openSAP Evolved Web Apps with SAPUI5

**Week 2 Unit 2:** Waking up your UI Controls with Events  
  
Exercises

PUBLIC



Table of ContentS

[Waking up your UI Controls with Events 3](#_Toc7507595)

[Add inputs and a button to the app 4](#_Toc7507596)

[Make the app interactable by implementing an event listener function in the controller 8](#_Toc7507597)

[Continue by using the event parameter syntax XML view 12](#_Toc7507598)

[Other Controller Lifecycle Events 16](#_Toc7507599)

[Related material 22](#_Toc7507600)

# Waking up your UI Controls with Events

### Summary

In this unit you will learn about

* Controls structure and how they work
* How to trigger events in SAPUI5 from the controller and from the view
* Other use cases of the controllers

### Preview

In the last unit you have made a simple app, containing a single sap.m.Image. Now let’s extend the application with some input fields which trigger different events by user interaction. They will be used for searching for movies by city and genre.

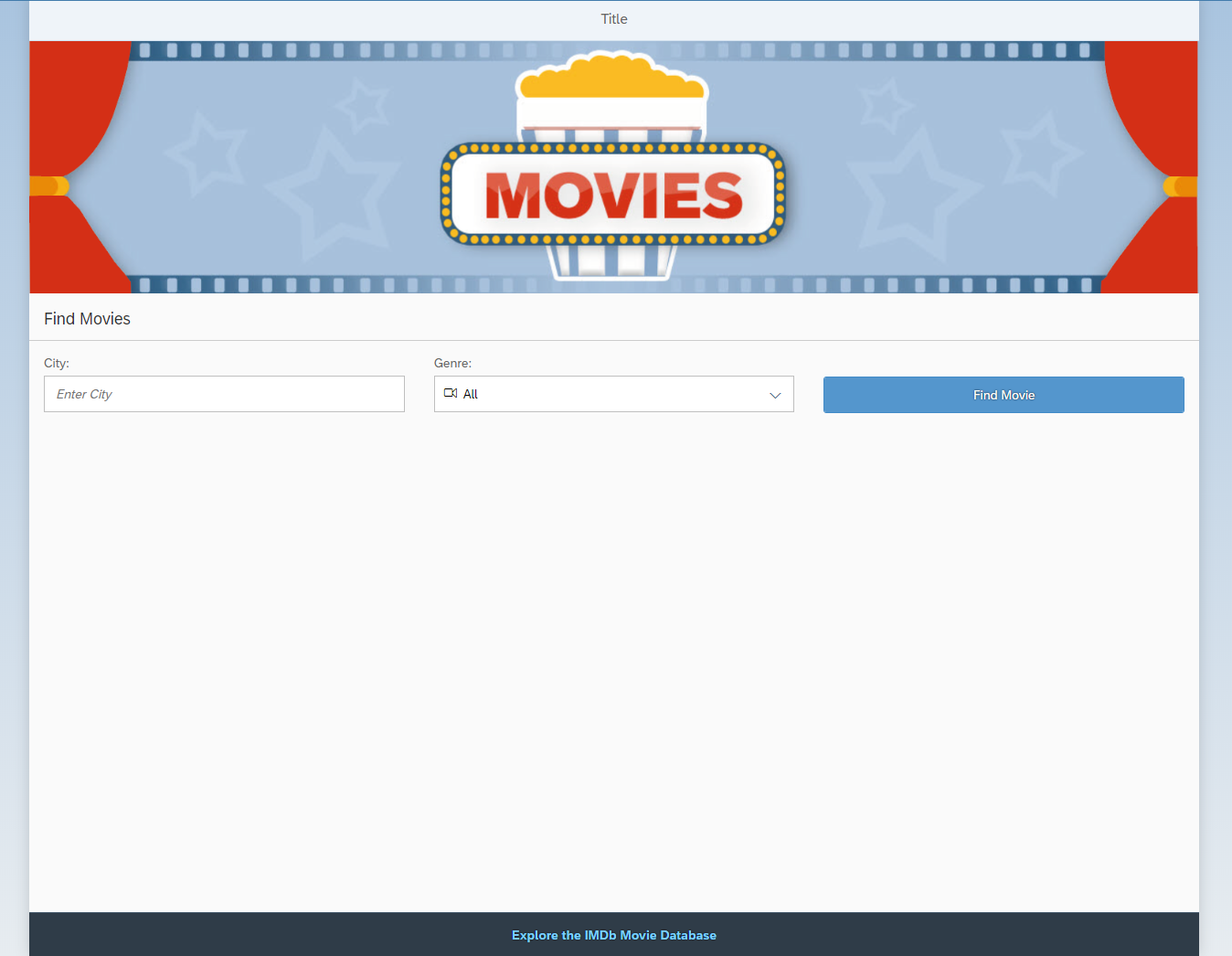


Figure 1 – The App at the end of the unit includes two search fields

## Add inputs and a button to the app

It is a good idea to enable the users of the movie app to search for trending movies and their showtimes. Let’s add some fields which will accept values for the city the user is interested in and the genre, which the user likes.

**webapp/view/App.view.xml**

|  |
| --- |
| <mvc:View controllerName="opensap.movies.controller.App"  displayBlock="true"  xmlns="sap.m"  xmlns:mvc="sap.ui.core.mvc"  xmlns:core="sap.ui.core">  …  <Page title="{i18n>title}">  <content>  <Image  src="images/MoviesHeader.png"  width="100%"  tooltip="Movie illustration">  </Image>  <Label text="City"  labelFor="city"/>  <SearchField id="city"  showSearchButton="false"  placeholder="Enter City"/>  <Label text="Genre"  labelFor="genre"/>  <Select id="genre">  <core:ListItem icon="sap-icon://video" key="" text="All"/>  <core:ListItem icon="sap-icon://physical-activity" key="Action" text="Action"/>  <core:ListItem icon="sap-icon://electrocardiogram" key="Horror" text="Horror"/>  <core:ListItem icon="sap-icon://paper-plane" key="SienceFiction" text="Science Fiction"/>  </Select>  <Label />  <Button text="Find Movies"  class="sapUiSmallMarginTop"  type="Emphasized"/>  </content>  <footer>  …  </footer>  </Page>  … |

Look deeper at the structure of an UI5 element - the UI definition and the behavior encapsulated as a reusable asset – in UI5 we name it control. For reference, we have already included a sap.m.SearchField, a sap.m.Select and a sap.m.Button controls in our movie app, each accompanied by a sap.m.Label control. Several configurations are available for every control – each can have *properties, aggregations*, *associations* and *events*.

