How to Create your Own SyncBo on MI 7.0 (from Definition to Java Programming)

Applies to:

Mobile Infrastructure 7.0, Mobile Asset Management 3.0 and Mobile Asset for Utilities 3.0.

For more information, visit the Mobile homepage.

Summary

The main purpose of this article is to explain step by step how to create a new Synchronization Object (SyncBO) in a Mobile Infrastructure 7.0 architecture. It comprises three main parts: ABAP part, MI server part and finally the Java part. After read this article, you would be able to create a brand new SyncBO from scratch and used it on a SAP's Mobile Java Application.

Author: Alvaro M. Guzmán A.

Company: SAP Multi Country Latin America MCLA (branch Venezuela)

Created on: 27 April 2010

Author Bio

I've been working in SAP LA since 2007 as a CRM consultant. My mayor skills are focused in web development areas like CRM IC, CRM Web UI, UCES, BSP, WS and Java Web components. Additionally I participate in Mobile Infrastructure projects as a developer.

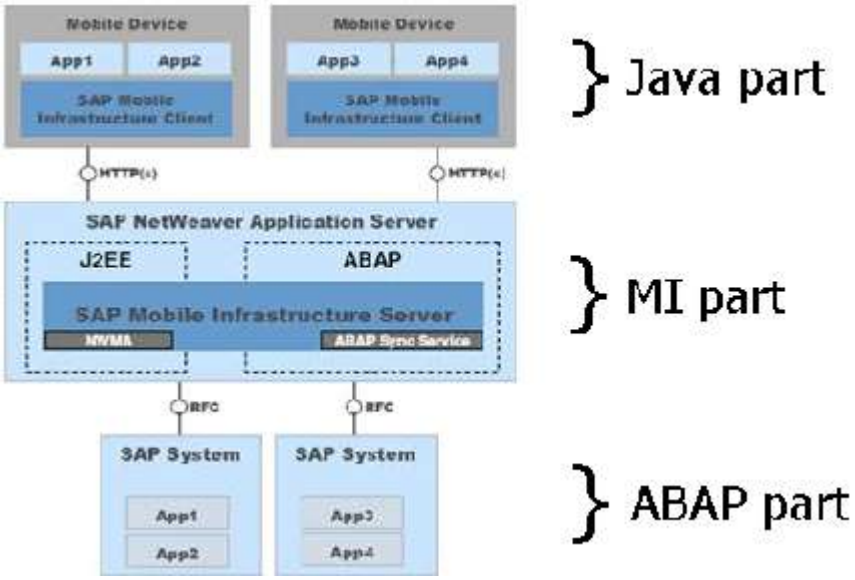
Table of Contents

Introduction

When we're working with SAP Mobile applications over Mobile Infrastructure 7.0 we always use Synchronization Business Objects (commonly known as SyncBOs) to exchange data with the backend. Most of the Mobile applications have their own SyncBOs who handle all the data they need to share. But what happened when you need to extend your model? For example if you need to add more information. The easiest approach for this is to extend your current SyncBOs by adding more fields or just use existing ones for additional purposes, for example separating different values with a pivot. This is a frequent approach and works in most of the cases.

But… what if we want a more elegant way to do this? What if we want to create our own SyncBOs?

Well, for those who are curious enough to do this kind of stuff this article intent to show you how to create a brand new SyncBO type S01 (2-Way Synchronization which is be the most complex). It comprises the ABAP part, the MI configuration part and finally the Java part… so you need a basic understanding of MI concepts as well as a basic development background (ABAP and Java) to understand it and take the most benefit of it.



There are several articles about this topic but my main idea was to put all together based on my experience so you'll see it covers the whole thing.

The Test Scenario

Imagine that you have a Z table in your backend system and you need the information store in that table in your handheld application. What do you do? First of all, you need to define your table's structures and proceeded to create the corresponding ABAP objects. Then we define our SyncBO in the MI server who is going to be responsible for synchronize the data between the backend and the mobile device and finally we update our mobile application so it can handle the new object.

Let's begin!

ABAP part

First of all we must create our table and its corresponding structures (in our example, we just have the Header structure (TOP), no items were modeled). Then we must build the corresponding ABAP objects in order to get all the registers of our Z table. For this, and keeping in mind that we finally are going to use a SyncBO (who from now on we are going to call ZNTC - Not Too Complex SyncBO) we need to create the corresponding set of BAPI Wrappers (ABAP Function Modules) who are in charge of send and retrieve from the backend system all the necessary data that our SyncBO is going to handle.

