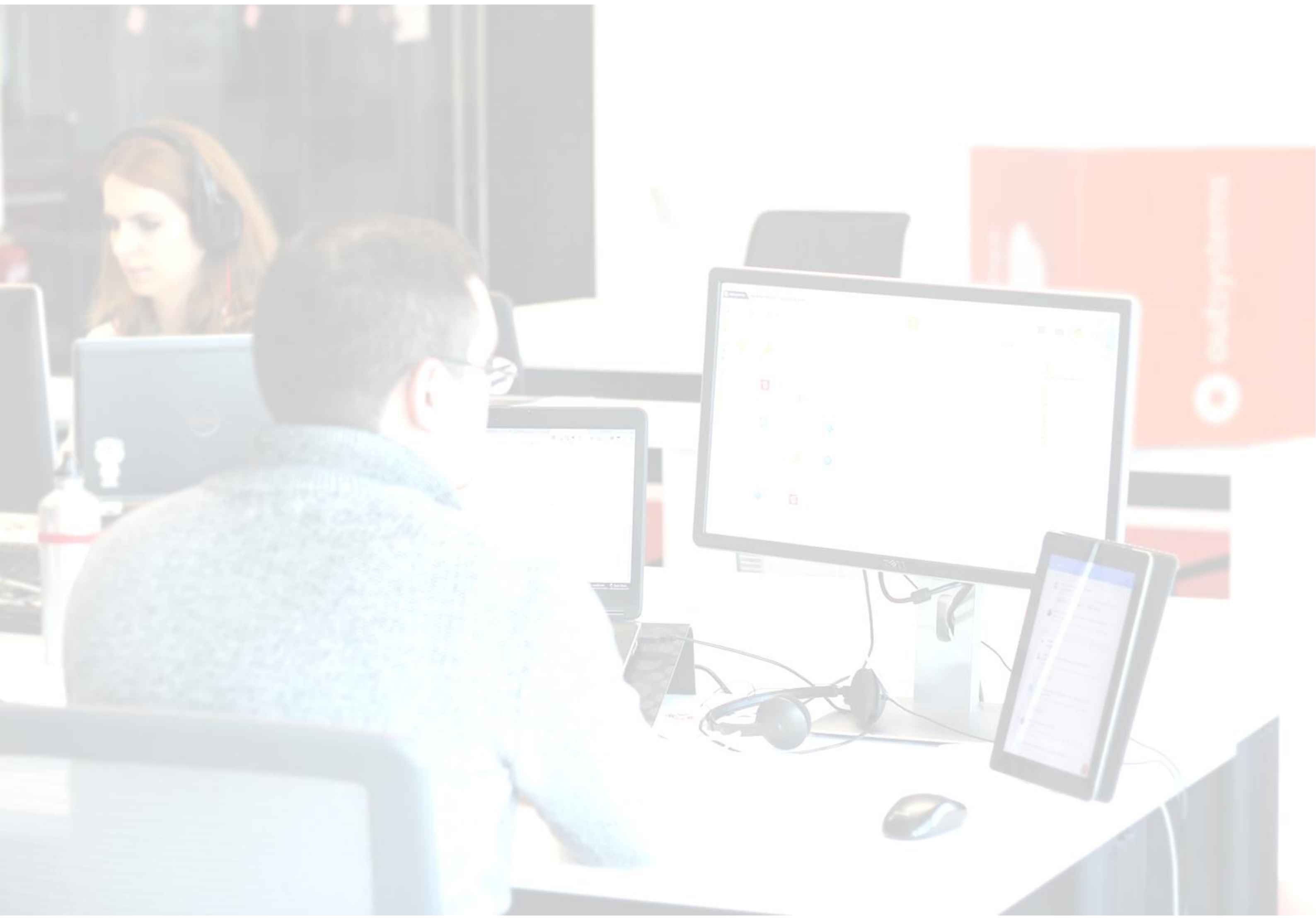
outsystems

DEVELOPING OUTSYSTEMS MOBILE APPS

# Plugins



# Introduction

Over the course of this set of exercise labs, you will create a mobile application. The application will focus on creating and managing To Dos. The To Dos will be persisted in a database so they can be accessed from and shared across multiple devices. To Dos will have attributes such as category, priority (low, medium or high), due date and they can be marked as important (starred) by the user.

Users of the To Do application will be able to access all of this information regardless of whether the device is online or offline. When offline, users will still be able to keep interacting with the application and changes will be saved locally in the device local storage. When the device returns to online mode, changes made while offline will automatically be synced to the server.

You constantly will be expanding your application, publishing it to the server and testing it in your mobile device. Throughout the process you will be learning and applying new OutSystems concepts.

At the end of this set of exercise labs, you will have a small, but well-formed application, spanning multiple screens and concepts that you can easily access from your mobile device.

In this specific exercise lab, you 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

# Table of Contents

# Part 1: Install Plugin from Forge

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

**1.** Install the **Camera** Plugin from Forge.

**a)** Switch to the **Applications** tab.

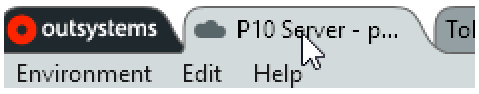


Figure 1. Applications tab

**b)** Click the 'Install Application' item to open Forge.

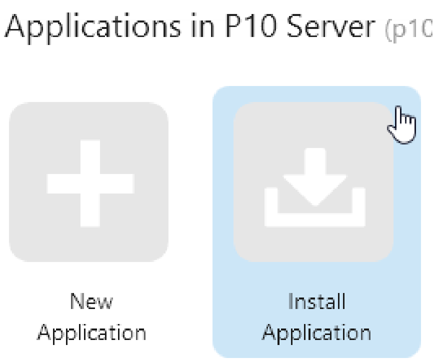
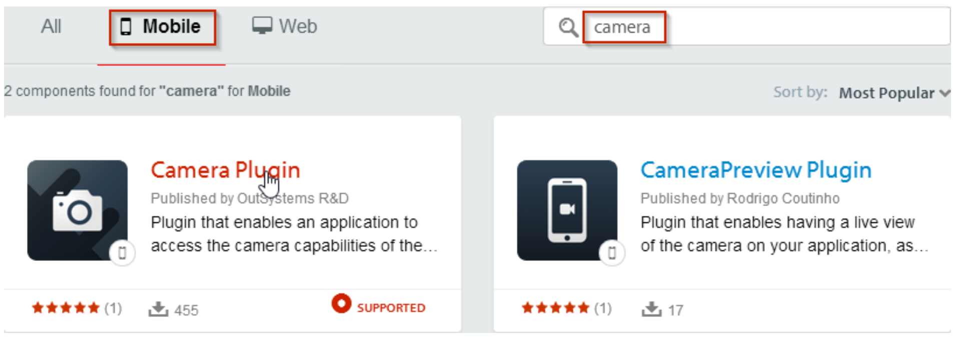


Figure 2. Install Application

**c)** Search for the **Mobile** Plugin 'Camera Plugin'.



2 components found for "camera" for Mobile

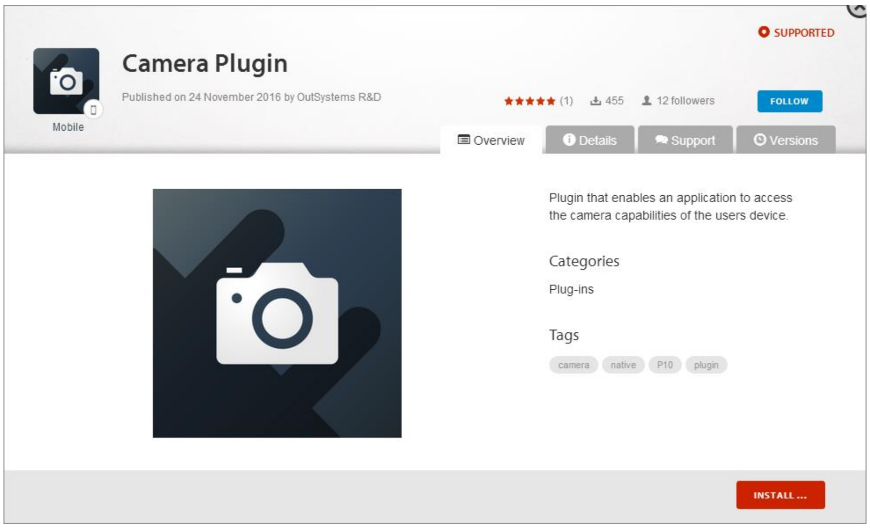
Camera Plugin
Published by OutSystems R&D
Plugin that enables an application to access the camera capabilities of the...
★★★★★ (1) 455

CameraPreview Plugin
Published by Rodrigo Coutinho
Plugin that enables having a live view of the camera on your application, as...
★★★★★ (1) 17

Figure 3. CameraPlugin on Forge

**d)** Click the **CameraPlugin** title to open the component details.

**e)** Click the 'Install...' Button to install the component.



Plugin that enables an application to access the camera capabilities of the users device.

