CSS exercise

Table of Contents

knowledge@outsystems.com

www.outsystems.com

Outline

In this exercise we will focus on:

- Building a simple application that makes use of built-in widgets and OutSystems UI patterns

Upon completion, you will have built a screen with multiple widgets and patterns working together, and you will have experienced first hand how OutSystems UI can accelerate your development with its pre-built patterns.

Resources

No extra resources are required.

Scenario

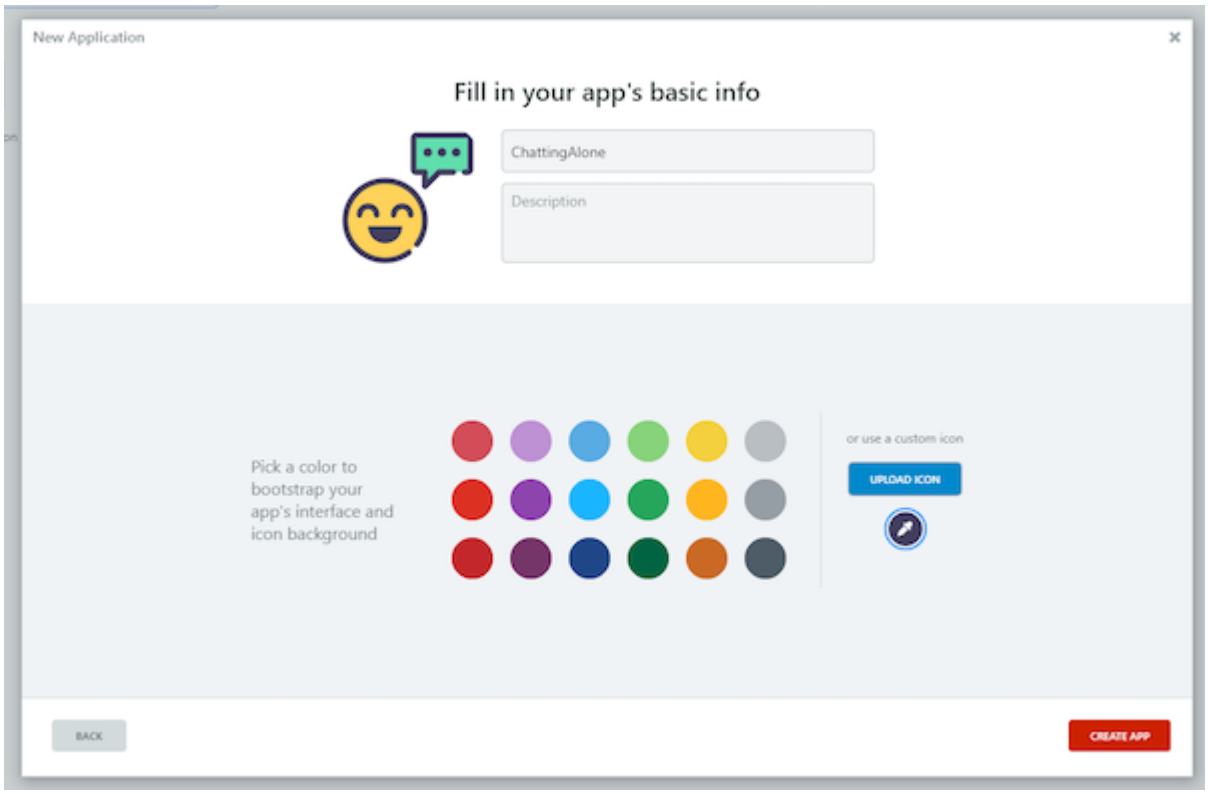
In this exercise, we will start by creating a new Reactive Web application called ***ChattingAlone*** with a single Reactive Web module called ***ChattingAlone***. We will then add a screen that will allow the user to write messages that will show up on the Message area. This screen can be a starting point to a messaging app and showcases several useful widgets and patterns.

How-To

In this section, we'll provide thorough step-by-step instructions for you to complete the exercise.

Getting Started

To start this exercise, we need first to create from scratch a new Reactive Web App and name it ***ChattingAlone***. We will use the above icon from Stockio.com. Next, create a new Reactive Web App module and name it ***ChattingAlone*** and we are done.



