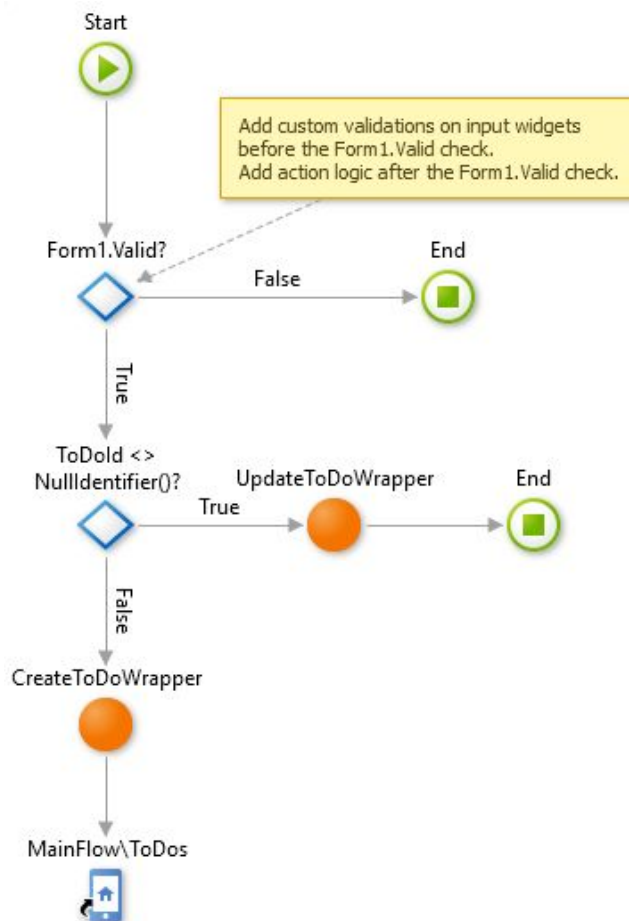


Logic and Code Reusability Exercise



Introduction

In this exercise Lab, we will create two new Server Actions. These Actions are a bit different from the ones we have been creating so far. They will be created under the Logic tab, meaning that will not be associated to any Screen. This means that these Action can be used in multiple context of the module, namely in a Screen Action. They can also be set as Public and shared to other modules.

Our new Server Actions will be created in the `ToDo_Core` module, and will be used to encapsulate the logic for creating new Todos and for updating existing Todos.

Then, we will make the `ToDo` Entity as read-only, which will make the Create and Update Entity Actions not available in the consumer modules. As a matter of fact, the only Entity Action available in consumer modules will be the **GetToDo**.

Thus, the only way to add changes to the `ToDo` Entity, from a consumer module, will be by calling the Wrapper Actions (to add and update Todos). This means that it will not be possible to delete a `ToDo` from a consumer module. To support that, new Actions would be needed to provide to the consumer modules those features.

This pattern is actually an OutSystems Best Practice. First, we avoid that the Entity is exposed with write permissions to any consumer module, without any security or other logic around it. Second, by creating these Actions, we can add additional business logic inside them, that by calling the Entity Actions directly we would not have. As an example, we can add validation rules (e.g: user has permissions to create a `ToDo`?) or exception handling to these Actions.

In a future lab, we will go back to these Actions and add more business logic to it. In this specific lab, we will focus on making the changes necessary to modify the exposure of the Entity from write to read-only permissions.

In this specific exercise lab, we will:

- Create two reusable Server Actions in the `ToDo_Core` module
- Make the `ToDo` Entity as read-only
- Replace the usages of the `CreateToDo` and `UpdateToDo` Entity Actions in the `ToDo` module, by the new Wrappers
- (Optional) Create a Server Action for creating / updating Resources in the database

Table of Contents

Introduction	2
Table of Contents	3
Create an Action to create a To Do	4
Create an Action to update a To Do	9
Make the ToDo Entity Read-only	11
(Challenge) Add an Action to Create / Update Resources	20
End of Lab	21

Create an Action to create a To Do

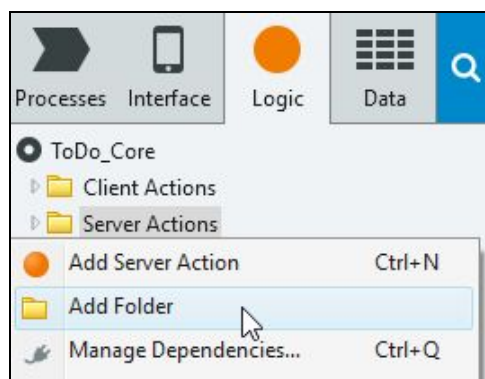
We will start this Lab by creating a new Server Action. This one will be defined under the Logic tab of Service Studio. Here, we have all the reusable Server and Client Actions, meaning that they can be used in different contexts and scopes of the module.

This particular Server Action will be created in the **ToDo_Core** module and will be used to create a new **ToDo** in the database. This functionality is already available with the **CreateToDo** (or **CreateOrUpdateToDo**) Entity Action, however the **ToDo** Entity is being exposed to the consumer modules with write permissions. In OutSystems, it is best practice to expose the Entities as Read-only and provide some public Actions with all the functionalities we want the consumer modules to use. This way, not only we can control which Actions to expose (instead of all Entity Actions), as well as we can add business logic around it.

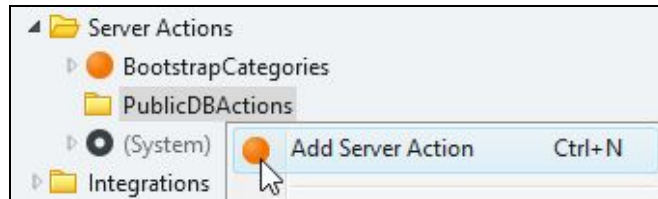
The Entity Actions are a black box, where we don't have access to its implementation. However, by using them in a Server Action Wrapper, we can build logic around it, including for instance Exception Handling, or data / permissions validations. This is also valid for Client Actions.