In the demo we used the id *property* of the select and the search field. The connection between each label and the labeled control is made by the *property* labelFor. Its value is the ID of the corresponding labeled field. Each label has also a text value, the SearchField has a defined placeholder *property*.



Figure 2 – The current state of the app

To beautify the appearance of our input control we add a SimpleForm control. It is in the sap.ui.layout library and is used mostly for structuring some controls in the UI.

**webapp/view/App.view.xml**

|  |
| --- |
| <mvc:View controllerName="opensap.movies.controller.App"  displayBlock="true"  xmlns="sap.m"  xmlns:mvc="sap.ui.core.mvc"  xmlns:core="sap.ui.core"  xmlns:f="sap.ui.layout.form">  …  <Page title="{i18n>title}">  <content>  <Image  src="images/MoviesHeader.png"  width="100%"  tooltip="Movie illustration">  </Image>  <f:SimpleForm  id="form"  editable="true"  layout="ColumnLayout"  title="Find Movies"  columnsM="2"  columnsL="3"  columnsXL="3">  <f:content>  <Label text="City"  labelFor="city"/>  <SearchField id="city"  width="100%"  showSearchButton="false"  placeholder="Enter City"/>  <Label text="Genre"  labelFor="genre"/>  <Select id="genre"  width="100%">  <core:ListItem  icon="sap-icon://video"  key=""  text="All"/>  <core:ListItem  icon="sap-icon://physical-activity"  key="Action"  text="Action"/>  <core:ListItem  icon="sap-icon://electrocardiogram"  key="Horror"  text="Horror"/>  <core:ListItem  icon="sap-icon://paper-plane"  key="ScienceFiction"  text="Science Fiction"/>  </Select>  <Label />  <Button text="Find Movies"  type="Emphasized"  class="sapUiSmallMarginTop"/>  </f:content>  </f:SimpleForm>  </content>  <footer>  …  </footer>  </Page>  …  </mvc:View> |

**Tip: Beautify XML or JavaScript**When copy and pasting code from the exercises or examples, the indendation and formatting of the code might become a bit messy sometimes. To automatically clean up your code, right-click anywhere in the code window and select *Beautify*. This will properly indent and reformat the current file.

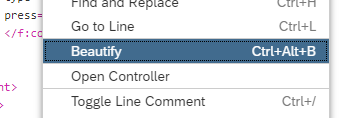


Figure 1 - Beautify Option in the right-click menu

**Note:** Although the code is working, there might be some validation warnings in the Web IDE in some of the code lines. If this is the case – ignore them.

The sap.ui.layout.SimpleForm control is used to structure the look of the app. Set its layout shape to ColumnLayout and define how much columns to be shown depending on the screen size via the columnM, columnL and columnXL *properties*.

The content *aggregation* of the SimpleForm is used to store the controls to be displayed in it. Using *aggregations****,*** you can nest some controls to be displayed inside others.

In the demo some ListItems to items *aggregation* of the Select are included. When one *aggregation* is a default one, you can miss writing its XML tag.

An *association* to the Select control is the currently selected item. You can think of it as some kind of a pointer to a semantic part of the control. It can be defined in the selectedItem *association* of the sap.m.Select. In the demo one is not set. In this situation the framework automatically sets the first ListItem as associated.

More about the structure of the control elements in the UI5 framework can be read in our [Demo Kit](https://ui5.sap.com/) application. It contains info about the API of all controls, some useful samples and additional documentation.

