NDL

Updating MX Apps

to use new features from MX v10

Document Number: UpApps4MX10DA108-002

Issue 4   Effective: 28th September 2023

Contents

Introduction 1

Synchronise() 1

Notifications 2

Step-by-step... 2

- App Studio 2
- OnAsyncCompletion 6
- IsDataSourceLocked 7
- Using Custom Tables 8
  - Creating a New Record 9
  - Modifying an Existing Record 10
- AddSystemNotification 15
- Error Handling 16

Summary 18

Introduction

This document demonstrates how existing MX apps, created using a version of Digitise Apps' predecessor MX prior to MX v10, can be updated to take advantage of newer features introduced in MX v10, primarily those related to the asynchronous communication functionality, after upgrading to Digitise Apps. The new features discussed in this document were introduced in MX v10 and continue to be supported in Digitise Apps.

This guide assumes a basic familiarity with MX/Digitise Apps and will not provide an in-depth discussion as to its usage, although new MX v10 features will be explained in some detail.

Synchronise()

Existing MX apps created using awi MX v8 or earlier rely on using **SyncDataSource()** and **LoadDataSource()** to communicate with the App Server. These two methods use synchronous communication to send data to and from the server, leaving the user unable to use their application while communication is in progress. Dependent on the amount of data being sent or received, this can quickly add up to a not-insignificant amount of time spent waiting for a given transfer to complete. These methods also often require the user to implement complicated error handling using **SetDataErrorOn()/Off()** and the like.

The **Synchronise()** Method was introduced in MX v10 and allows an application to communicate asynchronously (i.e. in the background) with the server and both send and receive data within a single method call. Thus **Synchronise()** can be used in place of both **SyncDataSource()** and **LoadDataSource()** allowing a user to send and receive data whilst still having access to the full range of functionality within their app.

Note that you can also use **Synchronise()** in synchronous mode which will stop your app whilst the data transfer is in progress, just like **SyncDataSource()** and **LoadDataSource()**, if you prefer.

Notifications

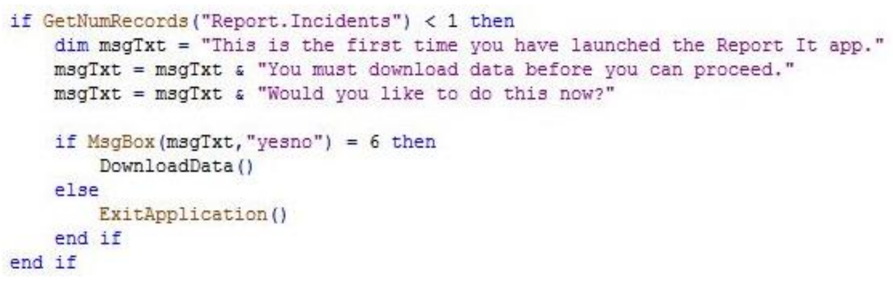
The **AddSystemNotification()** method was extended in MX v10 to work with all key platforms – Android, iOS , Windows Desktop and Windows Universal. It can be used completely independently of **Synchronise()** – for example, to alert a community nurse that he/she has an appointment due with a patient – but pairing it with **Synchronise()** greatly enhances the asynchronous communication user experience.

Step-by-step…

The remainder of this guide will take you through the process of converting an existing app to include MX v10 features. We will use the NDL demonstration app *Report It* to illustrate implementing the new features.

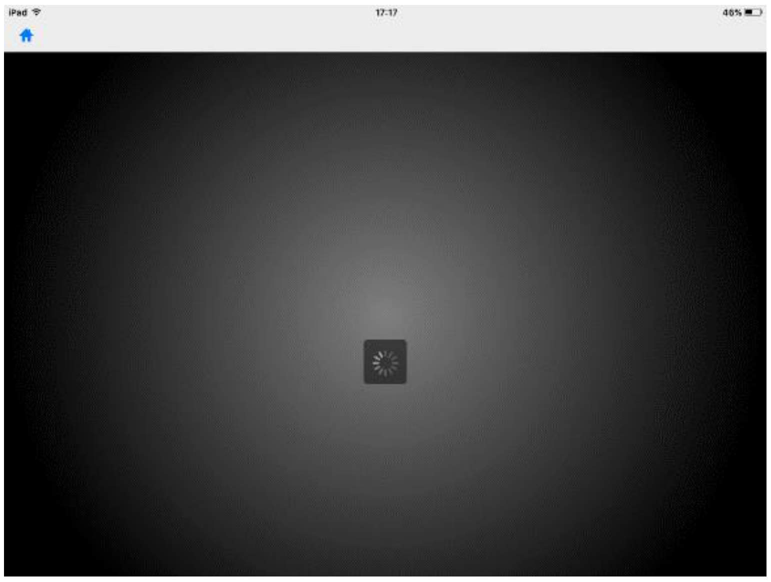
App Studio

When a user first downloads and opens the *Report It* app on a device, the app will check whether or not there is data on the device and if not, prompt the user to download data. This is a fairly common feature in MX/Digitise apps, particularly those that require a user to log in and have their username and password verified against a database before being able to use the app.

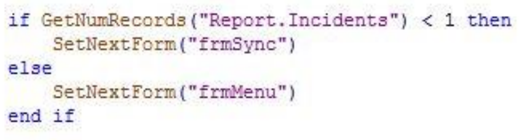


```
if GetNumRecords("Report.Incidents") < 1 then
    dim msgTxt = "This is the first time you have launched the Report It app."
    msgTxt = msgTxt & "You must download data before you can proceed."
    msgTxt = msgTxt & "Would you like to do this now?"

    if MsgBox(msgTxt,"yesno") = 6 then
        DownloadData()
    else
        ExitApplication()
    end if
end if
```

In our *Report It* app, this check is performed in the **Application.OnLoad** script – i.e. before we've actually loaded a form from our application. While this is perfectly sufficient from a purely functional standpoint, staring at a blank screen waiting for data to download somewhat diminishes the user experience:

