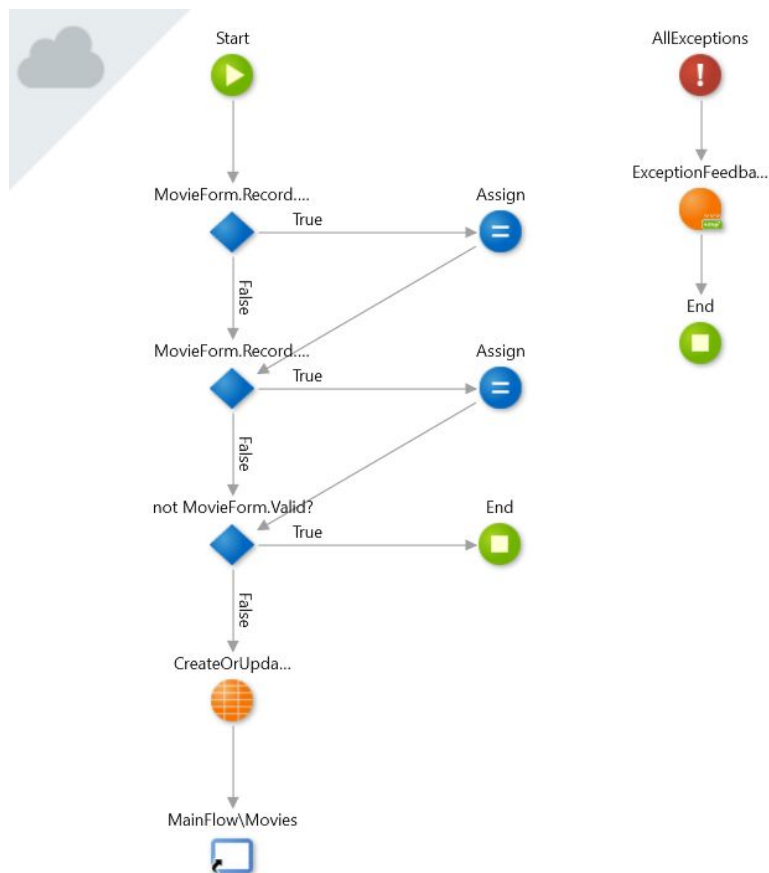


Input Validation Exercise



Introduction

At this point, we already have full functional Screens and Logic in our application. Three of our Screens expect inputs submitted by end-users of the application. Considering the MovieDetail and PersonDetail Screens, there is a Form where users can submit a new movie / person, or edit an existing one.

In such applications, it is important to consider that users make mistakes. They can submit wrong data, miss required fields, etc. So, it is important to define some rules that are validated before the data is submitted to the database.

In this lab, we will start by experimenting the OutSystems built-in validations (mandatory fields and correct data types), by adding examples of wrong data, using the Server and Client & Server validation methods.

Then, we will define our own rules, by implementing custom server-side validations. On the Save Actions of both Screens, we will modify the logic to assess if the data meets some requirements, before the movies and people are added to the database. Also, we will make sure that the application informs the end-user accordingly, if there is some wrong data being submitted, or if there is data missing.

In summary, in this specific exercise lab we will:

- Learn about client-side and server-side validations
- Learn about the default (mandatory and data type) validations
- Code custom (business rule) validations
- Give feedback when validation rules fail

Table of Contents

Introduction	2
Table of Contents	3
Testing the app: Built-in validations	4
Add Validations in the MovieDetail Screen	8
Testing the app: Validations for editing a movie	16
Add Validations in the PersonDetail Screen	17
Testing the app: Validations for editing people	18
End of Lab	19