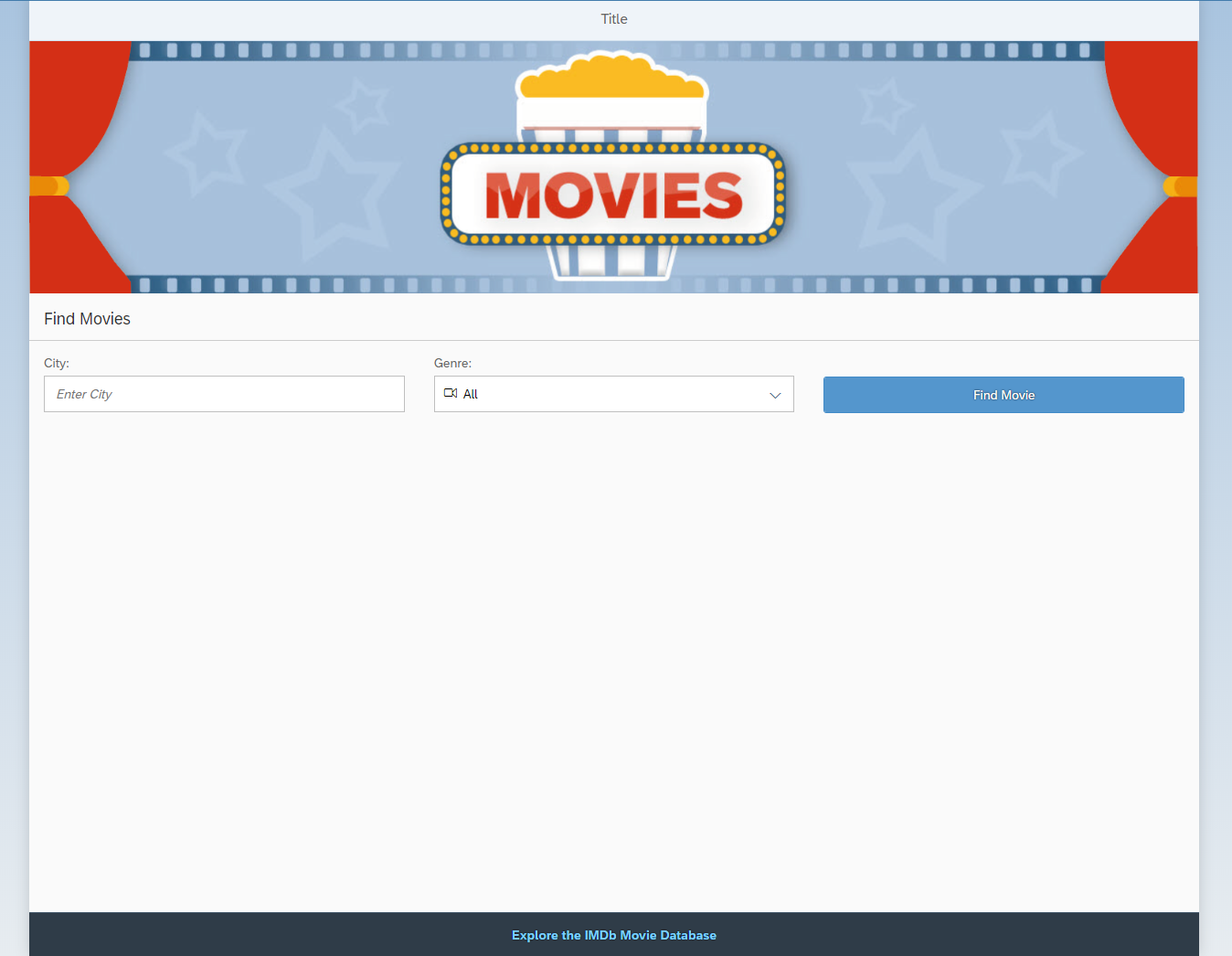


Figure 3 – The app now looks more ordered

## Make the app interactable by implementing an event listener function in the controller

After entering the values in the input fields, the user must somehow trigger the search functionality. For this, you will trigger an event which will notify the app that the entered values are ready to be searched.

**webapp/view/App.view.xml**

|  |
| --- |
| …  <Page title="{i18n>title}">  <content>  …  <Select id="genre"  width="320px">  …  </Select>  <Label />  <Button text="Find Movies"  type="Emphasized" class="sapUiMediumMarginTop"  press=".onPress"/>  </f:content>  </f:SimpleForm>  </content>  <footer>  …  </footer>  </Page>    … |

You declare that the app will use the press event of the *Find Movies* button in the XML view and connect an event handler function to it in the controller of the view. Here, you inform the user that the search has been triggered. This can be easily done by using a sap.m.MessageToast control, which displays a message at the lower center of the screen.

When the button is clicked, the HTML browser onclick event of the HTML button element is triggered, and the semantic UI5 press event of the sap.m.Button control is fired. Then the onPress function is evoked, which is in the corresponding controller of the view.

**webapp/controller/App.controller.js**

|  |
| --- |
| sap.ui.define([  "sap/ui/core/mvc/Controller"  ], function (Controller) {  "use strict";  return Controller.extend("opensap.movies.controller.App", {    onInit: function () {  },    onPress: function (oEvent) {  sap.ui.require(["sap/m/MessageToast"], function (oMessage) {  oMessage.show("Searching...");  });  }  });  }); |

In the handler function, display a MessageToast with a message to the user saying *Searching…* The sap.m.MessageToast control is loaded asynchronously using the sap.ui.require function when the press event is triggered. This will reduce the size of the application and potentially improve its initial loading time.

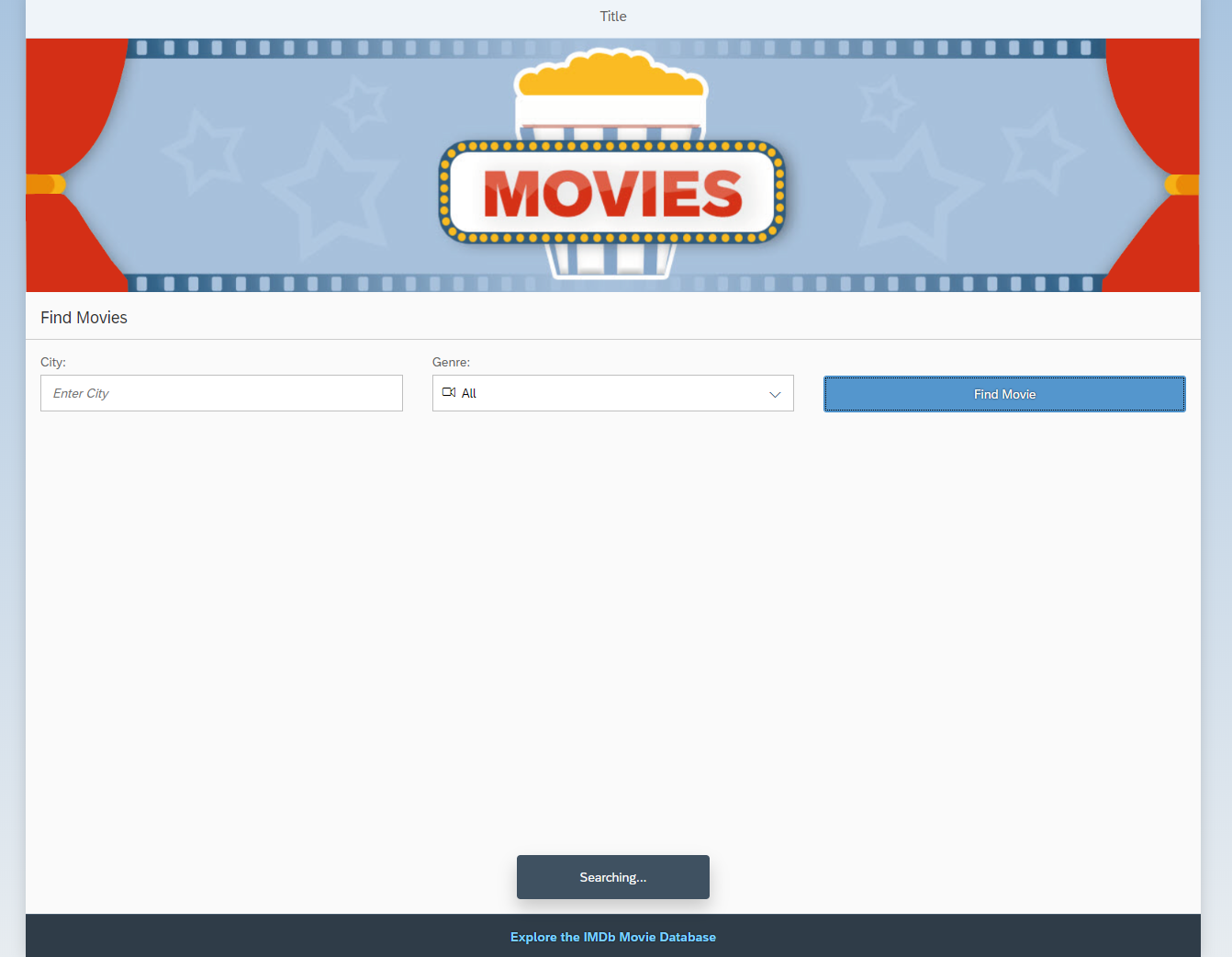


Figure 4 – Now when the button gets clicked a MessageToast appears on the screen

Let’s explore the handler function a bit more and therefore open the Chrome DevTools by pressing **F12**. On the *Sources* tab, open the App.controller.js file by pressing **Ctrl+O** and typing its name. Put a breakpoint on line 12 by clicking on the number at the beginning of the line.