These BAPI wrappers need to be defining according to the following standard actions:

- Get List
- Get Detail
- Create (optional)
- Change (optional)
- Delete (optional)

These wrappers MUST fulfill the following rules in order to make everything work:

- Must be Remote Function Module enabled (RFC)
- The RETURN parameter (type BAPIRET2) must be defined as Export or Table parameter
- All parameters can make reference to one structure only
- “Changing” parameters are not allowed
- Use of Exceptions are not permitted
- All functions modules must belong to the same Function Group

As an example we are showing the GELT_LIST’s function module signature, named ZMOBFTEC_ZNTC_GETLIST, who has a parameter called NTC_LIST that correspond to the structure of the table itself (ZNTC_HEADER):

```
FUNCTION ZMOBFTEC_ZNTC_GETLIST.
*"----------------------------------------------
*"*"Local interface:
*"  TABLES
*"      NTC_LIST STRUCTURE  ZNTC_HEADER
*"      RETURN STRUCTURE  BAPIRET2
*"----------------------------------------------
```

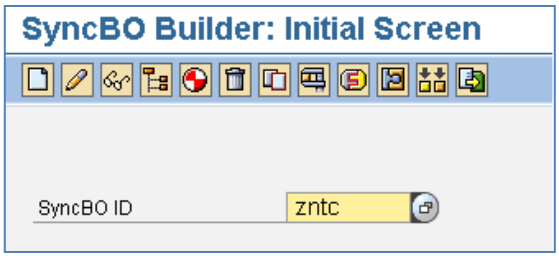
Note: You should be aware that the source code of this function module it is irrelevant for our example and it will depend on the business logic of the application. In our case we don’t have any IMPORT parameters so all the rows of the Z table will be retrieving using a simple SELECT function.

Once we create and implement our 5 function modules the ABAP part is finished and we can proceed with the definition of the ZNTC object at the MI server.

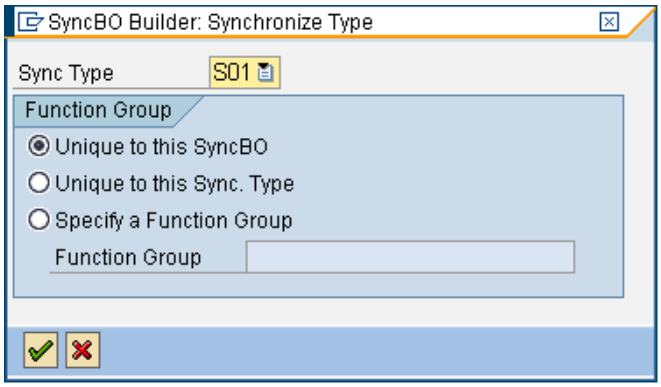
Mobile Infrastructure Server Part

In the MI part we must define our object in order to be used by the Mobile application. For this, logon into your MI server, execute TxCode merep_sbuilder (MI Builder) and choose create a new object option.

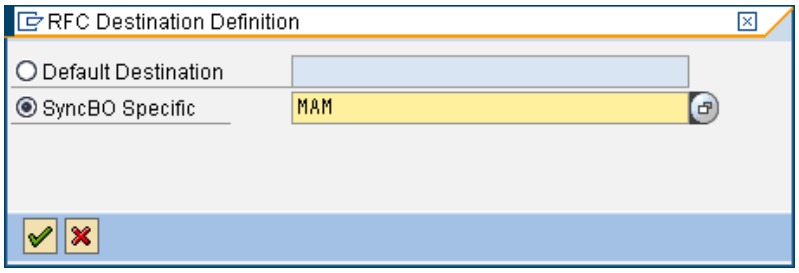
Choose the name of our new SyncBo, in this case zntc:



Select the Synchronization type, in this case should be S01 (2-Way Sync):