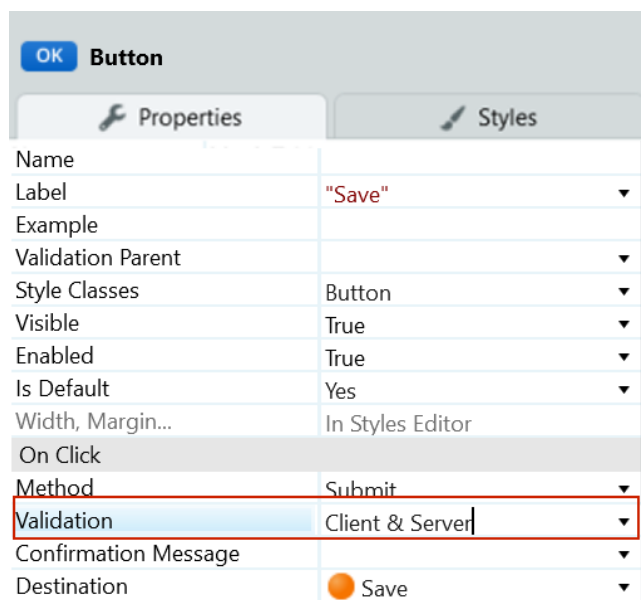
Testing the app: Built-in validations

Before we start developing new features, it is important to become familiar with the default behavior of input validations, both client-side and server-side. This will allow a better understanding of how custom validations work, and how to implement them in subsequent exercise parts.

- 1) Let's create a new Person in the application, with empty mandatory fields and wrong data types, to test the built-in validations. We will do it first with Server and then with the Client & Server option.
 - a) Open the application in the browser, navigate to the **People** Screen and click on the Link **New Person**.
 - b) Create a new Person named *Karren*, without a Surname, and with Date of Birth set as *Walker*. Click on the **Save** Button. Notice that, despite the mandatory field for the Surname is empty, and the Date of Birth has an invalid type, the record is still added to the database.

NOTE: As the logic of the **Save** Action does not have any behavior for the validations implemented, a new Person is added to the database with the **default values** in the Surname and the Date of Birth attributes.

- c) Go back to the Service Studio and open the **PersonDetail** Screen.
- d) Select the **Save** Button and on its **OnClick** properties, change the **Validation** to **Client & Server**.



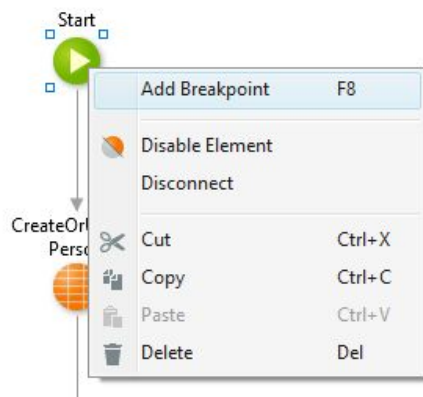
- e) Publish the module and open the OSMDb application in the browser.
- f) Open the **PersonDetail** Screen for *Karren* and try to edit its information, keeping the Name as *Karren*, no Surname, and the Date of Birth as Walker. Note that the built-in validations for the **Mandatory fields** and **Data Types** are now working.

The screenshot shows a form titled "Karren" with the following fields and validation messages:

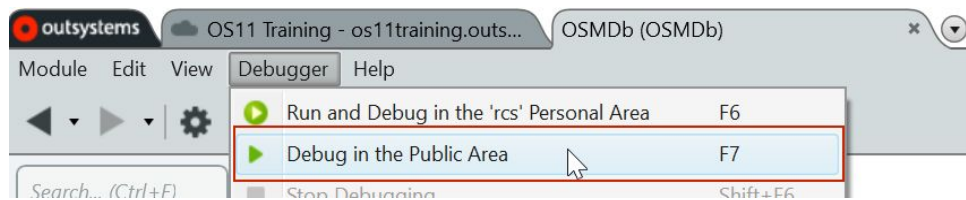
- Name ***: Input field containing "Karren".
- Surname ***: Input field is empty. Below it is a red message: "Required field!".
- Date Of Birth ***: Input field containing "Walker". Below it is a red message: "Date expected!".
- Date Of Death**: Input field containing the placeholder "YYYY-MM-DD".
- At the bottom left are two buttons: "Save" (blue) and "Back to List" (light blue).

NOTE: The client-side validations verify the **Mandatory** inputs and valid **Data Types**, **before** even the request is sent to the server. As opposed to the **Server** Validation option, the **Client & Server** option does not allow the record to be added to the database, verifying automatically if the Form is valid or not.

