outsystems

JavaScript Components integrating with existing libr

Table of Contents

knowledge@outsystems.com

www.outsystems.com

Outline

In this exercise we will be using a simple JavaScript Countdown component found at https://github.com/Dazix/SimpleCountDown, and make it available for any OutSystems developer to use without having to understand JavaScript.

Resources

This library relies on JavaScript to manipulate the DOM of an existing HTML container element, and then enabling/disabling the countdown on demand via method calls. In order to integrate any JavaScript library with OutSystems, we first need to import the necessary resources (JavaScript files, CSS styles, etc), then typically create reusable blocks that integrate those resources to provide their functionality and finally use those blocks in applications.

NOTE: All necessary files for this exercise can be found in the exercise's resources.

Scenario

In this exercise, the Countdown block is going to be used by developers to display to end-users a flip-number countdown. By the end of it, you will need to enhance it so developers have a better experience while using it in Service Studio.

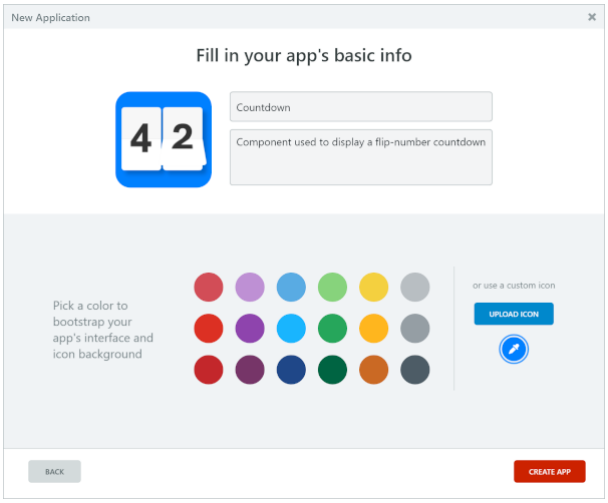
How-To

In this section, we will provide step-by-step instructions to guide you while you build the pattern.

Getting Started

First let us create the application that will provide the **Countdown** pattern for other applications to consume:

1) Create a new Reactive Web App called **Countdown** and choose as the application's custom icon the **countdown-icon.png** file you can find in the exercise's resources.



2) Add a new *Blank* module with the same name as your application

Load the required scripts

Next we will create the Countdown block that developers will use in their applications.

