PART I : Razor Pages (1st semester)

Who this tutorial is for

This tutorial is for students having an intermediate knowledge of C# and are looking to throughout understanding of developing web applications using ASP.NET Core RazorPages framework. This tutorial will take you through a long voyage, full of details, step by step with no fear to get lost on the way. Mr GitHub is all the time at your help. Mr GitHub can provide you with all the code examples used throughout the tutorial. Fasten your belt. You are going through the following chapters:

Chapter 1: HTML

In this chapter, you will get a good understanding of HTML. Indeed, we will cover more than what is necessary to work with in this tutorial. We focus on the very important HTML elements such as tables, forms, images, links…etc.

Chapter 2: CSS

In this chapter, you will learn the basics of CSS. You will get the necessary to understand what happens behind the scene when working with Bootstrap which is the subject of the next chapter. You will learn how to create and apply a CSS file to a page.

Chapter 3: Bootstrap

This chapter covers a big part of Bootstrap. You will learn the most important Bootstrap classes including container and navbar classes. You will also learn bootstrap applied to buttons, forms , inputs, form validation, tables, images …etc.

Chapter 4: Your First Razor Pages Application

After getting a correct knowledge of these fundamental technologies for building web applications, you are going to build your first application. You will first be introduced to the ASP.NET Core framework in general, its benefits ...etc. You will build the first Razor Pages application and explore the default file structure. You will also get the opportunity to deeply understand where the application bootstraps and where it ends by investigating the code execution sequence from the time it starts.

Chapter 5: Razor pages architecture

Before starting coding a web app, it is crucial to understand the architecture of the framework you are working with and how the user request is processed. This is the purpose of this chapter. In this chapter, you will get a good understanding of which class/method is intercepting, handling the request, how the request is routed and how the response is returned to the user.

Chapter 6 : EventMakerRazorPages application- GetAllEvents

After getting the hang of how a request is processed in a Razor Pages application, you will start implementing the EventMakerRazorPages application. It is about managing events in Denmark. In this chapter, you will implement the GetAllEvents method to display all the events from a list. At this stage, you are not going to work with a real data storage (i.e. SQL Database). For the moment, a List data structure can do the job.

Chapter 7 : Create a new event : CreateEvent

In this chapter, you will continue implementing the CRUD operations. In this chapter, you implement the CreateEvent method for adding new events to the list. Having some issues with the implementation, you will be introduced to the Singleton design pattern to solve the issues related to implementing the CreateEvent functionality.

Chapter 8: Data Validation and Singleton design pattern

In this chapter, you will implement the Singleton design pattern to solve the issues related to implementing the CreateEvent functionality. You will also address the question "***what if the user enters invalid data?***". To answer this question, you will learn and apply validation.

Chapter 9: Routing - EditEvent

In this chapter, you will implement the EditEvent method. To edit a specific event, data about this specific event is passed from a page to another page. This requires knowledge about routing. You will be introduced to routing in RazorPages before implementing this functionality. You will also implement “filtering the displayed events based on the city”. As an exercise, you are supposed to implement “Delete an event” and “display the details of an event”.

Chapter 10: Dependency injection

In this chapter, you will get rid of the code used to implement the Singleton Design pattern. Thanks to the way that ASP.NET Core implements Dependency Injection , a singleton service is defined, configured, and then used across the application. You will see how easy it is to implement this pattern.

Chapter 11: Repository Design Pattern - Json file storage

In this chapter, you will implement another data access layer using a Json file as data storage. To abstract the data access layer, the Repository Design Pattern is used. At the end of this chapter, you will realize how easy it is to incorporate another data access layer with any change to the existing code, which enhances maintainability.

Chapter 12: Testing

In this chapter, we will test the application that you have just built; we are mainly going to look at Unit Testing.

Chapter 13: Building a Real application: 1- * relationship using the Fake Repository with a list

In this chapter, we are pushing our implementation further towards real application behaviour. We are going to add another class and implement a **1-many** relationship. This time , we are still using the FakeEvent repository data access layer using a list

Chapter 14: Building a Real application: 1- * relationship using a json file with a dictionary

In this chapter, we are still implementing the **1-many** relationship. However , this time we are using the json file as data storage along with a dictionary as the data structure.

Chapter 1: Introduction to HTML 5

Introduction

In this chapter, you will be introduced to HTML. HTML stands for **H**yper **T**ext **M**arkup **L**anguage. The word “Markup” means that the language is a declarative; a human-readable language using tags and attributes to define the different elements of the page and their

content. So HTML is not a programming language, rather it is a standard for structuring the page content of the Web. We can modify the page content and its appearance by setting attributes on the different tags. In general, attributes are expressed as name-value pairs.

An example of an HTML page is given below.

```
<!DOCTYPE html>
<html>
    <head>
        <title>Enter a title, displayed at the top of the window.</title>
    </head>
    <!-- The information between the BODY and /BODY tags is displayed.-->
    <body>
        <h1>Enter the main heading, usually the same as the title.</h1>
        <p>Be <b>bold</b> in stating your key points. Put them in a list: </p>
        <ul>
            <li>The first item in your list</li>
            <li>The second item; <i>italicize</i> key words</li>
        </ul>
        <p>Improve your image by including an image. </p>
        <p><img src="http://www.mygifs.com/CoverImage.gif" alt="A Great HTML Resource"></p>
        <p>
            Add a link to your favorite <a href="https://www.dummies.com/">Web site</a>.
            Break up your page with a horizontal rule or two.
        </p>
        <hr>
        <p>Finally, link to <a href="page2.html">another page</a> in your own Web site.</p>
        <!-- And add a copyright notice.-->
        <p>&#169; Wiley Publishing, 2011</p>
    </body>
</html>
```

My First page

Let us look at the page´s HTML code in the figure above. As you can see, the <!DOCTYPE html> <html> , <head> and <body> tags represent the foundation of an HTML page. The way these tags are structured is shown below.

```
<!DOCTYPE html>
<html>
    <head>
     ……
    </head>

    <body>
     ……
    </body>
</html>
```

HTML5 elements are marked up using start tags and end tags. Tags are delimited using angle brackets with the tag name in between. The difference between start tags and end tags is that the latter includes a slash before the tag name.

Let us explore this structure one tag at time.

- <!DOCTYPE html>. This statement states that the document type is HTML 5. It should be placed at the top of the HTML page.
- <html>. This element represents the root element of your html page. Notice that, as it is true for most html elements, this element has an opening <html> and a closing tag </html>. This <html> element contains 2 important elements: the <head> and the <body> elements:
    - <head>. As we will see later on, this element contains metadata about the document like the title, styles, scripts and other meta information. Meta tags should also go in the <head> tag. The code below is an example of a <head> element content.

```
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>RazorPagesEventMaker</title>
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.min.css" />
    <link rel="stylesheet" href="~/css/site.css" />
</head>
```

    - <body> This e

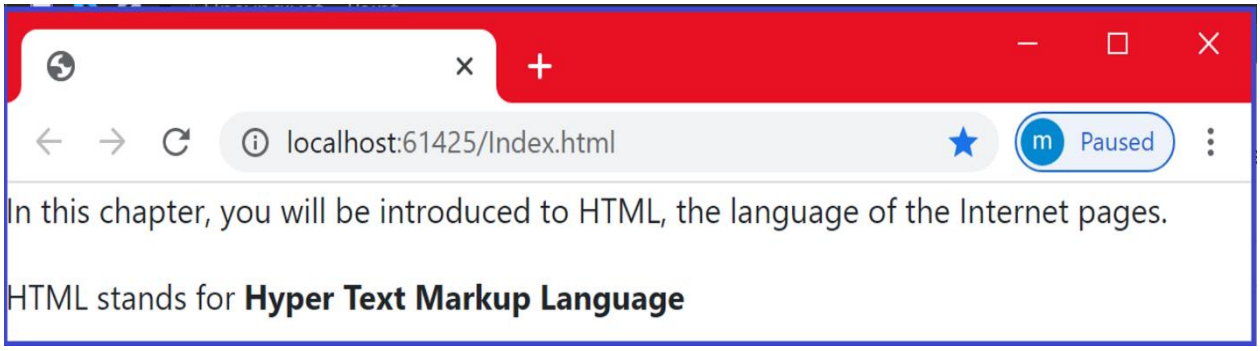
```
<body>
    <div>
        <p>
            In this chapter, you will be introduced to HTML, the language
            of the Internet pages
        </p>.
        HTML stands for <strong> Hyper Text Markup Language</strong>
    </div>
</body>
```

lement defines the document body, that is, the visible part of the page. The code snippet below shows an example of a very simple body content.

At the moment, do not worry if you do not understand what is a   or a element. We will cover most of the HTML elements in details in the next sections. Now, you need to understand the element as a container and the element a way to transform the text inside the element as bold.

Note that html elements are not case sensitive and some have no content (empty elements)

Thanks to the browser (the engine that interprets the html code), the content of the page is read and rendered in very informative laid-out pages that the user can understand. The rendering on the browser of the previous html code is shown in the figure below.



As can be seen, browsers do not display the html tags, but uses them to determine how to display the document. HTML5 has a broad browser support. It is compatible with the latest version of many browsers such as Apple safari, Google Chrome, Mozilla Firefox, Internet Explorer …etc.

HTML elements and attributes

HTML elements represent the components of an html page. An html elements may contain attributes that help add information and properties about the element. We can define global attributes that can be applied to any element. We can also define attributes that can be applied to specific elements.

They are generally expressed as name-value pairs. The following example illustrates how to mark up an image element with an attribute named src using a value of "image1.jpg".

< img src="image1.jpg">

Basic elements and attributes

<**p**>: The element defines a paragraph </**p**>

<**div**> </**div**> : the div element general purpose is to play the role of a container.

It helps set the different parts of a page and achieve good page layout.

<**br**>:The
 element defines a line break (a new line)

Comment : **<!-- This is a comment -->**

Note: Comments are very useful for debugging HTML (search for errors)

<**h1**>This element defines a heading, it is generally a title or subtitle</h1>

Note : We can apply a range of headings from **h1** to **h6** tags in descending order of importance, where **h1** defines the most important heading.

<**p style** ="color: red"> </**p**>

The style attribute is used to style the element . In the example above, we make this paragraph red.</**p**>

Note: You can also use the style attribute to define background color, font family, font size, text alignment…etc. In chapter 2, we will see how to define sophisticated style that can be applied to the whole page or specific element.

The example below is an example of applying the font family with the value "Calibri" to a paragraph.

<p **style**="font-family:Calibri;">**This is a paragraph. **

HTML Links

<a> element: The <a> HTML element , called anchor element, defines an hyperlink. It has many attributes:

href, target attributes

Consider the following hyperlink element

```
<a href="https://www.Zealand.com/corona/"  target="_blank"> About Corona</a>
```

href=https://www.Zealand.com/corona/. This part defines the address (URL) of the linked page.

target="_blank" This part indicates where to open the target page. We want to open the "Zealand.com/corona" page in a new page. You can also use the following values for the target attribute: _self , _parent and _top.

The text "About Corona" represents the text of the link. Instead of a text, you can also use an image.

HTML images element

<img> element: The <img> HTML element inserts an image into your page . You can specify the location of the image, its dimension (width, height) , alternative text to be displayed if the image cannot be found. Let us look at some of its attributes.

src , alt , style attributes

Consider the following HTML image element

```
<img src="image1.jpg" alt="Tour Eiffel" style="width:42px;height:42px;>
```

- <img> element defines an image .
- src="image1.jpg" The src attribute specifies the path of the image.
- alt="Tour Eiffel" In case the image cannot be displayed, you can specify an alternative text to be displayed using the alt attribute.

- style="width:42px; height:42px;> The style attribute is used to define the width and the height of the image.

HTML Form

HTML forms are mainly used to collect user inputs. A very simple example is a user registration to create accounts. A form is used to collect info such as name, email address, username, password …etc. Once we click on the "Submit" button, the entered data is sent to the back-end application for processing. The form may encapsulate controls such as text fields, text area, dropdown menus , radio buttons, checkboxes, buttons…etc. The code below shows the structure of a <form> element.

```
20 <form name="form1" action="Script URL" method="">
21     <!-- here go the form controls -->
22 </form>
```

The <form> element defines the template of the form.

The name attribute is used to identify the form.

The action attribute specifies the event handler that will handle the submitted data.

The method attribute defines the type of the http method used to handle the form . For form submission, the POST method is commonly used. We can also use the **Get** method. However, the main difference between these two methods is that, with "get" method, the data is displayed in the page´s address field. So if you want to send sensitive data, it is recommended to use the "**post**" method.

Let us look at the different elements that could be incorporated in a form.

<label> element

Let us consider the following html code.

```
<form action="/noaction" method="post">
    <label for="fname">FirstName:</label><br>
    <input type="text" id="firstname" name="fname" value="John"><br>

    <label for="lname">LastName:</label><br>
    <input type="text" id="lastname" name="lname" value="Doe"><br><br>
    <input type="submit" value="Submit">
</form>
```

The <label> elements are used to add a caption or a label to each input data you are collecting. Labels help the user enter the right data. The for attribute is used to specify the property for which you want to assign the input data.

<Input> element

We continue working with the code shown above.

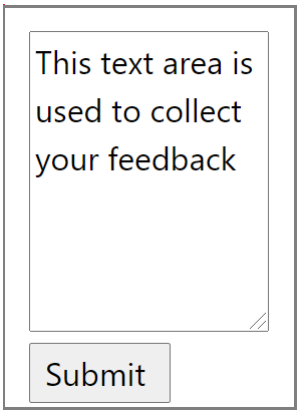
<input type="Text"> elements are used to catch the data entered by the user. The data can be text. In this case we use an <input> element whose type attribute has "Text" value.

<input type="submit"> defines a button that is susceptible of firing events. The value attribute defines the text displayed on the button. By clicking on this button, the form data will be sent to back-end.

TextArea

Sometimes the user is willing to enter many lines of text. The <textarea> element is used for this purpose. In the following example, we defined a text area which is 14 columns wide and 6 rows high.

```
<body>
      <form method="post" style="margin-left:12px; margin-top:12px;">
           <textarea cols="14" rows="6"></textarea>
             <br />
           <input type="submit" value="Submit " />
      </form>
</body>
```



Notice the use of the style attribute to set the left and the top margins. We used the "cols" attribute to define the width and the "rows" attribute to define the hight of the text area.

Radio buttons

Radio buttons allow you to set up a list of options, from which the user can pick just one. In this scenario, radio buttons should be used to force you to select only one option. The following example shows the use of radio buttons.

```
<form method="post" style="margin-left:12px; margin-top:12px;">
    <legend>What is Your Favorite Movie?</legend>
       <input type="radio" name="favorite_movie" value="Star Wars" checked>Star wars<br>
       <input type="radio" name="favorite_movie" value="Fast & Furious">Fast & Furious<br>
       <input type="radio" name="favorite_movie" value="Bad Boys">Bad Boys<br>
       <br>
       <input type="submit" value="Submit ">
 </form>
```

Notice *that we define a name for the radio button. We need to identify the control when posting the form. This helps know what has been selected by the user.* ***Notice*** *also that we pre-checked the first option using the "**checked**" attribute. This will force the user to select one of the available options.*

Note that you should use labels to tie your radio button and the descriptive text together, to allow the user to click a larger area when manipulating the radio button. We cover this aspect in the next section.

Radio buttons and labels

In the previous example, you have to click on the radio button itself. This may difficult if you are using a small device (smartphone). This can be solved using labels to tie the radio button to the attached text, which allows you click on the text as well.

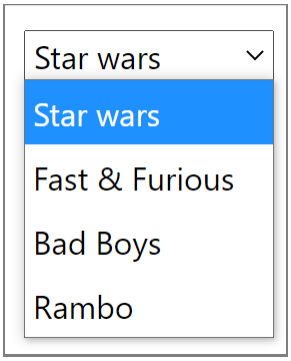
```
<form method="post" style="margin-left:12px; margin-top:12px;">
      <legend>What is Your Favorite Movie?</legend>
           <input type="radio" name="favorite_movie" value="Star Wars" id="wars" >
           <label for="wars">Star wars</label></br>

         <input type="radio" name="favorite_movie" value="Fast & Furious"   id="fast" >
         <label for="fast">Fast & Furious</label></br>

         <input type="radio" name="favorite_movie" value="Bad Boys"   id="boys" >
         <label for="boys">Bad Boys</label><br>

         <input type="submit" value="Submit ">
</form>
```

Notice the use of the "**id**" attribute in the input element and the use of the "**for**" attribute in the label element to assign the label to the corresponding input element.

Dropdown List

Another way to allow the user pick only one option is the use of dropdown list. There is a small difference between using radio buttons and dropdown list. The dropdown list forces the user to select an option , whereas the radio button does not force the user to pick one. **This is commonly used when ????**

```
<form method="post" style="margin-left:12px; margin-top:12px;">
                    <select name="Movies">
                        <option value="Star wars">Star wars</option>
                        <option value="Fast & Furious">Fast & Furious</option>
                        <option value="Bad Boys">Bad Boys</option>
                        <option value="Rambo">Rambo</option>
                    </select>
</form>
```



*Notice the use of the <**select**> element to create the dropdown-list.*

*The <**option**> element defines each option.*

By default, the first option in the list is selected.

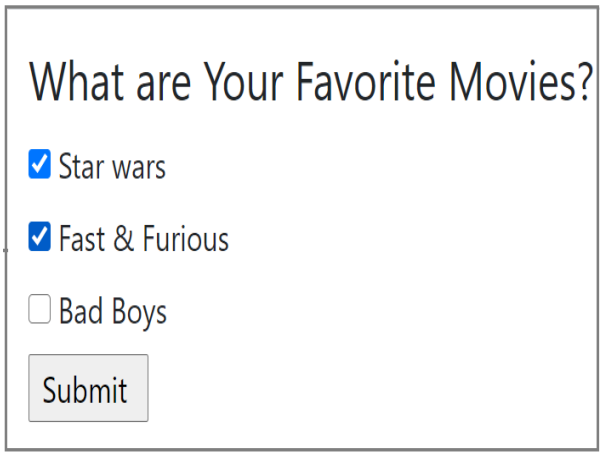

```
<select name="Movies">
   <option value="Star wars">Star wars</option>
   <option value="Fast & Furious">Fast & Furious</option>
   <option value="Bad Boys" selected="selected"> Bad Boys
</option>
   <option value="Rambo">Rambo</option>

</select>
```

***Note** that If you want to select a specific option , use the "**selected**" attribute*

Checkboxes

As opposite to the radio button, which allows you to pick only one option from a list of options, checkboxes allow you choose several options from a group of options. Let us consider the previous example that used radio buttons. Instead of radio buttons, let us use checkboxes .

```
<form method="post" style="margin-left:12px; margin-top:12px;">
         <legend>What are Your Favorite Movies?</legend>
            <input type="checkbox" name="favorite_movie" value="Star Wars" id="wars">
            <label for="wars">Star wars</label></br>

            <input type="checkbox" name="favorite_movie" value="Fast & Furious" id="fast">
            <label for="fast">Fast & Furious</label></br>

            <input type="checkbox" name="favorite_movie" value="Bad Boys" id="boys">
            <label for="boys">Bad Boys</label><br>

            <input type="submit" value="Submit ">
```



Notice *that the only thing I did change is the type of the input element from radio to chakcbox.*

Also Notice *as with all input elements, you need to define a name to be able to identify the checked elements when posting the form.*

Notice*, as done with radio buttons before, the use of labels to attach the checkbox to text, which allows a vast area of selection.*

Submit & Reset Buttons

You might notice that in the previous form examples , we have been using the "submit" button to submit the form. The "**submit**" button is also an input element whose type is "submit".

```
<input type="submit" value="Submit ">
```

Another button that we did not talk about is the "**reset**" button. The "Reset" button is also an input element whose type is "reset " and it is used to clear all inputs.

```
<input type="reset" value="Reset" />
```

Adding the Reset button to the previous example is illustrated in the figure below.

HTML Tables

HTML tables are used to display data into rows and columns. You create a table using the <**table**> element. This is the first thing you start with. Then you need to create the table header using the <**th**> element. Table rows are defined using the <**tr**> (stands for table row) tag , while the <**td**> (stands for table data) tag defines a table cell. The following table contains a header, three rows and four columns.

```
<table   <table style="margin-left:12px; margin-top:12px ; border: 2px solid black"
                                                                    width="75%">

            <tr style="background-color:azure">

                <th>Id</th>
                <th>Name</th>
                <th>Place</th>
                <th>Date</th>
            </tr>
            <tr>
                <td>1</td>
                <td>Marathon</td>
                <td>Copenhagen</td>
                <td>13-06-2020</td>
            </tr>
            <tr>
                <td>2</td>
                <td>Distortion</td>
                <td>Copenhagen</td>
                <td>03-08-2020</td>
            </tr>
            <tr>
                <td>3</td>
                <td>Roskilde Festival </td>
                <td>Roskilde</td>
                <td>01-09-2020</td>
            </tr>

</table>
```

- Notice that we have also added of a border of the table using the border attribute with the value `2px solid black"`
- We have also applied a background color to the first row only using the style attribute with the value `background-color:azure`.

The result is shown below.

Id	Name	Place	Date
1	Marathon	Copenhagen	13-06-2020
2	Distortion	Copenhagen	03-08-2020
3	Roskilde Festival	Roskilde	01-09-2020

You probably want to show data in a kind of a Grid with borders between rows and between columns. You can add borders for each row and for each column using the style attribute. This is probably going to be a cumbersome process. A very practical way is to define the style in the header and apply it to any element. So let us define a border that should be applied to the <**table**>, <**tr**> and <**td**> elements. You just need to the following style code in the header section

```
</head>
      <style>
       table, th, td {
                       border: 1px solid black;
                      }
      </style>
 </head>
```

The style is applied to the table, header and each

a 1px- black solid line border

```
<body>
   <table style="margin-left:12px; margin-top:12px";  width="75%">
         <tr style="background-color:azure ; ">
           <th>Id</th>
           <th>Name</th>
           <th>Place</th>
           <th>Date</th>
          </tr>
             <!-- the rest of the code -- >
```

Id	Name	Place	Date
1	Marathon	Copenhagen	13-06-2020
2	Distortion	Copenhagen	03-08-2020
3	Roskilde Festival	Roskilde	01-09-2020

Notice that we removed the border styling from the <table> element and we applied to the <table>, <th> and<td> elements.

Looking at the table above, you may notice that the text inside each cell is left aligned by default and you may want to add some padding (a padding is a space between the text and the border) or you want to center-align the text. As we did previously, we can easily define a style that does the job , apply it to the **concerned** elements and place it in the header.

The code below applies the padding style 6px 4px 4px 8px to the <th> and <td> elements. This padding corresponds respectively to the top, right, bottom and left padding.

```
th , td{
        padding: 6px 4px 4px 8px ;

       }
```

Let us incorporate this style into the previous one. The code below shows the final style. Notice that instead of applying the same border property with value " 1px solid black" to the <table>, <th> and <td>,we applied a border property with value " 3px solid black" to table and the header element while a "1px solid black" border is applied to the <td> element.

```
       <style>
                td {
                    border: 1px solid black;
                }

                table, th{
                    border: 3px solid black;
                }

                th, td {
                    padding: 6px 4px 4px 8px ;
                }
```

```
        </style>
```

Id	Name	Place	Date
1	Marathon	Copenhagen	13-06-2020
2	Distortion	Copenhagen	03-08-2020
3	Roskilde Festival	Roskilde	01-09-2020

Some words before closing this chapter : You might notice that we already used the <style> element to add styling to our page. The question is: what if I want to apply the same style to many pages ? Should I add it to every page ? won´t be tedious ? The next chapter is going to answer these questions. We are going to cover CSS (Cascading Style Sheet). It is going to be relatively short, so let us move to the next chapter.

Chapter 2: Cascading Style Sheet

Introduction

In the previous section about HTML, we could not avoid talking about CSS. Indeed, we were styling our page using either the style attribute to apply a style to a specific element or by suing the <style> element in the header to be able to apply it to many html elements in the page. We could assign a specific color to the background of a page, we could define align a text, we could assign color to text, we could set the font of a text and we could add many other styles. Adding styles to page elements this way, will be a nightmare if we are developing a large website and this is where CSS can come to our rescue . Thus, CSS allows control the layout of multiple web pages across the web application.

In this tutorial, we are not using CSS so much, may be not at all. We are mainly using Bootstrap , which we will cover in the next chapter. Bootstrap is the most popular CSS framework for developing responsive web application. So why this chapter? I think it is very important to have an overview about CSS to understand what is going on behind the scene when working with Bootstrap. You may also want to define your own CSS. This chapter introduces the basics about CSS. We focus most on forms and tables. However to get a deep understanding of CSS, I recommend you to consult the documentation.

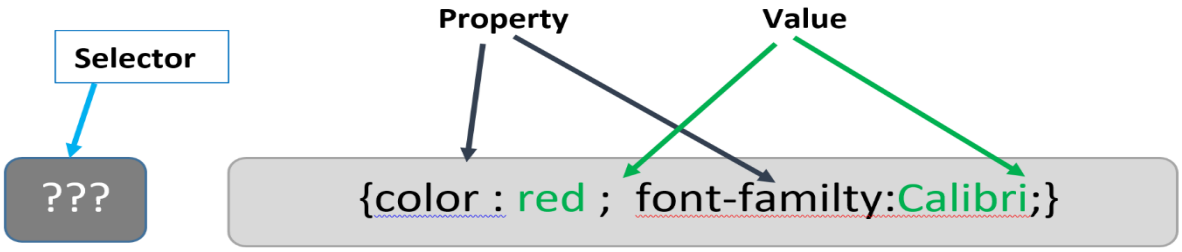
What is CSS ?

CSS stands for Cascading Style Sheets. The main key advantage of using CSS is the ability for a developer/designer to control the presentation of an HTML document. That is, to be able to control its look: text color, font, background color, images, layout design and many other effects.

Let us start with an example that we defined earlier in the previous chapter

```
<p  style="color: red; font-family:Calibri"> This is a red paragraph with the Calibri font family</p>.
```

In this example the style attribute is used to define the color and the font family of this paragraph <p> element. The example is illustrated below.



As shown above, the CSS syntax consists of **a selector** (which describe how to select an html element) and a block consisting of series of style declarations separated by a semicolon.

How to specify the selector ?

There are many ways to specify the selector. The selector part indicates how we select the element to which the CSS is applied. This forces us to uniquely identify the elements in our app, at least within the same page. As we will see in a moment, the element can be identified either using the **id** attribute or the **class** attribute.

Let us consider the following example again and see how we can apply it to html elements.

```
<p  style="color: red; font-family:Calibri"> This is a red paragraph with the Calibri font family</p>.
```

id selector

Instead of applying the red color and the Calibri font-family to only this specific paragraph, we can create the corresponding CSS as shown below. Notice the use of the id selector **#p2** . The # sign means that we are going to use the id attribute to select the element.

```
#p2 {color: red; font-family:Calibri";}
```

Then, to apply the CSS to **ANY** <p> element, we have just to specify the selector as the value of the id attribute as shown below.

```
<p  id="p2"> This is a red paragraph with the Calibri font family</p>.
```

Within the same page, the id attribute of an element must be unique. The id selector uses this unique id to select a specific element in a page.

Example

```
<style>
    #login {
        background-color: blue;
        color: white;
        padding: 20px;
        text-align: right;
    }
</style>

<h1 id="login">Subject</h1>

<h2 id="login">Class</h2>
```

In the example above, we defined an id selecor named "**login**". Then, we apply this style to the heading elements <h1> and <h2> with the "**login**" id .

class selector

We can also select a specific element by using its class attribute. We follow the same procedure as for the id selector. Let us consider the following example.

 This is a red paragraph with the Calibri font family .

This time, to apply the same style as before, we need to define a style sheet as follows:

.cofont {color: red; font-family:Calibri";}

Notice the use of the class selector .cofont. The **.** (dot) sign means that we are going to use the class attribute to select the element.

Then, to apply this CSS to **ANY** <p> element, we have just to specify the selector as the value of the class attribute as shown below.

```
<p  class="cofont"> This is a red paragraph with the Calibri font family</p>.
```

Example

```
<style>
.subject {
  background-color: orange;
  color: black;
  padding: 20px;
}
</style>

<h2 class="subject">Programming</h2>
<p>A lot of coding.</p>

<h2 class="subject">Design</h2>
<p>A lot of design.</p>

<h2 class="subject">PHP</h2>
<p>A lot of scripts.</p>
```

In this example, we defined a class named "**subject**" in the style attribute. Then, we apply this style to the three headers h2. The output is shown below.

Programming

A lot of coding.

Design

A lot of design.

PHP

A lot of scripts.

Combining class selectors and/or id attributes

We can also specify a style sheet by combining class selectors and/or id selectors. In this example, we apply the background color and the bottom border to **any element** having either the class attribute “main” or the class attribute “top-row”.

```
.main .top-row {
 background-color: #e6e6e6;
 border-bottom: 1px solid #d6d5d5;}
```

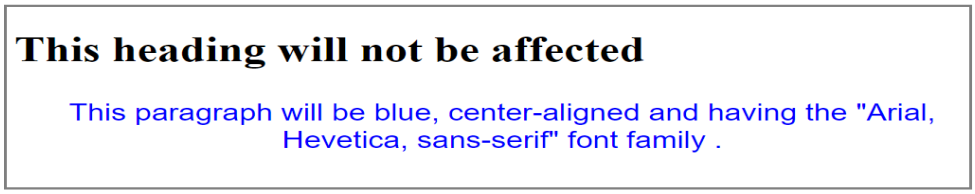
Style sheet on an element

We can also specify a style sheet by a class / id attributes **on an element.**

Example 1 : In this example, we apply the CSS style to any <p> element whose class attribute has the value “center”.

```
<!DOCTYPE html>
<html>
<head>
    <style>
        p.center {
            text-align: center;
            color: blue;
            font-family: Arial, Helvetica, sans-serif;
            }
    </style>
</head>
<body>
    <h2 class="center">This heading will not be affected</h1>
    <p class="center">This paragraph will be blue, center-aligned and having the
        "Arial, Hevetica, sans-serif" font family .</p>
</body>
</html>
```

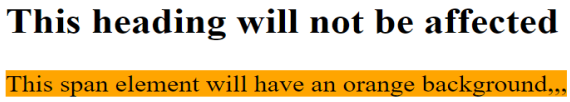
Notice how it is not applied to the heading<h2> even if the element has “center”as the class value. This is because it is not a <p> element



Example 2 : In this example, we apply the “orange” background color to a element having an id attribute of value “b”.

```
<!DOCTYPE html>
<html>
<head>
    <style>
        span#b {
            background-color: orange;
        }
    </style>
</head>
<body>
    <h2 id="b"> This heading will not be affected</h1>
    <span id="b">This span element will have an orange background</span>
</body>
</html>
```

Notice that the heading is not affected even if its id has a value “b”.

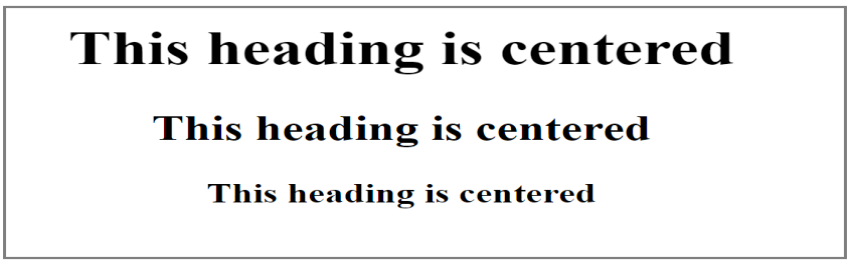


Grouping selectors

In case some, elements have the same style definition. In this case, it is better to group selectors in one line of code as follows:

```
<!DOCTYPE html>
<html>
<head>
    <style>
        h1, h2, h3 {
            text-align: center;
        }
    </style>
</head>
<body>
    <h1> This heading  is centered</h1>
    <h2> This heading  is centered</h2>
    <h3> This heading  is centered</h3>
</body>
```

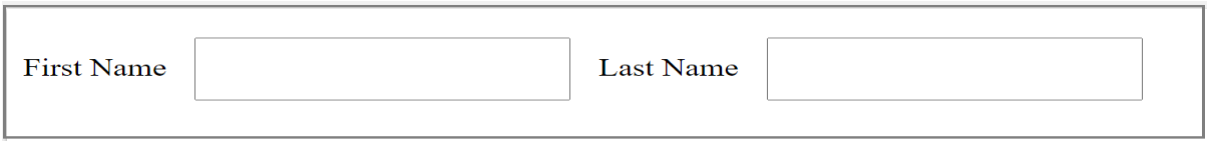
In this example, all h1, h2 and h3 will have their text center aligned, as shown below



CSS Form

In a form, there may be for example, many <input> elements. You can specify a style sheet to a specific <input> element by specifying its text type as shown below

```
<!DOCTYPE html>
<html>
<head>
    <style>
        Input[type=text] {
            Width: 25%;
            Padding: 10px 15px;
            Margin: 10px;
        }
    </style>
</head>
<body>
    <form>
        <label for="fname">First Name</label>
        <input type="text" id="fname" name="fname">
        <label for="lname">Last Name</label>
        <input type="text" id="lname" name="lname">
    </form>
</body>
</html>
```



Creating a CSS file

You can also define a style sheet in a separate external file and import it into your web pages. We define the external style by setting all the CSS Rules. The example below shows a simple example of creating and using a CSS file in a page. The CSS file CSSExample.css is not tied to any particular page and the great advantage of it is that it can be reused by many pages.

CSSExample.css

```
p{
     color:red;
     font-size:x-large;
     font-family:Verdana;
     background-color:gray;
 }

 h1{
     color:blue;
     background-color:yellow;
     font-familymily:Comic Sans MS;
 }

 .buttonControl{
     background-color:red;
     color:yellow;
 }
```

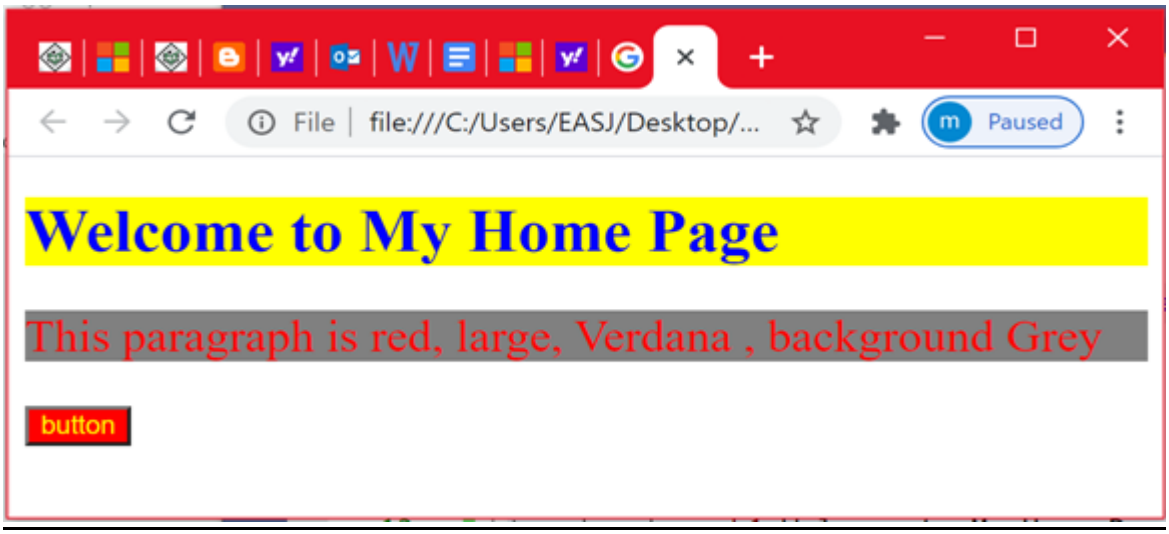
These CSS rules are applied to the following html page

HomePaae.html

```
<!DOCTYPE html>
<html>
<head>
    <head>
        <meta charset="utf-8" />
        <title>Home Page</title>
        <link rel="stylesheet" type="text/css" href="CSSExample.css" />
    </head>
</head>
<body>
    <form id="form1">
        <div>
            <h1>Welcome to My Home Page</h1>
            <p>
                This paragraph is red, large, Verdana , background Grey
            </p>
            <button class="buttonControl" id="button1">button</button>
        </div>
    </form>
 </body>
</html>
```

The output is shown below. As expected, the buttonControl class rules (`background-color:red;color:yellow;`)are applied to the button . The p rules are applied to the paragraph <p> element and the h1 rules are applied to the heading <h1> element.

So, you can easily create a CSS file, define numerous rules, and in the page add a link to the file. That's it.



Chapter 3 : Bootstrap

In this chapter, you will be introduced to bootstrap and you will learn the required basics to build a responsive website. In this chapter, you are going to dig into the most important HTML/ CSS and Helpers Bootstrap components. The goal is to give you a solid start on building responsive web sites. Even Though I am not covering the Bootstrap JavaScript component, the chapter contains the required HTML and CSS to build a responsive web site. For the moment , let us look at what is bootstrap and why you should use bootstrap.

Introduction to Bootstrap

Bootstrap is an open source front-end framework that helps web developers build easy and quick websites. It includes HTML and CSS based design templates for common user interface components like Buttons, Dropdowns, Typography, Tabs, Forms, Tables, Navigations, Alerts, Modals, Accordion, Carousel etc. along with optional JavaScript extensions. It is the most popular HTML, CSS and JavaScript framework to build complex and responsive mobile websites with basic knowledge of HTML, CSS and eventually some knowledge of JavaScript. As it is based on HTML, CSS and JavaScript, it can be used with any server side technology such as ASP.NET, JAVA, PHP etc.

As mentioned earlier, Bootstrap helps you build a responsive web application faster and easier. Responsive means that page elements (text , images etc.) adjust themselves to the screen size they are viewed on (phone, tablets and desktop) as the page grows or shrinks (when we resize the browser window). So you don't have to worry about your application not being compatible with multiple user

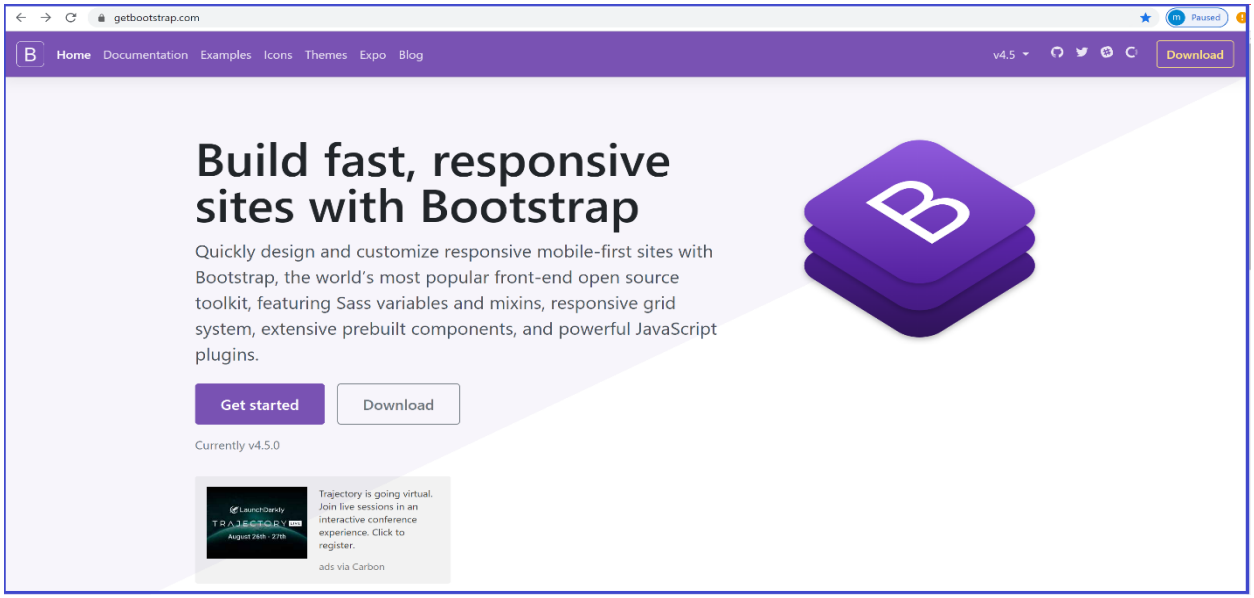
devices. Another advantage of using Bootstrap is the reduction of the development time. Indeed, instead of writing CSS from scratch, ready-made code is provided to you. You can even customize this code to fit your requirement needs.

Setup

I am using Bootstrap 4.5, which is compatible with all modern browsers. However it only support Internet Explorer 10 and above.

Using Bootstrap, you can either download Bootstrap from https://getbootstrap.com/ and host it on your computer **or** include it from a Content Delivery Network (CDN). The main drawback to using the second option is that you cannot work offline.

I recommend you to visit the Bootstrap web site https://getbootstrap.com/. It is a very good starting source full of examples, documentation, themes, icons , inspirations and all what you need to start building a quick , easy and responsive web site. .



To work with Bootstrap, you can use many HTML editor. Examples of the most popular editors are Atom, Notepad++ etc. I am using visual studio 2017.

Let us suppose that you are all the time online and you opt for the second option that requires you to include bootstrap from a Content Delivery Network. In this case, you need to add the CDN access path in the <**head**> element before all other stylesheets of your HTML page. As mentioned before, we will only cover the most important HTML /CSS bootstrap components and all what we need is to include the following code in the <**head**>element.

```
<!-- CSS only -->
<link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.5.0/css/bootstrap.min.css"
integrity="sha384-9aIt2nRpC12Uk9gS9baDl411NQApFmC26EwAOH8WgZl5MYYxFfc+NcPb1dKGj7Sk"
crossorigin="anonymous">
```

This piece of code includes Bootstrap´s compiled CSS in your page. However, if your components require JavaScript, you need to add the following <**script**> elements right before the closing<**/body**> tag of your page. Even though , these <script> elements are not mandatory in the context of this chapter, it is not a bad idea to include them anyway.

```
<!-- JS, Popper.js, and jQuery -->
<script src="https://code.jquery.com/jquery-3.5.1.slim.min.js" integrity="sha384-
DfXdz2htPH0lsSSs5nCTpuj/zy4C+OGpamoFVy38MVBnE+IbbVYUew+OrCXaRkfj" crossorigin="anonymous"></script>

<script src="https://cdn.jsdelivr.net/npm/popper.js@1.16.0/dist/umd/popper.min.js" integrity="sha384-
Q6E9RHvbIyZFJoft+2mJbHaEWldlvI9IOYy5n3zV9zzTtmI3UksdQRVvoxMfooAo" crossorigin="anonymous"></script>

<script src="https://stackpath.bootstrapcdn.com/bootstrap/4.5.0/js/bootstrap.min.js" integrity="sha384-
OgVRvuATP1z7JjHLkuOU7Xw704+h835Lr+6QL9UvYjZE3Ipu6Tp75j7Bh/kR0JKI" crossorigin="anonymous"></script>
```

Putting all the pieces together, your page should look like the figure below and you are ready to write some Bootstrap code.

```
<!doctype html>
        <html lang="en">
        <head>
```

Bootstrap requires the use of the HTML 5 DOCTYPE

```
        <!-- Required meta tags -->
        <meta charset="utf-8">

        <meta name="viewport"
          content="width=device-width, initial-scale=1, shrink-to-fit=no">


           <!-- Bootstrap CSS -->
            <link rel="stylesheet"
            href="https://stackpath.bootstrapcdn.com/bootstrap/4.5.0/css/bootstrap.min.css"
            integrity="sha384-9aIt2nRpC12Uk9gS9baDl411NQApFmC26EwAOH8WgZl5MYYxFfc+NcPb1dKGj7Sk"
            crossorigin="anonymous">

       <title>Learn Bootstrap !</title>
       </head>

       <body>


          <!-- Optional JavaScript -->
          <!-- jQuery first, then Popper.js, then Bootstrap JS -->
          <script src="https://code.jquery.com/jquery-3.5.1.slim.min.js"
           integrity="sha384-DfXdz2htPH0lsSSs5nCTpuj/zy4C+OGpamoFVy38MVBnE+IbbVYUew+OrCXaRkfj"
           crossorigin="anonymous"></script>

           <script src="https://cdn.jsdelivr.net/npm/popper.js@1.16.0/dist/umd/popper.min.js"
           integrity="sha384-Q6E9RHvbIyZFJoft+2mJbHaEWldlvI9IOYy5n3zV9zzTtmI3UksdQRVvoxMfooAo"
           crossorigin="anonymous"></script>

           <script src="https://stackpath.bootstrapcdn.com/bootstrap/4.5.0/js/bootstrap.min.js"
           integrity="sha384-OgVRvuATP1z7JjHLkuOU7Xw704+h835Lr+6QL9UvYjZE3Ipu6Tp75j7Bh/kR0JKI"
           crossorigin="anonymous"></script>

       </body>
</html>
```

To ensure proper rendering and touch zooming for all devices

To include Bootstrap CSS libraries

If you need Bootstrap Jquery and JavaScript libraries

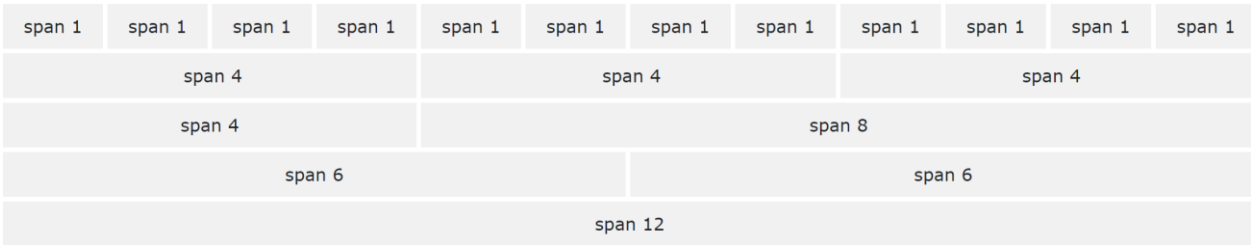
Note : If you opt for downloading Bootstrap and host it on your computer, make sure that the path to the Bootstrap file is correct.

