

Plugins Exercise

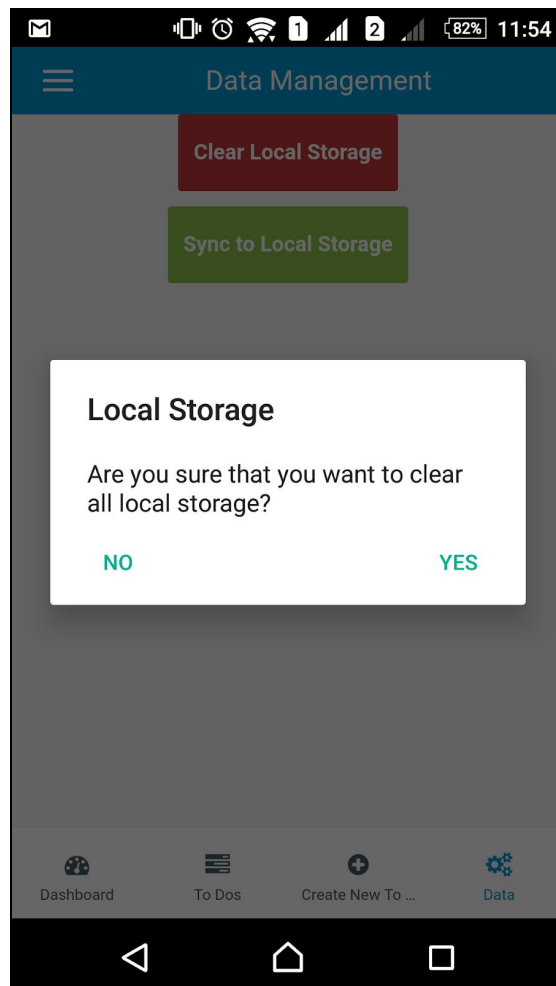


Table of Contents

Table of Contents	2
Introduction	3
Install Plugin from the Forge	4
Attach Camera Pictures to To Dos	5
Create a new Plugin	20
Test the Dialogs Plugin	27
End of Lab	33

Introduction

Over the course of this set of exercise labs, we created an application to manage ToDos, with four Screens, which includes local storage, data synchronization and Dashboards.

This final lab will extend the functionality of the application to use Plugins.

First, we will install the Camera Plugin from the Forge in our development environment, to use it in the ToDos application. In the ToDoDetail Screen, we will add the functionality to take a picture and attach it to a ToDo. On that Screen, if the ToDo has a Resource, it will be displayed below the Form with the remaining ToDo attributes. We will also allow the user to replace the picture if desired.

Then, we will create our own plugin, in a different application: the Dialogs Plugin. This will enable creating custom dialogs for alerting the end-user or to ask for the confirmation for an operation. To build a new Plugin, we will use JavaScript code, encapsulated in Client Actions that can be easily reused in logic flows, the OutSystems way: by dragging and dropping. The module of the new application needs to reference the Cordova Plugin repository to make sure that the JavaScript code can be used seamlessly.

This plugin will be used in the ToDos application,

As a summary, In this specific exercise lab, we will:

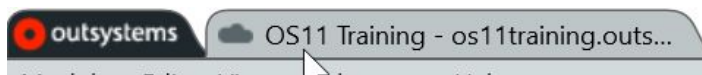
- Install the **CameraPlugin** from Forge
- Attach pictures taken with the device camera to To Dos
- Create a new plugin
- Test the plugin in the **ToDo** application

Install Plugin from the Forge

We will start this exercise lab by installing an existing plugin from the Forge. We can download a component from the [Forge](#) website, or install it directly in Service Studio. In this section, we will do the latter.

For this exercise, we will install the Camera Plugin. **NOTE:** If you are in a classroom Boot Camp, you may skip this Part of the exercise.

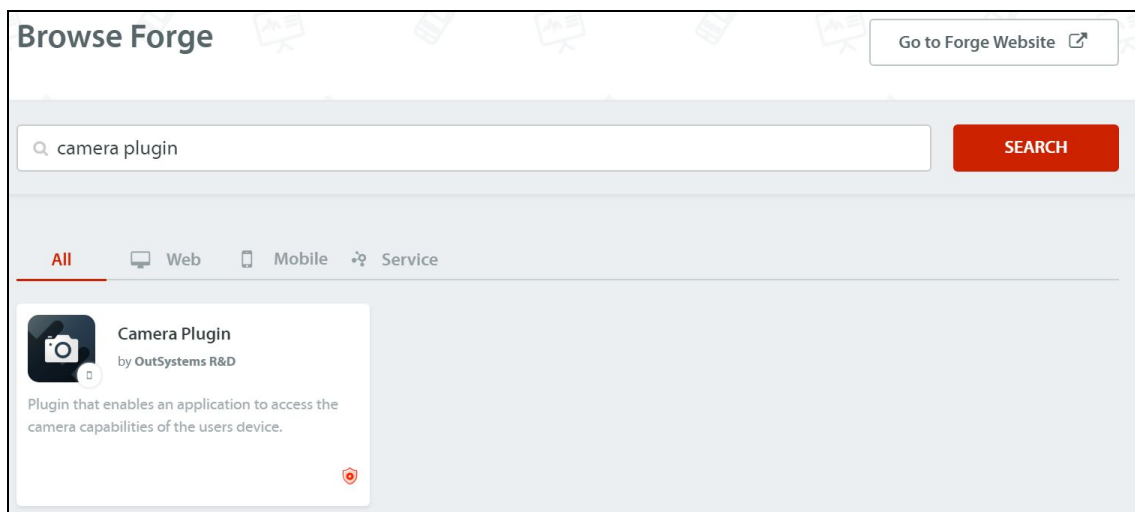
- 1) In Service Studio, switch back to the **Applications** tab.




- 2) Click the **Install Application** item to open the Forge.



- 3) Search for the **Camera Plugin** and open it.



- 4) Click the **Install** Button to install the component. If the component asks to install the dependencies, install them as well.





Camera Plugin

Stable version **3.0.0** (OutSystems 11)

Published on **26 September 2018** by [OutSystems R&D](#)


INSTALL


 Mobile



11 ratings

0 reviews

 **5262**

 **91**

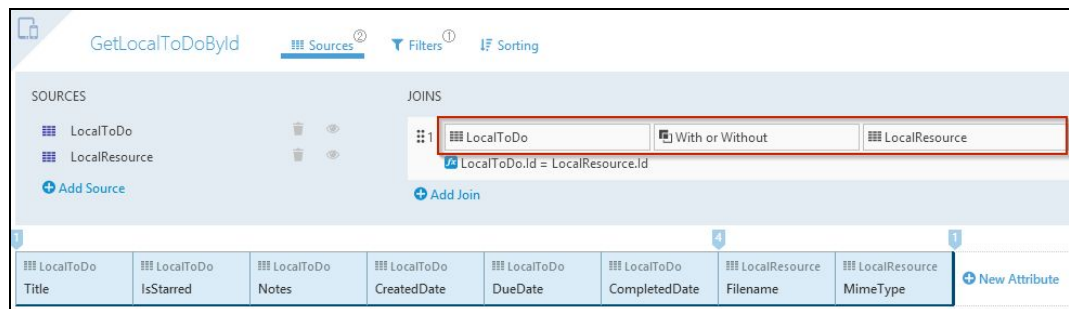
5) In the applications tab, in Service Studio, you can see the installation progress.



