

Data Synchronization Exercise

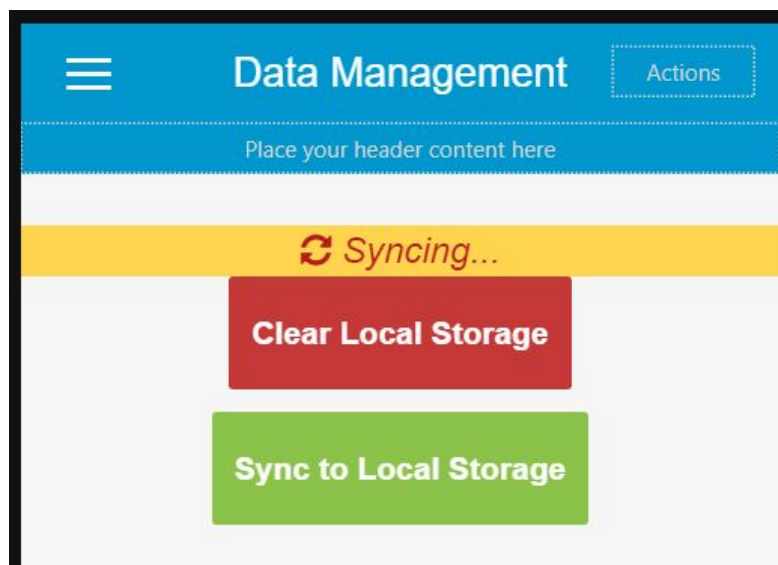


Table of Contents

Table of Contents	2
Introduction	3
Syncing Categories, Priorities and ResourceTypes	4
Create the DataManagement Screen	8
Testing the app: Create the first local ToDo	15
Enable Offline Changes of To Dos	16
Syncing ToDos and Resources	25
Testing the app: Synchronization and offline interaction	38
Automatic Synchronization	39
Is Syncing Feedback	40
Testing the app: automatic sync and feedback	42
End of Lab	42

Introduction

This exercise lab will handle two very important things in a mobile app: offline scenarios and data synchronization. For that matter, there are several important tasks that will be performed in this exercise.

First, we will use Service Studio accelerators to generate the Read-only synchronization logic for the Categories, Priorities and ResourceTypes. This will create the server-side and client-side logic for synchronizing these three elements. This logic will then be used in the OfflineDataSync Action, to be part of the complete synchronization process.

Then, we will add a new Screen to the application, DataManagement, to ensure that the synchronization can be manually triggered by the end-user. We will add the respective Menu and Bottom Bar entry for this Screen.

After that, we will change the logic to add / update Todos to work on offline scenarios, adding the data to local storage even if the app is offline. This will also require some changes in the data model in order to save information about the changes made while offline, which will later be useful for the synchronization, when the app gets online.

With all that logic ready, we will then manually create the Read / Write logic to synchronize the Todos and the Resources to the server. This will allow that Todos and Resources created in local storage are also sent to the server.

Finally, we will define automatic triggers for the synchronization, when the app gets online, or when the user logs in or resumes the app. We will also provide a message to the end-user so that it is clear when the synchronization is happening.

In summary, in this specific exercise lab, we will:

- Define the logic for synchronizing Todos, Resources, ResourceTypes, Categories and Priorities
- Enable creating and updating Todos while offline
- Create a new Screen to manually trigger the synchronization
- Define automatic triggers for the synchronization
- Display a message to the user that the synchronization process is happening

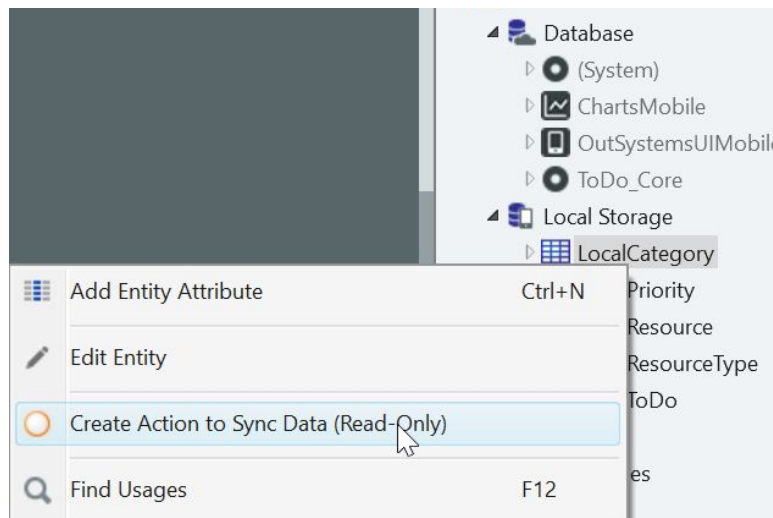
Syncing Categories, Priorities and ResourceTypes

At this point, the application has Database Entities and Local Storage Entities, with the Screens having as source the local storage data. Since the application started exclusively with Database Entities, there are still no data in the local storage.

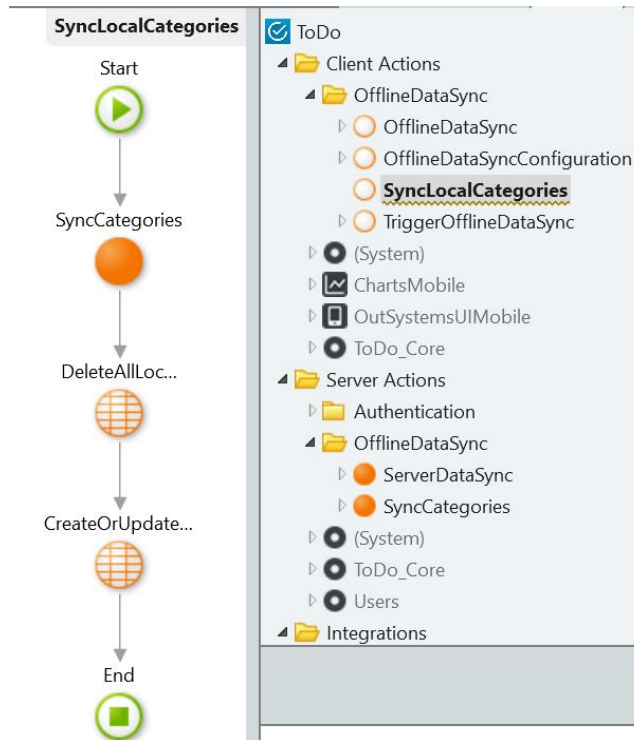
In this section, we will start defining the synchronization logic, starting by the **Category**, **Priority** and **ResourceType** Entities. For that purpose, we will use some OutSystems accelerators that automatically create logic for synchronizing data, from a Database Entity to Local Storage, using Read-Only and Read/Write patterns.

This automatically created logic will then be used in the **OfflineDataSync** Client Action. This Action runs when the synchronization process is triggered, and it is where the client-side logic of the synchronization is defined.

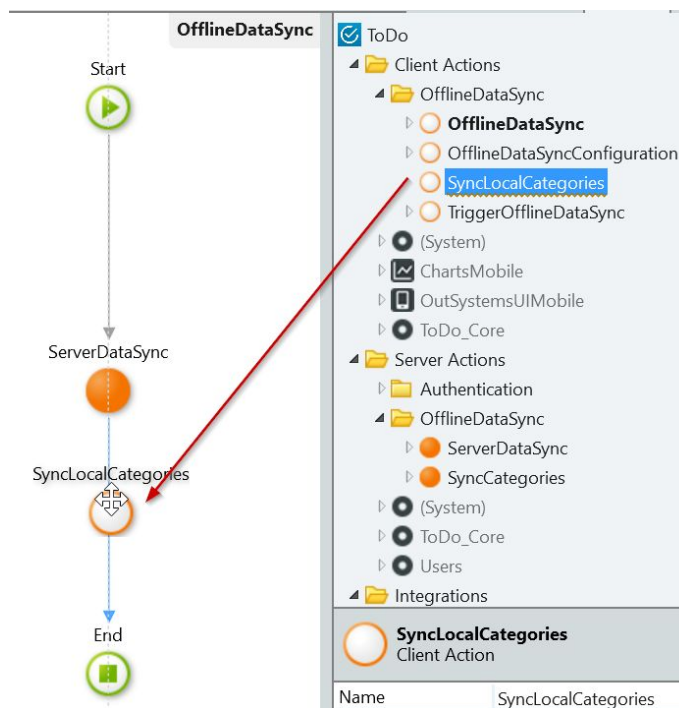
- 1) Create the **Read-Only** logic for synchronizing the **Category** data from the Database to the respective Local Storage Entity, using the Service Studio accelerators. Add the created logic to the **OfflineDataSync** Action to make it part of the synchronization.
 - a) Switch to the Data Tab, right-click **LocalCategory** and then select *Create Action to Sync Data (Read-Only)*.



This creates a Client Action under the OfflineDataSync folder in the Logic tab, **SyncLocalCategories**. The Action fetches the data from the Server, **SyncCategories** (also created by the accelerator), and then replaces all the data in the Local Storage for what was fetched from the Server. First, the **DeleteAllLocalCategories** clears all data from the **LocalCategory**, and then the **CreateOrUpdateAllLocalCategories** adds all Categories from the Database, to the respective Local Storage Entity.

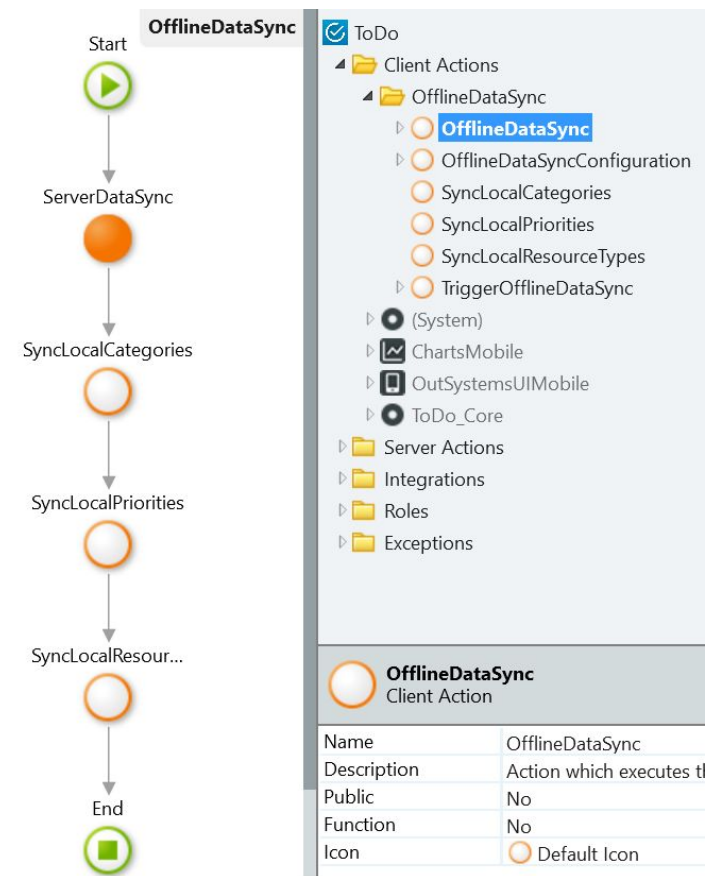


- b) In the Logic tab, open the **OfflineDataSync** Action, and drag the new **SyncLocalCategories** to the flow, below the ServerDataSync Action.



- 2) Repeat the previous steps for the **LocalPriority** and **LocalResourceType**. Both Entities will be synchronized using a Read-Only strategy.

- a) In the Data tab, right-click on the **LocalPriority** Entity and select *Create Action to Sync Data (Read-Only)*.
- b) Repeat the process for the **LocalResourceType** Entity.
- c) This creates two Client Actions: **SyncLocalPriorities** and **SyncLocalResourceType**. Both Actions have a similar logic to the SyncLocalCategory created above.
- d) Open the **OfflineDataSync** Client Action again, then drag both Client Actions (**SyncLocalPriorities** and **SyncLocalResourceTypes**) to the OfflineDataSync flow, below the SyncLocalCategories.



- e) Publish the module to save the changes in the server.