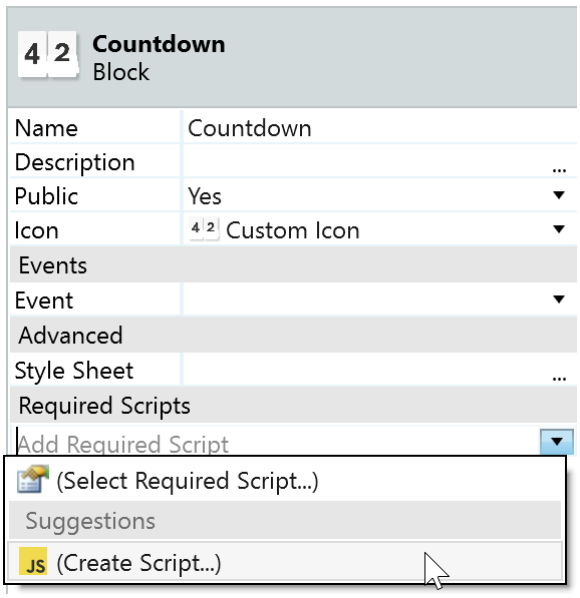
1) Add a new UI FLow called `Countdown`

2) Create a new block called `Countdown`as well

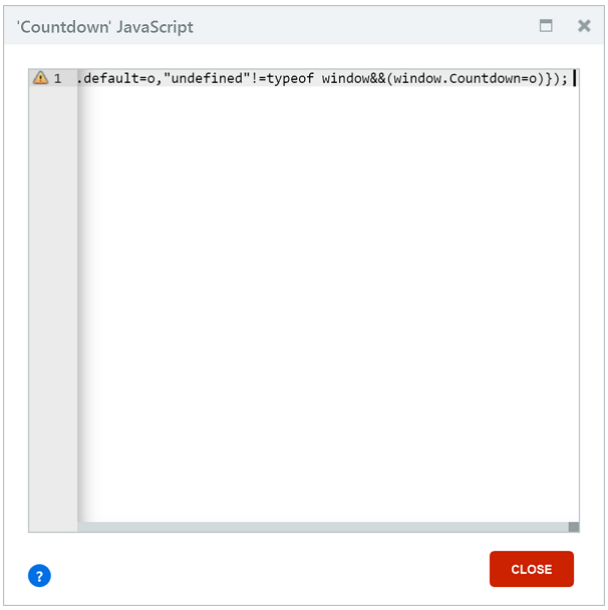
3) Change the block's Icon property to the **component-icon.png** file you can find in the exercise's resources

4) Set its Public property to `Yes`

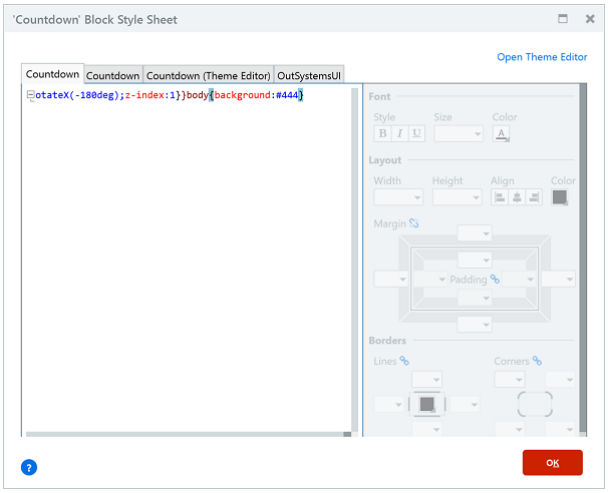
5) Add a new *Required Script to* the block



- Make sure the new script is named `Countdown` and copy the contents of the **Countdown.min.js** file to the script's *JavaScript* property



6) Open the CSS editor and copy the contents of the **Countdown.min.css** file to the block's CSS tab



Integrate JavaScript functionality in OutSystems

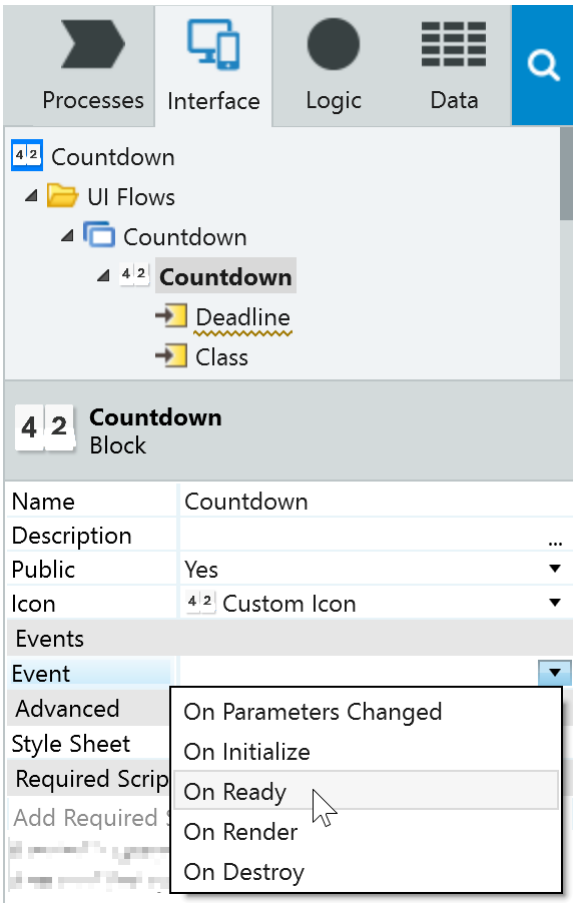
Now that we have imported the library resources, we will need to make use of its functionality to create a Countdown clock, start it and also stop it when needed.