Attach Camera Pictures to To Dos

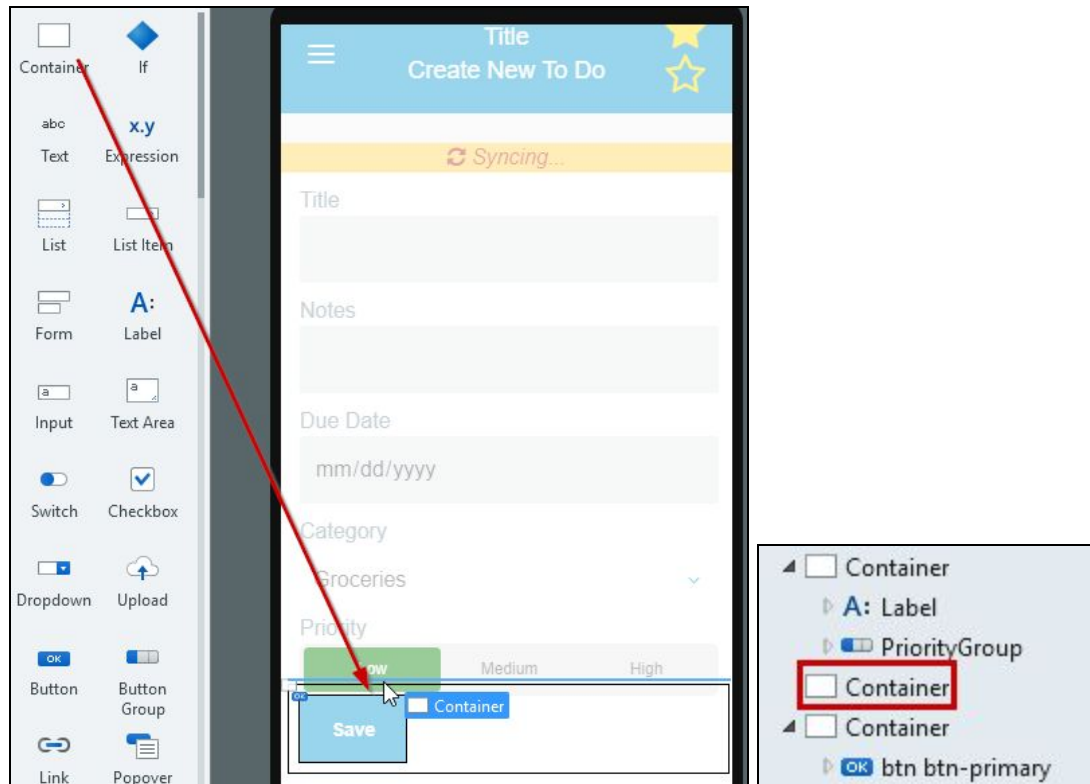
In this part of the exercise, we will modify the **ToDoDetail** Screen so that it's possible to attach pictures taken with the **Camera** Plugin. This will have two main steps. First, we will modify the Screen to display the picture, when it exists, and a message when it doesn't. Then, we will define an Action triggered on click, that will take the picture, create the resource in local storage, and if the app is online, also create it in the database.

- 1) Modify the **GetLocalToDoById** Aggregate of the **ToDoDetail** Screen to also retrieve the attached LocalResource, if it exists. Make sure that the Aggregate returns the Todos, with or without Resources.
 - a) Open the **GetLocalToDoById** of the **ToDoDetail** Screen.
 - b) From the **Data** tab, drag the **LocalResource** Entity and drop it inside the Aggregate editor.
 - c) Open the **Sources** tab and change the **Join** to *With or Without*.

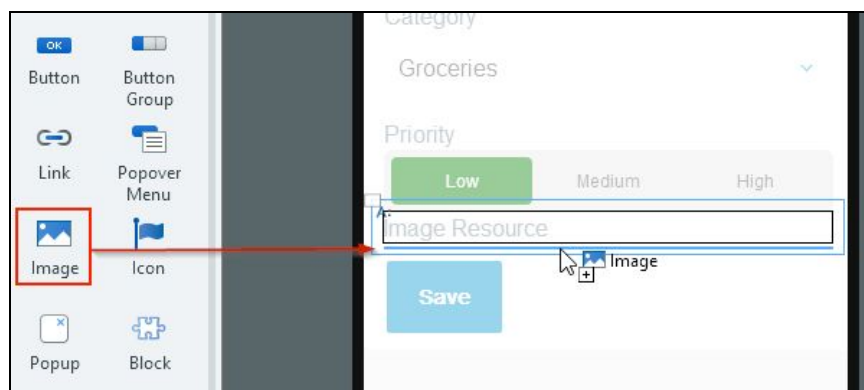


NOTE: Recall that Todos do not require to have attached resources, and therefore the join clause has to be changed to *With or Without*. If the join clause was left as *Only With*, we would only retrieve the To Do details for To Dos with an attached resource.

- 2) On the **ToDoDetail** Screen, display the image resource of a ToDo, if the ToDo has one. The image should be displayed right after the Priority ButtonGroup and before the Save Button.
 - a) Open the **ToDoDetail** Screen.
 - b) Drag a **Container** and drop it between the Priority input and the Save Button.



- c) Drag a new **Label** Widget and drop it inside the Container created in the previous step. Change the **Text** inside it to **Image Resource**.
- d) Drag an **Image** Widget and drop it below the Label, but still inside the surrounding Container.

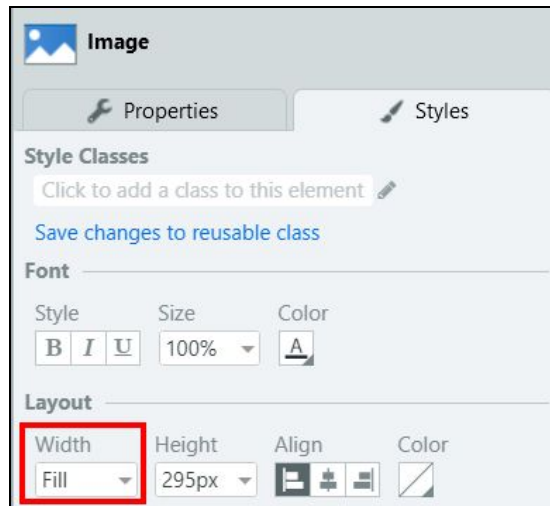


- e) Set the **Type** property of the Image to *Binary Data*, then set the **Image Content** property expression to

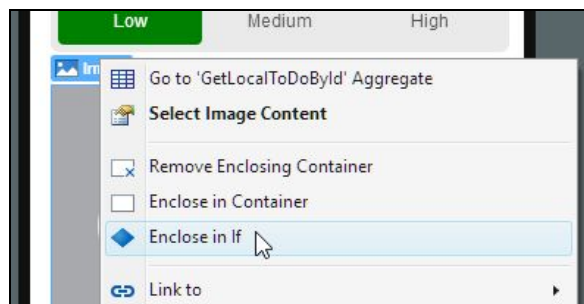
GetLocalToDoById.List.Current.LocalResource.BinaryContent

This will make sure that the Image will display the Binary Content of the Resource attached to the ToDo being edited in the **ToDoDetail** Screen.

- f) On the Styles Editor, set the **Width** of the Image to *Fill*.



- g) Right-click the Image then choose *Enclose in If*.



- h) Set the **Condition** property of the If to

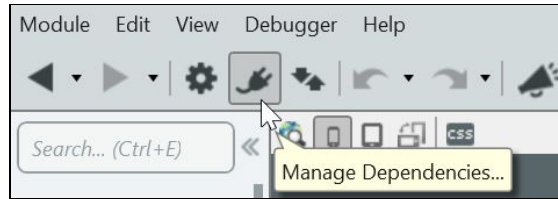
`GetLocalToDoById.List.Current.LocalResource.Id <> NullIdentifier()`

and

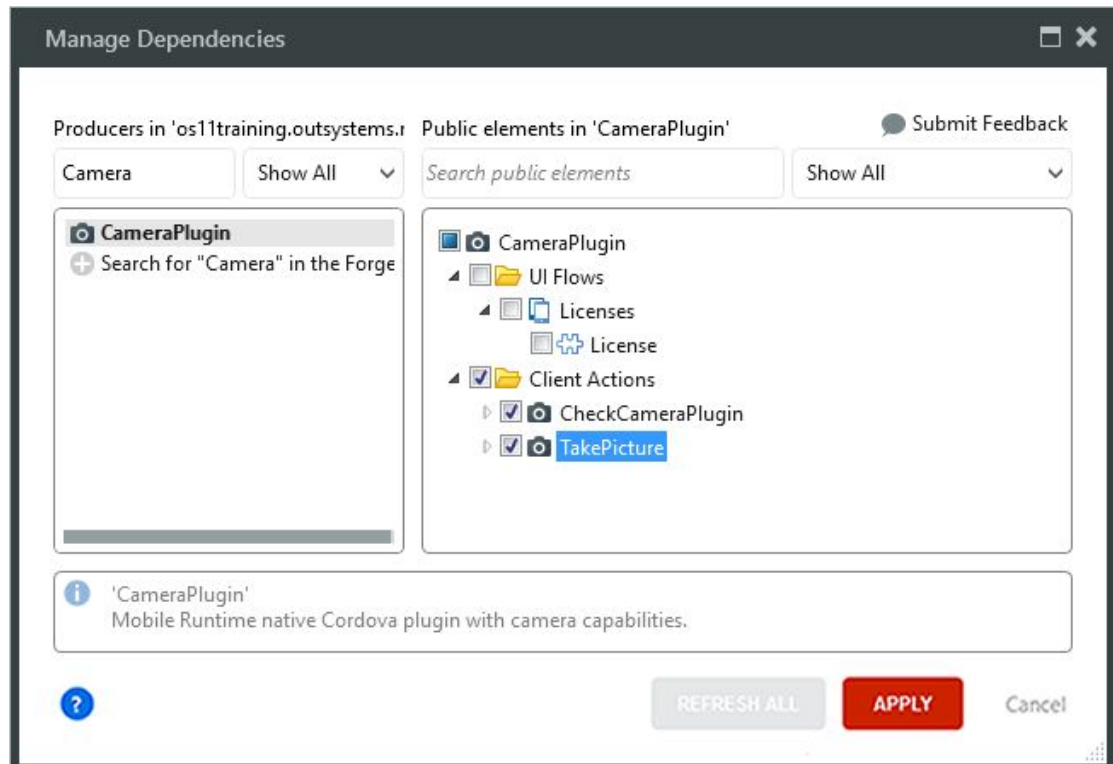
`GetLocalToDoById.List.Current.LocalResource.ResourceTypeId = Entities.ResourceType.Image`

This Condition will make sure that the Image only appears if the ToDo has a resource, and if it has, that it is an image.