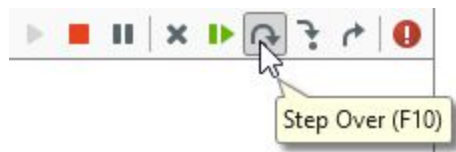
- 2) Let's start the Debugger in Service Studio, and look closely in detail to what is happening during the Server validations. We will do this with the Server option first and then with Client & Server. Make sure you look at the Valid properties of the Inputs and Form.
 - a) Return to Service Studio and change back the **Validation** property of the **Save** Button to *Server*.
 - b) Add a **Breakpoint** at the Start node in the **Save** Screen Action of the PersonDetail Screen.



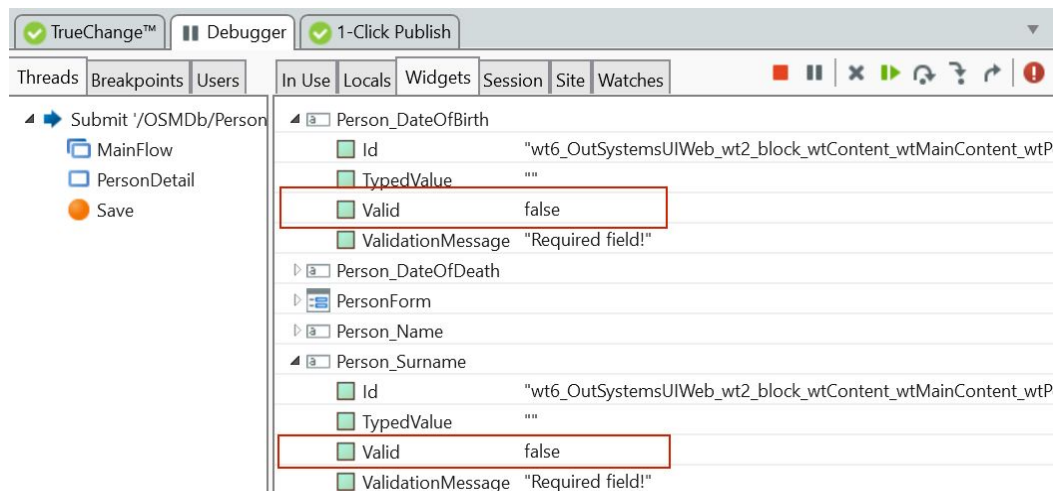
- c) Publish the module and in the **Debugger** menu, select **Debug in the Public Area**.



- d) Open the application in the browser and try to edit the record with Name *Karren*, with the same wrong information.
- e) Back in Service Studio, the execution of the application should have stopped at the Breakpoint we added. You can move on step-by-step by selecting the **Step Over** option in the **Debugger** tab.



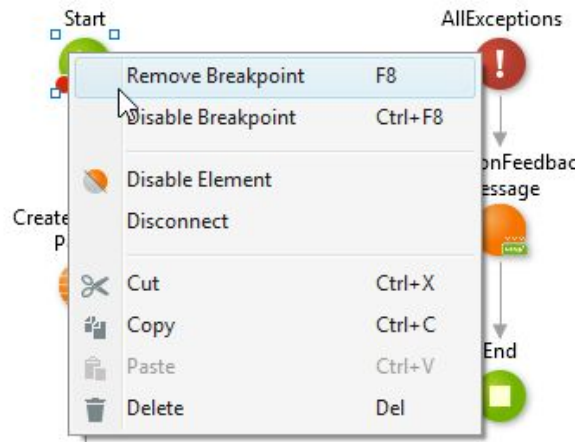
- f) During the Debug process, you can see in the **Widgets** tab, within the **Debugger** main tab, that the Inputs for the **Surname** and **Date of Birth** are set to **not Valid**. The errors do not appear in the page, because we did not implement the logic yet for the server-side validations, but OutSystems is in fact performing the built-in validation automatically. We just need to leverage that in our application.



NOTE: The Input Widget has a **Valid** property that determines if the data in the Input field is valid or not. Its value **is set to False** if it corresponds to a **Mandatory** Input not filled, or if its **Data Type** is not the one expected.