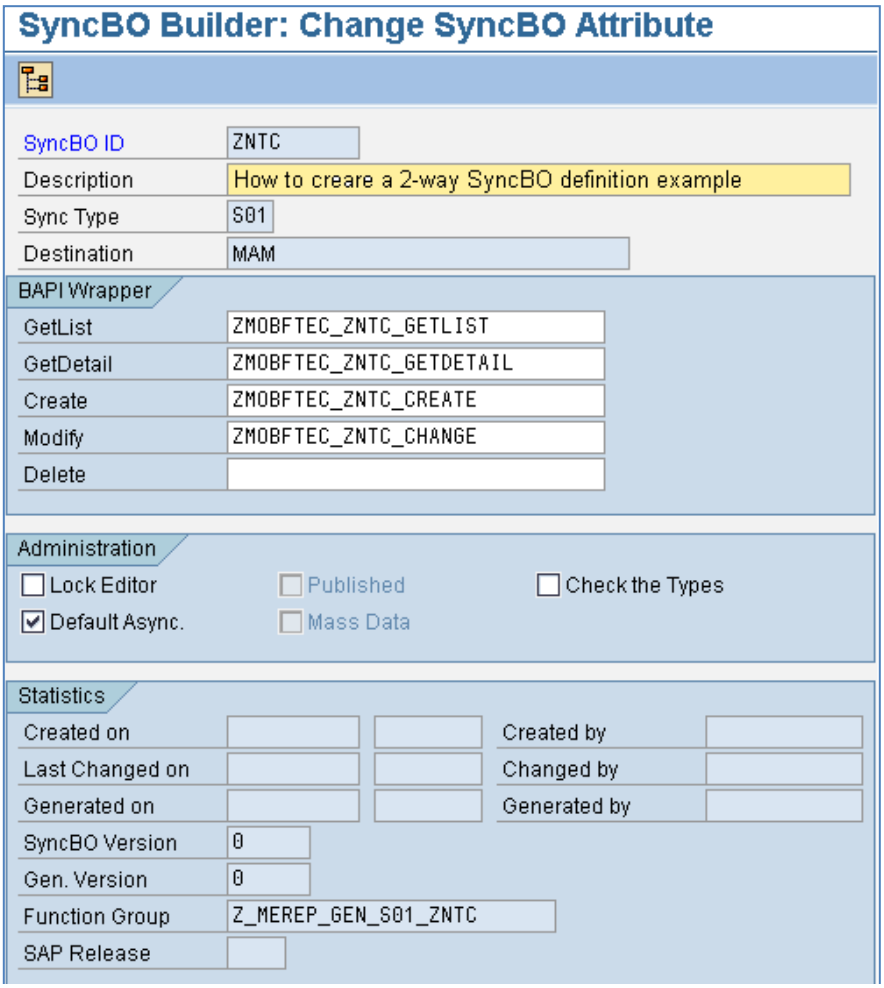


Choose the corresponding RFC destination where your function modules were created (see “ABAP part” section of this article). In our example the destination to be used is “MAM” because it was the name given to the destination since this SyncBO will be used in a SAP’s Mobile Asset Management application:



Note: The corresponding RFC destination must be maintained before performing this step.

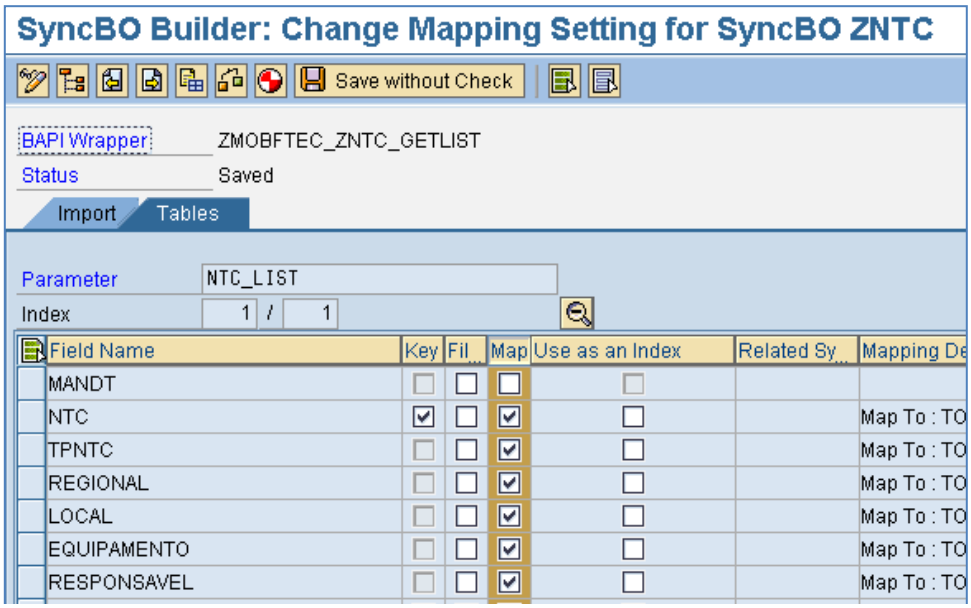
Choose the function modules previously created for the purpose of our example and also select the Function Group created:



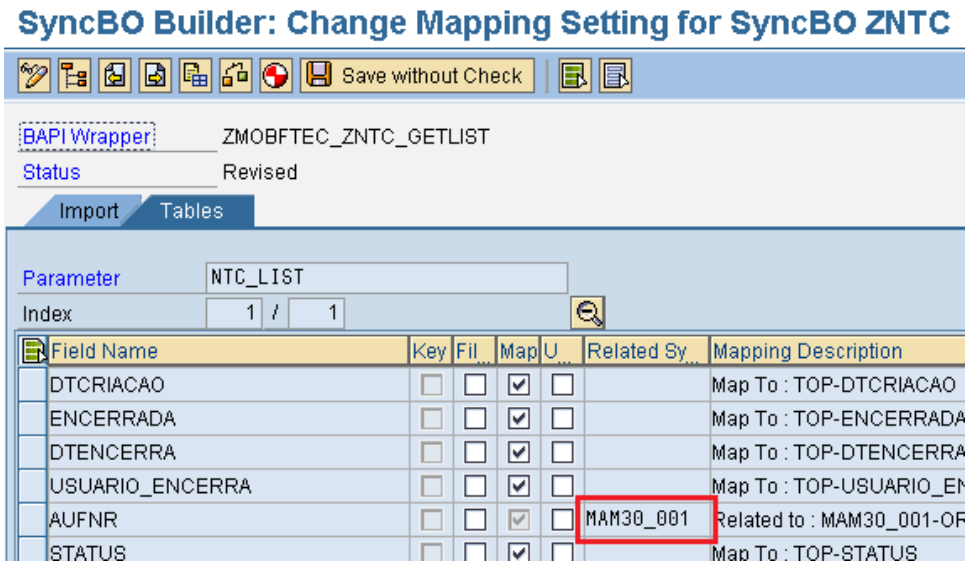
After this we're ready to make the mapping of the BAPI wrappers by choosing "Mapping Setting" button located on the top left of the screen above:

Here you can maintain the necessary parameter mapping. In this case the GETLIST method is displayed (as it is the first one) on the list.

Now select the "Tables" tab where you can find the structure of the parameter NTC_LIST, which is the same of the BAPI wrapper showed before:



As you can see we select a key field (in this case NTC) and then all the fields you want to be visible to the Mobile Clients. For this we just use the “Map” column. Additionally you can specify default values for a field, filtering and link to another SyncBO as you can see in the following screen where we create the binding of one of our fields to the SyncBO MAM30_001 (standard for MAM application):

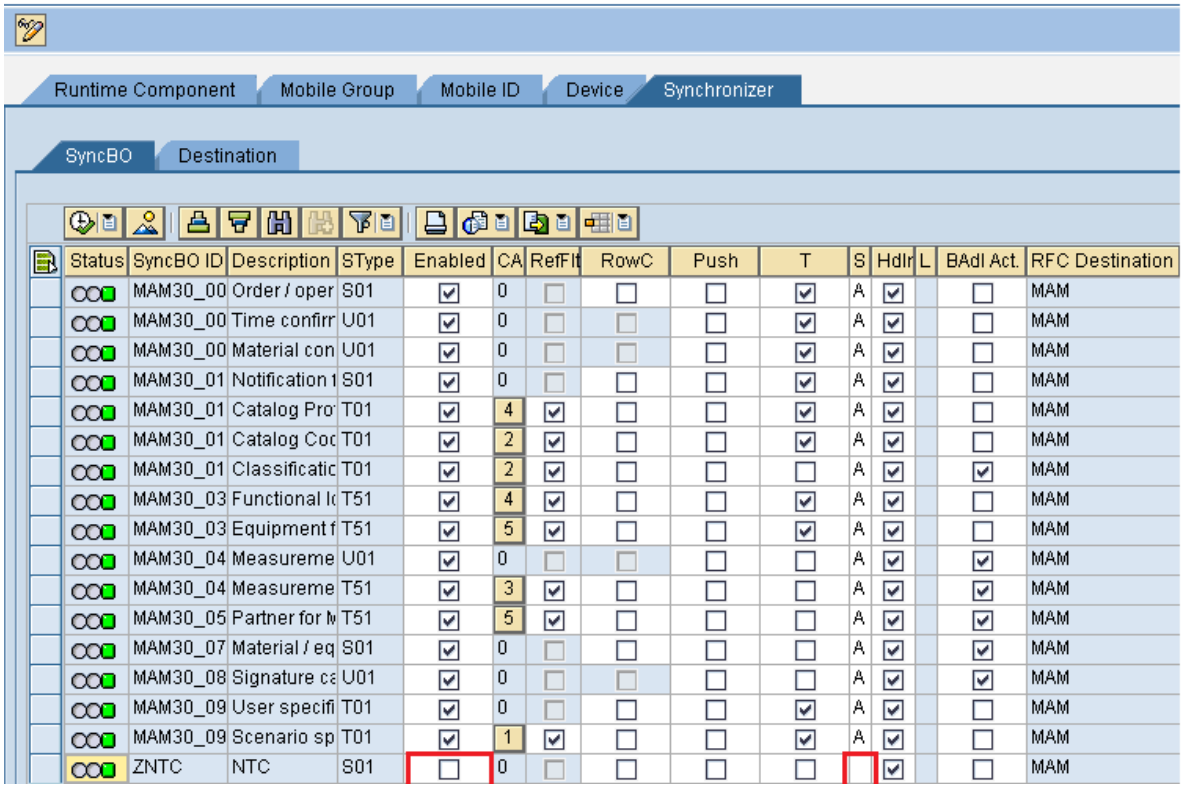


Depending on the SyncBO type the same procedure should be performing for the other BAPI wrappers (use the buttons  to navigate). In our case all the 5 should check and configured.