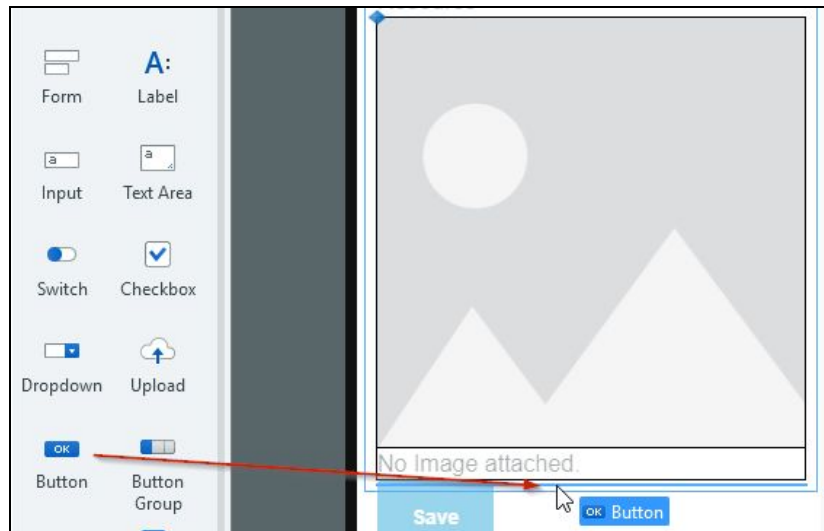
- i) In the **False** branch of the If, type *No Image attached.* and align the text to center.
- 3) Add a reference to the Camera Plugin Client Actions in the ToDo module.
 - a) Open the **Manage Dependencies** dialog by clicking the icon in the toolbar.



- b) On the list of producers on the left, select the **CameraPlugin**, then on the right tick the **CheckCameraPlugin** and **TakePicture** Client Actions.



- c) Click **Apply** to close the Manage Dependencies dialog.
- 4) Create the **Take Picture** Button, right below the If created above. Make sure that it is only visible if the ToDo exists in the local storage and it is not completed yet.
- a) In the **ToDoDetail** Screen, drag a **Button** Widget and drop it just below the If for the Image, but still inside the surrounding Container.



- b) Change the **Text** inside the Button to *Take Picture*.
- c) Drag an **Icon** Widget and drop it inside the Button, to the left of the text, then choose the *camera* icon in the **Pick an Icon** dialog.
- d) Change the **Size** property of the Icon to *Font size*.
- e) Select the **Button** and change the Style Classes property to
"btn btn-small btn-danger"
- f) Align the **Button** to center.



- g) Change the **Visible** property of the **Button** to

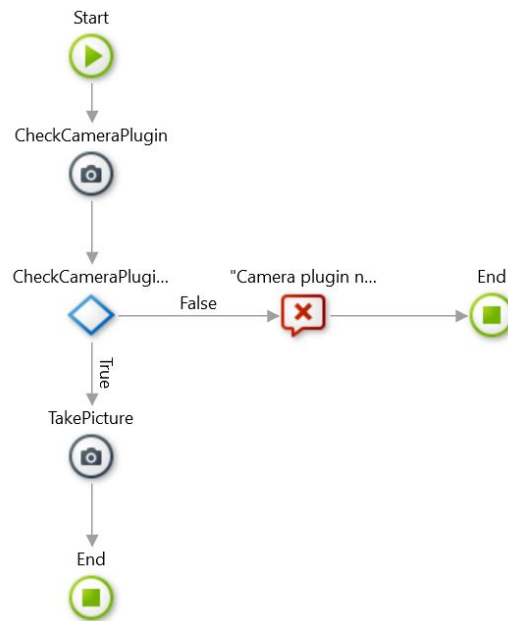
*GetLocalToDoById.List.Current.LocalToDo.Id <> [NullIdentifier\(\)](#) and
GetLocalToDoById.List.Current.LocalToDo.CompletedDate = [NullDate\(\)](#)*

NOTE: In order to attach a resource to a To Do, the To Do must exist. The condition on the **Visible** property will prevent users from attaching an image resource a new To Do that has not been saved yet, and also when the To Do was already marked as completed.

- 5) Now we need to create the logic to attach an image, taken with the device camera, to an existing To Do. This logic is a bit complex, so we will divide this in several steps. First, we will create the new Client Action that will run when the **Take Picture** Button is clicked, and create the logic to check if the Camera Plugin is available. If it is, the picture is taken using the **TakePicture** Camera Action.
- Double-click the Button to create a new Action and bind it to the **On Click** Event.
 - Drag a **Run Client Action** statement and drop it between the Start and the existing If. Select the *CheckCameraPlugin* in the **Select Action** dialog.
 - Delete the existing Comment, then select the **If** statement and change the **Condition** property to

CheckCameraPlugin.IsAvailable

- d) Drag a **Message** statement and drop it in the False branch connector.
- e) Set the **Type** property of the Message to *Error* and the **Message** property to *"Camera plugin not available."*.
- f) Drag a **Run Client Action** statement and drop it on the True branch connector, then choose the *TakePicture* Client Action from the **CameraPlugin** module.



- 6) The next step to build this logic is to guarantee that the step of taking a picture was indeed successful. If it was, then we can attach the resource to the *ToDo*, by setting the **LocalResource** record properly. If not, we should display an error message.
 - a) Drag an **If** statement and drop it between the *TakePicture* and *End*.
 - b) Set the **Condition** property of the new If to

TakePicture.Success

This Condition uses the *Success* output parameter of the *TakePicture* Action, to make sure that the action of taking the picture was indeed successful.

- c) Drag a **Message** and drop it on the False branch of the new If.
- d) Set the **Type** of the Message to *Error* and the **Message** to

TakePicture.Error.ErrorMessage

- e) Drag an **Assign** statement and drop it to the right of If statement. Then, create the *True* branch connector from the If to the Assign statement.

- f) Define the following assignments in the new Assign

```
GetLocalToDoById.List.Current.LocalResource.Id
= GetLocalToDoById.List.Current.LocalToDo.Id
```

```
GetLocalToDoById.List.Current.LocalResource.ResourceTypeId
= Entities.ResourceType.Image
```

```
GetLocalToDoById.List.Current.LocalResource.Filename
= "camera" + CurrDateTime() + ".jpg"
```

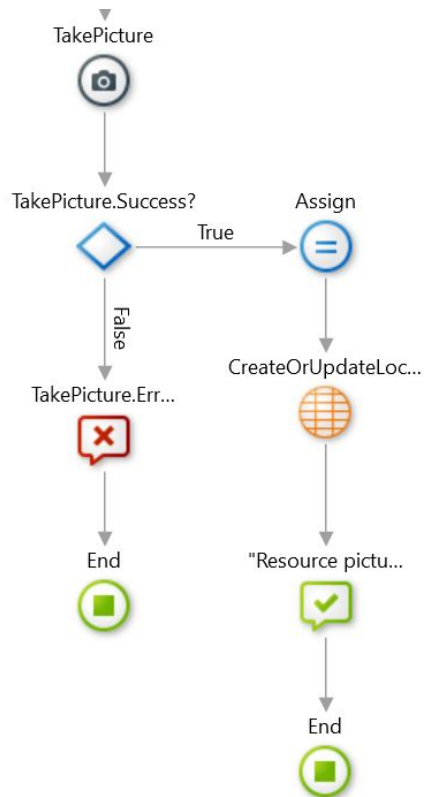
```
GetLocalToDoById.List.Current.LocalResource.MimeType
= "image/jpg"
```

```
GetLocalToDoById.List.Current.LocalResource.BinaryContent
= TakePicture.ImageCaptured
```

- g) Drag the **CreateOrUpdateLocalResource** Entity Action, from the Data tab, and drop it below the Assign. Then, create the True connector between both.
- h) Set the **Source** of the CreateOrUpdateLocalResource Entity Action to

```
GetLocalToDoById.List.Current.LocalResource
```