NOTE: These accelerators create self-contained logic for each pair of Database and Local Storage Entities, for instance the Categories. This means that doing this for the three Entities, will create three separate pieces of logic. When all of this is used together, for instance, in the OfflineDataSync Action, we actually have three calls to the server (one per each Database Entity), since logic was created for that using the accelerators. Having a Client Action calling the server multiple times should actually be done with care and only if strictly necessary, since there will be multiple communications being done to the server.

In these scenarios, the best practice is to try to wrap all the server logic in one single Server Action and call it. This way, we only have one connection to the server, instead of several.

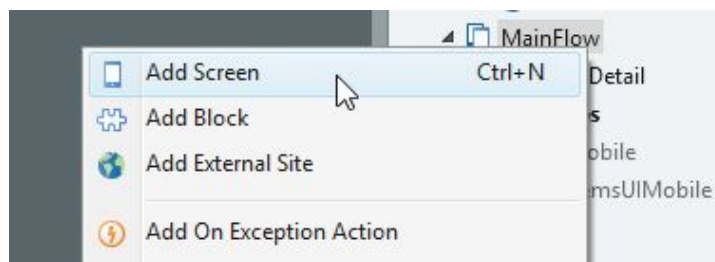
Create the DataManagement Screen

The logic for synchronization is already created, however the synchronization is not triggered in any form yet. A first approach to do that will be a manual trigger.

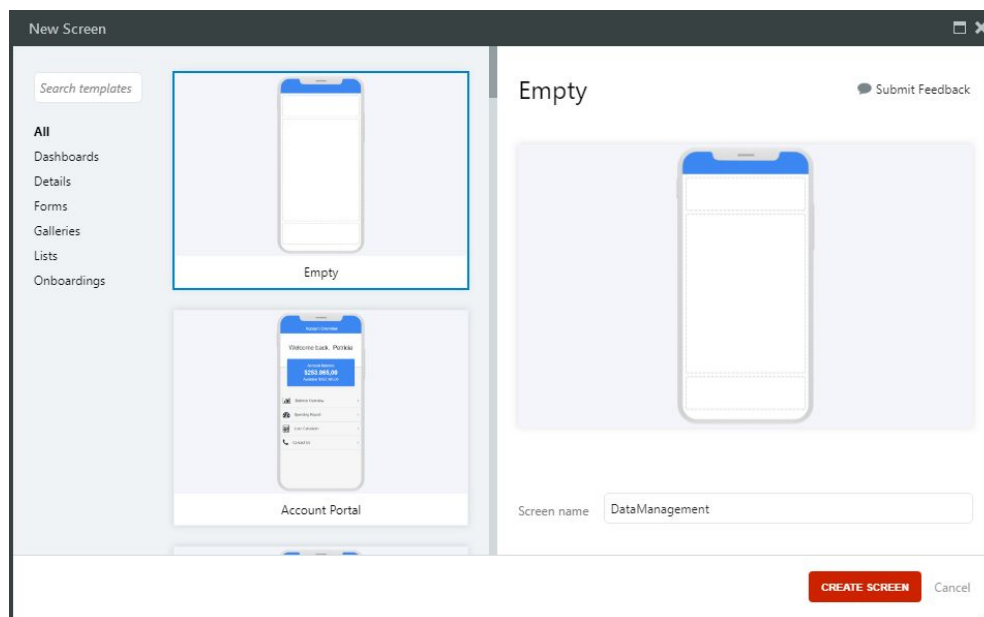
In this section, we will add a new Screen to the application, **DataManagement**, that will allow a user to manually trigger the synchronization process. This Screen will also have a button that will allow clearing all local storage, if the end-user desires.

Finally, we need to add a new Menu entry for this Screen, and update the Bottom Bar to properly link to this new Screen.

- 1) Create the *DataManagement* Screen, using the **Empty** template. Add a new entry to the Menu, using the *gears* icon, and link the respective Bottom Bar item to the Screen.
 - a) Under the Interface tab, right-click the MainFlow and select the Add Screen option.



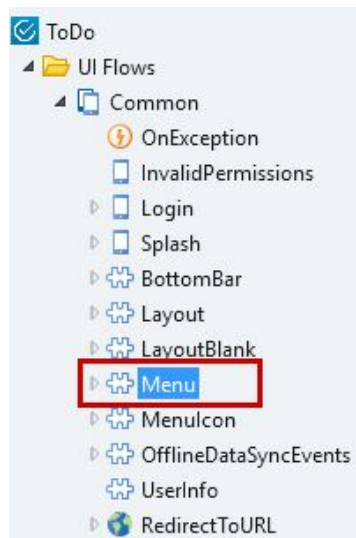
- b) In the New Screen dialog, select the **Empty** template and enter *DataManagement* as the Screen name.



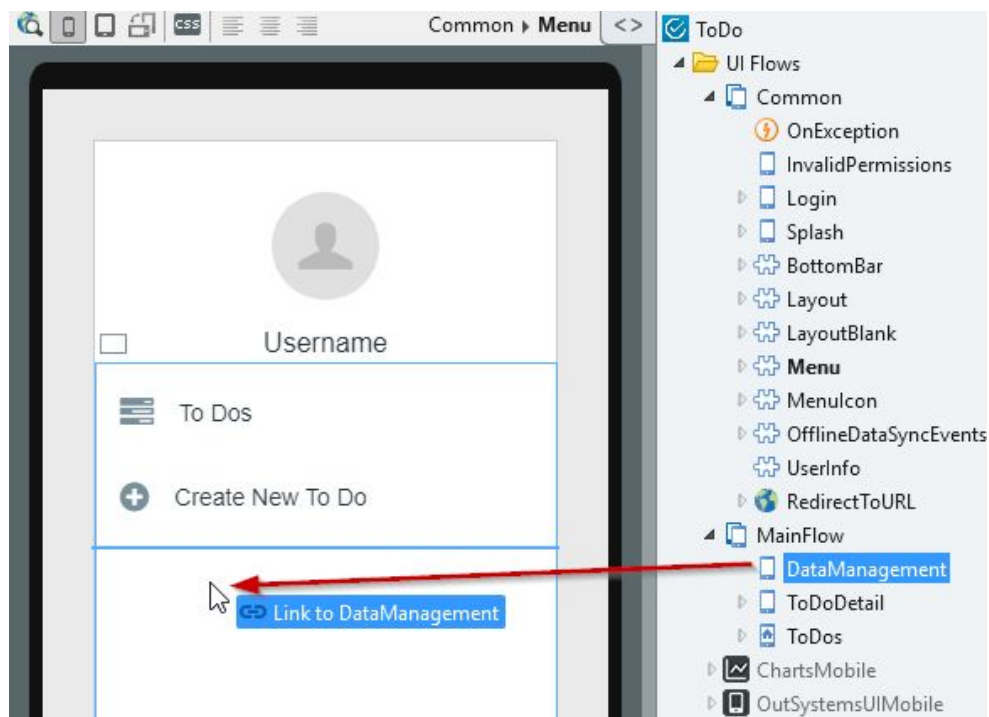
- c) Set the Title of the Screen as *Data Management*.



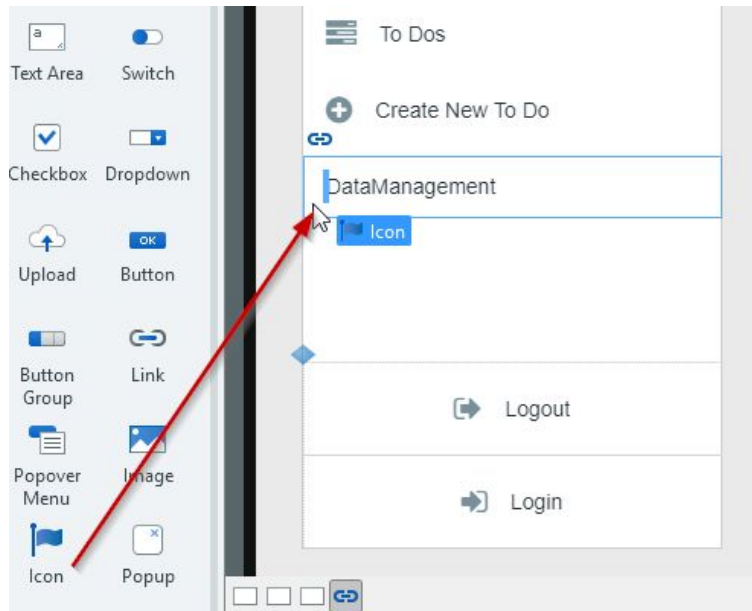
- d) Expand the Common flow and open the Menu Block



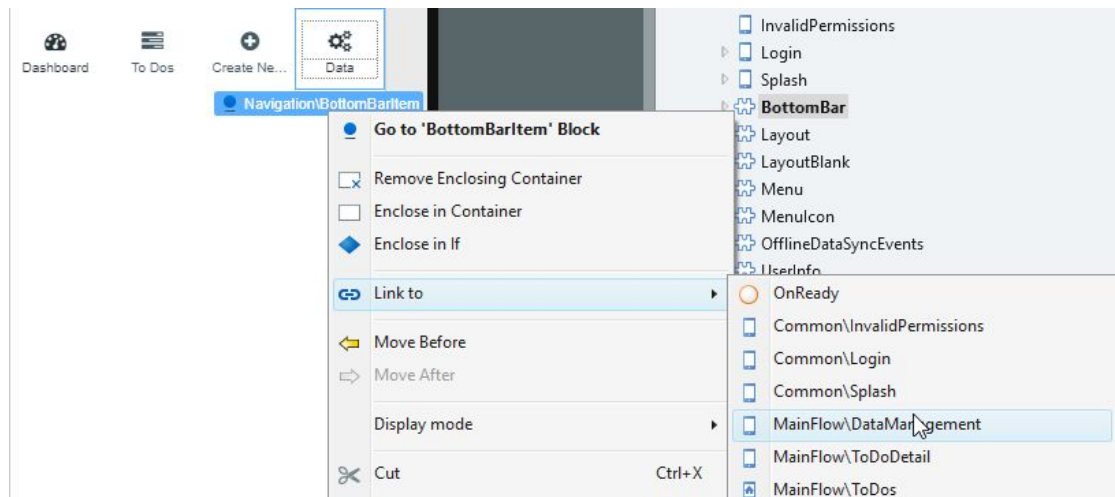
- e) Drag the **DataManagement** Screen below the **Create New To Do** Menu option



- f) Drag an **Icon** right before the text **DataManagement** and choose the *gears* icon.

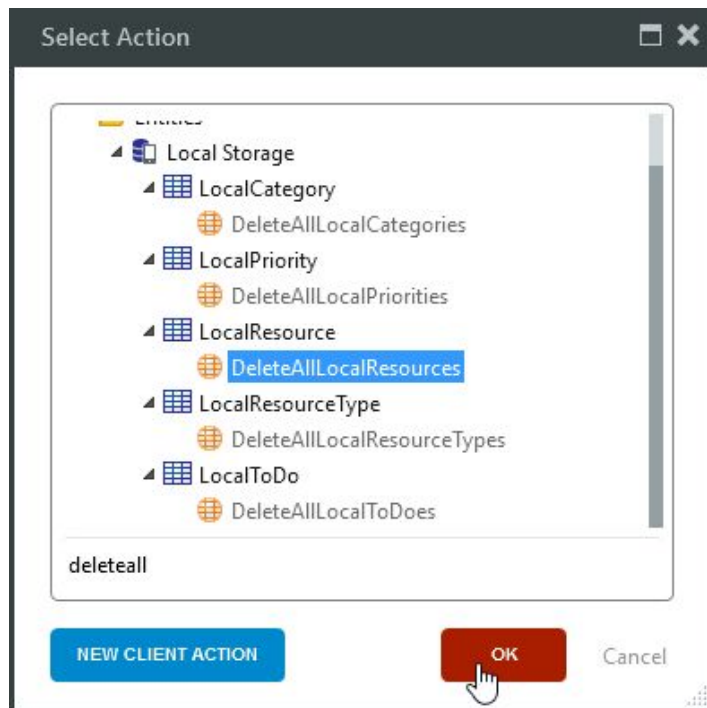


- g) Add an empty space between Data and Management to have *Data Management*.
- h) Open the **BottomBar** Block, also available under the Common UI Flow.
- i) Select the **Data** Bottom Bar Item and Link it to the **DataManagement** Screen.