After all the changes were made, we must select the “Generate/Activate” button  in order to generate all the objects and allowed to the MI server to build the “main” Function Module who contain the logic we just create. If you want to check it the name of this object should be something like this: Z_MEREP_GEN_F01_____ZNTC

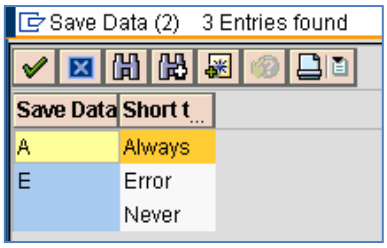
Next step is to update the synchronizer. For this execute the TxCode merep_pd in Edit mode, select Synchronizer tab and then SyncBO as shown:

Profile Dialog: Change Synchronizer



Status	SyncBO ID	Description	SType	Enabled	CA	RefFlt	RowC	Push	T	S	HdlR	L	BAdI Act.	RFC Destination
	MAM30_00	Order / oper	S01	☑	0	☐	☐	☐	☑	A	☑		☐	MAM
	MAM30_00	Time confirr	U01	☑	0	☐	☐	☐	☑	A	☑		☐	MAM
	MAM30_00	Material con	U01	☑	0	☐	☐	☐	☑	A	☑		☐	MAM
	MAM30_01	Notification t	S01	☑	0	☐	☐	☐	☑	A	☑		☐	MAM
	MAM30_01	Catalog Pro	T01	☑	4	☑	☐	☐	☑	A	☑		☐	MAM
	MAM30_01	Catalog Coc	T01	☑	2	☑	☐	☐	☑	A	☑		☐	MAM
	MAM30_01	Classificatic	T01	☑	2	☑	☐	☐	☐	A	☑		☑	MAM
	MAM30_03	Functional l	T51	☑	4	☑	☐	☐	☑	A	☑		☐	MAM
	MAM30_03	Equipment f	T51	☑	5	☑	☐	☐	☑	A	☑		☐	MAM
	MAM30_04	Measureme	U01	☑	0	☐	☐	☐	☐	A	☑		☑	MAM
	MAM30_04	Measureme	T51	☑	3	☑	☐	☐	☐	A	☑		☑	MAM
	MAM30_05	Partner for M	T51	☑	5	☑	☐	☐	☐	A	☑		☑	MAM
	MAM30_07	Material / eq	S01	☑	0	☐	☐	☐	☐	A	☑		☑	MAM
	MAM30_08	Signature ca	U01	☑	0	☐	☐	☐	☐	A	☑		☑	MAM
	MAM30_09	User specifi	T01	☑	0	☐	☐	☐	☑	A	☑		☐	MAM
	MAM30_09	Scenario sp	T01	☑	1	☑	☐	☐	☑	A	☑		☐	MAM
	ZNTC	NTC	S01	☐	0	☐	☐	☐	☐		☑		☐	MAM

Be sure of select the “Enabled” check box and select the corresponding type for saving (in our case we select A for “always”). If you don’t do this, your SyncBO won’t be available in your application:



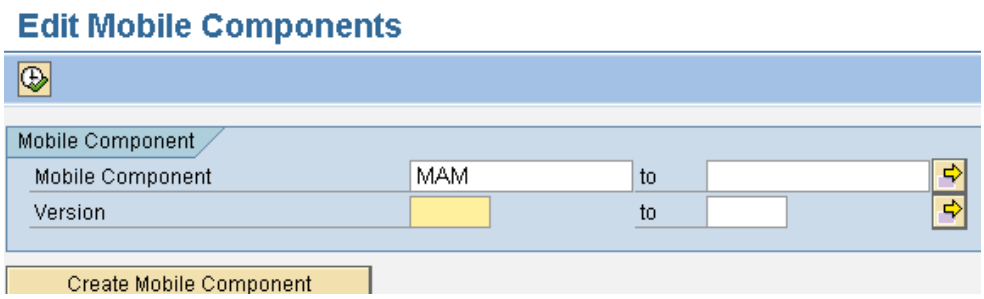
Save your changes. Now our ZNTC SyncBO is fully configured!

But… what about the Java application? Well, of course we need to change it but not yet. As our object’s configuration is ready we need to let the Java application know about our changes since still resides “only” on the MI Server.