In this first part of the Lab, we will create a Server Action that will use the **CreateToDo** Entity Action. This Action will later be used in the **ToDo** module.

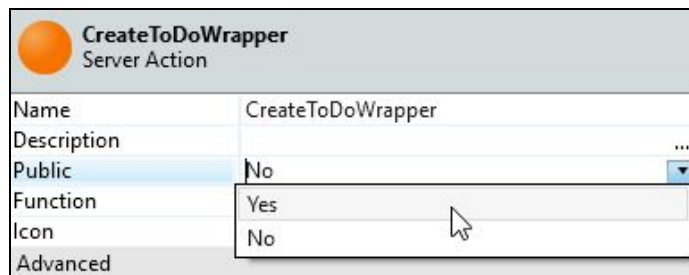
- 1) Add a Public Server Action to the **ToDo_Core** module named *CreateToDoWrapper*. Add eight Input Parameters: *Title* (Text, mandatory), *IsStarred* (Boolean), *Notes* (Text), *DueDate* (Date), *CreatedDate* (Date, mandatory), *CompletedDate* (Date), *CategoryId* (Category Identifier, mandatory) and *PriorityId* (Priority Identifier, mandatory). Also add an Output Parameter *ToDoId*, of type *ToDo Identifier*, to return the Id of the Record created in the database. Create this Action inside a new Folder called *Public DB Actions*.
 - a) Open the **ToDo_Core** module.
 - b) Switch to the **Logic** tab, right-click the **Server Actions** folder and select **Add Folder**. Set its Name to *PublicDBActions*.



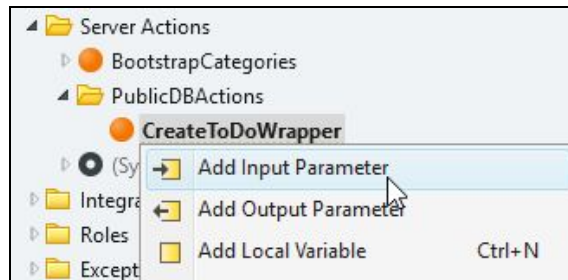
- c) Right-click on the new Folder and select **Add Server Action**. Call the Action *CreateToDoWrapper*.



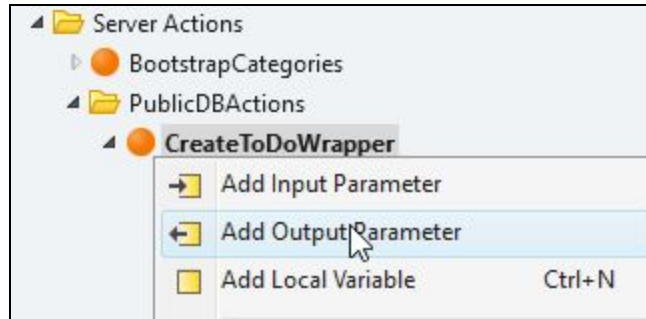
- d) Set the Action's **Public** property to *Yes*.



- e) Add an **Input Parameter** to the **CreateToDoWrapper** Action. Set its **Name** to *Title* and its **Data Type** to *Text*. Set it as mandatory.

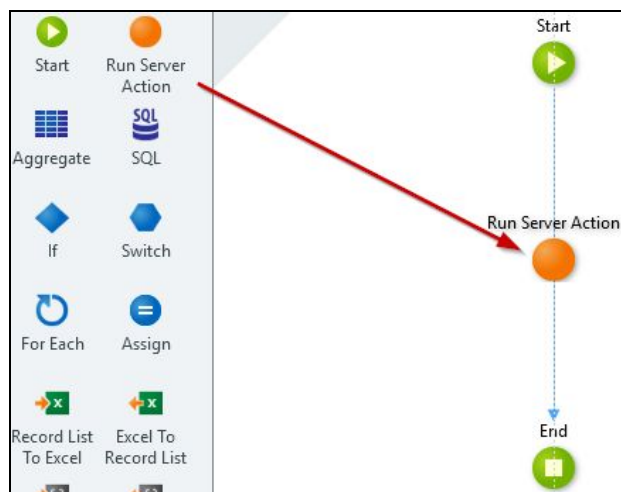


- f) Repeat this process for the *IsStarred*, *Notes*, *DueDate*, *CreatedDate*, *CompletedDate*, *CategoryId* and *PriorityId*. Keep the **Data Types** suggested by the platform and set the **IsStarred**, **Notes**, **DueDate** and **CompletedDate** as non-mandatory.
- g) Add an Output Parameter to the Action. Set its **Name** to *ToDold* and its **Data Type** to *ToDo Identifier*.

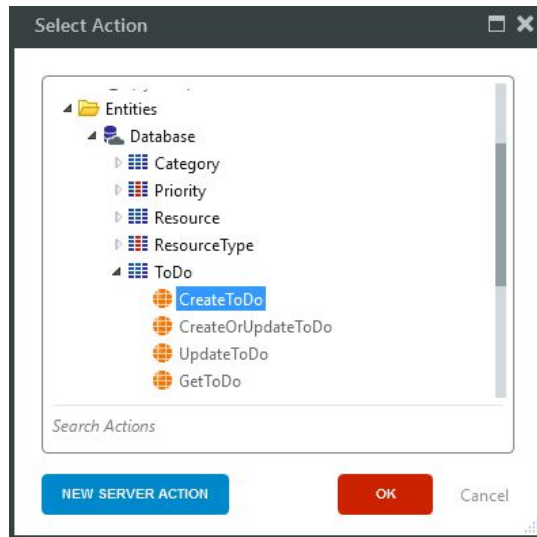


NOTE: This Action has a considerable number of Input Parameters, which represent the necessary attributes to pass the information to the server to create a new ToDo. As an alternative, we could pass an Input Parameter of type ToDo, with all information encapsulated in the Record. However, it is an OutSystems Best Practice to not pass records as Inputs, to avoid passing redundant information and large records. For instance, to create a ToDo we do not need the Id or even the UserId, since those are created within the wrapper.