- 2) Create a *Clear Local Storage* button in the **DataManagement** Screen, centered on the Screen and with the red color, to allow clearing the Local Storage if desired. Link it to an Action that deletes every record of all Entities in local storage.
- a) Open the **DataManagement** Screen.
 - b) Drag a **Button** Widget, drop it inside the Screen content. Align it to the center.

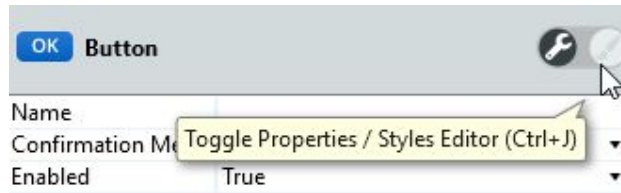
- c) Change the text of the Button to *Clear Local Storage*.
- d) Change the Button **Style Classes** property to *"btn btn-danger"*. This will make the Button have the color red.
- e) Double-click the Button to create the **ClearLocalStorageOnClick** Client Action.
- f) Drag a **Run Client Action** statement and drop it between Start and End, in the recently created Action flow.
- g) In the **Select Action** dialog choose **DeleteAllLocalResources**.



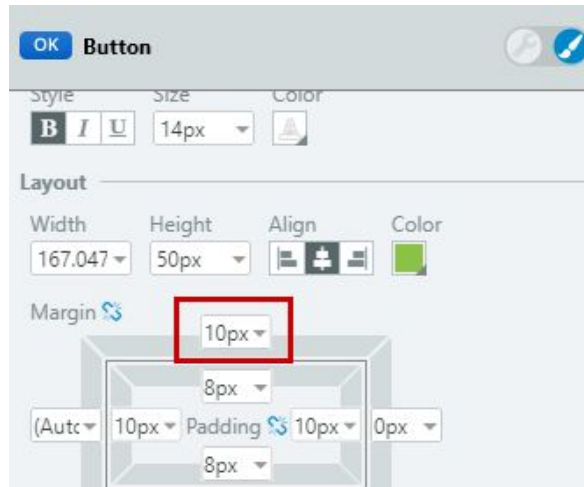
- h) Repeat the same process to add the **DeleteAllLocalToDoes**, **DeleteAllLocalCategories**, **DeleteAllLocalPriorities** and **DeleteAllLocalResourceTypes**.
- i) Drag a **Message** statement and drop it between the DeleteAllLocalResourceTypes and the End.
- j) Set the **Message** property to *"All data cleared!"*, and the **Type** to *Success*. The Action should look like the following screenshot



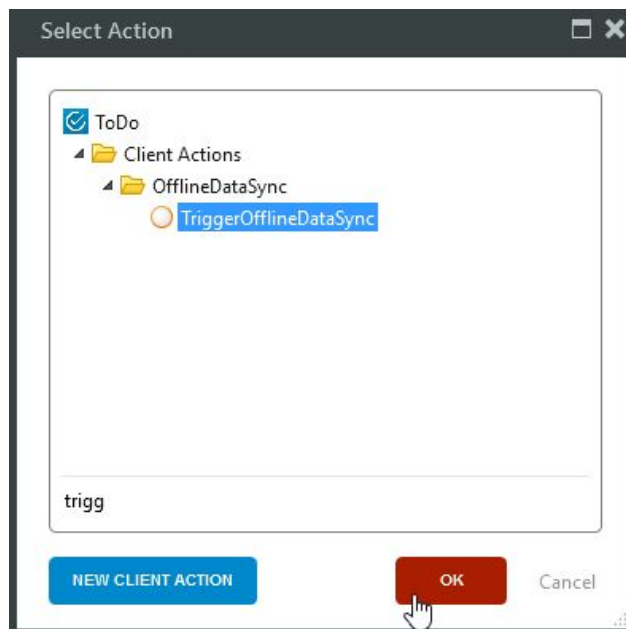
- 3) Create the *Sync to Local Storage* button in the **DataManagement** Screen, centered on the Screen and with the green color, to trigger the synchronization. Link it to an Action that starts the synchronization process in the background.
 - a) Return to the **DataManagement** Screen.
 - b) Drag another **Button** Widget and drop it below the **Clear Local Storage** Button.
 - c) Select the Button, align it to the center of the Screen and change its text to *Sync to Local Storage*.
 - d) Set the Button's **Style Classes** property to *"btn btn-success"*.
 - e) Switch to the Styles Editor of the Button.



- f) Set the **Margin** to the top as **10px**, to separate this Button from the one above.



- g) Double-click the Button to create the **SynctoLocalStorageOnClick** Client Action.
- h) Drag a **Run Client Action** and drop it between Start and End.
- i) In the Select Action dialog choose **TriggerOfflineDataSync**.



This will trigger the data synchronization to run in the background. This Action will call the OfflineDataSync **asynchronously**, without blocking the application or make the user wait to continue interacting with the app. On the other hand, if we indeed called the OfflineDataSync Action directly, that would happen.

- j) Publish the module to save the most recent changes to the server.

Testing the app: Create the first local ToDo

In this section, we will do a first test of the application, by creating a ToDo in local storage. By doing that, we need to make sure that the synchronization works, in order to have Categories available in Local Storage.

- 1) Open the application in the browser, or on the device. Notice that a message appears indicating that the app was updated to the latest version.
- 2) Navigate to the DataManagement Screen, using the Bottom Bar or the Menu, to make sure the changes work correctly.
- 3) Click on the Button to Sync to local storage.
- 4) Navigate to the Screen to create new ToDos. Make sure that the Categories are listed in the dropdown.
- 5) Navigate back to the DataManagement Screen and click on the Clear Local Storage Button.
- 6) Go back to the Screen to create a new ToDo and make sure that the Dropdown for the Categories is empty. If it is, it means that the Action to clear the local storage is working properly.
- 7) Navigate one last time to the DataManagement Screen and click again on the Sync to Local Storage Button.
- 8) Create a new ToDo and make sure that it is the only one that appears in the ToDos list. This happens because the ToDos is fetching data from local storage, and thus far, this is the only one created in local storage as well.

Enable Offline Changes of To Dos

At this point, when we create or update an existing ToDo, the ToDo is added to the local storage and to the server, which requires that the user is always online to perform that operation.

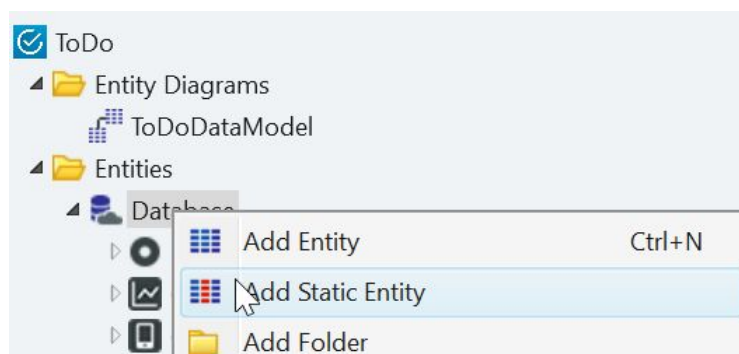
In this section, we will add some logic to guarantee that the ToDos can be created and updated while offline, without throwing an error. This way, when the app is online, the ToDo continues to be added / updated to both database and local storage. When the app is offline, the ToDo is exclusively added / updated to local storage.

However, the offline scenario is a bit more complex than that. When the app comes back online, we want to make sure that the database and local storage synchronize and there is data consistency between both. For that matter, we need to keep track of the changes made while offline, to make sure that only the necessary data is synchronized when the synchronization is triggered. As an example, consider that two ToDos were added with the app offline. If the user has ten ToDos in its application, the synchronization should only send these two ToDos to the server, instead of the whole list.

To help us with that, we need to modify the data model, to hold information about the changes made while offline. That information can be about ToDos **added** or **updated**.

- 1) Add a Sync Status to the **LocalToDos**, to track added and updated To Dos, while the device is offline. For that, we need to create a new Static Entity **SyncStatus**, with the Records *None*, *Added* and *Updated*.

- a) Switch to the **Data** tab and add a new Database Static Entity named *SyncStatus*.

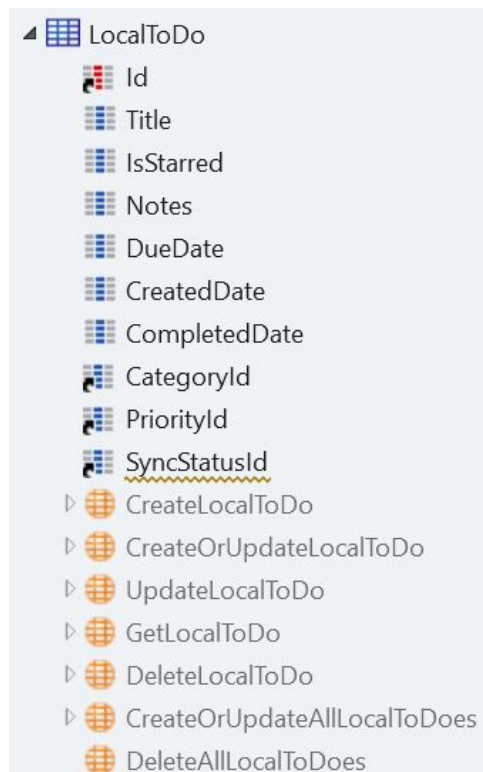


- b) Expand the **SyncStatus** Entity, then right-click the **Records** folder and choose *Add Record*, then set the new record name to *None*.
 - c) Add two more records named *Added* and *Updated*.



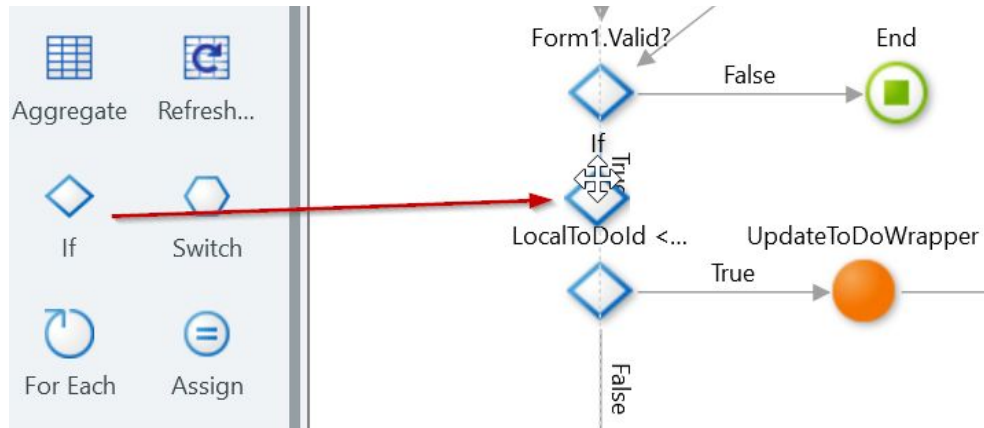
NOTE: The **None** record will track local To Dos that do not have modifications and are in sync with the server. The **Added** record tracks new To Dos created in the local storage but not yet synced to the server. The **Updated** will track the To Dos that exist on the server but have been modified locally.

- d) Add a new attribute to the **LocalToDo** Entity, and set its name to *SyncStatusId* then verify that the **Data Type** was set automatically to *SyncStatus Identifier*.



- 2) Modify the **SaveOnClick** Client Action, of the **ToDoDetail** Screen, to only create / update Todos in the database if the user is online. If it is not, update the **SyncStatus** of the LocalToDo to *Added* or *Updated*, depending on if the ToDo is being created or updated.

- a) Switch to the **Interface** tab and open the **SaveOnClick** Client Action of the **ToDoDetail** Screen.
- b) Drag a new **If** statement and drop it on top of the **True** branch connector, between the **Form1.Valid?** If and the **LocalToDo <> NullIdentifier()** If.