Categories

Plug-ins

Tags

camera native P10 plugin

Figure 4. CameraPlugin Dialog

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

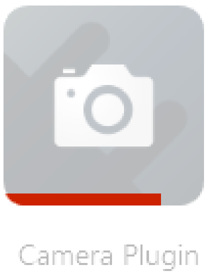


Figure 5. Component installation progress

# Part 2: Attach Camera Pictures to To Dos

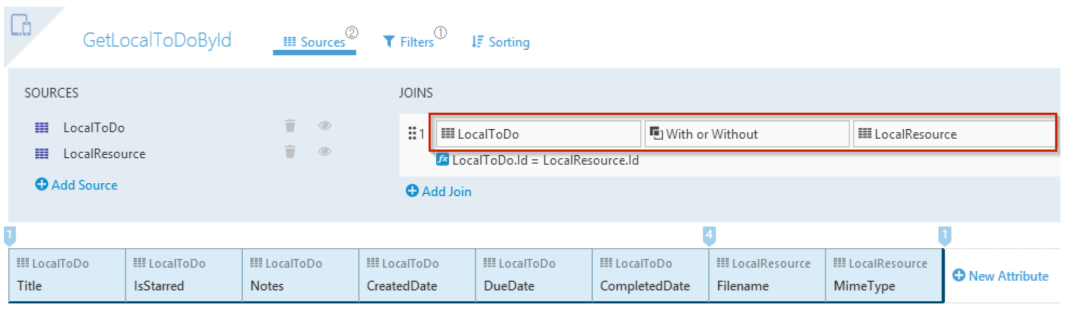
In this part of the exercise, you will modify the **ToDoDetail** Screen so that it's possible to attach pictures taken with the **Camera** Plugin.

**1.** Modify the **GetLocalToDoById** Aggregate of the **ToDoDetail** Screen to also retrieve the attached resource if it exists.

**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** from 'Only With' to 'With or Without'.



SOURCES

- LocalToDo
- LocalResource

JOINS

| | | | |
|---|---|---|---|
| 1 | LocalToDo | With or Without | LocalResource |
| | LocalToDo.Id = LocalResource.Id | | |

| LocalToDo | LocalToDo | LocalToDo | LocalToDo | LocalToDo | LocalToDo | LocalResource | LocalResource |
|---|---|---|---|---|---|---|---|
| Title | IsStarred | Notes | CreatedDate | DueDate | CompletedDate | Filename | MimeType |

Figure 6. GetLocalToDoById Sources tab

**NOTE:** Recall that To Dos 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 not be able to retrieve the To Do details for To Dos without an attached resource.

**2.** Display image resources of To Dos.

**a)** Open the **ToDoDetail** Screen.

**b)** Drag a **Container** Widget and drop it between the **Priority** and **Save** Button.

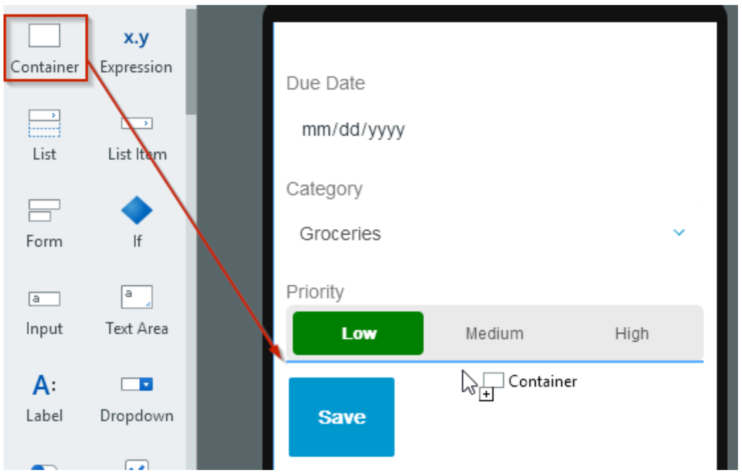


Figure 7. Drag and drop a Container

**c)** Drag a new **Label** Widget and drop it inside the Container created in the previous step, then change the **Text** inside it to 'Image Resource'.

**d)** Drag a **Lightbox Image** Widget and drop below the **Label** but still inside the surrounding **Container**.

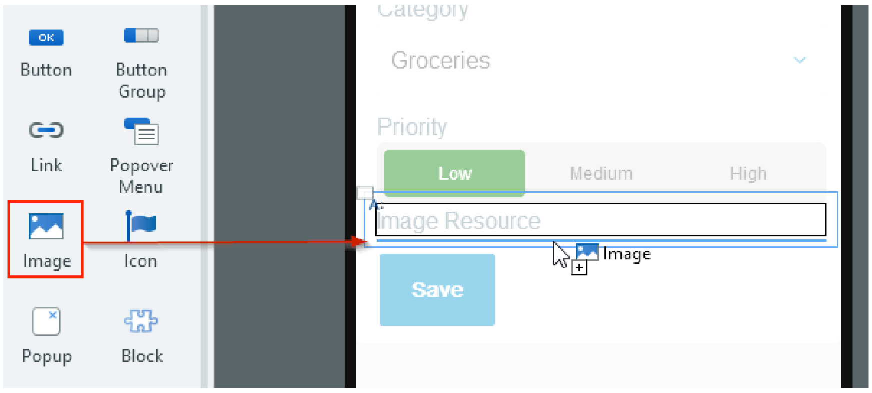


Figure 8. Drag and drop an Image

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

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

**f)** Set the **Width** of the Image to '(fill parent)'.

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

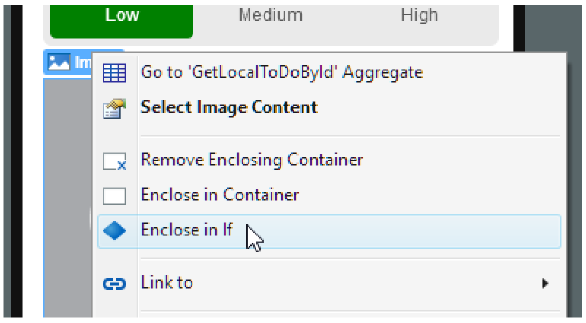


Figure 9. Enclose in If

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

```
GetLocalToDoById.List.Current.LocalResource.Id <>
NullIdentifier() and
GetLocalToDoById.List.Current.LocalResource.ResourceTypeId =
Entities.ResourceType.Image
```

**i)** In the **False** branch of the If Widget, write ‘No Image attached.’, and align the text to center, using the Styles Editor area.