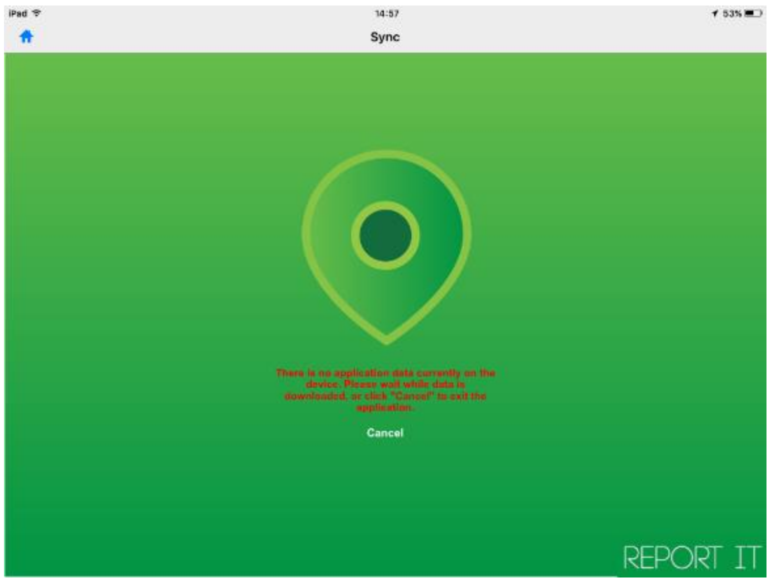


To avoid this, let's add in a "sync" form at the beginning of our *Report It* app. We can modify our **Application.OnLoad** script so that while it still checks for data present on the device, it will send the user to either our new sync form or our menu form, dependent on the outcome:



```
if GetNumRecords("Report.Incidents") < 1 then
    SetNextForm("frmSync")
else
    SetNextForm("frmMenu")
end if
```

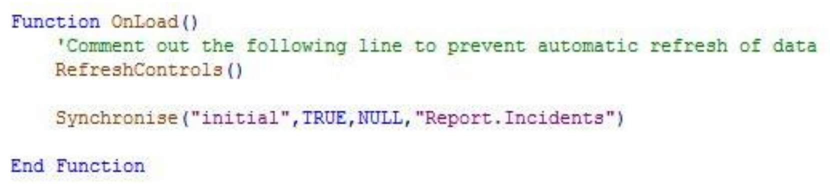
Our new sync form could look something like this:



There is no application data currently on the device. Please wait while data is downloaded, or click "Cancel" to exit the application.

As you can see, the form contains all of the information previously conveyed to the user through the use of a message box, but has a much more user-friendly look and feel.

Our sync form is where we will encounter the first of the new features introduced in MX v10: **Synchronise()**. We are going to incorporate **Synchronise()** into the **OnLoad** script of our sync form:



```
Function OnLoad()
    'Comment out the following line to prevent automatic refresh of data
    RefreshControls()

    Synchronise("initial",TRUE,NULL,"Report.Incidents")

End Function
```

As you can see here, our **Synchronise()** method takes four arguments. The first is what is known as the "Tag" parameter, and acts as a unique identifier for this specific **Synchronise()** transaction. You could have an app that calls **Synchronise()** in multiple places and requires a different outcome upon the completion of each of these transactions. The nature of MX v10's asynchronous communication is such that the application has no way of knowing which form the user might be on when the transaction completes so thus it is necessary to be able to identify the individual transactions using the "Tag". We've named our **Synchronise()** transaction "initial" – it is, after all, the first instance of **Synchronise()** in our application.

Next on the list in any **Synchronise()** transaction is the "IsAsync" parameter. This is a Boolean specifying whether or not we want this particular transaction to be carried out asynchronously.

Here's where it gets a little tricky – in this instance, we want the user to be held on the sync form until the transaction has finished. However, instead of setting the IsAsync parameter to "FALSE" and forcing our user to wait until the transaction completes, we're instead going to tell our app to carry out the data transaction asynchronously by using "TRUE" as our argument, and are instead going to "trap" the user on our form by not including any navigation buttons or other ways to move off the form. This may seem somewhat counter-intuitive, but all will be explained later.

The next two parameters are our data source parameters. When using **Synchronise()**, we can choose to just upload data, download data or both upload and download data in the one transaction. The first of these two parameters allows you to specify data sources to be uploaded to the server and the second, data sources to be downloaded to the device. In this case, as we've already established that the user has no data on their device at this point, we can take it as a given that there aren't going to be records to upload so we will specify the upload parameter as NULL, which means no data will be uploaded. Then for the download parameter we will specify the data sources we want to download to the device. Our *Report It* app only uses one table, so we're going to specify that as the argument: "**Report.Incidents**".

Note that, where required, multiple data sources can be specified in the data source parameters either explicitly using a comma separated list, such as "datasource1, datasource2, datasource3, …", or by specifying an empty string, **""**, in which case MX will use the **Sync Direction** property to determine the default data sources to include for the parameter containing the empty string. Each data source table has its own **Sync Direction** property which specifies whether the table should be included in a synchronisation if data sources are not explicitly specified. To exclude all data sources from a particular synchronisation you can specify the relevant data source parameter as NULL, as we have done here for the upload parameter.

We've now set up our very first **Synchronise()** transaction. So, what was the point of having our transaction execute asynchronously?

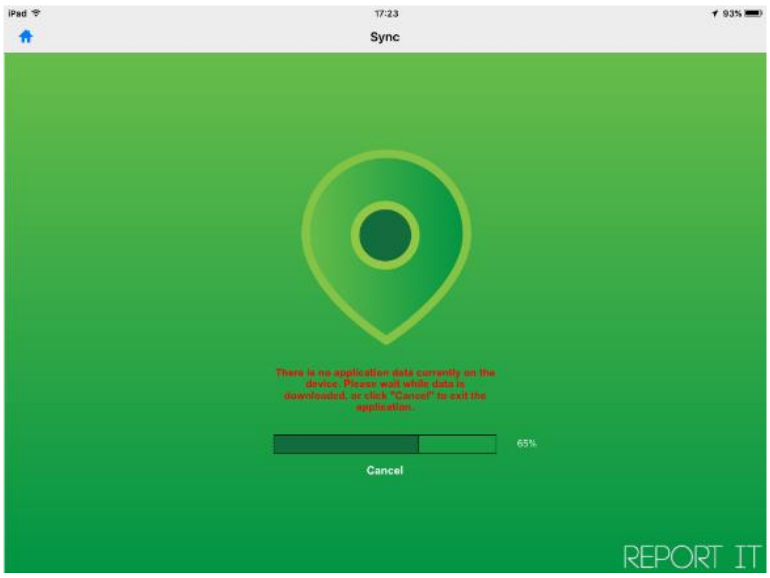
If we had set the IsAsync parameter to "FALSE", our **Synchronise()** transaction would essentially have mimicked a **LoadDataSource()** operation and would have forced the user to wait until the transaction was completed. A new feature of MX v10 is that the user has the option to cancel asynchronous transactions – by setting the IsAsync parameter to "TRUE", we are going to give the user the option to cancel the sync operation and exit the application by clicking the "Cancel" text on the form.