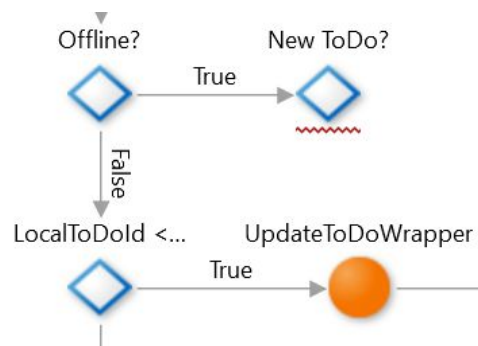


- c) Set the **Label** property of the new If to *Offline?*, and the **Condition** property to *not GetNetworkStatus()*

This Condition determines if the application is offline, since the **GetNetworkStatus()** Action returns *True* if the application is online.

- d) Drag another **If** statement and drop it on the right of the **Offline?** If created in the previous steps. Then, create the **True** branch connector from the existing If to the new one.
- e) Set the **Label** property of the last created If to *New ToDo?*, and the **Condition** property to

LocalToDoId = NullIdentifier()

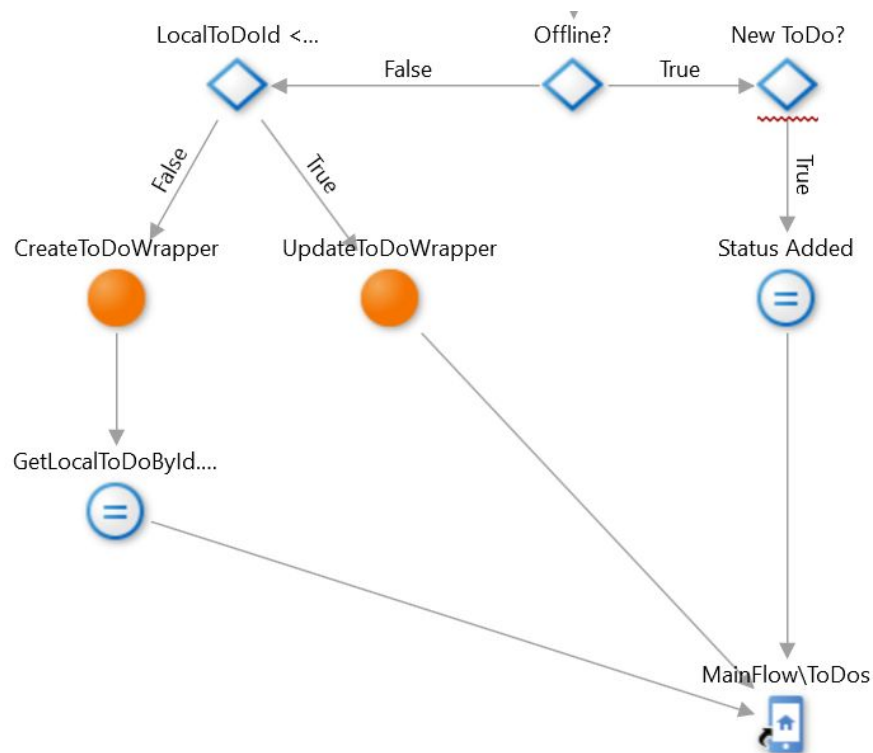


- f) Drag a new **Assign** statement and drop it below the **New ToDo?** If, then create the **True** branch connector from the If to the Assign statement.
- g) Select the **Assign** statement. Change its **Label** to *Status Added* and define the following assignment

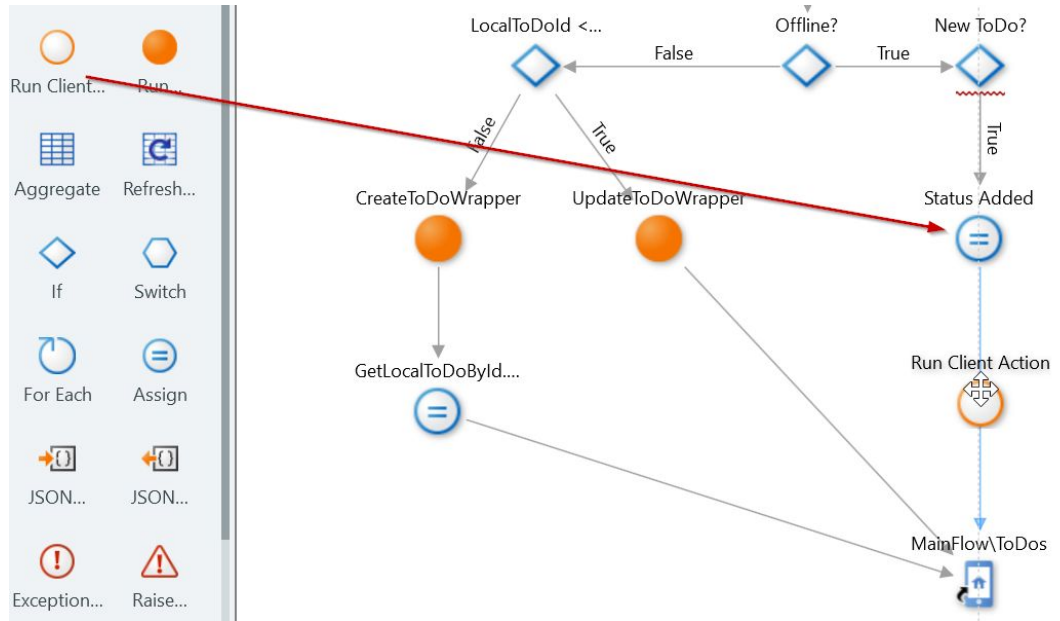
```
GetLocalToDoById.List.Current.LocalToDo.SyncStatusId =
Entities.SyncStatus.Added
```

This will define the LocalToDo as **Added**, which will later be used in the synchronization to create the ToDo in the database.

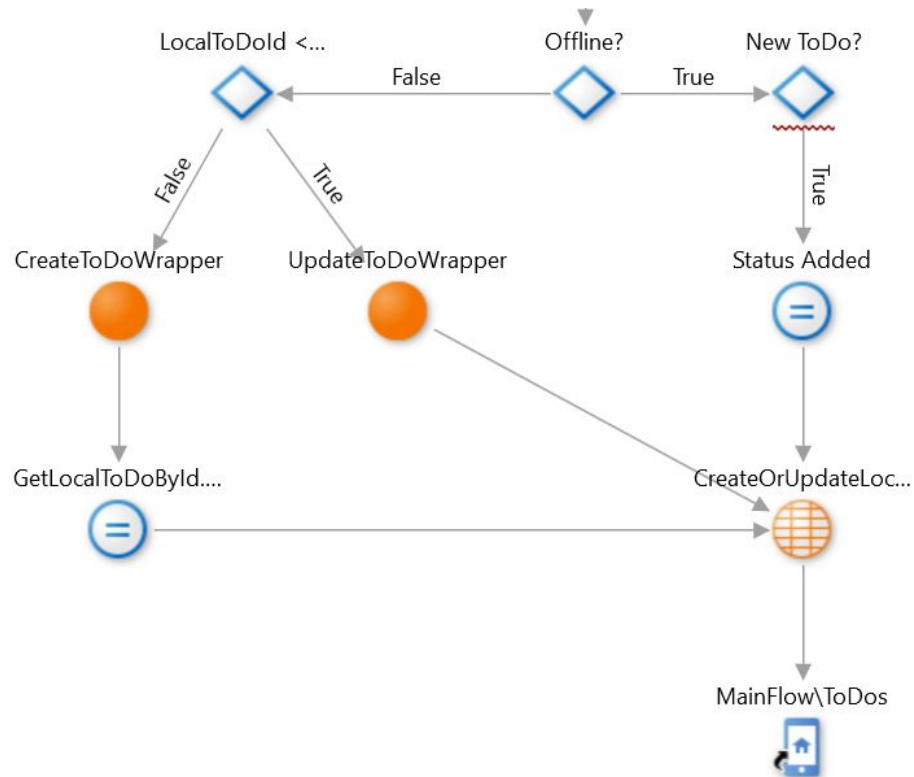
- h) Delete the **CreateLocalToDo** and **UpdateLocalToDo** Action calls. Connect all statements pending with the **MainFlow/Todos** Destination and delete the extra one. Rearrange the flow for the rest of the logic. The final part of the flow should look like the following screenshot.



- i) Drag and drop a **Run Client Action** statement and drop it between the latest Assign and the Destination, on the rightmost flow. Select the *CreateOrUpdateLocalToDo* Action in the **Select Action** dialog.



- j) Set the Source of the Action to be *GetLocalToDoById.List.Current*
- k) Connect the **UpdateToDoWrapper** and the **Assign** below the **CreateToDoWrapper** to the new Client Action.



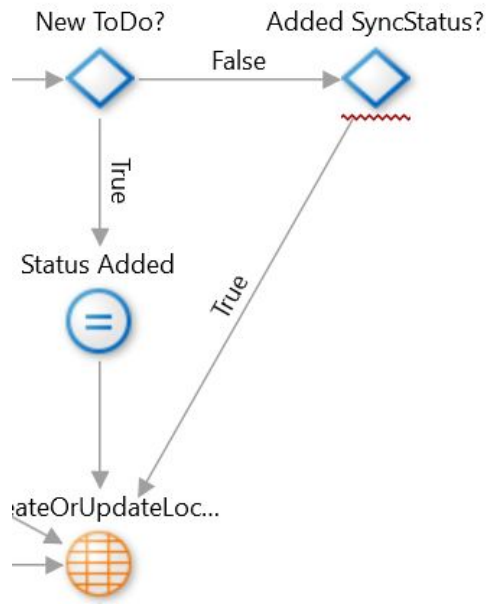
NOTE: These last couple of changes were not strictly necessary. We replaced the two separate calls to the `CreateLocalToDo` and `UpdateLocalToDo` into one, adjusting the logic flow to make sure it works properly. This way, if the app is online, we either add or update the `ToDo` in the database. Then, regardless if the app is offline or not, we create or update the `ToDo` in local storage.

- l) Drag a new **If** statement and drop it on the right of the **New ToDo?** If, then create the **False** branch from the existing If to the new one.
- m) Set the **Label** property of the newly created If to *Added SyncStatus?*, and the **Condition** property to

*GetLocalToDoById.List.Current.LocalToDo.SyncStatusId =
Entities.SyncStatus.Added*

This new branch will handle the cases where the `ToDo`s already exist and we are updating an existing one. This particular If will determine what to do if the `ToDo` already exists, and has its Sync Status as **Added**.

- n) Create the **True** branch connector from the **If** to the **CreateOrUpdateLocalToDo** statement.



If the `ToDo` was already “tagged” as **Added**, we will not change its `SyncStatus` and just update again the `ToDo` in local storage with the new information. It would not make sense to mark a new `ToDo` as **Updated**, when the `ToDo` was created after the app gets offline, thus this `ToDo` is still not in the server.