In a Form, if at least one of the Inputs inside it has its **Valid** property set to *False*, then the **Form.Valid** property is automatically set to *False*.

- g) Remove the Breakpoint from the Action, when the Debug is done.

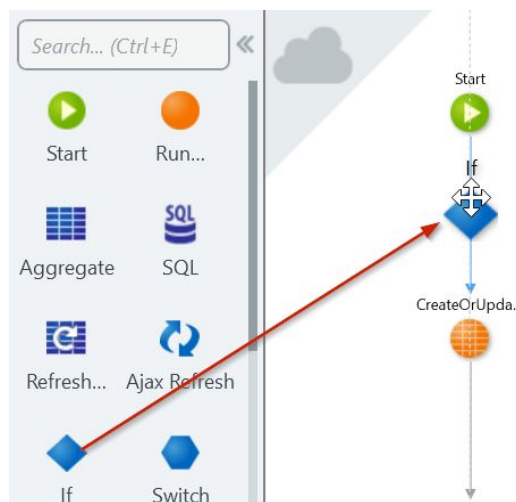


- 3) Change the Validation Method of the Save Button to Client & Server, in the **AddMovieParticipant** Screen, for enabling the **client-side** validations. This will solve the problem of the foreign key violation error, we have seen in the Lab: Data Queries and Widgets II, when we tried to add a Person to a movie with a certain Role, without filling all the mandatory fields. With Client & Server, since the built-in validations are checked on the client-side, we get an error before the request is sent, and we need to guarantee all mandatory fields are filled.

Add Validations in the MovieDetail Screen

Let's now add custom server-side validations to the Save Action in the MovieDetail Screen. Here, we will make sure that a movie cannot have any gross takings amount, if it wasn't released yet, among others. Also, we will make sure that when the built-in validations fail, with Server validations, we also get the errors appearing in the Form (just like with the Client & Server in the previous Section). Finally, we will use some Feedback Messages to give the end-user some feedback on the success of the Save Action.

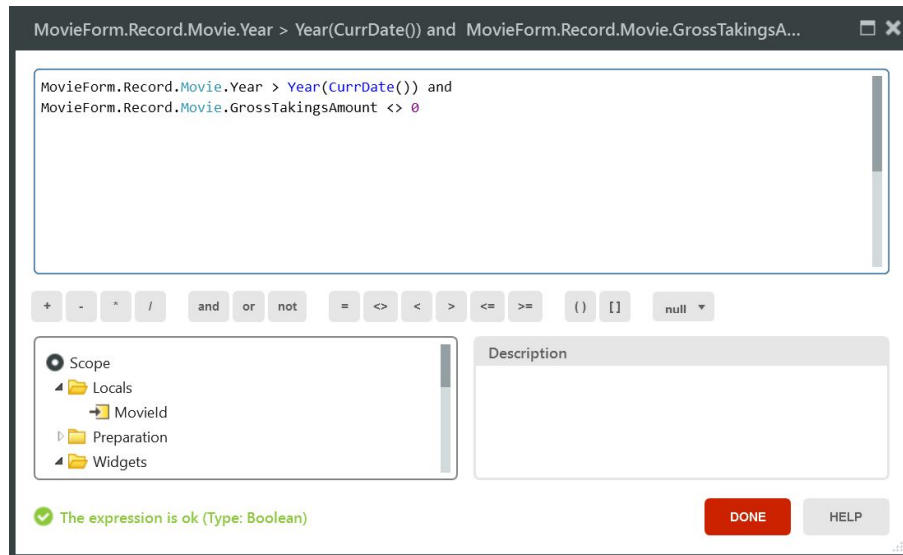
- 1) Add a (server-side) validation to the **Save** Action of the MovieDetail Screen that verifies that the Gross Takings of a yet to come movie must be zero.
 - a) Open the **MovieDetail's Save** Screen Action.
 - b) Drag and drop an **If** statement between the Start and the **CreateOrUpdateMovie** statement.



- c) Notice that the outward connector from the If to the **CreateOrUpdateMovie** has the label **False**. This means that the condition we specify in the If needs to check if the Input has errors. If it does not have errors, then the flow of the Action will continue normally.