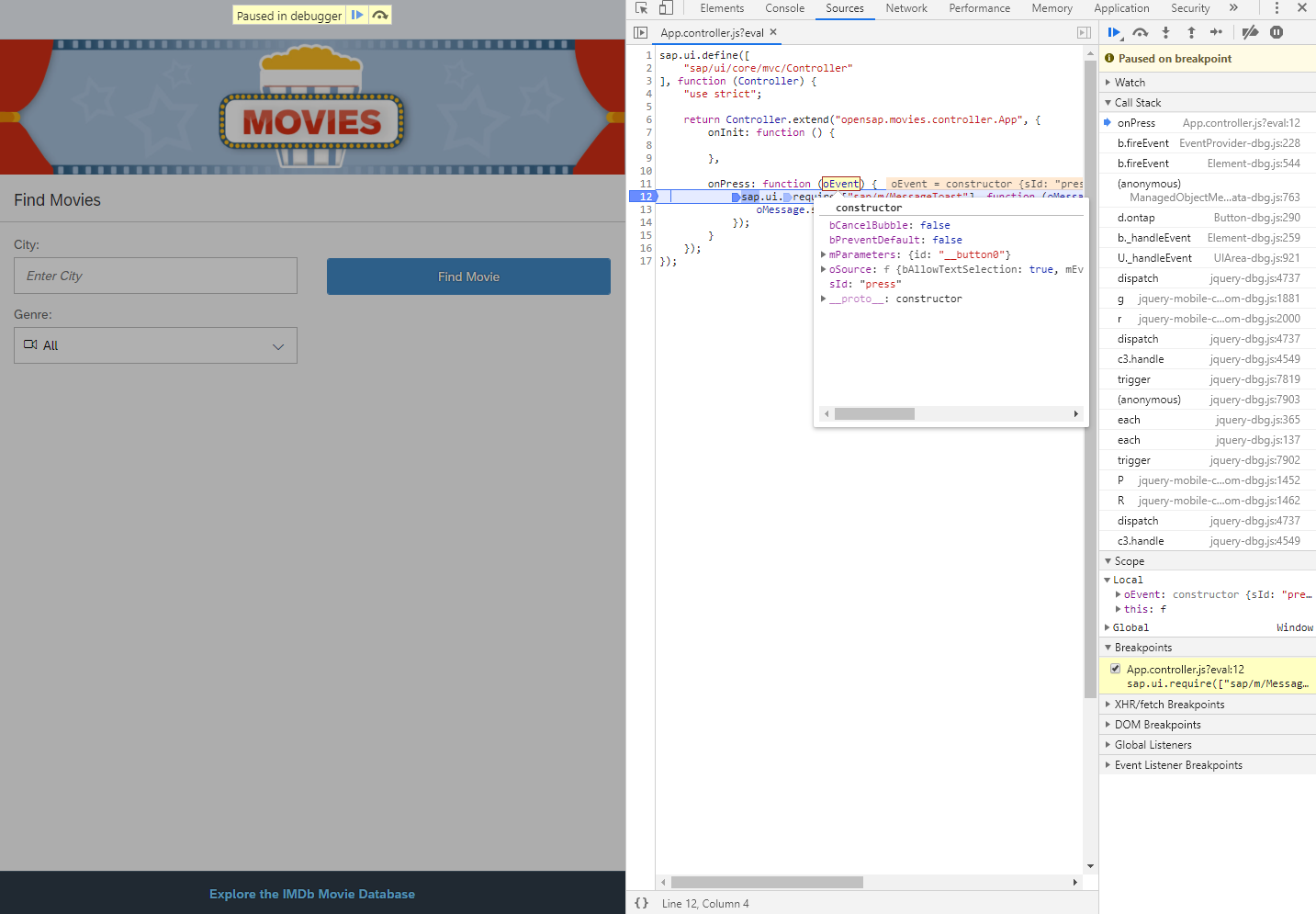


Figure 5 – When the browser stops at the breakpoint, you can see what the oEvent variable contains

To stop the code during its execution at the breakpoint, you have to trigger the corresponding event – press, for the *Find movies* button. Hover with the mouse over the first argument passed to the handler function – oEvent. This is a generic event object containing information about the control that triggered it, and passed semantic information about the event.

## Continue by using the event parameter syntax XML view

A newly available framework feature is the event parameter syntax in the XML views. It allows a more flexible way of handling events.

When declaring events in the XML, you can also specify a list of parameters that are passed to the handler. The passed value can then be used directly in the controller method. You can specify JavaScript literals like strings, numbers, arrays, and objects. We can also use expression binding syntax and translated text from the i18n model, which will be explained in the next unit.

Let’s try it out by extending the MessageToast text.

**webapp/view/App.view.xml**

|  |
| --- |
| …  <Page title="{i18n>title}">  <content>  …  <Select id="genre"  width="320px">  </Select>  <Label />  <Button text="Find Movies"  type="Emphasized" class="sapUiMediumMarginTop"  press=".onPress('for movies!')"/>  </f:content>  </f:SimpleForm>  </content>  <footer>  …  </footer>  </Page>    … |

**Note:** Although the code is working, there might be some validation warnings in the Web IDE in some of the code lines. If this is the case – ignore them.

Pass a string to the event handler function and use it in the controller.

**webapp/controller/App.controller.js**

|  |
| --- |
| sap.ui.define([  "sap/ui/core/mvc/Controller"  ], function (Controller) {  "use strict";  return Controller.extend("opensap.movies.controller.App", {    onInit: function () {  },    onPress: function (sValue) {  sap.ui.require(["sap/m/MessageToast"], function (oMessage) {  oMessage.show("Searching..." + sValue);  });  }  });  }); |

As there is a passed argument from the XML view, the first argument in the handler function is now the passed value. Therefore, you need to change the oEvent variable to sValue.