**3.** Add a reference to the Camera Plugin.

**a)** Open the **Manage Dependencies** dialog by clicking the icon in the toolbar.

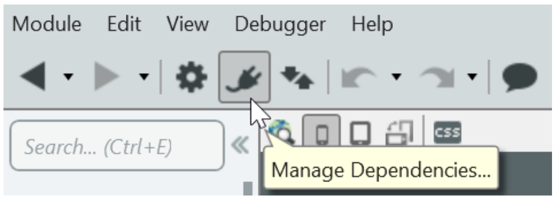


Figure 10. Manage Dependencies

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

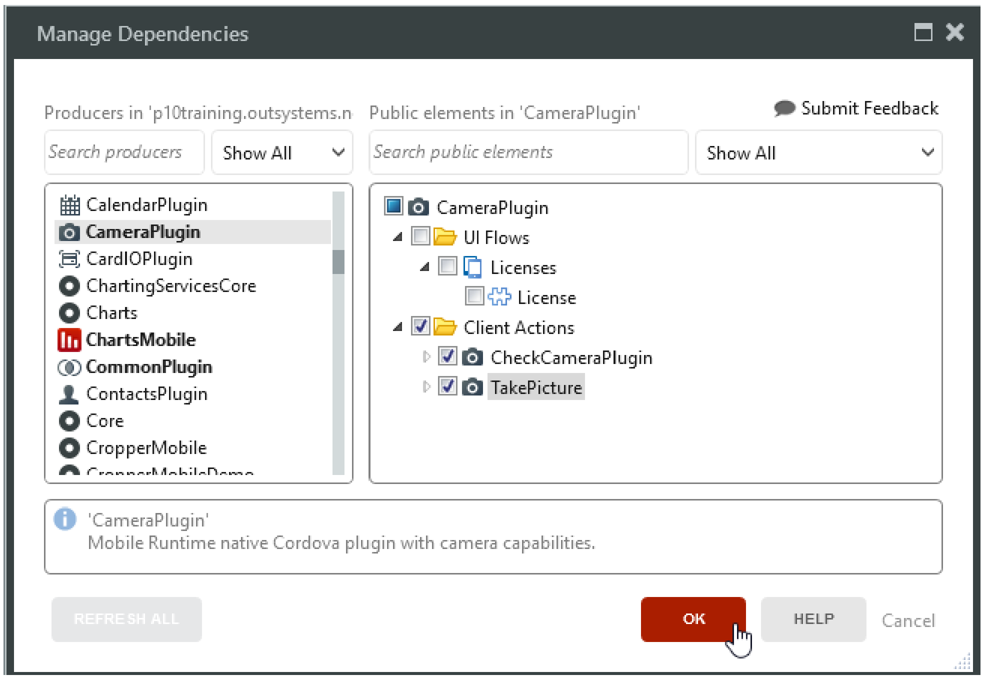


Figure 11. CameraPlugin references

**c)** Click **Ok** to close the Manage Dependencies dialog.

**4.** Create a Server Action wrapper to create or update resources.

**a)** Switch to the **Logic** tab, then right click the **Server Actions** and choose 'Add Server Action'.

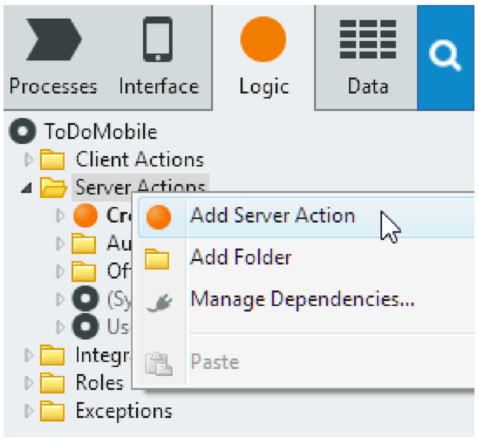


Figure 12. Add Server Action

**b)** Change the name of the Server Action to 'CreateOrUpdateResourceWrapper'.

**c)** Add an **Input Parameter** named 'Resource' and verify that the **Data Type** is set to 'Resource'.

**d)** Add an **Output Parameter** named 'ResourceId' and verify that the **Data Type** is set to 'Resource Identifier'.

**e)** In the Server Action flow, drag an **If** statement and drop it between Start and End, and set its **Condition** property to

```
Resource.Id <> NullIdentifier()
```

**f)** Drag an **Aggregate** statement and drop it to the right of the **If** statement, then create the **True** branch connector from the **If** to the **Aggregate**.

**g)** Double-click on the Aggregate to open its editor.

**h)** From the **Data** tab, drag the **ToDo** Database Entity from the **ToDo_Core** module.

**i)** Add the following filter to the Aggregate

```
ToDo.Id = Resource.Id
```

**j)** Return to the **CreateOrUpdateResourceWrapper** Server Action.

**k)** Drag another **If** statement and drop it on the right side of the **GetToDoById** Aggregate.

**l)** Set the **Condition** property of the If to

```
GetToDoById.List.Current.ToDo.UserId <> GetUserId()
```

**m)** Create the connector between the Aggregate and the If statement.

**n)** Drag a **Raise Exception** statement and drop it on the right of the If statement, and in the 'Select Exception' dialog select **InvalidToDoException**.

**o)** Set the **Exception Message** property to "To Do not found.".

**p)** Create the **True** branch connector from the If to the **Raise Exception** statement.

**q)** Drag a **Run Server Action** and drop it between the first created If and the End.

**r)** In the **Select** Action dialog, choose the **CreateOrUpdateResource** Entity Action.

**s)** Set the **Source** property to the 'Resource' Input Parameter.

**t)** Create the missing **False** branch connector from the second **If** to the **CreateOrUpdateResource** statement.

**u)** Add an **Assign** statement and drop it between the **CreateOrUpdateResource** statement and End, then define the following assignment

```
ResourceId = CreateOrUpdateResource.Id
```

**v)** The **CreateOrUpdateResourceWrapper** Server Action should look like this

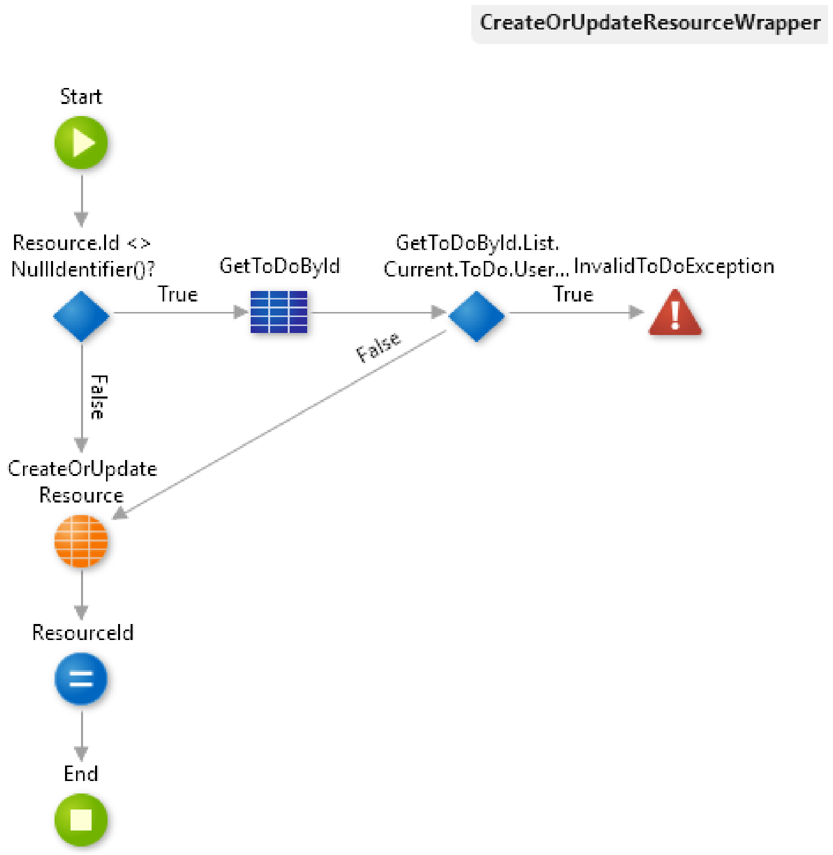
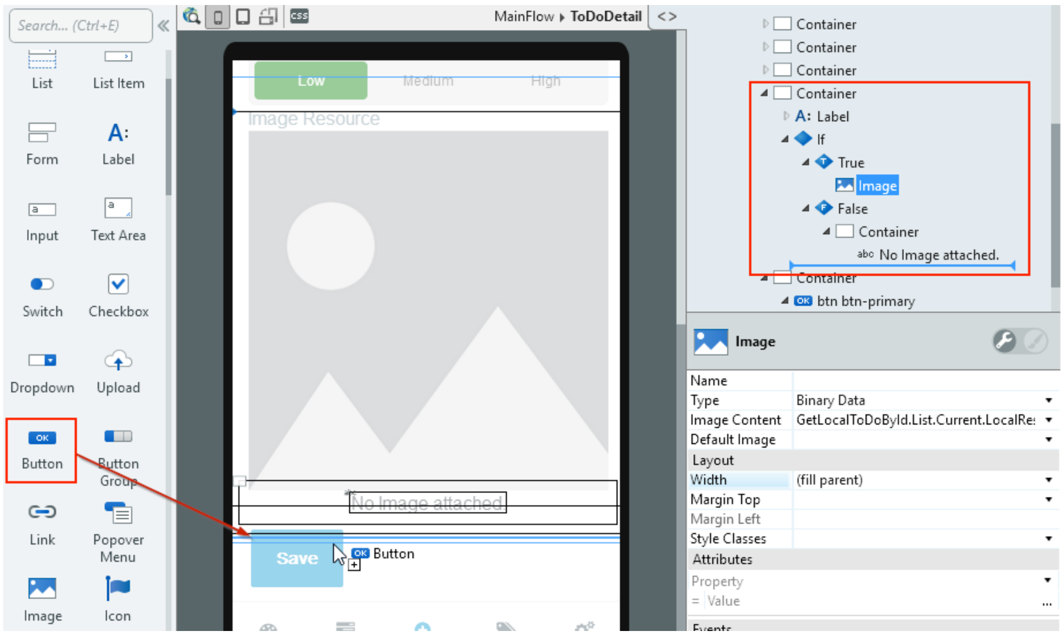