HTML/CSS Bootstrap Components

In this chapter, you are going to learn how to understand the Grid system, tables, List Groups, Pagination …etc. You will also learn how to create Navigation bars (navbar), Form &input, buttons & links, Jumbotron, Dropdowns, Alerts progress Bars, Labels and Badges, responsive utilities … etc. By the end of this chapter, you will get a lot about Bootstrap to start building responsive web sites.

- ***GRID SYSTEM***

  The Grid system is used for creating page layouts. As shown below in ***Figure xx***, it consists of 12 columns. Before digging into building a Grid using Bootstrap, you need to understand the Grid System and the meaning of these numbers 1, 2, 4, 6, 8 and 12.



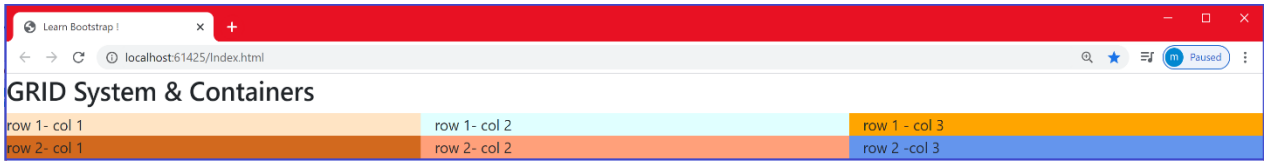
Suppose that you want to create a 4 equal column layout. As the width is the same for all columns, their width will be 3 units each because the sum should be equal to 12. However, if we want to create a 3 column layout (left side, Main content, right side) with ratio (1:4:1) this time, the Left side column is 2 units, the Main content column should be 8 units and the Right side column should be 2 units. This is because when adding the 3 numbers (2 Units + 8 Units + 2 Units), we should get 12 units.

In the code example below, we used the bootstrap class "**row**" to create row, then the class "**col**" is used to create each column inside this row. In this example, we created 2 rows having 3 columns each.

```
<body>
    <div>
        <h3>GRID  System & Containers </h3>
        <div class="row">
            <div class="col" style="background-color: bisque">row 1- col 1</div>
            <div class="col" style="background-color:lightcyan">row 1- col 2</div>
            <div class="col" style="background-color:orange">row 1 - col 3</div>
        </div>
```

```
        <div class="row">
            <div class="col" style="background-color:chocolate">row 2- col 1</div>
            <div class="col" style="background-color: lightsalmon">row 2- col 2</div>
            <div class="col" style="background-color:cornflowerblue">row 2 -col 3</div>
        </div>
    </div>
```

Rendering



As you can see, the whole screen is equally divided between the 3 columns. The page is not responsive, because as we resize the browser, the columns do not adjust and adapt to the screen size.

Another thing that you should understand is that bootstrap also provides several grid classes that you can use to create grid column layouts ranging from extra small devices like mobile phones to large devices like large desktop screens.

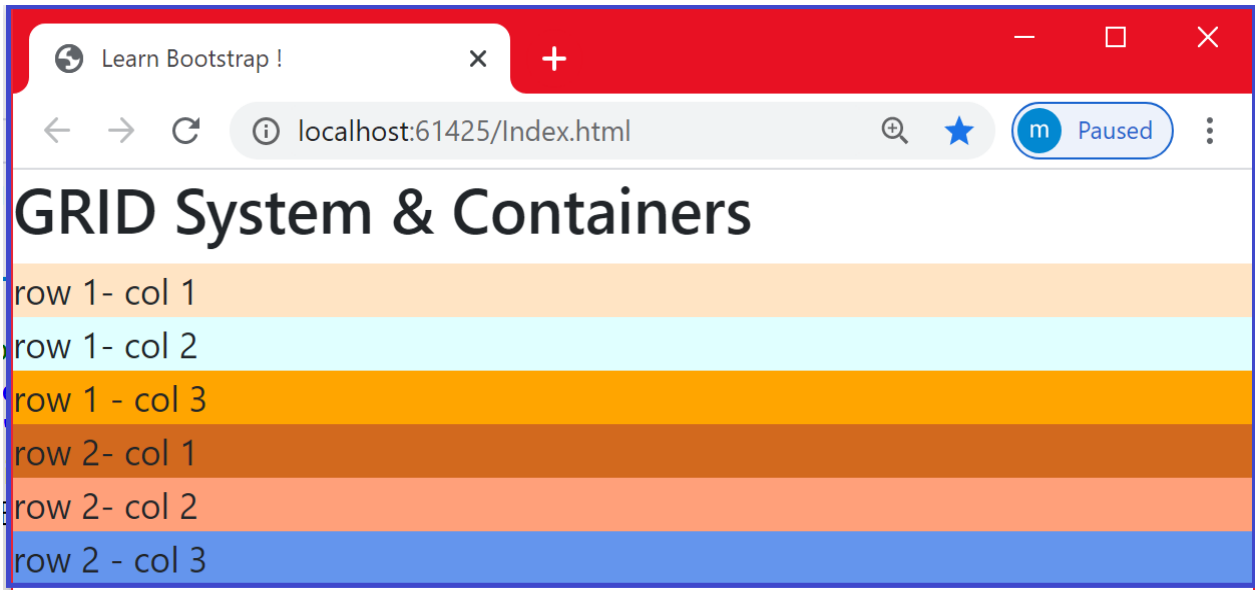
Device	Bootstrap Grid class
extra small devices - screen width less than 576px	.col-
small devices - screen width equal to or greater than 576px	.col-sm-*
medium devices - screen width equal to or greater than 768px	.col-md-*
large devices - screen width equal to or greater than 992px	.col-lg-*
xlarge devices - screen width equal to or greater than 1200px	.col-xl-*

Things get a little bit confusing at first, but I promise that you'll quickly get the hang of it through examples. We are going to continue working with the previous example and we want to make the page responsive.

As can be seen in the figure below, we used the class : class="col-sm-4". The **-sm** stands for small device (screen width equal or greater than 576px) and the number **4** means that the column spans over 4 columns of the 12-column Bootstrap Grid system.

```
<body>
```

```
    <div>
        <h3>GRID  System & Containers </h3>
        <div class="row">
            <div class="col-sm-4"  style="background-color: bisque">row 1- col 1</div>
            <div class="col-sm-4"  style="background-color:lightcyan">row 1- col 2</div>
            <div class="col-sm-4"  style="background-color:orange">row 1 - col 3</div>
        </div>

        <div class="row">
            <div class="col-sm-4" style="background-color:chocolate">row 2- col 1</div>
            <div class="col-sm-4" style="background-color: lightsalmon">row 2- col 2</div>
            <div class="col-sm-4" style="background-color:cornflowerblue">row 2 - col
3</div>
        </div>
    </div>
```



As can be seen, when the screen size goes below 576px, instead of being side by side, columns are automatically stacking on top of each other making the page responsive.

- ***Containers***

With the **.container** class, you can wrap the content into a container for proper padding and alignment.

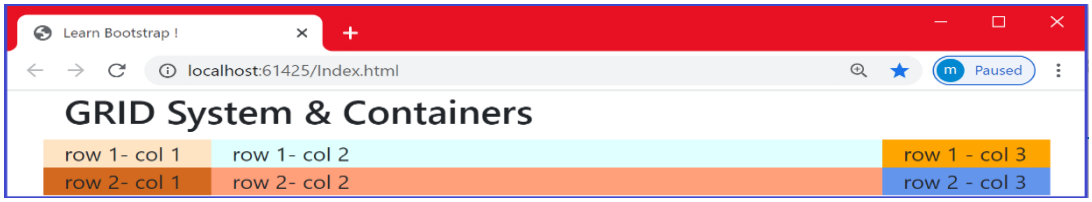
Let us continue with the previous example of Grid system. As you might notice, when displaying the content in the browser, the text "Grid System & Containers" and the rows are pushed against the left side of the browser. You usually want to have things toward the middle. To achieve that, you may think of using some CSS margin, padding …etc. However, with bootstrap, we can use the bootstrap class "**.container**" that applies a responsive **fixed width** container to the content. Containers are used for proper alignment and padding.

In the code example below, we used the "**.container**" class to the <**div**> element to make content pushed to the middle. Also, notice the 3 columns layout (2 units, 8 units, 2 units) that is used in this example.

```
<body>
      <div class="container">
        <h3> GRID  System & Containers </h3>
        <div class="row">
           <div class="col-md-2" style="background-color: bisque">row 1- col 1</div>
           <div class="col-md-8" style="background-color:lightcyan">row 1- col 2</div>
           <div class="col-md-2" style="background-color:orange">row 1 - col 3</div>
        </div>

        <div class="row">
        <div class="col-md-2" style="background-color:chocolate">row 2- col 1</div>
        <div class="col-md-8" style="background-color: lightsalmon">row 2- col 2</div>
        <div class="col-md-2" style="background-color:cornflowerblue">row 2 - col 3</div>
        </div>
      </div>
```

3 column layout having 2 , 8 and 2 units

Rendering

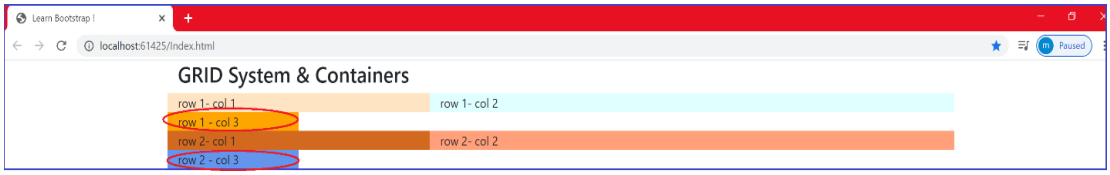


First , as expected , you might notice that the columns are not equal (the first column has 2 units, the second has 8 units and the last column has 2 units). You might also notice that the message and the grid get pushed to the middle.

Note that if you specify a Grid system whose sum of columns in each row is more than 12 columns as defined in the 12-column Grid system, the extra columns are wrapped into a new line.

Exercise : Try the following code without the container class.

```
<div class="row">
            <div class="col-md-4" style="background-color: bisque">row 1- col
1</div>
            <div class="col-md-8" style="background-color:lightcyan">row 1- col
2</div>
            <div class="col-md-2" style="background-color:orange">row 1 - col
3</div>
</div>
```



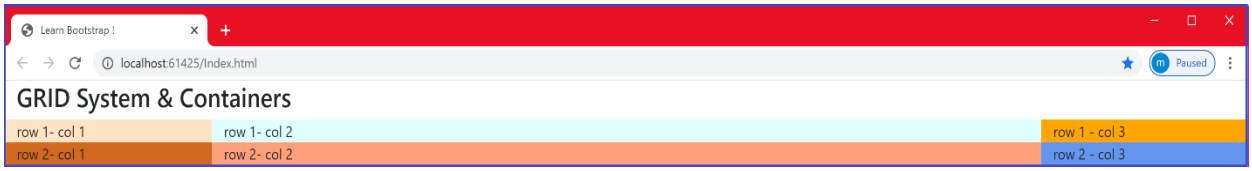
As can be seen, the 2 columns "**row1-col3** " and "**row2-col3**" are pushed into a new line.

- You can also define border, background color, text color of the container

```
<body>
      <div class="container  bg-dark text-white">
      <h3> GRID  System & Containers </h3>
```

- With the class "**container-fluid**" applied to the element, the container will stretch across the whole width of the viewport (screen) as shown the figures below.

```
<body>
        <div class="container-fluid">
          <h3> GRID  System & Containers </h3>
```



➢ ***Navbar***

Bootstrap navbar component is useful for creating responsive navigation header for a website. How to create a navigation bar? We apply the bootstrap ***.navbar*** class to a <nav> element.

In the code below, we created a vertical navigation bar having 3 links: Home, Contact and About.

```
<body>
    <nav class="navbar">

        <ul class="navbar-nav">

            <li class="nav-item">

                <a class="nav-link" href="#">Home</a>
```

*Apply the "**.navbar.nav**" class to the <ul> element*

*Add an item to the navigation by applying the "**.nav-item**" class to the <li> element*

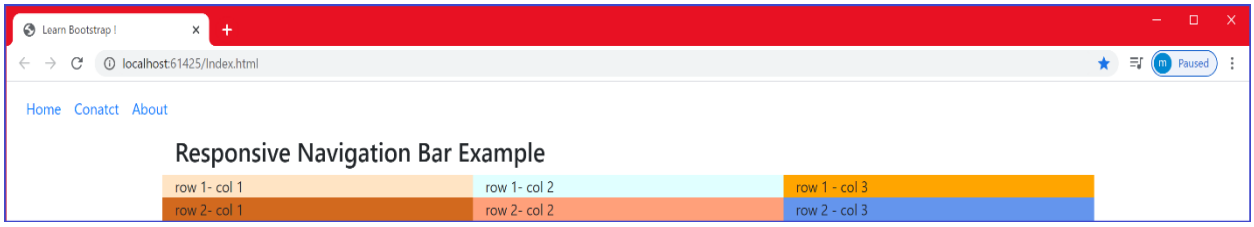
*Add a link and apply the "**.nav-link**" class to the <a> element*

```
            </li>
            <li class="nav-item">
                <a class="nav-link" href="#">Contact</a>
            </li>
            <li class="nav-item">
                <a class="nav-link" href="#">About</a>
            </li>
        </ul>
    </nav>
    <div  class="container">
        <h3>Navigation Bar Example </h3>
        <div class="row">
           <div class="col-sm-4"  style="background-color: bisque">row 1- col 1</div>
           <div class="col-sm-4"  style="background-color:lightcyan">row 1- col 2</div>
           <div class="col-sm-4"  style="background-color:orange">row 1 - col 3</div>
        </div>
          <!-- the rest of the code here →
```

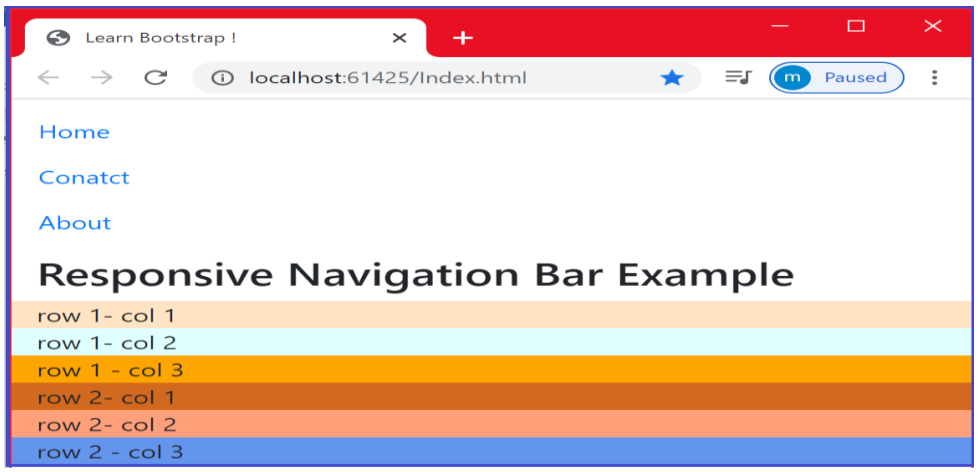
- Adding *the **navbar-expand-xl|lg|md|sm** class allows the navbar to be responsive and stack vertically depending on the screen size.*

```
<body>
      <nav class="navbar navbar-expand-sm">
            <ul class="navbar-nav">
                <li class="nav-item">
```

```
                <a class="nav-link" href="#">Home</a>
            </li>
            <li class="nav-item">
                <a class="nav-link" href="#">Contact</a>
            </li>
            <!-- the rest of the code here -->
```



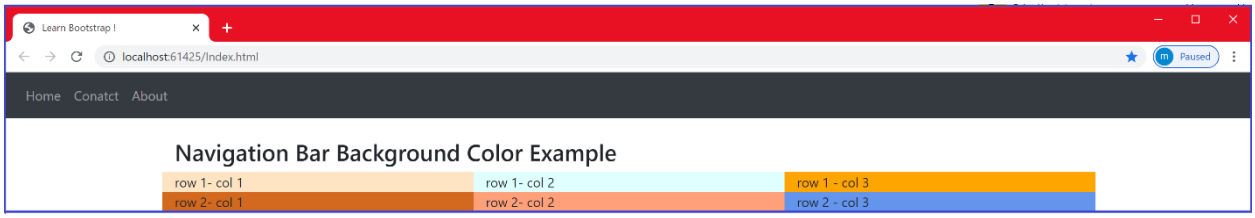
As can be seen, the navigation bar is responsive. When you resize the browser, if the screen size is less than 760px, the navigation bar stacks vertically as shown the Figure below.



- You can add the background color of the navbar using the .bg-color class.
- If the background is dark, you can make the text color white using the .navbar-dark class.

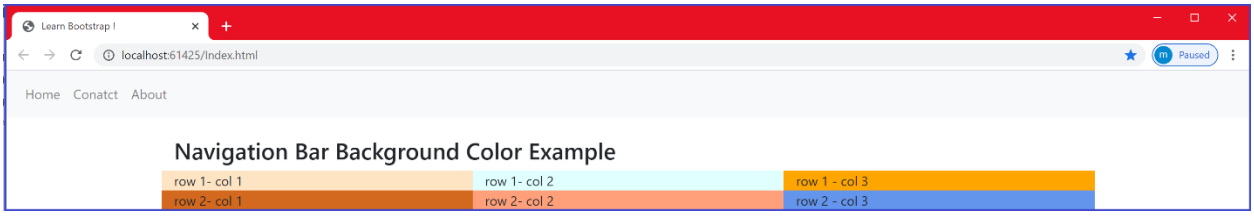
```
<body>
    <nav class="navbar navbar-expand-sm bg-dark navbar-dark">
```

```
            <ul class="navbar-nav">
                <li class="nav-item">
                    <a class="nav-link" href="#">Home</a>
                </li>
                <!-- the rest of the code here -->
```



- If the background is light, you can make the text color dark using the .navbar-light class.

```
<body>
      <nav class="navbar navbar-expand-sm bg-light  navbar-light">
            <ul class="navbar-nav">
                <li class="nav-item">
                    <a class="nav-link" href="#">Home</a>
                </li>
                <!-- the rest of the code here -->
```



Here are some bootstrap classes that can be used to setup the navbar background:

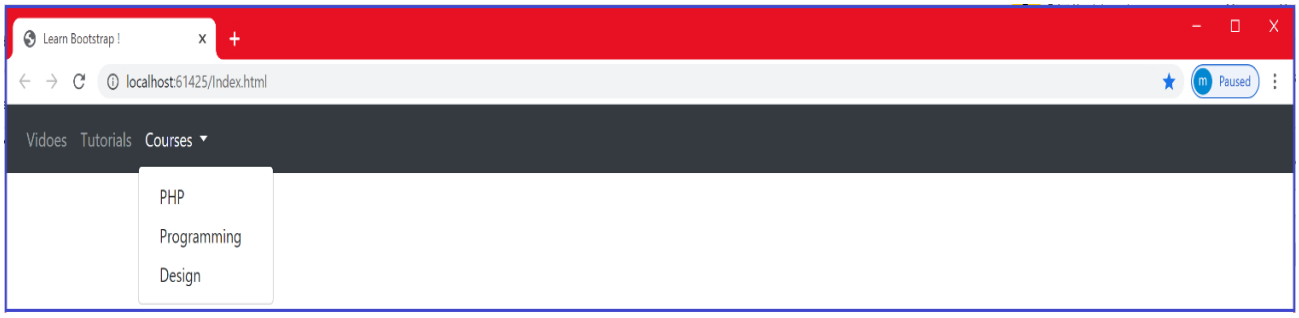
.bg-primary , -bg-success, .bg-warning, .bg-danger, .bg-secondary. .bg-dark and .bg-light

I recommend you to play with these classes.

- You can also make your navigation bar items into dropdown menus. In the code example below, the third item of the navigation bar is implemented as a Dropdown menu using the Bootstrap ".dropdown" class. We then apply the ".dropdown-toggle" class to the <a> element. Then we

apply the ".dropdown-menu" class to a element that wraps the dropdown menu and we apply the ".dropdown-item" class to each link in the dropdown menu.

```
<nav class="navbar navbar-expand-sm bg-dark navbar-dark">
    <!-- Links -->
    <ul class="navbar-nav">
       <li class="nav-item">
           <a class="nav-link" href="#">Vidoes</a>
       </li>
        <li class="nav-item">
             <a class="nav-link" href="#">Tutorials</a>
        </li>

       <!-- Dropdown -->
        <li class="nav-item dropdown">

             <a class="nav-link dropdown-toggle" href="#" id="navbardrop"  data-toggle="dropdown">
             Courses
             </a>

            <div class="dropdown-menu">
              <a class="dropdown-item" href="#">PHP</a>
              <a class="dropdown-item" href="#">Programming</a>
              <a class="dropdown-item" href="#">Design</a>
             </div>
        </li>
    </ul>

 </nav>
```



Buttons with *Bootstrap*

- *Basic Buttons*

You can create a basic button using the "**.btn**" Bootstrap class. However, if you want to customize your buttons to reflect its context, you need to apply what we call contextual classes.

Bootstrap provides several contextual classes that can be applied to tables, form , buttons and other components.

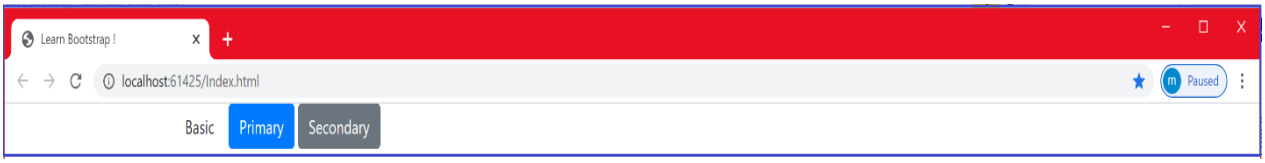
- .primary >> Indicates a ???
- .secondary >> Indicates a ???
- .Dark >> Indicates a ??
- .Light >> Indicates a ???
- .success >> Indicates a successful or positive action
- .info >> Indicates a neutral informative change or action
- .warning >> Indicates a warning that might need attention
- .danger >> Indicates a dangerous or potentially negative action

The figure below shows the effect of the different Bootstrap contextual classes applied to buttons

In the code below, we created a basic, primary and a secondary button.

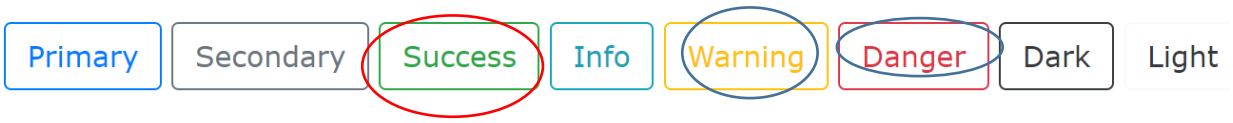
```
<div   class="container" >
                <button type="button" class="btn">Basic</button>
                <button type="button" class="btn btn-primary">Primary</button>
                <button type="button" class="btn btn-secondary">Secondary</button>
            </div>
```

Note the placement of these 3 buttons in a container, which push them toward the middle.



- *Button-outline*

The same way, you can create a button outline using the **".btn-outline**" class. You can also use the contextual classes with button outlines. In the code below, we created 3 button-outlines having respectively the ".success" , ".warning" and ".danger" Bootstrap conceptual class.




```
<div class="container">
    <button type="button" class="btn btn-outline-success">Success</button>
    <button type="button" class="btn btn-outline-warning">Warning</button>
    <button type="button" class="btn btn-outline-danger">Danger</button>
</div>
```

We can also decide the size of the button and the button-outline using the " **btn -*** " where the * can be one of these:

- lg: which stands for large
- md: which stands for medium
- sm: which stands for small

```
<div class="container">
      <h2>Button Sizes</h2>
      <button type="button" class="btn btn-success  btn-lg">Success</button>
      <button type="button" class="btn btn-outline-warning  btn-md">Warning</button>
      <button type="button" class="btn btn-danger  btn-sm">Danger</button>
</div>

<div class="container">
      <button type="button" class="btn btn-outline-success  btn-lg">Success</button>
      <button type="button" class="btn btn-outline-warning  btn-md">Warning</button>
      <button type="button" class="btn btn-outline-danger  btn-sm">Danger</button>
</div>
```

You can even make the button **unclickable** by adding the "disabled" attribute as follows .

```
<button type="button" class="btn btn-danger  btn-sm"  disabled>Danger</button>
```

➢ Forms && *Inputs*

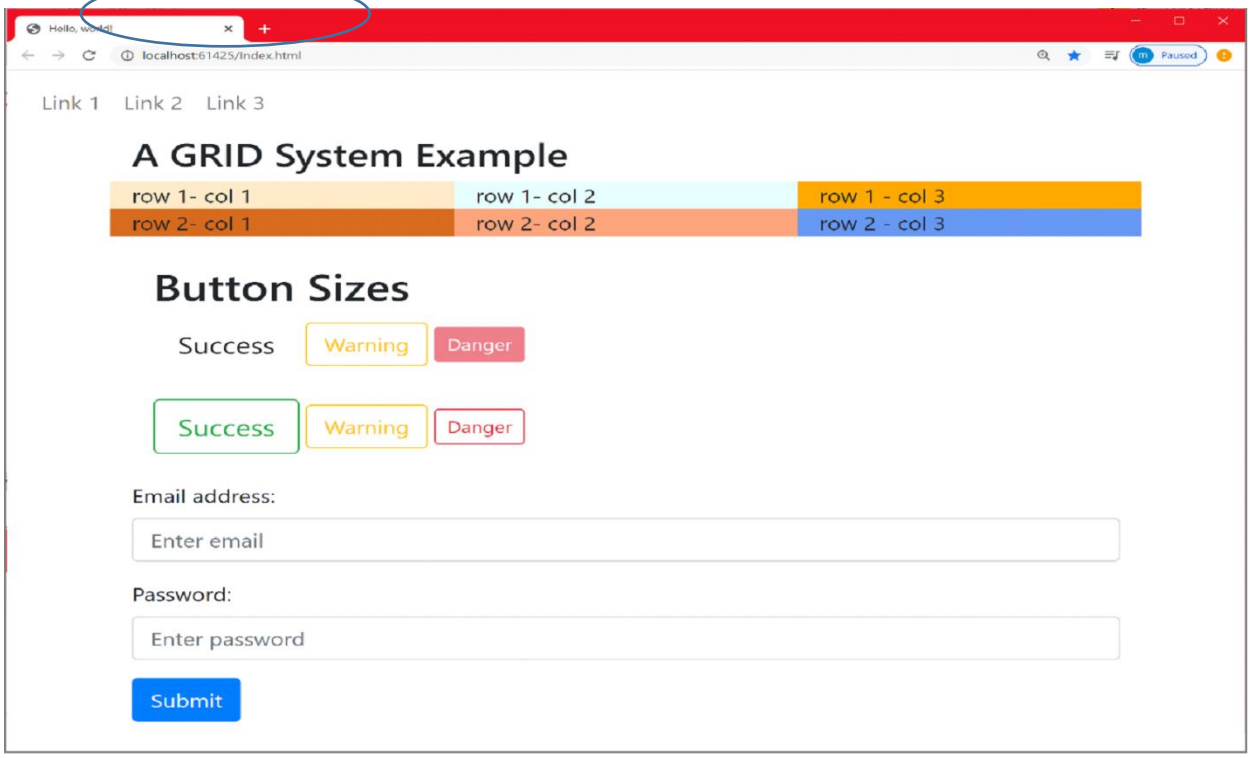
To use bootstrap with forms, we use the **".form-group**" class. In each form group, we use the **.form-control** class.

In the code below, we added a element with the ".form-group" class around each form control, to ensure proper margins. With the form-control class, you get a stacked form with some border….etc.

The following example creates a stacked form with two input fields.

```
<div class="container">
<form action="/action_page.php">
 <div  class="form-group">
  <label  for="email">Email address:</label>
  <input type="email" class="form-control" placeholder="Enter email" id="email">
 </div>
 <div class="form-group">
  <label for="pwd">Password:</label>
  <input type="password" class="form-control" placeholder="Enter password" id="pwd">
 </div>
```

```
  <button type="submit" class="btn btn-primary">Submit</button>
</form>
</div>
```



➢ Inline forms & Navbar

Sometimes, you want to group input elements side by side. For example, a login form that requires entering the username and the password and a submit button , or a search form. These scenarios can be implemented as a navigation bar as follows.

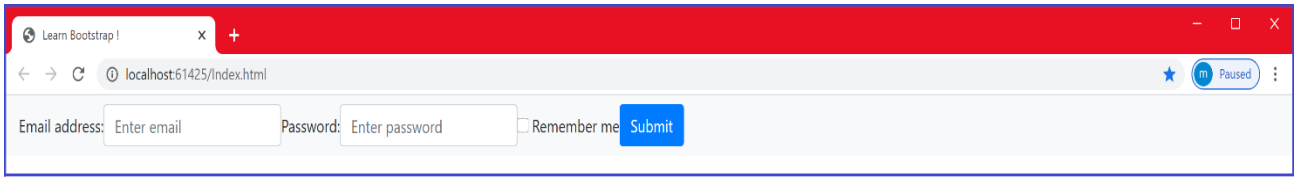
```
<body>
  <nav class="navbar navbar-expand-sm  bg-dark  navbar-dark">
    <form class="form-inline" action="/action_page.php">
```

```
        <input class="form-control mr-sm-1" type="text" placeholder="Search">
        <button class="btn btn-primary" type="submit">Search</button>
    </form>
  </nav>
```

In the code example above, we got an inline form, where all elements are inline and left-aligned by applying the class ".form-inline" class to the <form> element.

Example: Login form

```
<body>
  <nav class="navbar navbar-expand-sm bg-light navbar-dark">
     <form class="form-inline" action="/action_page.php">
        <label for="email">Email address:</label>
        <input type="email" class="form-control" placeholder="Enter email" id="email">
        <label for="pwd">Password:</label>
        <input type="password" class="form-control" placeholder="Enter password" id="pwd">

       <div class="form-check">
         <label class="form-check-label">
          <input class="form-check-input" type="checkbox"> Remember me
          </label>
        </div>
            <button type="submit" class="btn btn-primary">Submit</button>
     </form>
  </nav>
```



➢ *Form validation*

Bootstrap provides some validation classes that can be used to validate the user inputs.

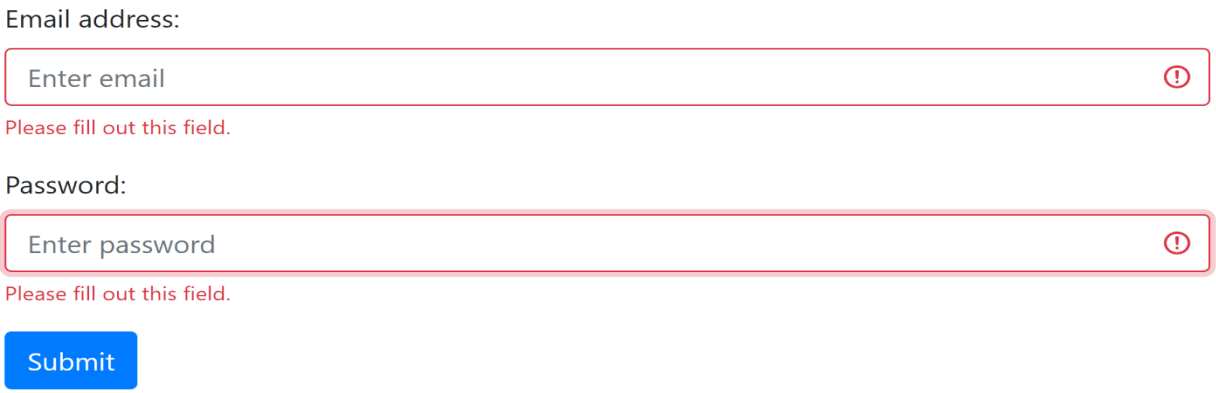
- .was-validated : to get feedback before submitting the form
- .needs-validation: to get feedback after submitting the form

To define a message to inform the user about what should be done before submitting, use

- .valid-feedback And .invalid-feedback classes

```
<form  action="/action_page.php" class="was-validated">
    <div class="form-group">
        <label for="email">Email address:</label>
        <input type="email" class="form-control" placeholder="Enter email" id="email" required>
        <div class="valid-feedback">Valid.</div>
        <div class="invalid-feedback">Please fill out this field.</div>
    </div>
    ……
……..
</form>
```

Note the "required" property that you should add for every input element



➢ *Input elements*

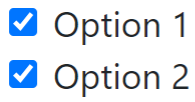
In this section, we are going to look at other input elements such as : checkboxes, radio buttons ,datetime, email, url, search, tel, and color.

Checkboxes/Radio buttons

```
<div class="form-check">
  <label class="form-check-label" for="check1">
    <input type="checkbox" class="form-check-input" id="check1" name="option1"  value="something" checked>Option 1
  </label>
  <label class="form-check-label" for="check2">
    <input type="checkbox" class="form-check-input" id="check2" name="option2"  value="something" checked>Option 2
  </label>
</div>
```

In the code example above, we wrap the checkboxes and labels inside a container. We used a element with the "form-check" class to ensure proper margin for the checkboxes and labels.

The .form-check-label class is used to label elements, while the .form-check-input is used to style checkboxes properly inside the .form-check container.



Tables

To add basic styling (light padding and horizontal dividers) to tables, use the **.table** class (

You can make the table stripped using the “table.stripped” class

```
<table class="table   table-striped">
```

You can add border using the “table.bordered” class

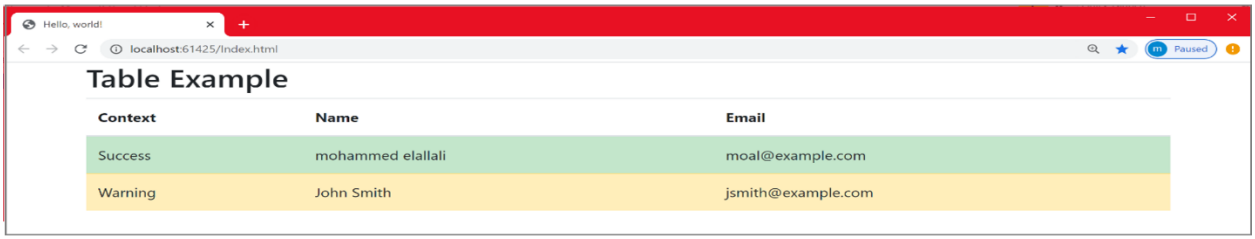
```
<table class="table   table-bordered">
```

You can make the table borderless using the table-borderless class.

```
<table class="table   table-borderless">
```

You can add color to the whole table, a row or a cell using what we call contextual classes. Here is an example of using contextual class to add color to rows.

```
<div class="container">
        <h2>Table Example</h2>
        <table class="table">
            <thead>
                <tr>
                    <th>Context</th>
                    <th>Name</th>
                    <th>Email</th>
                </tr>
            </thead>
            <tbody>
                <tr class="table-success">
                    <td>Success</td>
                    <td>mohammed elallali</td>
                    <td>moal@example.com</td>
                </tr>
                <tr class="table-warning">
                    <td>Warning</td>
                    <td>John Smith</td>
                    <td>jsmith@example.com</td>
                </tr>
            </tbody>
        </table>
</div>
```



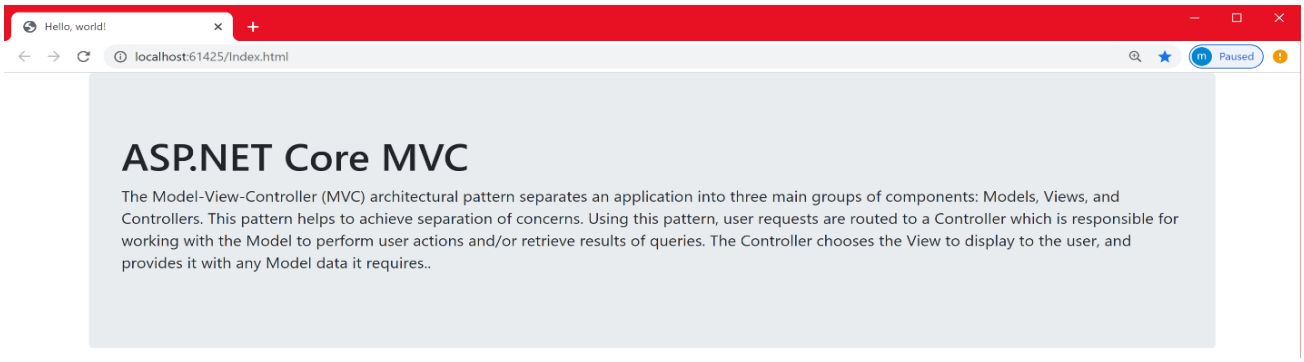
You can also use other contextual classes:

.table-primary , .table-danger , .table-active, .table-dark, .table-warning ……..etc.

- ***Jumbotron***

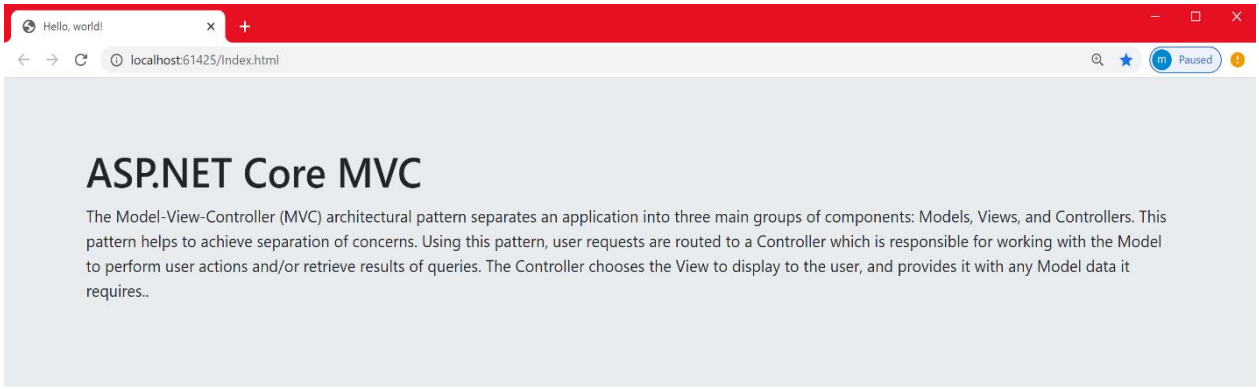
To create a jumbotron, use a element with the .jumbotron class , as shown below.

```
<div class="container">
   <div class="jumbotron">
     <h1>ASP.NET Core MVC</h1>
     <p>The Model-View-Controller (MVC) architectural pattern separates an application
       into three main groups of components:
       Models, Views, and Controllers. This pattern helps to achieve separation of
       concerns.Using this pattern, user requests are routed to a Controller which is
       responsible for working with the Model to perform user actions and/or retrieve
       results of queries. The Controller chooses the View to display to the user,
       and provides it with any Model data it requires..</p>
    </div>
</div>
```



To get a full-width jumbotron without rounded borders , add the .jumbotron-fluid class and a .container or .container-fluid inside of it:

```
<div class="jumbotron jumbotron-fluid">
      <div class="container">
        <h1>ASP.NET Core MVC</h1>
        <p>
                //  above text here
        </p>
      </div>
    </div>
```



➢ ***Typography && contextual classes***

Bootstrap typography makes it easy to create headings, paragraphs, ordered lists, unordered lists, inline elements, text alignment, text transformation etc. In this part, we are going to work with the previous text. For proper padding and alignment, we are applying the **".container**" class and we are placing this container in a jumbotron that spans across the whole screen size .

```
<div class="jumbotron jumbotron-fluid">
          <div class="container">
               <h1>ASP.NET Core MVC <small>pattern</small> </h1>
               <h1 >ASP.NET Core MVC <span class="small">pattern</span> </h1>
                <p>
                The <abbr title="Model View Controller"> MVC </abbr> architectural pattern
                separates an application into three main groups of components:
                <mark>Models, Views, and Controllers</mark>
                <span class="mark">Models, Views, and Controllers</span>
                </p>.
                <p><del>MVC is not an architecture</del> </p>
                <p class="font-weight-bold">
                        This pattern helps <u>to achieve separation of
                         Concerns </u>.</p>
                <p class=" text-left  text-uppercase"> Using this pattern, user requests are
                 routed to a Controller</p>
```

```
                <p class=" text-center">
                        The controller is responsible for working  with the Model to
                        perform user actions  and/or retrieve results of queries</p>

                 <p > <span class="h4">The Controller</span> chooses the View to display to
                        the user, and provides it with any Model data it requires </p>
                </div>
</div>
```

- `<small> pattern</small>` : to make the word "pattern" lighter, secondary text. Instead of using the <small> tag , you may use the ".small "class. These 2 statements give the same results:

```
<h1>ASP.NET Core MVC <small>pattern</small> </h1>
<h1 >ASP.NET Core MVC <span class="small">pattern</span> </h1>
```

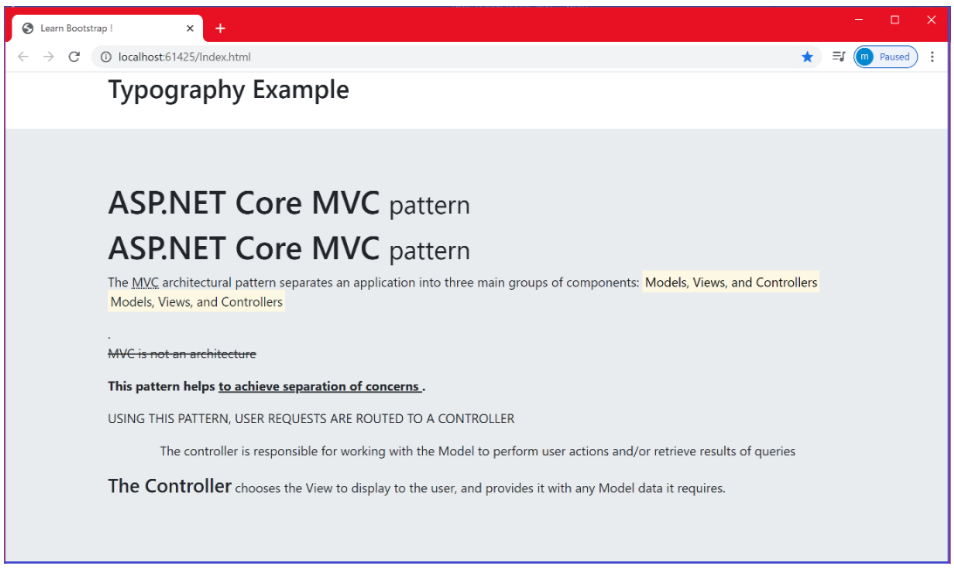
- `<mark> Models, Views, and Controllers </mark>` : to highlight the text with a yellow background. Instead of using the <mark> tag, you may use the ".mark" class. These 2 statements give the same results:

```
<mark>Models, Views, and Controllers</mark>
<span class="mark">Models, Views, and Controllers</span>
```

- `<del>MVC is not an architecture</del>` to strike through the text.
- The `"font-weight-bold">` class *to make the text bold*.