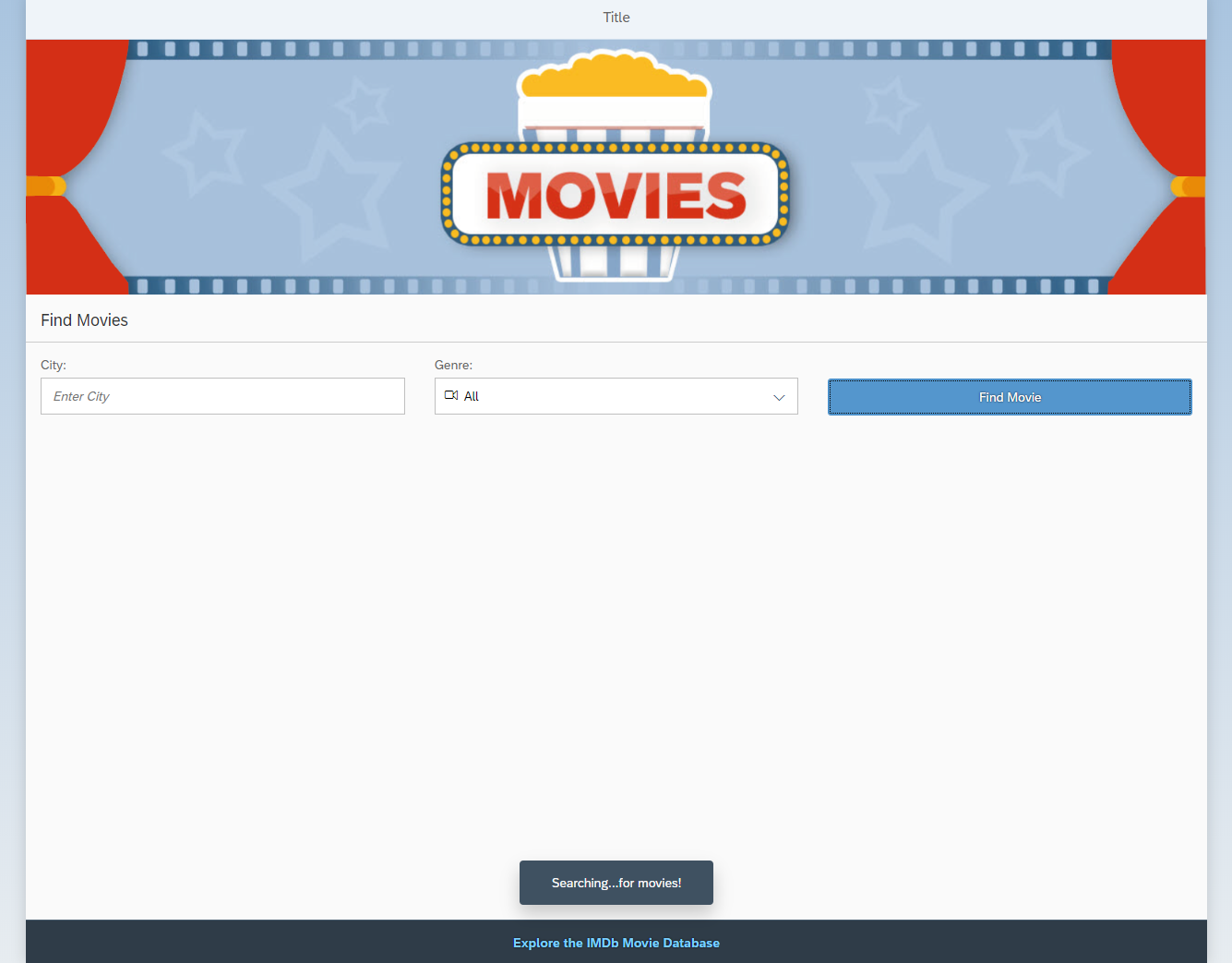


Figure 6 – Now the MessageToast contains the passed string

Again, let’s examine the code a bit further by setting a breakpoint at the exact same place in the handler function.