CreateOrUpdateResourceWrapper
Start
Resource.Id <> NullIdentifier()?
True
GetToDoById
GetToDoById.List.Current.ToDo.User...
True
InvalidToDoException
False
False
CreateOrUpdate Resource
ResourceId
End

Figure 13. CreateOrUpdateResourceWrapper Server Action

**5.** Create the 'Take Picture' Button.

**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.



Name
Type Binary Data
Image Content GetLocalToDoById.List.Current.LocalRe:
Default Image
Layout
Width (fill parent)
Margin Top
Margin Left
Style Classes
Attributes
Property
= Value
Events

Figure 14. Drag and drop a Button Widget

**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, using the Styles Editor area.

**g)** Your Screen should look like this

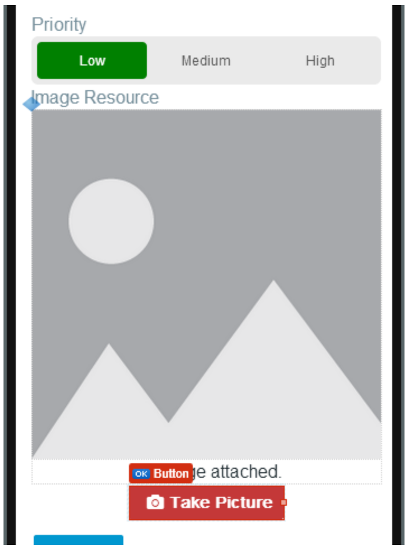


Figure 15. Take Picture Button

**h)** 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.

**6.** Attach an image taken with the device camera to an existing To Do.

**a)** Double-click the **Button** to create a new Client Action and bind it to the **On Click** Event.

**b)** Delete the existing Comment, then select the **If** statement and change the **Condition** property to ‘`GetNetworkStatus()`’.

**c)** Drag a **Message** statement and drop it on top of the **False** branch connector.

**d)** Set the **Type** property of the Message statement to ‘Error’, and set the **Message** property to "Unable to update To Dos while offline.".

**e)** Drag a **Run Client Action** statement and drop it between the **If** and End, then in the **Select Action** dialog choose the **CheckCameraPlugin** Client Action from the **CameraPlugin** module.

**f)** Drag an **If** statement and drop it between the **CheckCameraPlugin** and End, the set the **Condition** property of the **If** to

```
CheckCameraPlugin.IsAvailable
```

**g)** Drag a **Message** statement and drop it between the **If** and End.

**h)** Set the **Type** property of the **Message** to ‘Error’ and the **Message** property to “Camera plugin not available.”.

**i)** Drag a **Run Client Action** statement and drop it on the right of the **If** statement, then choose the **TakePicture** Client Action from the **CameraPlugin** module.

**j)** Create the **True** connector from the **If** to the **TakePicture** statement.

**k)** Drag an **If** statement and drop it on the right of the **TakePicture**, then create the connector from **TakePicture** to the new **If**.

**l)** Set the **Condition** property of the new If to ‘TakePicture.Success’

**m)** Drag an **Assign** statement and drop it below the **If** statement, then create the **True** branch connector from the **If** to the **Assign** statement.

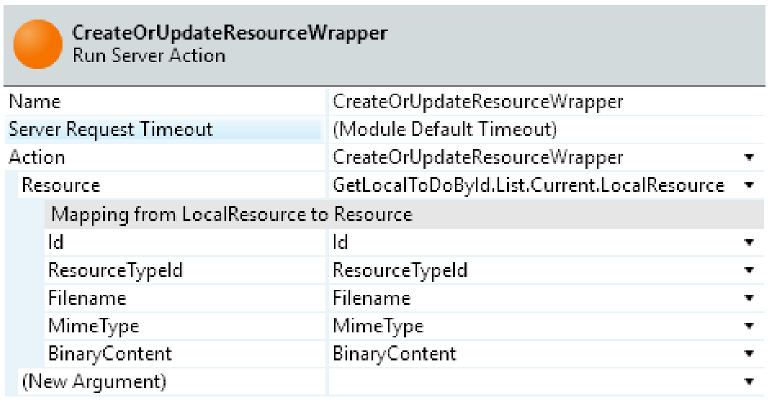
**n)** Select the Assign statement and define the following assignments

```
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
```

**o)** Drag a **Run Server Action** statement and drop it below the Assign statement, and select the **CreateOrUpdateResourceWrapper** Server Action in the **Select Action** dialog.

**p)** Create the connector between the Assign and **CreateOrUpdateResourceWrapper** statements.

**q)** Set the **Resource** parameter of the Server Action to ‘GetLocalToDoById.List.Current.LocalResource’, then define the mapping accordingly.


**CreateOrUpdateResourceWrapper**
Run Server Action

| | |
|---|---|
| Name | CreateOrUpdateResourceWrapper |
| Server Request Timeout | (Module Default Timeout) |
| Action | CreateOrUpdateResourceWrapper |
| Resource | GetLocalToDoById.List.Current.LocalResource |
| Mapping from LocalResource to Resource | |
| Id | Id |
| ResourceTypeId | ResourceTypeId |
| Filename | Filename |
| MimeType | MimeType |
| BinaryContent | BinaryContent |
| (New Argument) | |

Figure 16. CreateOrUpdateResourceWrapper mapping

**r)** Drag a **Run Client Action** statement and drop it below the **CreateOrUpdateResourceWrapper** statement, then in the **Select Action** dialog choose **CreateOrUpdateLocalResource**.

**s)** Create the connector between **CreateOrUpdateResourceWrapper** and the **CreateOrUpdateLocalResource** statements.

**t)** Set the **Source** property of the **CreateOrUpdateLocalResource** to 'GetLocalToDoById.List.Current.LocalResource'.

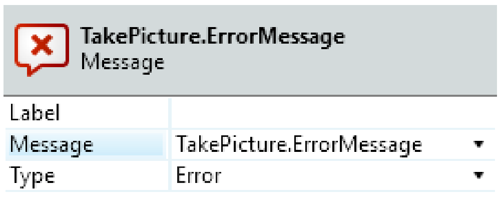
**u)** Drag a **Message** statement and drop it below the **CreateOrUpdateLocalResource** statement, then create the connector between both.

**v)** Set the **Type** of the Message statement to 'Success' and the message to "Resource picture saved.".

**w)** Drag an **End** statement and drop it below the success **Message** statement, then create the connector between both.

**x)** Drag a new **Message** statement and drop it on the right of the If statement that verifies the success of the **TakePicture** Client Action, then create the **False** connector from the **If** to the **Message** statement.