Create a new Empty Screen called ***Chat***, and make sure it has the anonymous role ticked.

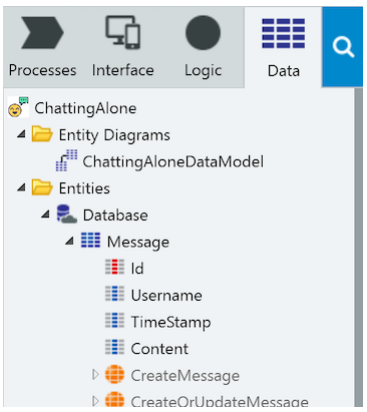
1. Creating database

First, we need to make sure the database can hold the required information.

1) Switch to the Data tab and create a new Entity called ***Message***.

2) Make sure it has the following attributes:

   - **Id**
   - **Username** of type *Text* and up to 250 characters in length
   - **TimeStamp** of type *DateTime*
   - **Content** of type *Text* with up to 1000 characters in length

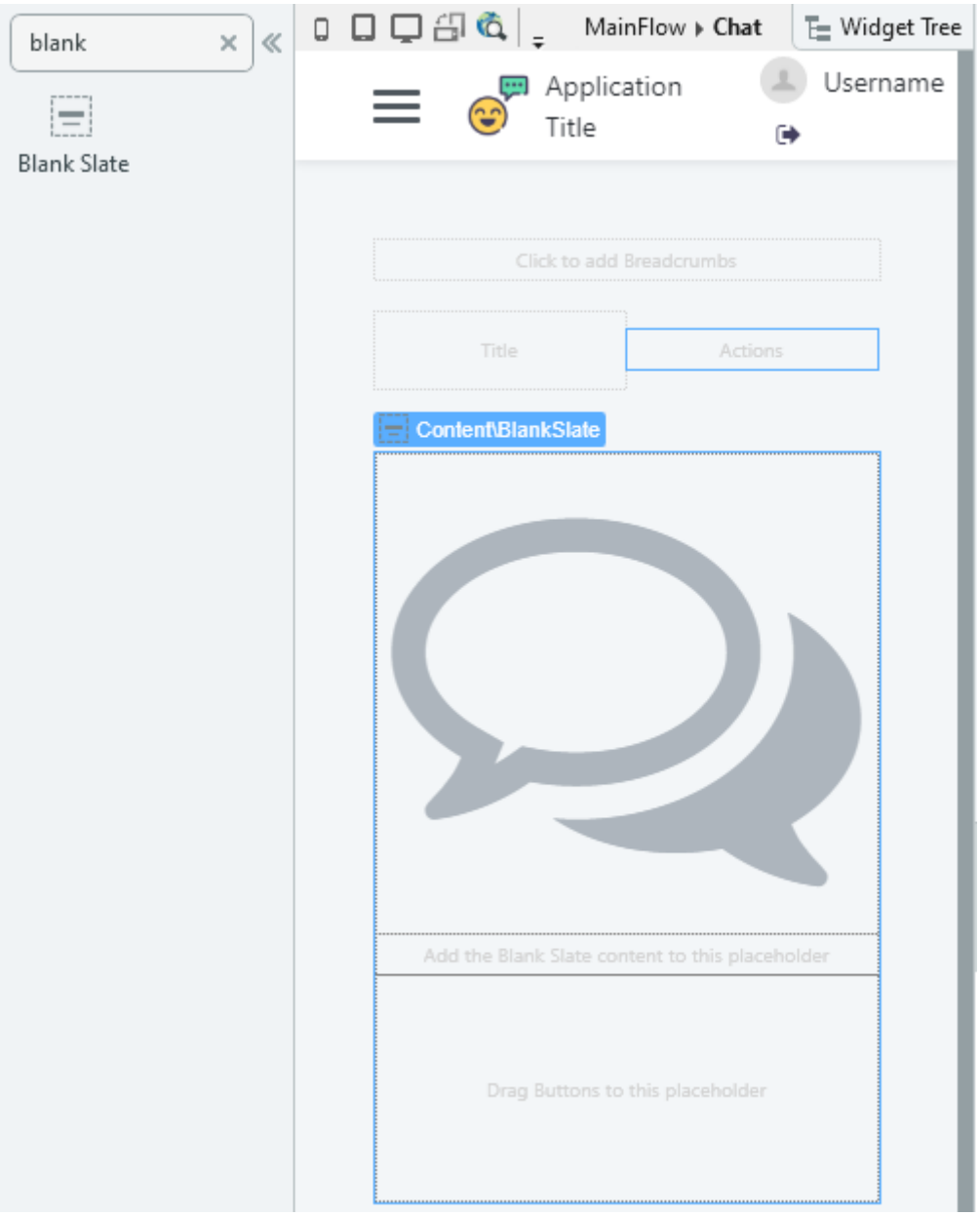


2. Using a Blank Slate

Instead of showing an empty area where messages would displayed if there were any, we will use the Blank Slate pattern to provide a visual queue when there is no data to display.

1) Open the *Chat* screen and add a new screen Aggregate that fetches all Message records from the Database.

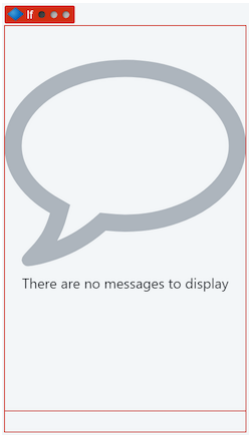
2) Now drag a **Blank Slate** pattern from the Toolbox unto the Main Content placeholder:



3) Change the icon shown to be `comment-o`.

4) On the *Content* placeholder write `"There are no messages to show"`.

5) Enclose the pattern in an **If** widget:



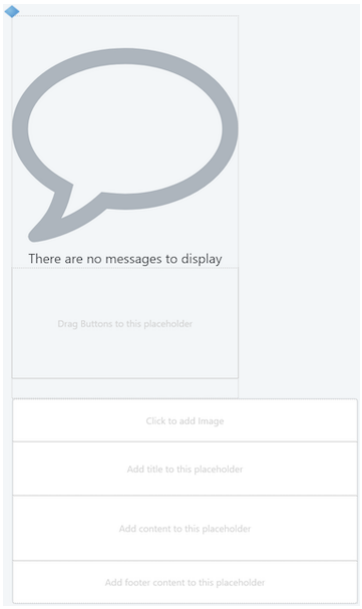