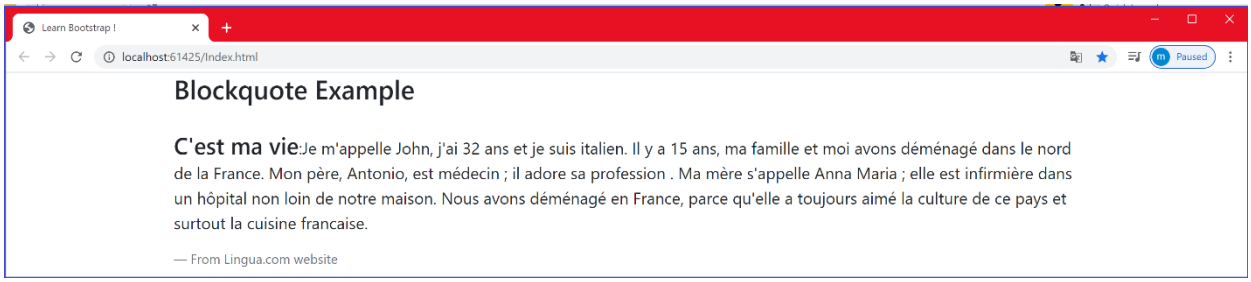
  NB: You can also use other font weights (normal, light …etc.)
- The `"text-left  text-uppercase"` to left align a text and make it uppercase
- The `"text-center"` to center-align a text.
- `<span class="h4">The Controller</span>` makes the text Controller inline with the rest of the text keeping the styling of an H4 element.

This is illustrated in the figure below.



You can also quote a block of content using the ".blockquote*" class.*

```
<div class="container">
    <blockquote class="blockquote">
    <p>
    <span class="h3"> C'est ma vie</span>:Je m'appelle John, j'ai 32 ans et je suis
     italien. Il y a 15 ans, ma famille et moi avons déménagé dans le nord de la
     France. Mon père, Antonio, est médecin ; il adore sa profession .  Ma mère
     s'appelle Anna Maria ; elle est infirmière dans un hôpital non loin de notre
     maison. Nous avons déménagé en France, parce qu'elle a toujours aimé la
      culture de ce pays et surtout la cuisine francaise.
    </p>
    <footer class="blockquote-footer">From Lingua.com  website</footer>
    </blockquote>
</div>
```



➢ ***Images: Add Images with Bootstrap***

```
<div class="container">
            <img src="image1.jpg" class="rounded" alt="Tour Eiffel" width="304"
height="236">
</div>
```

As shown below, we used the ".rounded" class to add rounded corners to the image


```
<div class="container">
      <img src="image1.jpg" class="rounded-circle" alt="Tour Eiffel" width="304" height="236">
</div>
```

As can be shown below, we used the ".rounded-circle" class to shape an image to a circle.

In the same way, you can use class="img-thumbnail" to shape the image to a thumbnail.

You can align images left and right using:

```
<img src="paris.jpg" class="float-left">
<img src="paris.jpg" class="float-right">
```

You can also make your images responsive, that automatically adjust to fit the size of the screen.

We create responsive images using the ".img-fluid" class as shown below:

```
<div class="container">
    <h4><small>Use the "rounded" class to add rounded corners to an image</small></h4>
    <img src="image1.jpg" class="img-fluid" alt="Tour Eiffel" width="304" height="236">
</div>
```

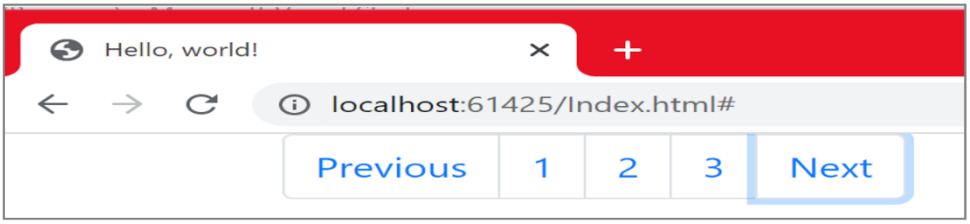
The .img-fluid class applies max-width: 100%; and height: auto; to the image:

- ***Pagination***

When you have a web site, you probably have many pages and you need to add pagination to the pages.

With bootstrap, we create pagination using the `.pagination` class to an `<ul>` element. Then add the `.page-item` to each `<li>` element and a `.page-link` class to each link inside `<li>` as shown below.

```
<div class="container">
        <ul class="pagination">
            <li class="page-item"><a class="page-link" href="#">Previous</a></li>
            <li class="page-item"><a class="page-link" href="#">1</a></li>
            <li class="page-item"><a class="page-link" href="#">2</a></li>
            <li class="page-item"><a class="page-link" href="#">3</a></li>
            <li class="page-item"><a class="page-link" href="#">Next</a></li>
        </ul>
</div>
```



Chapter 4 : Your First Razor Pages Application

What to learn in this chapter?

In this chapter, you will be introduced to some important features of the ASP.NET Core framework in general, such as extensibility, maintainability, routing, cross-platform …etc. You will also have the opportunity to build your first ASP.NET Core application using Razor Page. You will also explore the file organization and the code execution sequence when running the application.

Introduction to ASP.NET Core

Today web development is changing fast. Applications are more and more requiring modular frameworks. For example: There is a high demand for cross platform behavior, being able to run on many different platforms. ASP.NET Core supports a component based and modular architecture, making the application more lightweight as it only incorporates the components it needs. ASP.NET Core supports many important features such as:

Extensibility :

> **Definition**: Extensibility is a measure of the ability to extend a system and the level of effort required to implement the extension. Extensions can be through the addition of new functionality or through modification of existing functionality (www.wikipedia.org).

ASP.NET Core applications are independent-components based, having well defined .NET interfaces. This makes your design more flexible as it is easy to extend the app components by either adopting the default implementation of the component or entirely replace it with your own implementation.

Testability

As mentioned before, an ASP.NET Core application is independent-component based. Each component has a well-defined .Net interface. The framework was built with some design principles in mind (i.e. separation of concerns..etc). This will of course enhance maintainability of your application because each component can be isolated and thus unit tested separately without worrying about dependencies on external components or infrastructure. **In chapter 12** , we will dig into testing in details

Powerful Routing System

When you are visiting a web site, you are probably requesting some resources on the net. How will the URL that you type on the browser address match a unique resource?. This is the job of what we call the routing process. With ASP.NET Core, the way we locate resources in a web application has also evolved. ASP.NET Core provides a very powerful, understandable and clean routing format. The structure of the URL is becoming straightforward for the

web users to understand it by removing the technical details. Routing is treated in chapter 9.

Cross-Platform

What is cross-platform means ? Cross-platform software is a type of software application which works on multiple operating systems or devices, which are often referred to as platforms. A platform means an operating system such as Windows, Mac OS, Android or iOS. When a software application works on more than one platform, the user can utilize the software on a wider choice of devices and computers (www.bobology.com).

Unlike Previous versions of ASP.NET which were Windows specific, ASP.NET Core is cross-platform both for development and for deployment. ASP.NET Core applications can run on different platforms- including Linux and OS X/mac OS. Microsoft has created a cross-platform development tool called Visual Studio Code to allow development on these platforms.

What are Razor Pages?

ASP.NET Core provides a web development framework based on the Model-View-Controller (MVC) pattern. On top of that sits the Razor Pages framework, which is the main focus in this tutorial. ASP.NET Core Razor Pages is a new set of tools , a page-centric development approach for building web applications. Let us jump right away into coding. We are going to build a Razor Pages application to manage events that take place in Denmark. The user of the application will be able to display the list of all events, add new events, delete,

edit, display the details of a specific event, filter the list of events based on the city name where the event is taking place …etc.

Create a simple Razor Pages application

Prerequisites

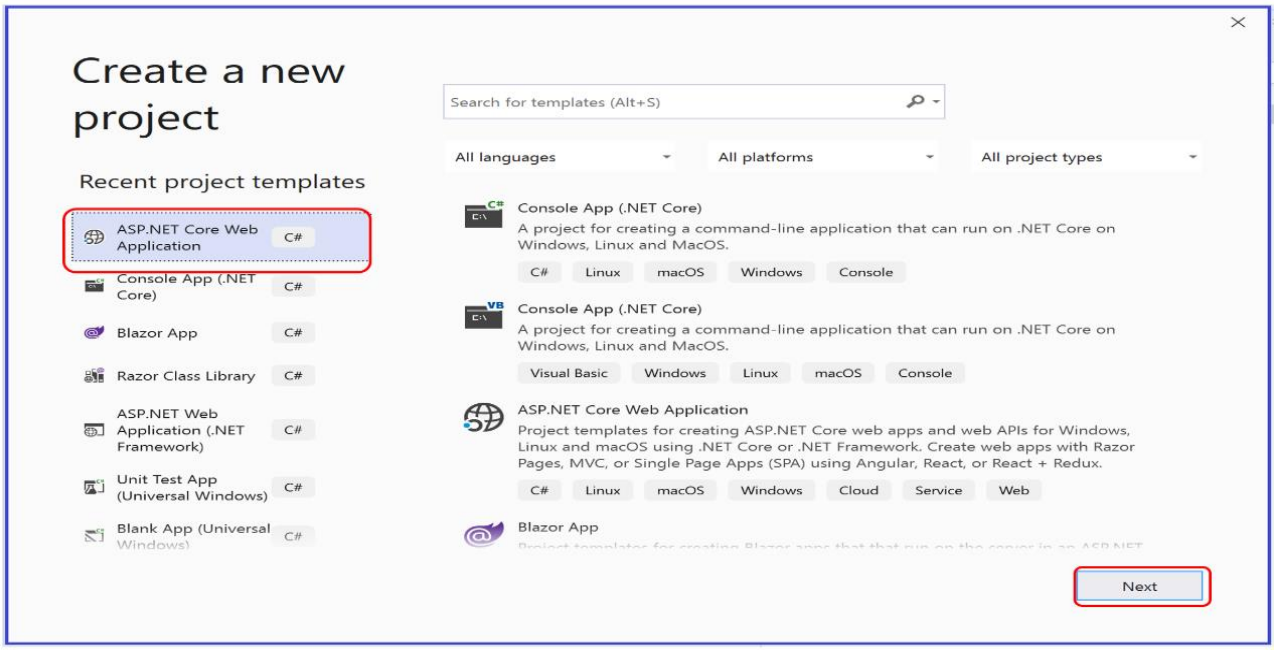
- Visual studio
- Visual studio Code
- Visual Studio for Mac

- Visual Studio 2019 16.4 or later with the ASP.NET and web development workload
- .NET Core 3.1 SDK or later

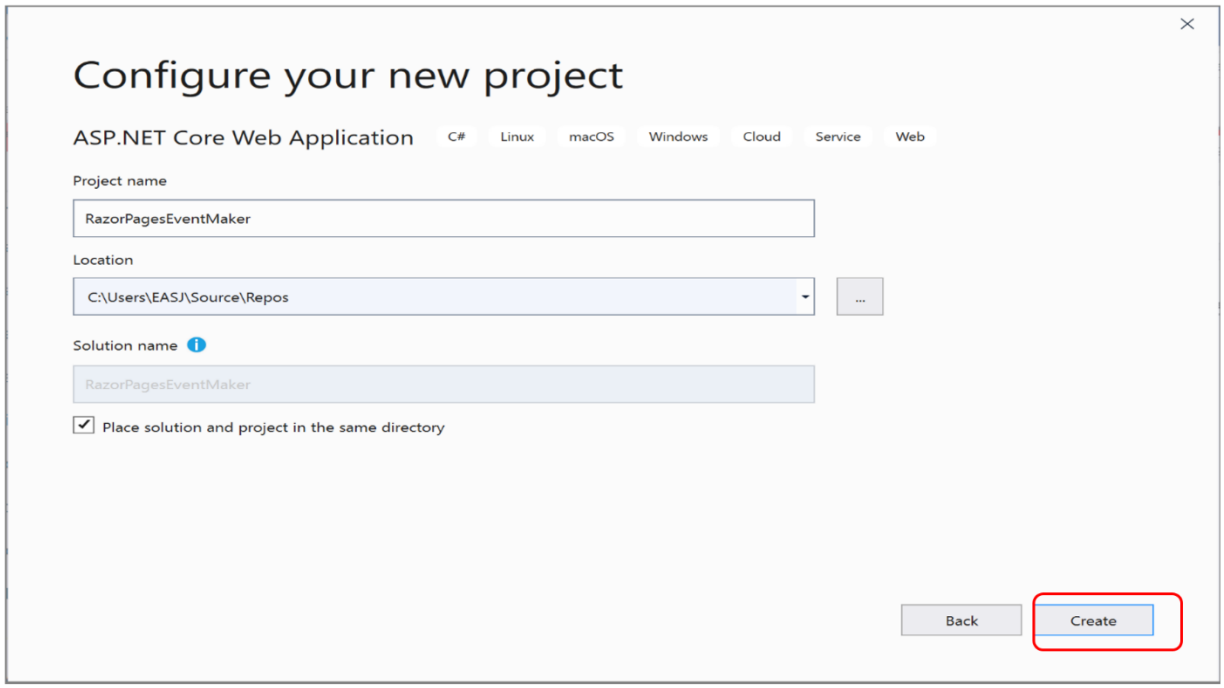
We will take things step by step.

In this tutorial, I am working with Visual Studio 2019 16.7.2

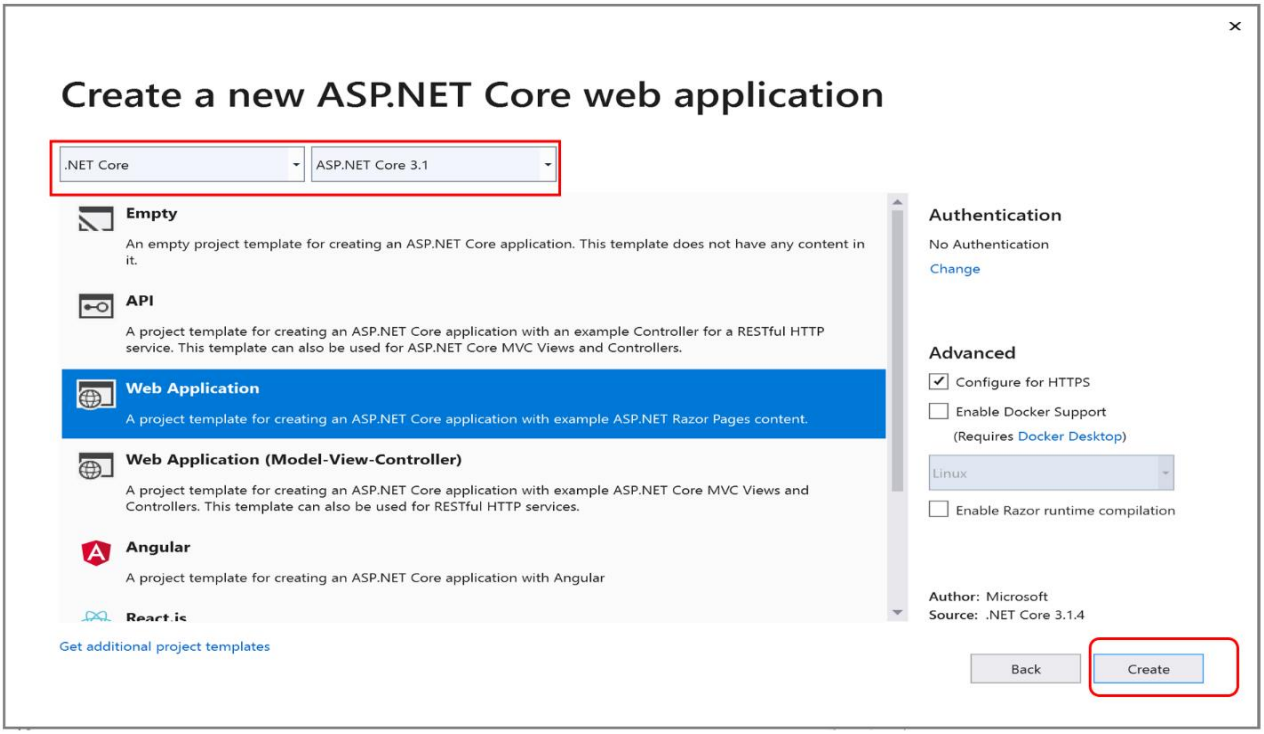
- Launch Visual studio and in the File menu, select **Create a new project**.
- Select **ASP.NET Core Web Application** , then select **Next,** as shown in The figure below.



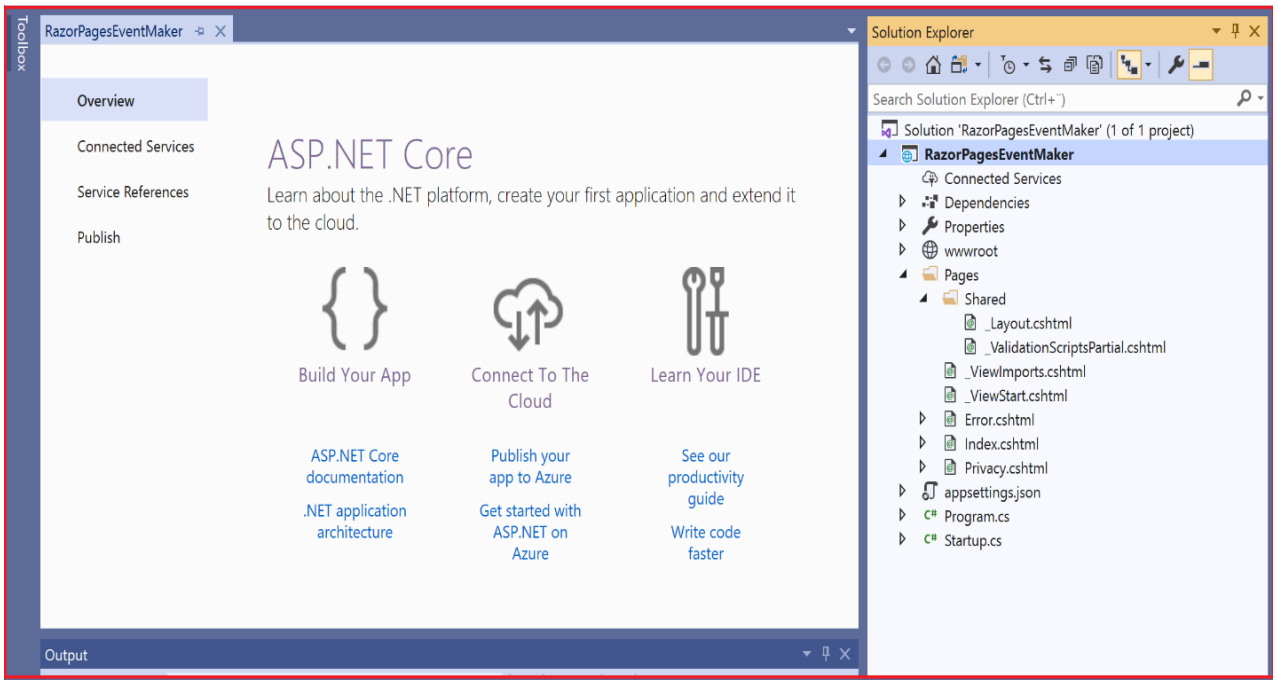
- hvad så bror: Name the project as “ **RazorPagesEventMaker”** and click on ”**Create”**.



- Select ***ASP.NET Core 3.1*** . Do not consider any authentication for the moment.
- Select **Web Application template**, and click on **Create.**



Visual studio creates a working Razor pages application for you. the new created app has the structure shown in the figure below.



Project files

Just few words about the project files. By default, a "**Pages**" folder is created. This "**Page**s" folder is by default the root folder. By root folder we mean the folder that is visited at the start of the application. It encapsulates all our Razor pages.

Within the "**Page**s" folder, there is another folder called "Shared", which contains pages that are common to all pages in the application. An example of such pages is the **"_Layout.cshtml**" page. As we will see in a moment when we run our application, this page provides the navigation menu at the top of the page and the copyright notice at the bottom to any page that is using it.

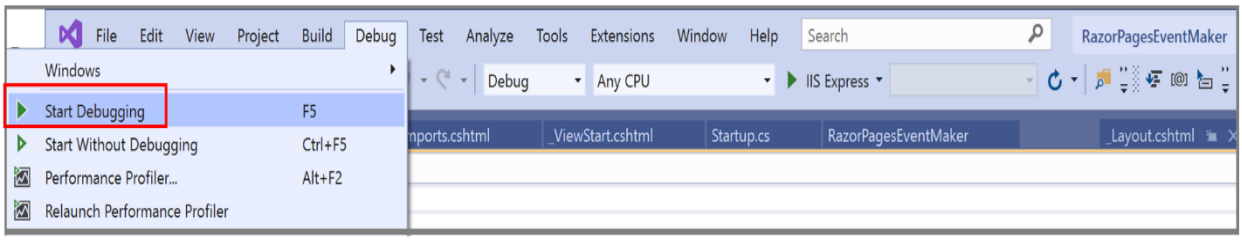
The **appSettings.json** file contains configuration data. We will be using this file much in Part II (second semester) when we set the connection string , for example.

_ViewImports.cshtml is where to place any directive that you want to be available across all Razor Pages so that you don't have to add them to pages individually.

_ViewStart.cshtml contains code that is executed at the start of each Razor Page's execution. The most common use for the ViewStart file is to set the layout page for each Razor Page.

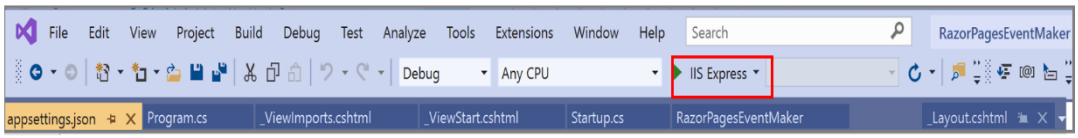
Now, let us run the application. As you will see, many things will make sense to you.

You can run the application by selecting **Start Debugging** (another alternative is using start without debugging) from the Debug menu as shown below.



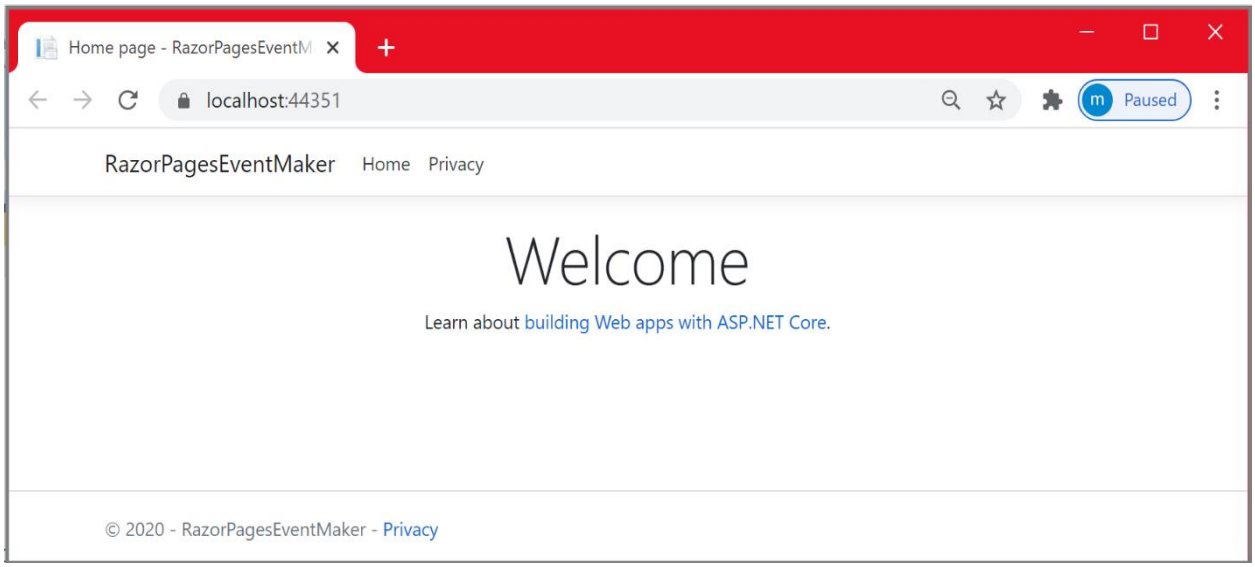
The application compiles and runs on the "**IIS Express**" application server.

- **NB**: You can also run the application by clicking on the **ISS Express** button.



Then, the web browser opens and displays the application content. The content of the Index.cshtml page is rendered. The figure below shows the

output. Notice the presence of the top menu that is part of the "**_Layout.cshtml**" we mentioned earlier.



The workflow of the application- Where the application bootstraps ?

Let us understand what happened when we run the application , looking at the code execution sequence from the time you run your application until the display of the content on the browser.

When running the app, the starting point of the application is the main method defined in the **Program.cs file**. As shown in the code snippet below , we are calling the CreateHostBuilder method from the main method. This method, in turn, uses the Statup.cs class that is used to configure application services. We will dig into services in later chapters, but for the moment let us have a look at the Startup.cs file.

Program.cs

RazorPagesEventMaker | RazorPagesEventMaker.Program

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Threading.Tasks;
5  using Microsoft.AspNetCore.Hosting;
6  using Microsoft.Extensions.Configuration;
7  using Microsoft.Extensions.Hosting;
8  using Microsoft.Extensions.Logging;
9
10 namespace RazorPagesEventMaker
11 {
       0 references
12     public class Program
13     {
           0 references
14         public static void Main(string[] args)
15         {
16             CreateHostBuilder(args).Build().Run();
17         }
18
           1 reference
19         public static IHostBuilder CreateHostBuilder(string[] args) =>
20             Host.CreateDefaultBuilder(args)
21                 .ConfigureWebHostDefaults(webBuilder =>
22                 {
23                     webBuilder.UseStartup<Startup>();
24                 });
25     }
26 }
```

Startup.cs

ConfigureServices

This class defines the startup logic for the application. In the **ConfigureServices** method, which is called by the runtime, we define all the services (reusable components) that we want to have access to via dependency injection. Dependency injection is a very important concept in ASP.NETCore that we are going to talk about later on in chapter 10.

```
23      // This method gets called by the runtime. Use this method to add services to the container.
        0 references
24      public void ConfigureServices(IServiceCollection services)
25      {
26          services.AddRazorPages();
27      }
```

Configure method

In the Configure method shown in the listing below, we are mainly using some routings. In simple words, in this method, we are specifying how the app responds to user requests. Note that the order in which these app components are placed is important.

```
29      // This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
        0 references
30      public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
31      {
32          if (env.IsDevelopment())
33          {
34              app.UseDeveloperExceptionPage();
35          }
36          else
37          {
38              app.UseExceptionHandler("/Error");
39              // The default HSTS value is 30 days. You may want to change this for production scenarios, see https://aka.ms/aspnetcore-hsts
40              app.UseHsts();
41          }
42
43          app.UseHttpsRedirection();
44          app.UseStaticFiles();
45
46          app.UseRouting();
47
48          app.UseAuthorization();
49
50          app.UseEndpoints(endpoints =>
51          {
52              endpoints.MapRazorPages();
53          });
54      }
55
```

*The **MapRazorPages** method is used to set up the default route. If you remember, we* said earlier that the **Index.cshtml** is the one that is returned to the user when we first run the application.

You may already wonder how we know that. As you can see, there is no visible indication about that. However, for the moment, you should admit that

the **MapRazorPages** call above ensures that endpoint routing is set up for Razor Pages and the default page is Index.cshtml.

Let us have look at the markup of the **Index.cshtml** page that we pretend it is displayed when running the application.

Index.cshtml* | appsettings.json | Program.cs | _ViewImports.cshtml | _ViewStart.cshtml | Startup.cs | RazorPagesEventMaker

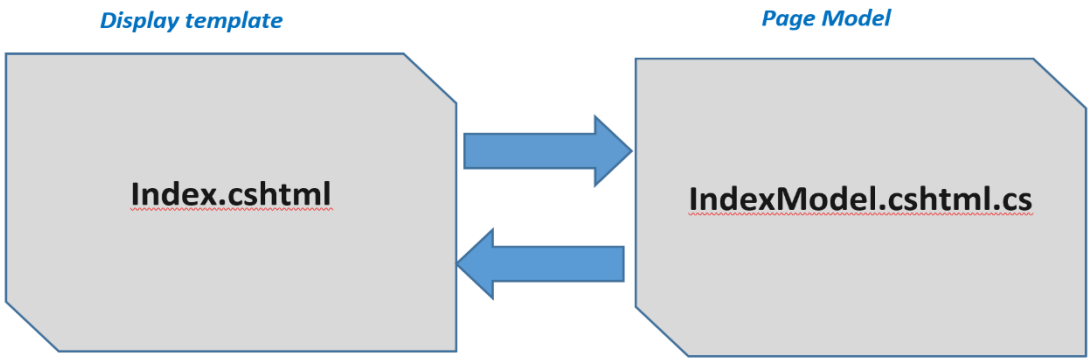
```
1 @page
2 @model IndexModel
3 @{
4     ViewData["Title"] = "Home page";
5 }
6 <div class="text-center">
7     <h1 class="display-4">Welcome</h1>
8     <p>Learn about <a href="https://docs.microsoft.com/aspnet/core">building Web apps with ASP.NET Core</a>.</p>
9 </div>
```

The html Markup of the ***Index.cshtml*** page is shown in the listing above. For the moment, if you do understand this code, do not worry because we will have a lot of opportunities to cover this in the future.

- **Line 1** : the @page directive indicates that it is a Razor page. So, placing the @page directive at the beginning of a page is critical. As we will see , when we cover routing , the only content that we can add to this line is the route template.

- **Line 2:** The @model **IndexModel** specifies the model for the page. Indeed, a Razor page is a pair of files:
  - A ***.cshtml*** file that contains HTML markup with C# code using Razor syntax. It is called the display template.

- A ***.cshtml.cs*** file that contains C# code that handles data in a data source. It is called the page model.

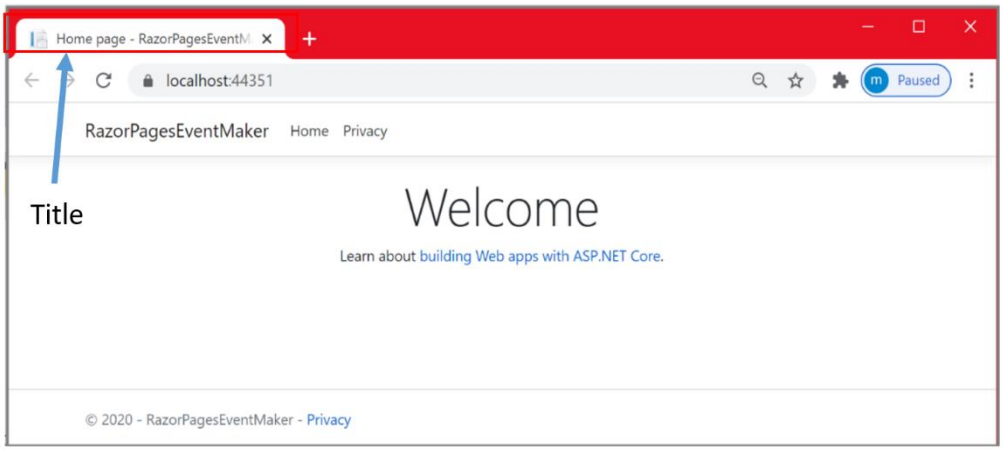


Notice that the name of the model is the same as the name of the page ending with Model. However, it is important to know that you can work with a display template without a PageModel.

- **Lines 3-5** : We denote a Razor code block using `@{` code here `}.` In this code, we are trying to pass the title of this razor page "**Home page**" to the _Layout.cshtml view using the **ViewData** dictionary. We will have the opportunity to work with ViewData , a string key based dictionary . Below, I am going to explain how the title is set in the _Layout.cshtml view.

- **Line 6-9** : The html code is a <**div**> element to which we applied the bootstrap class "text-center" to center the text. The element contains a heading <**h1**> that uses display-4 bootstrap class to make the heading stand out. It also contains a paragraph that encapsulates a text and a link (or an anchor element). This

HTML markup corresponds exactly to what was rendered when we first run the application, as shown below.

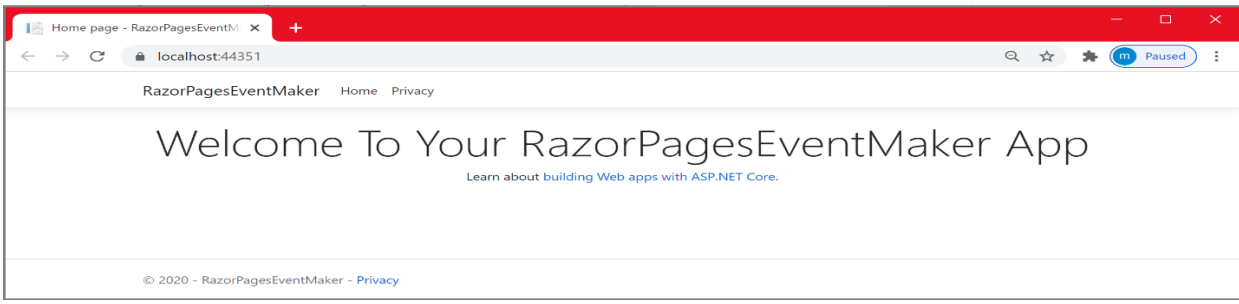


If you are in a doubt about the first page that is displayed , try to change some text and run the application again to see the effect. As shown below, In the Index.cshtml file, I did change the text "**Welcome**" into "**Welcome To Your RazorPagesEventMaker App**". t

Index.cshtml* | appsettings.json | Program.cs | _ViewImports.cshtml | _ViewStart.cshtml | Startup.cs | RazorPagesEventMaker

```
1  @page
2  @model IndexModel
3  @{
4      ViewData["Title"] = "Home page";
5  }
6  <div class="text-center">
7      <h1 class="display-4">Welcome To Your RazorPagesEventMaker App</h1>
8      <p>Learn about <a href="https://docs.microsoft.com/aspnet/core">building Web apps with ASP.NET Core</a>.</p>
9  </div>
10
```

As you can see, the change is indeed reflected in the figur below.



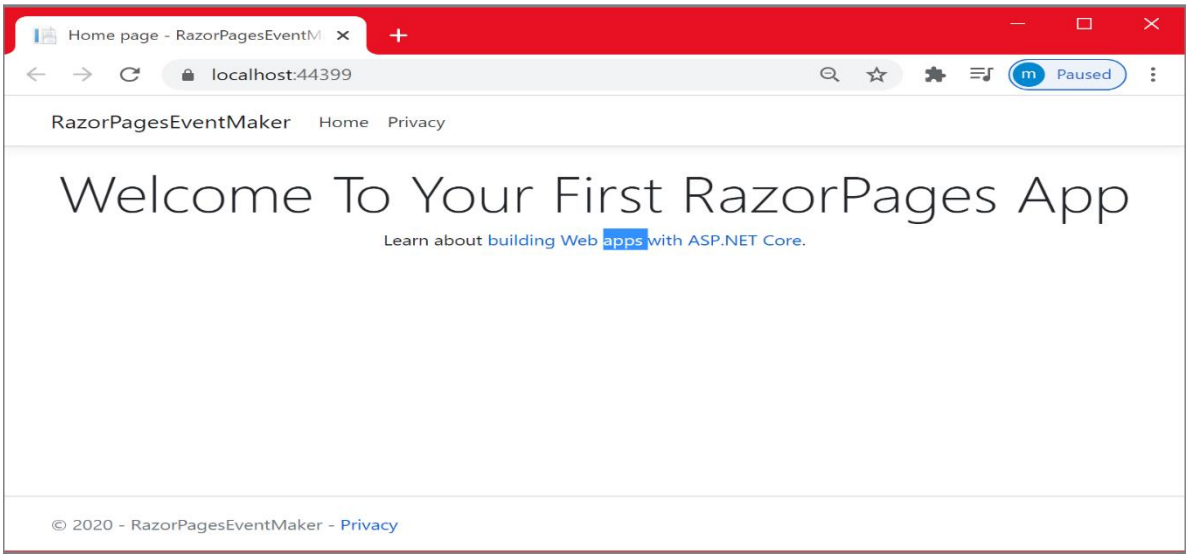
You can even try another trick. Instead of hard coding this static text, let us get it dynamically using the Index.cshtml display and its model as shown below. In the code snippet, we created a property **Message** that is initialized to the string "**Welcome To Your First RazorPages App**".

```
11  public class IndexModel : PageModel
12  {
13      private readonly ILogger<IndexModel> _logger;
        1 reference
14      public string Message { get; set; }
        0 references
15      public IndexModel(ILogger<IndexModel> logger)
16      {
17          _logger = logger;
18      }
19
        0 references
20      public void OnGet()
21      {
22          Message = "Welcome To Your First RazorPages App";
23      }
24  }
25
```

On running the application, the **OnGet()** method is invoked and the Message property is initialized. To get the value of the Message on the razor page, we use the **@Model.Message** as it is shown below. That's it. You do not need to understand the magic behind this. It is a simple example on how the page and its model exchange data. Do not worry; we will get into it in detail when working with real objects. This is the subject of the chapter about model binding.