**y)** Set the **Type** property of the **Message** to 'Error' and the message to 'TakePicture.Error.ErrorMessage'.


TakePicture.ErrorMessage
Message

| | |
|---|---|
| Label | |
| Message | TakePicture.ErrorMessage |
| Type | Error |

**Figure 17. Error Message**

**z)** Drag and **End** statement and drop it on the right of the error **Message**, then create the connector between both statements. The **TakePictureOnClick** Client Action should look this

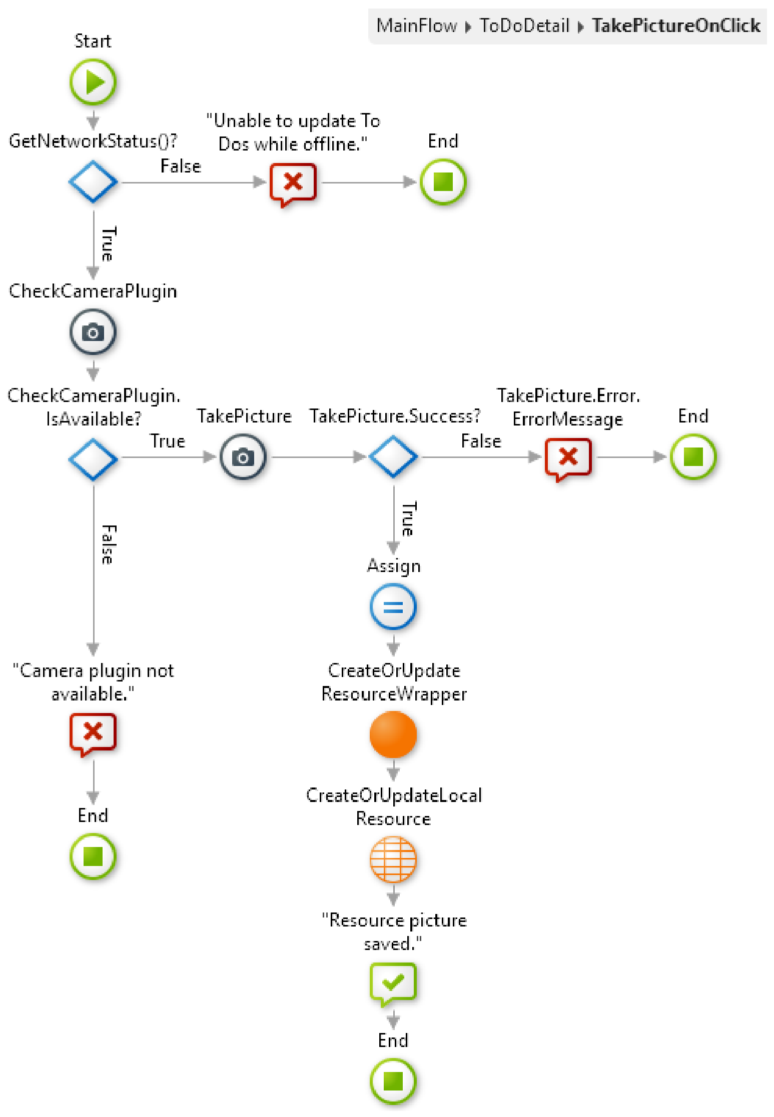

MainFlow ▸ ToDoDetail ▸ TakePictureOnClick
Start
GetNetworkStatus()?
False
"Unable to update To Dos while offline."
End
True
CheckCameraPlugin
CheckCameraPlugin.IsAvailable?
True
TakePicture
TakePicture.Success?
False
TakePicture.Error.ErrorMessage
End
True
False
Assign
"Camera plugin not available."
End
CreateOrUpdateResourceWrapper
CreateOrUpdateLocalResource
"Resource picture saved."
End

Figure 18. TakePictureOnClick Client Action

**7.** Publish and test the application.

**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.

**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.

To generate a new version simply switch to the Applications tab of Service Studio and open the 'Native Platforms' tab inside your application. Then select the desired platform and click the generate button.

**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.

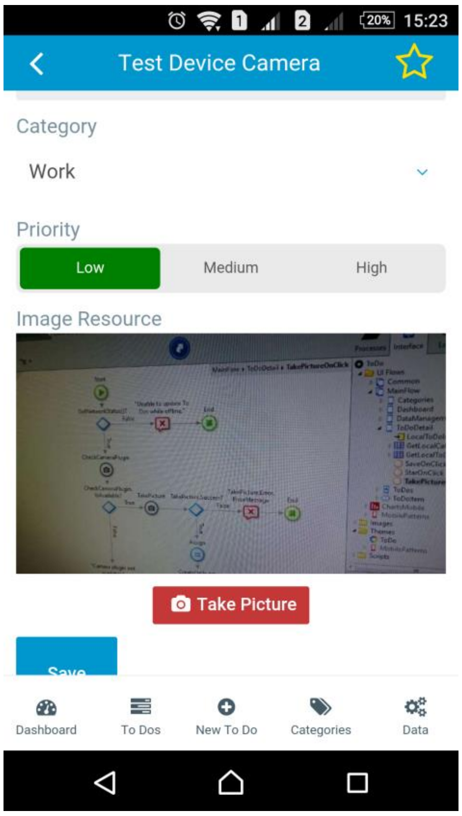


Figure 19. Camera Plugin picture

# Part 3: Create a new Plugin

In this part of the exercise, you will create a new plugin.

**1.** Create a new Application.

**a)** Switch to the Applications tab.

**b)** Click the 'New Application' icon to create a new application.

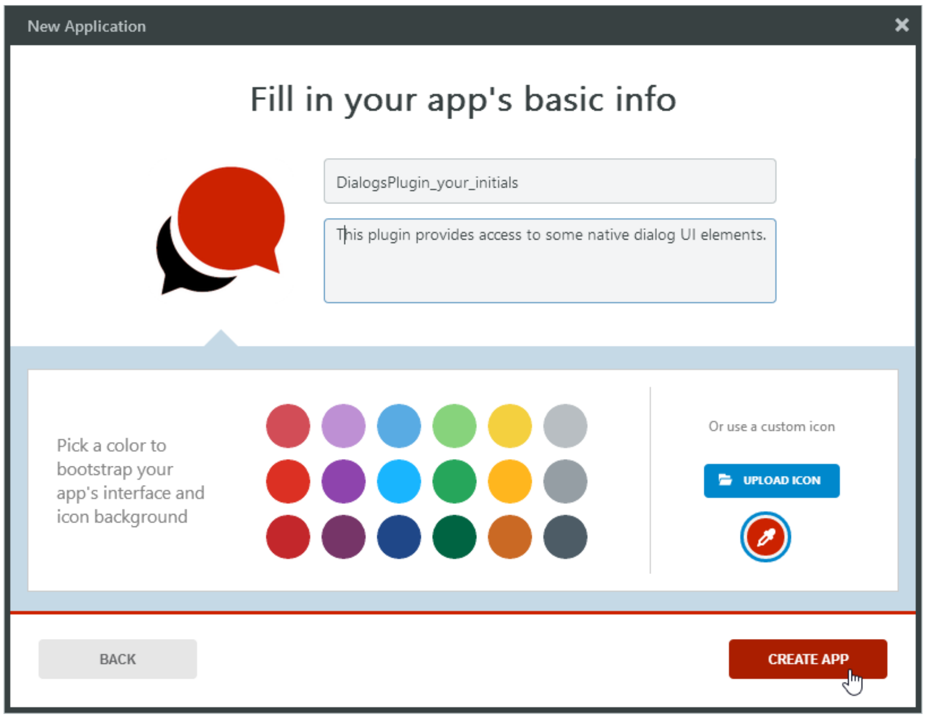
**c)** In the **New Application** dialog choose 'Mobile App', then click Next.