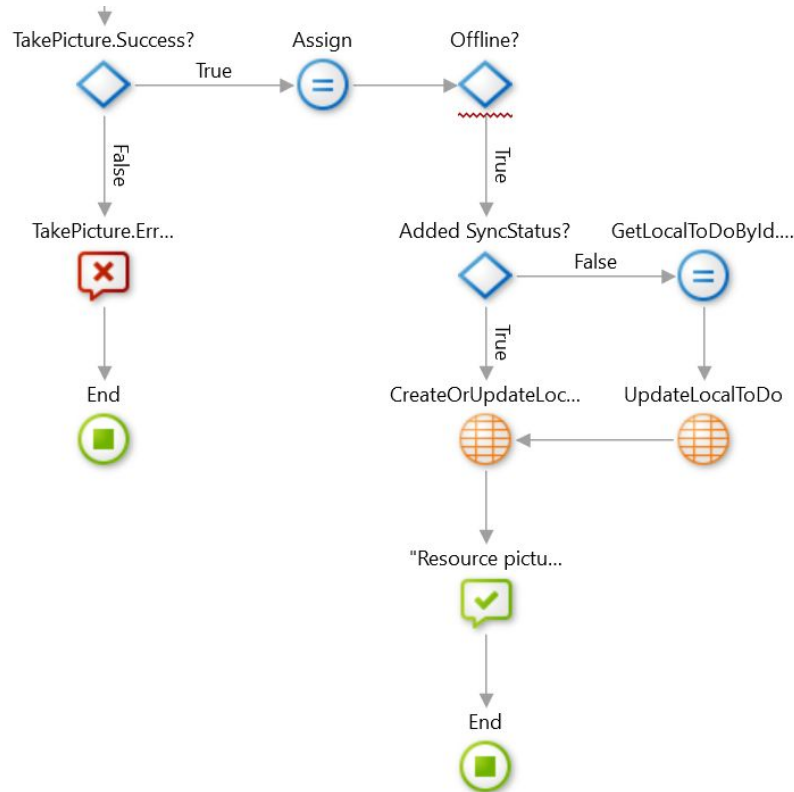
- i) Drag a **Message** statement and drop it below the **CreateOrUpdateLocalResource** Entity Action, then connect both.
- j) Set the **Type** of the Message to Success and the **Message** property to *"Resource picture saved."*.
- k) Drag an **End** node and drop it below the Message, then connect both elements.



- 7) At this point, we need to take into account the offline scenario. When the app is offline, the resource needs to be created in the local storage, but also we need to check and set appropriately the sync status of the respective **LocalToDo**. So, we need to first check if the app is offline. If it is, we have a special scenario that may happen. We may find a scenario where the app was offline, a ToDo was created and its sync status was set to Added / Updated. Then, the app came back online, the sync was triggered (if configured that way) and now we are taking a picture, while the synchronization is not over yet. To avoid adding Resources, without the Todos existing in the database, we need to also make sure the ToDo is there and it is updated. Then, the **LocalResource** must be created / updated in local storage.
- Drag an **If** statement and drop it between the Assign and the CreateOrUpdateLocalResource Action.
 - Adjust the flow by dragging the new If next to the Assign. Don't forget to also select the other nodes below the If, to make sure that the flow is aligned.
 - Set the If **Label** to *Offline?* and the **Condition** to

not GetNetworkStatus()

- d) Right-click on the If statement and select *Swap Connectors*. The existing flow has the logic that will be followed if the app is offline.
- e) Drag another **If** statement and drop it below the *Offline?* If, before the *CreateOrUpdateLocalResource* Action.
- f) Set the **Label** to *Added SyncStatus?* and the **Condition** of the new If to
`GetLocalToDoById.List.Current.LocalToDo.SyncStatusId =
 Entities.SyncStatus.Added`
- g) Right-click on the **Added SyncStatus?** If and select *Swap Connectors* to make the path down the *True* branch. If the ToDo has the Added status, it means that it is already in local storage.
- h) Drag an **Assign** and drop it to the right of the **Added SyncStatus?** If, then create the False branch connector from the If to the Assign. Define the following assignment
`GetLocalToDoById.List.Current.LocalToDo.SyncStatusId =
 Entities.SyncStatus.Updated`
- i) Drag the **UpdateLocalToDo** Entity Action and drop it below the Assign, then create the connector between both. Since, the status of the ToDo is *Updated*, we need to make sure that the ToDo is up to date in the local storage, before creating the resource associated with it.
- j) Set the **Source** of the Entity Action to
`GetLocalToDoById.List.Current.LocalToDo`
- k) Create a connector between the *UpdateLocalToDo* and the *CreateOrUpdateLocalResource*.



- 8) Finally, on the online scenario, we need to check the sync status of the ToDo. If it is different than *None*, we need to update it or add it in the database, depending on its Sync Status. Otherwise, we just need to create / update the resource. After the resource is added to the database, the flow must proceed to the local storage logic created in the previous step.
 - a) Drag another **If** and drop it to the right of the *Offline?* If, then create the False branch connector from the *Offline?* If to the new If.
 - b) Set the **Label** property of the new If to *SyncStatus <> None?* and the **Condition** to
`GetLocalToDoByld.List.Current.LocalToDo.SyncStatusId <> Entities.SyncStatus.None`
 - c) Drag one more **If** to the right of the **SyncStatus <> None?** And create the *True* branch between both Ifs.
 - d) Set the **Label** property of the new If to *SyncStatus = Updated?* and the **Condition**
`GetLocalToDoByld.List.Current.LocalToDo.SyncStatusId = Entities.SyncStatus.Updated`
 - e) Drag a **Run Server Action** and drop it to the right of the If created just before, then select the *UpdateToDoWrapper*

- f) Create the *True* branch connector from the **If** to the **UpdateToDoWrapper**.
- g) Set the Input parameters of the **UpdateToDoWrapper** as follows
 - ToDoId:** *GetLocalToDoById.List.Current.LocalToDo.Id*
 - Title:** *GetLocalToDoById.List.Current.LocalToDo.Title*
 - IsStarred:** *GetLocalToDoById.List.Current.LocalToDo.IsStarred*
 - Notes:** *GetLocalToDoById.List.Current.LocalToDo.Notes*
 - DueDate:** *GetLocalToDoById.List.Current.LocalToDo.DueDate*
 - CreatedDate:** *GetLocalToDoById.List.Current.LocalToDo.CreatedDate*
 - CompletedDate:** *GetLocalToDoById.List.Current.LocalToDo.CompletedDate*
 - CategoryId:** *GetLocalToDoById.List.Current.LocalToDo.CategoryId*
 - PriorityId:** *GetLocalToDoById.List.Current.LocalToDo.PriorityId*
- h) Drag another **Run Server Action** and drop it below the **SyncStatus = Updated?**. Select the *CreateToDoWrapper* Action.
- i) Connect the *False* branch of the *SyncStatus = Updated?* If to the **CreateToDoWrapper** Action.
- j) Set the Input parameters of the **CreateToDoWrapper** as follows
 - Title:** *GetLocalToDoById.List.Current.LocalToDo.Title*
 - IsStarred:** *GetLocalToDoById.List.Current.LocalToDo.IsStarred*
 - Notes:** *GetLocalToDoById.List.Current.LocalToDo.Notes*
 - DueDate:** *GetLocalToDoById.List.Current.LocalToDo.DueDate*
 - CreatedDate:** *GetLocalToDoById.List.Current.LocalToDo.CreatedDate*
 - CompletedDate:** *GetLocalToDoById.List.Current.LocalToDo.CompletedDate*
 - CategoryId:** *GetLocalToDoById.List.Current.LocalToDo.CategoryId*
 - PriorityId:** *GetLocalToDoById.List.Current.LocalToDo.PriorityId*

At this point, we are considering the scenarios where the ToDo, attached to this resource, may have been created or updated in the last offline interaction, and the synchronization might not have been finished yet. This is the piece of logic that we needed to add to avoid the scenario where a Resource was added / updated to the database, and the respective ToDo was not there yet, since the sync is still being done (or has not been done yet).

- k) Drag an **Assign** and drop it below the **UpdateToDoWrapper**, connect both, then define the following assignment

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

Connect the **CreateToDoWrapper** to this Assign.