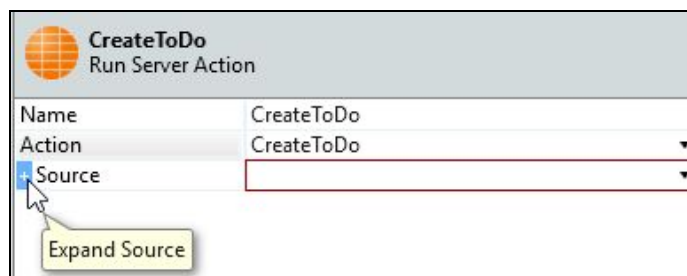
- 2) Now that we have the Action created, it's time to define the logic within it. Initially, this Action will add a ToDo record to the database, using the **Create** Entity Action and the input parameters of the **CreateToDoWrapper** Server Action.
 - a) Drag and drop a **Run Server Action** statement to the Action flow.



- b) In the Select Action dialog, choose the **CreateToDo** Entity Action.



- c) In the **Source** property of the Action, select the Expand Source option, by clicking on the empty space right before the **Source**.



This allows creating a ToDo Record, without actually passing a record as Source, but still a value for each one of the attributes of the record.

NOTE: An alternative to this would require creating a **Local Variable of type ToDo**, add an **Assign** before the CreateToDo call to assign each attribute of the Local Variable ToDo to the values of the Input Parameters of the CreateToDoWrapper, and then pass the Local Variable as **Source** of the CreateToDo.

- d) Assign all the attributes under the **Source** to the respective Input Parameters.

CreateToDo Run Server Action	
Name	CreateToDo
Action	CreateToDo ▼
Source	
Title	Title ▼
UserId	▼
IsStarred	IsStarred ▼
Notes	Notes ▼
DueDate	DueDate ▼
CreatedDate	CreatedDate ▼
CompletedDate	CompletedDate ▼
CategoryId	CategoryId ▼
PriorityId	PriorityId ▼

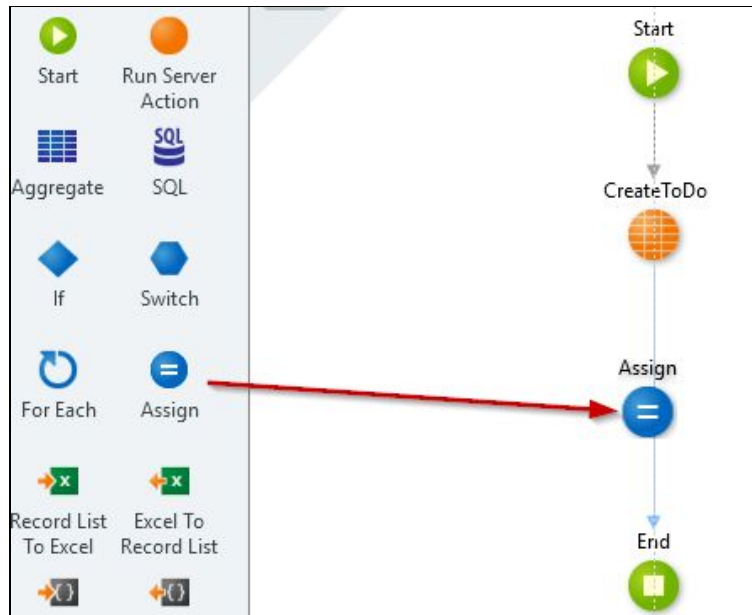
Notice that the Id does not appear in the list of attributes. The reason for that is that since we are using the Create Action, and the Id is Auto-Number, the Action will automatically create a unique Id for the Record.

- e) Assign the UserId value to the user currently logged in:

GetUserId()

CreateToDo Run Server Action	
Name	CreateToDo
Action	CreateToDo ▼
Source	
Title	Title ▼
UserId	GetUserId() ▼
IsStarred	IsStarred ▼
Notes	Notes ▼
DueDate	DueDate ▼
CreatedDate	CreatedDate ▼
CompletedDate	CompletedDate ▼
CategoryId	CategoryId ▼
PriorityId	PriorityId ▼

- f) Drag an Assign and drop it after the **CreateToDo** Action.



g) Set the assignment to:

ToDoId = CreateToDo.Id

This assigns the *Id* of the ToDo created to the Output Parameter.

	ToDoId Assign
Label	
Assignments	
ToDoId	
x.y = CreateToDo.Id	

Create an Action to update a To Do

In this section of the exercise, we will follow the same strategy of the previous step, and create an Action to update an existing ToDo in the database.

Just like in the previous section, the update functionality is already available with the UpdateToDo (or CreateOrUpdateToDo) Entity Action, however since the Entity will not be exposed with write permissions, the update wrapper needs to be done as well.

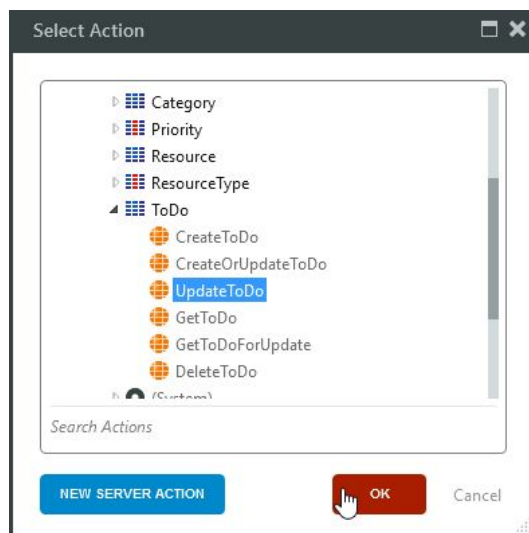
- 1) Add a Public Server Action to the **ToDo_Core** module named *UpdateToDoWrapper*. Add nine Input Parameters: *ToDoId* (ToDo Identifier, mandatory), *Title* (Text, mandatory), *IsStarred* (Boolean), *Notes* (Text), *DueDate* (Date), *CreatedDate* (Date, mandatory),

CompletedDate (Date), *CategoryId* (Category Identifier, mandatory) and *PriorityId* (Priority Identifier, mandatory). Add this Action to the **Public DB Actions** folder.

