs — a method of providing selection criteria or a limit on the amount of returned data should be specified, and a message should be sent to the calling program.** This message can notify the caller that a large amount of data is being sent. Some mechanism is needed for handling massive amounts of data. If a Java or Visual Basic program calls a BAPI that returns a large amount of data, all the data will be sent from the application server to the client.

✔ **Make the key of a modified record very specific.** This will minimize the locking of the record by different BAPIs. If a key is less specific (partial), then the chances of locking by multiple BAPIs are increased.

✔ **Define the processes and situations in which the BAPI will be used.** Before you begin building your BAPI, consider issues such as how the process will flow.

In our scenario, using the FlightCustomer object, for example, a request for a list of FlightCustomers is processed. Once the calling program receives the list, the flow could involve a request for details about a specific customer. By applying this concept and creating a complete model scenario, you can ensure that the BAPIs involved in your application will complement each other.

✔ **A BAPI should provide its functionality exclusively to one SAP Business Object.** It's a good idea to design your BAPIs to function independently of one another. This is one of the fundamental principles of good object-oriented program design. In this way, a GetDetail BAPI is not dependent upon the invocation of a GetList BAPI.

In our simple example, a GetDetail BAPI could be used independently of retrieving a list of FlightCustomers, since the user may already know which FlightCustomer they want details on.

A BAPI should provide its functionality exclusively to one SAP Business Object. It's a good idea to design your BAPIs to function independently of one another. This is one of the fundamental principles of good object-oriented program design. In this way, a GetDetail BAPI is not dependent upon the invocation of a GetList BAPI. In our simple example, a GetDetail BAPI could be used independently of retrieving a list of FlightCustomers.

Conclusion

BAPIs are one of the primary technologies for interfacing with R/3. Creating BAPIs requires a great deal of ABAP knowledge, a good working knowledge of the application area, and some knowledge of what external developers need to interface with R/3. At the heart of the last requirement is documentation. Poor documentation can make the best and most efficient BAPI unusable. Good and thorough

documentation can make the BAPI a very valuable component.

Documenting the BAPI should be considered a major step in the BAPI development cycle. It is the responsibility of the BAPI developer to provide documentation on all aspects of the BAPI. Application developers access the technical metadata and documentation of BAPIs from the Business Object Repository. Readers of BAPI documentation are often not R/3 experts or business application experts.

The best way to learn to create BAPIs and become good at it is, of course, to develop some. Try it, and hopefully you will find creating your own BAPIs challenging and rewarding.

Ralph Melone received a B.A. in Liberal Arts and an M.S. in Information Sciences from Villanova University. He also holds an M.B.A. in Marketing from Temple University. He joined SAP in March 1998 in the Curriculum Development Area. There, he is responsible for the integration technology curriculum, which includes BAPI, RFC, Internet, and object-oriented technologies. Ralph is also a Microsoft Certified Instructor, Engineer, and Solution Developer. Prior to joining SAP, he also had his own consulting firm, and has held management and IT positions with AT&T and ADP. Ralph can be reached at ralph.melone@sap.com.