6) The blank slate should only be displayed when the Aggregate returns no records, so let us set the **If** widget's condition to `GetMessages.List.Empty`.

7) Publish your module and check that the blank slate is being displayed.

3. Building new message area

The next step is to create an area where the user can input new messages.

1) Drag and drop a **CardSectioned** pattern after the **If** widget



2) Plate the text `"New message"` on the card's *Title* placeholder

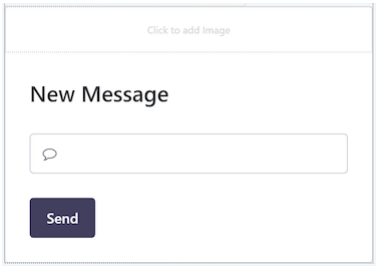
3) Drag and drop an **InputWithIcon** pattern into the card's *Content* placeholder

4) Change the icon of the **InputWithIcon** pattern to `comment-o`

5) Set the **Input**'s Variable property to a *New Local Variable*, and change the name of that variable to `ChatInputText`

6) Place a **Button** widget on the *Footer* placeholder of the card and change its label to `Send`.

7) Define the Button's *OnClick* event handler to be a *New Client Action*



Publish your changes and check that you can now see the message sending section on your screen, right below the blank slate.

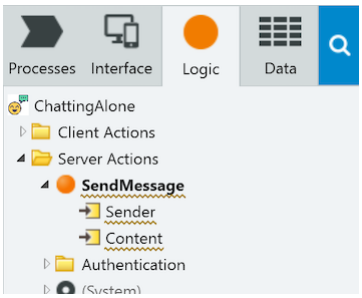
Congratulations! You now have the user interface for your users to start sending messages, now let's add the behavior.