- o) Drag a new **Assign** statement and drop it below the **Added SyncStatus?** If. Then, create the **False** branch connector from the If to the Assign statement.
- p) Select the **Assign** statement. Change its **Label** to *Status Updated* and define the following assignment

```
GetLocalToDoById.List.Current.LocalToDo.SyncStatusId =
Entities.SyncStatus.Updated
```

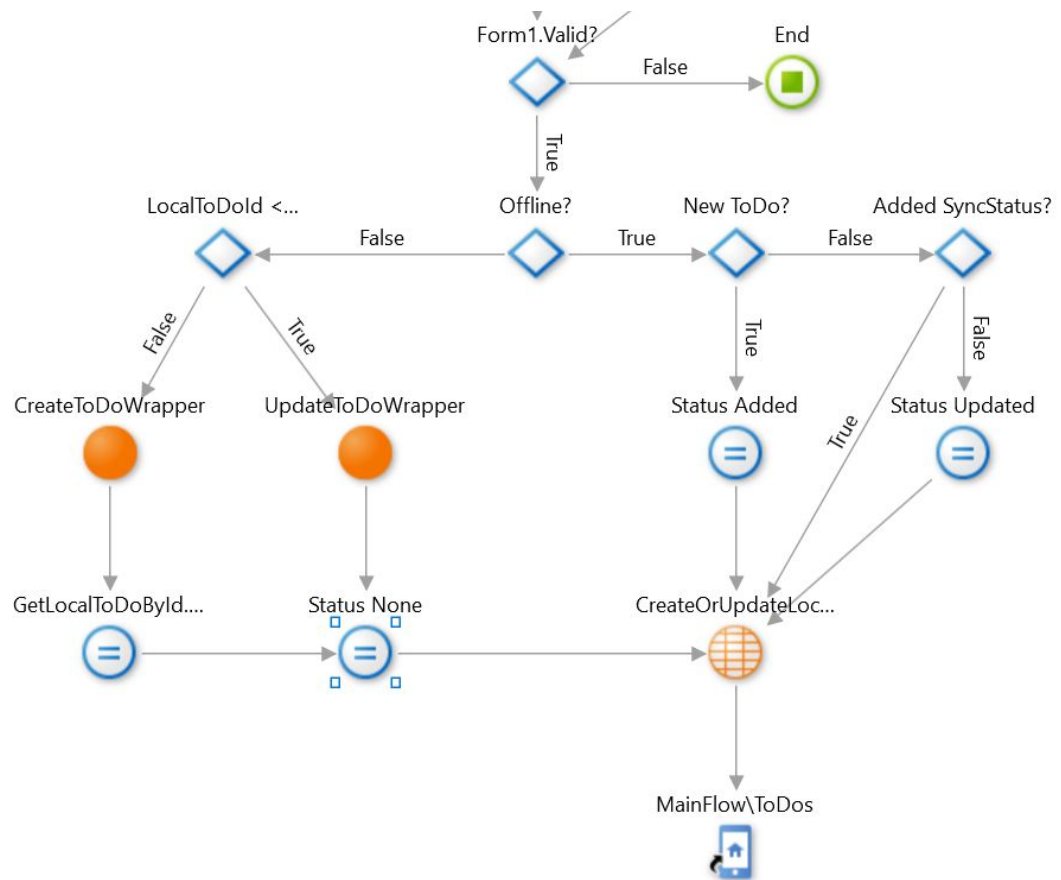
If the ToDo was not marked as Added, it means that it already existed before the app gets offline. Thus, we are updating a ToDo that exists in the database and change its Sync Status to **Updated**.

- q) Create the missing connector from the **Assign** statement to the **CreateOrUpdateLocalToDo** statement.
- r) Drag a new Assign and drop it between the **UpdateToDoWrapper** and the **CreateOrUpdateLocalToDo**. Connect the Assign right below the CreateToDoWrapper to this new one.
- s) Select the new Assign. Set its **Label** to *Status None* and add the following assignment

```
GetLocalToDoById.List.Current.LocalToDo.SyncStatusId =
Entities.SyncStatus.None
```

If the ToDo is added / updated in the database, since the app is online, it is saved in local storage with the Sync Status set to **None**. Remember that the Sync Status is only useful for the future synchronization of data with the server, when the app gets back online.

- t) The final section of the **SaveOnClick** Client Action should look like the following screenshot



- 3) Modify the **StarOnClick** Client Action of the **ToDoDetail** Screen to update the Sync Status of the To Do. Follow the same strategy applied in the previous step.
 - a) Open the **StarOnClick** Client Action.
 - b) Change the **Label** property of the existing **If** to *ToDo exists?*
 - c) Drag a new **If** Widget and drop it on top of the **True** branch connector, between the existing If and the **UpdateToDoWrapper** statement.
 - d) Set the **Label** property of the new **If** to *Offline?*, and the **Condition** property to *not GetNetworkStatus()*
 - e) Drag a new **If** statement and drop it below the **Offline?** If, then create the **True** branch connector between both Ifs.
 - f) Set the **Label** property of the new If to *Added SyncStatus?* and the **Condition** property to


```

          GetLocalToDoById.List.Current.LocalToDo.SyncStatusId =
          Entities.SyncStatus.Added
          
```


This follows the same strategy as in the SaveOnClick. If the ToDo was already marked as **Added**, it will not be marked as **Updated**.

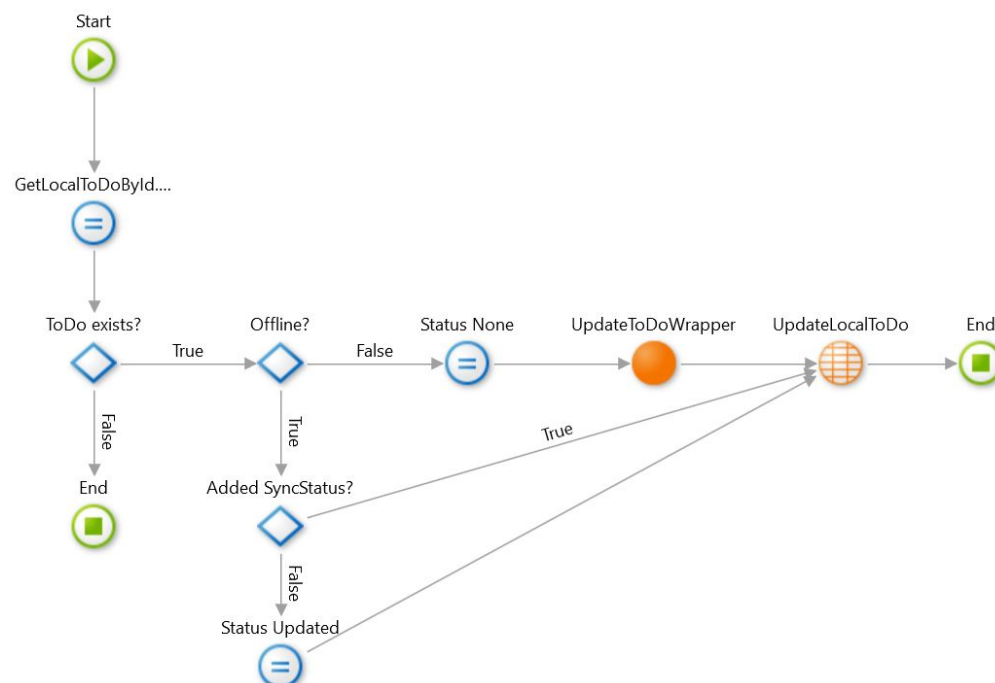
- g) Create the **True** branch connector from the **Added SyncStatus?** If to the **UpdateLocalToDo** statement.
- h) Drag an **Assign** statement and drop it below the **Added SyncStatus?** If, then create the **False** branch connector between both.
- i) Select the **Assign** statement created in the previous step. Set its **Label** as *Status Updated* and define the following assignment

```
GetLocalToDoById.List.Current.LocalToDo.SyncStatusId =
Entities.SyncStatus.Updated
```

- j) Create the missing connector from the **Assign** to the **CreateOrUpdateLocalToDo** statement.
- k) Drag a new **Assign** statement and drop it between the **Offline?** If and the **UpdateToDoWrapper**. Set its **Label** to *Status None* and define the following assignment

```
GetLocalToDoById.List.Current.LocalToDo.SyncStatusId =
Entities.SyncStatus.None
```

- l) Your **StarOnClick** Client Action should look like this



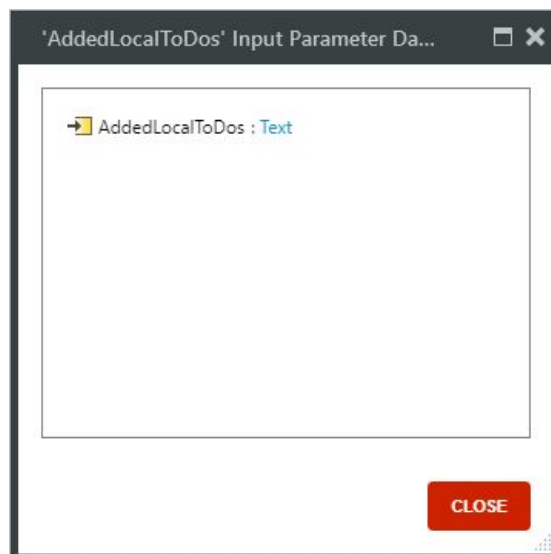
Syncing Todos and Resources

Now that we prepared the application to add Todos while offline, it is time to go back to the synchronization process and create the logic to synchronize Todos and their Resources, if they exist. At this point, no single Todo has a Resource, but the logic can easily be created already. This way, in the future lab where the Resources will be added to the Todos, the synchronization logic will be ready.

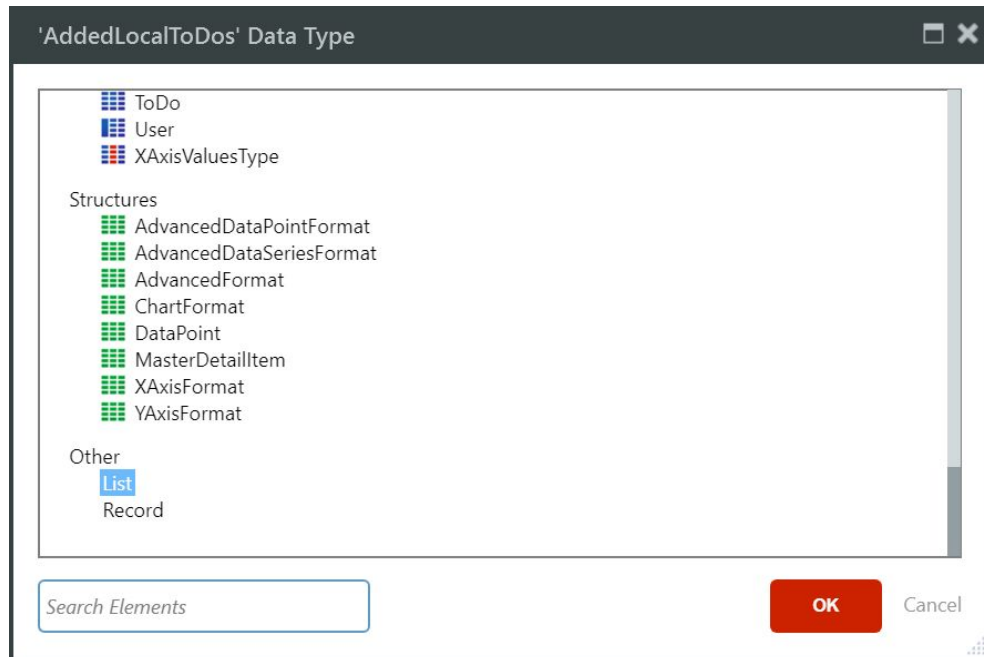
The logic for synchronization will work based on the Sync Status. The Todos that are marked as **Added** will be created in the database and the Todos marked as **Updated** will be updated in the database.

For that, we will change the **ServerDataSync** and the **OfflineDataSync** Actions, starting by the first one.

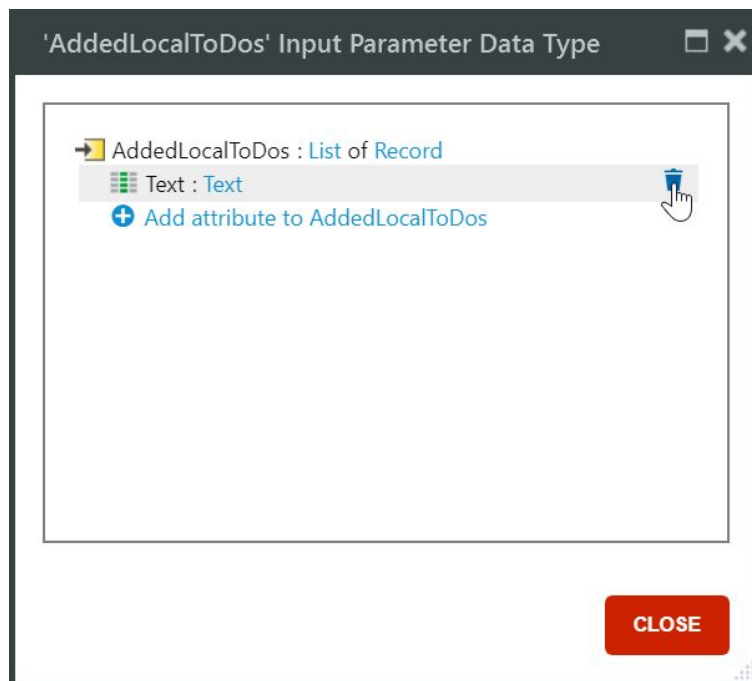
- 1) Create the Input Parameters required for **ServerDataSync**: *AddedLocalTodos*, with the List of Todos that were added in local storage, and *UpdatedLocalTodos*, with the List of Todos that were updated in the local storage.
 - a) In the **Logic** tab, locate and open the **ServerDataSync** Server Action under the **OfflineDataSync** folder of the Server Actions.
 - b) Add a new Input Parameter named *AddedLocalTodos*.
 - c) Double-click the **Data Type** property to edit the data type of the Input Parameter.
 - d) Click the **Text** data type to change it.