```
@page
@model IndexModel
@{
    ViewData["Title"] = "Home page";
}
<div class="text-center">
    <h1 class="display-4">@Model.Message</h1>
    <p>Learn about <a href="https://docs.microsoft.com/aspnet/core">building Web apps with ASP.NET Core</a>.</p>
</div>
```

As expected, the @Model.messge statement is rendered as follows.



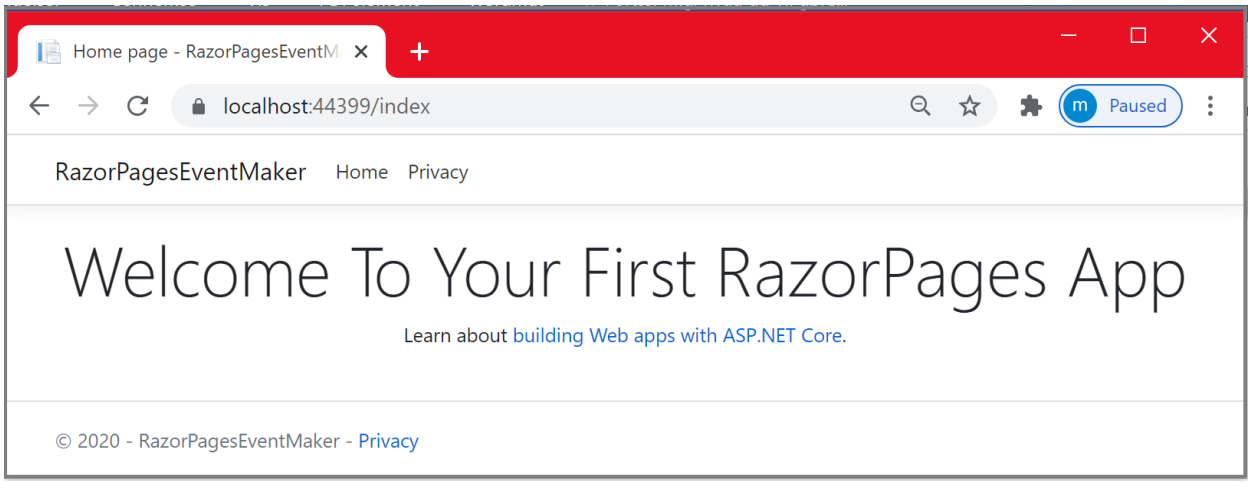
Now that we know that the Index.cshtml razor page is displayed on running the application, **how does the mechanism of selecting a page work? What if I want to display another page** ?

It is a routing issue and as we did not cover routine yet, it will not be convenient to answer this question at the moment. However, we previously mentioned that we can incorporate the routing template at the @page directive. As you can see below, there is no route template at the Index.cshtml Razor page. This means that the Index.cshtml page can be invoked on the root URL: **localhost:44399**/.

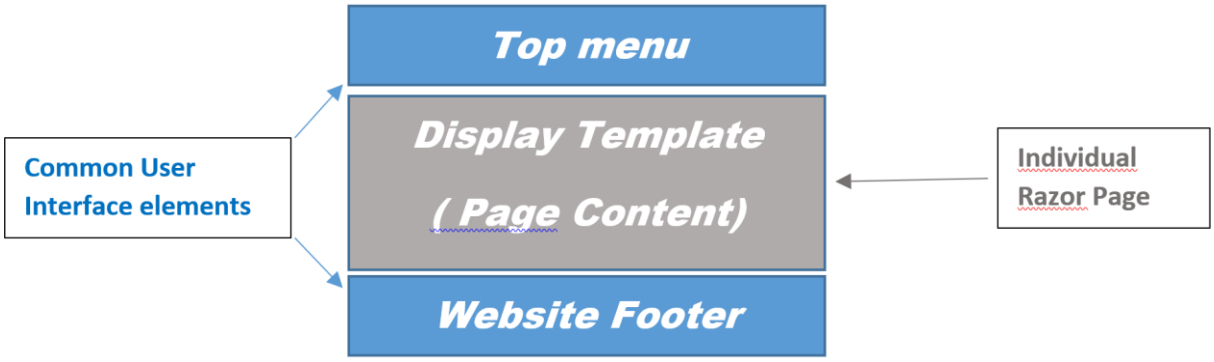
```
1 @page
2 @model IndexModel
3 @{
4     ViewData["Title"] = "Home page";
5 }
6 <div class="text-center">
7     <h1 class="display-4">@Model.Message</h1>
8     <p>Learn about <a href="https://docs.microsoft.com/aspnet/core">building Web apps with ASP.NET Core</a>.</p>
9 </div>
```

@page means it a razor page that can be invoked on the root URL: localhost:44399/ because there is no URL specified

Notice that if you navigate to localhost:44399/index, you will also get the Index.cshtml razor page displayed. This is shown below. This is because the Index.cshtml is set as the default by the system. We say that this page maps to both URLs.



Now, let us first look at one important page, which is the **_Layout.cshtml** that is created by default. With Razor Pages a layout is somehow another component that includes common user interface elements that are used by every component in the apps. Doing so, we are following the "**Do not Repeat Yourself**" principle. These components could be menus, copyright messages, and company logos. The default layout for a Razor page application is illustrated in the figure below.



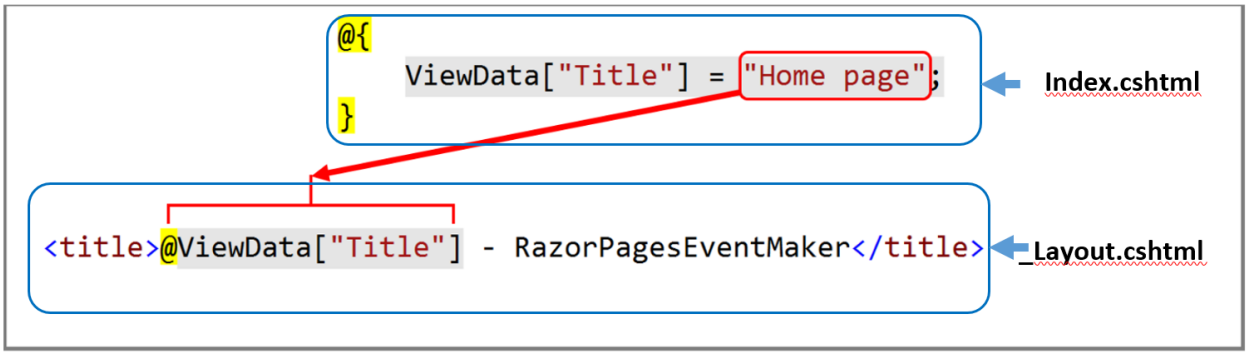
With Razor pages, the default layout is the view **"_Layout.cshtml"** located within the Views/Shared folder. I said "View" because it is not a razor page as the file does not start with the @page directive (see the figure below).

Let us explore the code of this page**.**

In the header, we defined the title for any razor page that uses this layout view. **How the _Layout view knows about the title of the Index.cshtml razor page?** This is done using the **ViewData** dictionary property.

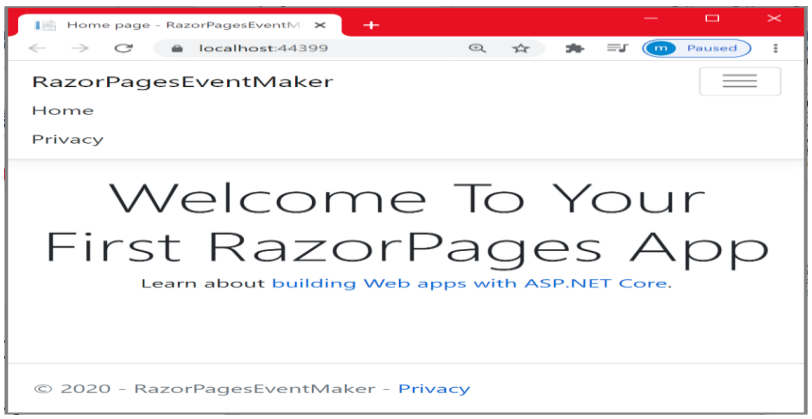
One way to pass data from the PageModel to the content page is to use ViewData. It is a dictionary of objects. The only constraint is that the key should be a string.

Let us look at how it works. In the code snippet below, we first set the **Title** property of ViewData to "Home page" in the Index razor page. Then, as the ViewData dictionary is automatically made available to the _Layout.cshtml view, we get the stored value by referring to it using the key as follows : **@ViewData["Title"]**. This way, any page that uses the **_Layout.cshtml** can pass its title to the layout page. As expected, the result is the following title: "**Home page - RazorPagesEventMaker**".



However when working with objects, it is not a good idea to access object properties in the page content through the use of ViewData. Using ViewData is error prone and you will not benefit from strong typing: **IntelliSense and compile-time checking**. ViewData is most used when working with **layout pages**.

In the header, you may also recognize the code to include Bootstrap´s compiled CSS in the view. HTML elements of the page are indeed using some Bootstrap classes that you may have seen in the chapter about Bootstrap. One important use of Bootstrap is the class "**Container**" applied to the top menu. If you remember, this class will make your top menu responsive. That means, when the size of the device is reduced, the top menu collapses to a hamburger menu as shown below.



In the header, you may also recognize the **<script>**elements at the bottom just before the closing<Body> tag, in case the page requires JavaScript script file and/or jQuery script file.

When you run your application, you may notice the existence of a top menu. The top menu is defined in the layout view as the following three anchor elements shown below.

```
<a class="navbar-brand" asp-area="" asp-page="/Index">RazorPagesEventMaker</a>

<a class="nav-link text-dark" asp-area="" asp-page="/Index">Home</a>

<a class="nav-link text-dark" asp-area="" asp-page="/Privacy">Privacy</a>
```

This page uses Tag helpers , which is the subject of a subsequent chapter. At this stage, I should just mention that we are using **asp-page** tag helper to specify the URL of the page that is displayed when clicking on the link. In a subsequent chapter, we will discuss in detail some important tag helpers and their use.

The first two links display the Index.cshtml razor page while the last one displays the Privacy.cshtml razor page. These three files are located at the root folder which is by

default the **"Pages**" folder. In order to add a new item at the top menu, you need just to create a new anchor element pointing to the razor page that you want to display.

@RenderBody()

The most important part of this file is the **@RenderBody()** part used to specify the location in the layout markup where the content of a view is rendered. So every time a razor page is displayed, it is rendered at this place.

References

- https://www.learnrazorpages.com/asp-net-core
- https://docs.microsoft.com/en-us/aspnet/core/tutorials/razor-pages/razor-pages-start?view=aspnetcore-3.1&tabs=visual-studio
- https://www.learnrazorpages.com/razor-pages
- https://www.c-sharpcorner.com/article/fundamentals-in-asp-net-core-razor-pages/

Chapter 5: Razor pages architecture

Introduction:

In this chapter, you will get a good understanding of the Razor pages architectural pattern . You will also get a deep understanding of how user

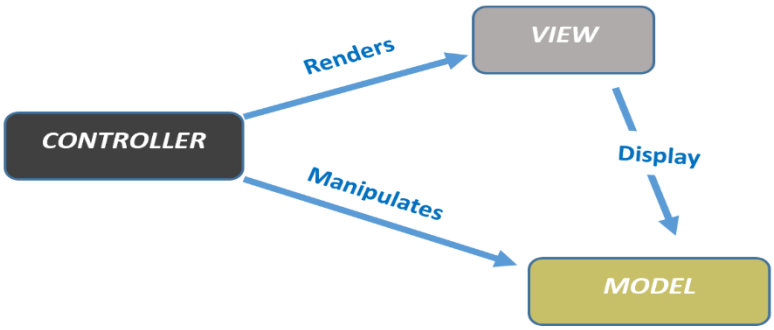
requests are handled by the application and how a response is returned to the user.

MVC && Razor Pages architectures

If you have knowledge about MVC, the MVC architecture has three components: Model, View and Controller. ASP.NET Core MVC is not the subject of this tutorial. My intention to briefly introduce the MVC architecture is not to compare these two technologies, but rather to show how modern software architecture is built today.

One of the considerations in designing software is the "separation of concerns" principle. In general, ASP.NET Core applications adhere to this principle.

The ASP.NET Core MVC architectural pattern is illustrated in the figure below.



Each component has its own concern :

- The model encapsulates the **business logic**. The model uses validation logic to enforce business rules for the data.

- The view enforces the **UI logic**. Views should not perform business logic nor interact with a data source for example. Rather, views should work with the data provided by the controller.
- As the controller is the component that directly receives input from the user, it is responsible for the **input logic**.

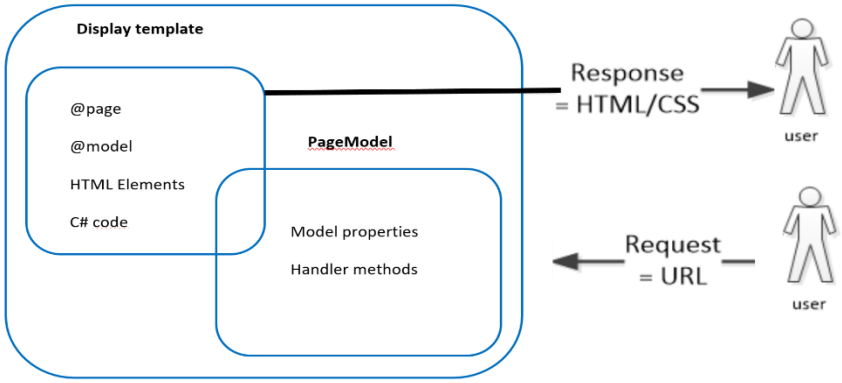
Razor pages architecture

Compared to the MVC pattern, the razor pages architecture has only two components: the **display template**, which is the view whereas the **PageModel** represents the model and the controller. Razor Pages aim to enhance separation of concerns as well. The UI layer (the **.cshtml** *view* file) enforces UI Logic, while the PageModel (the **.cshtml.cs** file) takes care of the processing logic for the page. The PageModel class is made available to the view file via the **@model** directive as we have seen before.

Either using MVC or Razor pages, Such separation of concerns helps manage the application complexity and maintainability. Indeed, you can easily implement and test the UI code without depending on the business code. It also facilitates automated unit/GUI testing and enables greater flexibility for teams to work independently on each other (View team && Processing logic team).

How Razor Pages process a user request ?

Before starting building our EventMaker application, it is important to understand how a request is processed. When the user initiates a request by clicking, for example, on a submit button on the page, the **PageModel handler method** intercepts the request and its code is automatically executed , then another or the same Razor page may be returned. You can also return a file (i.e Json) or a simple string as we have seen earlier. You can also redirect to another resource. The process is illustrated in the figure below



But How we map requests to handler methods?.

The mapping mechanism is based on a naming convention of the handler methods.

So how handler methods are named?

Handler methods are named using the pattern **On<verb>** where verb is one of the HTTP verbs (Post, Get, Put, Delete…etc.). For example The **OnGet** method is selected for **GET requests** and the **OnPost** method is selected for **POST requests.** We also use

On<verb> with **Async** appended to specify that the method is asynchronous (Asynchronous programming is covered in the second semester).

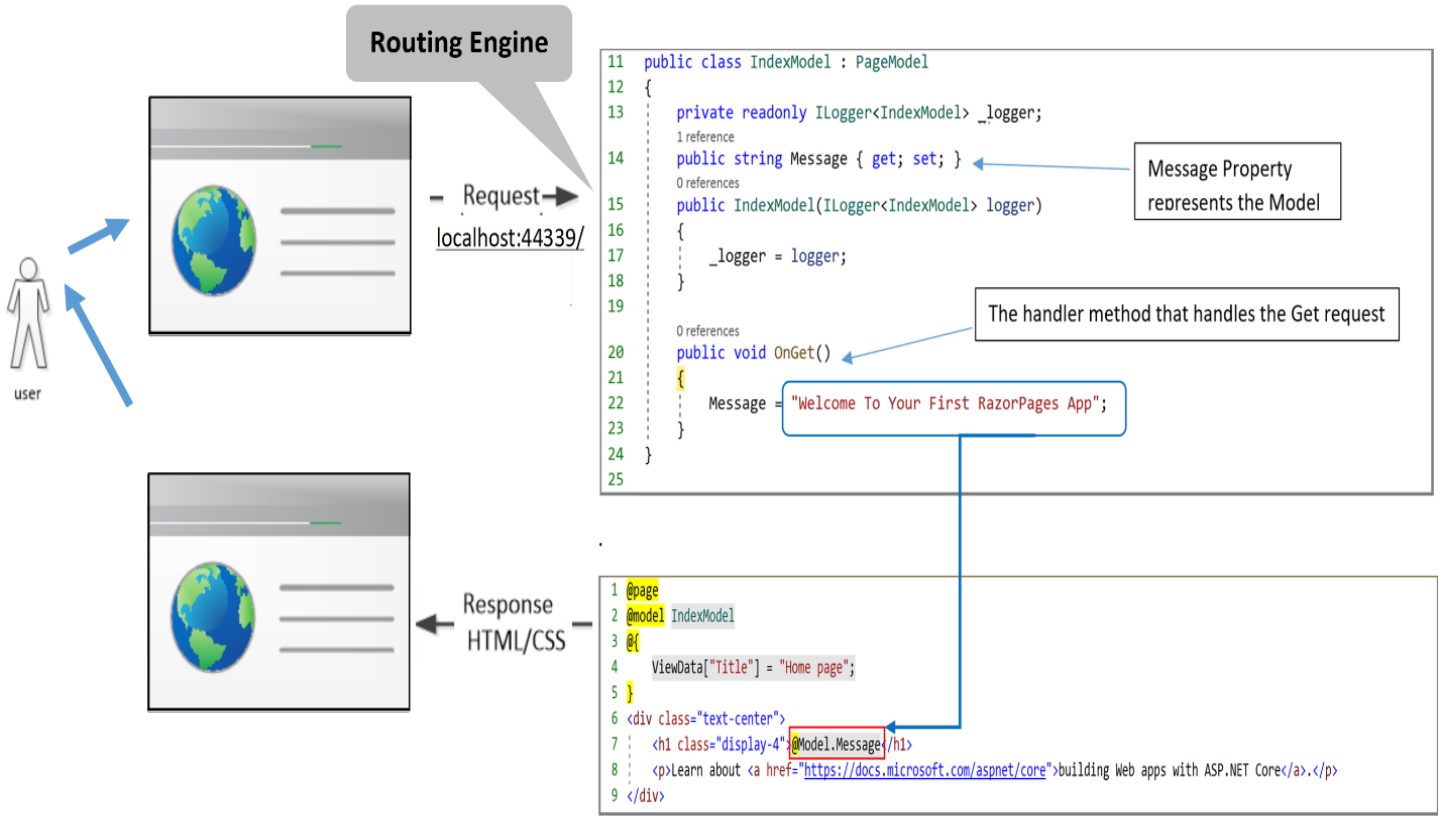
The only requirement for the **handler methods is that they should be public** and can have any return type. It is common that a handler method has a return type of **void** (or Task if asynchronous) or **an action result** (**IActionResult** type is better).

Let us look at the code and see what happened. As a code example, we consider the code example seen in chapter 4.

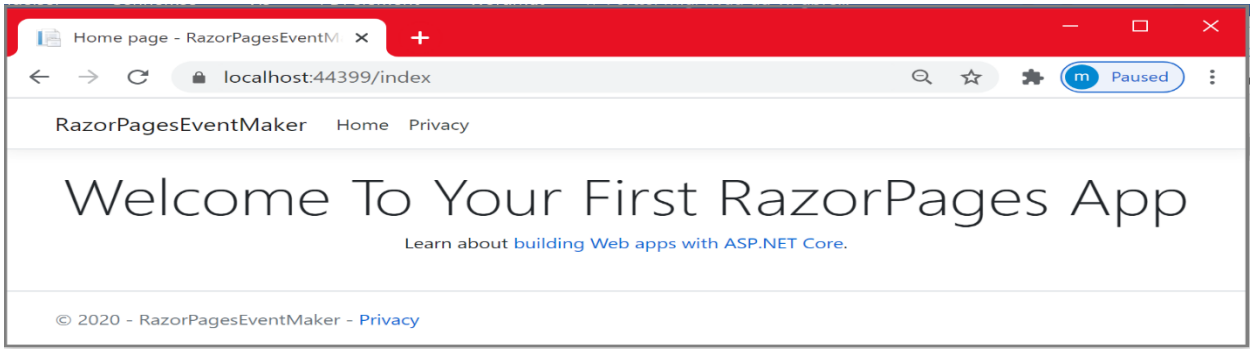
The code below shows the IndexModel class. The generated IndexModel class inherits from **Microsoft.AspNetCore.Mvc.RazorPages.PageModel**, which has a number of properties that enable you to work with various items associated with the HTTP request. In this class, we have defined a property called "**Message**". This property is initialized in the OnGet method. The **OnGet** method returns a void type, that means the index.cshtml razor page is returned by default. In this razor page, we used the **@Model.Message** statement to get the value of this property. This is simply because all the properties and methods that you apply to the PageModel class are available on the **Model** property in the Razor Page. (the **@** sign is used because we are in the C# world)

So, if we run the application and navigate to localhost:44399/index or just localhost:44339/ (because as I mentioned before , the Index.cshtml is the default page) , the OnGet is invoked simply because we send a Get request (that means, we want to display the index page content). The property is then

initialized and its value is passed and displayed on the page using the **Model** property.



As expected, the @Model.messge statement is rendered as follows.



Question: Try to replace the method **OnGet** with the following code. Did you get the same output? If yes, try to figure out how ?

HINT: Explore the type returned by the Page() method.

```
public IActionResult OnGet()
    {
       Message = "Welcome to Your First Razor pages App";
       return Page();
    }
```

References

- https://www.twilio.com/blog/introduction-asp-net-core-razor-pages
- https://www.codeproject.com/Articles/1208668/From-MVC-to-Razor-Pages

Chapter 6: EventMakerRazorPage application

Now that you get a correct understanding of the RazorPages architectural pattern and how requests are processed, we are going to dig into coding our EventMaker application. In this chapter, we will implement the following user story "**as a traveler, I want to view all the events occurring in Denmark**".

Add a data model class

The model class represents the data of the app. It is common to use the model in relation to a data source (i.e. Database). We use Model objects to store, retrieve and update the data into a database. Unfortunately, in this section, we are not going to create any database. We are going to use a fake repository having a list of Event objects. The use of a relational Database through a data access layer (i.e. Entity Framework) is covered in Part II (second semester).

Create the Event Model class.

- Right-click the **RazorPagesEventMaker** project > **Add** > **New Folder**. Name the folder ***Models***.
- Right click on the *Models* folder. Select **Add** > **Class**. Name the class **Event**.
- Add the following **properties** to the Event class.

```
 6  namespace RazorPagesEventMaker.Models
 7  {
        8 references
 8      public class Event
 9      {
            5 references
10          public int Id { get; set; }
            5 references
11          public string Name { get; set; }
            5 references
12          public string Description { get; set; }
            5 references
13          public string City { get; set; }
            5 references
14          public DateTime DateTime { get; set; }
15      }
16  }
```

- **Id** field is unique for each Event object.
- **Name** is the name of the event.
- **Description** is the description about the event

- **City** is the city in which the event is occurring
- **DateTime** is the date and the time .

Create the FakeEventRepository.cs class.

- Right-click on the *Models* folder, then ------> **Add** -------> **Class**. Name the file **FakeEventRepository.*cs***.
- Add the following code in the **FakeEventRepository.cs** file.

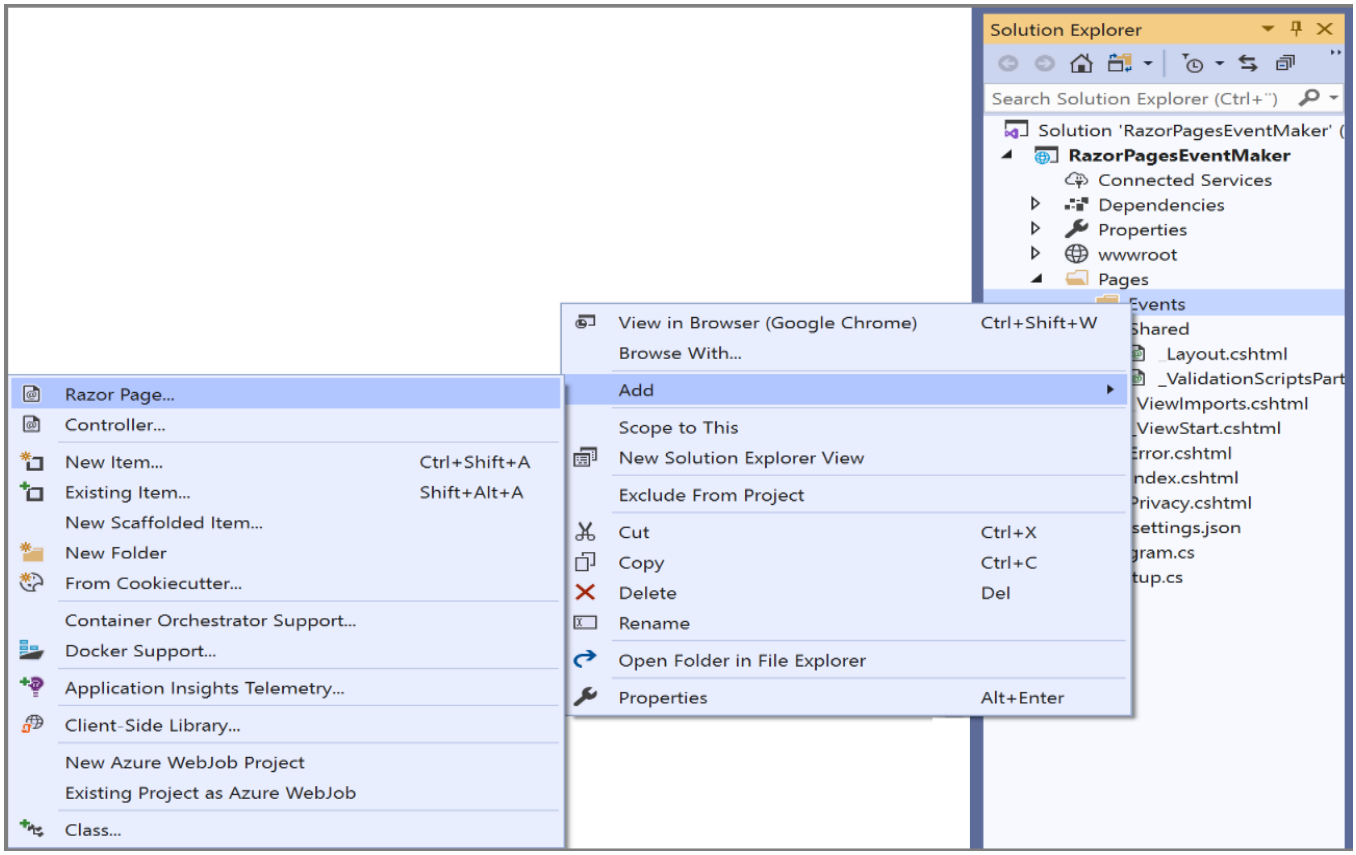
FakeEventRepository.cs*

```
8   public class FakeEventRepository
9   {
        7 references
10      private List<Event> events { get; }
        0 references
11      public FakeEventRepository()
12      {
13          events = new List<Event>();
14          events.Add(new Event() {Id = 1,Name = "Roskilde Festival", Description = " A lot of music",
15              City = "Roskilde", DateTime = new DateTime(2020, 6, 9, 10, 0, 0) });
16          events.Add(new Event(){ Id = 2, Name = "CPH Marathon", Description = " Many Marathon runners",
17              City = "Copenhagen", DateTime = new DateTime(2020, 3, 6, 9, 30, 0) });
18          events.Add(new Event() {Id = 3, Name = "CPH Distorsion", Description = " A lot of beers",
19              City= "Copenhagen",  DateTime = new DateTime(2019, 6, 4, 14, 0, 0) });
20          events.Add(new Event() {Id = 4, Name = "Demo Day", Description = "Project Presentation",
21              City = "Roskilde", DateTime = new DateTime(2020, 6, 9, 9, 0, 0)});
22          events.Add(new Event() {Id = 5, Name = "VM Badminton", Description = "Badminton",
23              City = "Århus",DateTime = new DateTime(2020, 10, 3, 16, 0, 0)});
24      }
        0 references
25      public IEnumerable<Event> GetAllEvents()
26      {
27          return events.ToList();
28      }
29  }
```

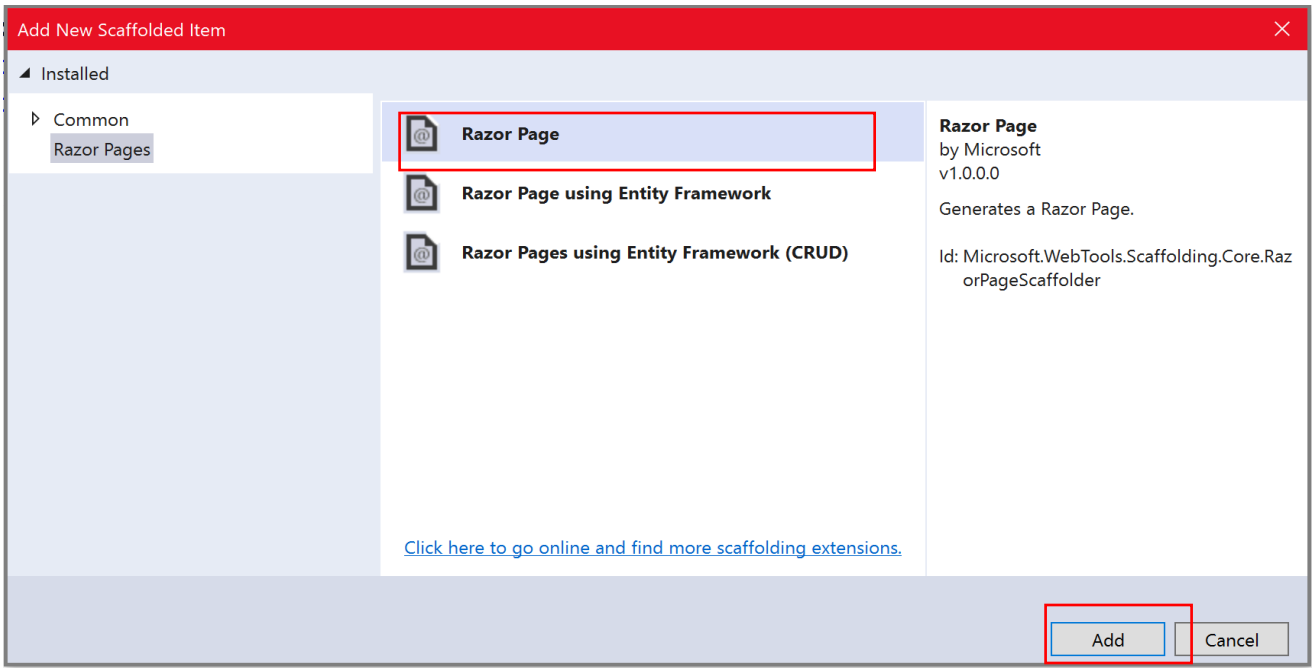
The FakeEventRepository.cs file contains the instance field "**events**" of type List<Event>. In the constructor, we have initialized our list with five Event objects. In the GetAllEvents method, we return the list of events. To display the list, we should add a Razor page. Let us add a new page.

- In the "**Pages**"folder, create a new folder and name it as "**Events**"

- In the folder **Pages/Events/** let us create a razor page , name it "**Index**" .



- Select "**Razor Page**". This will generate an empty Razor page.
- Then, click on the "**Add"** button**.**



As shown below, we want to create a razor page named "**Index**". At the same time, we want to generate PageModel class and use the layout page.

- Enter **Index** as the name of the new Razor Page
- Select "**Generate PageModel class**" and "**Use a layout page**" options.
- Click on the "**Add**" button.

The display template "**Index.cshtml**" along with its PageModel class "**Index.cshtml.cs**" **is** created.

They are shown below.

```
@page
@model RazorPagesEventMaker.IndexModel
@{
    ViewData["Title"] = "Index";
}

<h1>Index</h1>
```

```
    public class IndexModel : PageModel
    {
        0 references
        public void OnGet()
        {

        }
    }
```

Let us first explore and add code to the **PageModel** class. As we have seen earlier, all the properties that are passed to the page (display template) are first defined in this Page Model class. We added some code that you are probably familiar with. Let us explore this code.

```
public class IndexModel : PageModel
{
    private FakeEventRepository repo;
    2 references
    public List<Event> Events { get; private set; }
    0 references
    public IndexModel()
    {
        repo = new FakeEventRepository();
    }
    0 references
    public void OnGet()
    {
        Events = repo.GetAllEvents();
    }
}
```

In the IndexModel class we did the following :

- **Line 13**: we defined a FakeEventRepository instance field named “repo” to access the public methods in the FakeEventRepoistory class.
- **Line 17**: The instance field is initialized in the constructor, which is a good place to initialize instance fields.
- **Line 14**: We defined the “**Events**” property. This property will be accessed by the page content.

- **Line 21**: As we want to navigate to the Index page, we are sending a Get request and the OnGet handler method is invoked. Inside this method, we initialize our Events property by calling the GetAllEvents method implemented in the FakeEventRepository.cs class

In the display template (**page content**), we are using a table to display the list of Event objects. We choose Table because you may be familiar with tables. We cover tables in the first three chapters about HTML , CSS and Bootstrap. The code of the display template “Index.cshtml” is shown below.

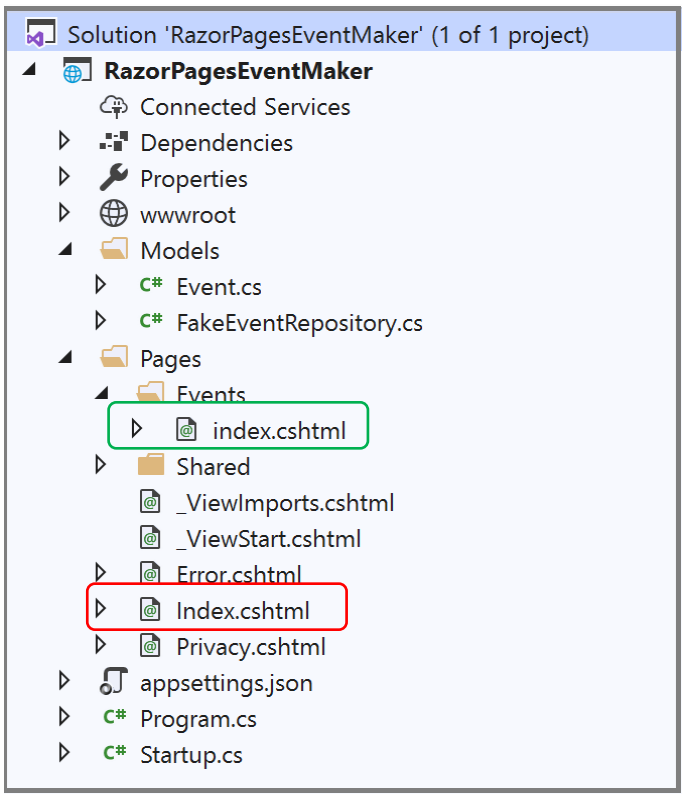
index.cshtml*

```
@page
@model RazorPagesEventMaker.IndexModel
@{
    ViewData["Title"] = "index";
}
<h1>List of events </h1>
<table class="table">
    <thead>
        <tr>
            <th>
                Id
            </th>
            <th>
                Name
            </th>
            <th>
                Description
            </th>
            <th>
                Place
            </th>
            <th>
                DateTime
            </th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Events)
        {
            <tr>
                <td>
                    @item.Id
                </td>
                <td>
                    @item.Name
                </td>
                <td>
                    @item.Description
                </td>
                <td>
                    @item.City
                </td>
                <td>
                    @item.DateTime
                </td>
            </tr>
        }
    </tbody>
</table>
```

Let us look at the most important parts of this code. First , notice how the C# code start with the @ sign.

- The **@model** directive specifies the type of the model passed to the Razor Page. The @model makes the list of Event objects "**Events**" property available to the page.
- We are using the bootstrap class Table to style the table.
- ***@foreach*** statement is used to loop through the collection of events (Model.Events) that was made available to the page. Notice the integration of C# code in the page using the **@** sign.
- **@item.Id** statement is used to get the value of the Id property of each item in the collection. That's it, very simple.

Only one small thing left before we run our application. Remember that when we run our application, the default Index page, located in the root folder "Pages" is invoked. However, **what if we want to display the Index page located in the "Pages/Events" folder instead?** Let us have a look at the file structure shown below. As you can see, the path to access the index page is **Events/index.** So let us place a link at the top menu that leads us to the index page.



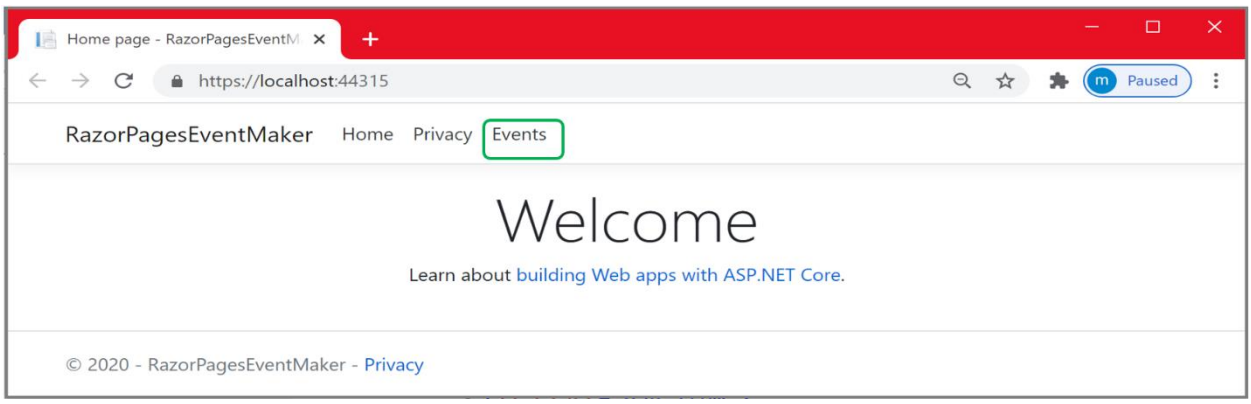
If you remember, the top menu is part of the **_Layout.cshtml** page. Let us use this page to add a link to the "Events/Index" page. The code below shows the new link.

```
<li class="nav-item">
    <a class="nav-link text-dark" asp-area="" asp-page="/Privacy">Privacy</a>
</li>
<li class="nav-item">
    <a class="nav-link text-dark" asp-area="" asp-page="Events/Index">Events</a>
</li>
```

The only interesting part about this line of code is the use of the **asp-page** tag helper to specify the page to navigate to . We have used this Tag helper earlier.

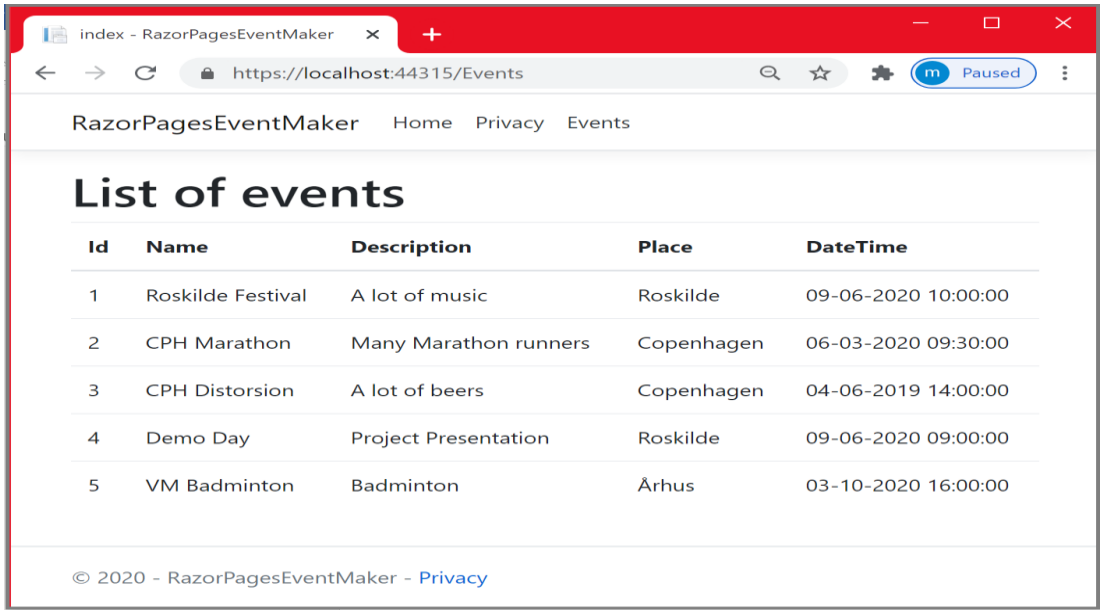
- Add this line of code and run the application.

As can be seen, notice the presence of the new link at the top menu



Clicking on the "Events" link, leads us to the Index page in the Events folder.

- **Click on** the "Events" link. The list of events is displayed on the page as shown below. It is very easy, **isn't it** ?



In the next chapter, we are going to implement the following user story "***as administrator, I will be able to create a new event***". We will start implementing the user story, and then later on, we dig into the validation concept. By validation, we check the validity of the data before it is sent to the server.

References

- https://www.learnrazorpages.com/razor-pages
- https://www.learnrazorpages.com/razor-pages/model-binding
- https://www.youtube.com/watch?v=oZvtODtG_Jk&list=PL6n9fhu94yhX6J31qad0wSO1N_rgGbOPV&index=7
- https://wakeupandcode.com/razor-pages-in-asp-net-core-3-1/#params

Hvad så Gutter Chapter 7: Create new events Oscar told me how to do this

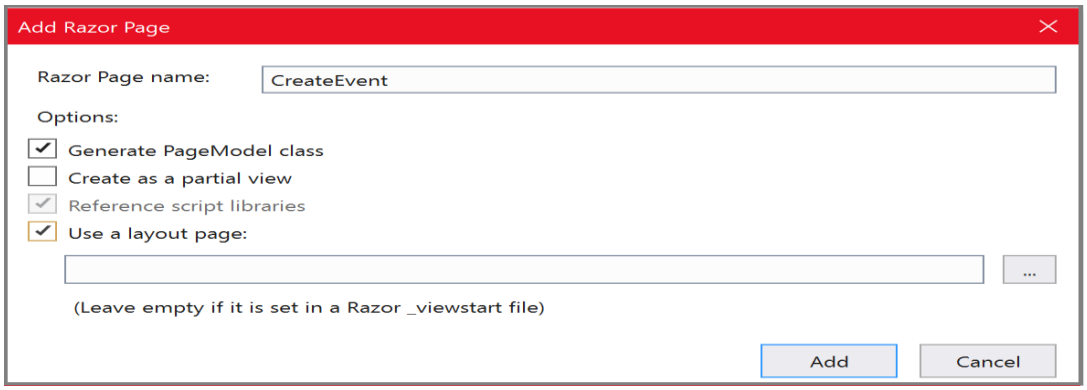
oscar pls

Introduction

In the previous chapter, we were able to display the list of Event objects. We usually want to perform **CRUD** operations on our model objects. By CRUD Operations, we mean **Create** Event objects, **Read** a specific Event object, **Update** an Event object or **Delete** an Event object. Precisely, when we create Event objects for example, we have to fill up a form with the required data and then submit the form to be processed at the server side.

In this chapter, we continue working on the app from the previous chapter. We are going to implement the user story: **“As an administrator, I will be able to create a new event”**. So let us start creating the razor page for creating a new event. To add a new razor page we follow the same procedure as we did for creating the Index.cshtml razor page.

- In the “**Events**” folder, create a new razor page and name it “**CreateEvent**” as shown below.
- Click **Add**



The following display template “**CreateEvent.cshtml**” and its corresponding **PageModel** class are created.

CreateEvent.cshtml*

```
1  @page
2  @model RazorPagesEventMaker.CreateEventModel
3  @{
4      ViewData["Title"] = "CreateEvent";
5  }
6  <h1>CreateEvent</h1>
```

CreateEvent.cshtml.cs*

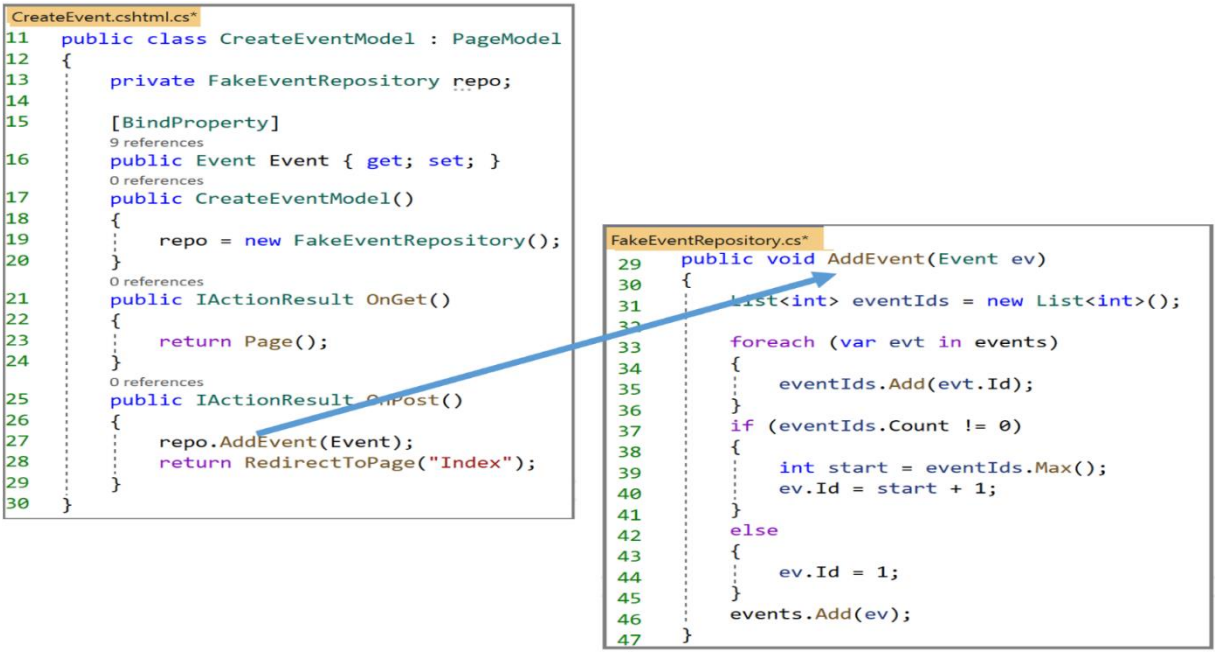
```
10     public class CreateEventModel : PageModel
11     {
           0 references
12         public void OnGet()
13         {
14
15         }
16     }
```

The CreateEvent.cshtml.cs PageModel class

Let us start with the PageModel **CreateEvent.cshtml.c**s file. Before digging into the code, let us explain what is needed to implement this functionality. This is going to be helpful to understand the code for creating a new event .

In the Index page, we need a link that leads us to the "**CreateEvent**" page. As mentioned in the previous chapter, clicking on this link, makes the **OnGet** method be invoked. Note that nothing to be initialized in the **OnGe**t because we have nothing to display. We get an empty form. **However**, when we fill up and submit the form, the **OnPost** method is going to be invoked this time. This is because we are sending data to the server. We should then provide the new event object to the **OnPost** method. This is done by defining an **Event** object property to which we bind the data submitted in the form. In the **OnPost** method, we use a reference to the **FakeEventRepository** class to access the code for implementing the addition of the new Event object to the list.

This may be confusing at first, but I promise that you'll quickly get the hang of it when looking at the code. The code below shows the "**CreateEvent"** PageModel class along with the implementation for adding the new Event object to the list.



Let us examine the code in these two files:

In the CreateEventModel, you may recognize the following :

- We created a FakeEventRepository reference and the Event property defined to catch the data retrieved from the form fields. The property is decorated with the [**BindProperty**] attribute to tell model binding to target the public Event property. Note that properties are bound for HTTP Post requests by default.
- What about the OnGet method? When is it invoked ? simply when we navigate to this page. Let us look at the OnGet code. This method returns an ***IActionResult*** type. Inside this action method, we are calling the Page() method that renders the actual Razor page. So it will simply display the "CreateEvent" page.

Note that the following line of code will also work , **why ?:**

```
public void OnGet(){ }
```

- **What about the OnPost handler method**? Of course, this is invoked when submitting the form. We use the repo reference to call the AddEvent method whose implementation is shown on the right hand side.
- Then we call the **RedirectToPage** method to redirect the user to the Index page.
- In the AddEvent method , we did the following:
  - ✔ **Line 33-36** : we use "**foreach**" to loop through the collection and we populate a list of int with the ids of the items in the collection.
  - ✔ **Line 37- 41** : if List<int> eventIds is not empty, we get the maximum integer number(Max) and we set the id of the new created event to Max +1
  - ✔ **Line 42-45** : otherwise, if the list is empty, we set the id to 1,
  - ✔ **Line 46**: We add the Event object to the list

Rules

- When the page is first navigated to, the "OnGet " is invoked because the **HTTP** `GET` verb was used for the request.
- When the "Save" button is pressed, the form is posted using the post method and the `OnPost()` handler is invoked resulting in adding a new event object to the list as shown below.

Now let us look at the display template, **the CreateEvent.cshtml** file**.**

As mentioned before, we are going to use a form for creating Event objects.

The “CreateEvent.cshtml” code is shown below.

CreateEvent.cshtml*

```
@page
@model RazorPagesEventMaker.CreateEventModel
@{
    ViewData["Title"] = "CreateEvent";
}
<h1>CreateEvent</h1>
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div class="form-group">
                <label asp-for="@Model.Event.Name" class="control-label"></label>
                <input asp-for="@Model.Event.Name" class="form-control" />
            </div>
            <div class="form-group">
                <label asp-for="@Model.Event.Description" class="control-label"></label>
                <input asp-for="@Model.Event.Description" class="form-control" />
            </div>
            <div class="form-group">
                <label asp-for="@Model.Event.City" class="control-label"></label>
                <input asp-for="@Model.Event.City" class="form-control" />
            </div>
            <div class="form-group">
                <label asp-for="@Model.Event.DateTime" class="control-label"></label>
                <input asp-for="@Model.Event.DateTime" class="form-control" />
            </div>
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>
<div>
    <a asp-page="Index">Back to List</a>
</div>
```

Let us look at the part of the code that deserves some explorations:

- The HTML part of the page includes a form that uses the **POST** method that will probably initiate a Post request and invoke the OnPost handler method.

- Notice that we removed the auto-generated **"Id"** group from this Markup because the id is automatically incremented in the code (see implementation later on in this chapter)

- We added the following link to be able to navigate back to the Index page. The path is simply "Index" because the Index page and the "CreateEvent" page are in the same folder.