- d) Set the **Condition** property for this If to

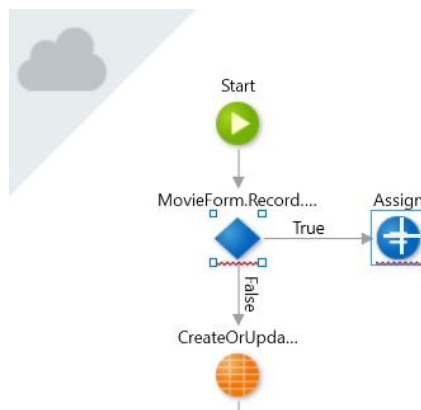
*MovieForm.Record.Movie.Year > Year(CurrDate()) and
MovieForm.Record.Movie.GrossTakingsAmount <> 0*



This condition checks if the Gross Takings Amount is different than zero, when the movie Year is bigger than the Current Date (if the movie is still to be released). If this is True, it means it will fail our custom validation. If this is False, it means it passed the validation and will proceed to the **CreateOrUpdateMovie** Action.

NOTE: Notice the use of the built-in functions **Year()** and **CurrDate()** in the condition. These, and many others, can be found in the Scope tree (of the Expression Editor), under the **Built-In Functions** folder.

- e) Drag and drop an **Assign** statement to the right of the If. Create a second connector from the If, in this case to the Assign.



- f) Notice that the new outward connector from the If to the Assign has the label **True**. Considering how the condition was made, the code will follow this path if there *are* problems with the Input values.

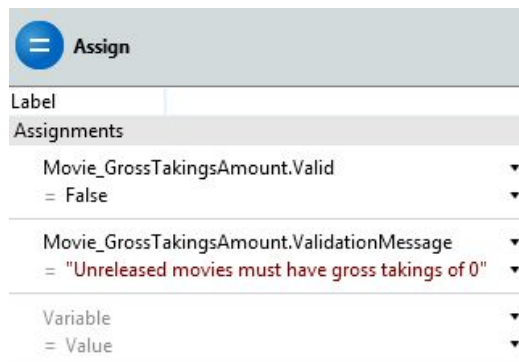
NOTE: Once this second connector is created, the TrueChange error related to the If statement disappears. This is because the If, unlike most other statements, requires *two* connectors that indicate the alternate paths to take depending on the Condition being True or False.

If it ever feels more natural to write the inverse of the If condition (e.g. write the condition as *CheckOk* instead of *not CheckOk*), we can swap the True/False connectors on the If. The quickest way to do so is to right-click the If statement and select **Swap Connectors** from the pop up menu.

- g) Add the following two assignments to the Assign statement

Movie_GrossTakingsAmount.Valid = False

Movie_GrossTakingsAmount.ValidationMessage = "Unreleased movies must have gross takings of 0"

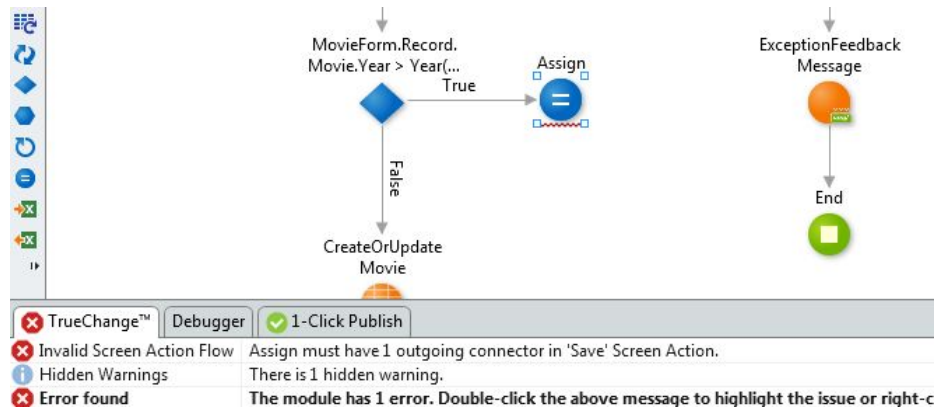


Assign	
Label	
Assignments	
Movie_GrossTakingsAmount.Valid	= False
Movie_GrossTakingsAmount.ValidationMessage	= "Unreleased movies must have gross takings of 0"
Variable	= Value

This Assign sets that the Gross Takings Amount input is not valid, and the appropriate message to appear on the page (when the input is not valid). This is done here, since this will be the flow that will be followed when the If Condition is evaluated to True, meaning that it failed our custom validation.