- e) Select *List* in the next dialog and then click **Ok**.

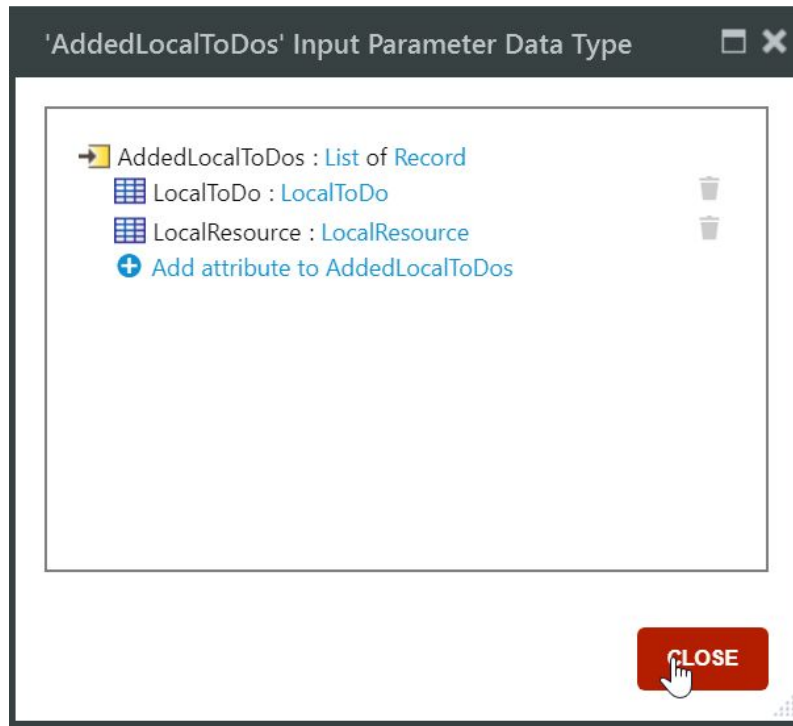


- f) Click the **Text** type again and then choose *Record* from the list of available types.
- g) Click the **trash** icon to remove the **Text** attribute.




- h) Click the **Add attribute to AddedLocalToDos**, then set the attribute name to *LocalToDo* and verify that the type has changed to **LocalToDo**.

- i) Click the **Add attribute to AddedLocalToDos** and set the new attribute name to *LocalResource*, then verify that its data type was set to **LocalResource**.
- j) Click **Close** to close the data type editor.



- k) Add another Input to the **ServerDataSync** Server Action named *UpdatedLocalToDos*, then define the **Data Type** as in the previous steps.
- 2) Create the logic in the **ServerDataSync**, for the Todos that were added in local storage, while the app was offline. For that, we will create a ToDo in the database for each LocalToDo in the **AddedLocalToDos** List.
- a) Open the **ServerDataSync** Server Action.
 - b) Drag a **For Each** statement and drop it under the **LogMessage** statement.
 - c) Drag a **Run Server Action** statement and drop it on the right of the For Each. In the **Select Action** dialog choose the **CreateToDoWrapper** Action.
 - d) Create the **Cycle** connector from the For Each to the CreateToDoWrapper.
 - e) Select the **For Each** and set the **Record List** property to the *AddedLocalToDos* Input Parameter.
 - f) Select the **CreateToDoWrapper** and set its Input Parameter accordingly, using the LocalToDos attributes from the AddedLocalToDos List. For instance, set the **Title** to *AddedLocalToDos.Current.LocalToDo.Title*

Do the same logic for the remaining Input Parameters of the Action.

 CreateToDoWrapper Run Server Action	
Name	CreateToDoWrapper
Action	PublicDBActions\CreateToDoWrapper
Title	AddedLocalToDos.Current.LocalToDo.Title
IsStarred	AddedLocalToDos.Current.LocalToDo.IsStarred
Notes	AddedLocalToDos.Current.LocalToDo.Notes
DueDate	AddedLocalToDos.Current.LocalToDo.DueDate
CreatedDate	AddedLocalToDos.Current.LocalToDo.CreatedDate
CompletedDate	AddedLocalToDos.Current.LocalToDo.CompletedDate
CategoryId	AddedLocalToDos.Current.LocalToDo.CategoryId
PriorityId	AddedLocalToDos.Current.LocalToDo.PriorityId


NOTE: The For Each allows iterating through all the LocalToDos in the **AddedLocalToDos** List. Then, within the loop we use the **CreateToDoWrapper** to add each one to the database.

- g) Drag an **If** statement and drop it below the **CreateToDo** statement, then create the connector between both.
- h) Set the **Label** property of the **If** to *Has Resource?*, and the **Condition** to

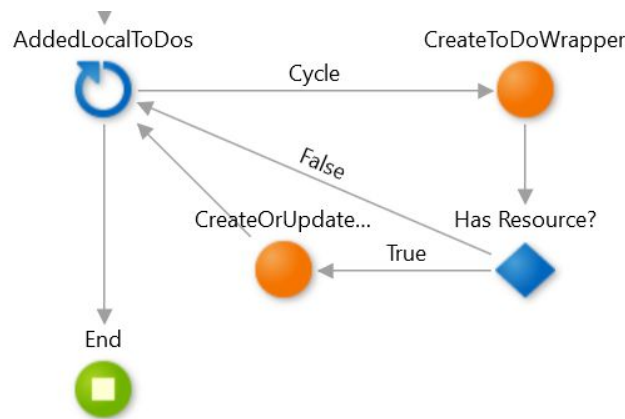
$$AddedLocalToDos.Current.LocalResource.Id \neq NullIdentifier()$$
- i) Drag another **Run Server Action** statement and drop it on the left of the **Has Resource? If**.
- j) In the **Select Action** dialog choose the **CreateOrUpdateResourceWrapper** Action and then create the **True** branch connector from the If to the Action.
- k) Set the Input Parameters of the **CreateOrUpdateResourceWrapper** accordingly, with the information stored in the **AddedLocalToDos** List. As an example, set the **Filename** to

AddedLocalToDos.Current.LocalResource.Filename

In the case of the **ToDoId**, set it to *CreateToDoWrapper.ToDoId*, the output of the CreateToDoWrapper Action, since the Resource has the same Id of the ToDo.


 CreateOrUpdateResourceWrapper Run Server Action	
Name	CreateOrUpdateResourceWrapper
Action	PublicDBActions\CreateOrUpdateResourceWrapper
ToDoId	CreateToDoWrapper.ToDoId
ResourceTypeId	AddedLocalToDos.Current.LocalResource.ResourceTypeId
Filename	AddedLocalToDos.Current.LocalResource.Filename
MimeType	AddedLocalToDos.Current.LocalResource.MimeType
BinaryContent	AddedLocalToDos.Current.LocalResource.BinaryContent

- l) Create the connector from the CreateOrUpdateResourceWrapper to the For Each, and the missing **False** branch connector from the **Has Resource?** If to the For Each.




This way, for every new ToDo added to local storage, we create the ToDo in the database, and if it has a Resource, we also add it to the database.

- 3) Now that the AddedLocalToDos List is handled with, we need to repeat the same logic for the **UpdatedLocalToDos** List. In this case, every single ToDo updated in the local storage, must be updated as well in the database.
 - a) Drag another **For Each** statement and drop it under the existing For Each. Set the **Record List** property to the *UpdatedLocalToDos* Input Parameter.
 - b) Drag a **Run Server Action** and drop it on the left of the **For Each** created in the previous step.
 - c) In the **Select Action** dialog choose the **UpdateToDoWrapper** Action.
 - d) Create the **Cycle** connector from the **For Each** to the **UpdateToDo** Entity Action.
 - e) Set the Input Parameters of the **UpdateToDoWrapper** accordingly, using the values in the **UpdatedLocalToDos** List.

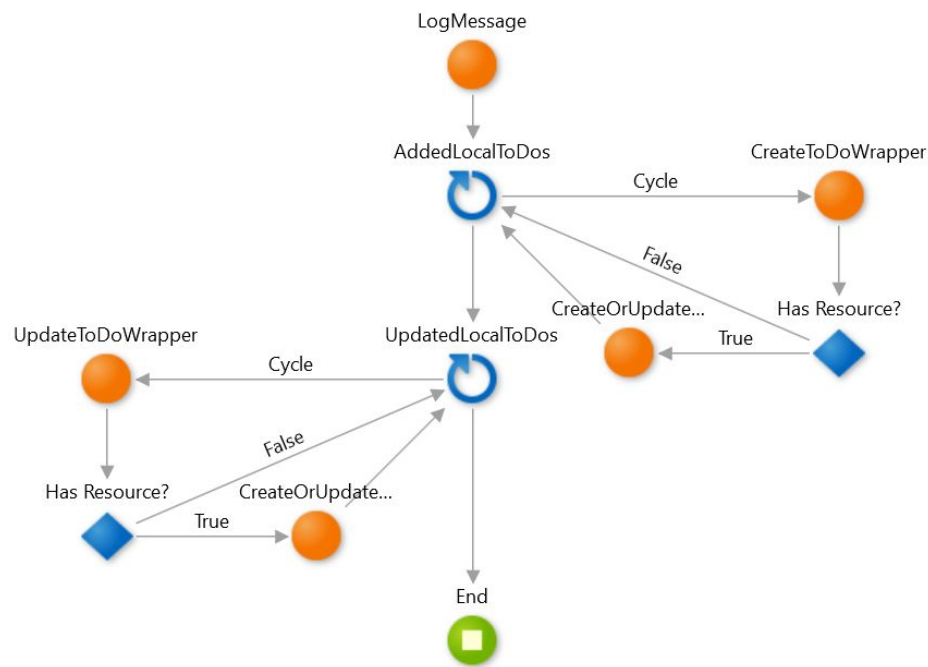
 UpdateToDoWrapper Run Server Action	
Name	UpdateToDoWrapper
Action	PublicDBActions\UpdateToDoWrapper
ToDold	UpdatedLocalToDos.Current.LocalToDo.Id
Title	UpdatedLocalToDos.Current.LocalToDo.Title
IsStarred	UpdatedLocalToDos.Current.LocalToDo.IsStarred
Notes	UpdatedLocalToDos.Current.LocalToDo.Notes
DueDate	UpdatedLocalToDos.Current.LocalToDo.DueDate
CreatedDate	UpdatedLocalToDos.Current.LocalToDo.CreatedDate
CompletedDate	UpdatedLocalToDos.Current.LocalToDo.CompletedDate
CategoryId	UpdatedLocalToDos.Current.LocalToDo.CategoryId
PriorityId	UpdatedLocalToDos.Current.LocalToDo.PriorityId

- f) Drag an **If** statement and drop it below the **UpdateToDoWrapper** statement, then create the connector between both.
- g) Set the **Label** property of the **If** to *Has Resource?*, and the **Condition** property to *UpdatedLocalToDos.Current.LocalResource.Id <> NullIdentifier()*
- h) Drag a **Run Server Action** statement and drop it on the right of the last **Has Resource?** If statement.
- i) In the **Select Action** dialog choose the **CreateOrUpdateResourceWrapper** Action.
- j) Create the **True** branch connector from the **Has Resource?** If to the CreateOrUpdateResourceWrapper2 statement.
- k) Set the Input Parameters of the **CreateOrUpdateResourceWrapper2** accordingly. Set the **ToDold** to *UpdatedLocalToDos.Current.LocalToDo.Id*

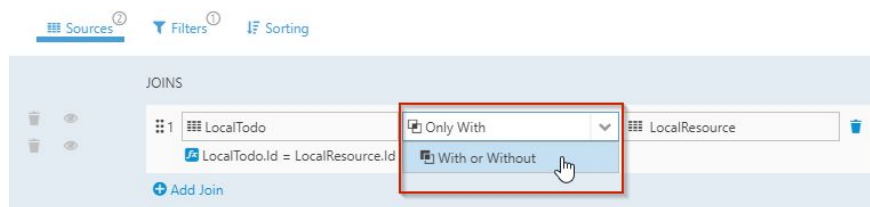
 CreateOrUpdateResourceWrapper2 Run Server Action	
Name	CreateOrUpdateResourceWrapper2
Action	PublicDBActions\CreateOrUpdateResourceWrapper
ToDold	UpdatedLocalToDos.Current.LocalToDo.Id
ResourceTypeId	UpdatedLocalToDos.Current.LocalResource.ResourceTypeId
Filename	UpdatedLocalToDos.Current.LocalResource.Filename
MimeType	UpdatedLocalToDos.Current.LocalResource.MimeType
BinaryContent	UpdatedLocalToDos.Current.LocalResource.BinaryContent

- l) Create the connector from the CreateOrUpdateResourceWrapper2 back to the For Each statement, and the missing **False** branch connector from the **Has Resource?** If to the For Each.

m) The modified part of **ServerDataSync** should look like this.