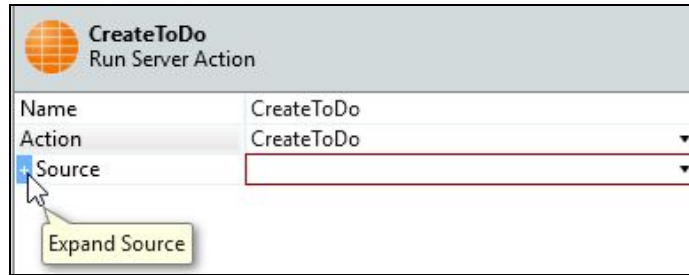
- a) Right-click on the **PublicDBActions** Folder and select **Add Server Action**. Call the Action *UpdateToDoWrapper*.
- b) Set the Action's **Public** property to *Yes*.
- c) Add the eight **Input Parameters** to the **UpdateToDoWrapper** Action. Add them in the following order: *ToDold*, *Title*, *IsStarred*, *Notes*, *DueDate*, *CreatedDate*, *CompletedDate*, *CategoryId* and *PriorityId*. Keep the DataTypes suggested by the platform and set the **IsStarred**, **Notes**, **DueDate** and **CompletedDate** as non-mandatory.

NOTE: This Action has a slight difference in terms of the Input Parameters, when compared with the *CreateToDoWrapper*. Here, we have the *ToDold*, to identify the *ToDo* being updated.

- 2) Implement the logic to update a *ToDo* in the database, using the **Update** Entity Action and the input parameters of the *UpdateToDoWrapper* Server Action.
 - a) Drag and drop a **Run Server Action** statement to the Action flow.
 - b) In the Select Action dialog, choose the **UpdateToDo** Entity Action.



- c) In the **Source** property of the Action, select the Expand Source option, by clicking on the empty space right before the **Source**.



This allows creating a ToDo Record, without actually passing a record as Source, but still a value for each one of the attributes of the record.

NOTE: An alternative to this would require creating a **Local Variable of type ToDo**, add an **Assign** before the CreateToDo call to assign each attribute of the Local Variable ToDo to the values of the Input Parameters of the CreateToDoWrapper, and then pass the Local Variable as **Source** of the CreateToDo.

- d) Assign all the attributes under the **Source** to the respective Input Parameters. Don't forget to assign the **UserId** value to the user currently logged in.

Make the ToDo Entity Read-only

Now that the Actions were created, we can continue implementing the pattern by exposing the ToDo Entity as Read-only. This way, every consumer module of this Entity will only have access to the **GetToDo** Entity Action.

Since we are changing the producer module, the consumer (ToDo) will be outdated. So, we need to go back to the ToDo module and refresh the dependencies to the Entity, as well as add the new dependency to the **CreateToDoWrapper** and **UpdateToDoWrapper** Actions.

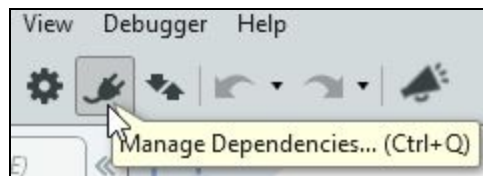
Then, we can use these new Actions to add / update ToDos in the database.

- 1) Change the **Expose Read Only** property of the ToDo Entity to Yes and publish the ToDo_Core module.

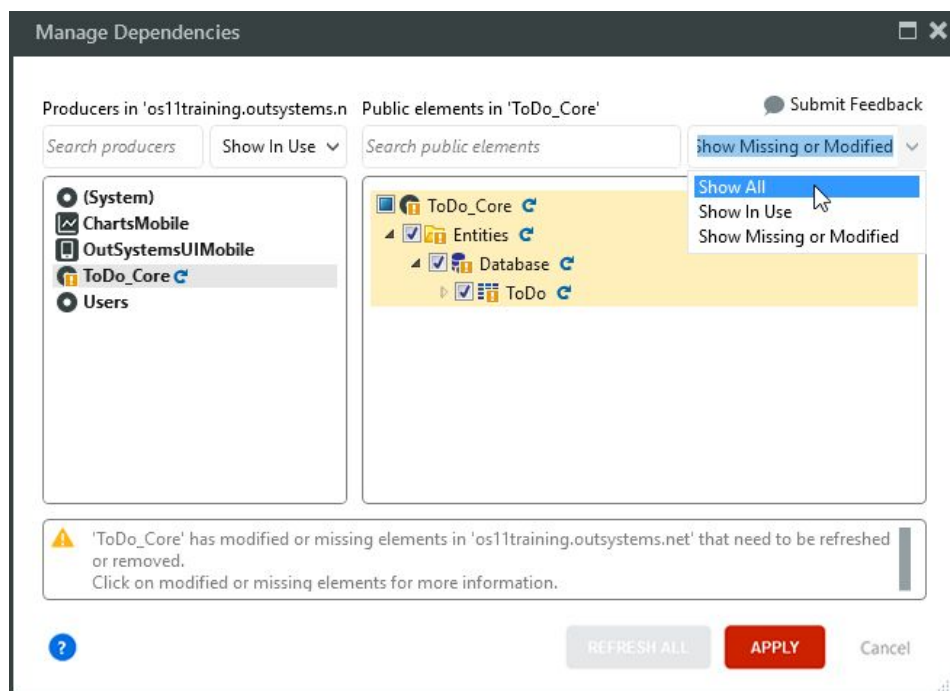
ToDo Entity	
Name	ToDo
Description	...
Public	Yes
Expose Read Only	No
Indexes	Yes
More...	No

- 2) Now that the Entity is exposed as read-only, we need to go to the ToDo module, refresh the dependencies and add the new ones (**CreateToDoWrapper** and **UpdateToDoWrapper** Actions). This will cause some errors that we need to fix, since the CreateOrUpdateToDo Action is no longer available in the ToDo module.