- l) Drag another **UpdateLocalToDo** Entity Action, drop it below the new Assign and connect both statements. Set its **Name** to *UpdateLocalToDoOnline*.
- m) Set the **Source** input of the **UpdatedLocalToDoOnline** to

GetLocalToDoById.List.Current.LocalToDo

- n) Drag a **Run Server Action** statement and drop it below the **SyncStatus <> None?** If, and select the *CreateOrUpdateResourceWrapper* Server Action in the **Select Action** dialog.
- o) Create a connector from the **UpdatedLocalToDoOnline** to the **CreateOrUpdateResourceWrapper**.
- p) Set the inputs of the **CreateOrUpdateResourceWrapper** as follows

ToDoid: *GetLocalToDoById.List.Current.LocalResource.Id*

ResourceTypeId:

GetLocalToDoById.List.Current.LocalResource.ResourceTypeId

Filename: *GetLocalToDoById.List.Current.LocalResource.Filename*

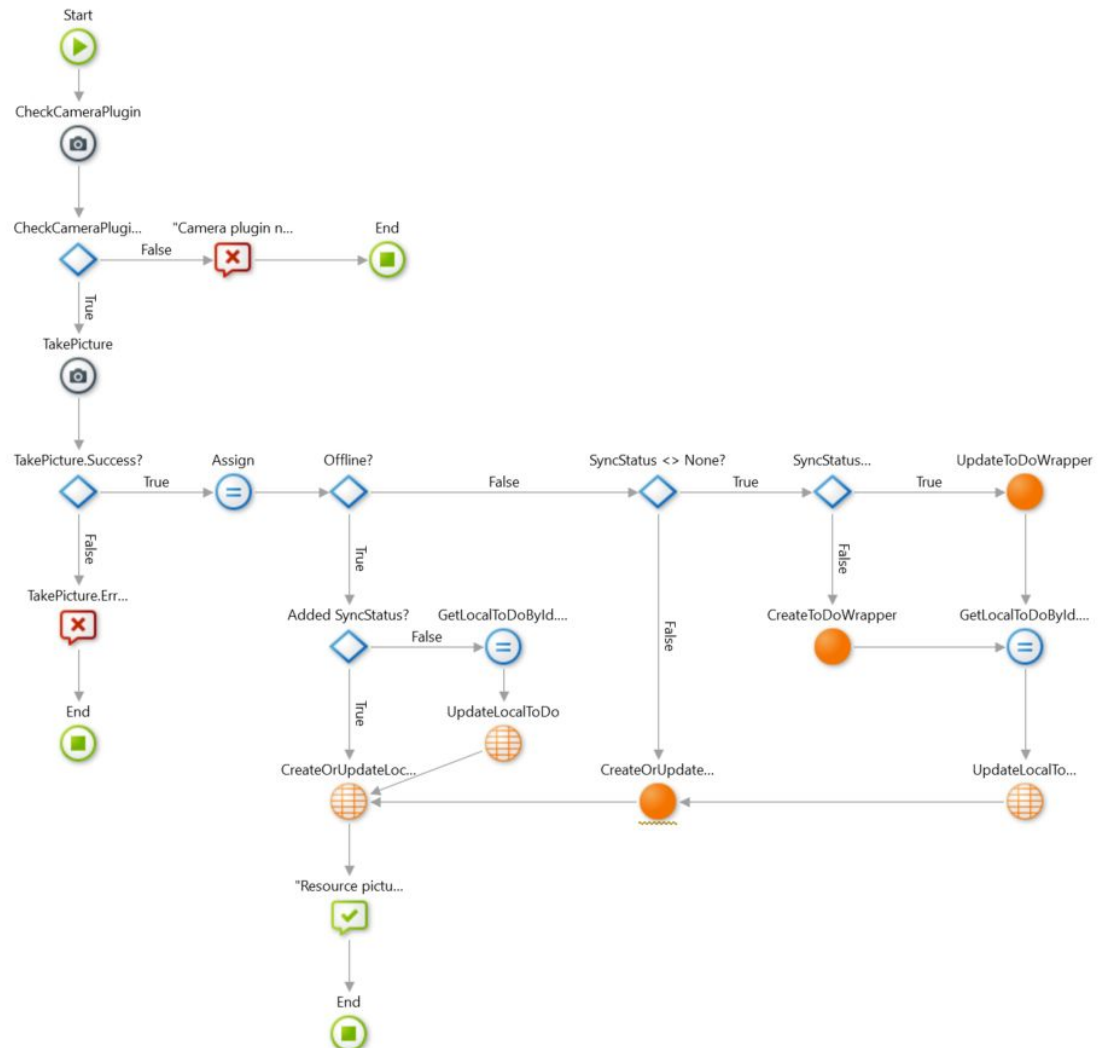
MimeType: *GetLocalToDoById.List.Current.LocalResource.MimeType*

BinaryContent: *GetLocalToDoById.List.Current.LocalResource.BinaryContent*

- q) Create the False branch connector from the **SyncStatus <> None?** If to the **CreateOrUpdateResourceWrapper**.
- r) Create a connector from the **CreateOrUpdateResourceWrapper** to the **CreateOrUpdateLocalResource**.

NOTE: The **CreateOrUpdateResourceWrapper** Server Action call has a warning about multiple calls to the server. In some situations, this will be true (when the condition **SyncStatus <> None** is met), but not in all cases (**SyncStatus = None**). It would be possible to just do one call to the server, although it would make the logic a bit more complex. Nevertheless, this is something that should be taken into account and prevented when possible.

s) The final flow should look like this

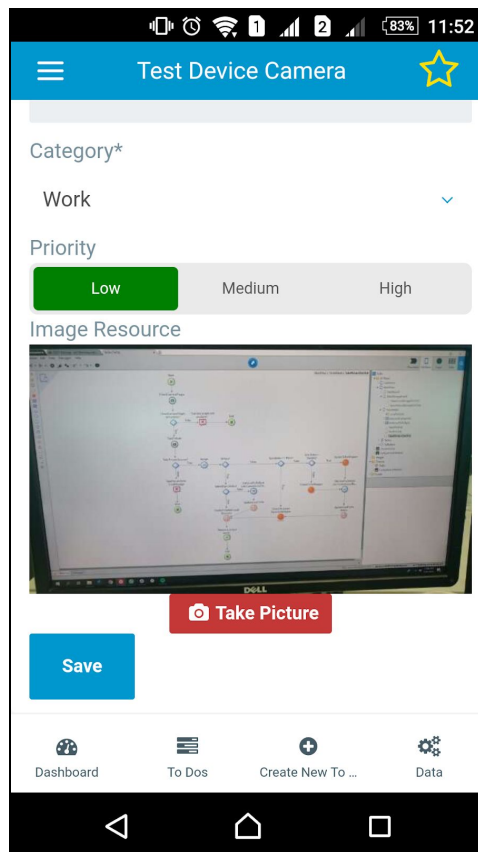


NOTE: The logic may look complex at first, so it is important to take our time and make sure that it makes sense. Taking the picture and creating the resource is easy. However, for syncing purposes, in particular when doing an offline interaction, the logic becomes a bit harder, since the sync status need to be saved. This causes several possibilities that must all be addressed on the logic.

- 9) Publish the module and test the application to make sure that the camera works.
 - a) Click the **1-Click Publish** button to publish the module to the server.
 - b) Generate a new mobile application, then install it on your mobile device. You may also use OutSystems Now.

NOTE: Whenever a plugin is added, it is required to re-generate the mobile application and install it again. The reason for this is to pack the plugin code into the application package. If the Native Platforms are already configured, and the app has been previously generated, when published the generation will be triggered.

- c) Open the application in your device.
- d) Create a new To Do with the Title *Test Device Camera* in the **Work** category and **Low** priority, then click the **Save** Button.
- e) Open the **Test Device Camera** To Do, then click the **Take Picture** Button.
- f) The devices' camera application should open. Take a picture with it.
- g) You should see the picture in the **ToDoDetail** Screen.



Create a new Plugin

In this part of the exercise, we will define a new plugin from scratch: the Dialogs Plugin. This will allow to define some custom dialogs and make them appear on our application as Alerts, with possibility to confirm an operation as a double-checking feature.

For that purpose, we will create a new application, with a Blank module, and define three Actions based on JavaScript code: *CheckDialogsPlugin*, *Alert* and *Confirm*.

- 1) Create a new Application, called *DialogsPlugin* and use the icon in the Resources folder.
Create a new **Mobile Blank** module.
 - a) Switch to the Applications tab.
 - b) Click the **New Application** icon to create a new application, then choose **Phone App** and click Next.
 - c) Set the application **Name** to *Dialogs Plugin_<your_initials>*, and fill in the **Description**.
 - d) Click the **Upload Icon** and choose the *dialogs-icon.png* from the Resources folder.
 - e) Click the **Create App** Button to finish the app creation.