We are going to do this by using the new **CancelTransaction()** method and the unique tag pertaining to the instance of **Synchronise()** that we're cancelling. In this case, we want the app to cancel our "initial" **Synchronise()** transaction and then exit the application:

```
Function OnClick()

    CancelTransaction("initial")
    ExitApplication()

End Function
```

Leaving our "initial" instance of **Synchronise()** as an asynchronous transaction also frees up our application to carry out other tasks while the data transfer is in progress. Let's take advantage of this (and another new MX v10 method) and implement our own progress bar to inform the user of how their **Synchronise()** transaction is progressing.

Let's add a timer and three static text controls to our sync form. We're going to use two of the static controls to form our progress bar, while the third is going to be used to display the percentage completion of our **Synchronise()** transaction:



There is no application data currently on the device. Please wait while data is downloaded, or click "Cancel" to exit the application.

We need to set the interval for the timer we've just placed on our form. Let's do this by using the **SetTimerInterval()** method in our form **OnLoad** script. We need to make sure that our timer is being triggered fairly frequently or there's no point in including it – a timer that only has chance to update once or twice before our **Synchronise()** transaction terminates isn't going to be much use. Let's use an interval of 500 milliseconds.

We now need to include some code in our **OnTimer** event to update our progress bar.

We can use the new MX v10 method **GetTransactionProgress()** with our "initial" tag. When called, this method will return the percentage completion of the Synchronise transaction with the tag "initial". We can then use the result of our **GetTransactionProgress()** method and the **.width** operator on our two static controls to update the size of our progress bar. Finally, we can set our third static control to reflect the obtained percentage value.

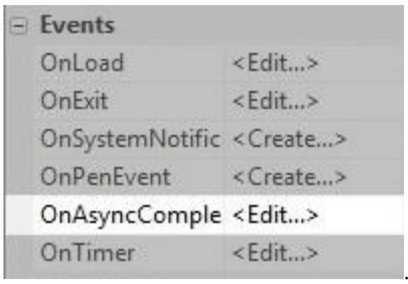
```
Function OnTimer()

    dim progress = GetTransactionProgress("initial")
    imgSProgress.width = progress * (imgSProgressBar.width / 100)
    SetControlValue("txtSPercent",progress & "%")

End Function
```

If we wanted to pursue this further, we could use the **GetTransactionState()** method to display more information to the user. Similar to **GetTransactionProgress()**, this method takes a tag pertaining to a specific **Synchronise()** transaction and returns its current state – so sending data, waiting, or receiving data, as examples. This provides more information to the user than is strictly necessary in the case of our *Report It* app, so we're not going to include it here.

OnAsyncCompletion

Let's assume the user decides not to cancel the **Synchronise()** transaction. As previously stated, we've essentially "trapped" the user on the sync form by omitting any navigation buttons. How, then, does the user access the rest of the content in the app?

Located in the Events section of the Application Properties, there are several new events that have been introduced in MX v10:



Events	
OnLoad	<Edit...>
OnExit	<Edit...>
OnSystemNotific	<Create...>
OnPenEvent	<Create...>
OnAsyncComple	<Edit...>
OnTimer	<Edit...>

.

We're going to focus on one of them: **OnAsyncCompletion**.

OnAsyncCompletion (as the name may suggest) is an application-level event triggered when an asynchronous transaction completes.

```
Function OnAsyncCompletion(Tag, Success)

    if Tag = "initial" then
        SetNextForm("frmMenu")
    end if

End Function
```

In our **OnAsyncCompletion** script, we are going to tell the application to take the user to the menu screen once the asynchronous transaction completes. We could just simply use the **SetNextForm()** method to accomplish this but we're planning on adding more **Synchronise()** functionality into our app later, so we'll use an **if** statement so the change of form only happens when our "initial" synchronisation completes.

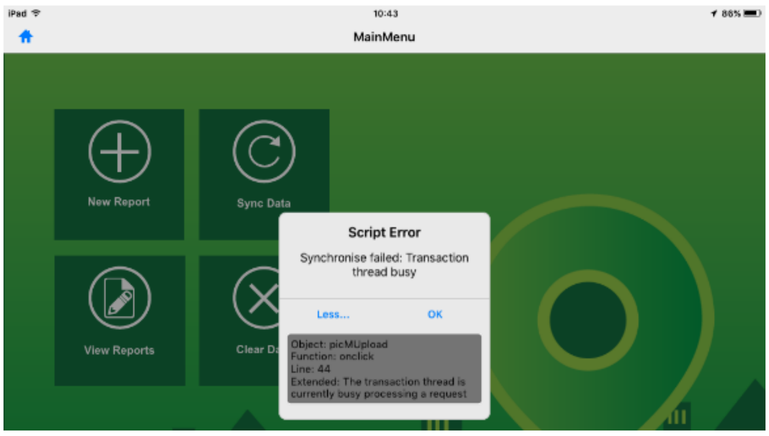
IsDataSourceLocked

So, we've added in a "sync screen" to our app, we've successfully implemented a Synchronise transaction and corresponding action in the **OnAsyncCompletion** event, and our user can now progress to the menu form of our *Report It* app. What comes next?

Well, our user will need to be able to send and receive data to and from the remote database, so let's go to the menu form and add in another **Synchronise()** command in the OnClick script of our "Sync Data" button.

This leaves us with an interesting problem. When a **Synchronise()** transaction is running asynchronously, the data source it uses is essentially "locked for editing". So what happens if the user clicks the "Sync Data" button, initialises a **Synchronise()** transaction, then clicks the "Sync" button again?

Well, an app can only process one **Synchronise()** transaction at once and so it gives an error:



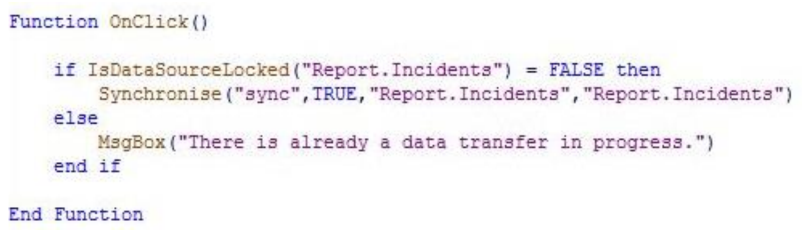
Script Error

Synchronise failed: Transaction thread busy

```
Object: picMUpload
Function: onclick
Line: 44
Extended: The transaction thread is currently busy processing a request
```

Fortunately, there is a way to deal with this: **IsDataSourceLocked()**.