4. Add Send Message behavior

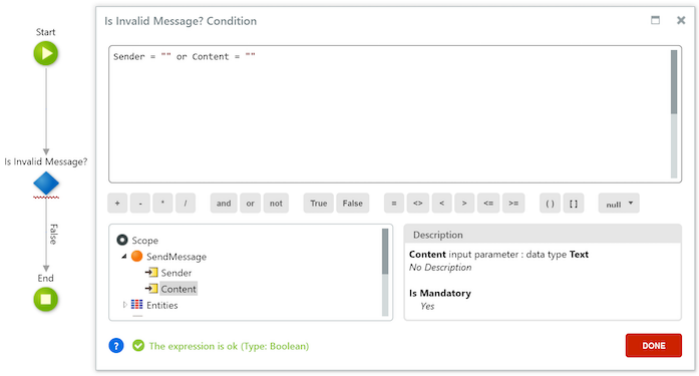
For our behavior, we will need a new Server Action to create a new entity record in the Database, and a Screen Action to user interaction.

1) Let's start by creating a new Server Action named ***SendMessage***

2) Add two *Text* Input Parameters to the action named ***Sender*** and ***Content***:



3) Using an **If** tool, check that neither Input Parameter is an empty Text:



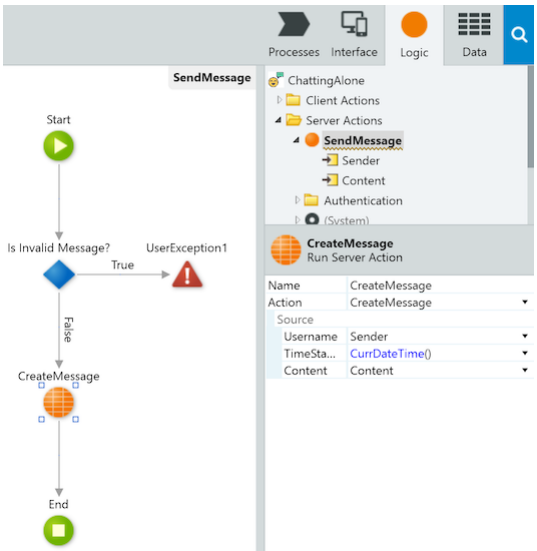
4) Connect the *True* branch of the **If** to a **Raise Exception** node and raise a *New User Exception*.

5) Set that exception's Message property to `"Cannot send empty message."`

6) The *False* branch should call the **CreateMessage** Entity Action before the **End** node.

7) Use the *Expand* button left of the **CreateMessage**'s Input Parameter to define the record to be saved :

   - *Username*: `Sender`
   - *TimeStamp*: `CurrDateTime()`
   - *Content*: `Content`



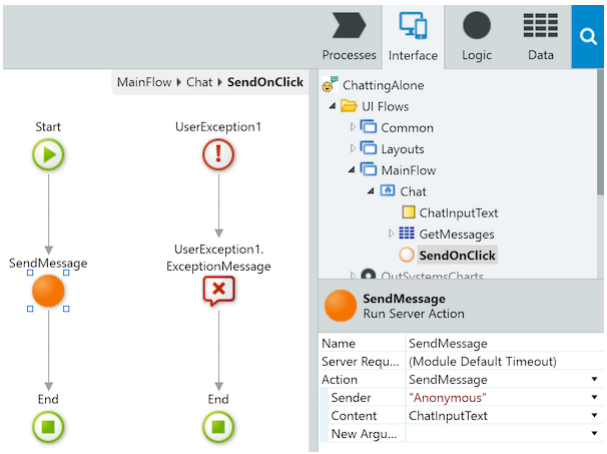
Now, lets use the newly created server action on the screen.

1) Go to the *Chat* screen and open the Screen Action created earlier.

2) Drag a **Run Server Action** to the action flow and call the *SendMessage* action.

3) Set the *Sender* parameter to `"Anonymous"` and the *Content* parameter to the be `ChatInputText`, the screen's local variable.