**d)** Select the **Phone** template then click Next.

**e)** Set the application **Name** to 'DialogsPlugin_<your_initials>', and fill in the **Description**.

**f)** Click the 'Upload Icon' and choose the **dialogs-icon.png** from the Resources folder.

**g)** Click the **Create App** Button to create the new application.

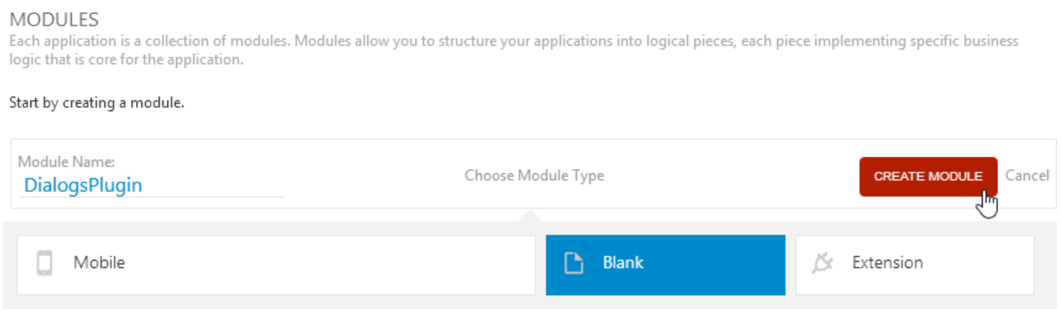

New Application
Fill in your app's basic info
Pick a color to bootstrap your app's interface and icon background
Or use a custom icon
UPLOAD ICON
BACK
CREATE APP

DialogsPlugin_your_initials

This plugin provides access to some native dialog UI elements.

Figure 20. Create DialogsPlugin application

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


MODULES
Each application is a collection of modules. Modules allow you to structure your applications into logical pieces, each piece implementing specific business logic that is core for the application.

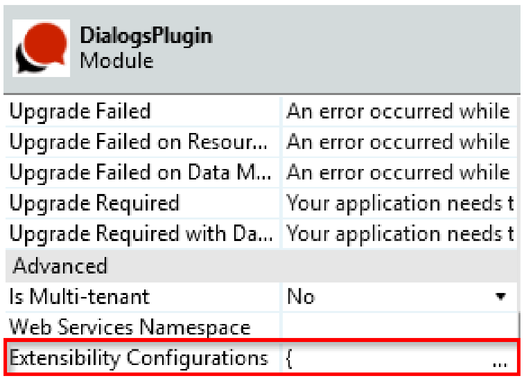
Start by creating a module.

Module Name: DialogsPlugin

Figure 21. Create Blank module

**2.** Reference the **DialogsPlugin** repository and modify the module icon.

**a)** Select the **DialogsPlugin** module in the Elements Area, then double-click the **Extensibility Configurations** property to edit it.


DialogsPlugin
Module

| | |
|---|---|
| Upgrade Failed | An error occurred while |
| Upgrade Failed on Resour... | An error occurred while |
| Upgrade Failed on Data M... | An error occurred while |
| Upgrade Required | Your application needs t |
| Upgrade Required with Da... | Your application needs t |
| Advanced | |
| Is Multi-tenant | No |
| Web Services Namespace | |
| Extensibility Configurations | { ... |

Figure 22. Create Blank module

**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 'dialogs-icon32.png' image file from the Resources folder.

**3.** Create the **CheckDialogsPlugin** Client Action to verify if the **DialogsPlugin** has been loaded.

**a)** Switch to the **Logic** tab.

**b)** Right-click the **Client Actions** folder and choose 'Add Client Action'.

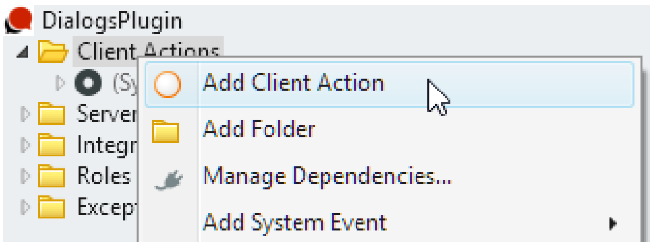


Figure 23. Add Client Action

**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'.

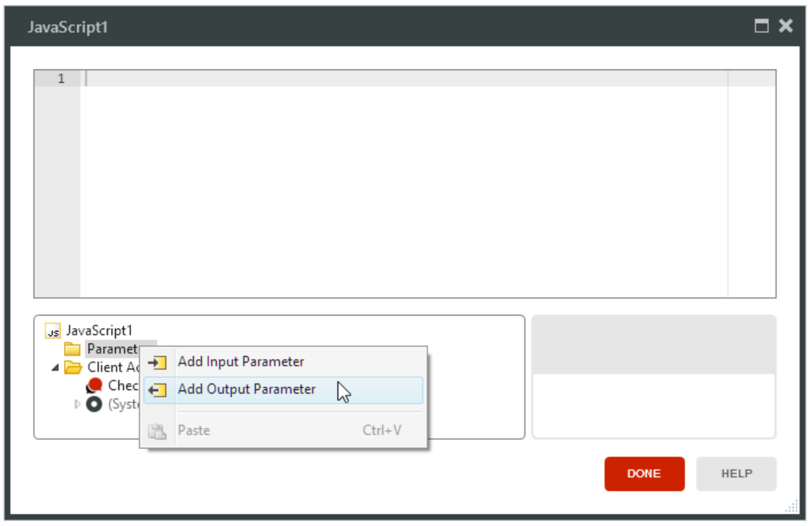


Figure 24. Add Output Parameter inside JavaScript statement

**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 that will display a native alert with a customized Message and Button.

**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 is **Name** to 'Message', **Data Type** to 'Text', and **Is Mandatory** to 'Yes'.

**e)** Add another Input Parameter and set is **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)** 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";

    }
```

**o)** Click **Done** to close the JavaScript editor.

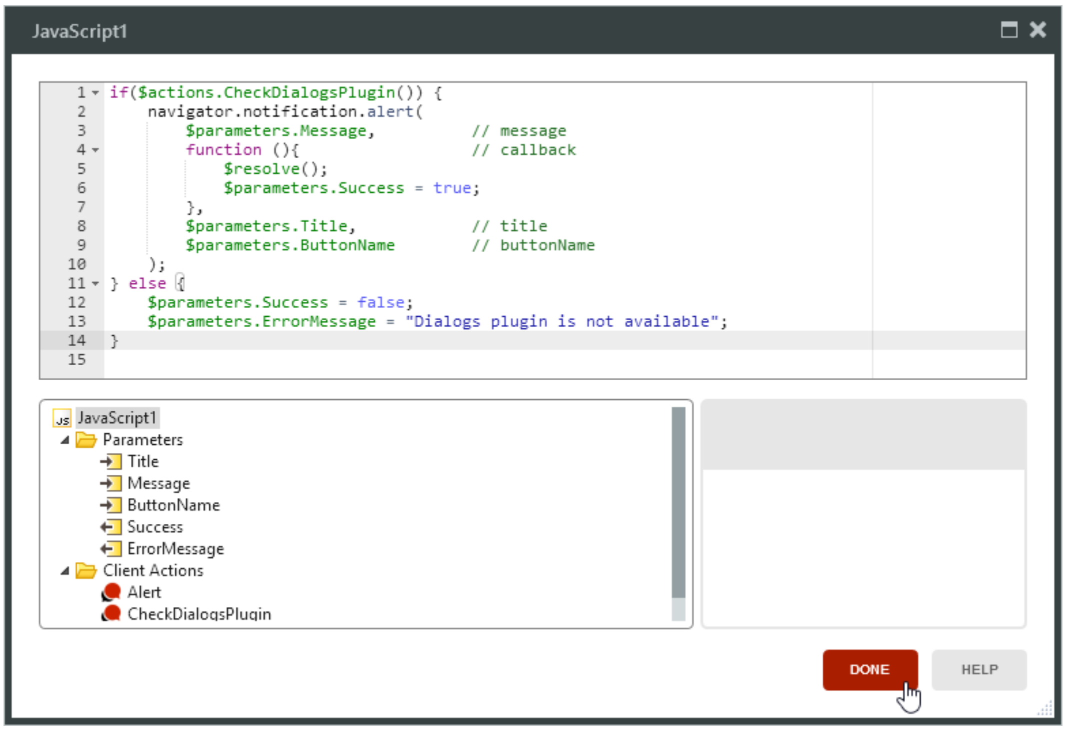
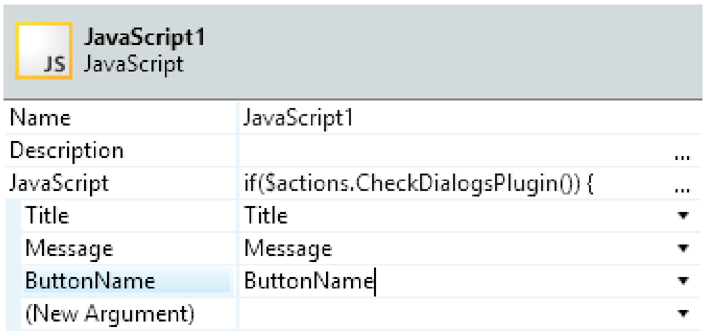

```
if($actions.CheckDialogsPlugin()) {
    navigator.notification.alert(
        $parameters.Message,          // message
        function (){                  // callback
            $resolve();
            $parameters.Success = true;
        },
        $parameters.Title,            // title
     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

Figure 25. JavaScript editor

**p)** Select the **JavaScript** statement and define the Input Parameters to use the corresponding Client Action Input Parameters.