IsDataSourceLocked() is another method introduced in MX v10, and (again, as the name suggests) checks to see whether or not a data source has been locked by a **Synchronise()** transaction. Let's add it into the **OnClick** script of our "Sync Data" button:



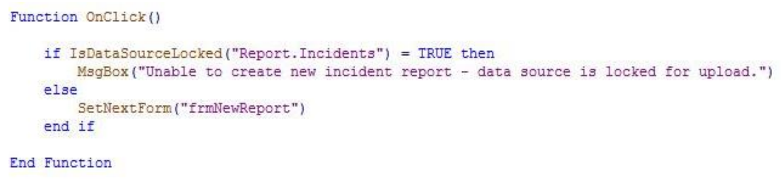
```
Function OnClick()

    if IsDataSourceLocked("Report.Incidents") = FALSE then
        Synchronise("sync",TRUE,"Report.Incidents","Report.Incidents")
    else
        MsgBox("There is already a data transfer in progress.")
    end if

End Function
```

Here, we've chosen just to add a message box informing the user that a **Synchronise()** is already in progress if our data source is locked. We could also use a combination of **IsDataSourceLocked()** and **CancelTransaction()** to cancel the pre-existing **Synchronise()** transaction and instigate a new one, if required.

So, we've now fixed the errors that we encountered if we tried to initialise multiple **Synchronise()** transactions. This has brought another issue to light though – if our data source is locked whilst a **Synchronise()** transaction is in progress, this means our user can't write data to it. So how can we allow them to continue to use the app while a data transfer is in progress?

Using Custom Tables

There are two ways to get round the locked data source problem. The first is quite simple – we can add in another **IsDataSourceLocked()** check in the **OnClick** script of our "New Report" button:



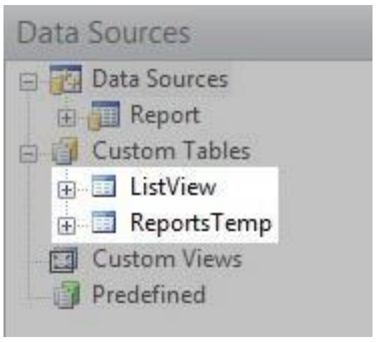
```
Function OnClick()

    if IsDataSourceLocked("Report.Incidents") = TRUE then
        MsgBox("Unable to create new incident report - data source is locked for upload.")
    else
        SetNextForm("frmNewReport")
    end if

End Function
```

While this is the simplest way to circumvent the problem, it has some fairly obvious limitations – namely, why allow asynchronous communication if the user can't fully use the app during data transfer? We could improve upon it by using a "yesno" message box and giving the user the option to cancel the transaction, but it still leaves a lot to be desired in terms of good user experience.

The second method is slightly more complicated but will mean the user is able to access all of the functionality in their app while a data transfer is in progress. It involves using Custom Tables.

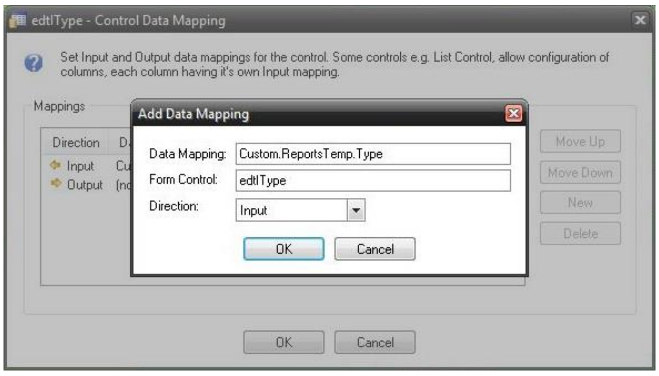
Creating a New Record

Let's create a custom table from our data source table. In the case of our *Report It* app, we're going to create our "**Report.Incidents**" table as a custom table and call it "**Custom.ReportsTemp**". Our **ReportsTemp** table should be identical to our Incidents table, in that all fields should be the same, with all the same properties. We're also going to create another identical table called "**Custom.ListView**":



When our user creates a new record in the app, instead of having the app create that record in our **Incidents** table, we're going to have it use our **ReportsTemp** custom table.

In order to implement this, we need to change all the input/output mappings on our "New Report" and "View Reports" forms from **Report.Incidents** to **Custom.ReportsTemp** – let's do this now.

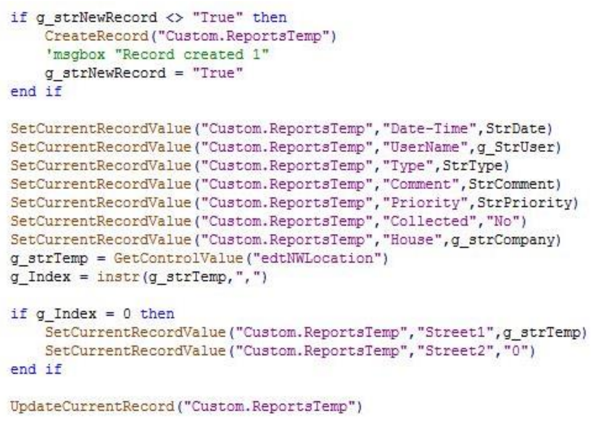


Set Input and Output data mappings for the control. Some controls e.g. List Control, allow configuration of columns, each column having it's own Input mapping.

Data Mapping: Custom.ReportsTemp.Type
Form Control: edtlType
Direction: Input

We're also going to change the input mappings of all the controls contained within our "tmpCollect" template from **Report.Incidents** to **Custom.ListView**.

We also need to modify the code in the **OnLoad** script of our "New Report" form so that on any occasions where data was being written programmatically to our **Report.Incidents** table, that data is instead now directed to our **Custom.ReportsTemp** table.



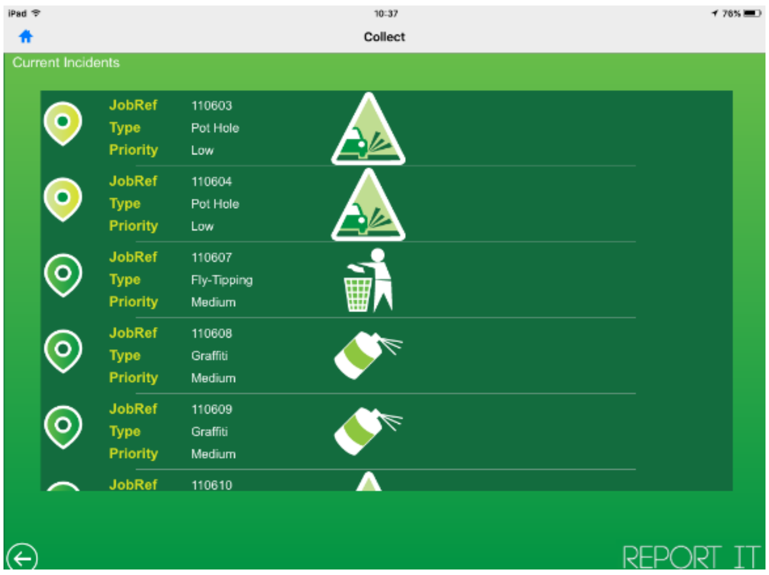
```
if g_strNewRecord <> "True" then
    CreateRecord("Custom.ReportsTemp")
    'msgbox "Record created 1"
    g_strNewRecord = "True"
end if

SetCurrentRecordValue("Custom.ReportsTemp","Date-Time",StrDate)
SetCurrentRecordValue("Custom.ReportsTemp","UserName",g_StrUser)
SetCurrentRecordValue("Custom.ReportsTemp","Type",StrType)
SetCurrentRecordValue("Custom.ReportsTemp","Comment",StrComment)
SetCurrentRecordValue("Custom.ReportsTemp","Priority",StrPriority)
SetCurrentRecordValue("Custom.ReportsTemp","Collected","No")
SetCurrentRecordValue("Custom.ReportsTemp","House",g_strCompany)
g_strTemp = GetControlValue("edtNWLocation")
g_Index = instr(g_strTemp,",")

if g_Index = 0 then
    SetCurrentRecordValue("Custom.ReportsTemp","Street1",g_strTemp)
    SetCurrentRecordValue("Custom.ReportsTemp","Street2","0")
end if

UpdateCurrentRecord("Custom.ReportsTemp")
```