```
<div>
    <a asp-page="Index">Back to List</a>
</div>
```

- The only code you are not familiar with is the following:

```
<div class="form-group">
    <label asp-for="@Model.Event.Name" class="control-label"></label>
    <input asp-for="@Model.Event.Name" class="form-control" />
</div>
```

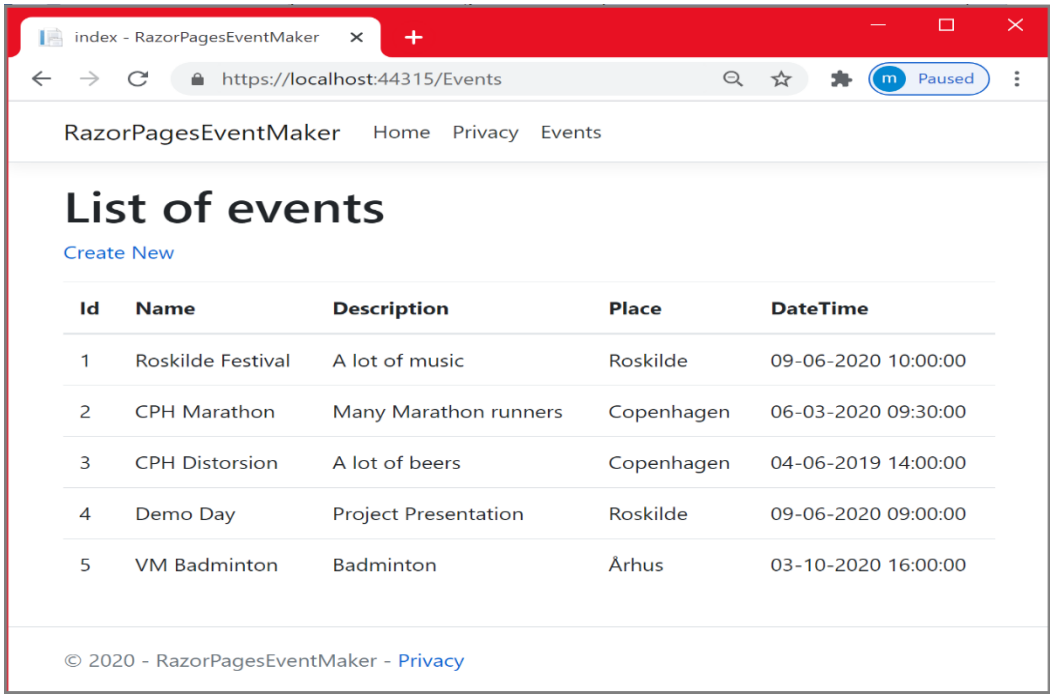
We are using the **Label asp-for** tag helper. Tag Helpers components are executed at the server to generate the corresponding html code. The most important benefit they provide over the use of HTML elements is the strong typing with the model properties.

We are missing a small thing. It is a link to navigate to the "CreateEvent" page. The appropriate place for this link is in the Index page at the top. The code for this link is shown below.

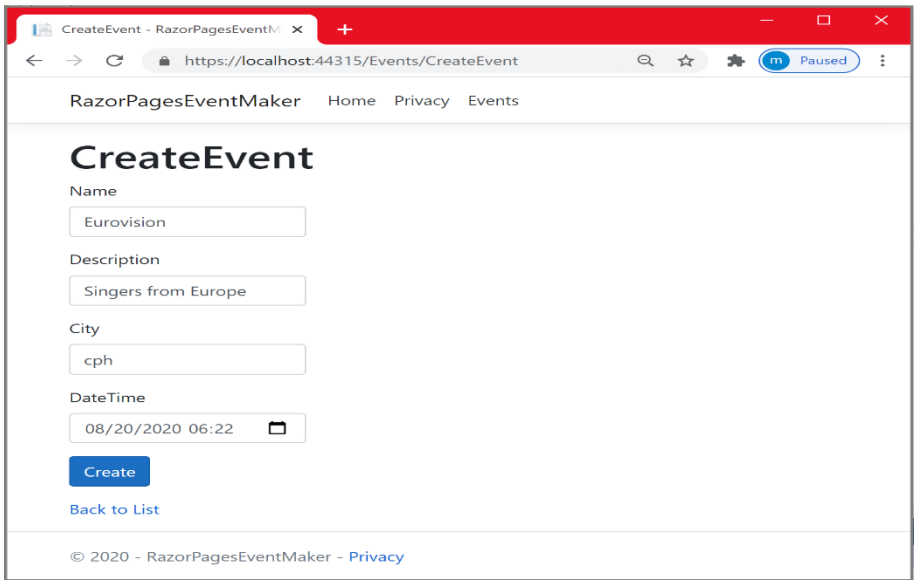
index.cshtml*

```
@page
@model RazorPagesEventMaker.IndexModel
@{
    ViewData["Title"] = "index";
}
<h1>List of events </h1>
<p>
    <a asp-page="CreateEvent">Create New</a>
</p>
```

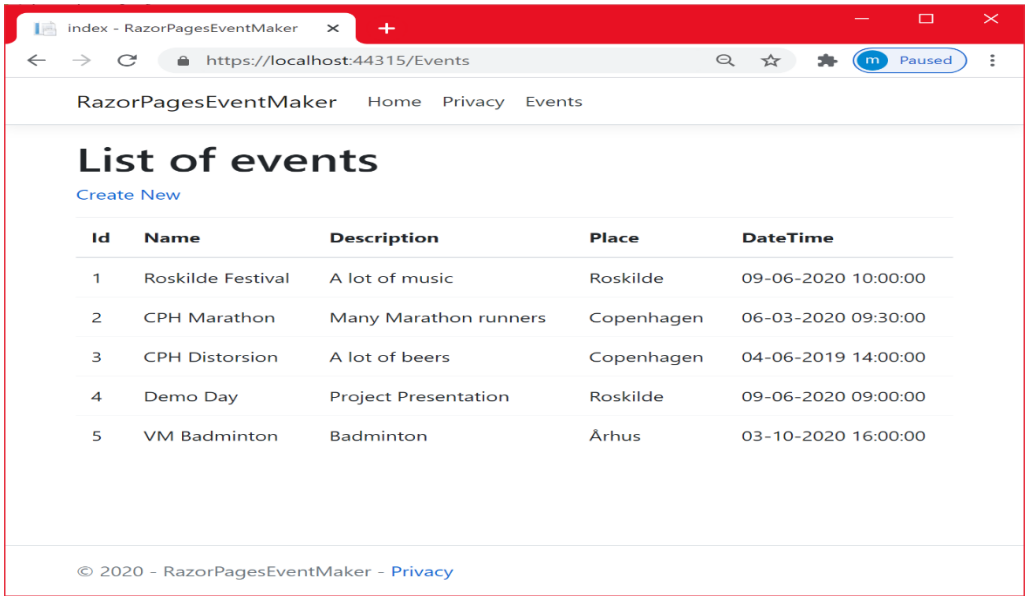
Now that the link is added , let us run the application and Click on the "**Events**" link.



- Click on the "**Create New**" link, the "CreateEvent" page is displayed. Notice the presence of the "**Back to List**" link that redirects us to the Index page.

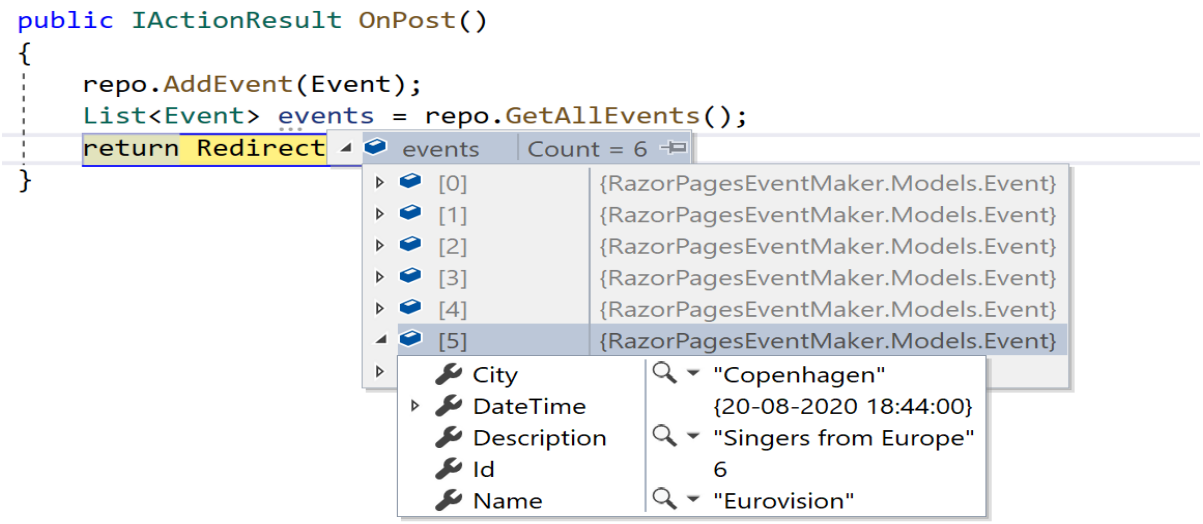


- Fill up the form and click on the "Create" button to submit the form. The output is shown below.



As you can see, the new Event object is not displayed on the list. **What is the problem then**?

Let us track the problem by putting *a breakpoint* at the **OnPost** method. The following code snippet shows the Event objects of the list along with the new Event object. The new event is indeed added to the list but it is not displayed on the Index page. What could be the issue?



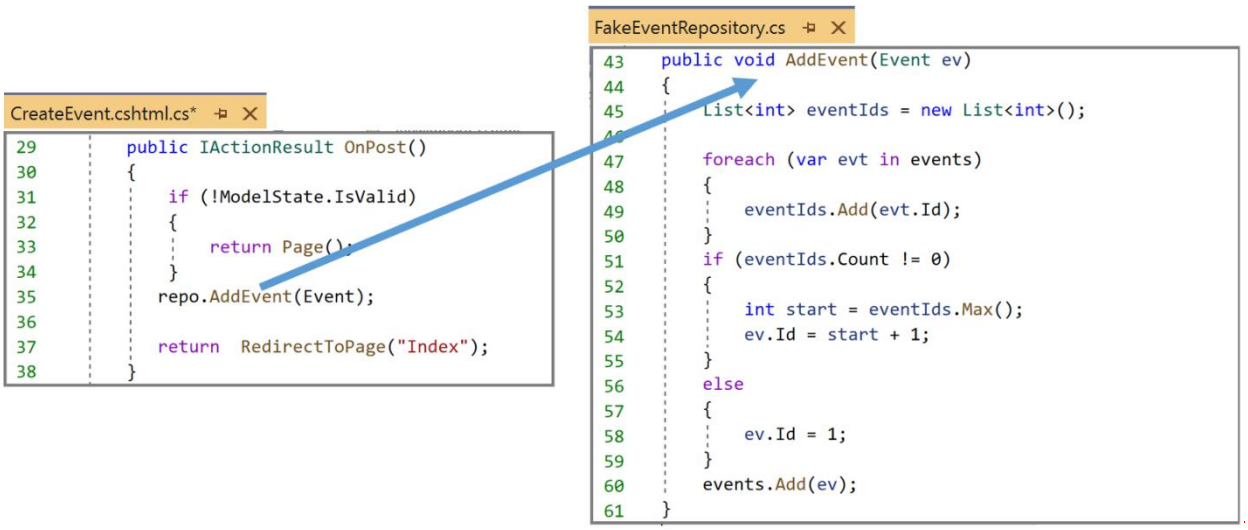
In the next chapter, we are going to figure out why the new event is not displayed. Another aspect that we did not take into consideration when creating the Event object is whether the entered data is valid or not. What if the user enters a past date? What if the email has not the right format? …etc.

References

- https://www.learnrazorpages.com/razor-pages/tag-helpers/
- https://www.learnrazorpages.com/razor-pages/forms

Chapter 8 : Validation && Singleton Design Pattern

We ended the previous chapter with an issue in the implementation of creating a new event. The issue was that the new created Event object is added to the list but it is not displayed.



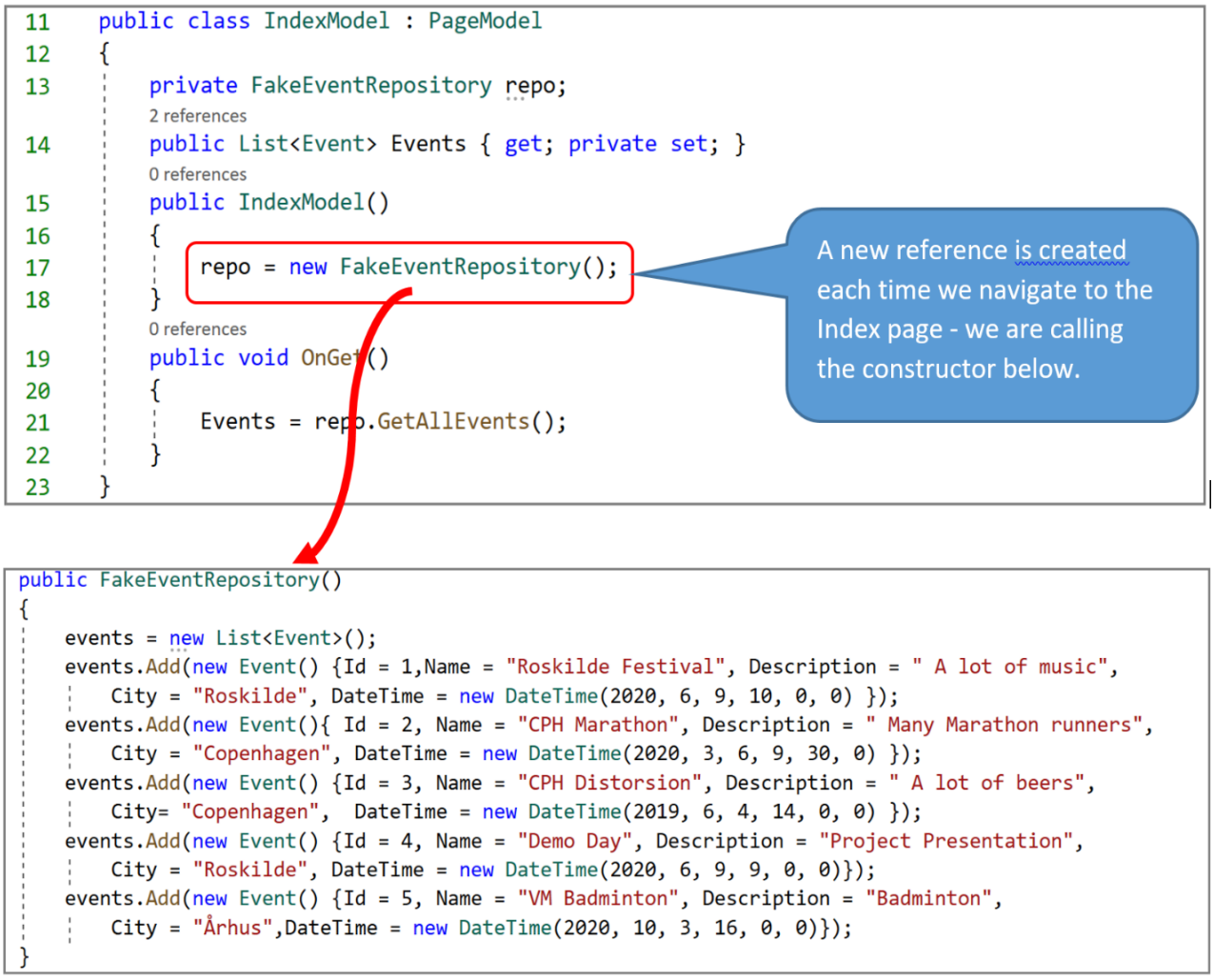
What is the problem then?

To answer this question, you need to understand how Razor Pages application processes a user request, and this takes us to the chapter about architecture. We know that , each time a page is requested, the request is intercepted by the PageModel of the page and mapped to one of its handler methods. That means, each time a request is sent, ***the constructor*** of the PageModel is called.

With this in mind, let us look at the code below. After adding a new Event object to the list, we redirect the user to “Index.cshtml” page and this is where the problem happened.

```
public IActionResult OnPost()
{
    repo.AddEvent(Event);
    return RedirectToPage("Index");
}
```

What is the problem: When redirecting the user to Index.cshtml, we are requesting this razor page. Thus, the **constructor of IndexModel** is called. As can be seen from the code snippet from the figure below, when the constructor is called, we are creating a new instance of the FakeEventRepository class. The consequence of this is that even if we added a new Event object to the list, requesting the Index.cshtml page will initialize the ***repo instance field*** with a new FakeEventRepository instance , calling the FakeEventRepository constructor and initializing the list with only the 5 objects.



How to solve this problem?

We should make sure that only one instance of the FakeEventRepository class is created. This way, we can keep track of our collection (i.e items added, deleted…etc.). To ensure only one instance is created, we should implement the **Singleton design pattern**. The code implementation of the Singleton design pattern is shown in the figure below. Later on in chapter 10 about dependency injection, we are going to avoid writing all this code about the Singleton design pattern. We are going to configure our FakeEventRepository as a service and we are going to inject this service

into our PageModel class constructor using dependency injection. Dependency injection is one of the features that makes ASP.NET Core suitable for building robust , extensible web apps.

In the code below, we made our constructor private. This ensures creating only one instance of FakeEventRepository class. To instantiate an instance, we use the **Instance** public property. Notice how in the get method, we create the unique instance only if it is null (**not created yet**).

FakeEventRepository.cs*

```
8 references
public class FakeEventRepository
{
    9 references
    private List<Event> events { get; }
    private static FakeEventRepository _instance;
    2 references
    private FakeEventRepository()
    {
        events = new List<Event>();
        events.Add(new Event() {Id = 1,Name = "Roskilde Festival", Description = " A lot of music",
            City = "Roskilde", DateTime = new DateTime(2020, 6, 9, 10, 0, 0) });
        events.Add(new Event(){ Id = 2, Name = "CPH Marathon", Description = " Many Marathon runners",
            City = "Copenhagen", DateTime = new DateTime(2020, 3, 6, 9, 30, 0) });
        events.Add(new Event() {Id = 3, Name = "CPH Distorsion", Description = " A lot of beers",
            City= "Copenhagen",  DateTime = new DateTime(2019, 6, 4, 14, 0, 0) });
        events.Add(new Event() {Id = 4, Name = "Demo Day", Description = "Project Presentation",
            City = "Roskilde", DateTime = new DateTime(2020, 6, 9, 9, 0, 0)});
        events.Add(new Event() {Id = 5, Name = "VM Badminton", Description = "Badminton",
            City = "Århus",DateTime = new DateTime(2020, 10, 3, 16, 0, 0)});
    }
    1 reference
    public static FakeEventRepository Instance
    {
        get
        {
            if (_instance == null)
            {
                _instance = new FakeEventRepository();
            }
            return _instance;
        }
    }
```

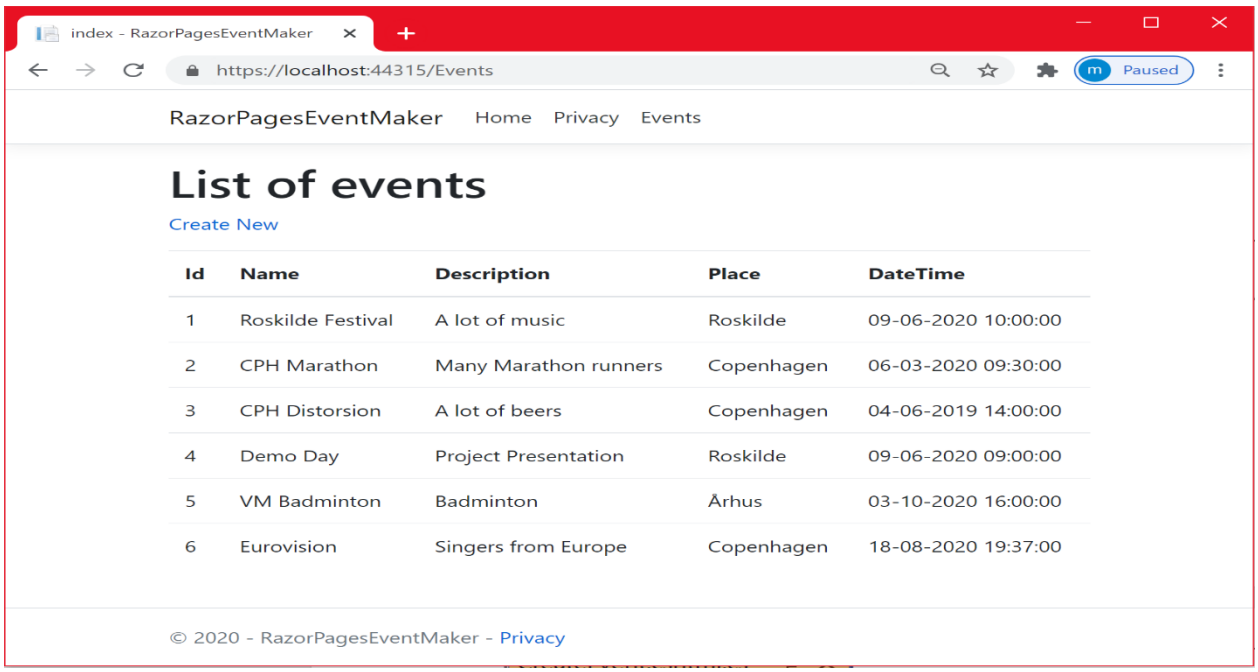
In the figure below, notice how we instantiate the single reference to the FakeEventRepository class in the PageModel constructor. The code snippet shows how we instantiate the FakeEventRepository reference in the

constructor of the CreateEventModel class. The same applies to the IndexModel class and eventually all other PageModel classes that need a reference to the FakeEventRepository.

CreateEvent.cshtml.cs*

```
11    public class CreateEventModel : PageModel
12    {
13        private FakeEventRepository repo;
14        [BindProperty]
          9 references
15        public Event Event { get; set; }
          0 references
16        public CreateEventModel()
17        {
18            repo = FakeEventRepository.Instance;
19        }
          0 references
20        public IActionResult OnGet()
21        {
22            return Page();
23        }
          0 references
24        public IActionResult OnPost()
25        {
26            repo.AddEvent(Event);
27            return RedirectToPage("Index");
28        }
29    }
```

Now that we applied the singleton design pattern , let us run the application and create a new Event object.



As can be seen in the figure above, the new created Event object is displayed on the list. You can add more objects to the list without any problem.

Data Validation

Even if we are able to add Event objects to the list, another problem arises. It may probably happen that the user enters an invalid data (invalid type, invalid format…etc.). We need a mechanism that allows us to validate user data. For example, we probably want the date of the new event to be in the future date. We do so to avoid bothering the server side with invalid data or to avoid letting invalid data get into the database!. In addition to that , we want to enhance the user experience by displaying adequate error messages to the user in case or error.

ASP.Net Core provides validation support using what we call **Data Annotations. The dataAnnotations** namespace provides many Built-in validation attributes such as: **Required**, **RegularExpression, Range**. **MinLength**, **MaxLength, Compare** (for example when comparing password and confirm password)….etc.

These attributes specify validation rules that should be fulfilled by the model properties. Decorating a property with the [**Required**] attribute indicates that the property must have a value. [**RegularExpression**] specifies what characters can be input. The [**Range**] attribute specifies a lower and upper limit for the value of the property.

To implement validation, let us start by looking at the Event model, decorating the properties with some validation attributes. The code below shows how we decorate the Event model properties with validation attributes. We have decorated the "**Name**"property with the [**Required**] attribute because it is mandatory. We used the [**StringLength**] attribute to restrict the length of the "**City**" property to max 18 characters. The **DateTime** property is required and must be in the range 1/8/2020 - 1/8/2021 and so on…

```
Event.cs
 9   public class Event
10   {
         9 references
11       public int Id { get; set; }
12
13       [Display(Name = "Event Name")]
14       [Required(ErrorMessage = "Name of the Event is required"), MaxLength(30)]
         9 references
15       public string Name { get; set; }
         9 references
16       public string Description { get; set; }
17
18       [Required]
19       [StringLength(18, ErrorMessage = "Name of the city can not be longer than 18 chars")]
         9 references
20       public string City { get; set; }
21
22       [Required(ErrorMessage = "The date is required")]
23       [Range(typeof(DateTime), "10/1/2020", "10/1/2021",
24       ErrorMessage = "Value for {0} must be between {1} and {2}")]
         9 references
25       public DateTime DateTime { get; set; }
26   }
```

We need to check whether the submitted model data is valid or not when submitting the form. As shown below, In the OnPost method, we check whether the state of the model is valid or not.

```
24   public IActionResult OnPost()
25   {
26       if (!ModelState.IsValid)
27       {
28           return Page();
29       }
30       repo.AddEvent(Event);
31       return RedirectToPage("Index");
32   }
```

Very Important: In the case, the state of the model is not valid, it is important to use the statement : **return Page()**. This makes the actual page display the error messages. Otherwise, you may call the page constructor , which creates a new instance of the page and you miss the error messages.

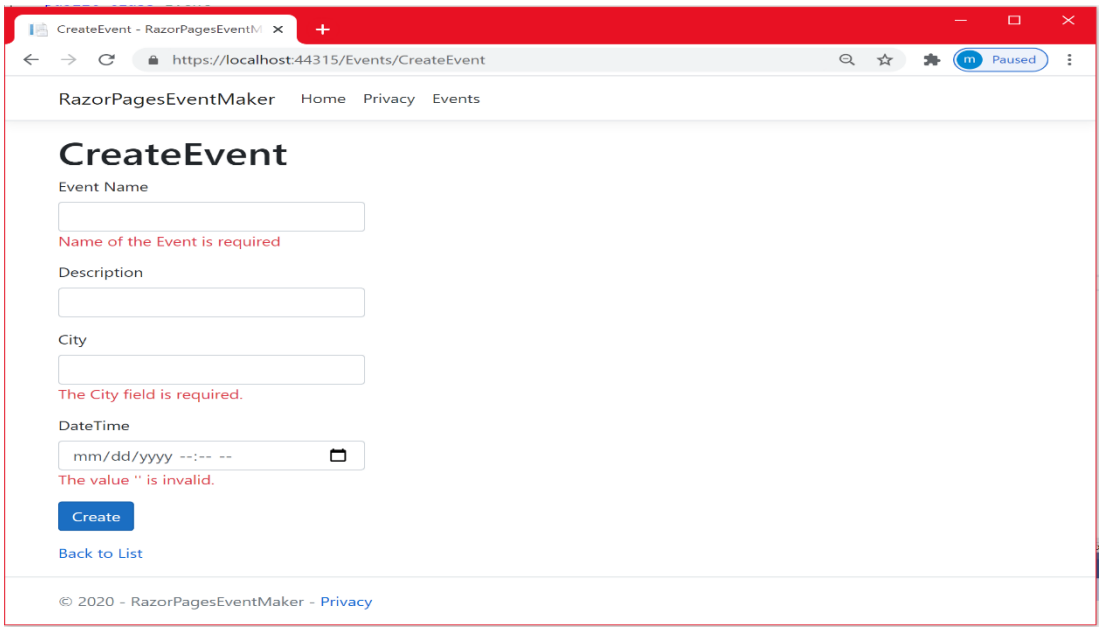
Let us explore the code a little bit more. When the form is submitted, the **OnPost** method is executed. In the code snippet above, we first check whether the validation succeeded or not, by calling the ModelState.IsValid method. The (**ModelState.IsValid**) statement evaluates any validation attributes that have been applied to the Event object. If the validation is successful, we add the Event object to the list and we redirect the user to the index page. However, if ModelState.IsValid is evaluated to false, the form is not posted to the server and the call of the Page() method makes the "CreateEvent" be redisplayed with the error messages for eventual correction of the data.

We have not finished yet and there is something left. We need a way to display error messages on the page for the user. The following markup shows how we add validation tag helpers to the "Name" property. The ***asp-validation-for*** validation tag helper displays the error message for a single property of the model. It is generally placed after an input tag helper for the same property. As can be seen below, we used a span element to display the error message.

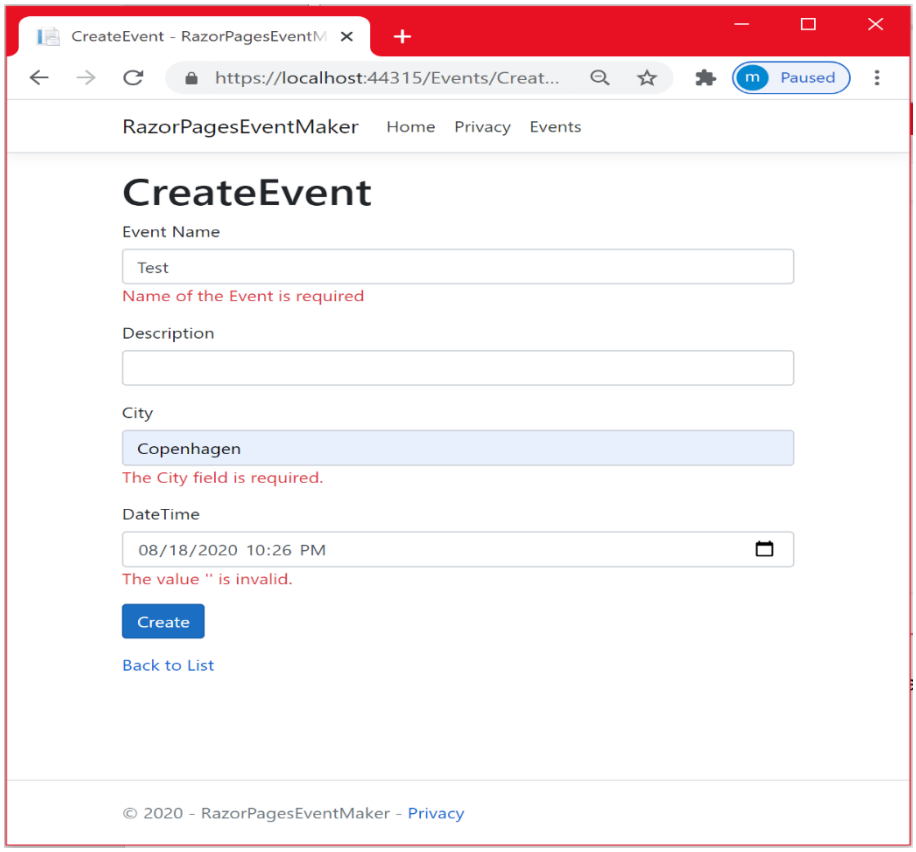
```
<div class="form-group">
    <label asp-for="Event.Name" class="control-label"></label>
    <input asp-for="Event.Name" class="form-control" />
    <span asp-validation-for="Event.Name" class="text-danger"></span>
</div>
```

You probably recognized the Bootstrap class "**text-danger**" to display the error message in red (see Chapter 3 about Bootstrap)

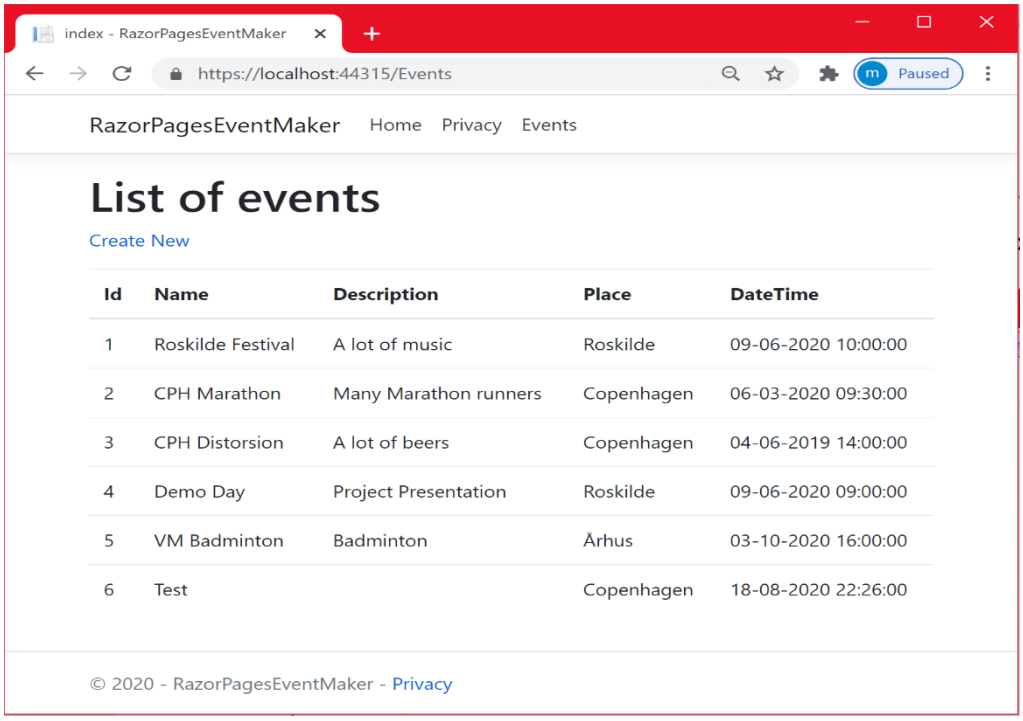
After applying the validation tag helper to all the properties, run the application and with no data , submit the form by clicking on the "**Create**" button. Notice how the form has automatically rendered an appropriate validation error message for each field that requires a valid input value. As can be seen, all the error messages are displayed under the corresponding fields.



Let us fill up validate data and submit the form again. Note that "Description" is not required.



As can be seen, the validation passes and the new Event object is created.



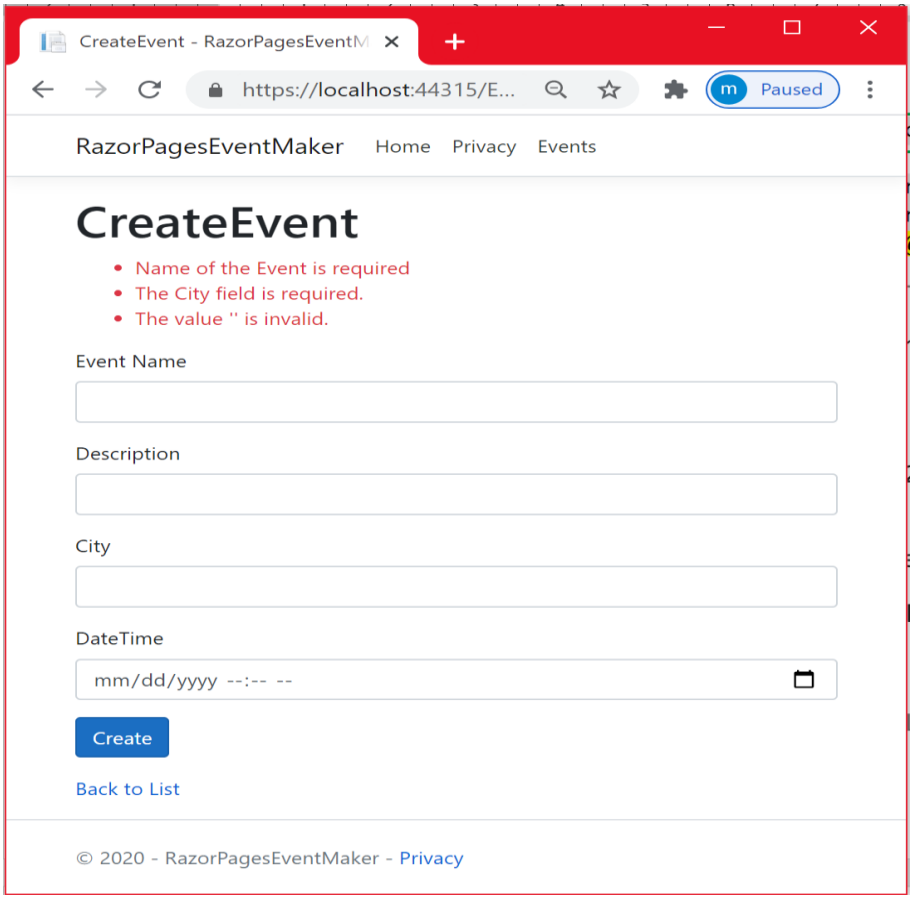
One more thing before we close this chapter is that sometimes you want to display the error messages at the top. For that, we use the so-called **Validation Summary Tag Helper : `asp-validation-summary.` The validation Tag Helper** is used to display a summary of validation error messages. It is normally placed at the top of the form as shown below.