4) Add a new **Exception Handler** for the *UserException1* exception, we will want to display the exception message to the user via the **Message** tool:



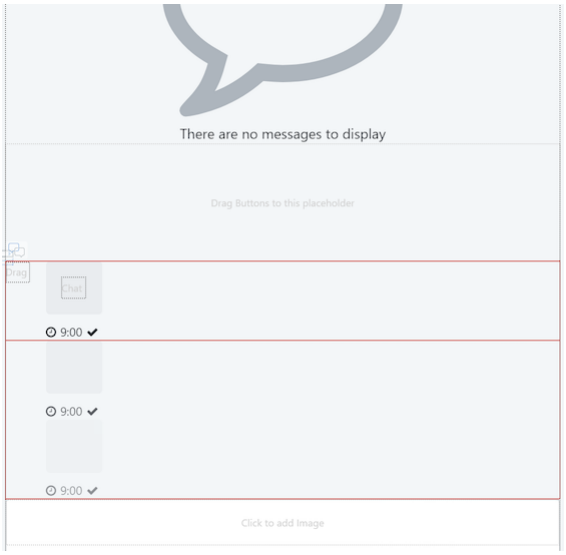
Publish your application and check if there is an error message displayed when nothing is typed.

5. Building a list of messages

Let us return to the Interface tab, our screen needs to display the messages sent. We will use a **List** widget, with a **Chat Message** Pattern inside, for to display individual messages.

1) Drag and drop a **List** widget to the *False* branch of the **If** widget.

2) Drag a **Chat Message** pattern inside the list:

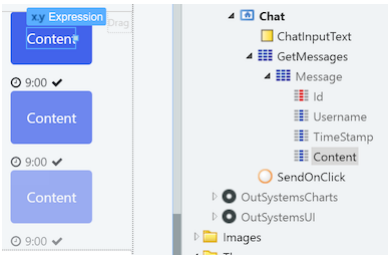


3) Set the **List**'s *Source* property to `GetMessages.List`.

4) Set the properties of the **ChatMessage** pattern to:

- DisplayOnRight: `True`
- Time: `GetMessages.List.Current.Message.TimeStamp`
- MessageStatus: `Entities.MessageStatus.Read`

5) Drag the aggregate's *Content* attribute to the *Content* placeholder of the **ChatMessage** pattern:



Publish your module. When you open the *Chat* screen on a browser it should now show the messages you previously sent.

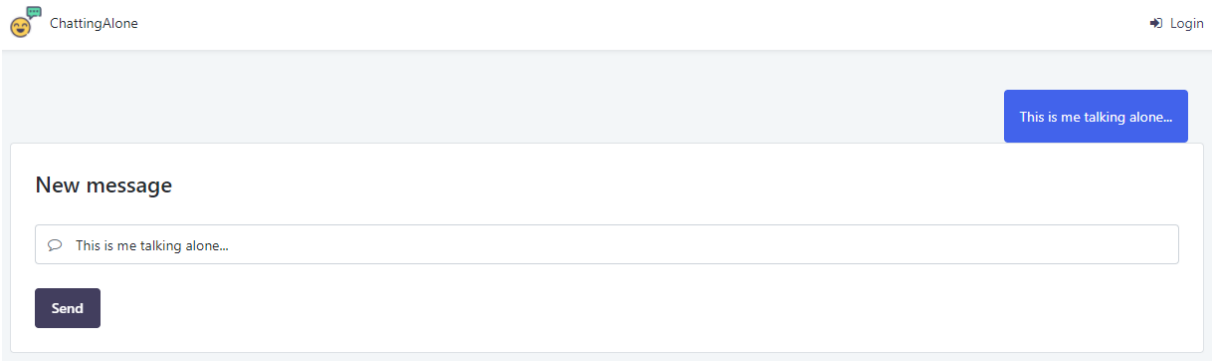
Refreshing the screen

If the user tries to send a new message, they will not see any changes on the message list. Let us address that

1) Open the *SendOnClick* Screen Action and drag a **Refresh Data** tool before the **End** node.

2) Make sure to select the GetMessages aggregate as its *Data Source*.

Publish your module again. At this point you should be able to see your messages showing up as you send them:



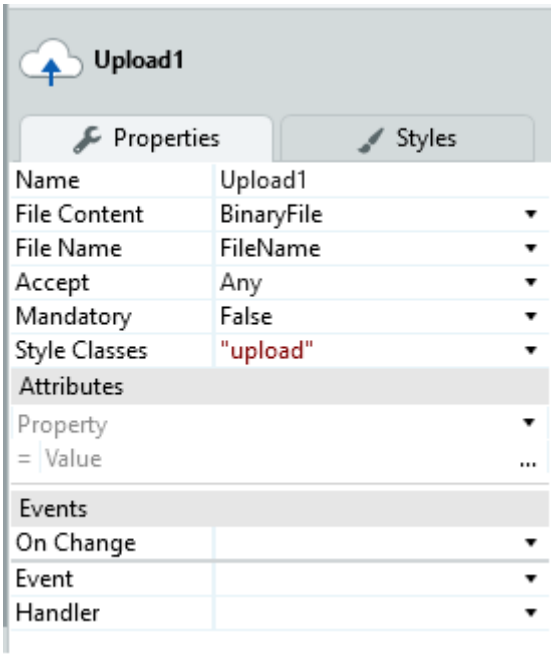
Sending images on chat (user interface)

Sometimes we use real time chat apps to send images to other users.