1) Enhance the **Countdown** block with the required elements and input parameters necessary to configure and use the library, namely the deadline for the countdown clock and the style class name to be used for the container that will display it:

   - Add a new *DateTime* input parameter named `Deadline`
   - Add a second input parameter, of type *Text* and named `Class`
   - Drag a new *Container* to the block's canvas, and assign to the container's *Style Classes* property the value of the **Class** input parameter

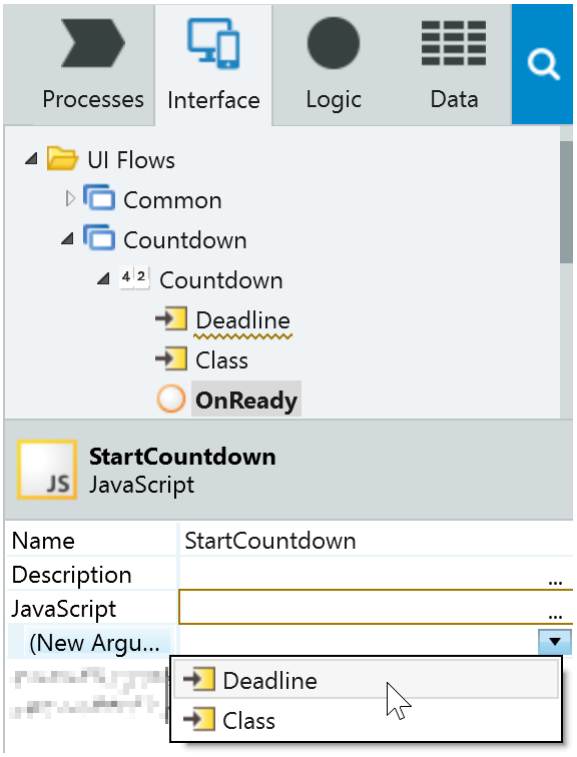
3) In order to initialize the library, we need to create a new instance of a `Countdown` JavaScript object and call its `start()` method. This can only happen after the DOM for the block has been loaded and is ready to be rendered.

   a) Select the **Countdown** block and add an event handler for the *OnReady* event



   b) Drag the *JavaScript* tool to the action flow of the newly created **OnReady** screen action and name it `StartCountdown`

c) Select the **StartCountdown** node and add both input parameters of the block as new pass-through arguments



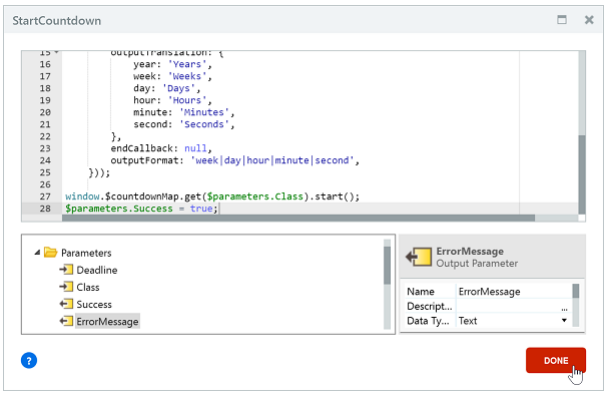
d) Open the **StartCountdown** node and add two output parameters:

- A *Boolean* named `Success`
- A *Text* named `ErrorMessage`

e) Paste the following JavaScript code into the code editor area above the Parameters section of the **StartCountdown** JavaScript node:

```
if (typeof window.$countdownMap === 'undefined') {
window.$countdownMap = new Map();
}
if (window.$countdownMap.has($parameters.Class)) {
$parameters.Success = false;
$parameters.ErrorMessage = "Countdown for class '" +
$parameters.Class + "' already exists";
return;
}
window.$countdownMap.set($parameters.Class, new Countdown(
{
cont: document.querySelector('.' + $parameters.Class),
date: $parameters.Deadline.valueOf(),
outputTranslation: {
year: 'Years',
week: 'Weeks',
day: 'Days',
hour: 'Hours',
minute: 'Minutes',
second: 'Seconds',
},
endCallback: null,
outputFormat: 'week|day|hour|minute|second',
}));
window.$countdownMap.get($parameters.Class).start();
$parameters.Success = true;
```