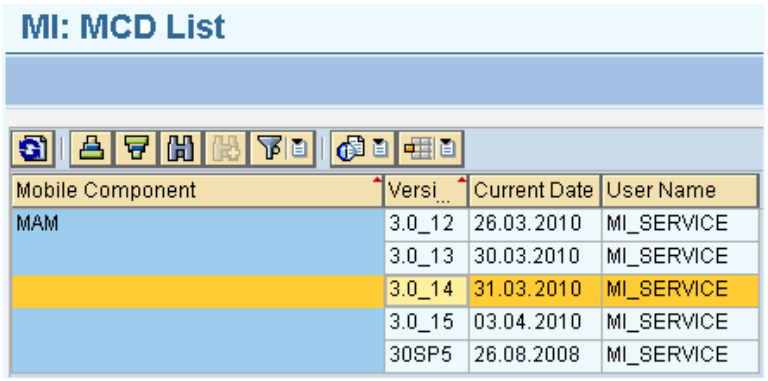
For this we need to export our Metadata which is an XML file with all the SyncBO’s definition that resides in the MI Server, in this case it will be include our ZNTC object.

Download Metadata

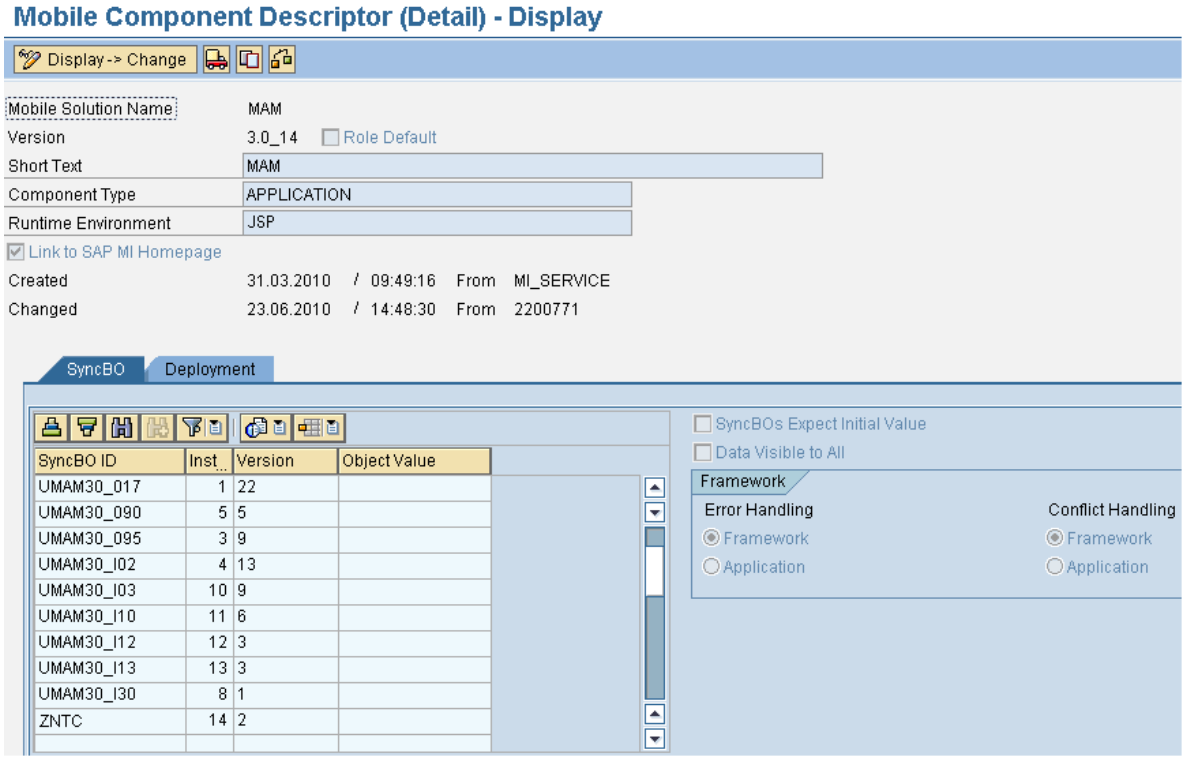
First of all we need to know the version of the component we’re currently using on the handheld and if necessary (recommended) create a new one. For this, we can use TxCode mi_mcd which update/create the Mobile Component Descriptor (MCD) of our application in the MI server, in this case “MAM” as shown:



After select the corresponding component the list of available MCD is show:

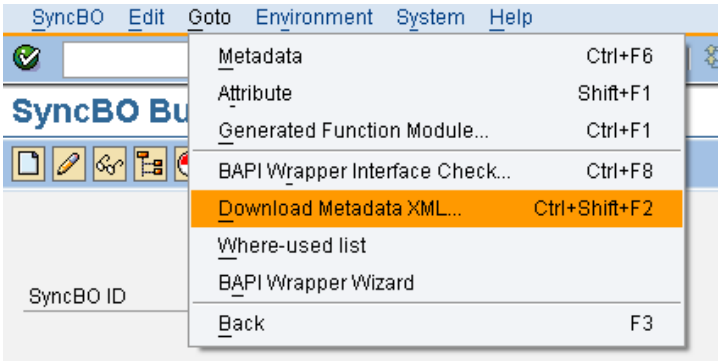


When we create/update a version we need to define the sequence in which the SyncBOs should be installed on the device (Mobile Infrastructure 7.0):



As you can notice, our object ZNTC resides at the bottom of the list and the installation order is 14 (last one since it depends on other objects).