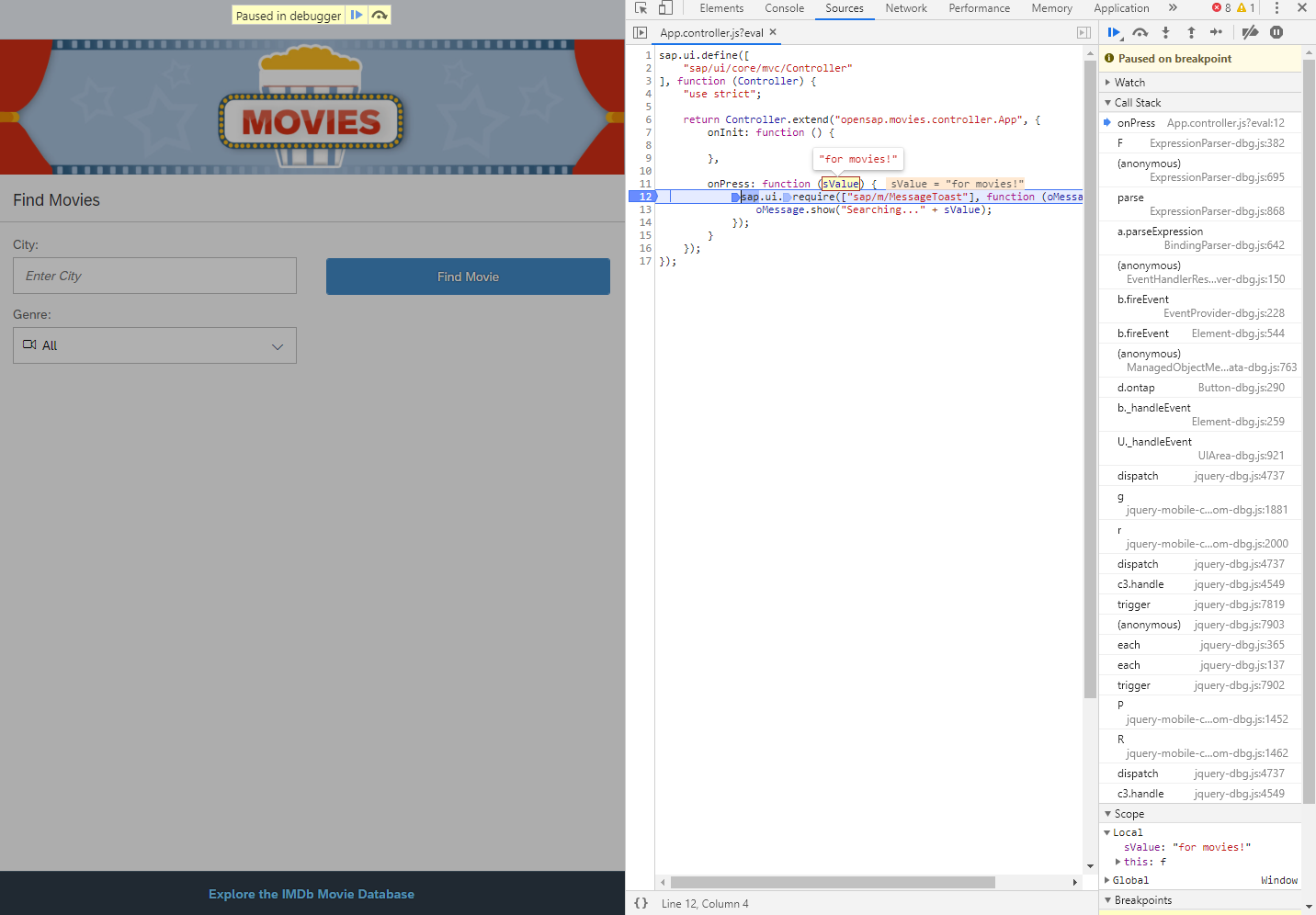


Figure 7 – When the browser stops at the breakpoint, you can see what the sValue variable contains

You can see that the first passed argument to the handler function is the one included in the XML.

For very simple event handlers, you can directly define the event logic in the view. For demo purposes, show a simple message to the user, when they click on the image.

**webapp/view/App.view.xml**

|  |
| --- |
| …  <Page title="{i18n>title}">  <content>  <Image  src="images/MoviesHeader.png"  width="100%"  tooltip="Movie illustration"  press="sap.m.MessageToast.show('Do you feel like going to the movies?')">  </Image>  <f:SimpleForm  id="form"  editable="true"  layout="ColumnLayout"  title="Find Movies"  columnsM="2"  columnsL="3"  columnsXL=”3”>  <f:content>  <Label text="City"  labelFor="city"  class="sapUiSmallMarginTop"/>  …  </content>  </Page>  … |

**Note:** Although the code is working, there might be some validation warnings in the Web IDE in some of the code lines. If this is the case – ignore them.

Here, when the semantic press event is fired, the script in the quotes is executed. Again, a simple text is shown to the user via a sap.m.MessageToast.

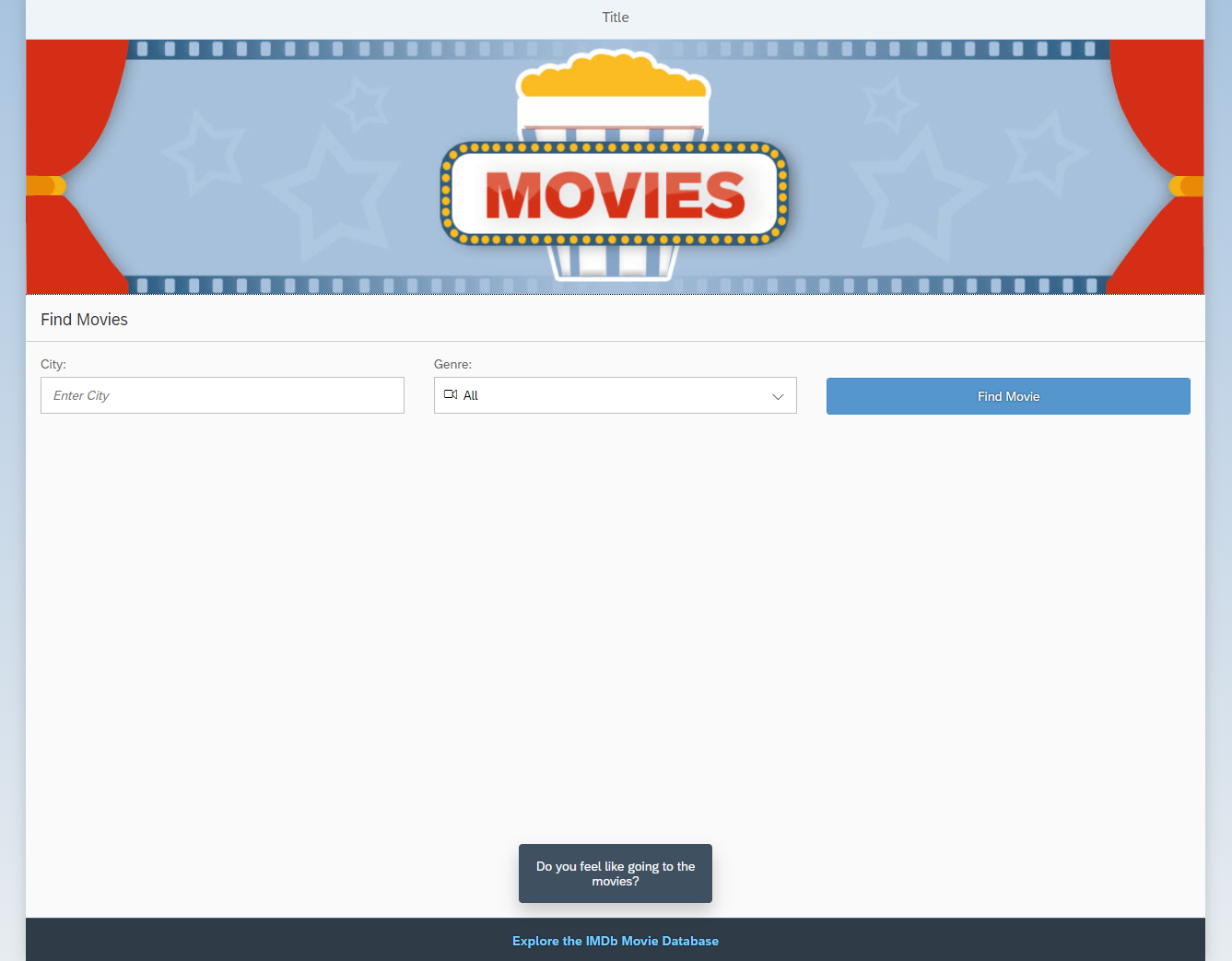


Figure 8 –Again, when the image gets clicked, a MessageToast appears on the screen

## Other Controller Lifecycle Events

A controller is not only used for event handlers. It has also some specific events, which are called “lifecycle hooks” – onInit, onBeforeRendering, onAfterRendering and onExit. They are invoked by the UI5 framework at key points in the lifecycle of the application.