JavaScript1
JavaScript

| | |
|---|---|
| Name | JavaScript1 |
| Description | ... |
| JavaScript | if($actions.CheckDialogsPlugin()) { ... |
| Title | Title |
| Message | Message |
| ButtonName | ButtonName |
| (New Argument) | |

Figure 26. JavaScript parameters

**q)** 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 confirm a user action.

**a)** Create a new Client Action named ‘Confirm’.

**b)** Set the **Public** property to ‘Yes’ and change the Client Action **Icon** to the ‘dialogs-icon32.png’ image, located in the **Resources** folder.

**c)** Add a new Input Parameter named ‘Title’ to the Client Action, and set its **Data Type** to ‘Text’.

**d)** Repeat the previous step for the ‘Message’ Input Parameter.

**e)** Add an Output Parameter named ‘Confirmed’ with **Data Type** set to ‘Boolean’.

**f)** Repeat the previous step for the ‘Success’ Output Parameter.

**g)** Add another Output Parameter named ‘ErrorMessage’ with **Data Type** set to ‘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)** Add an Input Parameter to the JavaScript statement named ‘Title’ with **Data Type** set to ‘Text’.

**k)** Repeat the previous step for the ‘Message’ Input Parameter of the JavaScript statement.

**l)** Add an Output Parameter named ‘Confirmed’ with **Data Type** set to ‘Boolean’.

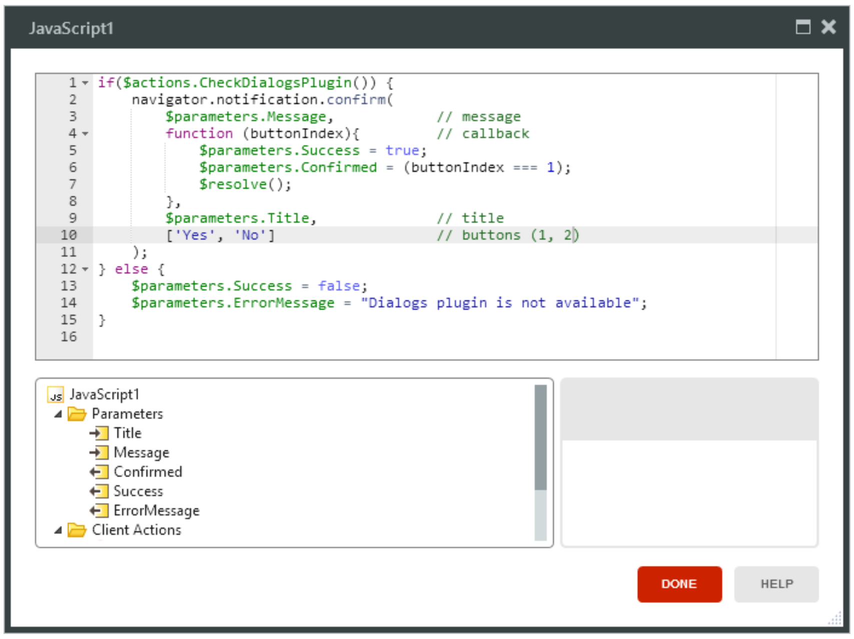
**m)** Repeat the previous step for the ‘Success’ Output Parameter.

**n)** Add another Output Parameter named ‘ErrorMessage’ with **Data Type** ‘Text’.

**o)** 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";

}
```

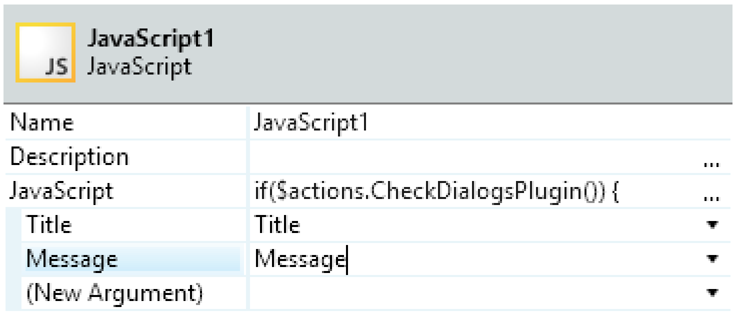
**p)** Click **Done** to close the JavaScript editor.



```
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 (1, 2)
    );
} else {
    $parameters.Success = false;
    $parameters.ErrorMessage = "Dialogs plugin is not available";
}
```

Figure 27. Confirm JavaScript editor

**q)** Set the **Title** and **Message** properties of the JavaScript statement to the Input Parameters of the Client Action.



| | |
|---|---|
| Name | JavaScript1 |
| Description | |
| JavaScript | if($actions.CheckDialogsPlugin()) { |
| Title | Title |
| Message | Message |
| (New Argument) | |

Figure 28. Confirm JavaScript properties

**r)** 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
```

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

# Part 4: Test the Dialogs Plugin

In this part of the exercise, you will test the Dialogs Plugin in the **ToDo** application.

**1.** Add references to the new Actions, created in 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 **Ok** to close the dialog.

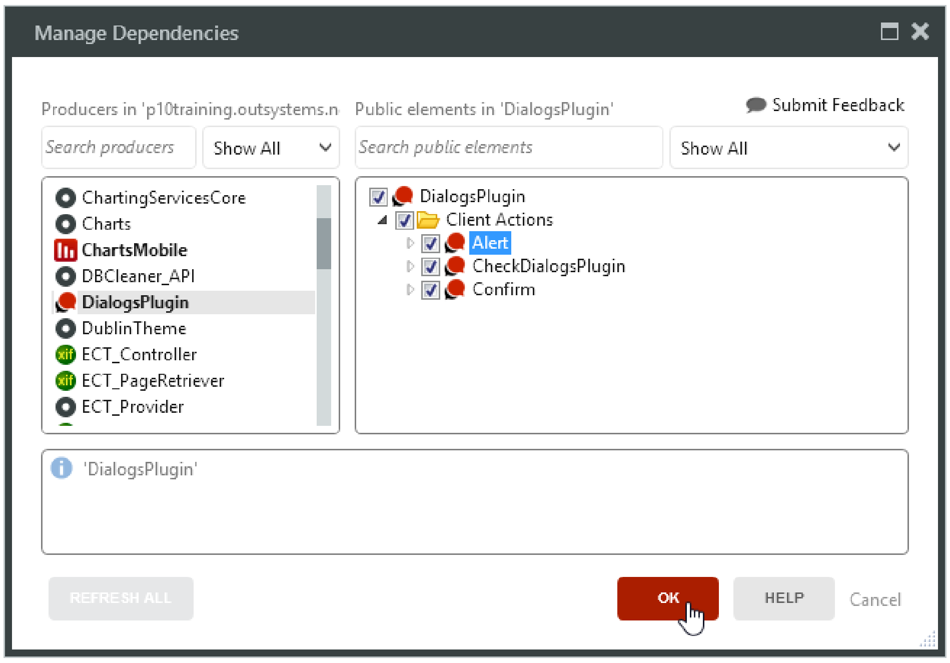


Figure 29. DialogsPlugin dependencies

**2.** Change the 'Clear Local Storage' Button to present an Alert dialog instead of a Message.

**a)** Open the **ClearLocalStorageOnClick** Client Action of the **DataManagement** Screen.

**b)** Drag a **Run Client Action** statement and drop it between the Start and **ClearAllLocalStorage** statements.

**c)** In the **Select Action** dialog, choose the **CheckDialogsPlugin** Client Action from the **DialogsPlugin** module.