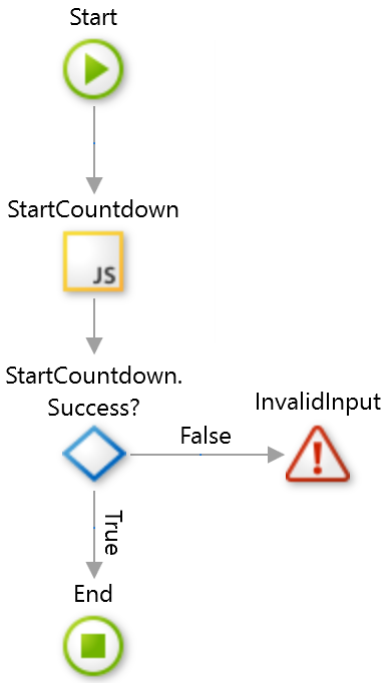
f) The final **StartCountdown** JavaScript node should be similar to the screenshot:



g) Drag an *If* tool after the **StartCountdown** node and set its *Condition* to `StartCountdown.Success`

- The *False* branch should raise a new *User Exception* named `InvalidInput`, with its *Exception Message* property set to `StartCountdown.ErrorMessage`
- The *True* branch should simply end

h) Your OnReady event handler should look similar to the following.



4) When the block is no longer needed we need to make sure it is properly discarded, performing any cleanup tasks required: in this case that would mean stopping the Countdown if it's still running

   a) Add a second event handler to the block, this time for the *OnDestroy* event

   b) Drag the JavaScript tool to the action flow of the newly created **OnDestroy** screen action, name it `StopCountdown` and add the *Class* input parameter of the block as its sole argument

   c) Open the **StopCountdown** node and add the same two output parameters:

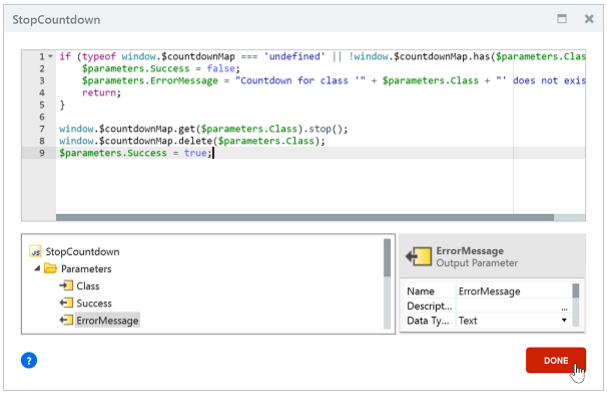
      - A *Boolean* named `Success`
      - A *Text* named `ErrorMessage`

   d) Paste the following JavaScript code into the code editor area:

```
if (typeof window.$countdownMap === 'undefined' ||
!window.$countdownMap.has($parameters.Class)) {
$parameters.Success = false;
$parameters.ErrorMessage = "Countdown for class '" +
$parameters.Class + "' does not exist";
return;
}
window.$countdownMap.get($parameters.Class).stop();
window.$countdownMap.delete($parameters.Class);

$parameters.Success = true;
```

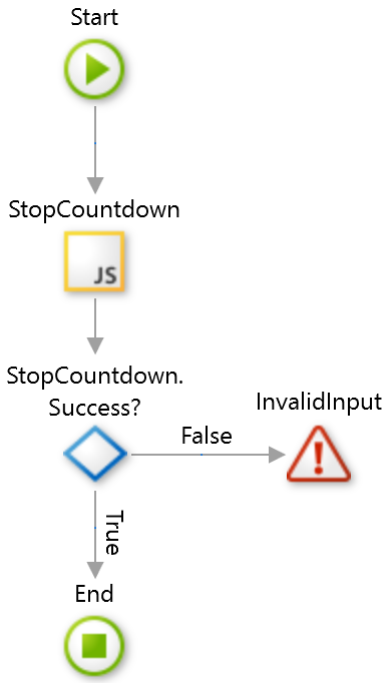
e) The final **StopCountdown** JavaScript node should be similar to the screenshot:



f) Drag an *If* tool after the *StopCountdown* node and set its *Condition* to `StopCountdown.Success`