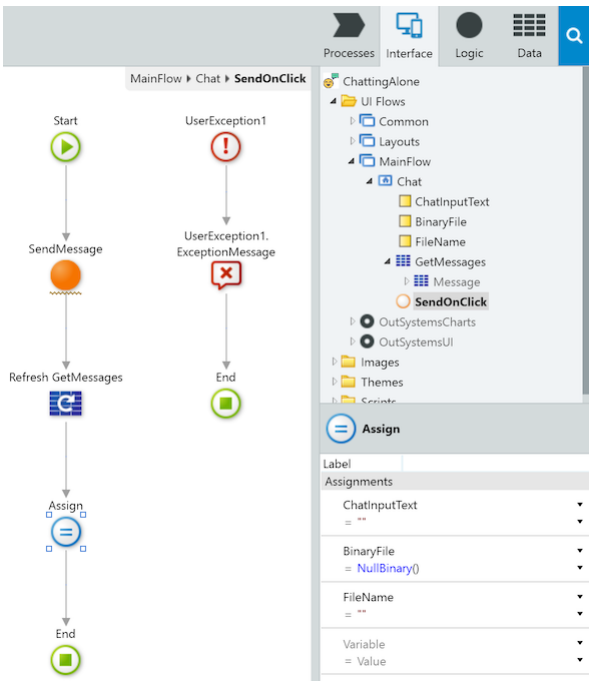
1) Drag the **Upload** widget and drop it in the *Content* placeholder of the **CardSectioned** right after the **InputWithIcon** pattern.

2) Create two local variables on the **Chat** screen, a *Binary Data* called ***BinaryFile*** and a *Text* called ***FileName***.

3) Set the *File Content* property of the **Upload** widget to `BinaryFile`and the *File Name* property to `FileName`



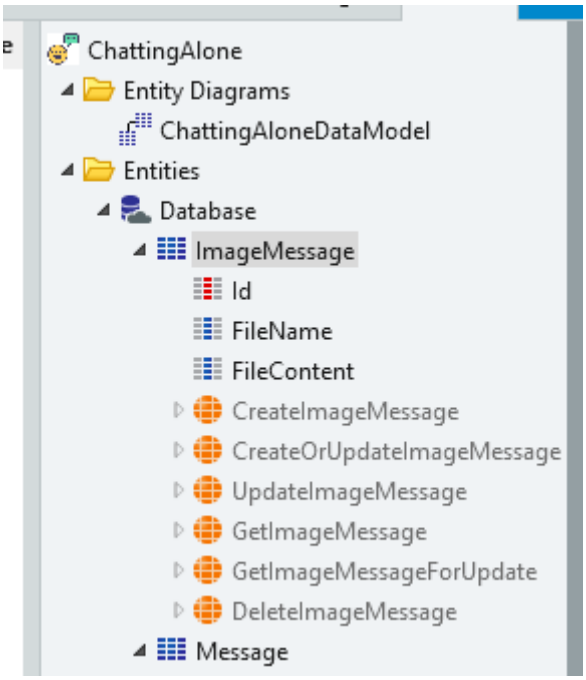
4) On the client action, before the action ends add an **Assign** node to reset all variables to their default values.