```
<h1>CreateEvent</h1>
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly " class="text-danger"></div>
            <div class="form-group">
                <label asp-for="@Model.Event.Name" class="control-label"></label>
                <input asp-for="@Model.Event.Name" class="form-control" />
                <span asp-validation-for="@Model.Event.Name" class="text-danger"></span>
            </div>
```

In the code snippet above, the asp-validation-summary has the value "ModelOnly", which displays the Model-level validation only.

The attribute **`asp-validation-summary`** can have 2 other values:

- ValidationSummary.**All**: It displays both the property and the model level validations.
- ValidationSummary.**ModelOnly**", which displays the Model-level validation only.
- ValidationSummary.**None:** It does not perform any validation.



References

- https://docs.microsoft.com/en-us/aspnet/core/tutorials/razor-pages/validation?view=aspnetcore-3.1&tabs=visual-studio
- https://www.learnrazorpages.com/razor-pages/tag-helpers/validation-message-tag-helper
- https://docs.microsoft.com/en-us/aspnet/core/mvc/views/tag-helpers/built-in/?view=aspnetcore-3.1

Chapter 9 : Routing in ASP.NET Core Razor Pages

Default Routing with Razor Pages

As mentioned before in chapter 5, the first component that receives the incoming requests from the user (the browser) is the PageModel class and a handler method within the PageModel is invoked. The handler method manipulates the data model (i.e connection to a database to manipulate data), passes the model to the page, then finally displays the Page.

The question is, how the request is mapped to the requested page or resource in general? There should probably exist a mechanism for mapping a request to the right Razor page. This is called **Routing**.

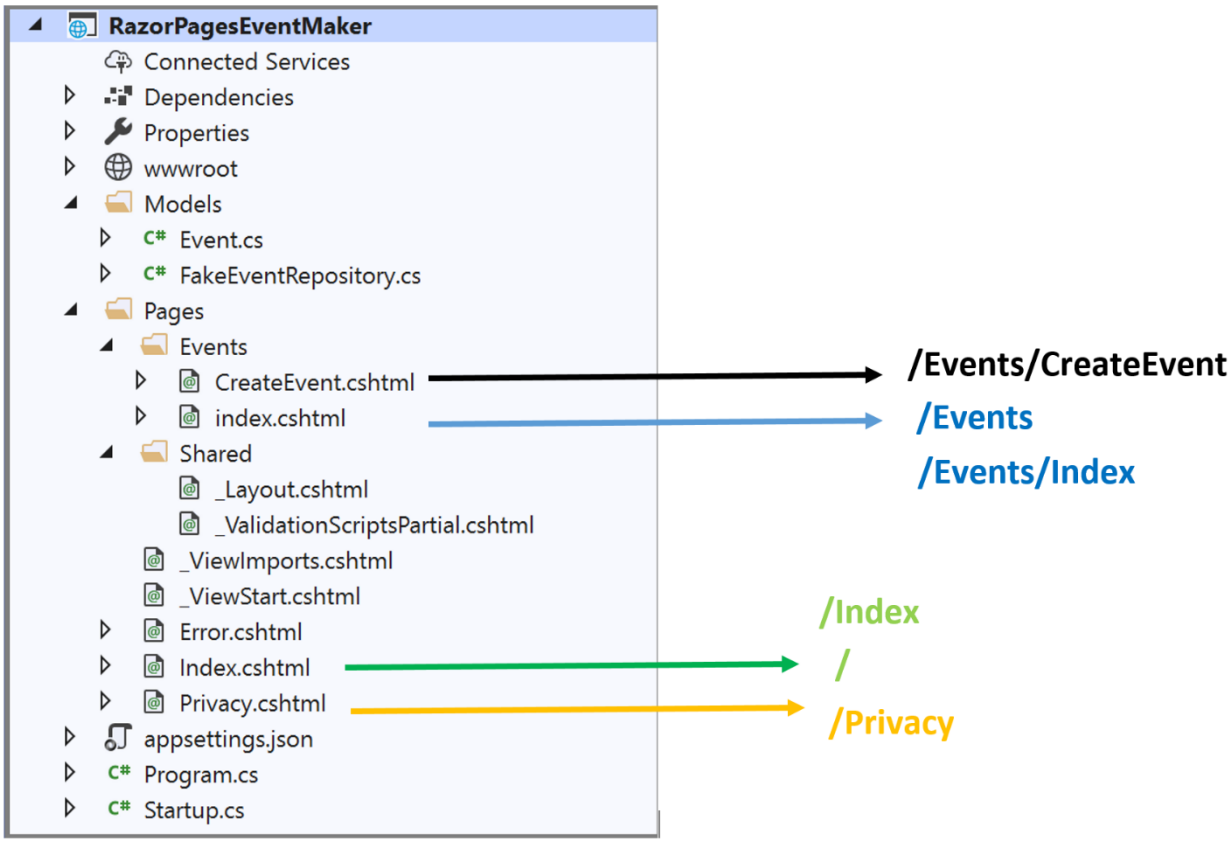
In short, Routing is the mechanism of matching URLs to Razor pages. In ASP.NET Core, this is achieved by a middleware component (request pipeline). The middleware is composed of many subcomponents having a specific task each. All these components are configured in the Configure method of the Startup.cs file. One important component is the one that ensures the application is secure via authentication (checking whether the user is the right person) and authorization (checking whether the user is allowed to access specific parts of the application). These security features are the subject of another chapter, at the moment , let us focus on routing.

How URLs are mapped?

By default, Routing is based on matching **URLs to file paths**, where the folder Pages is by default the root Razor Pages folder and the *Index* Razor page is considered the default page in any folder or any subfolder having an Index page.

Considering the file structure shown below, the Index.cshtml Razor page within the Pages folder is going to be displayed when we run the application. This page matches the root URL https://localhost:port/ and the URL https://localhost:port/index because the Index page is the default one. We say that Index.cshtml has two routes.

How to display the Index.cshtml that is inside the Events folder? We specify its URL taking into consideration the file path. We can reach this file by browsing to both : https://localhost:port/Events and https://localhost:port/Events/index because as mentioned before, Index is considered as the default page in any folder.



How to change the default routing ?

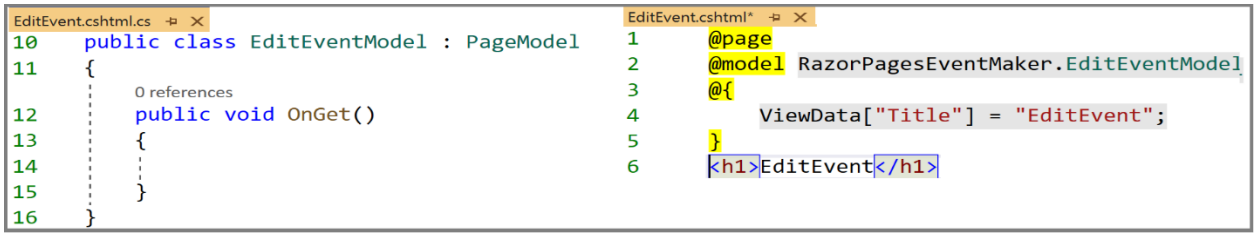
Overriding : We need to tell the system to use another route to reach the Index razor pages in the Events folder using the **@page** directive.

For example,Let us suppose that the @page directive of your index.cshtml page is set to: **@page "/Events" .** If you navigate to the following URL : ***root:port/Events*** , you will display the Index page.

Edit an Event

Let us start implementing the user story "***As an administrator, I will be able to edit a specific event***". Let us first create the Razor Page "**EditEvent**" in the

"**Events**"folder. The procedure is the same, no need to show the details again. The "EditEvent.cshtml" and its PageModel class "EditEventModel" are created.

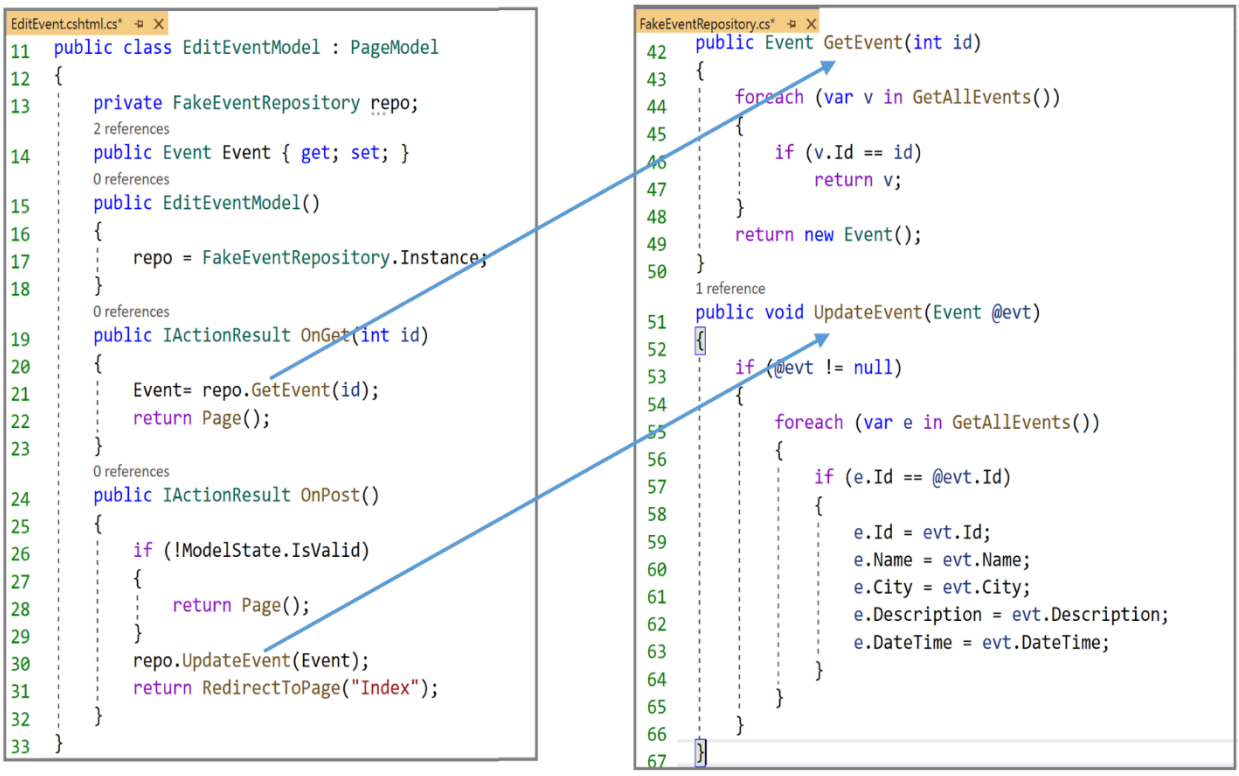


As we did before let us start with the EditEventModel class. But, let me first remind you what Editing is. Editing is to get a specific Event object, make some changes to it, and then post the object

We need to pass the id (which is unique for each Event object) as the parameter of the OnGet method. This is going to be used to fetch the right Event from the FakeEvenRepository list.

Do we need the OnPost method? Yes of course. When we make some changes to the event object, we should first check that the data is valid and then save the change to the list.

Let us dig into the code. The figure below shows the "**EditEventModel**" class along with the "**GetEvent**" and the "**UpdateEvent**" methods implementations of the FakeEventRepository.



```
EditEvent.cshtml.cs*
11  public class EditEventModel : PageModel
12  {
13      private FakeEventRepository repo;
        2 references
14      public Event Event { get; set; }
        0 references
15      public EditEventModel()
16      {
17          repo = FakeEventRepository.Instance;
18      }
        0 references
19      public IActionResult OnGet(int id)
20      {
21          Event= repo.GetEvent(id);
22          return Page();
23      }
        0 references
24      public IActionResult OnPost()
25      {
26          if (!ModelState.IsValid)
27          {
28              return Page();
29          }
30          repo.UpdateEvent(Event);
31          return RedirectToPage("Index");
32      }
33  }
```

```
FakeEventRepository.cs*
42  public Event GetEvent(int id)
43  {
44      foreach (var v in GetAllEvents())
45      {
46          if (v.Id == id)
47              return v;
48      }
49      return new Event();
50  }
    1 reference
51  public void UpdateEvent(Event @evt)
52  {
53      if (@evt != null)
54      {
55          foreach (var e in GetAllEvents())
56          {
57              if (e.Id == @evt.Id)
58              {
59                  e.Id = evt.Id;
60                  e.Name = evt.Name;
61                  e.City = evt.City;
62                  e.Description = evt.Description;
63                  e.DateTime = evt.DateTime;
64              }
65          }
66      }
67  }
```

We have seen most of this code. It is self-explanatory. What is worth looking at is how to pass a parameter to the **OnGe**t method ? But, before answering this question, let us code the “EditEvent.cshtml” display template. The code of this page is shown below.

As you can see, it is not surprising that this code is similar to the code of “**CreateEvent**” page. Indeed, we are displaying the details of a specific item, we are making some changes and then we are saving the item. So there is no need to explain the code another time. It is an exercise for you to try to explore and understand the code.

EditEvent.cshtml*

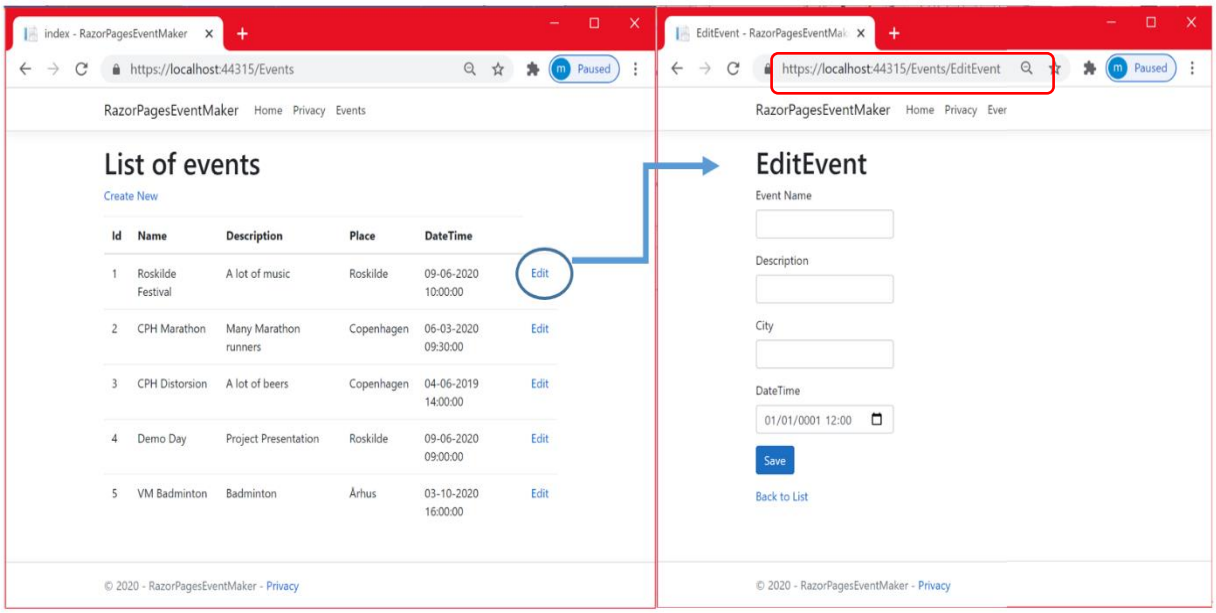
```
@page
@model RazorPagesEventMaker.EditEventModel
@{
    ViewData["Title"] = "EditEvent";
}
<h1>EditEvent</h1>
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <input type="hidden" asp-for="@Model.Event.Id" />
            <div class="form-group">
                <label asp-for="@Model.Event.Name" class="control-label"></label>
                <input asp-for="@Model.Event.Name" class="form-control" />
                <span asp-validation-for="@Model.Event.Name" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="@Model.Event.Description" class="control-label"></label>
                <input asp-for="@Model.Event.Description" class="form-control" />
                <span asp-validation-for="@Model.Event.Description" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="@Model.Event.City" class="control-label"></label>
                <input asp-for="@Model.Event.City" class="form-control" />
                <span asp-validation-for="@Model.Event.City" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="@Model.Event.DateTime" class="control-label"></label>
                <input asp-for="@Model.Event.DateTime" class="form-control" />
                <span asp-validation-for="@Model.Event.DateTime" class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Save" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>
<div>
    <a asp-page="index">Back to List</a>
</div>
```

Again, we are missing a link to this page. Let us add the link in the “**Index**” page as shown below.

```
30  <tbody>
31      @foreach (var item in Model.Events)
32      {
33      <tr>
34          <td>
35              @item.Id
36          </td>
37          <td>
38              @item.Name
39          </td>
40          <td>
41              @item.Description
42          </td>
43          <td>
44              @item.City
45          </td>
46          <td>
47              @item.DateTime
48          </td>
49          <td>
50              <a asp-page="EditEvent">Edit</a>
51          </td>
52      </tr>
53      }
54  </tbody>
```

We have added a simple link using the **asp-page** Tag helper, which was used in many occasions earlier.

Run your application and select the item to be edited.



Notice that the links work well, however we could not pass data from the **index** page to the "**EditEvent**" page. It is obvious because we did not tell the system

which item was selected. ***Is it routing issue***? Of course it is. Look at the URL on the right hand side. We navigated to **localhost:44315/Events/EditEvent**. What we want is to be able to navigate to something like **localhost:port/Events/EditEvent/id** **(not really this but like this)** , where the id is what uniquely identifies the selected item. This way, we can pass all the selected item data to the "EditEvent" page. I will show two ways to do that using:

- ✔ ***QueryString***
- ✔ ***Route parameters***

Query string

We want to pass the id of the selected Event object from the **Index** page to the **EditEvent** page as part of the Query string.

> **Reminder** : A query string is the portion of a URL where data is passed to a web application by assigning values to specified parameters.

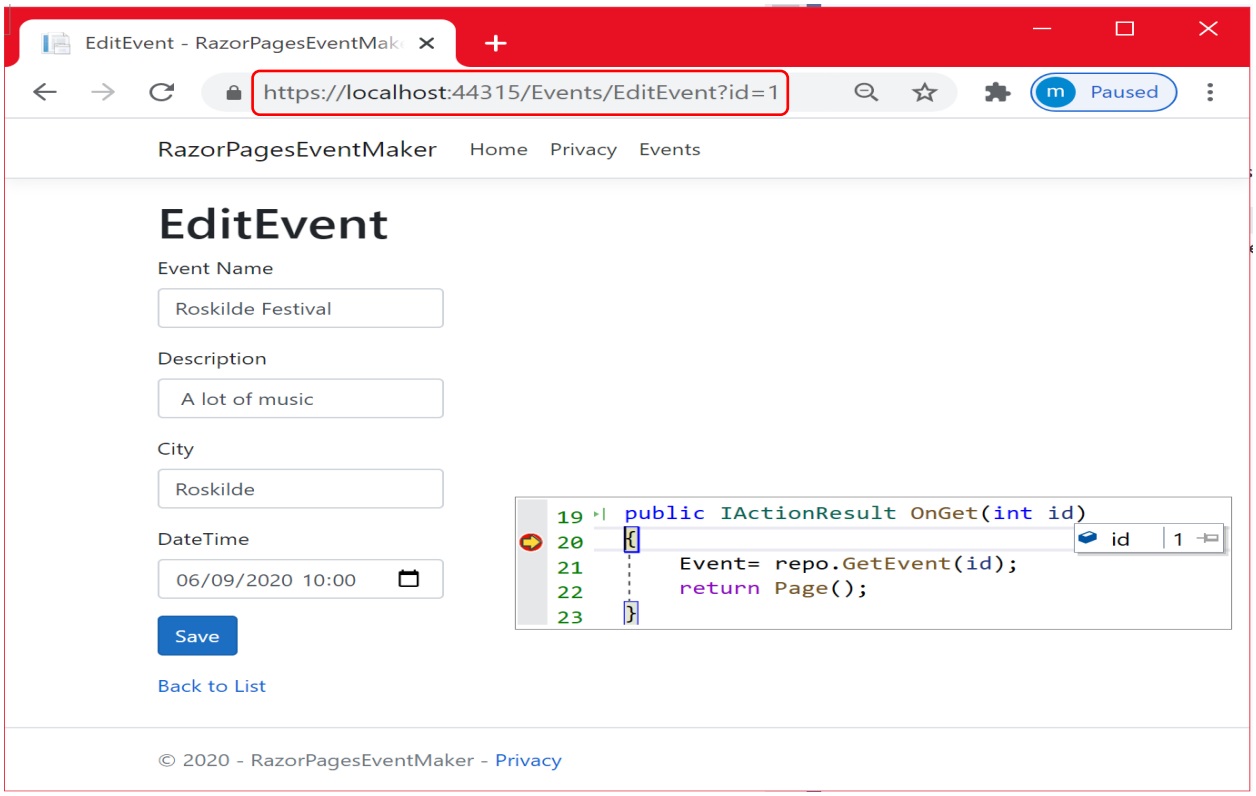
We can use either the tag-helper **asp-route-id** or **asp-route-name**. Let us use the first option. In the Edit link located in the Index page (shown below), add the asp-route-id tag helper as shown below.

```
<td>
      <a asp-page="EditEvent" asp-route-id="@item.Id">Edit</a>
</td>
```

When we click on this link, two **magic** things happen:

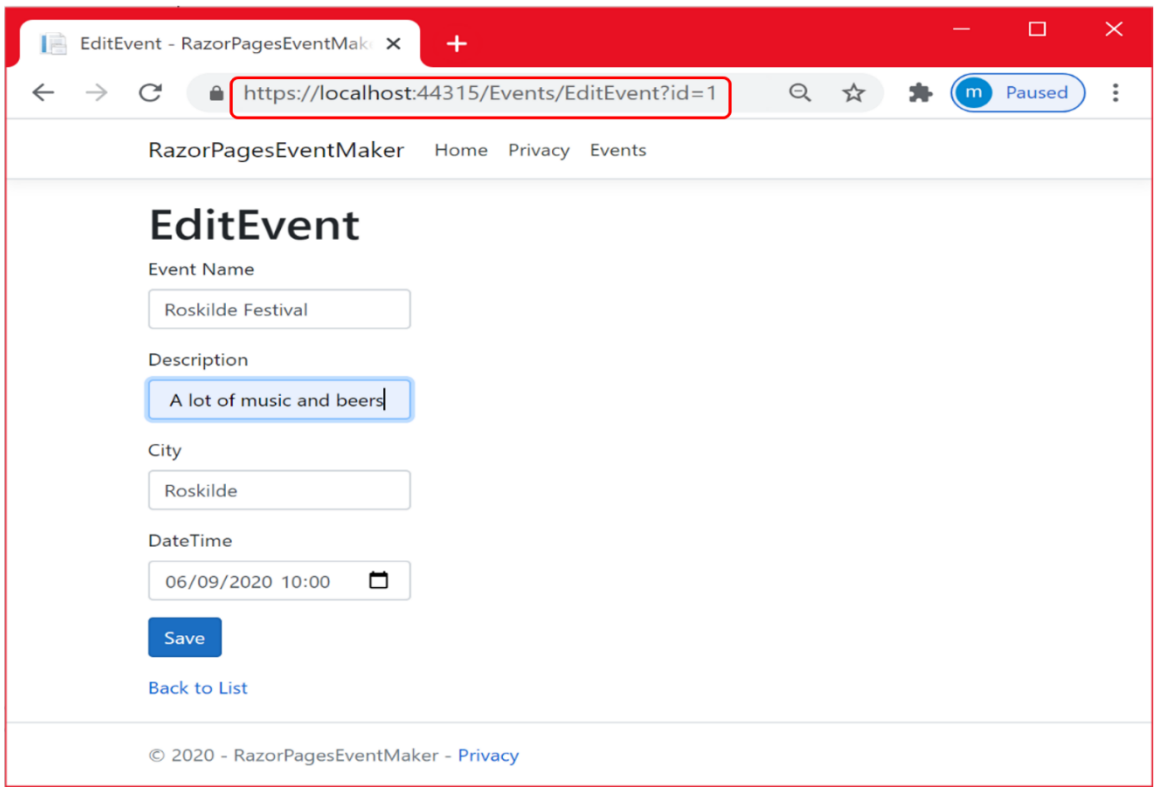
- ✔ By default the id of the selected item is passed as Query-String parameter.
- ✔ The Model-binding engine maps the Id in the query-string parameter value to the id parameter of the OnGet handler method

Let us check these two assumptions. Run the application and select the item whose id is 1.

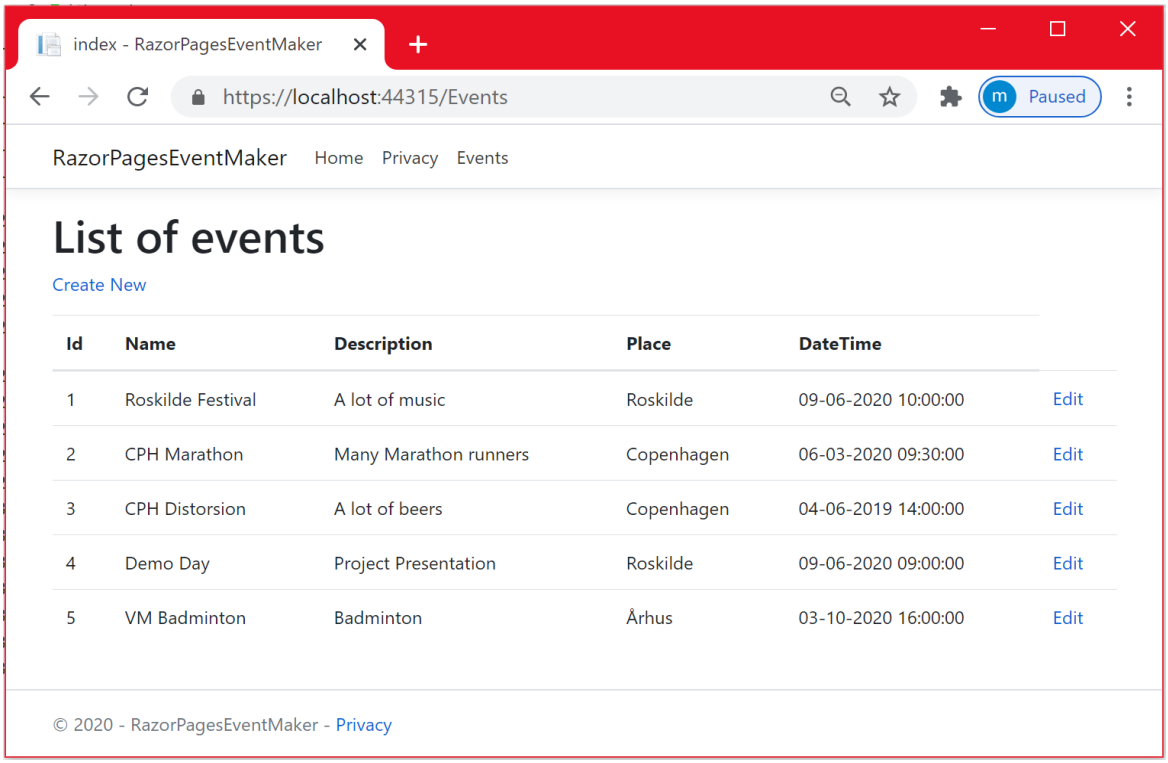


As can be clearly seen , the id of the selected item is passed as Query-String parameter (see the URL) and as a parameter to the OnGet method as well.

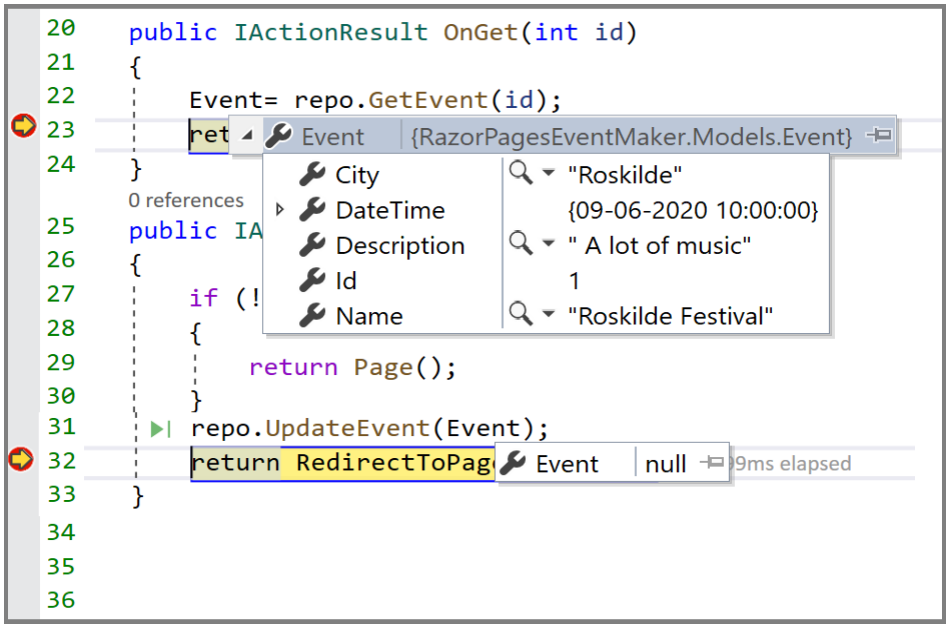
Let us make some changes to the description property (a lot of music ---> a lot of music and beers)



The figure below shows the output when clicking on the "**Save**" button.



As you can see , the selected item 1 is not updated. Let us investigate the cause of this issue by placing a break-point in the OnGet method and another one in the OnPost method. This is illustrated below.

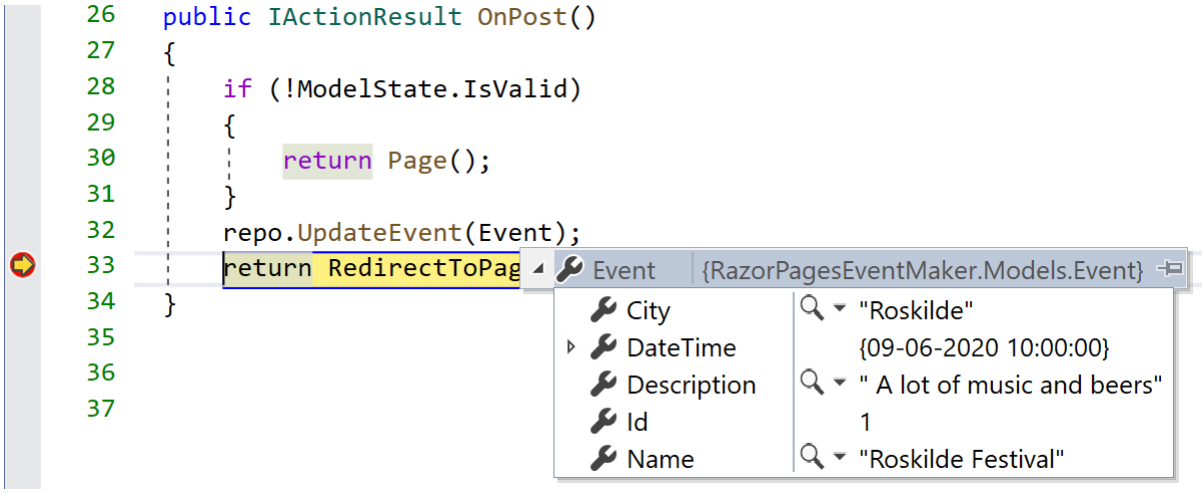


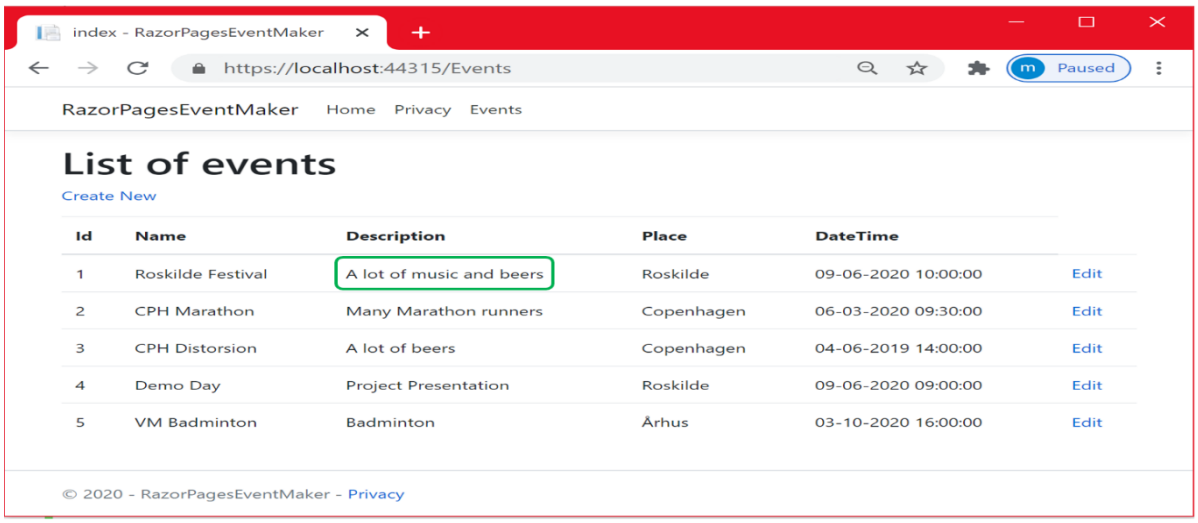
As you can see, there is no problem with the OnGet method. The issue is with the OnPost method. The **Event** property in the EditEvent Page Model is not correctly initialized. We are willing to perform data binding from the **EditEvent.cshtml** page to this **Event** property . It is a binding problem. We are not able to bind the data submitted by the form to the Event property . The problem can be solved simply decorating the Event property with the [**BindProperty**] attribute. [**BindProperty**] allows you to bind properties for **HTTP POST** requests by default. You can also bind properties to the Get request. In the next chapter, we will look at that case.

Important : Note that no decoration attribute is needed if you want to pass data from the PageModel to the page content. The binding is done automatically.

❑ Decorate the event property with the [**BindProperty**] attribute and run the application.

As shown below, the issue with the **OnPost** method is resolved. The data is updated correctly.





Route parameters

We have seen how to pass the id value from the Index page to the EditEvent page via the Query-string. This method is vulnerable to security issues as data is passed as part of the URL. This time, let us investigate how we can pass parameter data as part of the route and not through the query string. As we

will see, the method is straightforward. You need just to specify a route parameter that has the same type and the same id name as the handler method parameter.

> **Reminder**: When we introduced Razor pages in Chapter 1, we mentioned that a Razor Page starts with the **@page** directive and the only content that can be placed after this directive is the route parameter(s). We also mentioned previously in this chapter that we want to navigate to something like this **localhost:port/Events/EditEvent/id.**

I think it will make sense to add the id as a route parameter to the **EditEvent** page. Let us do that. We just need to specify a route template at the @page directive as shown below.

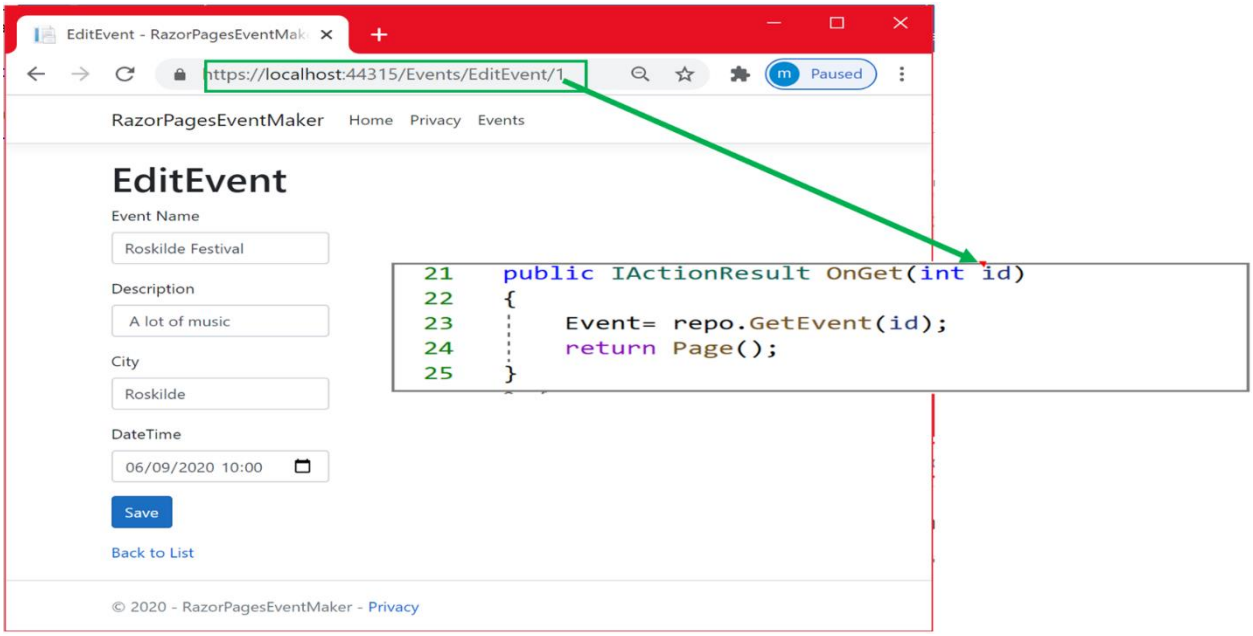
EditEvent.cshtml*

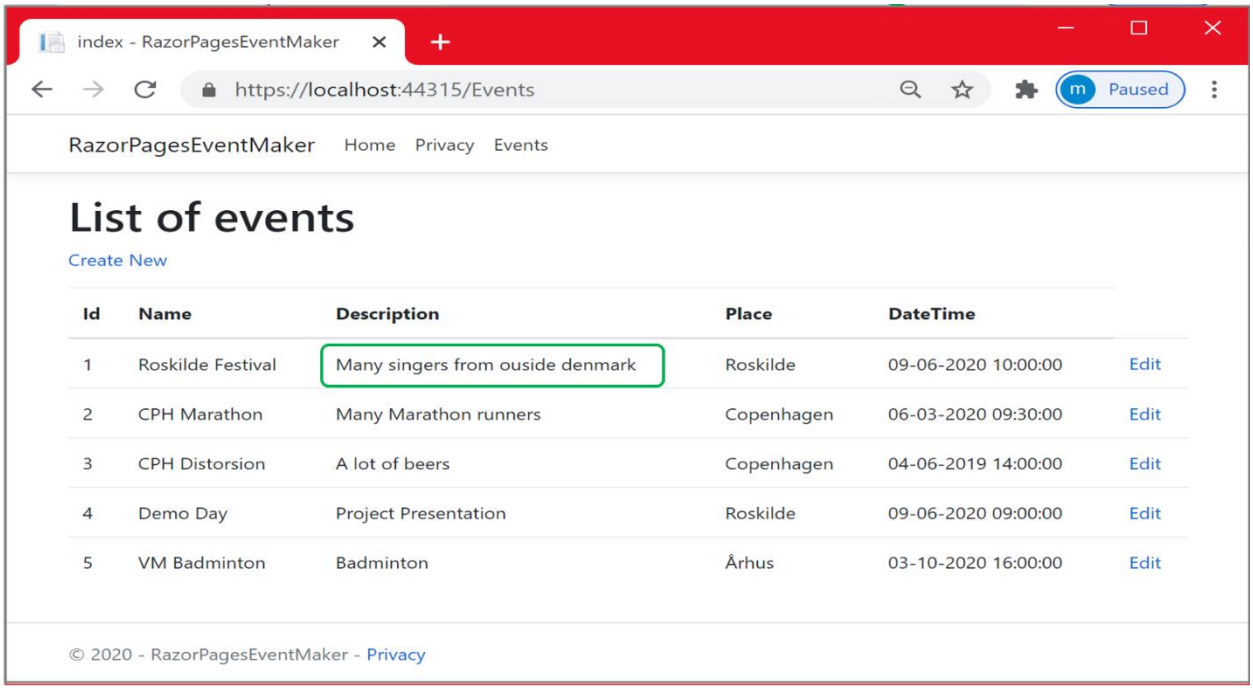
```
1  @page "{id:int}"
2  @model RazorPagesEventMaker.EditEventModel
3  @{
4      ViewData["Title"] = "EditEvent";
5  }
```

> **WAW**, that's it. This statement tells the page that it is expecting a mandatory id as a parameter. The id is initialized with the value provided by the selected item in the index page (through the link). This is illustrated below

Let us run the application.

As shown below, notice the URL used to reach the Events/EditEvent page for a specific Event object. This time, the URL is no longer Events/EditEvent but ***Events/EditEvent/id,*** adding the id that was included in the route template.





Behind the scene, the Model-binding is automatically mapping the id in the route parameter to the id passed as a parameter to the **OnGet** handler method.

More about routing

With ASP.NET Core, routing is a very important topic that deserves more exploration. In this section, you are going to learn more about routing. You have just seen how we used the id as a route parameter.

- ✔ You could make this id optional by just appending the ? sign to the id (like **id?).** This means that the id? Can take the null value. The id can be characters, numbers…etc.
- ✔ If the id is optional and you do not specify any id in the URL, you will get an empty item because the system does not know which page to display.

Constraint parameters

- Sometimes, you want to restrict the type of the route parameter. For example, you want to only pass integers as a parameter. This is illustrated in the figure below.

```
1    @page "{id:int}"
2        @model RazorPagesEventMaker.EditEventModel
3        @{
4            ViewData["Title"] = "EditEvent";
5        }
```

- If you add a constraint and the id is optional, the **?** is always added at the end of the parameter as illustrated below.

```
@page "{id:int?}"
    @model RazorPagesEventMaker.EditEventModel
    @{
        ViewData["Title"] = "EditEvent";
    }
```

- Sometimes you want to restrict the parameter to be bigger than 0 for example, because your primary keys start at 1, you want to avoid 0. You can do it by specifying a minimum value using the min constraint as shown below.

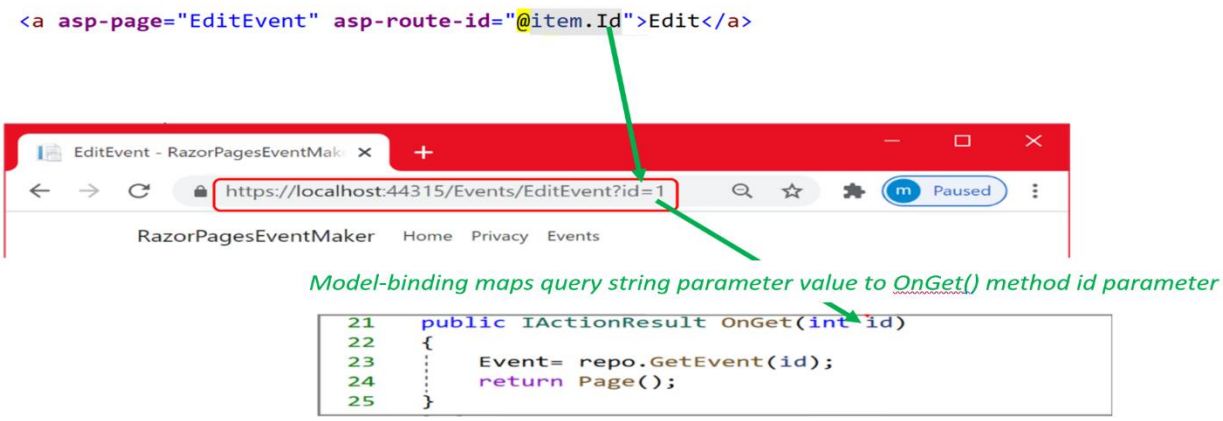
```
@page "{id:min(1)}"
    @model RazorPagesEventMaker.EditEventModel
    @{
        ViewData["Title"] = "EditEvent";
    }
```

- You can even define a maximum as follows

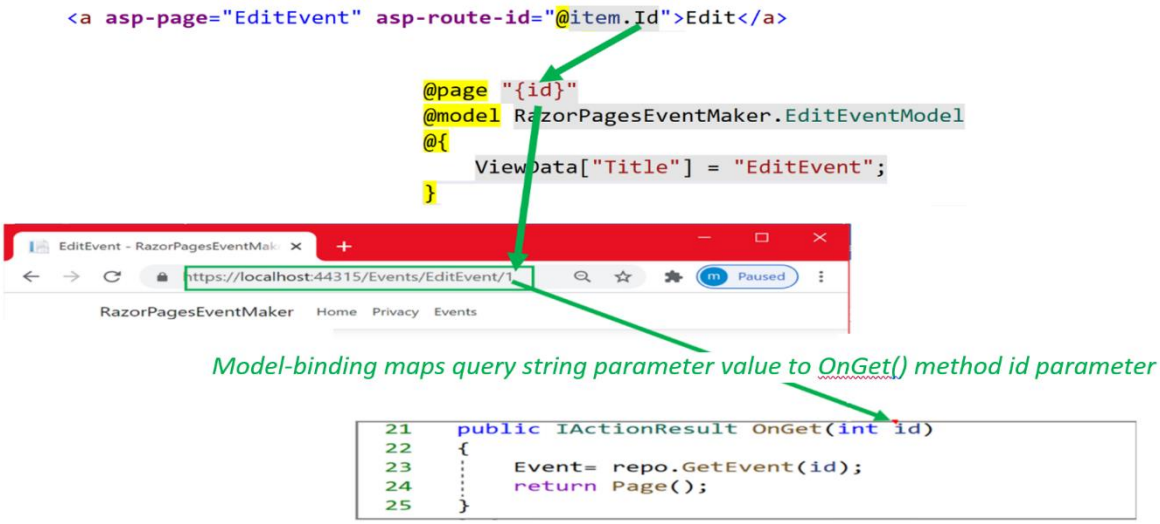
```
@page "{id:min(1):max(10)}"
    @model RazorPagesEventMaker.EditEventModel
    @{
        ViewData["Title"] = "EditEvent";
    }
```

Let us close this chapter summarizing what we did in this chapter. We have seen how to pass the id from the Index page to the EditEvent page. This is shown in the two following illustrations:

Using QueryString



Using Route parameter



References

- https://docs.microsoft.com/en-us/aspnet/core/tutorials/razor-pages/da1?view=aspnetcore-3.1
- https://www.learnrazorpages.com/razor-pages/routing
- https://wakeupandcode.com/razor-pages-in-asp-net-core-3-1/#params

Chapter 10: Dependency injection

In Chapter 8, we used the Singleton Design pattern to make sure that only one instance of the FakeEventRepository is created. ASP.NET Core made the singleton design pattern implementation more easier through Dependency injection. As we will see, we do not need to write a lot of code. As we will see, we are going to get rid of all the code implementing the singleton Design pattern.

In this chapter, We start with an introduction to the Dependency Injection concept. We will implement this concept in our application to replace the old code. Then, as we get the hang of almost everything, we will be able to implement many user stories this time. We consider implementing the following user stories:

- ✔ As a user , I will be able to filter the events based on the city name
- ✔ "as administrator, I will be able to remove an event".
- ✔ As an administrator, I will be able to see the details of a specific event.

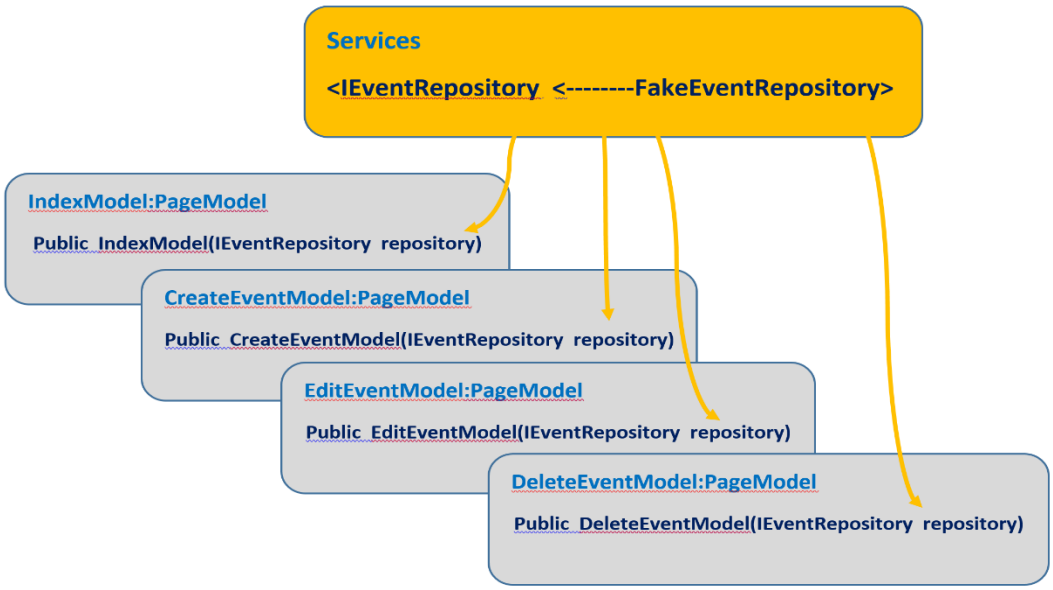
What is Dependency Injection ?

Dependency Injection is one of the important features that make ASP.NET Core applications easier to maintain, easier to extend...etc. Indeed, ASP.NET Core has built-in support for dependency injection.

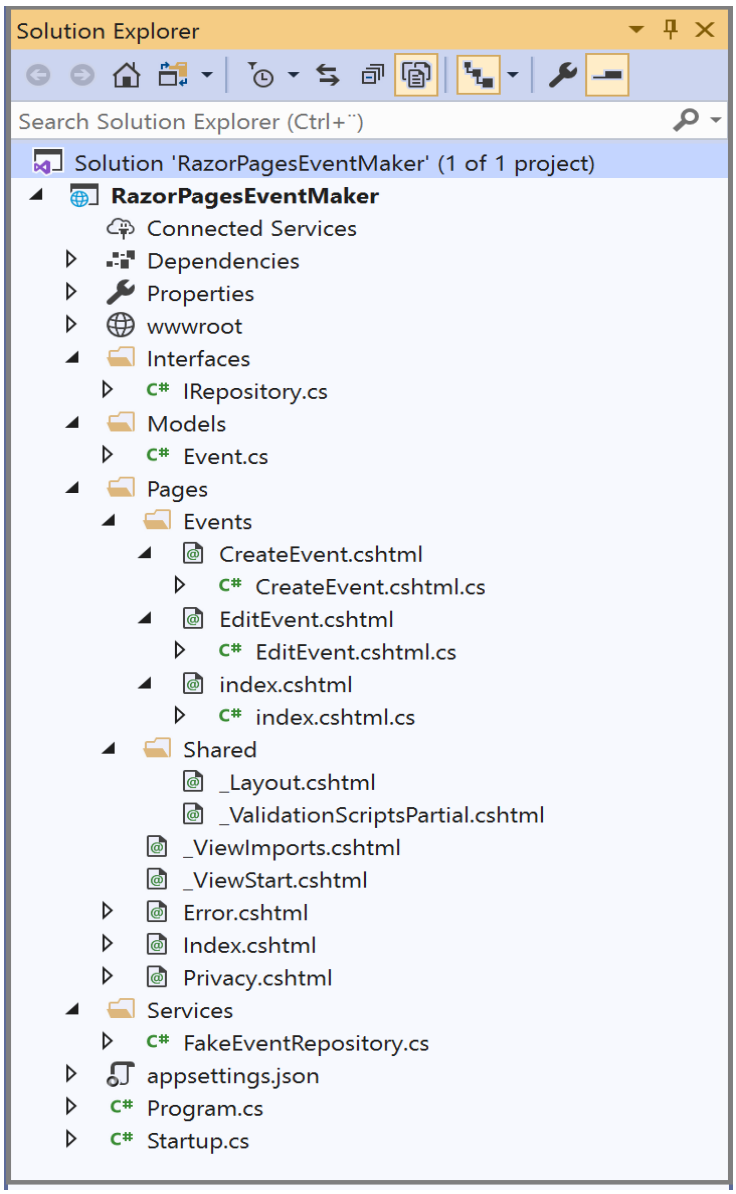
In simple words, Dependency injection consists of class(es), called services, that expose some public operations(mini-services). The service is usually provided as an interface. **Why**? Simply because interfaces are usually used to abstract implementations and infer loose coupling in the code .

> **Important**: It is common to design against interfaces instead of concrete implementation to gain loose coupling. Indeed, we can later on design another data access layer; let us say ***SQLRepository*** that also implements **IEventRepository**. By injecting **IEventRepository** reference, we do not stick to any concrete implementation and we can easily shift from one implementation to another without making our Razor Application aware of that. In the next chapter, we are going to implement another data access layer that works with **Json** file and you will see how flexible it is to add the new data access layer using Dependency injection.

Then, the service is registered in the **ConfigServices** method of the Startup.cs file. That´s it. The service is automatically available to the application and can be injected in any class that may use it. The model below illustrates the concept of dependency injection via the PageModel constructors.



Let us apply the Dependency Injection concept to our application. We make some changes to the file structure. We create a folder named "**Services".** We moved the FakeEventRepository class to this folder. We also created a folder named **Interfaces.** This folder will contain the interface that our service FakeEventRepository is implementing. The new structure along with the interface are shown below.



As can be seen, the **IEventRepository** interface defines these 4 operations that are already implemented.