Take a closer look at these events by introducing the logging concept of the framework. Applications can set a log level so that only critical errors are shown in productive mode and more debug information is shown in a development environment.

To ease debugging of complex application logic, it’s a good idea to write debug logs.

For more information about logging and tracking messages go to [Logging and Tracing](https://ui5.sap.com/#/topic/9f4d62c6648a423d85aaf2bfc2c7ddfe) in the Demo Kit.

**webapp/index.html**

|  |
| --- |
| <!DOCTYPE HTML>  <html>  <head>  <meta charset="utf-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>movies</title>  <script id="sap-ui-bootstrap"  src="../../resources/sap-ui-core.js"  data-sap-ui-theme="sap\_belize"  data-sap-ui-resourceroots='{"opensap.movies": "./"}'  data-sap-ui-compatVersion="edge"  data-sap-ui-logLevel="debug"  data-sap-ui-oninit="module:sap/ui/core/ComponentSupport"  data-sap-ui-async="true"  data-sap-ui-frameOptions="trusted">  </script>  </head>  <body class="sapUiBody">  <div data-sap-ui-component data-name="opensap.movies" data-id="container" data-settings='{"id" : "movies"}'></div>  </body>  </html> |

First, enable the app to show information necessary for debugging. You do this in the index.html file while bootstrapping UI5.

**webapp/controller/App.controller.js**

|  |
| --- |
| sap.ui.define([  "sap/ui/core/mvc/Controller",  "sap/base/Log"  ], function (Controller, Log) {  "use strict";  return Controller.extend("opensap.movies.controller.App", {    onInit: function () {  Log.info("Controller has been initialized.");  },    onExit: function () {  Log.info("Controller will shortly be destroyed.");  },  onPress: function (sValue) {  sap.ui.require(["sap/m/MessageToast"], function (oMessage) {  oMessage.show("Searching..." + sValue);  });  }    });  }); |

When you run the app and open the browser console, you can see the logged message.

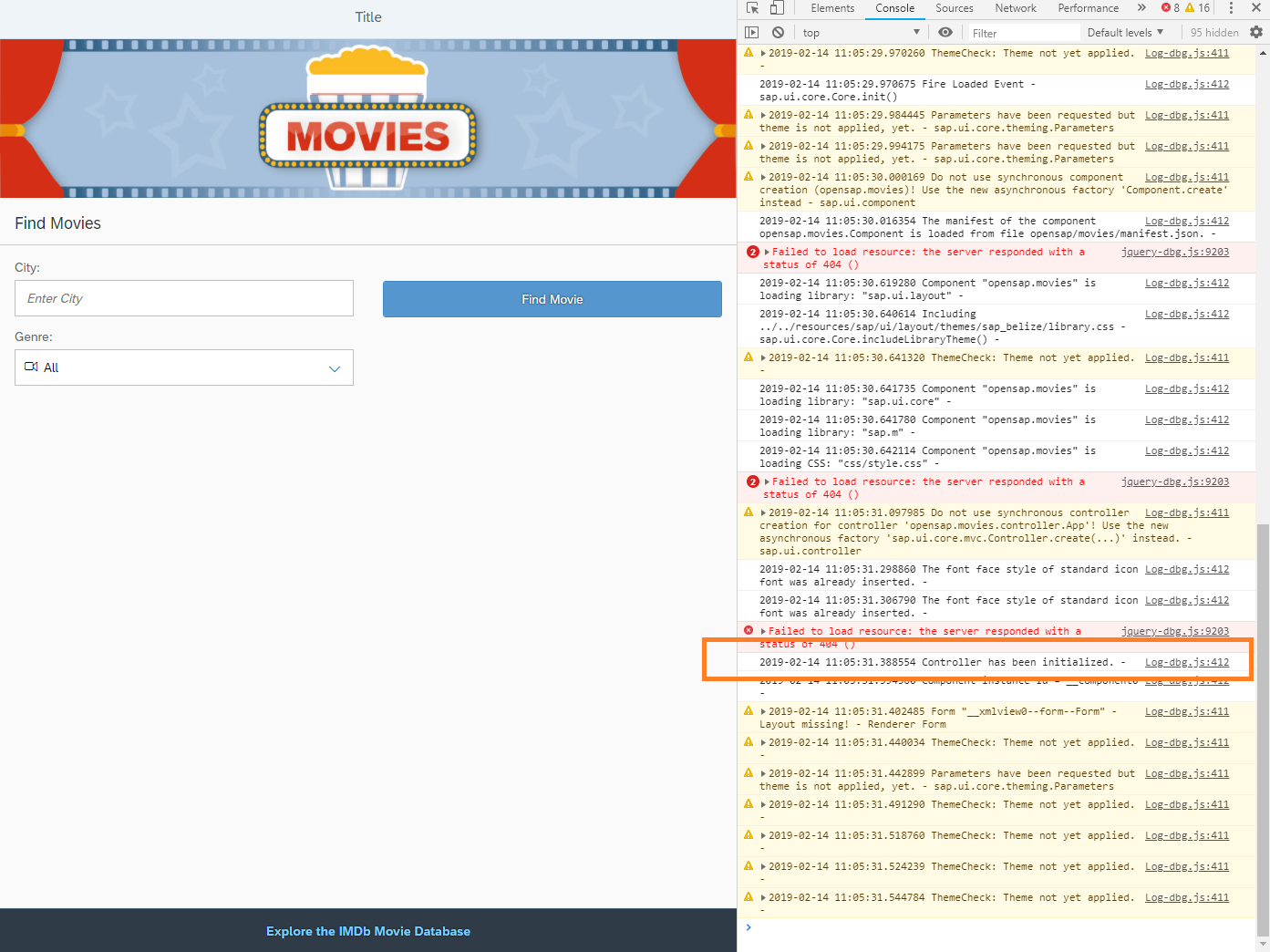


Figure 9 – You can see the logged message in the browser console.

The message is logged when the controller is initialized, because the framework has called the onInit lifecycle hook. This method can also contain some queries to the backend for example.

onExit is called when the view is closed, so you will see the message when the view is destroyed. This method can help by destroying some objects in it.