New Application

Fill in your app's basic info

DialogsPlugin

The plugin provides access to some native dialog UI elements

Pick a color to bootstrap your app's interface and icon background

Or use a custom icon

UPLOAD ICON

BACK

CREATE APP

- f) Choose the **Blank** module and then click the **Create Module** Button.

Develop Native Platforms

Modules

Modules allow you to structure your application into several pieces, each piece implementing a specific purpose.

Module name: DialogsPlugin

Choose module type: Blank

CREATE MODULE CANCEL

- 2) In the new module, reference the **Dialogs Plugin** Cordova repository in the **Extensibility Configurations** of the module. Also modify the module icon to use the same of the application. There is a smaller icon available in the Resources.
 - a) Select the **DialogsPlugin** module in the Elements Area, then double-click the Extensibility Configurations property to edit it.

DialogsPlugin Module

Invalid Email	Email expected!
Upgrade Messages	
Upgrade Complete	Your application has b...
Upgrade Failed	An error occurred whil...
Upgrade Failed on Res...	An error occurred whil...
Upgrade Failed on Dat...	An error occurred whil...
Upgrade Required	Your application needs...
Upgrade Required with...	Your application needs...
Advanced	
Is Multi-tenant	No
Web Services Namesp...	
Extensibility Configura...	

- b) Add the following to the **Extensibility Configurations** dialog

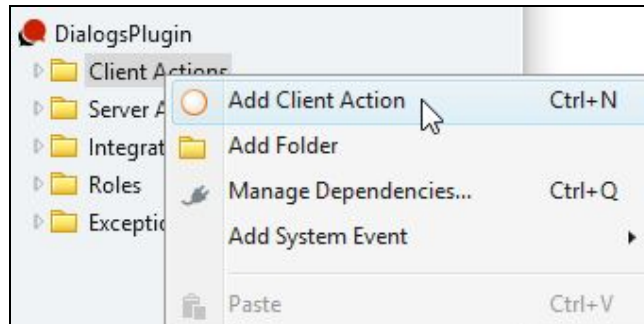
```
{
  "plugin":
  {
    "url": "https://github.com/apache/cordova-plugin-dialogs.git"
  }
}
```

NOTE: The documentation about the Dialogs Plugin is available at <https://cordova.apache.org/docs/en/latest/reference/cordova-plugin-dialogs/>

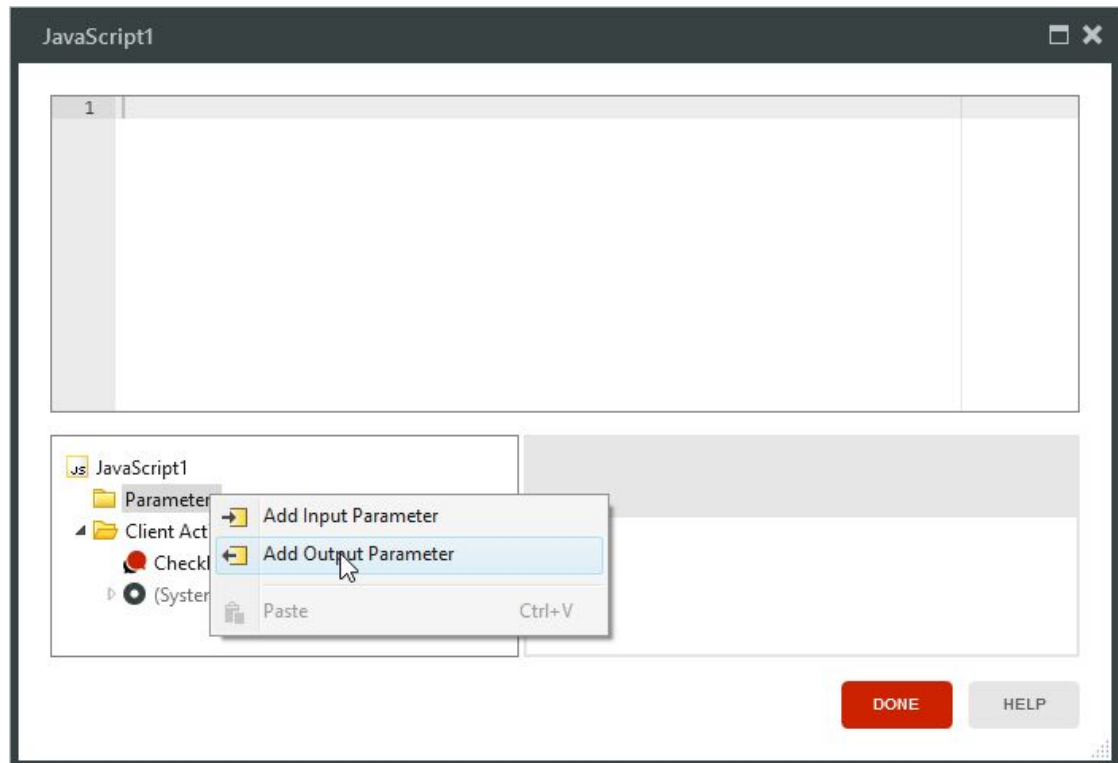
- c) Click **Close** to close the dialog.
 - d) Open the drop down for the **Icon** property and choose (*Change Icon...*), then select the *DialogsPlugin32.png* image file from the **Resources** folder.

- 3) Create the *CheckDialogsPlugin* Client Action. This Action will verify if the **DialogsPlugin** has been loaded on the device and can be used. We need to use JavaScript to help us define this logic. The Action should return if the plugin is available or not.

- a) Switch to the **Logic** tab.
- b) Right-click the **Client Actions** folder and choose *Add Client Action*.



- c) Set the new Client Action **Name** to *CheckDialogsPlugin*, then change the **Public** property to *Yes*.
- d) Open the drop down for the **Icon** property and choose *(Change icon...)*. Then, select the *dialogs-icon32.png* image from the **Resources** folder.
- e) Right-click the Client Action and choose *Add Output Parameter*.
- f) Set the **Name** of the Output Parameter to *IsAvailable* and **Data Type** to *Boolean*.
- g) Drag a **JavaScript** statement and drop it between Start and End.
- h) Double-click the **JavaScript** statement to open the JavaScript editor.
- i) Right-click the **Parameters** folder and add an Output Parameter named *IsAvailable*.



- j) Verify that the **Data Type** of the Output Parameter is set to *Boolean*.
- k) Add the following JavaScript code to the editor

```
$parameters.IsAvailable = !!navigator.notification;
```

NOTE: The *\$parameters.IsAvailable* corresponds to the created Output Parameter defined in previous steps.

- l) Click **Done** to close the JavaScript editor.
 - m) Drag an **Assign** statement and drop it between the JavaScript statement and End, then define the following assignment
- ```
IsAvailable = JavaScript1.IsAvailable
```
- 4) Create the *Alert* Client Action, with the purpose of displaying a native alert, with a customized Message and Button. This Action will expect three Input Parameters: *Title* (Text, mandatory) for the title of the dialog, *Message* (Text, mandatory) for the actual message displaying in the dialog, and *ButtonName* (Text, mandatory) for the name inside the Button. The Action also has two Output Parameters: *Success* (Boolean) and *ErrorMessage* (Text). To help with this, we need some JavaScript code.
    - a) In the **Logic** tab, create a new Client Action named *Alert*.



- b) Set the **Public** property to *Yes* and change the **Icon** of the Client Action to the image file *dialogs-icon32.png*, located in the **Resources** folder.
- c) Add a new Input Parameter and set its **Name** to *Title*, **Data Type** to *Text* and **Is Mandatory** to *Yes*.
- d) Add another Input Parameter and set its **Name** to *Message*, **Data Type** to *Text*, and **Is Mandatory** to *Yes*.
- e) Add another Input Parameter and set its **Name** to *ButtonName*, **Data Type** to *Text*, and **Is Mandatory** to *Yes*.
- f) Add an Output Parameter with **Name** *Success* and **Data Type** *Boolean*.
- g) Add another Output Parameter with **Name** *ErrorMessage* and **Data Type** *Text*.
- h) Drag a **JavaScript** statement and drop it between the Start and End.
- i) Double-click the **JavaScript** statement to open the JavaScript editor.
- j) Inside the JavaScript editor, add an Input Parameter to the JavaScript statement by right-clicking the **Parameters** folder, then set its **Name** to *Title* and **Data Type** to *Text*.
- k) Repeat the previous step for the **Message** and **ButtonName** Input Parameters with the same data type.
- l) Inside the JavaScript editor, add an Output Parameter named *Success* with **Data Type** *Boolean*.
- m) Add another Output Parameter named *ErrorMessage* with **Data Type** set to *Text*.