Once this task is done and we can actually see our object in the list we are ready to download the Metadata file. For this, we need execute TxCode merep_sbuilder, select the option “Download Metadata XML” from the menu and save the file into our machine. This file is going to be needed later in the Java integration:



This file should have the new definition of the ZNTC object as well as all the other definitions:

```
<?xml version="1.0" encoding="utf-8" ?>
- <MeRepApplication schemaVersion="1.1" id="MAM" version="3.0_14">
    <Property name="CLIENT.BUILDNUMBER">201003301</Property>
    <Property name="C_APPLRESOLVE" />
    <Property name="DATA_VISIBLE_SHARED" />
    <Property name="E_APPLRESOLVE" />
    <Property name="HOMEPAGE.INVISIBLE" />
    <Property name="INITVALUE" />
    <Property name="RUNTIME">JSP</Property>
    <Property name="TYPE">APPLICATION</Property>
  + <SyncBO id="UMAM30_017" version="22" type="timedTwoWay" allowCreate="false" a
  + <SyncBO id="UMAM30_016" version="4" type="timedTwoWay" allowCreate="false" all
  + <SyncBO id="UMAM30_095" version="9" type="timedTwoWay" allowCreate="false" all
  + <SyncBO id="UMAM30_I02" version="13" type="twoWay" allowCreate="false" allowMo
  + <SyncBO id="UMAM30_090" version="5" type="timedTwoWay" allowCreate="false" all
  + <SyncBO id="UMAM30_001" version="15" type="twoWay" allowCreate="true" allowMo
  + <SyncBO id="UMAM30_006" version="6" type="upload" allowCreate="true" allowModify
  + <SyncBO id="UMAM30_I30" version="1" type="timedTwoWay" allowCreate="false" allo
  + <SyncBO id="UMAM30_010" version="15" type="twoWay" allowCreate="true" allowMo
  + <SyncBO id="UMAM30_I03" version="9" type="twoWay" allowCreate="false" allowModi
  + <SyncBO id="UMAM30_I10" version="6" type="upload" allowCreate="true" allowModify
  + <SyncBO id="UMAM30_I12" version="3" type="upload" allowCreate="true" allowModify
  + <SyncBO id="UMAM30_I13" version="3" type="upload" allowCreate="true" allowModify
  - <SyncBO id="ZNTC" version="2" type="twoWay" allowCreate="true" allowModify="false"
    - <TopStructure name="TOP">
      - <Field name="SYNC_KEY" type="N" length="10" decimalLength="0" signed="false"
          <Input type="create">false</Input>
          <Input type="modify">false</Input>
        </Field>
```

Once we have in our hands this file, we’re ready to perform the changes in our Java Application and start using our new object.

Java Part

What we got so far? We have our ZNTC object BAPI wrappers in the backend (ABAP part) and the MI server is aware about its existence and is able to communicate with it. So, we only need to update the mobile application to complete our example.

Before we get into the source code we need to substitute the current meRepMeta.xml file located in the java application root directory by the new file we just got from the MI server. With this action we're updating the metadata of all our SyncBOs in the application, which includes new additions and structure changes.

Note: After this action it is necessary to build a new .war file with the application and deploy it in the MI Server!!! Without this we won't be able see the changes in the Client.

Once this is done, perform a Client's reset and then synchronize with the MI server in order to get the new SyncBo(s) descriptor(s). This step must be completed before continue with further steps.

Note: Regarding the java application source code changes, we can make this work in many ways; as the main purpose of this article is let you know how to implement a new SyncBO I'm proposing an easy way that I found more didactic; it is only one approach.

So, I divided this part of my article in 4 steps that covers the minimum coding that you should build in order to have access to the ZNTC object.

Note: For any Java development activities we are using NetWeaver Developer Studio 7.0.

1. Create BO JavaBean interface and implementation

This interface must extend from interface BusinessObject and implement all the getters and setters for all the required attributes in the object:

```
public interface ZMamZNTC extends BusinessObject
```

Note: Ensure to add the method getField(String fieldName) because it is going to be used later.