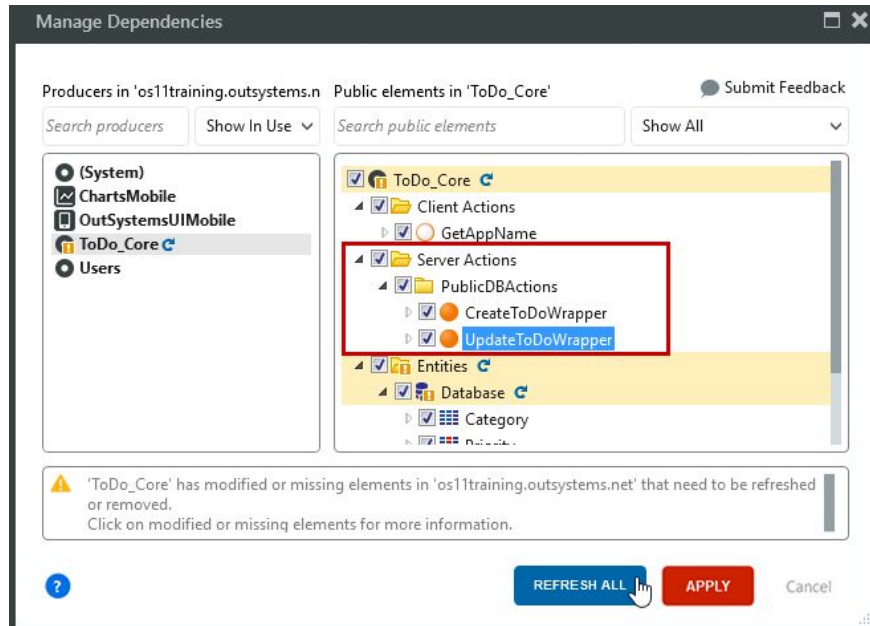
- Open the ToDo module.
- Open the Manage Dependencies window.



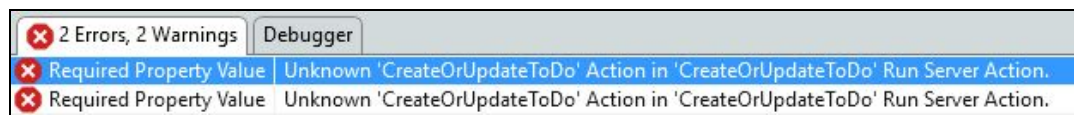
- Select the ToDo_Core on the Producers section on the left. Then, on the Public elements on the right, select the Show All option in the dropdown.



- d) Select the **Wrapper** Actions and **Refresh All** dependencies. Click **Apply** to exit. This will cause some errors that we need to fix.

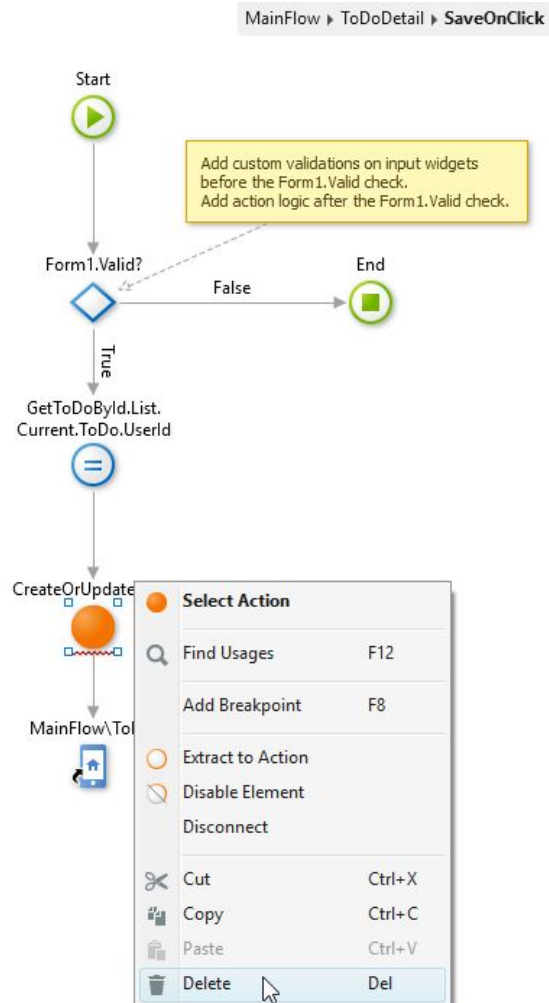


- 3) Let's fix the first error in the module. Replace the usage of the **CreateOrUpdateToDo** Entity Action, in the **SaveOnClick** Action of the **ToDoDetail** Screen, with the new wrappers. Do not forget that we were using the **CreateOrUpdate** Action and now we have one to **Create** and one to **Update**, so adjust the logic accordingly.
- a) Double-click on the first error to open its location in the module.