- 4) Modify the **OfflineDataSync** Client Action to send the changes made, while the device was offline, to the server. This logic will prepare the **AddedLocalToDos** and **UpdatedLocalToDos** Lists to send to the **ServerDataSync** Server Action, so it should be defined before the ServerDataSync Action call.
 - a) Open the **OfflineDataSync** Client Action located under the OfflineDataSync folder of the **Client Actions**.
 - b) Drag a new **Aggregate** and drop it between the Start and ServerDataSync.
 - c) Open the Aggregate, and from the **Data** tab, drag the **LocalToDo** Entity and drop it on the editor.
 - d) Repeat the previous step for the **LocalResource** Entity.
 - e) Rename the Aggregate to *GetAddedLocalToDos*
 - f) In the **Sources** tab, change the existing join clause between the **LocalToDo** and **LocalResource** to *With or Without*.



This will make the Aggregate fetch all the Todos, regardless if they have or not a Resource attached to it.

- g) Switch to the **Filters** tab and add the following filter

LocalToDo.SyncStatusId = Entities.SyncStatus.Added

This filter will guarantee that only the LocalTodos with Sync Status set as **Added**, will be fetched from local storage. The result of this Aggregate will be the **AddedLocalTodos** List.

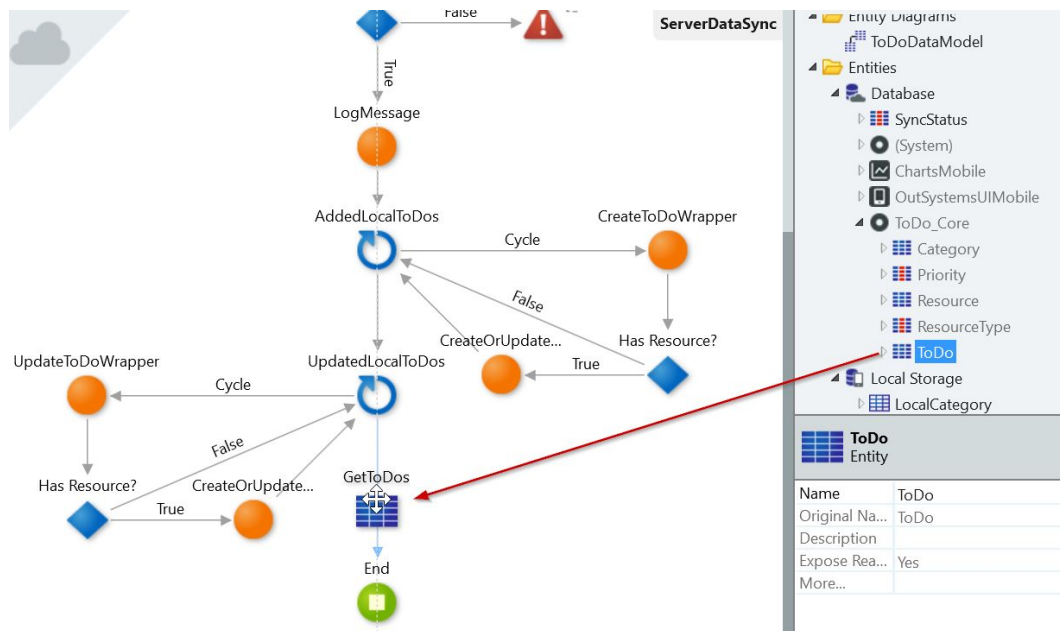
- h) Return to the OfflineDataSync Client Action.
- i) Drag another **Aggregate** and drop it between the GetAddedLocalTodos and the ServerDataSync statements.
- j) Rename it to *GetUpdatedLocalTodos* and set its **Sources** to be **LocalToDo** and **LocalResource**, with an *With or Without* join clause.
- k) In the Filters tab, add the following filter

LocalToDo.SyncStatusId = Entities.SyncStatus.Updated

This Aggregate has the same idea as the one above, but this time for the LocalTodos with Sync Status set as **Updated**.

- l) Return to the **OfflineDataSync** Client Action.
 - m) Select the **ServerDataSync** statement, then in the properties area set the **AddedLocalTodos** Parameter to *GetAddedLocalTodos.List* and the **UpdatedLocalTodos** Parameter to *GetUpdatedLocalTodos.List*.
- 5) Now that we prepared the logic to send the Todos added and updated to the server, the only thing left regarding the synchronization logic is to update the local storage with the data returned from the server. After the Todos and Resources are added / updated to the database, we must also update the local storage with the fresh data from the server, so that both sources of data are synchronized and match. For this to happen, the **ServerDataSync** must return all the Todos and Resources.
- a) Under the Logic tab, right-click the **ServerDataSync** and select *Add Output Parameter*.
 - b) Set the Output Parameter's **Name** to *LocalTodos* and make sure its **Data Type** is set to *LocalToDo List*.
 - c) Repeat the previous two steps and create the *LocalResources* Output Parameter, with *LocalResource List* as **Data Type**.
 - d) Open the **ServerDataSync** Action.

- e) From the **Data** tab, drag the **ToDo** Entity between the **UpdatedLocalTodos** For Each and the End node to create the *GetTodos* Aggregate.



- f) Open the **GetTodos** Aggregate and add the following filter to only get the Todos of the logged in user.

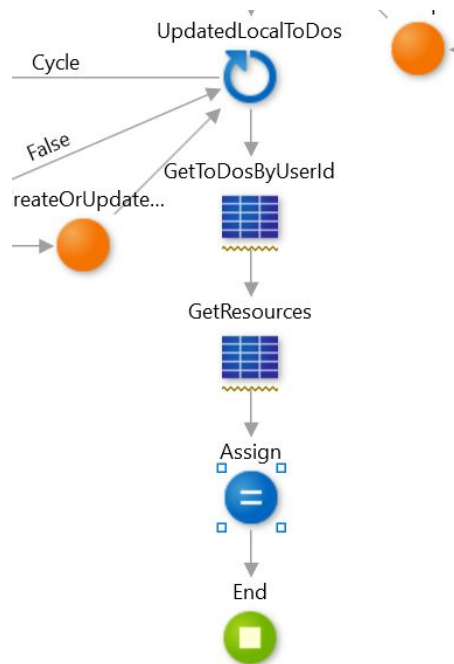
ToDo.UserId = GetUserId()

- g) Return to the **ServerDataSync** Action and drag and drop the **Resource** Entity below the **GetTodosByUserId** Aggregate.
- h) Open the **GetResources** Aggregate and also filter the Resources (and Todos) by the user currently logged in

ToDo.UserId = GetUserId()

NOTE: Recall that the join condition is set as **Only With**, which means that will return only Resources associated to To Dos of the logged in user.

- i) Return to the **ServerDataSync** Action, drag an **Assign** and drop it between the **GetResources** and the End.



j) Define the following assignments

LocalToDos = GetToDoByUserId.List

LocalResources = GetResources.List

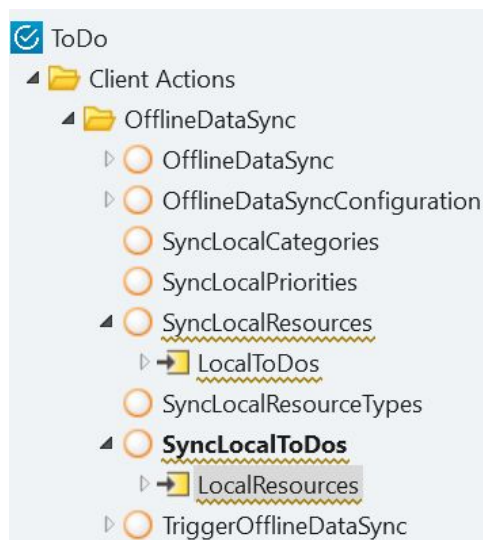
Make sure that the mappings between the Database Entities and the Local Storage Entities are done properly.

LocalToDos	▼		
= GetToDoByUserId.List	▼		
Mapping from ToDo to LocalToDo			
Id	ToDo.Id	▼	
Title	ToDo.Title	▼	
IsStarred	ToDo.IsStarred	▼	
Notes	ToDo.Notes	▼	
DueDate	ToDo.DueDate	▼	
CreatedDate	ToDo.CreatedDate	▼	
CompletedDate	ToDo.CompletedDate	▼	
CategoryId	ToDo.CategoryId	▼	
PriorityId	ToDo.PriorityId	▼	
SyncStatusId		▼	
		Assignments	
		LocalResources	▼
		= GetResources.List	▼
		Mapping to LocalResource	
		Id	Resource.Id
		ResourceTypeId	Resource.ResourceTypeId
		Filename	Resource.Filename
		MimeType	Resource.MimeType
		BinaryContent	Resource.BinaryContent

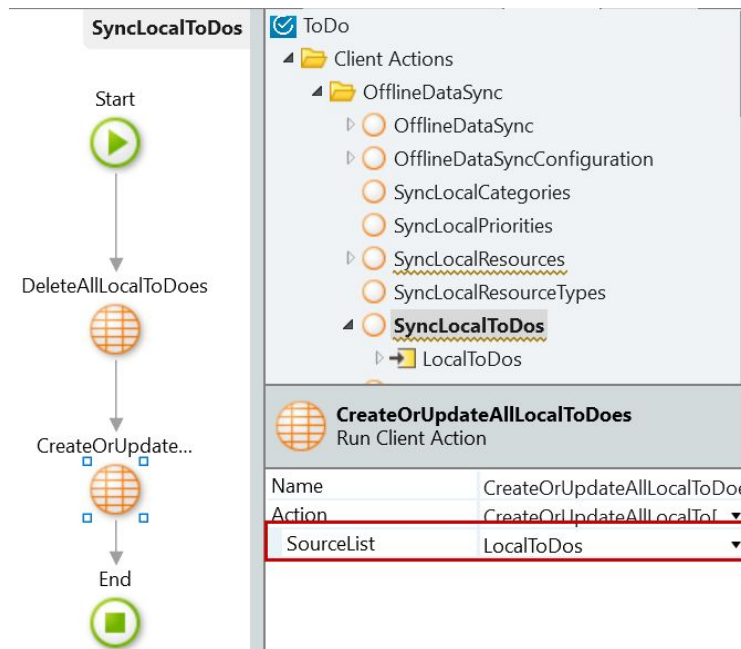
6) Now that we have the ToDos and the Resources from the server as Outputs of the ServerDataSync Action, we must use them in the **OfflineDataSync** to update the local storage with that information. This is already done for the Categories, Priorities and ResourceTypes, and we will manually create similar logic to the ToDos and Resources. To help us with that, we will create two Client Actions, *SyncLocalToDos* and

SyncLocalResources, that will clear the local storage of Todos and Resources respectively, before adding the Todos and Resources obtained from the server. The data from the server will be passed as Input Parameters to these two Actions. These two Actions will then be used in the *OfflineDataSync* to complete the synchronization logic.