---

**NOTE:** JavaScript statements act as black boxes, therefore all data that is needed inside the JavaScript code must be sent as Input Parameter into the JavaScript statement. The same happens with data sent out of the JavaScript statement.

---

- n) Add another Output Parameter named *ErrorMessage* with **Data Type** set to *Text*.
- o) In the text editor area add the following JavaScript code

```

if($actions.CheckDialogsPlugin()) {
 navigator.notification.alert(
 $parameters.Message, // message
 function (){ // callback
 $resolve();
 $parameters.Success = true;
 },
 $parameters.Title, // title
 $parameters.ButtonName // buttonName
);
} else {
 $parameters.Success = false;
 $parameters.ErrorMessage = "Dialogs plugin is not available";
}

```

- p) Click **Done** to close the JavaScript editor.
- q) Select the **JavaScript** statement and define the Input Parameters to use the corresponding Client Action Input Parameters.

| JavaScript1<br>JS JavaScript |                        |
|------------------------------|------------------------|
| Name                         | JavaScript1            |
| Description                  | ...                    |
| JavaScript                   | if(\$actions.Checl ... |
| Title                        | Title ▼                |
| Message                      | Message ▼              |
| ButtonName                   | ButtonName ▼           |
| Success                      | ▼                      |
| ErrorMessage                 | ▼                      |

- r) Drag an **Assign** statement and drop it between the JavaScript and End, then define the following assignments

```


Success = JavaScript1.Success
ErrorMessage = JavaScript1.ErrorMessage

```

- 5) Create a *Confirm* Client Action to provide a confirmation process for the end-user. This Action will have two Input Parameters: *Title* (Text, mandatory) and *Message* (Text, mandatory). It also has three Output Parameters: *Confirmed* (Boolean) that returns if the user confirmed the operation or not, *Success* (Boolean) to make sure the Action was successful and *ErrorMessage* (Text) to be displayed when an error occurs. As above, we need some JavaScript to help us defining the logic for this Action.
- Create a new Client Action named *Confirm*.
  - Set the **Public** property to *Yes* and change the Client Action **Icon** to the *dialogs-icon32.png* image, located in the **Resources** folder.
  - Add two new Input Parameters to the Client Action, with **Text Data Type**. The Inputs are named *Title* and *Message*.
  - Add an Output Parameter named *Confirmed* with **Data Type** set to *Boolean*.
  - Repeat the previous step for the *Success* Output Parameter.
  - Add another Output Parameter named *ErrorMessage* with **Data Type** set to *Text*.
  - Drag a **JavaScript** statement, drop it on the Action flow and open it.
  - Add the proper Input Parameters and Output Parameters, matching the ones on the Client Action, just like in the previous steps.
  - In the JavaScript code editor write the following

```
if($actions.CheckDialogsPlugin()) {
 navigator.notification.confirm(
 $parameters.Message, // message
 function (buttonIndex){ // callback
 $parameters.Success = true;
 $parameters.Confirmed = (buttonIndex === 1);
 $resolve();
 },
 $parameters.Title, // title
 ['Yes', 'No'] // buttons
);
} else {
 $parameters.Success = false;
 $parameters.ErrorMessage = "Dialogs plugin is not available";
}
```

- j) Click **Done** to close the JavaScript editor.
- k) Set the **Title** and **Message** properties of the JavaScript statement to the Input Parameters of the Client Action.

|                                                                                                                                   |                                          |
|-----------------------------------------------------------------------------------------------------------------------------------|------------------------------------------|
| <div>  <b>JavaScript1</b><br/> JavaScript </div> |                                          |
| Name                                                                                                                              | JavaScript1                              |
| Description                                                                                                                       | ...                                      |
| JavaScript                                                                                                                        | if(\$actions.CheckDialogsPlugin()) { ... |
| Title                                                                                                                             | Title ▼                                  |
| Message                                                                                                                           | Message ▼                                |
| (New Argu...                                                                                                                      | ▼                                        |

- l) Drag an **Assign** statement and drop it between the JavaScript and End, then define the following assignments

*Confirmed = JavaScript1.Confirmed*

*Success = JavaScript1.Success*

*ErrorMessage = JavaScript1.ErrorMessage*

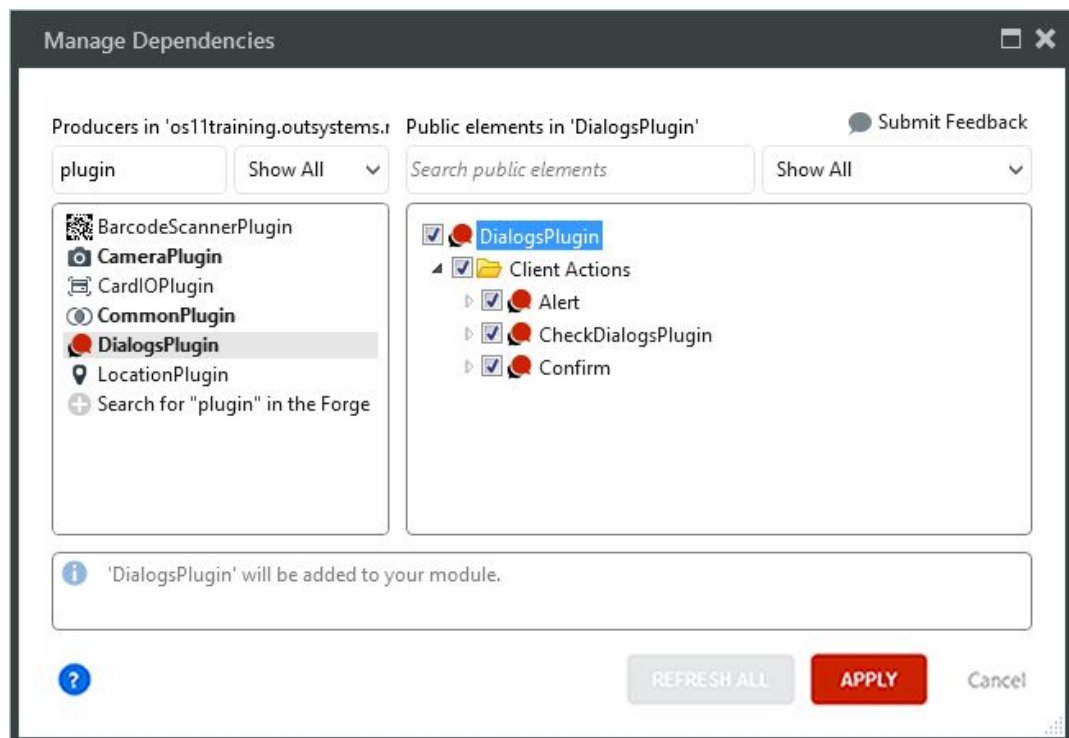
- m) Click the **1-Click Publish** button and verify that the module was successfully published.

## Test the Dialogs Plugin

In this part of the exercise, we will reference the Dialogs Plugin from the **ToDo** application.

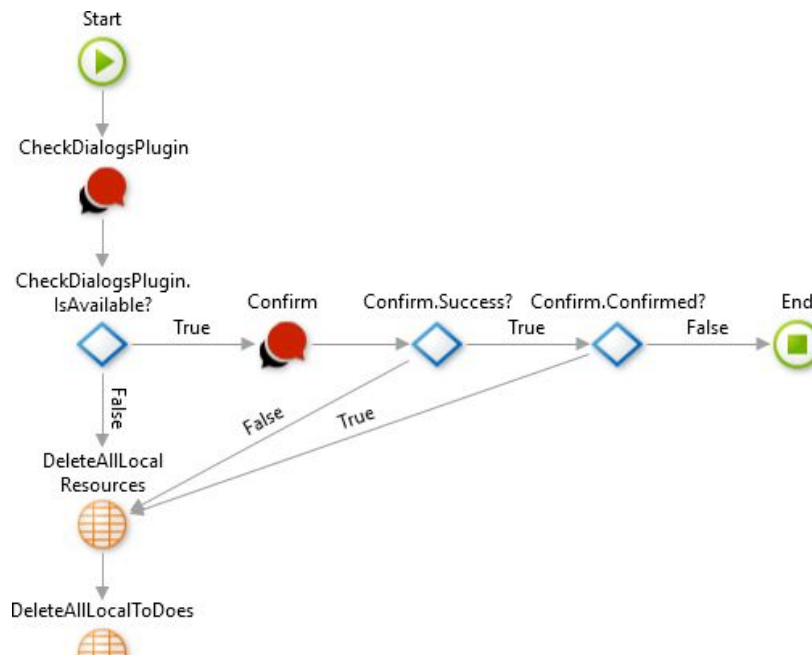
We will make sure that a confirmation dialog appears when the user tries to clear all local storage, in the **DataManagement** Screen. Also, if the user confirms and the local storage is cleared, and **Alert** message will appear informing the user of that.