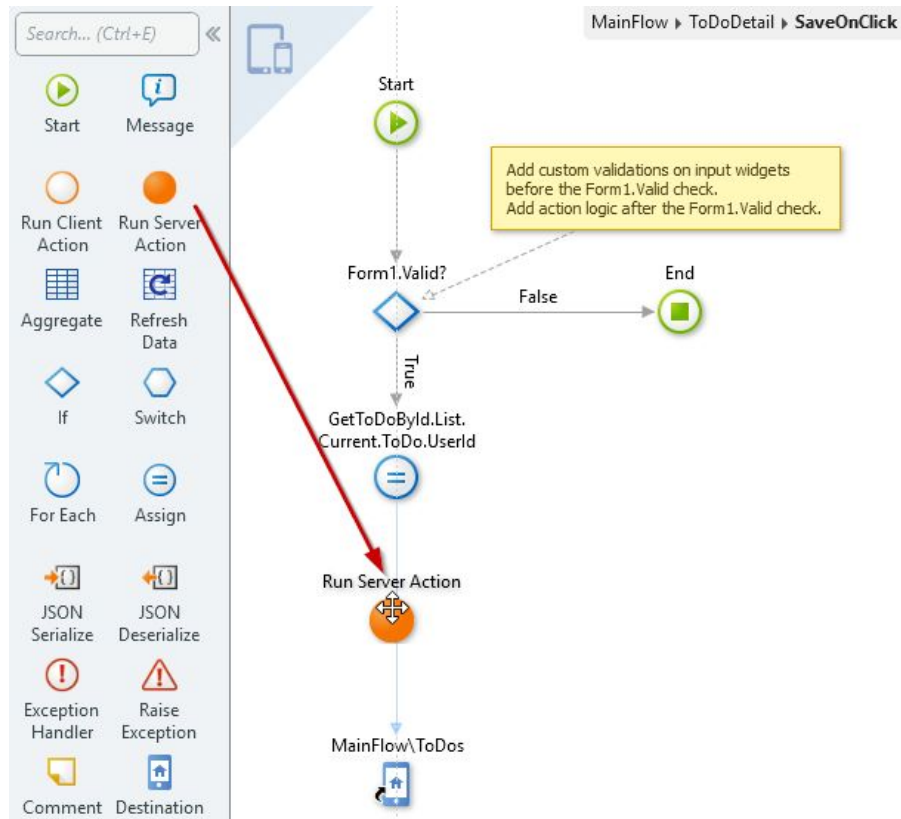


NOTE: Since the **ToDo** Entity is Read-only, we don't have access to its **CreateOrUpdate** Entity Action in the consumer module. So, in the **SaveOnClick** Action of the **ToDoDetail** Screen we have the error that this Action is unknown.

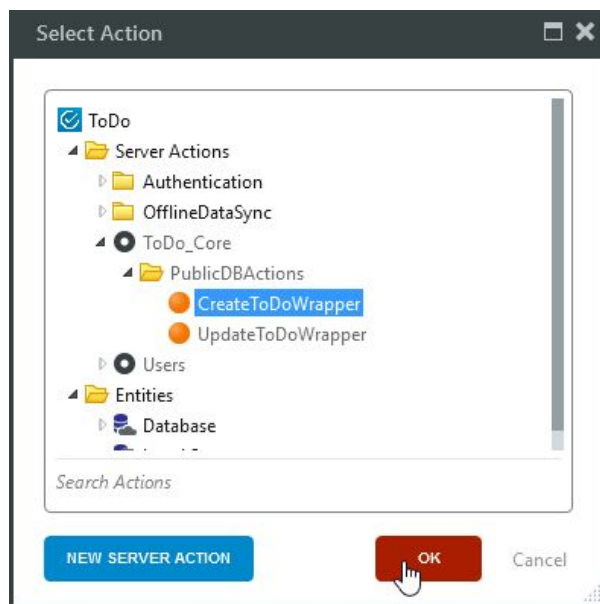
- b) Delete the call to the **CreateOrUpdateToDo** Action from the flow.



- c) Drag a new Run Server Action statement below the Assign and before the End.



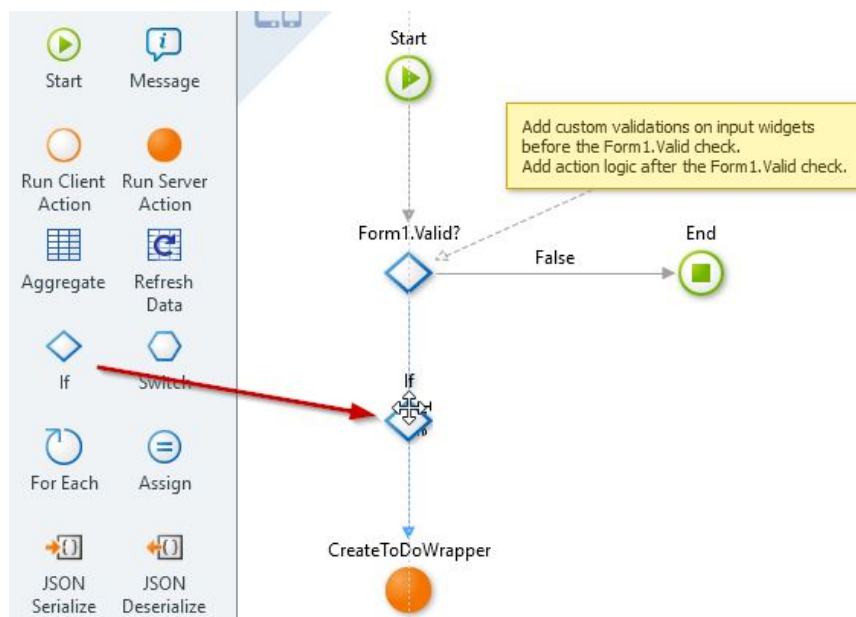
d) In the Select Action dialog, select the **CreateToDoWrapper** Action.



e) Set the Inputs of the Action as in the following screenshot. This guarantees that we are getting the information from the input widgets in the Form of the ToDoDetail Screen, submitted by the user.