- The *False* branch should raise a new *User Exception* named `InvalidInput`, with an *Exception Message* of `StopCountdown.ErrorMessage`
- The *True* branch should simply end

g) Your **OnDestroy** event handler should look similar to the following.



Enhancing the Countdown block for developers

The **Countdown** block is going to be used by developers to display to end users a flip-number countdown but, as is, developers will not have any preview of what the countdown will look like when they are designing their screens. Worse even, since the only element of the block is an invisible container, they won't even be able to select it directly in the canvas (it will be only accessible from the *Widget Hierarchy* breadcrumb below or the *Widget Tree* on the right.

A common approach to be able to preview this kind of blocks at design time is to provide some sort of content that's going to be displayed inside Service Studio, and different from the actual contents used at runtime.

NOTE: When previewing the contents of a **Block's instance**, Service Studio will do static analysis of the code to determine at design-time what is expected to be visible and use that as the preview. When you have an *If* widget in a **Block**, Service Studio cannot determine which branch will be visible, and as such uses the contents of the *True* branch. We can use this behaviour to our own advantage to provide our own preview for patterns.

1) Provide a preview for the block by using a screenshot of how the countdown will be shown to end-users

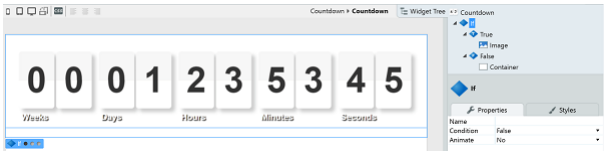
    a) Open the **Countdown** block's canvas, drag an *If* widget before the single existing container and set its *Condition* to `False`

b) Move the existing container into the *False* branch of the *If* widget

c) In the Interface tab, add the **preview.png** image resource to your module

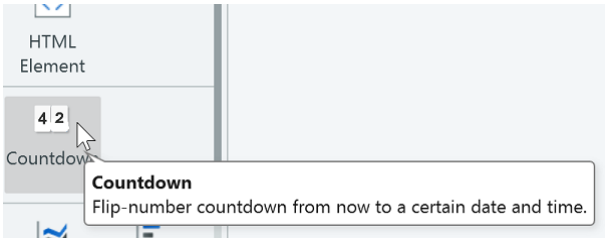
NOTE: Dragging an image file directly to the *Image* folder will add that image to the module's image resources

d) Drag the newly added **preview** image resource to the *True* branch of the *If* widget



2) Finally, in order to explain to other developers what the block does, we will also set the block's Description property to `Flip-number countdown from now to a certain date and time.`

NOTE: Blocks that have a *Custom Icon* will automatically be displayed by Service Studio in the Toolbox. Adding a description makes the tooltip a lot more useful



Challenges

1) Allow the developers to define the format of the countdown displayed by a block's instance. This is defined by the *outputFormat* value in the *JavaScript* code

```
24          outputFormat: 'week|day|hour|minute|second',
25      }));
26
27  window.$countdownMap.get($parameters.Class).start();
```

NOTE: The current output format **week|day|hour|minute|second** should still be the default, but the developers will be able to customize individual instances of the block

2) Trigger an *Event* when the countdown ends. When this happens, the JavaScript library will execute the callback function referenced by the *endCallback* value

```
20              minute: 'Minutes',
21              second: 'Seconds',
22          },
23          endCallback: null,
```

NOTE: In JavaScript, when you want to pass a reference to a function (instead of executing it) you use the name of the function without the final parentheses.

You will need to implement a `TriggerFinished` screen action to trigger the new event and then change the value of *endCallback* to `$actions.TriggerFinished`