NOTE: The invalidation of inputs normally follows the pattern above: set its **Valid** property to false and its **ValidationMessage** to the explanation of the problem detected. It's important to emphasize that, while detecting a validation problem normally involves looking at the Form's Record value, flagging a validation error is *always* done using an individual input's properties.

- h) Notice that the **TrueChange** tab indicates that an outward connector is still needed for the Assign. Since you will be adding further validations next, ignore the error for now.



- 2) Add a new custom validation to verify that the Gross Takings can never be a negative number. This one should be added next to the previous one.

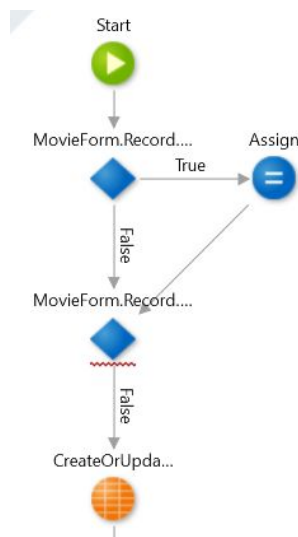
- a) Drag and drop an **If** statement between the If created above and the **CreateOrUpdateMovie** statement.

- b) Set the **Condition** property for this If to

MovieForm.Record.Movie.GrossTakingsAmount < 0

This verifies if the Gross Takings Amount is smaller than zero.

- c) Create the outbound connector from the Assign statement, of the last step, to the recently created If statement. The **TrueChange** validation error for the Assign should have disappeared.

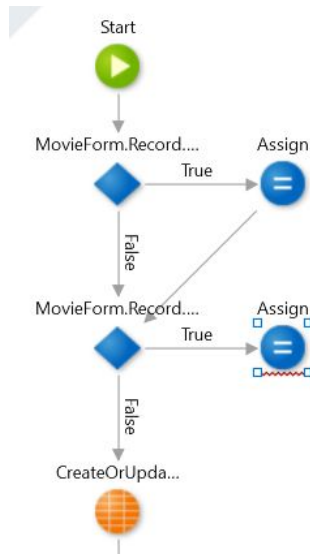


- d) Drag and drop a new **Assign** statement to the right of the If.
- e) Create the second connector from the If to the new Assign statement.
- f) Add the following two assignments to the **Assign** statement

`Movie_GrossTakingsAmount.Valid = False`
`Movie_GrossTakingsAmount.ValidationMessage = "Gross takings amount cannot be negative"`

Assign	
Label	
Assignments	
Movie_GrossTakingsAmount.Valid	= False
Movie_GrossTakingsAmount.ValidationMessage	= "Gross takings amount cannot be negative"
Variable	= Value

- g) As before, ignore the missing outgoing connector error for now.



- 3) Add the final validation that checks if the Form is **Valid**, meaning that every Form Input is **Valid**. If it is, proceed to the **CreateOrUpdateMovie** statement. If not, end the **Save** Action, without adding the record.
 - a) Drag and drop an **If** statement between the previously created If, and the **CreateOrUpdateMovie** statement.