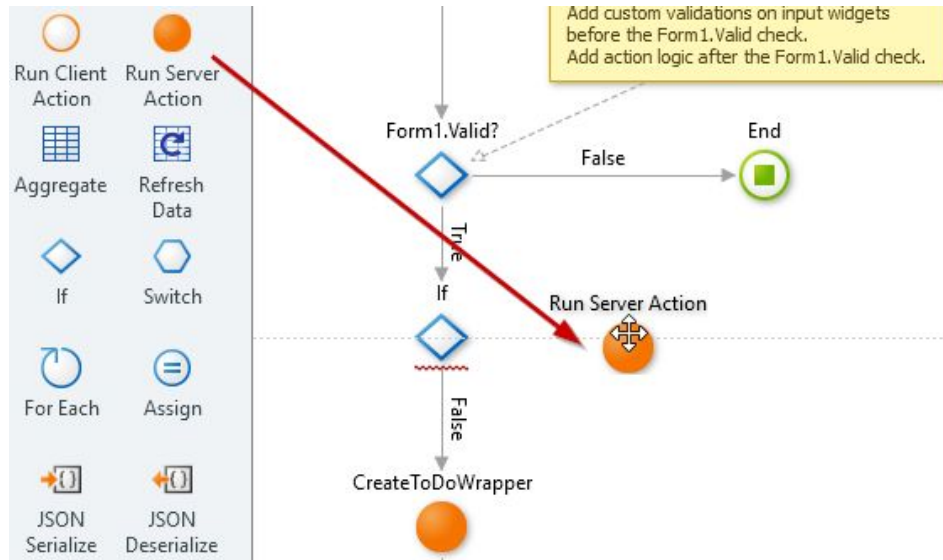
CreateToDoWrapper Run Server Action	
Name	CreateToDoWrapper
Server Request Timeout	(Module Default Timeout)
Action	PublicDBActions\CreateToDoWrapper ▼
Title	GetToDoById.List.Current.ToDo.Title ▼
IsStarred	GetToDoById.List.Current.ToDo.IsStarred ▼
Notes	GetToDoById.List.Current.ToDo.Notes ▼
DueDate	GetToDoById.List.Current.ToDo.DueDate ▼
CreatedDate	GetToDoById.List.Current.ToDo.CreatedDate ▼
CompletedDate	GetToDoById.List.Current.ToDo.CompletedDate ▼
CategoryId	GetToDoById.List.Current.ToDo.CategoryId ▼
PriorityId	GetToDoById.List.Current.ToDo.PriorityId ▼

- f) The Assign in the **SaveOnClick** Action is not needed anymore, since it is done inside the Wrapper Action. So, let's delete it.
- g) This Action just creates a ToDo in the database. So, we need to make sure that the use case of updating an existing ToDo is also covered. Drag an **If** statement and drop it before the **CreateToDoWrapper**.




- h) Set the If condition to

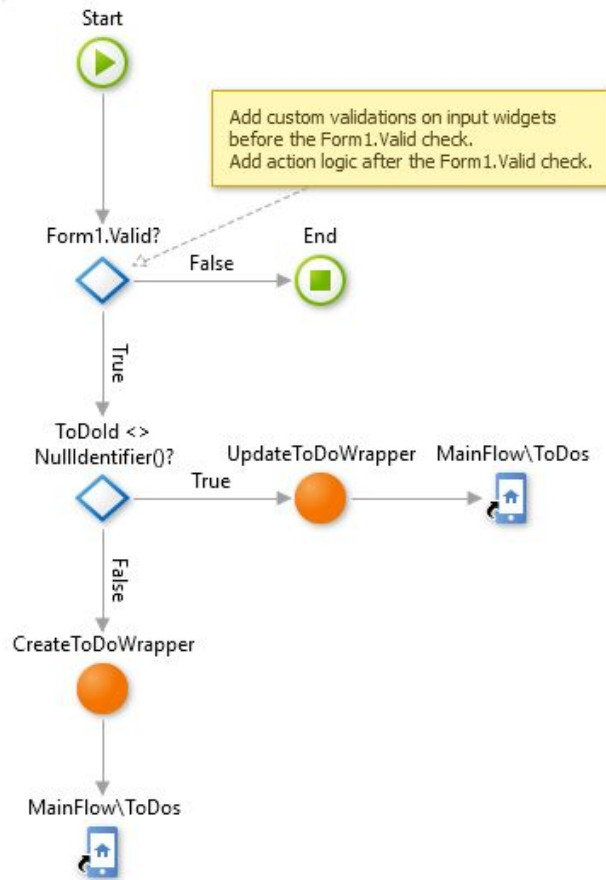
$$ToDoId \neq NullIdentifier()$$
- i) Drag a **Run Server Action** and drop it on the right of the recently added If. Select the **UpdateToDoWrapper** Action in the dialog that appears.



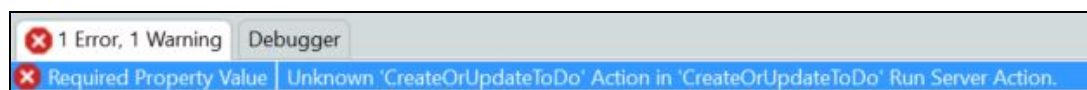
- j) Drag a new Destination statement and drop it on the right of the UpdateToDoWrapper Action, and set the Destination to the Todos Screen. Connect all the statements together to create a flow.
- k) Set the Input Parameters for the **UpdateToDoWrapper** accordingly

 UpdateToDoWrapper Run Server Action	
Name	UpdateToDoWrapper
Server Request Timeout	(Module Default Timeout)
Action	PublicDBActions\UpdateToDoWrapper ▼
ToDold	ToDold ▼
Title	GetToDoByld.List.Current.ToDo.Title ▼
IsStarred	GetToDoByld.List.Current.ToDo.IsStarred ▼
Notes	GetToDoByld.List.Current.ToDo.Notes ▼
DueDate	GetToDoByld.List.Current.ToDo.DueDate ▼
CreatedDate	GetToDoByld.List.Current.ToDo.CreatedDate ▼
CompletedDate	GetToDoByld.List.Current.ToDo.CompletedDate ▼
CategoryId	GetToDoByld.List.Current.ToDo.CategoryId ▼
PriorityId	GetToDoByld.List.Current.ToDo.PriorityId ▼