**d)** Drag an **If** Widget and drop it between the **CheckDialogsPlugin** and **ClearAllLocalStorage** statements, 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** statement.

**h)** Drag an **If** Widget and drop it on the right of the **Confirm** statement, then create the connector between both.

**i)** Set the **Condition** property of the **If** to

```
Confirm.Success
```

**j)** Drag another **If** Widget 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 **ClearAllLocalStorage** statement.

**m)** Create the **True** branch connector from the 'Confirm.Confirmed?' **If** to the **ClearAllLocalStorage** statement.

**n)** Drag an **End** node and drop it on the right of the 'Confirm.Confirmed?' **If** Widget, then create the **False** branch connector from the **If** to the End.

**NOTE:** With the condition above the flow will reach the new End statement when the **DialogsPlugin** was successful and the user has chosen 'No' in the Confirm dialog.

**o)** Drag another **If** Widget and drop it between the **ClearAllLocalStorage** and **Message** statement, then set the **Condition** property to ‘`CheckDialogsPlugin.IsAvailable`’.

**p)** Drag a **Run Client Action** statement and drop it on the right of the **If** statement.

**q)** In the **Select Action** dialog choose the **Alert** Client Action from the **DialogsPlugin** module.

**r)** Create the **True** branch connector from the **If** to the **Alert** statement.

**s)** Set the **Title** property of the **Alert** statement to “Local Storage”, the **Message** property to “All data cleared!”, and the **ButtonName** to “Dismiss”.

**t)** Drag an **If** Widget and drop it on the right of the **Alert** statement, and create the connector between both.

**u)** Set the **Condition** property of the If to ‘Alert.Success’.

**v)** 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.

**w)** Create the **False** branch connector from the **If** to the existing **Message** statement.

**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.

**x)** Your **ClearLocalStorageOnClick** Client Action should look like this

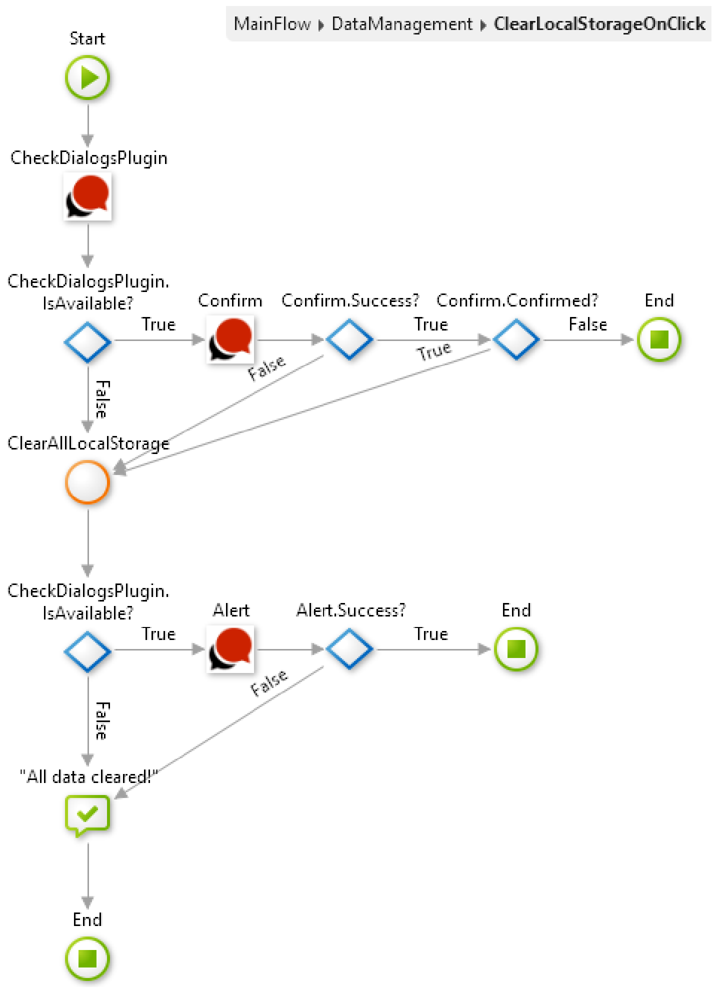


Figure 30. ClearLocalStorageOnClick

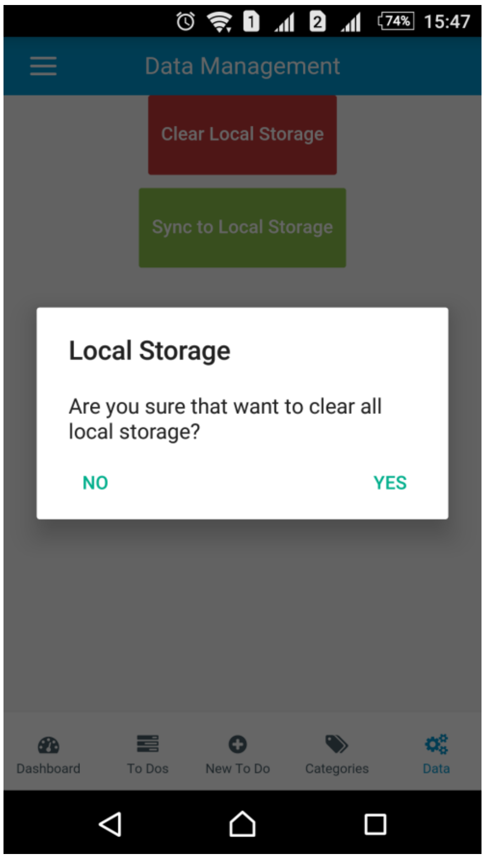
**3.** Publish and test the application.

**a)** Generate a new native application for your device.

**NOTE:** When a new plugin is added to the application, a new native application is generated both for iOS and Android. You can see that in the steps of the 1-Click Publish tab.

**b)** After the generate process is complete, install the new application in the device.

**c)** Navigate to the **Data Management** Screen and press the 'Clear Local Storage', you should see the native confirmation dialog.



Local Storage

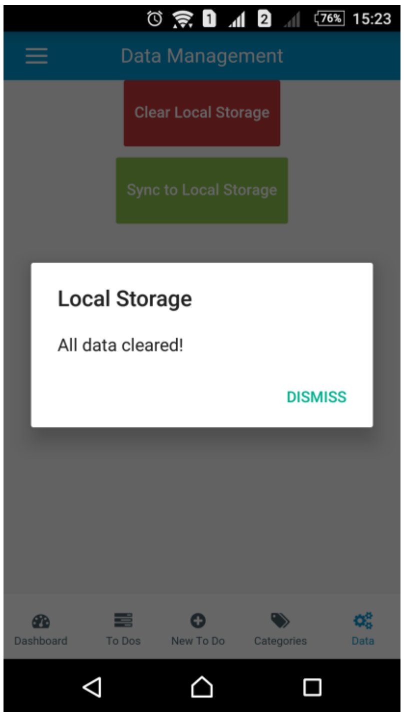
Are you sure that want to clear all local storage?

NO YES

Figure 31. Android Confirm dialog

**d)** If you select ‘No’, nothing will be changed.

**e)** Select ‘Yes’, and then you should see the **Alert** dialog.



Local Storage

All data cleared!

DISMISS

Figure 32. Android native alert

## End of Lab

In this exercise, you installed the **Camera Plugin** from Forge.

After installing the plugin, you modified the **ToDoDetail** Screen to make it possible for the users to add pictures taken with the camera to the To Dos.

You have also created a new plugin named **DialogsPlugin**. This plugin provides access to some native UI elements, including alerts and prompts. The **DialogsPlugin** was then integrated in the **ToDo** app, by showing alerts instead of messages.

In most occasions, the users of your mobile app will not have to update it manually, after installing it in their devices. This happens because OutSystems will automatically push the updates to their devices, when you publish a new mobile app version. In this case, since a new plugin was added to the application, users will have to install a new mobile app package. Check https://success.outsystems.com/Documentation/10/Delivering_Mobile_Apps/Mobile_App_Update_Scenarios for more information.

# List of Figures

Here is the list of screenshots and pictures used in this exercise.