- b) Set the **Condition** property for this If to
not MovieForm.Valid

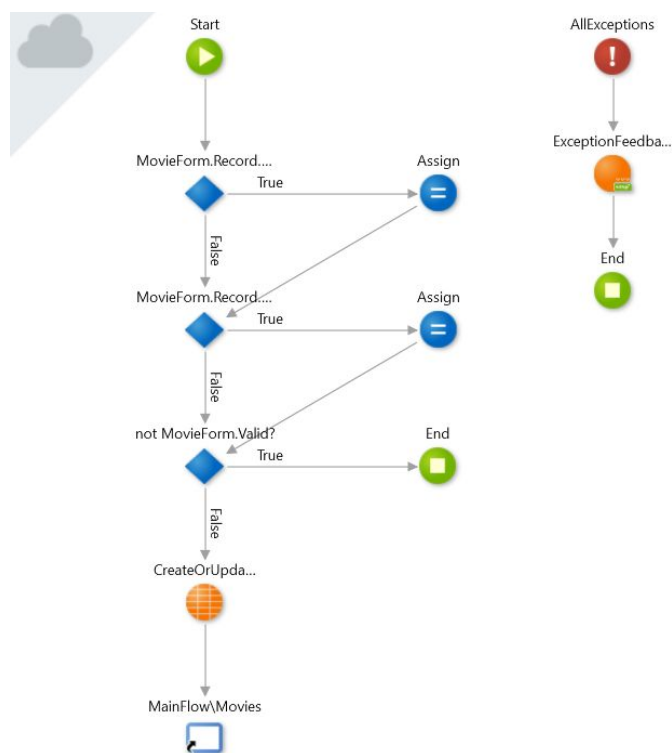
This verifies if the Form is Valid. This property cannot be changed with an Assign. It changes automatically if at least one Input of the Form is not Valid.

- c) Create the connector from the previous Assign to the new If.
- d) Drag and drop a new **End** statement to the right of the If and create the second connector from the If to it.

NOTE: Unlike the original termination statement, that redirected to the **Movies** Screen, after successfully saving the movie, in the case of a failed validation, the user should remain on the same Screen.

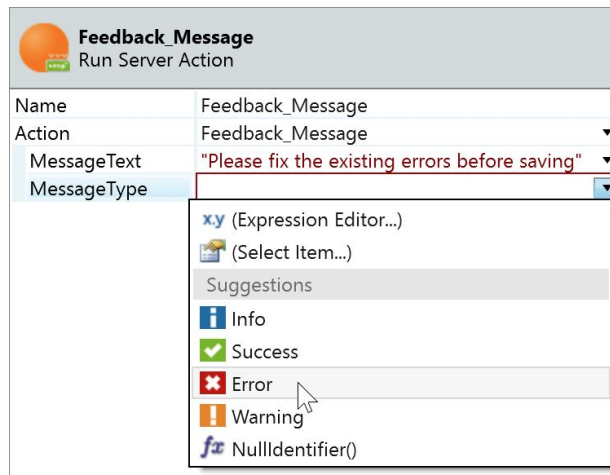
There is no limit to the number of termination statements (**End** or **Destination**) that an Action may have.

- e) The module should have no errors and the **Save** Action should look like this



- 4) Give feedback to the user, using the **Feedback_Message** Action, on the Success or Error outcome of the **Save** Action.

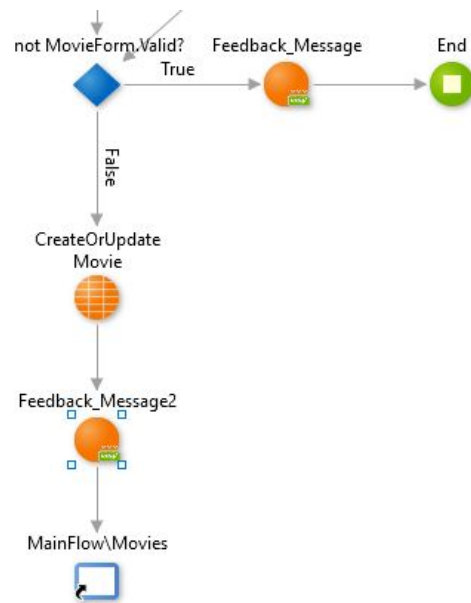
- a) Drag and drop a **Run Server Action** statement between the Movie Form Condition and the End statement.
- b) On the **Select Action** dialog, select *Feedback_Message* and click **OK**.
- c) Set the **Feedback_Message** input parameters as shown below
MessageText: "Please fix the existing errors before saving"
MessageType: Entities.MessageType.Error