Sending images on chat (business logic)

1) Create a new database Entity called ***ImageMessage***

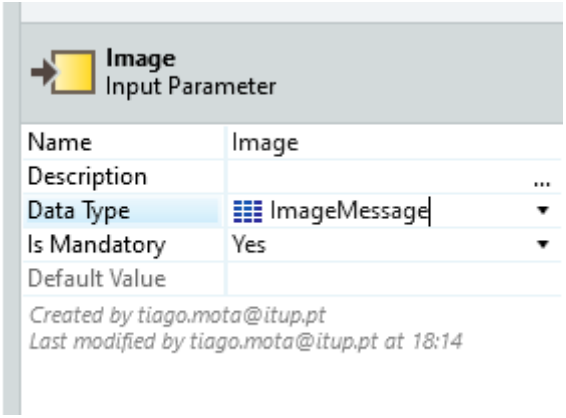
2) Add two new attributes to the entity, ***FileContent*** and ***FileName***. Confirm they were automatically assigned the correct data types, *BinaryData* and *Text* respectively.



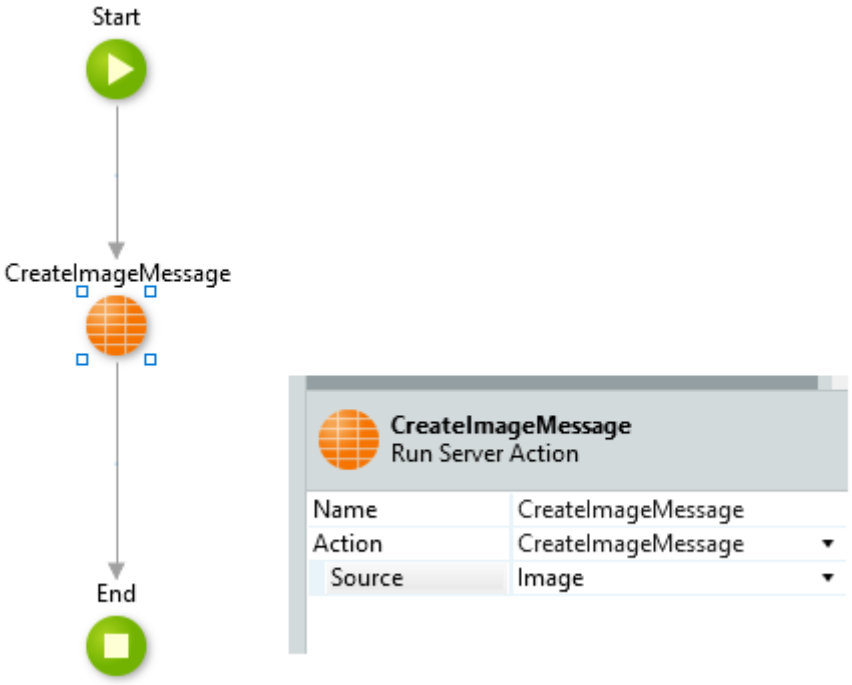
3) Change the **Id** attritute's data type to **Message Identifier** to establish a one to one relation.

4) Create a new ***SendImage*** server action

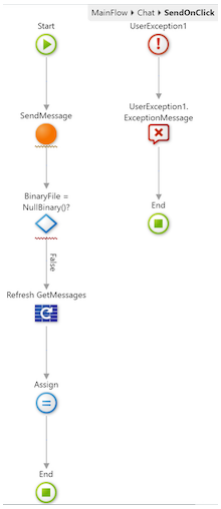
5) Add an input parameter called ***Image*** of **ImageMessage** data type.



6) Drag a **Run Server Action** tool to the action flow to call the ***CreateImageMessage*** entity action and pass the *Image* as input parameter.

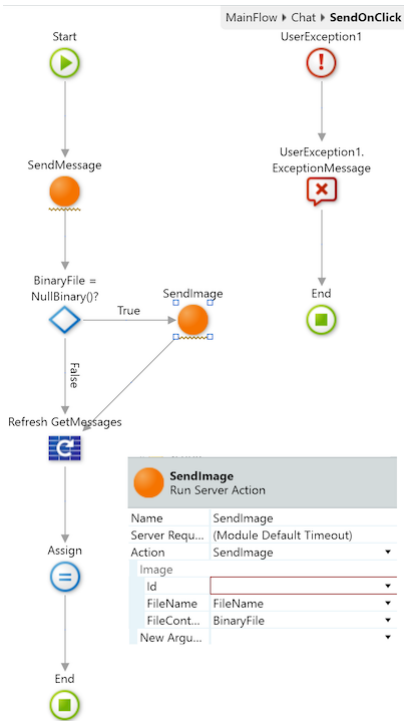


7) On the screen action, add an **If** after the *SendMessage* action call, to check if there is an image to be sent:



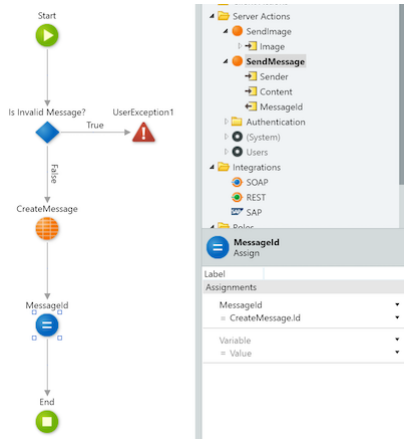
8) Drag a *Run Server Action* on the *True* branch to call the **SendImage** action, use the *Expand* button next to its **Image** Input Parameter to pass it the **FileName**

and **BinaryFile** variables as part of its input parameter. Note however that the **Id** is still missing.

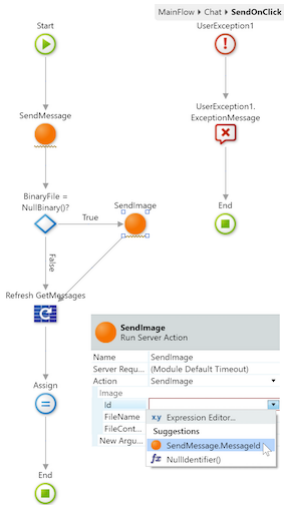


9) Create a **MessageId** Output Parameter for the **SendMessage** action, we will need the id of the new message so we can assign it to the image.

10) Open the **SendMessage** action flow and add an **Assign** tool where we set the *MessageId* output parameter to the output of the **CreateMessage** entity action.



11) Go back to the client action and you should now be able to set the missing *Id* attribute of the **SendImage**'s input parameter to `SendMessage.MessageId`.



Showing messages with images

Now that we can send messages that can have an image attached, we also need to display them on the message list.

1) Open the *Chat* screen's Aggregate and drag the **ImageMessage** entity to it, this should create a *With or Without* join between the **Message** and **MessageImage** entities. Change the name of the Aggregate back to `GetMessages`

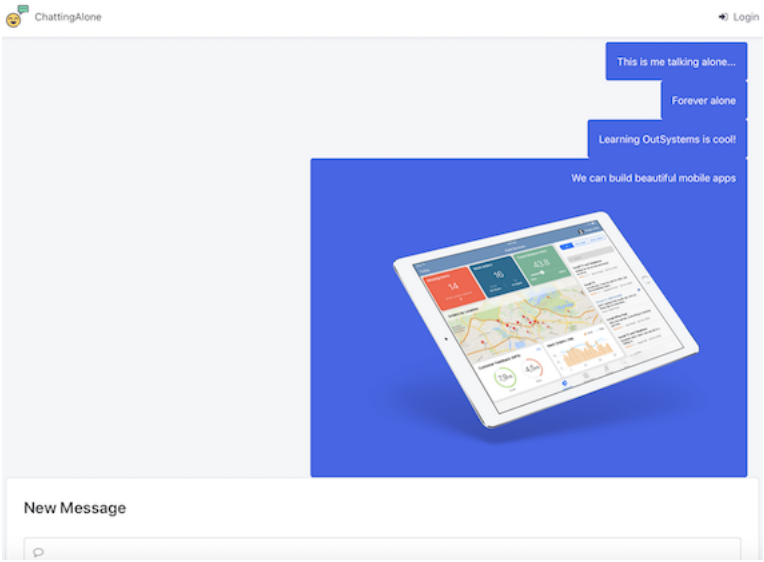
2) Drag an **If** widget to the Content placeholder of the **ChatMessage** pattern, after the Expression displaying the message text.

3) Set the **If**'s condition to `GetMessages.List.Current.ImageMessage.Id <> NullIdentifier()`, so that if there's an image on the message it is displayed.

4) Place an **Image** widget on the *True* branch of the **If**, change its *Type* to `Binary Data`, and its Image Content to `GetMessages.List.Current.ImageMessage.FileContent`.

Once you're all set, publish your changes and open the page in the browser.

After sending a few messages, including with images,we would see something similar to this



Thank you!