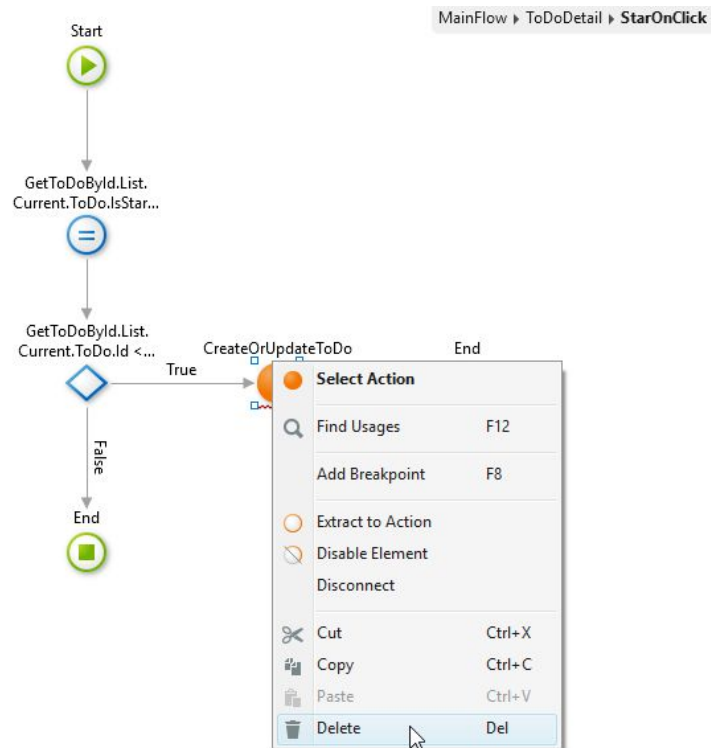
- l) The Action flow should look like the following screenshot



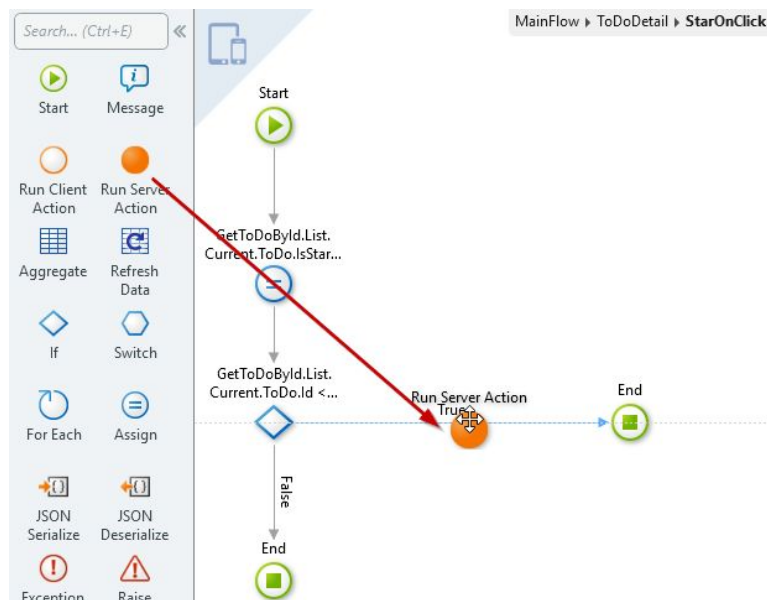
- 4) Following the same strategy, now we will fix the last error in the module. Replace the usage of the **CreateOrUpdateToDo** Entity Action, in the **StarOnClick** Action of the **ToDoDetail** Screen, with the new wrappers. In this case, we are just updating existing ToDos, so it will be different than the previous step.
 - a) Double-click on the remaining error to open its location.




- b) Delete the call to the **CreateOrUpdateToDo** Action from the flow.



- c) Drag a new Run Server Action statement to the True branch of the If, before the End statement.



- d) In the Select Action dialog, select the **UpdateToDoWrapper** Action.
- e) Set the Inputs of the Action as in the following screenshot. This guarantees that we are getting the information from the input widgets in the Form of the ToDoDetail Screen, submitted by the user.

 UpdateToDoWrapper Run Server Action	
Name	UpdateToDoWrapper
Server Request Timeout	(Module Default Timeout)
Action	PublicDBActions\UpdateToDoWrapper ▼
ToDoid	GetToDoById.List.Current.ToDo.Id ▼
Title	GetToDoById.List.Current.ToDo.Title ▼
IsStarred	GetToDoById.List.Current.ToDo.IsStarred ▼
Notes	GetToDoById.List.Current.ToDo.Notes ▼
DueDate	GetToDoById.List.Current.ToDo.DueDate ▼
CreatedDate	GetToDoById.List.Current.ToDo.CreatedDate ▼
CompletedDate	GetToDoById.List.Current.ToDo.CompletedDate ▼
CategoryId	GetToDoById.List.Current.ToDo.CategoryId ▼
PriorityId	GetToDoById.List.Current.ToDo.PriorityId ▼

- 5) Publish the module and make sure that the application continues to work properly, even with these new changes.

(Challenge) Add an Action to Create / Update Resources

In the previous sections we implemented a pattern that is best practice in OutSystems, by exposing an Entity as Read-only, and then create Server Actions to expose functionality to change the Entity data in consumer modules.

This is also valid for the remaining Entities of the data model, which can lead to something similar to an API, with multiple Actions providing functionalities to the consumer modules, while hiding others. For instance, with the ToDo Entity exposed with write permissions, it would be possible in the consumer module to delete a Record from the Entity. This way, it is not possible.

As an optional challenge for this Lab, let's create one Server Action for Creating or Updating Todos in the Database. Don't forget to refresh the dependencies, even if the Resources are not being used yet in the ToDo module.

NOTE: The following labs will continue with the premise of this part being completed. When not completing this challenge, we can use the Entity Actions directly instead.

End of Lab

In this exercise, we created two reusable Server Actions: one to create new Todos and one to create update Todos. These Actions were created in the ToDo_Core and used the CreateToDo and UpdateToDo Entity Actions to add / update a ToDo record to the database.

We also changed the ToDo Entity to be exposed as Read-only, to make sure that not all the Entity Actions were available to any of the consumer modules.

Then, we refreshed the dependencies in the ToDo module, to use the new Actions and to get the new version of the Entity, and adjusted the logic to use the new Server Actions.

As an optional challenge, we could also use the same pattern for adding / updating Resources to the database, to make sure our application followed OutSystems Best Practices.