```
 9    public interface IEventRepository
10    {
          3 references
11        List<Event> GetAllEvents();
12
          2 references
13        Event GetEvent(int id);
14
          2 references
15        void AddEvent(Event ev);
16
          2 references
17        void UpdateEvent(Event evt);
18
19    }
```

As mentioned before, in ASP.NET Core, services should be registered in the **ConfigureServices()** method of the **Startup.cs** file. ASP.NET core provides the following three methods to register services with the dependency injection container. The method that we use determines the lifetime of the registered service.

- ✔ **AddSingleton()** method creates a Singleton service. A Singleton service is created when it is first requested. This same instance is then used by all the subsequent requests. So in general, a Singleton service is created only one time per application and that single instance is used throughout the application lifetime.

- ✔ **AddTransient()** method creates a Transient service. A new instance of a Transient service is created each time it is requested.

- ✔ **AddScoped()** - This method creates a Scoped service. A new instance of a Scoped service is created once per request within the scope. For example, in a web application it creates 1 instance per each http request but uses the same instance in the other calls within that same web request.

You may already figure out which method to use. We want to implement the Singleton pattern behavior, so, we are going to use the AddSingleton method as show below.

```
25  public void ConfigureServices(IServiceCollection services)
26  {
27      services.AddRazorPages();
28      services.AddSingleton<IEventRepository, FakeEventRepository>();
29  }
```

We are almost done. Some small details are left:

- Make **FakeEventRepository.cs** implement the IEventRepository interface and make its constructor public.
- Remove the code about the implementation of Singleton Design pattern from FakeEventRepository.cs file.
- Inject the service in the **CreateEventModel** constructor. The code below shows only the portion of this class where we should make changes.

```
12  public class CreateEventModel : PageModel
13  {
14      // FakeEventRepository repo;
15      IEventRepository repo;
16      [BindProperty]
        13 references
17      public Event Event { get; set; }
        0 references
18      public CreateEventModel(IEventRepository repository)
19      {
20          //repo = FakeEventRepository.Instance;
21          repo = repository;
22      }
```

As can be seen, the injected IEventRepository reference is used to initialize the repo instance field.

Exercise: Do the same changes in the **EditEventModel** and the **IndexModel** classes. When done, run the application. The application is going to run PERFECT. Try to add new events to the list.

Now that we get the hang of almost everything, let us implement more user stories. Let us start with the following user story

✔ As a user , I will be able to filter the events based on the city name

The reason for implementing this user story is that we may have a huge number of events which do not fit in one screen. We want to filter them out based on the city. The appropriate place to add this functionality is the Index Razor Page. Simply because we are using this page to display the list of events.

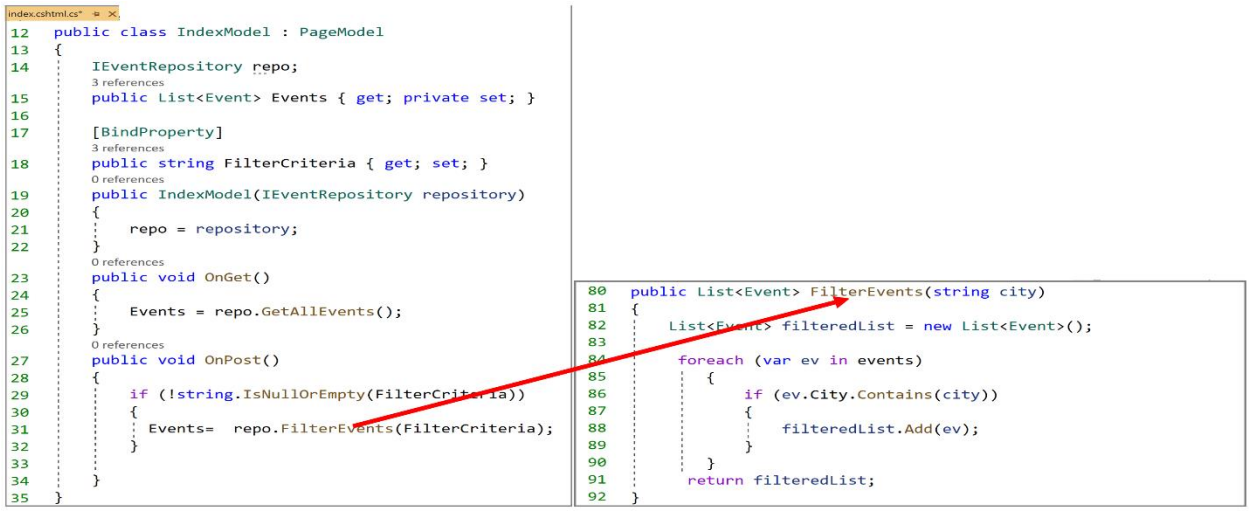
The code below shows the form that is used to enter the filtering criteria. As can be seen, we placed the form at the top of the Index page.

index.cshtml

```
@page
@model RazorPagesEventMaker.IndexModel
@{
    ViewData["Title"] = "index";
}
<h1>List of events </h1>
<p>
    <a asp-page="CreateEvent">Create New</a>
</p>
<form method="post">
    <p>
        Search: <input type="text" asp-for="FilterCriteria" />
        <input type="submit" value="Filter" />
    </p>
</form>
```

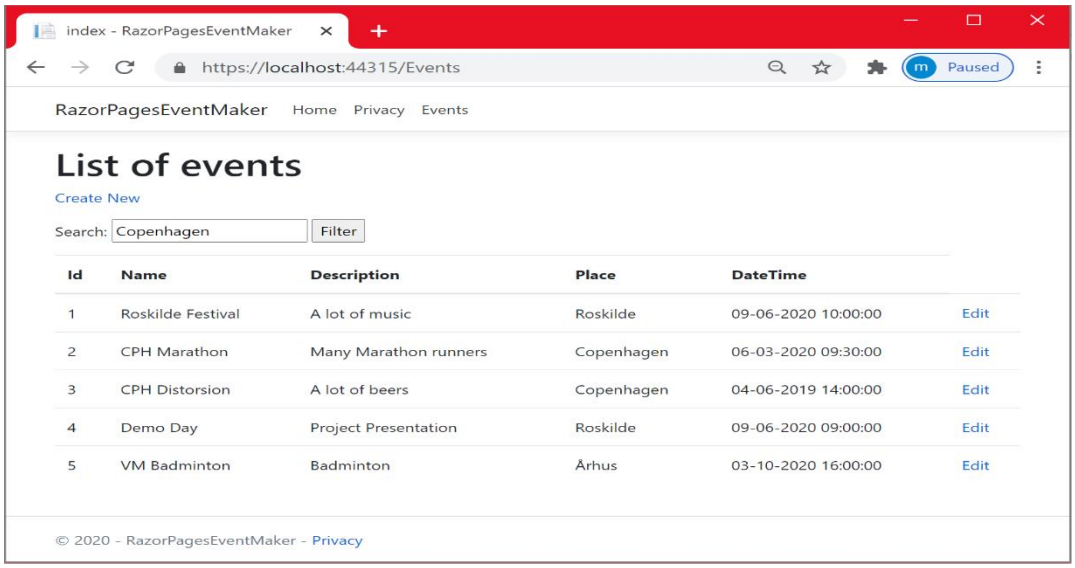
- Notice the use of the **asp-for** Tag helper in the form. What is specified in the asp-for attribute is a property whose value is evaluated against the model. That means, "FilterCriteria" should be a property in the IndexModel.
- Notice also the use of the "**post**"method for the form. That means , we should bind the form data to the IndexModel class as we want to pass the submitted data to this class.

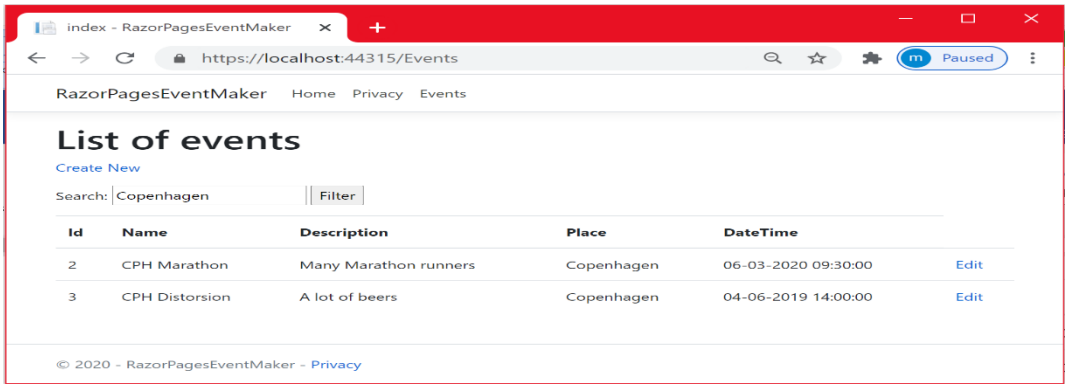
Let us explore the code further . As shown below, we defined the "**FilterCriteria**" property in the IndexModel. In the previous chapter, we have seen how we decorate properties to pass data from the display template into the PageModel. We use the [**bindProperty]** attribute as shown below. On the post method, if the string criteria is neither null nor empty, we call the FilterEvents method to return only those comply to the criteria. It is up to you to figure out how we implement filtering.



NB: do not forget to add the FilterEvents operation to the interface

- Run the application and Enter Copenhagen as filter criteria.
- Click on the “Filter” button . As you can see, the list of events was filtered





Exercise: Another method for implementing filtering

- In the **Index.cshtml** page, remove the method=”post” attribute from the form. Get is the default method for the form.

- In the **Index.cshtml.cs** file, as Get is the default method, the OnGet method is the one that will be invoked. **Place** the OnPost code in the OnGet method as shown below.

```
public void OnGet()
{
    Events = repo.GetAllEvents();
    if (!string.IsNullOrEmpty(FilterCriteria))
    {
        Events = repo.FilterEvents(FilterCriteria);
    }
}
```

- Remove the OnPost method
- Decorate the FilterCriteria property with the [BindProperty(**SupportsGet**=**true**)] attribute.
- Run the application and Enter a filter criteria. Try to explain why it also works.

Service injection into a Page

One thing left before closing this chapter. In the previous section of this chapter, we performed dependency injection via the constructor of the Index.cshtml.cs class. The Index.cshtml page had access to service through the **Events** property. Indeed, we used the **@Model.Events** in the **foreach** loop to access the list .

Important : With ASP.NET Core, the page can also access the service by injecting the data service directly into the page using the **@inject** directive. No need to go through the PageModel class.

In the code example below, we used the **@inject** directive to inject the **IEventRepository** service. By doing so, you provide the page with a reference that can be used to invoke the service. In the code below, the reference is used to get the number of the events in the list.

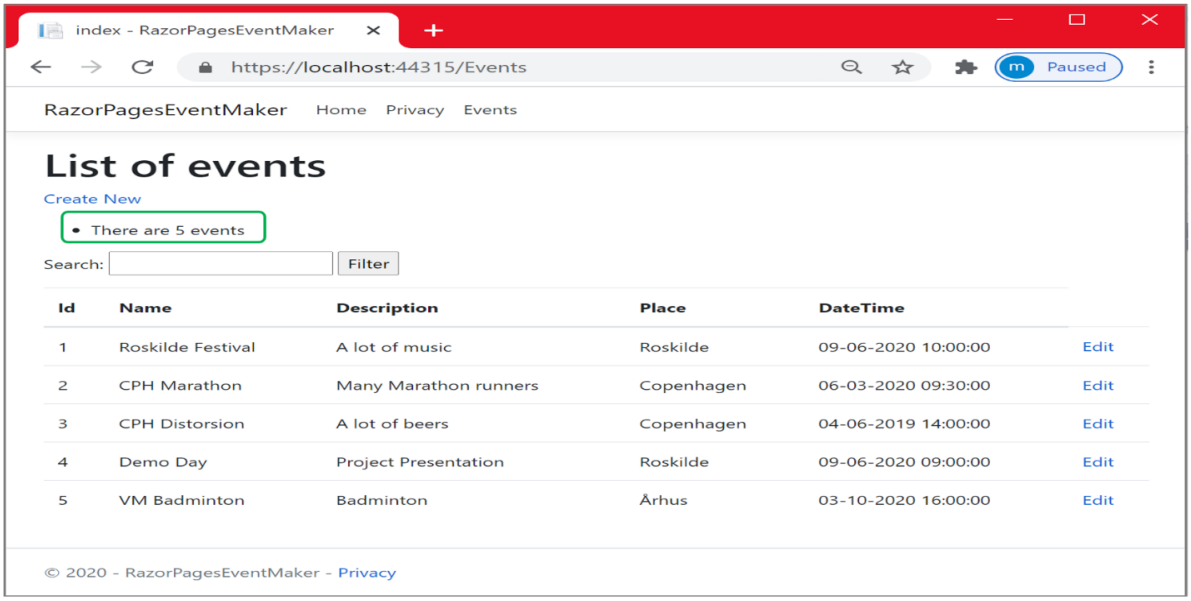
index.cshtml

```
@page
@model RazorPagesEventMaker.IndexModel
@using RazorPagesEventMaker.Interfaces
@inject IEventRepository  repository
@{
    ViewData["Title"] = "index";
}
<h1>List of events </h1>
<p>
    <a asp-page="CreateEvent">Create New</a>
</p>
<div>
    <ul>
        <li> There are @repository.GetAllEvents().Count events</li>
    </ul>
</div>
```

Using the @inject directive , you can even replace the ***foreach*** loop with the following code:

```
@foreach (var item in repository.GetAllEvents() )
```

As can be seen in the figure below, we could use the service to display the number of the events in the list.



So far, we have been working with the **FakeEventRepository** class, a data access layer dealing with data from a list. In the next chapter, we are going to implement another data access layer, this time, to persist our data in a file. We are going to use a **Json** file. Before digging into this, you have an exercise to solve

Exercise

Now that you get the hang of all what we have been through, try to implement the user stories "be able to delete a specific event" and "be able to display the details of a specific event". They are similar to "Edit a specific event". If you get stuck, look at the solution in GitHub.

References

- https://www.learnrazorpages.com/advanced/dependency-injection

Chapter 11 : Repository Design Pattern

Introduction

So far, we have been working with the **FakeEventRepository** class, a data access layer dealing with data from a list. In this chapter, we are going to implement another data access layer.This time, we want to persist our data in a file. With web applications, **Json** is still one of the preferred formats to store and process data. In this chapter, you will also see how easy it is to extend our application without any change to the existing code. This enhances the maintainability of our application because, once the existing code was tested, there is no need to test it again.

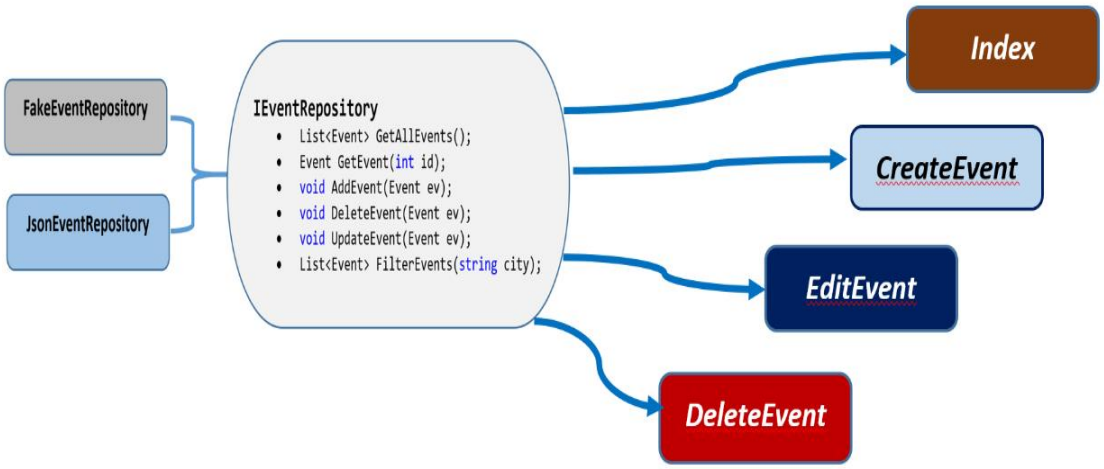
When we have many data access layers in an application, it is common to abstract the data access layer using the Repository Design Pattern. The idea with this pattern is to somehow abstract the way the application works with the data access layer without worrying about whether the implementation is towards the fake list or the json file.

Repository design pattern has many advantages: The code is cleaner and reusable, loosely coupled app and easy to maintain. Later on in **Part II** (**second semester**), you will see how easy to incorporate the SQL Server data access layer. So let us first look at this Design Pattern.

What is Repository design pattern

It is an abstraction of the data access layer. That means, we are not bound to a specific implementation of the data access layer (no idea about the details of how data is saved or retrieved from the underlying data source). The implementation is part of the respective classes that implement such abstraction. So we can have two repositories , the ***FakeEventRepository*** that saves and retrieves data from a list and the **JsonEventRepository** that saves and retrieves data from a json file). Each of these two repositories encapsulates its own implementation of the data access layer. The Repository Design pattern allows us to shift between the two implementations with minimal effort(by changing the set up in the services configuration).

The figure below illustrates the Repository Design Pattern applied to our application.

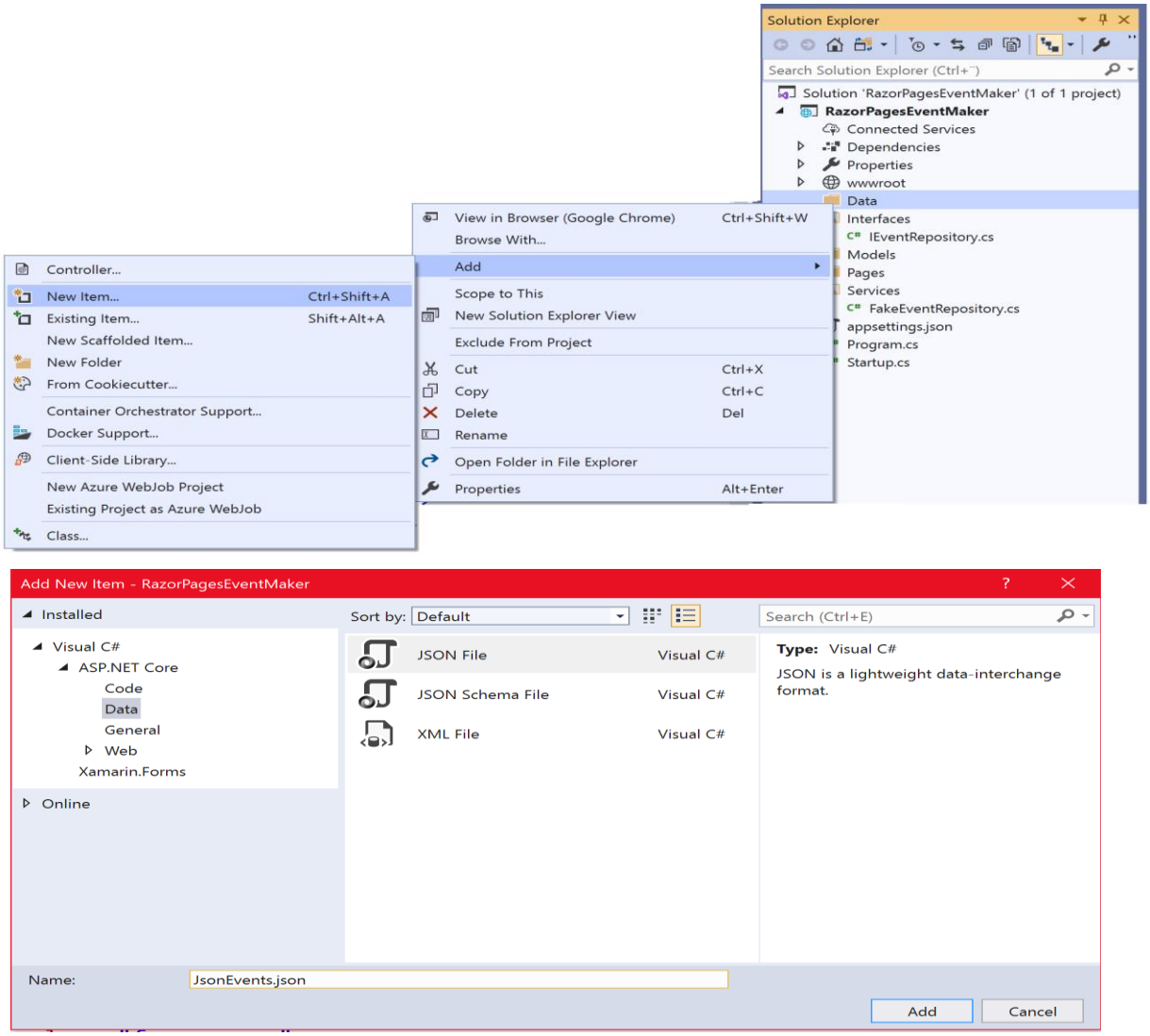


The Interface adds abstraction by specifying what operations are supported by the service and not how they are supported. In other words, it is about what the service is capable of doing but not how it does that. For example, the ***AddEvent (Event ev)*** method defined below ensures that a new event object is added to the underlying data source. **How**? This is not specified, because the interface is a contract to which each class that implements such interface must comply and provide its own implementation. Let us dig into the implementation of the JsonEventRepository data access layer.

What is a Json file

A json file is a file that stores data in a well-organized, easy to access manner. Json stands for **J**ava**S**cript **O**bject **N**otation, which is a lightweight, text based and human-readable format. It is a text file, so you can create a json file using any text editor. Let us create the json file using visual studio.

- Create a new folder, name it “**Data**”
- Create a json file, and name it “**JsonEvents.json**”, as shown below
- Click on the “Add” button. An empty file is created.



- Add the following data to the file.

```
[
  {
   "Id": 1,
   "Name": "Roskilde Festival",
   "Description": " A lot of music ",
   "City": "Roskilde",
```

```
  "DateTime": "2021-03-22T00:00:00"
 },
 {
  "Id": 2,
  "Name": "CPH Marathon",
  "Description": " Runners from other
countries",
  "City": "Copenhagen",
  "DateTime": "2020-10-25T08:55:00"
 }
]
```


JsonEventRepository data access layer

- In the Services folder, **add** a new class and name it "**JsonEventRepository**". This class will encapsulate the implementation of the json file data access layer.
- I suppose that you have previously implemented "**DeleteEvent**" and "**FilterEvents**" in previous chapters. **Make** the JsonEventRepoistory implement the following interface.

```
 9   public interface IEventRepository
10   {
         5 references
11       List<Event> GetAllEvents();
         3 references
12       Event GetEvent(int id);
         2 references
13       void AddEvent(Event ev);
         2 references
14       void DeleteEvent(Event ev);
         2 references
15       void UpdateEvent(Event ev);
         2 references
16       List<Event> FilterEvents(string city);
17   }
```

Let us implement one method at time:

GetAllEvents () **:** The code below shows the implementation of this method. Only a portion of the JsonEventRepository is shown.

JsonEventRepository.cs

```
1 reference
11  public class JsonEventRepository:IEventRepository
12  {
13      string JsonFileName = @"C:\Users\EASJ\Desktop\ASP.NET RAZOR PAGES\Kopi\RazorPagesEventMaker\Data\JsonEvents.json";
14
        6 references
15      public List<Event> GetAllEvents()
16      {
17          return JsonFileReader.ReadJson(JsonFileName);
18      }
```

JsonFileReader.cs*

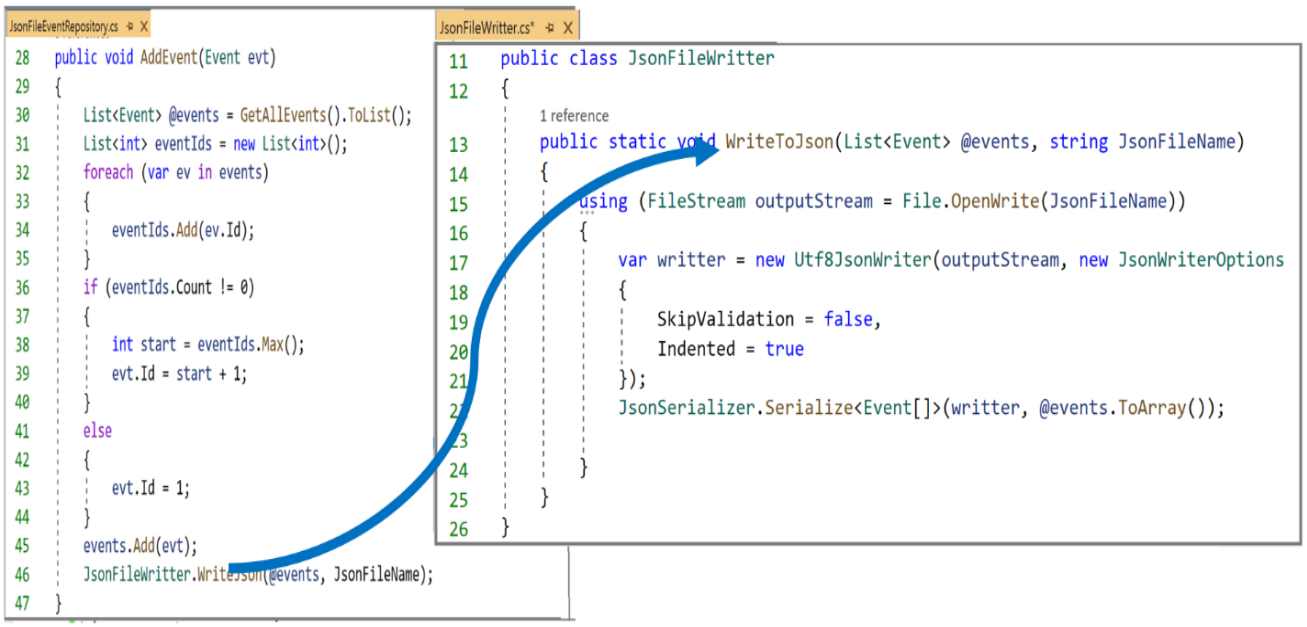
```
10  public class JsonFileReader
11  {
        1 reference
12      public static List<Event> ReadJson(string JsonFileName)
13      {
14          using (var jsonFileReader = File.OpenText(JsonFileName))
15          {
16              return JsonSerializer.Deserialize<List<Event>>(jsonFileReader.ReadToEnd());
17          }
18      }
19  }
```

Let us explore the code for displaying the 2 events from the file.

- **In the JsonEventRepository class**, the JsonFileName string defines the path of the json file. Notice the use of the **@** sign to escape the sequence of backslash (\) signs in the path.
- We create a folder named "**Helpers**" that will contain some help classes. In this folder , we added the above JsonFileReader class, where we defined the above static method ReadJson method. Let us explore this code:
  - ✓ **Line 14**: We use the using keyword to make sure that the resources we are using (File object) are released after the using close.

- **Line 16**: We use the JsonFileReader to read the content of the file to the end. Then, we deserialize the content into a List<Event> object , which is returned by the method.

CreateEvent(Event evt) **:** The code below shows the implementation of this method. Only a portion of the JsonEventRepository is shown



```
JsonFileEventRepository.cs
28  public void AddEvent(Event evt)
29  {
30      List<Event> @events = GetAllEvents().ToList();
31      List<int> eventIds = new List<int>();
32      foreach (var ev in events)
33      {
34          eventIds.Add(ev.Id);
35      }
36      if (eventIds.Count != 0)
37      {
38          int start = eventIds.Max();
39          evt.Id = start + 1;
40      }
41      else
42      {
43          evt.Id = 1;
44      }
45      events.Add(evt);
46      JsonFileWritter.WriteJson(@events, JsonFileName);
47  }
```

```
JsonFileWritter.cs*
11  public class JsonFileWritter
12  {
        1 reference
13      public static void WriteToJson(List<Event> @events, string JsonFileName)
14      {
15          using (FileStream outputStream = File.OpenWrite(JsonFileName))
16          {
17              var writter = new Utf8JsonWriter(outputStream, new JsonWriterOptions
18              {
19                  SkipValidation = false,
20                  Indented = true
21              });
2               JsonSerializer.Serialize<Event[]>(writter, @events.ToArray());
3
24          }
25      }
26  }
```

The code for AddEvent() method is very similar to the AddEvent method in the FakeEventRepository class. It is not a good practice to leave duplicate code in your design. We have to perform some refactoring. In spite of this, we are not doing any refactoring at the moment.

In the "**Helpers"** folder, we added the above JsonFileWritter class, where we defined the above static method WriteToJson method. Let us explore its code.

Line 15: We use the using keyword to make sure that the **FileStream** object is released after the **using** close.

Line 17-21 : We first create an instance of JsonWriterOptions and pass it into the writer in order to enable validation (skipValidation=false) and be able to format the output with indentation(indented=true). Then a writer is created by passing the Filestream object and the JsonWriterOptions object.

Line 22: The serialize method is used to write a JSON representation of the passed Event array using the writer.

EditEvent(Event evt) **:** The code below shows the implementation of this method. The code is given below. Again this code is very similar to the EditEvent method implementation in the FakeEventRepository. Notice the use of the help method WriteToJson (code reuse) by the UpdateMethod.

JsonEventRepository.cs*

```
3 references
public void UpdateEvent(Event @evt)
{
    List<Event> @events = GetAllEvents().ToList();

    if (@evt != null)
    {
        foreach (var e in @events)
        {
            if (e.Id == @evt.Id)
            {
                e.Id = evt.Id;
                e.Name = evt.Name;
                e.City= evt.City;
                e.Description = evt.Description;
                e.DateTime = evt.DateTime;
            }
        }
    }
    JsonFileWritter.WriteToJson(@events, JsonFileName);
}
```

The code below shows the GetEvent(int id) and the FilterEvents(string city) methods. The code is the same as the one implemented in the FakeEventRepository. We need to refactor our code.

JsonEventRepository.cs

```
public Event GetEvent(int id)
{
    foreach (var v in GetAllEvents())
    {
        if (v.Id == id)
            return v;
    }
    return new Event();
}

3 references
```

```
public List<Event> FilterEvents(string city)
{
    List<Event> filteredList = new List<Event>();
    List<Event> @events = GetAllEvents().ToList();