We then need to do the same for the **OnClick** script of the "GPS", "Camera" and "Save" buttons on the form. We should also check for less-obvious instances of "**Report.Incidents**" being used in our app – for example, we need to change the **DeleteRecord()** in the **OnClick** script of the "Back" button on the "New Reports" form. (We can use ctrl + F to easily find all instances of "**Report.Incidents**" in our application).

So, now we have our **Report.Incidents** table, containing all the data we've downloaded from our remote database, and we have our **Custom.ReportsTemp** table, containing any new records that we've created in our application. But what happens if our user wants to modify a pre-existing record in our **Report.Incidents** table?

Modifying an Existing Record

When our user decides they want to modify a pre-existing record, they're going to do this by navigating to the View Reports (frmCollect) form and selecting a record from the listview control.



JobRef	110603
Type	Pot Hole
Priority	Low
JobRef	110604
Type	Pot Hole
Priority	Low
JobRef	110607
Type	Fly-Tipping
Priority	Medium
JobRef	110608
Type	Graffiti
Priority	Medium
JobRef	110609
Type	Graffiti
Priority	Medium
JobRef	110610

The listview contains data mapped from our **Custom.ListView** table – we'll go into more detail about how this table is populated later. They will then be taken to the Info (frmInfo) form, where all the fields will be populated with details about the incident they've chosen to look at. The user is able to modify the Comments field on the form, as well as indicating whether or not the incident has been completed. None of this functionality is affected by whether or not our **Report.Incidents** table is locked, as we can still read data from the table while a sync is in progress. However, once our user clicks the Save button on the form, they'll encounter issues if our data source is locked at the time.

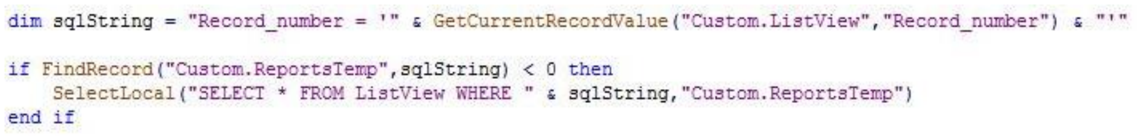
So, how do we get round this?

Well, there are again two ways we could facilitate this.

The first (and simplest) would be to allow the user to view and modify a record from the **Report.Incidents** table on the Info form, but prevent them from saving any changes to the record until the **Report.Incidents** table is unlocked. We could implement this by checking the status of the **Report.Incidents** table in the **OnLoad** script of the Info form, and hiding or displaying the form's Save button dependent upon the outcome. We could then modify our **OnAsyncCompletion** script to check whether or not the Info form was the current form at the time the **Synchronise()** transaction terminated, then display the Save button and allow the user to modify the record if so. While this is a perfectly viable solution, our underlying aim here is to ensure the user has access to as much of their application's functionality as is possible while communication between their device and the server is taking place.

That brings us to our second (and slightly more elegant) solution, which again involves the use of our **ReportsTemp** custom table. As previously mentioned, we can still read data from our **Report.Incidents** table while it is locked for editing. This means that when the user selects the record they want to edit, we can copy that record over into our **ReportsTemp** custom table and are then free to modify it without being hampered by our data source being locked. While this sounds reasonably simple, there is slightly more to consider here.

Let's go to the **OnLoad** script of our Info form. Firstly, we are going to obtain the record ID of the selected record from the **Custom.ListView** table. We're then going to attempt to find the record with that ID in our **Custom.ReportsTemp** table. If we find it, we know that we've either created or modified that record in our current "app session". If we don't find it, we know that the record is a pre-existing incident present in our **Report.Incidents** table that needs copying over into our **Custom.ReportsTemp** table before we can modify it. We can do this using a **SelectLocal()** statement – we know that our **Record_number** column is used as the primary key for this table and is therefore a unique value, so our **SelectLocal()** command will only copy across the single record that we want to work with.

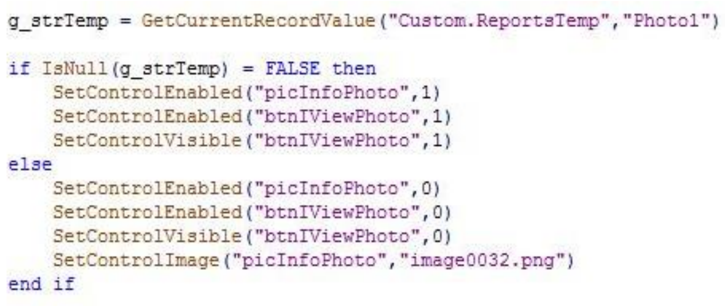


```
dim sqlString = "Record_number = '" & GetCurrentRecordValue("Custom.ListView","Record_number") & "'"

if FindRecord("Custom.ReportsTemp",sqlString) < 0 then
    SelectLocal("SELECT * FROM ListView WHERE " & sqlString,"Custom.ReportsTemp")
end if
```

At present, we actually have two copies of this record (assuming of course that we're using one pulled from our **Report.Incidents** table). At some point, we will need to copy all of the created and modified records from our **Custom.ReportsTemp** table into our **Report.Incidents** table so they can be synced with the database, which we will not be able to do until we've deleted all the duplicates from the table. However, we can't do this until we know for a fact that the table isn't locked for syncing, so we're going to leave them in place for now.

When displaying the contents of the record on the Info form, we can use another new MX v10 method in the form's **OnLoad** event to check whether the record contains a photograph: **IsNull()**.