- 1) In the **ToDo** module, add references to the new Actions from the **DialogsPlugin**.
  - a) Open the **ToDo** application module.
  - b) Click the **Manage Dependencies** button in the toolbar.
  - c) In the **Manage Dependencies** dialog, select the *DialogsPlugin* module and then tick the *Alert*, *CheckDialogsPlugin* and *Confirm* Client Actions.
  - d) Click **Apply** to close the dialog.



- 2) Change the **Clear Local Storage** Button to ask for confirmation using the Dialogs Plugin **Confirm** Action, before deleting any data from Local Storage. Don't forget to check if the plugin exists before using the Action and also to check if the Action was successful and confirmed.
  - a) Open the **ClearLocalStorageOnClick** Client Action of the **DataManagement** Screen.

- b) Drag a **Run Client Action** statement and drop it right after the Start node.
- c) In the **Select Action** dialog, choose the *CheckDialogsPlugin* Client Action from the **DialogsPlugin** module.
- d) Drag an **If** statement and drop it immediately after the **CheckDialogsPlugin**, then set the Condition property to  
`CheckDialogsPlugin.IsAvailable`
- e) Drag a new **Run Client Action** statement and drop it to the right of the If statement, then select the *Confirm* Client Action from the **DialogsPlugin**.
- f) Set the **Title** parameter to *"Local Storage"*, and the **Message** parameter to *"Are you sure that you want to clear all local storage?"*.
- g) Create the **True** branch connector from the If to the **Confirm** Action.
- h) Drag an **If** statement and drop it on the right of the **Confirm** Action, then create the connector between both.
- i) Set the **Condition** property of the If to  
`Confirm.Success`
- j) Drag another **If** statement and drop it on the right of the previous one, then create the **True** branch connector from the existing to the new one.
- k) Set the **Condition** property of the new If to  
`Confirm.Confirmed`
- l) Create the **False** branch connector from the **Confirm.Success?** If to the first **Delete All** Entity Action statement. This means that if the Dialogs did not work successfully, the process for deleting the local storage continues as expected.
- m) Create the **True** branch connector from the **Confirm.Confirmed?** If to the first **Delete All** Entity Action statement. If the user confirms that the local storage should be deleted, then the delete logic will proceed.
- n) Drag an **End** and drop it on the right of the **Confirm.Confirmed?** If, then create the **False** branch connector from the If to the End. If the user does not confirm, then the logic will not be executed and the Action ends.
- o) The initial part of the flow should look like this



**NOTE:** The logic created above is designed in such way that in the case that the Dialogs Plugin is not available (e.g. the app installed in the device was not upgraded yet), the logic will still work without errors.

When integrating with newly added native plugins it is important that the application is designed in such way that allows for both use cases: plugin is available and not available. This way, we ensure that end-users have an error-free experience in case the plugin is not yet available in the native app they have installed on their device, or even if the mobile device does not support the plugin.

- 3) Before the **ClearLocalStorageOnClick** Action ends, we want to notify the user that the local storage was deleted, using the **Alert** Action from the **DialogsPlugin**. If the plugin is not available, the built-in Message should be used instead.
  - a) Drag an **If** and drop it between immediately before the Message statement, then set the **Condition** property to
   
*CheckDialogsPlugin.IsAvailable*
  - b) Drag a **Run Client Action** statement and drop it on the right of the If statement.
  - c) In the **Select Action** dialog choose the *Alert* Client Action from the **DialogsPlugin**.
  - d) Create the **True** branch connector from the If to the **Alert** statement.

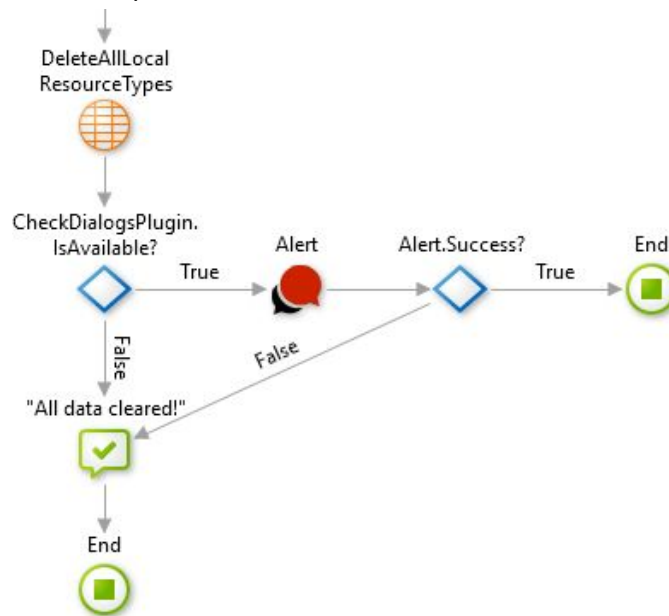
- e) Set the **Title** property of the **Alert** Action to *"Local Storage"*, the **Message** parameter to *"All data cleared!"*, and the **ButtonName** to *"Dismiss"*.
- f) Drag an **If** Widget and drop it on the right of the **Alert** statement, and create the connector between both.
- g) Set the **Condition** property of the If to  
*Alert.Success*
- h) Drag an **End** statement and drop it on the right of the If statement, then create the **True** branch connector from the If to the End.
- i) Create the **False** branch connector from the If to the existing Message at the end of the flow.

---

**NOTE:** The logic above is defined in such a way that if the **DialogsPlugin** is not available, the data will be cleared and a **Message** will appear. However, if the **DialogsPlugin** is available, then the user will have to confirm to clear the local storage data and then will see an alert.

---

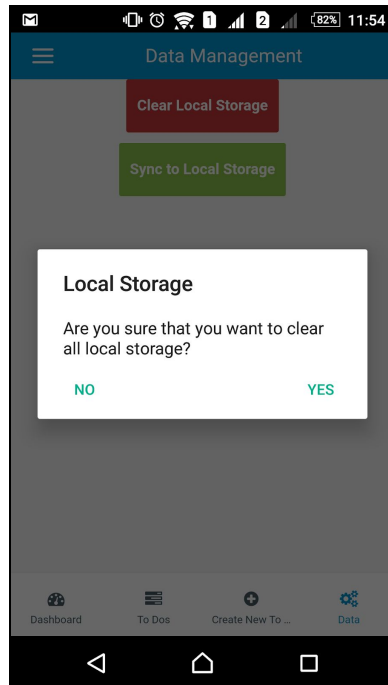
- j) The final part of the flow should look like this



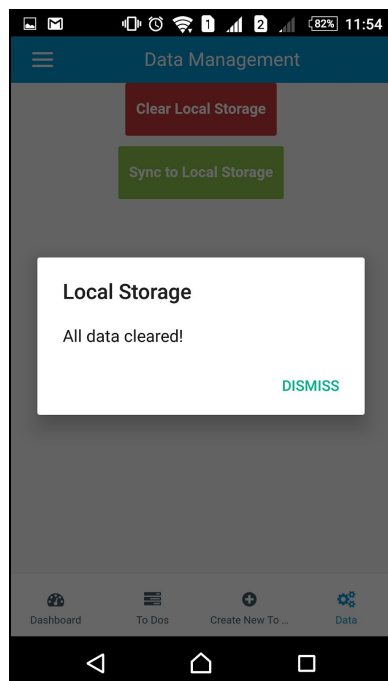
- 4) Publish the module and test the application on the device. Make sure that the dialogs properly appear when they are expected.
  - a) Generate a new native application for your device.
  - b) After the generate process is complete, install the new application in the device.



- c) Navigate to the **DataManagement** Screen and press the **Clear Local Storage**. You should see the native confirmation dialog.



- d) If you select *No*, nothing will be deleted from the Local Storage.
- e) Select *Yes*, and then you should see the **Alert** dialog.



## End of Lab

In this exercise, we added plugins to the ToDo application.

First, we installed the Camera Plugin from Forge, and added the functionality to Take a Picture with the device and associate it to a To Do, as a Resource. This had some impacts as well on the offline interaction and synchronization, which led us to create some logic to handle some edge cases.

Then, we created the Dialogs Plugin, to allow having Alerts and Confirmation dialogs in the ToDo application. We first created a new application for the plugin. Then, we used JavaScript to implement the plugin functionality. Finally, we referenced the plugin in the ToDo application and used the dialogs for the Clear Local Storage functionality.

For this lab, we needed to generate a new version of the ToDo native application, since a new plugin was added to the app. This is one of the cases that require a new version of a native app, alongside a change of name or change of icon of the app.

This lab is the final one for the ToDo app.