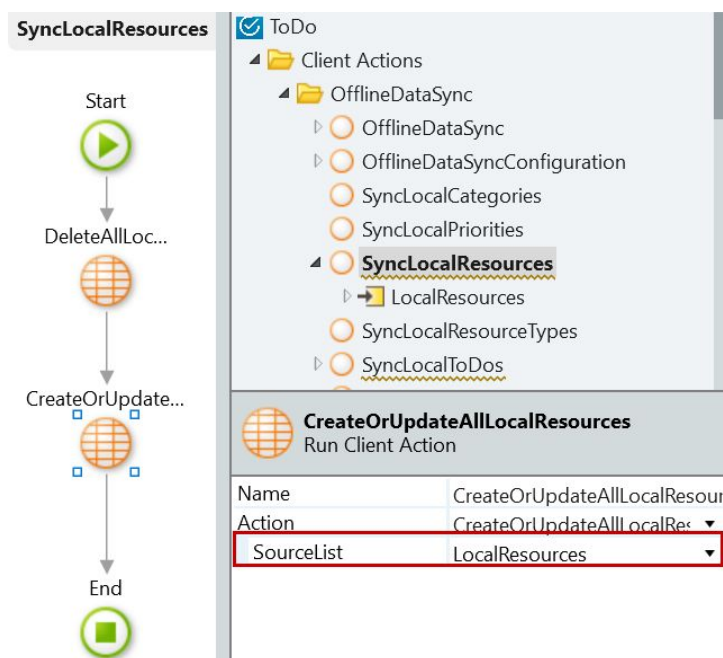
- Under the **OfflineDataSync** folder in the Client Action, create the *SyncLocalTodos* Client Action.
- Right-click the new Action and select *Add Input Parameter*. Set its **Name** to *LocalTodos* and make sure its **Data Type** is set to *LocalToDo List*.
- Repeat the previous two steps and create the Client Action *SyncLocalResources*, with an Input *LocalResources*, with **Data Type** *LocalResource List*.



- Open the **SyncLocalTodos** Action and drag and drop a **Run Client Action** to its flow. In the next dialog, select the **DeleteAllLocalToDoes** Entity Action. This will guarantee that all the Todos are deleted, before creating the new ones with the data coming from the server.
- Drag a new **Run Client Action** and drop it after the previous one. Select the **CreateOrUpdateAllToDoes** Entity Action and pass it as **Source** the *LocalTodos* Input Parameter.



f) Repeat the previous steps for the **SyncLocalResources** Action.



- g) Open the **OfflineDataSync** Action.
- h) Drag and drop the **SyncLocalTodos** Action to the **OfflineDataSync** flow, right before the End statement. Set the **LocalTodos** Input Parameter to *ServerDataSync.LocalTodos*. This way, we are passing the new Action the Todos that were returned from the server.

- i) Drag and drop the **SyncLocalResources** Action to the **OfflineDataSync** flow, right before the End statement. Set the **LocalResources** Input Parameter to *ServerDataSync.LocalResources*.
- j) Drag a **Message** statement and drop it just before the End.
- k) Set the **Message** parameter to *"Server data synced to Local Storage."* and the **Type** to *Info*.
- l) The **OfflineDataSync** Client Action should look like this

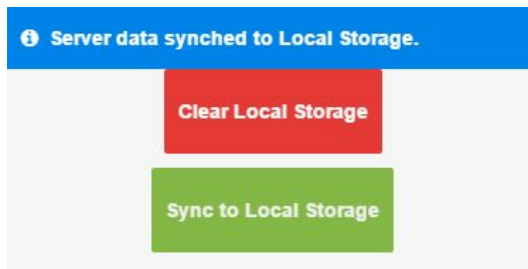


Testing the app: Synchronization and offline interaction

In this part of the exercise, it is time to make sure that the synchronization and the offline interaction works properly. So, we will first synchronize the data with the server and guarantee that all the Todos we add before will be synced and appear in the Todos Screen.

Then, we will make the app go offline and add a new Todo and update an existing one. Then, we will turn the app back online, trigger the synchronization and make sure that the changes made in local storage are synchronized properly.

- 1) Click the **1-Click Publish** button to publish the module to the server.
- 2) Open the native application on the device, or use OutSystems Now to do it.
- 3) Navigate to the **Data Management** Screen, then click the **Sync to Local Storage** Button.
- 4) After some seconds, you should see the synchronization feedback message.

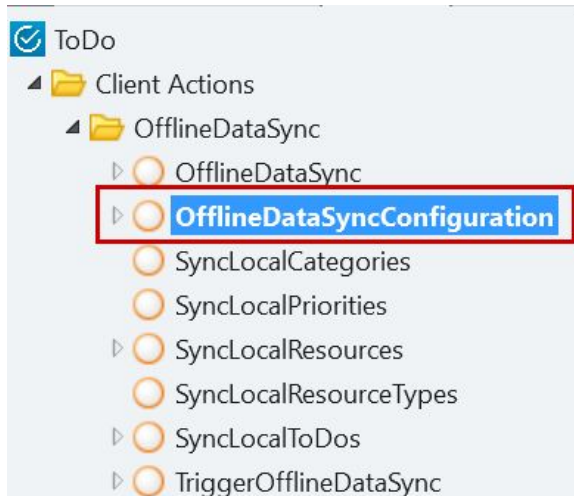


- 5) Navigate to the **Todos** Screen and verify that the To Dos appear.
- 6) Now make sure that you turn off all the networks in the device to make the app offline.
- 7) Create a new Todo and update an existing one.
- 8) Then, turn the app back online.
- 9) Navigate back to the **DataManagement** Screen and trigger the synchronization. Make sure that the new data appear properly on the Todos Screen.
- 10) Clear the local storage and trigger the synchronization again to make sure that the data was properly coming from the server.

Automatic Synchronization

Now that the synchronization logic is defined, and the application works offline, we can tweak it even more to make sure that the synchronization is also triggered automatically. For that, we will make sure the synchronization is triggered when the app gets back online, when the user logs in and when the app is resumed.

- 1) Under the **Logic** tab, open the **OfflineDataSyncConfiguration** Client Action.



- 2) Select the **SyncOnline** Assign statement. Change the assignment to

SyncOnline = True

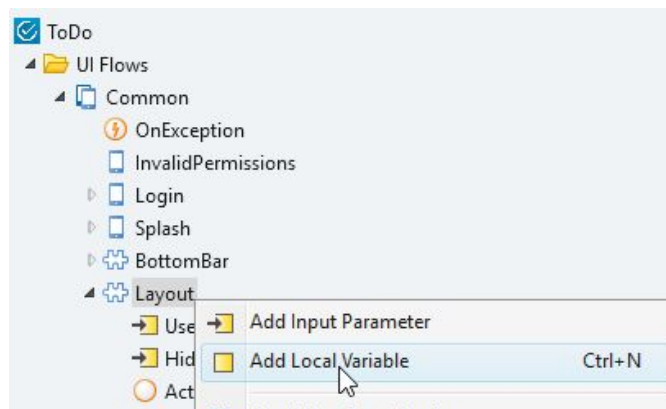


- 3) Change the **SyncOnLogin** and **SyncOnResume** Assigns to *True*, just like in the previous step.
- 4) Select the **RetryOnError** Assign and change the assignment to
RetryOnError = True
- 5) Publish the module to the server.

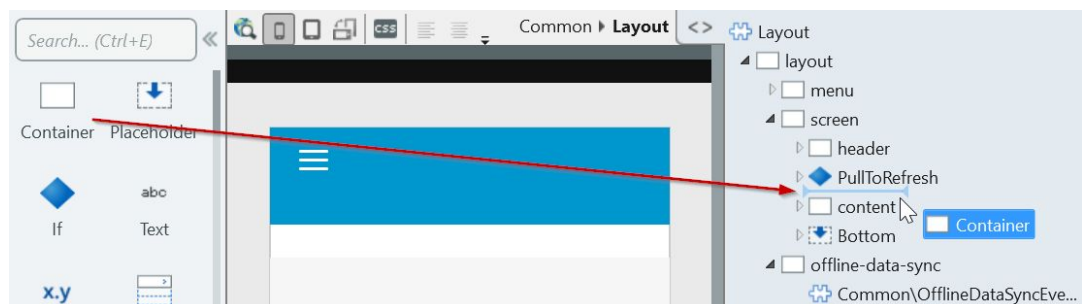
Is Syncing Feedback

To finalize this lab, we want to give the user some feedback when the synchronization is happening. For that, we will add a **Syncing...** message to every Screen in the application.

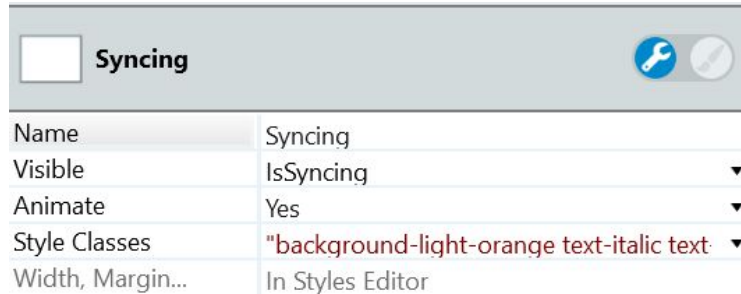
- 1) Adjust the Layout Block of the application to hold a visual feedback message at red, with yellow background, displaying **Syncing...** This message should appear in the header of all Screens.
 - a) Switch to the **Interface** tab and open the **Layout** block.
 - b) Add a new **Local Variable** to the Layout Block, name it *IsSyncing* and make sure its **Data Type** is set to *Boolean*.



- c) Drag a **Container** and drop it just above the **content** Container, and at the same level as the **PullToRefresh** If Widget.



- d) Set its **Visible** property to *IsSyncing* and **Name** the Container *Syncing*.
 - e) Align the Container to the center and set its **Style Classes** to *"background-light-orange text-italic text-dark-red"*



- f) Drag an **Icon**, drop it inside the Container and choose the *refresh* icon.
 - g) Set the **Size** property of the Icon to *Font size*.
 - h) Click to the right of the icon and type *Syncing...* (with an empty space before).
- 2) Now, we need to make sure that all the Screens display this message when the synchronization is happening. We already set the **Visible** property to the **IsSyncing** Variable, so right now we need to set the IsSyncing Variable to *True*, when the synchronization starts, and to *False* when it is not occurring.
 - a) Open the **ActionHandler_OnSyncStartTrigger** Client Action under the **Layout** Block.
 - b) Add an **Assign** statement just after the Start, and define the following assignment

IsSyncing = True

Since this Action runs when the synchronization starts, triggered by an Event, we make sure that the **IsSyncing** variable is set to *True*.
 - c) Open the **ActionHandler_OnSyncCompleteTrigger** Client Action.
 - d) Add an **Assign** statement just after the Start, and define the following assignment

IsSyncing = False

Following the same logic, since this Action runs when the sync is completed, the **IsSyncing** Variable must be set to *False*.
 - e) Open the **ActionHandler_OnSyncErrorTrigger** Client Action and add an **Assign** statement as the one before.
 - f) Publish the module to save the changes on the server.

Testing the app: automatic sync and feedback

In this part of the exercise, we will test the application and the automatic synchronization triggers. Also, we need to make sure that the feedback created in the previous section properly appears on the Screens.

- 1) Open the application on the device and logout.
- 2) Login again with your user. Make sure that the message indicating the data synchronization was done appears.
- 3) Navigate to the **DataManagement** Screen and clear the local storage.
- 4) Then, make the app offline.
- 5) Turn on again the device network and make sure the **Syncing...** message appears again on the device.
- 6) Navigate to the ToDos Screen and guarantee that all the ToDos appear.

End of Lab

In this exercise, we defined the offline interaction and data synchronization logic for the application.

After this exercise, the application works either online or offline, meaning that Todos can be created / updated when the application is offline. When it is online, it will work as before and send the data immediately to the server. When offline, it will only save the Todos in the local storage, and then save a status of the changes made to the Todos, which was later useful for the synchronization.

Regarding the synchronization logic, we defined a Read-only approach for the Priorities, Categories and ResourceTypes, using Service Studio accelerators. For the Todos and Resources, we manually defined a Read / Write behavior, using the information saved while offline.

We also defined ways to trigger the synchronization. A manual option by defining a new Screen to help end-users to trigger it by clicking on a Button, and an automatic option by setting the automatic triggers OnLogin, OnOnline and OnResume on the OfflineDataSyncConfiguration Action.

Finally, we defined a feedback message for the users to know when the data is being synchronized.