IsNull() simply allows us to check whether or not a value is NULL – in this case, the value is taken from the "**Photo1**" field of our current **Custom.ReportsTemp** record. If the value is NULL, we know a photograph wasn't taken at the time the record was created, and we can therefore hide/disable the relevant controls. Conversely, if the value isn't NULL we know a photograph was taken and so can show/enable the relevant controls:



```
g_strTemp = GetCurrentRecordValue("Custom.ReportsTemp","Photo1")

if IsNull(g_strTemp) = FALSE then
    SetControlEnabled("picInfoPhoto",1)
    SetControlEnabled("btnIViewPhoto",1)
    SetControlVisible("btnIViewPhoto",1)
else
    SetControlEnabled("picInfoPhoto",0)
    SetControlEnabled("btnIViewPhoto",0)
    SetControlVisible("btnIViewPhoto",0)
    SetControlImage("picInfoPhoto","image0032.png")
end if
```

The user can now edit the details for their chosen incident as required, obviously provided all the relevant controls were input/output-mapped from/to our **Custom.ReportsTemp** table, as described earlier. Once the user clicks the "Save" icon, we want to update the current record and move the user to the View Reports form.

As mentioned previously, the listview control on the form contains information pulled from our **Custom.ListView** table. We need to work out how to populate this table with data from both our **Custom.ReportsTemp** and **Report.Incidents** tables without attempting to enter any duplicate records.

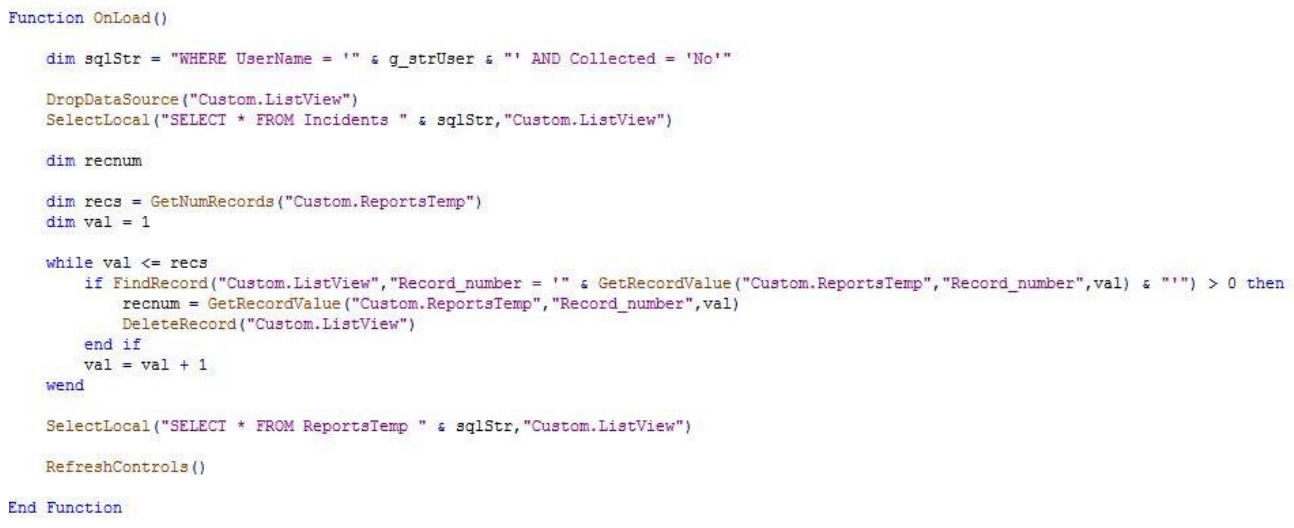
We can do this in the **OnLoad** script for the View Reports form.

The first thing for us to do is clear all the data from our **Custom.ListView** table – we're about to populate it with new data, and don't want any clashes.

Next, we want to copy all the records from **Report.Incidents** into our **Custom.ListView** table. We can use a **SelectLocal()** to achieve this, and as we're only reading the data we don't need to worry about the table being locked for syncing.

We then want to copy all the relevant records from our **Custom.ReportsTemp** table. At this point, the **Custom.ListView** table is an exact copy of the **Report.Incidents** table – thus, any records from **Report.Incidents** that we copied into **Custom.ReportsTemp** to modify are now present in **Custom.ListView**. If you have sufficient SQL knowledge you could merge the two tables using an appropriate SQL Select statement in a second **SelectLocal()** call. Alternatively, we can manually search for duplicate records and delete them from the

Custom.ListView table before copying in the "new" modified versions from **Custom.ReportsTemp**. We can do this by using **GetNumRecords()** to determine the size of the **Custom.ReportsTemp** table, cycle through the table using either a while statement or a for loop, and delete any records present in **Custom.ReportsTemp** from our **Custom.ListView** table. Once we've cleared the table of any duplicates, we can then go ahead and copy the full **Custom.ReportsTemp** table in our **Custom.ListView** table using a **SelectLocal()** command. This will give us our listview populated with all the records from both of our tables:



```
Function OnLoad()

    dim sqlStr = "WHERE UserName = '" & g_strUser & "' AND Collected = 'No'"

    DropDataSource("Custom.ListView")
    SelectLocal("SELECT * FROM Incidents " & sqlStr,"Custom.ListView")

    dim recnum

    dim recs = GetNumRecords("Custom.ReportsTemp")
    dim val = 1

    while val <= recs
        if FindRecord("Custom.ListView","Record_number = '" & GetRecordValue("Custom.ReportsTemp","Record_number",val) & "'") > 0 then
            recnum = GetRecordValue("Custom.ReportsTemp","Record_number",val)
            DeleteRecord("Custom.ListView")
        end if
        val = val + 1
    wend

    SelectLocal("SELECT * FROM ReportsTemp " & sqlStr,"Custom.ListView")

    RefreshControls()

End Function
```

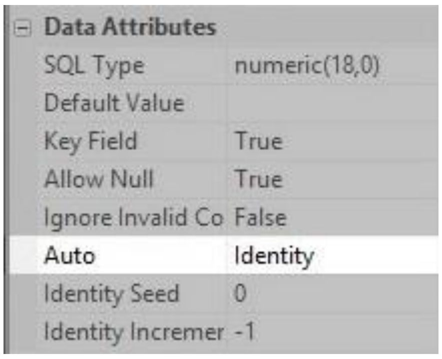
We can also use a variation upon this method to merge our **Report.Incidents** and **Custom.ReportsTemp** tables in preparation for initiating a **Synchronise()** transaction.