**webapp/controller/App.controller.js**

|  |
| --- |
| sap.ui.define([  "sap/ui/core/mvc/Controller",  "sap/base/Log"  ], function (Controller, Log) {  "use strict";  return Controller.extend("opensap.movies.controller.App", {    onInit: function () {  Log.info("Controller has been initialized.");  },  onBeforeRendering: function () {  Log.info("The view will shortly be rendered.");  },    onAfterRendering: function () {  Log.info("The view has been rendered.");  },    onExit: function () {  Log.info("Controller will shortly be destroyed.");  },  onPress: function (sValue) {  sap.ui.require(["sap/m/MessageToast"], function (oMessage) {  oMessage.show("Searching..." + sValue);  });  }    });  }); |

onBeforeRendering is called before the rendering has happened. It’s useful, for example, for checking the device parameters.

The onAfterRendering is called when the rendering has already happened on the page. It’s used to check something in the DOM for example.

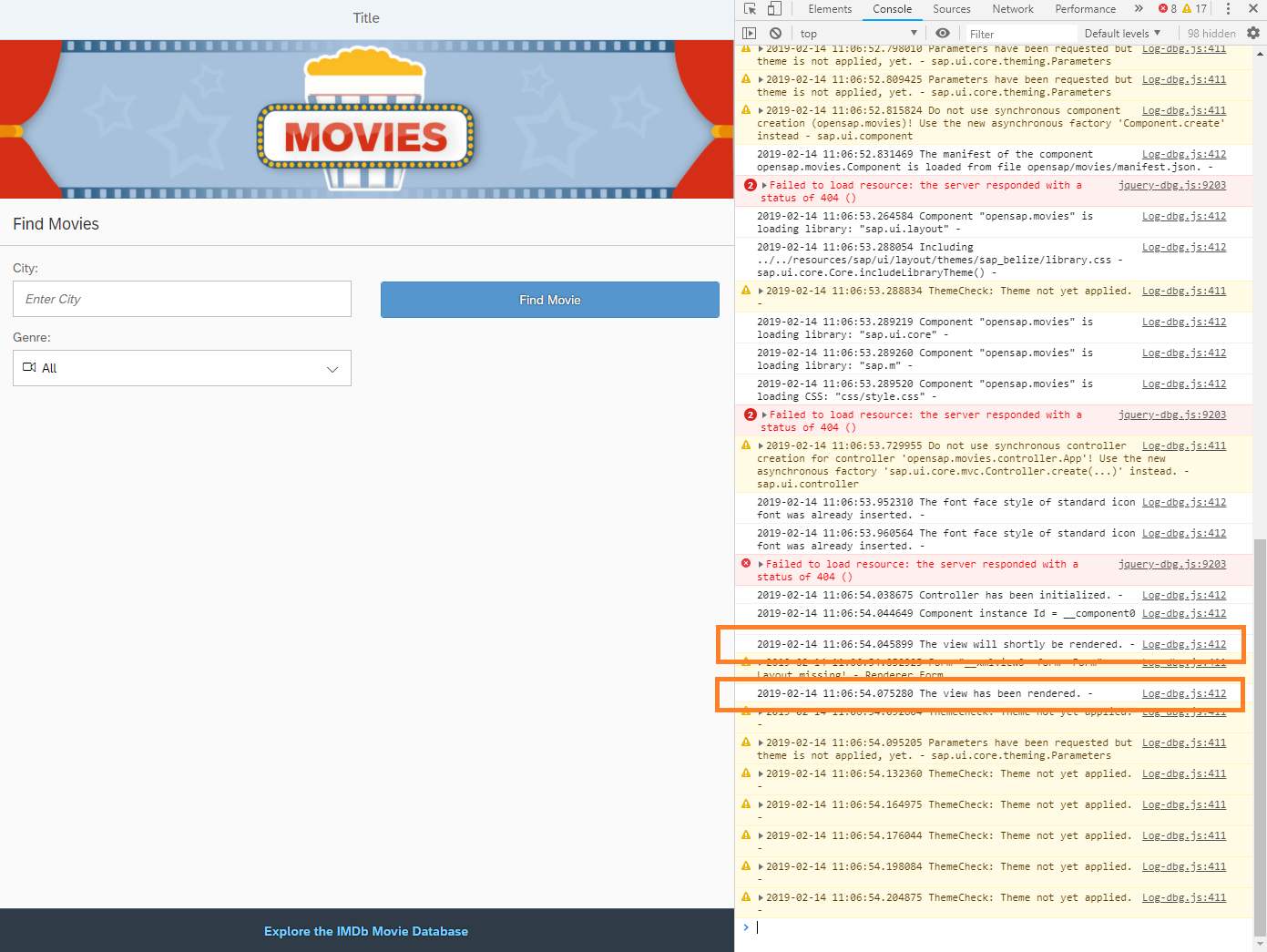


Figure 10 – You can see the logged messages in the browser console right after the one from the onInit function.

The controller is also used for adding helper functions to the event handler functions or for formatters.

**webapp/index.html**

|  |
| --- |
| <!DOCTYPE HTML>  <html>  <head>  <meta charset="utf-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>movies</title>  <script id="sap-ui-bootstrap"  src="../../resources/sap-ui-core.js"  data-sap-ui-theme="sap\_belize"  data-sap-ui-resourceroots='{"opensap.movies": "./"}'  data-sap-ui-compatVersion="edge"  data-sap-ui-logLevel="debug"  data-sap-ui-oninit="module:sap/ui/core/ComponentSupport"  data-sap-ui-async="true"  data-sap-ui-frameOptions="trusted">  </script>  </head>  <body class="sapUiBody">  <div data-sap-ui-component data-name="opensap.movies" data-id="container" data-settings='{"id" : "movies"}'></div>  </body>  </html> |

Logging more messages in the console might slow down the execution time of the app. As now the logging concept is shown only for demo purposes, remove the flag from the index.html file.

# Related material

* [Demo Kit: API Reference](https://ui5.sap.com/#/api)
* [Demo Kit: Predefined CSS Margin Classes](•%09https:/ui5.sap.com/#/topic/777168ffe8324873973151dae2356d1c)
* [Blog: Event Parameter Syntax in XML views](https://blogs.sap.com/2018/08/09/ui5ers-buzz-34-new-event-parameter-syntax-in-xmlviews/)
* [Demo Kit: Logging and Tracing](https://ui5.sap.com/#/topic/9f4d62c6648a423d85aaf2bfc2c7ddfe)