The implementation class must extend from AbstractBO and implement WritableBO as well as the new interface:

```
public class ZMamZNTCImpl extends AbstractBO implements ZMamZNTC,WritableBO
```

The method *getField* could be implemented as follows (sample code with two hypothetical attributes, "xyz" and "abc"):

```
public Object getField(String fieldName)
{
	Object obj = null;

	if (fieldName.equalsIgnoreCase("xyz"))
		return this.getXyz();
	else if (fieldName.equalsIgnoreCase("abc"))
		return this.getAbc();

	return obj;
}
```

Additionally the method *setField* could be implemented as follows (notice the different types we're using):

```
public void setFieldValue(String fieldName,String fieldValue)
{
	try
	{
		BasisFieldType bft = getFieldDescriptor(fieldName).getFieldType();
		String type = bft.getBackendRepresentation();

		if (!type.equals("P") || !type.equals("D") || !type.equals("T") )

row.modifyFieldValue(getFieldDescriptor(fieldName),fieldValue);
		else if (type.equals("P"))
		{
			BigDecimal bd = new BigDecimal(fieldValue);
			row.modifyFieldValue(getFieldDescriptor(fieldName),bd);
		}
		else if (type.equals("D"))
		{
			Calendar cal = DateTimeUtils.formatDate2(fieldValue);
			Date date = new Date(cal.getTimeInMillis());
			row.modifyFieldValue(getFieldDescriptor(fieldName),date);
		}
		else if (type.equals("T"))
		{
			Calendar cal = DateTimeUtils.formatTime(fieldValue);
			Time time = new Time(cal.getTimeInMillis());
			row.modifyFieldValue(getFieldDescriptor(fieldName),time);
		}
	}
	catch (Exception e)
	{
		e.printStackTrace();
	}
}
```

Note: Be careful with the type of the ABAP attribute.

The method *write* could be implemented like this:

```
public void write() throws BOException
{
	try
	{
		row.getSyncBo().modifyRow(row);
	}
	catch (Exception e)
	{
		throw new BOException("Failed writing Row: " + e.toString(),e);
	}
}
```

2. Create the SyncBO Manager

Once we have the SyncBO JavaBean ready it is necessary to build the BOManager interface and implementation classes. Interface definition:

```
public interface ZMamZNTCManager
```

At least we need to declare the following methods in the interface:

```
   public ZMamZNTC createZMamZNTC() throws BOException;

   public BOList getZMamZNTCs() throws BOException;

   public void insertZMamZNTC (ZMamZNTC object) throws BOException;

   public void updateZMamZNTC(ZMamZNTC parent) throws BOException;

   public void deleteZMamZNTC(String key) throws BOException;

   public BOList getZMamZNTCs(Condition condition, SortOrder order, int startIndex,
int maxResults) throws BOException;

   public ZMamZNTC lookupZMamZNTC(String key) throws BOException;
```

The implementation class must extend from AbstractBOManager and implement RowBOManager as well as the new interface:

```
public class ZMamZNTCManagerImpl extends AbstractBOManager implements
RowBOManager,ZMamZNTCManager
```

Ensure to add these additional methods in this class:

```
   public ZMamZNTC createZMamZNTC() throws BOException

   private final ZMamZNTC createCheckZMamZNTC() throws BOException
   {
          ZMamZNTC object = (ZMamZNTC)naming.create(ZMamZNTC.class.getName());

          return  object;
   }
```

3. Create the SyncBO Query Builder (optional)

Optionally we can create a Query Builder interface and implementation classes for database integration. The interface must extend from interface QueryBuilder:

```
public interface ZMamZNTCQueryBuilder extends QueryBuilder
```

The implementation class must extend from AbstractQueryBuilder (who implements most of the methods) and implement the new interface:

```
public class ZMamZNTCQueryBuilderImpl extends AbstractQueryBuilder implements
ZMamZNTCQueryBuilder
```

4. Configuration Interface/Implementation

After implement all the classes we must include those changes in the Zcore.configure file (who is in charge of all the java changes in our MAM application) indicating every pair of implementation or substitutions as follows:

```
com.test.sdn.mam.bo.ZMamZNTCManager=com.test.sdn.mam.bo.impl.ZMamZNTCManagerImpl
com.test.sdn.mam.bo.ZMamZNTC=com.test.sdn.mam.bo.impl.ZMamZNTCImpl
com.test.sdn.mam.bo.query.ZMamZNTCQueryBuilder=com.test.sdn.mam.bo.query.impl.ZMamZNT
CQueryBuilderImpl
```

Finally we can instantiate our class like this:

```
ZMamZNTC object = (ZMamZNTC)naming.create(ZMamZNTC.class.getName());
```

Once we have our instance created we can use all the methods we define.

Conclusion

At the end of this article, readers are ready to create their own Synchronization Objects from scratch, starting with the definition at the backend passing through the configuration in the MI Server and ending with changes in the Java application.

Final step would be used this object (ZNTC) in your application according to the business rules of your development.

Copyright