Let's return to the **OnClick** script of our "Sync Data" button on the menu form. Currently, when the user clicks the "Sync" button the app uploads and then downloads data from and to our **Report.Incidents** table. However, the introduction of our **Custom.ReportsTemp** table means that any records that the user created or modified within the application are now created/modified in the **Custom.ReportsTemp** table, and so won't be uploaded when calling the **Synchronise()** command. So before we can sync the data, we need to copy any records we've created in our **Custom.ReportsTemp** table into our **Report.Incidents** table, making sure we don't create duplicates in the table, and then delete all data from our custom table so it can be used again. Let's use a while loop, a **SelectLocal()** and a **DropDataSource()** to accomplish this.

The OnClick event script for the Sync Data button will now look like this:

```
Function OnClick()

    if IsDataSourceLocked("Report.Incidents") = FALSE then
        dim val = 1

        while val <= GetNumRecords("Custom.ReportsTemp")
            dim recNum = GetRecordValue("Custom.ReportsTemp","Record_number",val)
            if FindRecord("Report.Incidents","Record_number = '" & recNum & "'") > 0 then
                DeleteRecord("Report.Incidents")
            end if

            val = val + 1
        wend

        SelectLocal("SELECT * FROM ReportsTemp","Report.Incidents")
        DropDataSource("Custom.ReportsTemp")
        DropDataSource("Custom.ListView")

        dim synctag = "sync"

        if IsServerAvailable() = TRUE then
            SetSelectString("Report.Incidents","UserName = '" & g_strUser & "' AND Collected = 'No'")
            Synchronise(synctag, TRUE, "Report.Incidents","Report.Incidents")
        else
            if IsNetworkAvailable() = TRUE then
                MsgBox("Server unavailable. Try again later.")
            else
                MsgBox("Network unavailable. Check your settings and try again.")
            end if
        end if

    else
        MsgBox("There is already a sync in progress.")
    end if

End Function
```

One important point to note is that we need to make sure we're using our data source fields' identity properties correctly. Given that the user will be creating records directly into our **Custom.ReportsTemp** table and then copying these records over into our main **Report.Incidents** table we need to make sure that the Data Attribute Auto property for our primary key field in our custom table is set to "Identity", while the corresponding property in our **Report.Incidents** and **Custom.ListView** tables is set to "None":



Data Attributes	
SQL Type	numeric(18,0)
Default Value	
Key Field	True
Allow Null	True
Ignore Invalid Co	False
Auto	Identity
Identity Seed	0
Identity Incremer	-1

Check also that the primary key field in our custom table has its Identity Seed property set to 0 (zero) and its Identity Increment property set to -1, as shown in the diagram above. This will prevent MX assigning an Identity to a new record which matches an existing identity in a record downloaded from the remote data source.

Note that our example above aims to provide an illustration of how you might incorporate the new Synchronise() function into an existing app and raises some of the issues around using asynchronous transfers. It isn't intended to be a full and complete discussion of using asynchronous transactions. If you choose to allow asynchronous transactions and you also implement a mechanism to allow users to continue to add and modify records whilst such transfers are in progress, you will need to consider carefully all aspects of the process, such as avoiding the potential for duplicate records to be created in any custom tables used to temporarily store records. For example, if you allow users to cancel a data transfer or a transfer doesn't complete successfully you could end up with newly created records in the synchronised data source table which are still marked as modified after the transaction has ended. If these records are using an Identity field as their primary key, new records created in a temporary custom table whilst the transfer was in progress could potentially have duplicate primary keys.

AddSystemNotification

At the moment, in our *Report It* app we've implemented our **Synchronise()** transaction and solved the locked-data source problem, enabling the user to continue using the application whilst a sync is in progress. This brings us to another issue – how is our user going to know when their **Synchronise()** transaction has finished? How are they going to know whether or not their transaction was successful?

Let's return to our **OnAsyncCompletion** script. Our **if** statement clause currently only applies to those **Synchronise()** transactions with the tag "initial". Let's modify this to encompass our "sync" transactions and incorporate an extended MX v10 feature: notifications. Previously notifications were only available for Android and iOS devices. In MX v10 this has been extended to cover Windows and Windows Phone devices as well.

The simplest way to think of notifications is as slightly more complicated message boxes. A message box is intended to inform the user of a specific piece of information, triggered by a specific event. Similarly, notifications are designed to inform the user of something, but can be triggered by an event or at a specific time and aren't bound to a particular form. Unlike message boxes, they are displayed to the user even when their app isn't running. Note, however, that on Windows devices, the MX Client must be running for notifications to display, which if you are using Standalone Apps effectively means the app must be running.

As an example, the notifications feature is used in NDL's District Nursing app to inform the user when their next appointment with a patient is due, and is triggered several minutes before said appointment. When a notification is triggered it will appear regardless of the current form the user is on in the application (or, on most platforms, whether or not they're using the application at all). We are going to use the notifications feature in our *Report It* app to let the user know when a **Synchronise()** transaction has completed, and whether or not it has completed successfully.

Notifications are initiated using the **AddSystemNotification()** method. Similar to the **Synchronise()** method, the first parameter in any **AddSystemNotification()** call is the tag parameter, used as a unique identifier for that notification. The notification in our **OnAsyncCompletion** script will only be instantiated when our “sync” transaction terminates, so for ease of use let’s call our notification “sync” too.

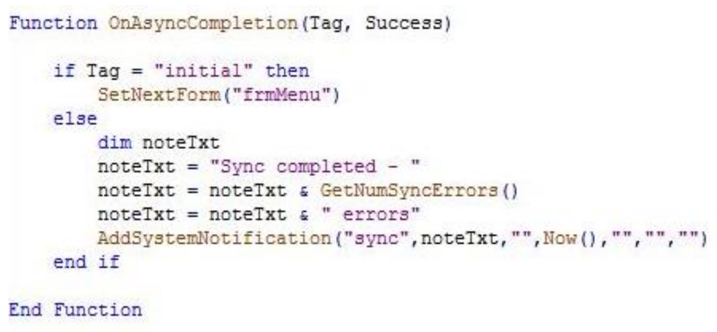
The second parameter of the **AddSystemNotification()** is the message parameter. We’re going to use another new MX v10 method - **GetNumSyncErrors()** – to create a notification message which will let the user know whether or not our **Synchronise()** transaction completed successfully.

The third parameter is the action/title parameter. We are going to leave the action argument blank. On iOS devices if an action argument is specified and the user clicks on the “action” section of the notification pop-up then the **OnSystemNotification** application-level event is triggered. So, we could specify “View errors” for our action argument and have the **OnSystemNotification** event take the user to our error-handling form (explained in the next section below.). In this case, we don’t want the user to navigate off our New Reports form without either updating or deleting the record they’re working on, so we’re going to leave this parameter blank. On Android, Windows and Windows Phone devices, however, this parameter specifies a title to be displayed in the notification pop-up and tapping on the notification will trigger the **OnSystemNotification** event, whether a title is included or not.