Feedback_Message Run Server Action	
Name	Feedback_Message
Action	Feedback_Message
MessageText	"Please fix the existing errors before saving"
MessageType	<div> <div>xy (Expression Editor...)</div> <div>(Select Item...)</div> <div>Suggestions</div> <div>Info</div> <div>Success</div> <div>Error</div> <div>Warning</div> <div>NullIdentifier()</div> </div>

- d) Drag and drop a **Run Server Action** between the **CreateOrUpdateMovie** and the **Destination** statements. Set it to be a *Feedback_Message*. This will create a **Feedback_Message2** statement in the flow.
- e) Set the **Feedback_Message2** properties as shown below
MessageText: "Movie " + MovieForm.Record.Movie.Title + " saved successfully !"
MessageType: Entities.MessageType.Success

f) After these changes, the final part of the Action flow should look like this



g) Publish the module using the **1-Click Publish** button.

Testing the app: Validations for editing a movie

In this section of the Lab, we will test the input validations that we added in the previous Lab. Don't forget to test the custom-made server-side validations, by introducing invalid information while editing existing movies:

- 1) Open the application in the browser and in the **Movies** Screen, select the *Avatar 2* for editing.
- 2) Set the **Gross Takings** to *500* and click on **Save**. As the movie is not out yet, you should see the **Validation Message** defined in the previous part of the Lab.
- 3) Set the **Gross Takings** to *-50* and click **Save**. You should see the error regarding the negative Gross Takings Amount.
- 4) Set the **Gross Takings** to "hello" and click **Save**. You should have a different error, in the **Gross Takings** Input box, due to the incorrect **Data Type** (built-in validation).

Add Validations in the PersonDetail Screen

Now that we have added and tested custom validations for movies, we will also do the same for the PersonDetail Screen. We will add validations to verify if the Birth Date is before the Current Date, and if the Person's Date of Birth is later than the Person's Date of Death.

- 1) Add a validation to the **Save** Action in the **PersonDetail** Screen that verifies that the date of birth needs to be before the current date.

NOTE: This exercise section will not be broken down further. You can refer to what you did earlier in the exercise for pointers. Be aware of the built-in functions *NullDate()* and *CurrDate()*, that may be useful for implementing these validations.

- 2) Add a second validation to the same **Save** Action that verifies that the date of death, if specified, needs to be later than the date of birth.
- 3) Add the final validation to verify if everything in the Form is correct, including the detection of unfilled mandatory and wrong typed inputs.
- 4) Give feedback to the user using the **Feedback_Message** Action, on the **Success** or **Error** cases of the **Save** Action.

Testing the app: Validations for editing people

To finish the lab, we will test the custom validations for editing people. To test them, we need to edit the information of a Person in the **PersonDetail** Screen, with invalid information, like a Birth Date in the future, or a person dying before being born. Make sure that the corresponding validation messages appear.

- 1) Publish the application and open the detail page of *Harrison Ford*.
- 2) Set the **Date of Birth** to tomorrow and click **Save**. You should see the **Validation Message** appearing, indicating that the date of birth cannot be in the future.
- 3) Set the **Date of Death** to today, without changing the invalid date of birth, and click **Save**. You should now see both Input boxes invalid, as both validations fail.
- 4) Set the **Date of Birth** back to '1942-07-13' and the **Date of Death** to '1942-07-12' and click **Save**. The validation regarding the date of death being after the date of birth should fail and the corresponding validation messages appear.

End of Lab

In this Lab, we enabled the detection of wrong information specified on the Inputs of both detail Screens (MovieDetail and PersonDetail).

We have seen that the built-in validations of **Mandatory** and **Data Types** apply to both client-side and server-side validations. We also used the Debugger to inspect the values of the Widget Variables, to understand when they are Valid and when they are invalid.

We implemented custom server-side validations, for movies and for people, where we created some rules that the movies and people should follow, before being added to the database.

Feedback of validation errors can be done both in-place, next to the offending Input, by setting the Input to not **Valid** and define a **Validation Message**. Additionally, we also used **Feedback_Message** Actions, so that the user can see a pop-up with a message sliding down the Screen, indicating the success or failure of the operation.