The fourth parameter is the date and time the notification should be displayed. In this case, we’re using the notification as a way of informing the user that their **Synchronise()** transaction has completed, so we want to display the notification straight away.

The fifth, sixth and seventh parameters are “sound”, “repeat” and “options”, respectively – we don’t want our notification to play a sound or repeat and we’re happy with the default options so we’re going to leave all these parameters blank.

Our finished script will look something like this:



```
Function OnAsyncCompletion(Tag, Success)

    if Tag = "initial" then
        SetNextForm("frmMenu")
    else
        dim noteTxt
        noteTxt = "Sync completed - "
        noteTxt = noteTxt & GetNumSyncErrors()
        noteTxt = noteTxt & " errors"
        AddSystemNotification("sync",noteTxt,"",Now(),"","","")
    end if

End Function
```

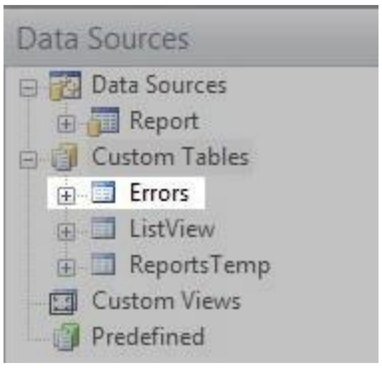
Error Handling

So, we’ve now implemented several **Synchronise()** transactions and have used the notification feature to inform the user when a **Synchronise()** transaction completes, as well as whether or not the transaction has completed successfully. In its current state, our application's system for handling errors is limited, at best – we can tell the user whether or not their transaction encountered any errors but we can't tell them what errors or why.

Synchronise() brings with it a whole host of error-handling methods. We're going to use three of these methods in our app: **GetNumSyncErrors()**, which we've already seen above, **GetSyncErrorRecordIndex()** and **GetSyncErrorDescription()**. We're going to use these three methods to populate a custom table containing details of any sync errors, and then display it to the user.

First let's create a new form in our app, containing a listview control that will display all of our error data. We should also create a template to use as the listview input.

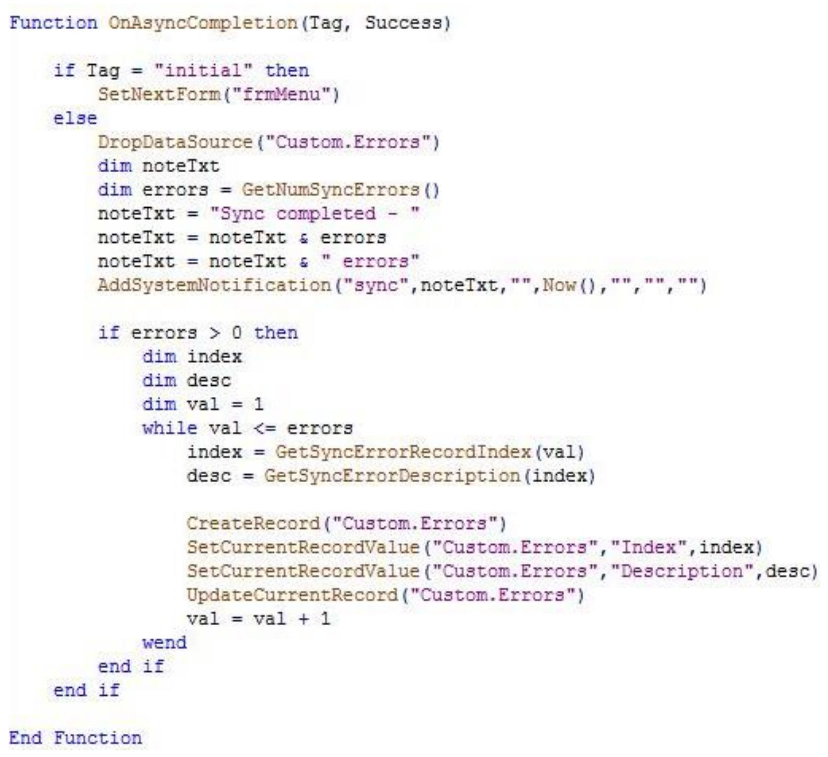
Next, we are going to create a new custom table in our application to store the error data. We want to have columns in our table for the error index and error description. Let's call this table "**Custom.Errors**".



Now we have the infrastructure in place to store our data, lets return to our **OnAsyncCompletion** script. As before, we can use **GetNumSyncErrors()** to obtain the number of errors encountered during a specific transaction.

We're then going to cycle through these errors, obtain the record index of the error using **GetSyncErrorRecordIndex()** and then use that index and then **GetSyncErrorDescription()** method to get the description of the error.

We're then going to create a record in our **Custom.Errors** table and update that record with the index and description of the error.



```
Function OnAsyncCompletion(Tag, Success)

    if Tag = "initial" then
        SetNextForm("frmMenu")
    else
        DropDataSource("Custom.Errors")
        dim noteTxt
        dim errors = GetNumSyncErrors()
        noteTxt = "Sync completed - "
        noteTxt = noteTxt & errors
        noteTxt = noteTxt & " errors"
        AddSystemNotification("sync",noteTxt,"",Now(),"","","")

        if errors > 0 then
            dim index
            dim desc
            dim val = 1
            while val <= errors
                index = GetSyncErrorRecordIndex(val)
                desc = GetSyncErrorDescription(index)

                CreateRecord("Custom.Errors")
                SetCurrentRecordValue("Custom.Errors","Index",index)
                SetCurrentRecordValue("Custom.Errors","Description",desc)
                UpdateCurrentRecord("Custom.Errors")
                val = val + 1
            wend
        end if
    end if

End Function
```

There is also a fourth error-handling method called **GetSyncErrorDataSource()**. Since our *Report It* app only syncs one table – our **Report.Incidents** table – we're not going to include this method. However, it is a useful tool to bear in mind when implementing error handling for apps using several tables.

As the details of any errors obtained during syncing are stored in our custom table, it means the user can receive a notification informing them that errors are present, finish the task they were undertaking and go back and look at the error data later.

Summary

We've now covered the basics of rewriting existing applications to use several new features and methods introduced in MX v10, and have looked at implementing asynchronous communication, notifications and error handling. For further information, refer to the *Digitise Apps Online Help* available by clicking the Help button, ?, within the Digitise Apps App Studio or App Manager utilities.

The Haybarn at Parkhill Walton Road Wetherby West Yorkshire LS22 5DZ

01937 543 500

Support: **01937 543 510**

support@ndl.co.uk