    foreach (var ev in events)
    {
        if (ev.City.Contains(city))
        {
            filteredList.Add(ev);
        }
    }
    return filteredList;
}
```

Exercise : As you have seen, we encountered a lot of code that was duplicated. Try to refactor your code to remove all duplicate code. Then implement the DeleteEvent method.

Now that we finish implementing the JsonEventRepository.cs data access layer , we need to register the service in the ConfigureServices method as shown below.

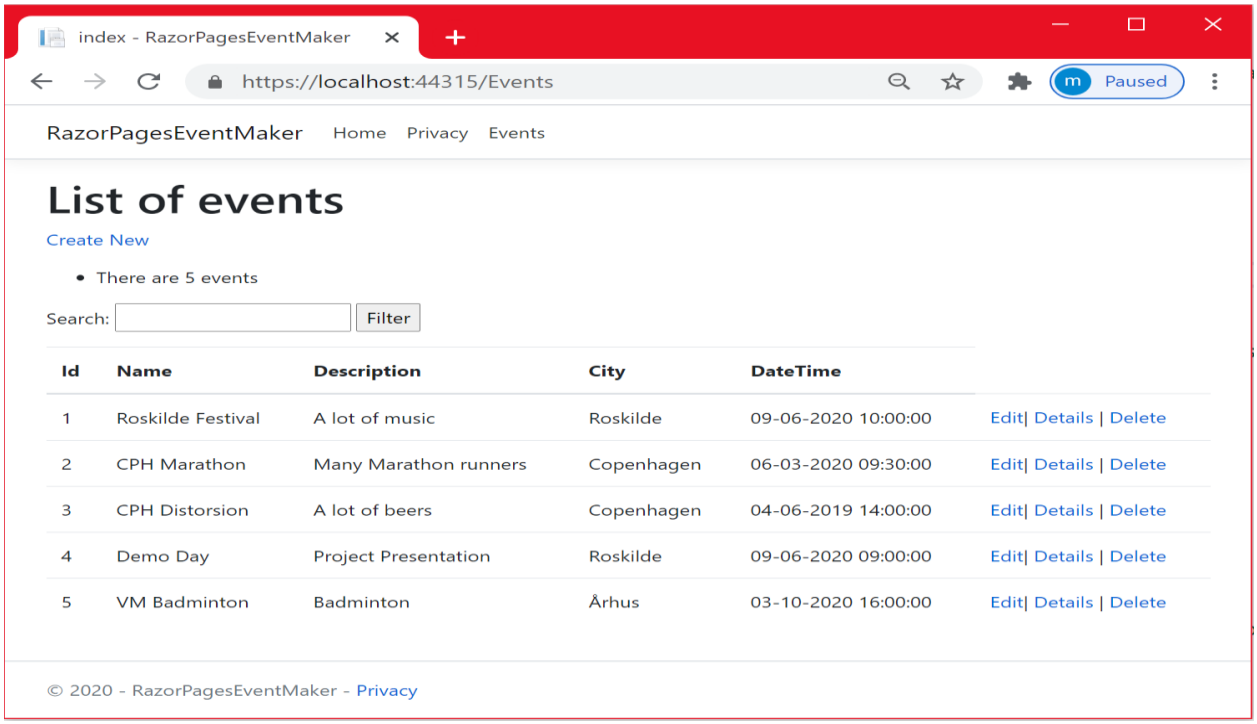
```
26  public void ConfigureServices(IServiceCollection services)
27  {
28      services.AddRazorPages();
29
30      services.AddSingleton<IEventRepository, FakeEventRepository>();
31
32      services.AddTransient<IEventRepository, JsonEventRepository>();
33  }
```

Important : Notice the use of the AddTransient to add this service to the container. **Why** ? Simply because we have a permanent data storage in the file and we can fetch data whenever we request it.

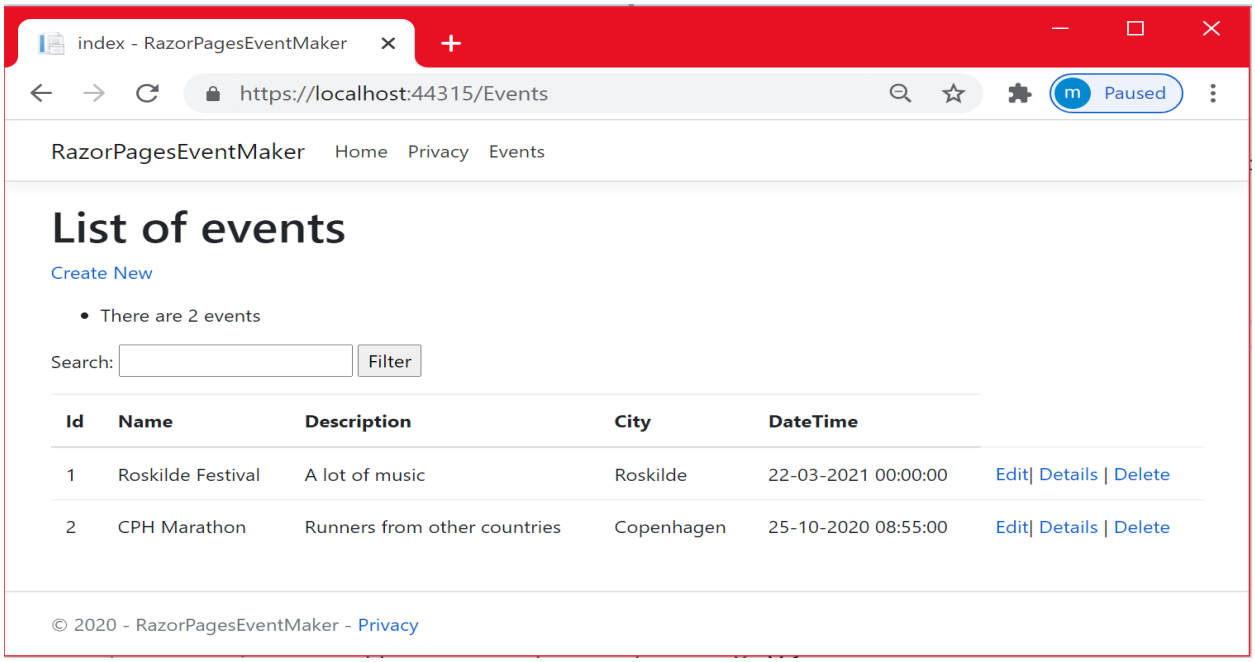
Remember that our **PageModel classes** get the **IEventRepository** service injected. These classes have no idea about the implementation. So, how does ASP.NET Core know which one to choose? We have just to disable one of the services in the startup.cs file

Let us say that we want to use the FakeEventRepository. A primitive way to do that is to comment the other service in the ConfigureServices method.

Run the application. The output is shown below



Let us shift to the JsonEventRepository service by commenting the FakeEventRepository. The output is shown below



Before closing this chapter, I want to draw **some very important conclusions**:

- ✔ It is easy to inject a service using dependency injection
- ✔ During the implementation of the JsonEventRepository service, I did not touch any piece of code from the existing application. On the contrary, some code was duplicated requiring some refactoring.
- ✔ This enhances extensibility. I could add another data access layer without changing the existing code.
- ✔ This enhances code reusability. We reused some code
- ✔ This enhances maintainability as I do not need to test again the existing code. Testing is the subject of the next chapter

References

- https://www.mikesdotnetting.com/article/337/whats-new-in-net-core-3-0-for-razor-pages
- http://zetcode.com/csharp/json/

Chapter 12 : Testing ASP.NET Core Razor Pages

Introduction

When developing software, especially when using the Agile approach, the software is subjected to changes. Performing these changes may cause the software to fail for some reasons. Thus, we need to test at least the most critical parts of the software.

This chapter is about performing Unit testing the way it should be. I said "it should be" because It is rare that people (at least our 1st semester students) take into consideration code dependencies when performing unit testing.

What is unit testing?

> **Unit testing** can be defined as testing the functionality of each method in the class i**n isolation** without dependencies and infrastructure (like a database).

The word "isolation" is the most important in the above definition. One way to isolate the code under test is via loose coupling. In the previous chapter, we used dependency injection not for show. You remember that we injected the interface reference of type IEventRepository into the constructor of the PageModel class. Thus, any class that implements IEventRepository can be passed in the constructor. Doing so, we made our PageModel classes loosely coupled with the data access layer classes because the PageModel class is not aware of the data access layer implementation (whether it is FakeEventRepository or JsonEventRepository). This enhances and favors testability (Unit testing) because we can replace the data access layer implementation by mock objects and work with mock objects as if they are real.

Let us look at our application. The PageModel classes code is depending on the data access layer code. To test this code, we are going to use the **Moq** framework to simulate dependencies with the data access layer and be able to test the PageModel classes independently of any concrete implementation of the data access layer.

What is a Mock framework?

Mocking is replacing the behavior of classes and interfaces (in our case, data access layer interface) by Mock objects that imitate this behavior as if they are real. A mocked object is a fabricated object with a predetermined set of property and

method behaviors used for testing .This way, we are sure that the code we want to test works on its own and does not fail because of errors in the code it depends on.

How mock works?

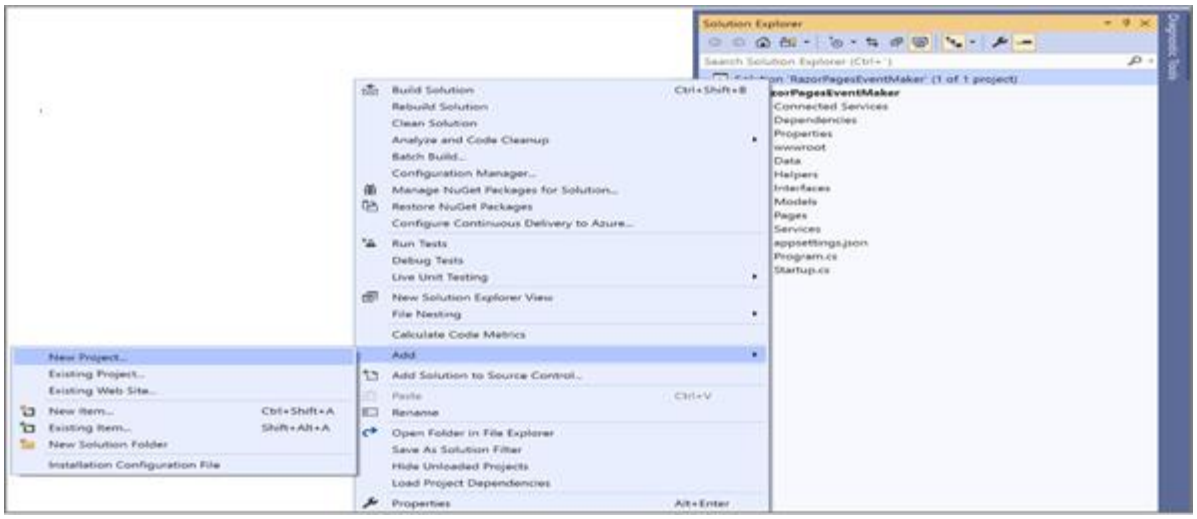
In this chapter, we are going to use the Moq framework. Do not be afraid, the framework is very simple. You have just to create a mock object. Once it is created, you can call methods on the mock object including parameters and return values. You can also set parameters defined in the dependencies. You can also verify that the methods you set up are being called in the tested code. Does it make sense to you? No. Is it confusing? Yes, at least now. Do not worry, let us dig into test code.

Unit Testing

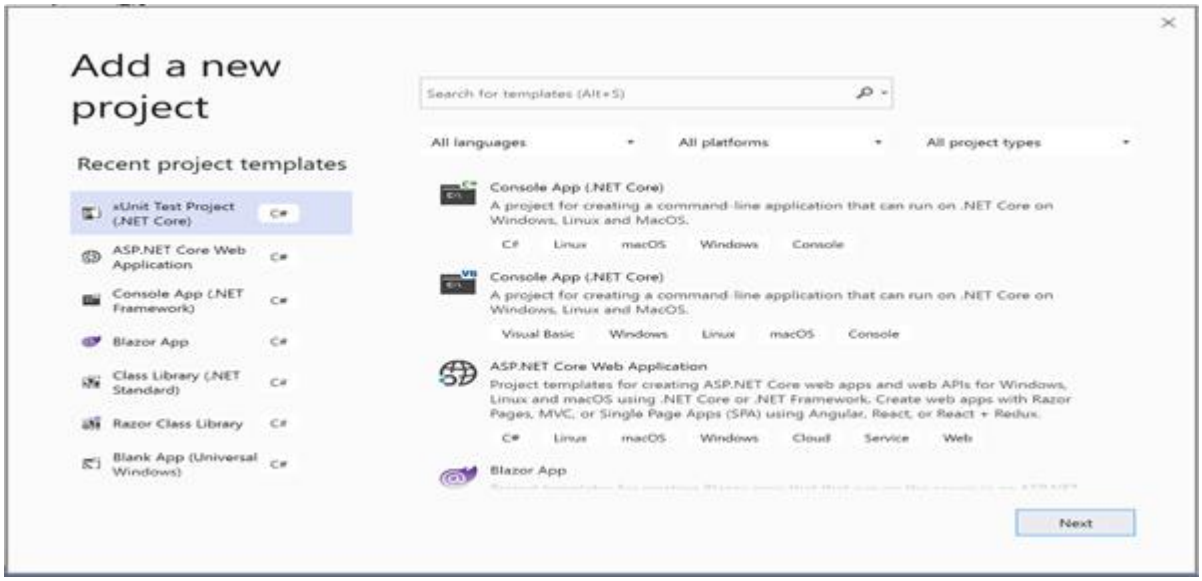
As mentioned before, our PageModel classes are going to be unit tested in isolation from the data access layers code using **Moq** framework. In terms of testing framework, We are going to use the **xUnit** test framework. Previously during this semester, you have been working with Unit Test and you might be familiar with other testing frameworks. No problem, you have to know that test concepts and test implementations across different test frameworks are similar but not identical. No matter the framework that you use, it is the same concept.

Building the test project

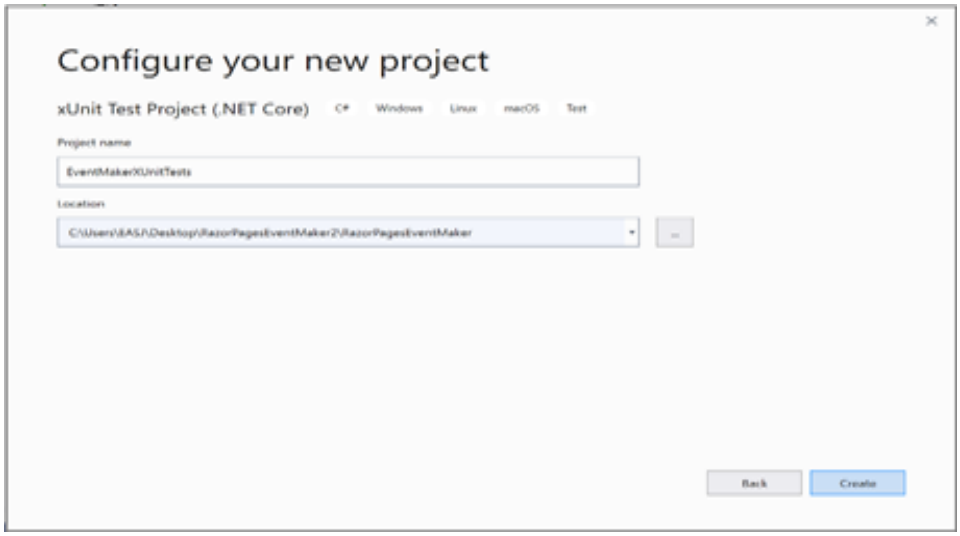
- Right click on the solution and **add a new project**



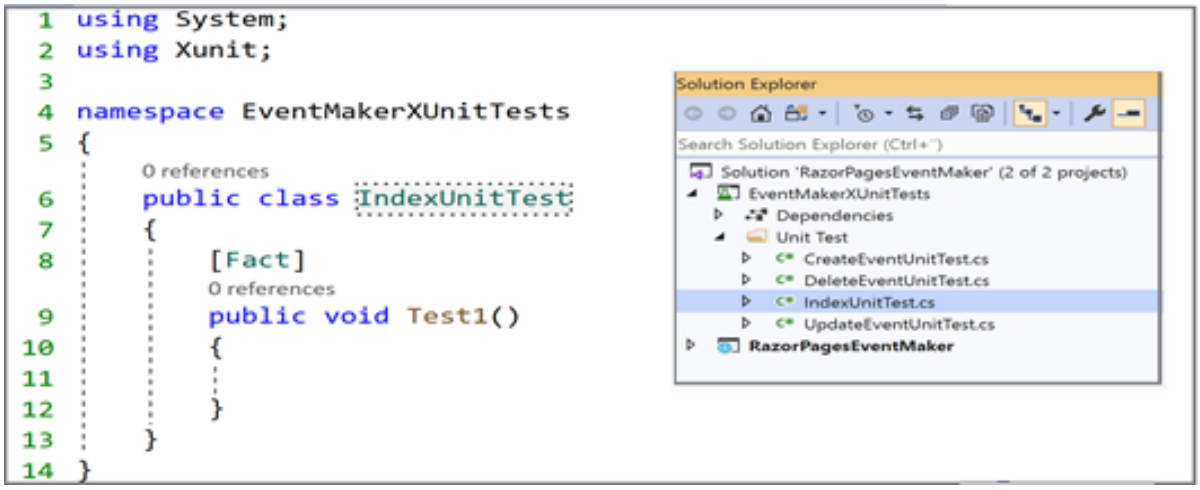
- Select "***xUnit Test Project (.NET Core)***" and click on **Next**.



- Give it a name and click on "**Create**".

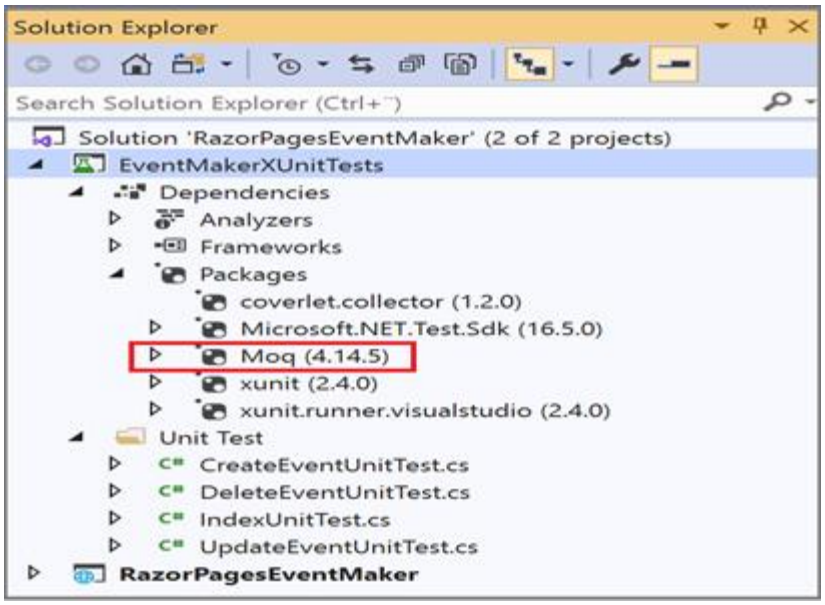


- Create a folder to encapsulate unit testing for all the PageModel classes and create test classes for Index, CreateEvent, UpdateEvent and DeleteEvent classes as shown below.

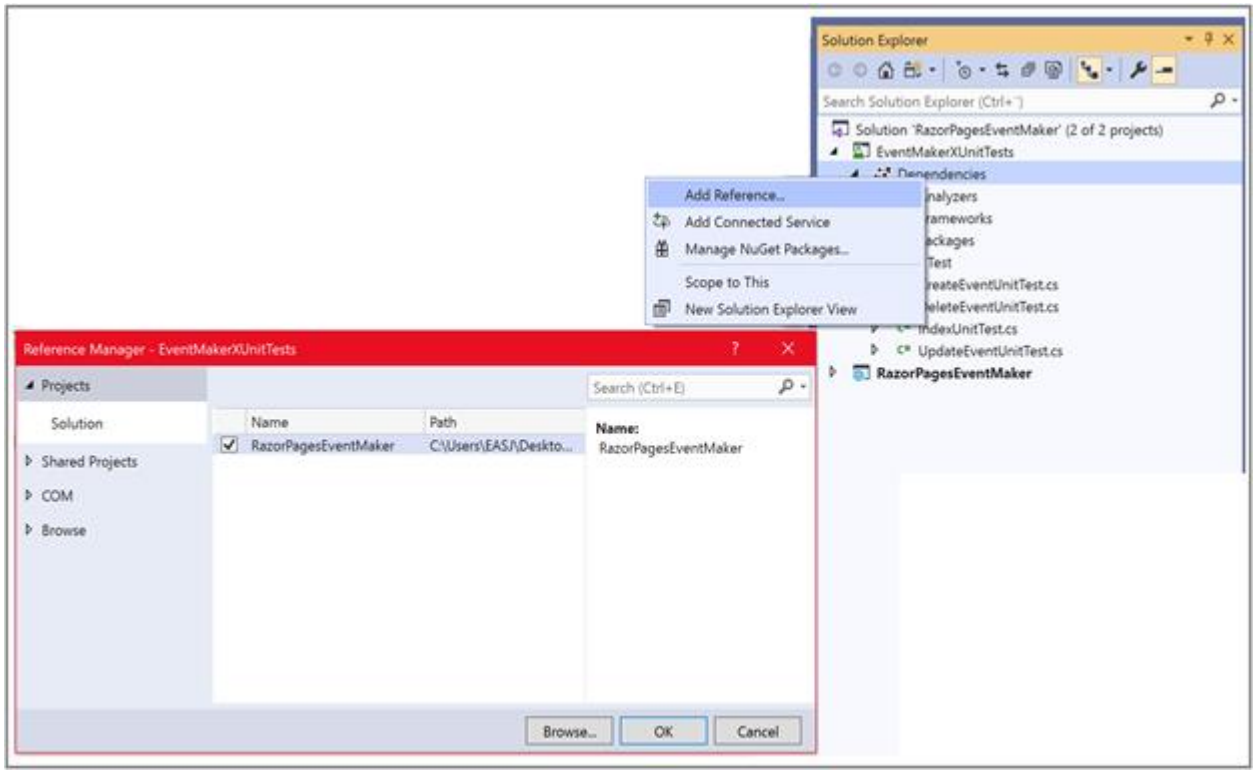


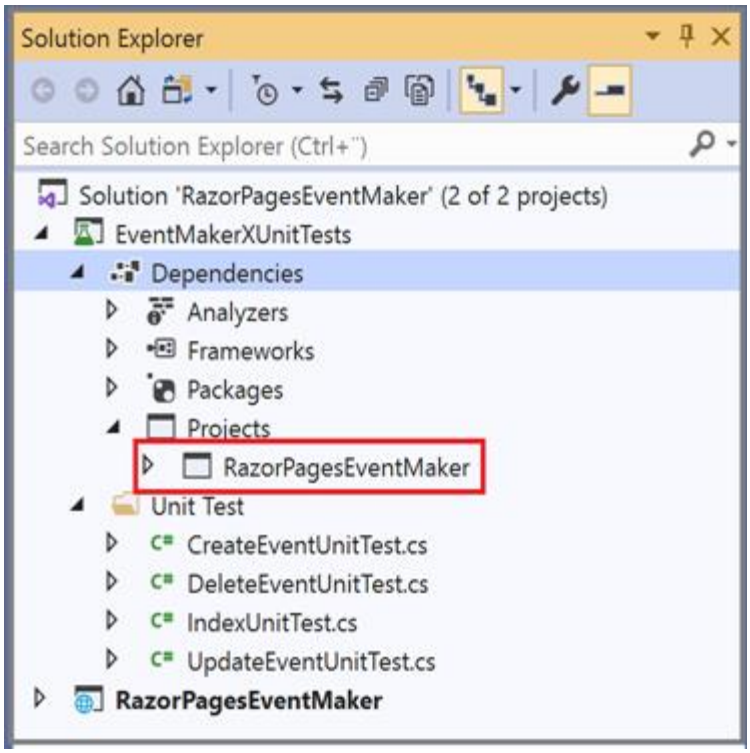
.

Using the NuGet packages, add "**Moq"** a friendly mocking framework from **.Ne**t to your test project.This is shown below. Now, we are ready to start testing.



As we want to mock the IEventRepository interface, we need to have a reference to the RazorPagesEventMaker project as shown below.





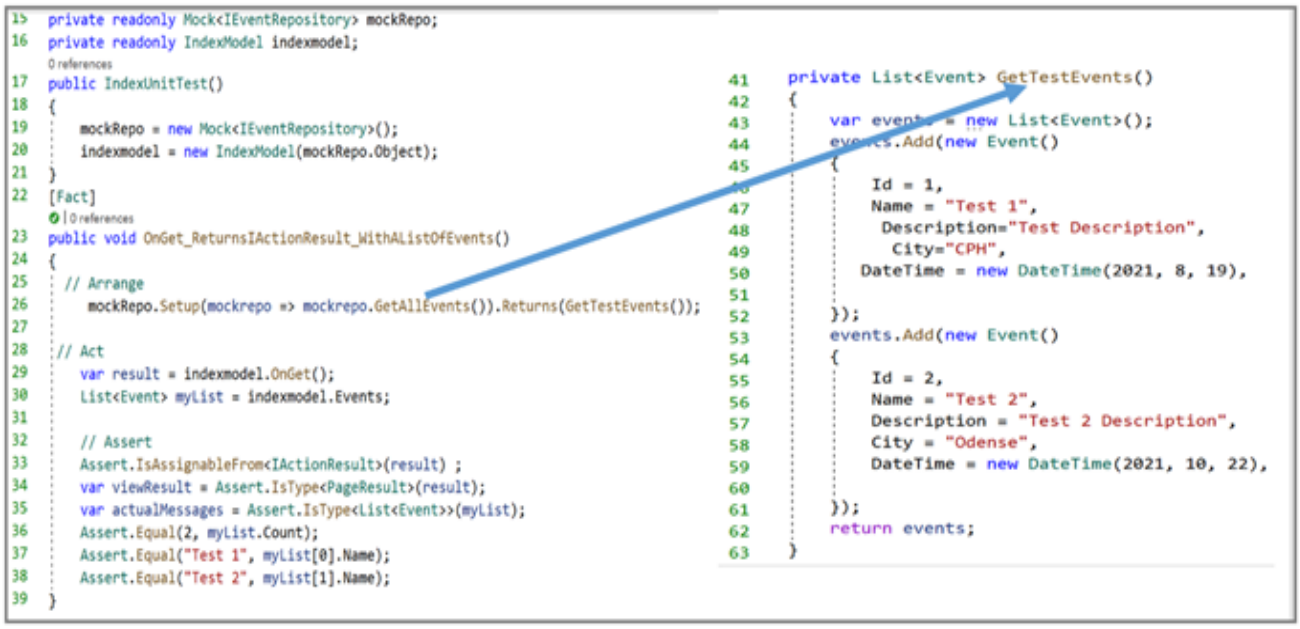
Unit Testing index.cshtml.cs

The figure below shows the OnGet method of the index.cshtml.cs that we want to unit test.

Test case1: We want to test the following test case :

The OnGet method returns the right type and a List of events.

The figure below shows the test code.



```
15  private readonly Mock<IEventRepository> mockRepo;
16  private readonly IndexModel indexmodel;
    0 references
17  public IndexUnitTest()
18  {
19      mockRepo = new Mock<IEventRepository>();
20      indexmodel = new IndexModel(mockRepo.Object);
21  }
22  [Fact]
    0 references
23  public void OnGet_ReturnsIActionResult_WithAListOfEvents()
24  {
25    // Arrange
26      mockRepo.Setup(mockrepo => mockrepo.GetAllEvents()).Returns(GetTestEvents());
27
28  // Act
29      var result = indexmodel.OnGet();
30      List<Event> myList = indexmodel.Events;
31
32      // Assert
33      Assert.IsAssignableFrom<IActionResult>(result) ;
34      var viewResult = Assert.IsType<PageResult>(result);
35      var actualMessages = Assert.IsType<List<Event>>(myList);
36      Assert.Equal(2, myList.Count);
37      Assert.Equal("Test 1", myList[0].Name);
38      Assert.Equal("Test 2", myList[1].Name);
39  }

41  private List<Event> GetTestEvents()
42  {
43      var events = new List<Event>();
44      events.Add(new Event()
45      {
47          Id = 1,
48          Name = "Test 1",
49           Description="Test Description",
50            City="CPH",
51         DateTime = new DateTime(2021, 8, 19),
52
53      });
54      events.Add(new Event()
55      {
56          Id = 2,
57          Name = "Test 2",
58          Description = "Test 2 Description",
59          City = "Odense",
60          DateTime = new DateTime(2021, 10, 22),
61
62      });
63      return events;
    }
```

Let us explore and explain the test code:

Line 15 : we create an instance field of type **Mock<IEventRepository>** to imitate a reference to the data access layer so that it can access all its public methods, properties…etc. This instance is initialized in the constructor (line 19).

Line 16 : we create an instance field of type IndexModel. This instance will be used to access the methods that will be tested. It is also initialized in the constructor (line 20).

> **Notice** the use of the **Arrange-Assert-Act** pattern that you are familiar from the chapter on testing.

//Arrange

Line26:

> **mockRepo.Setup(mockrepo=>mockrepo.GetAllEvents()).Returns (GetTestEvents());**

This line of code is the most important. Without this statement, the mock object cannot do anything. In this line of code, we instruct the mock object to imitate a reference to IEventRepository, call the GetAllEvents which is part of the

IEventRepository interface, and then call the GetTestEvent method (shown on the right hand side) to returns a list of 2 events (also called test events). What is important to underline here is that instead of relying on the GetAllEvents method to return the list of events from any of the two data access layers, the method is returning a fictive/fabricated list.

- Notice the use of what we call lambda expression (***mockrepo => mockrepo.GetAllEvents()***), a concept that you are not familiar with yet. Do not worry; at this stage no need to know about lambda expressions. They are simply a way to pass a method as a parameter.

//Act

Line 30: **var result = indexmodel.OnGet();**

In this line of code, we use the IndexModel reference to call the OnGet method.

// Assert

Assert.IsAssignableFrom<IActionResult>(result) ;

In this line of code, we are asserting that the result object has the given type or a derived type. We expect that the test pass because the OnGet method is returning a PageResult type, which derives from the IActionResult .

var viewResult = Assert.IsType<PageResult>(result);

In this line of code, we are asserting that the result object has exactly the given type and not a derived type. As mentioned before, the OnGet method returns exactly a PageResult type and we expect that the test pass.

var actualMessages =Assert.IsType<List<Event>>(indexmodel.Events);

In this line of code, we are using the Index Model reference to get the Events property . Then we assert that the return type is List<Event>. The test will pass because we expect the Events property to be of type List<Event>.

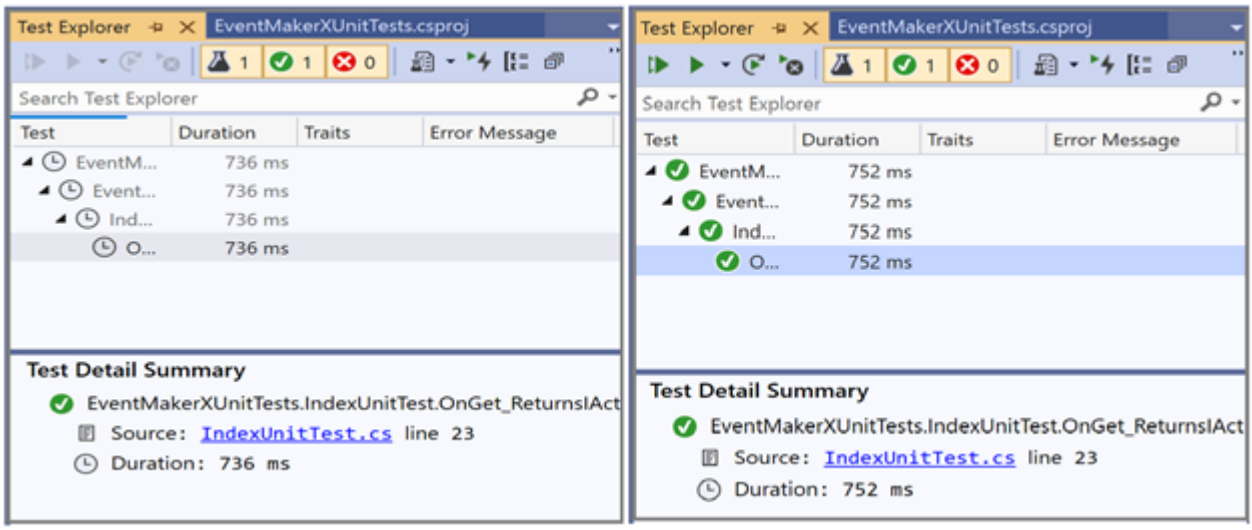
Assert.Equal(2, myList.Count);

Assert.Equal("Test 1", myList[0].Name);

Assert.Equal("Test 2", myList[1].Name);

In these 3 lines of code, we assert that the number of test events is 2 , the name of the first one is “Test 1” and the name of the second one is “Test 2”

Let us **run** the test. The figure below shows the test result output. As expected, all tests pass.



The figure below shows a list of the most commonly used xUnit.net **Assert** methods

Name	Description
`Equal(expected, result)`	This method asserts that the result is equal to the expected outcome. There are overloaded versions of this method for comparing different types and for comparing collections. There is also a version of this method that accepts an additional argument of an object that implements the `IEqualityComparer<T>` interface for comparing objects.
`NotEqual(expected, result)`	This method asserts that the result is not equal to the expected outcome.
`True(result)`	This method asserts that the result is `true`.
`False(result)`	This method asserts that the result is `false`.
`IsType(expected, result)`	This method asserts that the result is of a specific type.
`IsNotType(expected, result)`	This method asserts that the result is not a specific type.
`IsNull(result)`	This method asserts that the result is `null`.
`IsNotNull(result)`	This method asserts that the result is not `null`.
`InRange(result, low, high)`	This method asserts that the result falls between `low` and `high`.
`NotInRange(result, low, high)`	This method asserts that the result falls outside `low` and `high`.
`Throws(exception, expression)`	This method asserts that the specified expression throws a specific exception type.

Exercise: Try to figure out a few other test cases and perform their unit test.

You can also look at the test code on Github for many other test cases.

Unit Testing EditEvent.cshtml.cs

Test Case1 : GetEvent() method returns an existing Event object and the OnGet method returns the right type.

The code below shows the test code for testing the OnGet method shown below on the right hand side.

```
15  public class EditEventUnitTest
16  {
17      private readonly Mock<IEventRepository> mockRepo;
18      private readonly EditEventModel editmodel;
        0 references
19      public EditEventUnitTest()
20      {
21          mockRepo = new Mock<IEventRepository>();
22          editmodel = new EditEventModel(mockRepo.Object);
23      }
24      [Fact]
        0 references
25      public void OnGetReturnsIActionResultWithEventFound()
26      {
27          // Arrange
28          int testEventId = 1;
29          string eName = "Marathon";
30          Event @event = new Event()
31          {
32              Id = testEventId,
33              Name = eName
34          };
35          mockRepo.Setup(repo => repo.GetEvent(testEventId))
36              .Returns(@event);
37
38          // Act
39          var result = editmodel.OnGet(testEventId);
40          // Assert
41          Assert.IsType<PageResult>(result);
42          Assert.Equal(testEventId, @event.Id);
43          Assert.Equal(eName, @event.Name);
44      }
```

EditEvent.cshtml.cs

```
21  public IActionResult OnGet(int id)
22  {
23      Event= repo.GetEvent(id);
24      if (Event == null)
25      {
26          return null;
27      }
28      return Page();
29  }
```

Let us explore and explain the test code:

Instance fields and their initialization were discussed in the previous section.

// Arrange

Line 28-24 :

In these lines of code, we define an integer and a string variable. The values of these variables are used to initialize a new Event object. We then set up the mock object to call the GetEvent (testEventId) method and return the test event **@event**.

// Act

var result = editmodel.OnGet(testEventId); // call of the OnGet method

// Assert

Assert.IsType<PageResult>(result);

Assert.Equal(testEventId, @event.Id);

Assert.Equal(eName, @event.Name);

Assert.IsAssignableFrom<IActionResult>(result) ;

No need to explain the code in these Assert methods. You have seen this code before.

Test Case 2 : GetEvent() method returns null, the OnGet method also returns null.

The figure below shows the test code and the OnGet method code

EditEventUnitTest.cs

```
48  [Fact]
49  public void OnGet_method_Event_Is_Null()
50  {
51      // Arrange
52      int testEventId = 1;
53      var mockRepo = new Mock<IEventRepository>();
54      mockRepo.Setup(repo => repo.GetEvent(testEventId)).Returns(() => null);
55
56      // Act
57      var result = editmodel.OnGet(testEventId);
58
59      // Assert
60      Assert.IsNotType<PageResult>(result);
61      Assert.Null(result);
62
63  }
```

EditEvent.cshtml.cs

```
21  public IActionResult OnGet(int id)
22  {
23      Event= repo.GetEvent(id);
24      if (Event == null)
25      {
26          return null;
27      }
28      return Page();
29  }
```

The figure below shows a list of the most commonly used xUnit.net Assert methods

Exercise: Try to figure out a few other test cases and perform their unit test.

You can also look at the test code on Github for many other test cases.

Unit Testing EditEvent.cshtml.cs

Test Case1 : GetEvent() method returns an existing Event object, the OnGet returns the right type.

The code below shows the test code for testing the OnGet method that is shown below on the right hand side.

```
15  public class EditEventUnitTest
16  {
17      private readonly Mock<IEventRepository> mockRepo;
18      private readonly EditEventModel editmodel;
        0 references
19      public EditEventUnitTest()
20      {
21          mockRepo = new Mock<IEventRepository>();
22          editmodel = new EditEventModel(mockRepo.Object);
23      }
24      [Fact]
        0 references
25      public void OnGetReturnsIActionResultWithEventFound()
26      {
27          // Arrange
28          int testEventId = 1;
29          string eName = "Marathon";
30          Event @event = new Event()
31          {
32              Id = testEventId,
33              Name = eName
34          };
35          mockRepo.Setup(repo => repo.GetEvent(testEventId))
36              .Returns(@event);
37
38          // Act
39          var result = editmodel.OnGet(testEventId);
40          // Assert
41          Assert.IsType<PageResult>(result);
42          Assert.Equal(testEventId, @event.Id);
43          Assert.Equal(eName, @event.Name);
44      }
```

EditEvent.cshtml.cs

```
21  public IActionResult OnGet(int id)
22  {
23      Event= repo.GetEvent(id);
24      if (Event == null)
25      {
26          return null;
27      }
28      return Page();
29  }
```

Let us explore and explain the test code:

Instance fields and their initialization were discussed in the previous section.

// Arrange

Line 28-35 :

```
int testEventId = 1;
string eName = "Marathon";
Event @event = new Event()
{
    Id = testEventId,
    Name = eName
};
mockRepo.Setup(repo => repo.GetEvent(testEventId)).Returns(@event);
```

In these lines of code, we define an integer and a string variable. The values of these variables are used to initialize a new Event object. We then set up the mock object to call the GetEvent (testEventId) method and return the test event @event.

// Act

var result = editmodel.OnGet(testEventId); // call of the OnGet method

// Assert

```
Assert.IsType<PageResult>(result);
Assert.Equal(testEventId, @event.Id);
Assert.Equal(eName, @event.Name);
Assert.IsAssignableFrom<IActionResult>(result) ;
```

No need to explain the code in these Assert methods. You have probably seen this code before.

Test Case 2 : GetEvent() method returns null, the OnGet method also returns null.

The figure below shows the test code and the OnGet method code

EditEventUnitTest.cs

```
48  [Fact]
49  public void OnGet_method_Event_Is_Null()
50  {
51      // Arrange
52      int testEventId = 1;
53      var mockRepo = new Mock<IEventRepository>();
54      mockRepo.Setup(repo => repo.GetEvent(testEventId)).Returns(() => null);
55
56      // Act
57      var result = editmodel.OnGet(testEventId);
58
59      // Assert
60      Assert.IsNotType<PageResult>(result);
61      Assert.Null(result);
62
63  }
```

EditEvent.cshtml.cs

```
21  public IActionResult OnGet(int id)
22  {
23      Event= repo.GetEvent(id);
24      if (Event == null)
25      {
26          return null;
27      }
28      return Page();
29  }
```

Let us explore the test code:

We start by creating instance fields of type Mock<IEventRepository> and EditEventModel. These instances are then initialized in the constructor of the test class (this is not shown here).

//Arrange

Line 54: **mockRepo.Setup(repo => repo.GetEvent(testEventId)).Returns(() => null**);

In this line of code, we instruct the mock object to call the GetEvent method which is part of the IEventRepository interface, and then returns null. According to the code

under test, if null is returned upon the call of the GetEvent (int id) method, we expect the OnGet method to return null.

```
//Act
```

Line 57: **var result = editmodel.OnGet()**; // we call the OnGet method

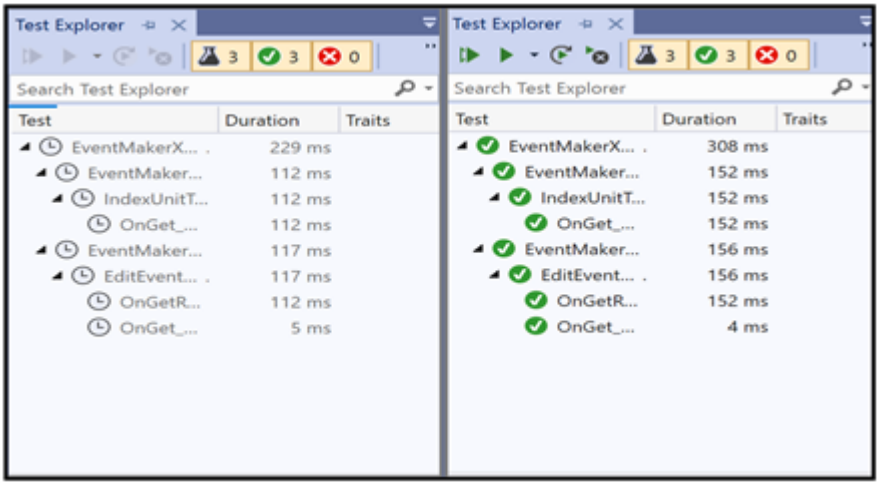
```
// Assert
```

Line 60: Assert.IsNotType<PageResult>(result);

Line 61: Assert.Null(result);

We expect that the return value is null. So it is not of type PageResult of course.That is why we make these 2 assertions.

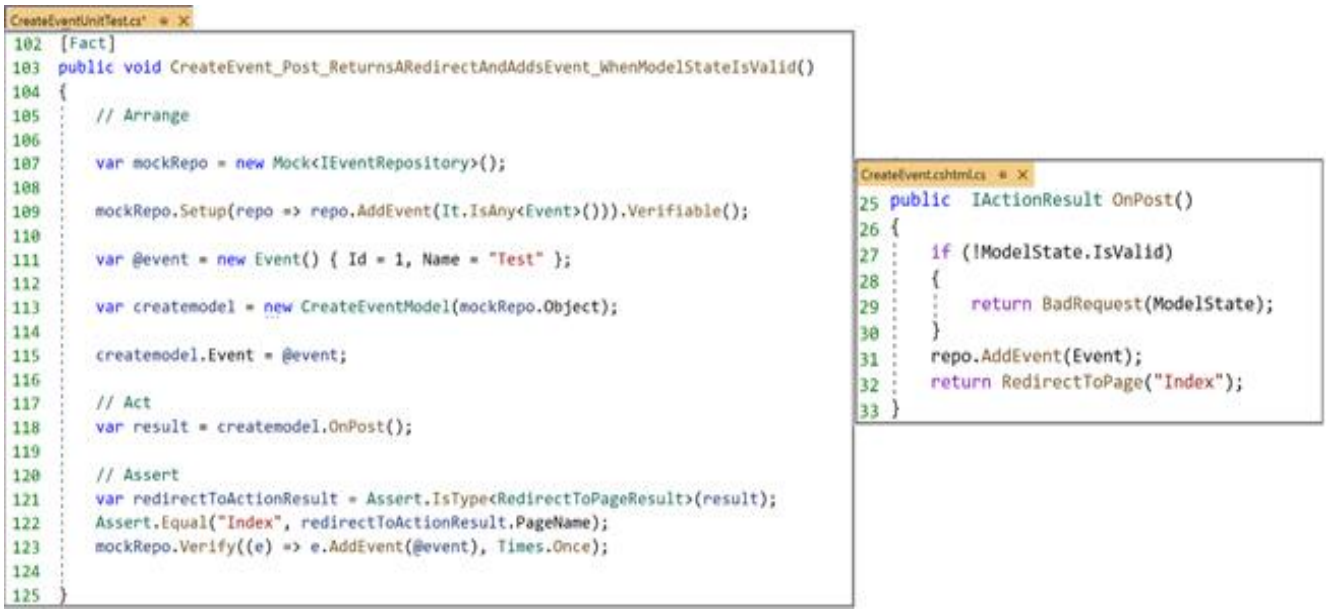
Let us run all the 3 tests performed until now. The output is shown below.



Unit Testing CreateEvent.cshtml.cs

Test Case1 : If the model is valid, the AddEvent() method is called once and the user is redirected to the Index page.

The code below shows the test code for testing the OnPost method.



```
CreateEventUnitTest.cs*
102  [Fact]
103  public void CreateEvent_Post_ReturnsARedirectAndAddsEvent_WhenModelStateIsValid()
104  {
105      // Arrange
106
107      var mockRepo = new Mock<IEventRepository>();
108
109      mockRepo.Setup(repo => repo.AddEvent(It.IsAny<Event>())).Verifiable();
110
111      var @event = new Event() { Id = 1, Name = "Test" };
112
113      var createmodel = new CreateEventModel(mockRepo.Object);
114
115      createmodel.Event = @event;
116
117      // Act
118      var result = createmodel.OnPost();
119
120      // Assert
121      var redirectToActionResult = Assert.IsType<RedirectToPageResult>(result);
122      Assert.Equal("Index", redirectToActionResult.PageName);
123      mockRepo.Verify((e) => e.AddEvent(@event), Times.Once);
124
125  }
```

```
CreateEvent.cshtml.cs
25  public  IActionResult OnPost()
26  {
27      if (!ModelState.IsValid)
28      {
29          return BadRequest(ModelState);
30      }
31      repo.AddEvent(Event);
32      return RedirectToPage("Index");
33  }
```

Let us explore the most important parts of the test code:

//Arrange

Line 109: **mockRepo.Setup(repo => repo.AddEvent(It.IsAny<Event>())).Verifiable();**

In this line of code, we instruct the mock object to call the AddEvent method passing any value of type Event as a parameter. The Verifiable() method allows us to check (in the Assert part) that the AddEvent() method is invoked by the mock object.

Line 115: **createmodel.Event = @event;**

We set the Event property to the new created and valid @event object. Otherwise we will pass null to the AddEvent() method and this method will not be invoked.

//Act

Line 118: **var result = createmodel.OnPost()**; // we call the OnPost method

// Assert

var redirectToActionResult = Assert.IsType<RedirectToPageResult>(result);

In this line of code, we assert that the return type is exactly RedirectToPageResult as expected

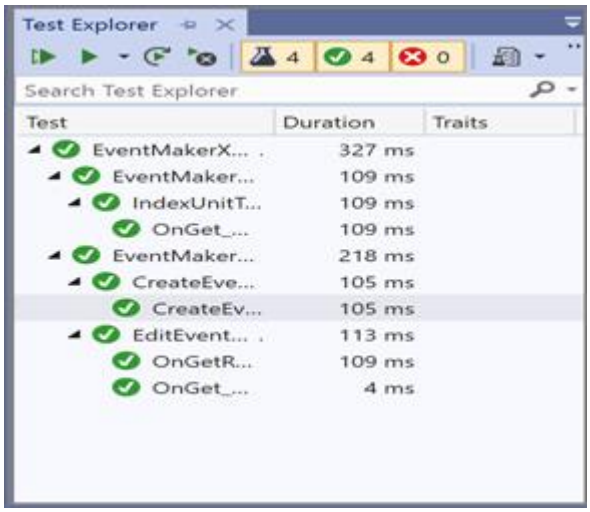
Assert.Equal("Index", redirectToActionResult.PageName);

In this line of code, we assert that the page to which the user is redirected is named “Index”.

mockRepo.Verify((e) => e.AddEvent(@event), Times.Once);

In this line of code, we check whether the AddEvent() method is called once.

Let us run all the 4 tests performed until now. The output is shown below.



<u>Test case 2</u> : if the model state is invalid, the OnPost method will return he badRequest

CreateEventUnitTest.cs*

```
18  [Fact]
19  public void OnPost_InValidState()
20  {
21      // Arrange
22      var mockRepo = new Mock<IEventRepository>();
23      var createmodel = new CreateEventModel(mockRepo.Object);
24      createmodel.ModelState.AddModelError("key1", "The Text field is required.");
25
26      // Act
27      var result = createmodel.OnPost();
28
29      // Assert
30      Assert.IsNotType<PageResult>(result);
31      var badRequestResult = Assert.IsType<BadRequestObjectResult>(result);
32      Assert.IsType<SerializableError>(badRequestResult.Value);
33  }
```

CreateEvent.cshtml.cs

```
25  public  IActionResult OnPost()
26  {
27      if (!ModelState.IsValid)
28      {
29          return BadRequest(ModelState);
30      }
31      repo.AddEvent(Event);
32      return RedirectToPage("Index");
33  }
```

- The most important piece of test code is **Line 24**. We used the AddModelError() method to simulate an invalid model state.
- We assert that the return type is the return type of the BadRequest () method. You can explore the definition of the BadRequest method. You have just to right-click on it and select “Go to Definition”. This method is defined as shown below. Notice the BadRequestObjectResult return type used in the Assert statement (**Line 31**).

Unit Testing DeleteEvent.cshtml.cs

Test1 : the DeleteEvent() method is called once and the user is redirected to the Index page.

The code below shows the test code for testing the OnPost method.

```
24  [Fact]
    0 references
25  public void Can_Delete_Valid_Events()
26  {
27      // Arrange - create an event
28      Event @event = new Event { Id = 2, Name = "Test2" };
29      mockRepo.Setup(m => m.GetAllEvents()).Returns(new List<Event> {
30              new Event {Id = 1, Name = "Test1"},@event,
31              new Event { Id = 3, Name = "Test3"},});
32      var deletemodel = new DeleteEventModel(mockRepo.Object);
33      deletemodel.Event = @event;
34
35      // Act -
36      var result= deletemodel.OnPost();
37
38      // Assert - ensure that the repository delete method was invoked
39      var redirectToActionResult = Assert.IsType<RedirectToPageResult>(result);
40      Assert.Equal("Index", redirectToActionResult.PageName);
41      mockRepo.Verify(m => m.DeleteEvent(@event), Times.Once);
42      Assert.NotNull(result);
43  }
```

DeleteEvent.cshtml.cs

```
29  public IActionResult OnPost()
30  {
31      if (Event != null)
32      {
33          repo.DeleteEvent(Event);
34      }
35
36      return RedirectToPage("Index");
37  }
```

I think that you are familiar with most of the above code. It is self-explanatory. Go ahead and explore it yourself.

Let us **run t**he application with the six tests we have implemented until now. The output is shown below.

That's it, we are done with Unit testing. Let us draw some conclusions:

- Remember the definition of Unit testing. It is about Testing a unit (i.e. a class) in isolation. Therefore, if you are testing a class with its dependencies, rather than perform Unit testing, you are performing integration tests.
- Moq framework is a performant and very easy tool to mock dependencies.
- To use the Moq framework, it is crucial to abstract dependencies. Interfaces are commonly used for this purpose. Dependency injection was the key for that.

References

- https://spin.atomicobject.com/2017/08/07/intro-mocking-moq/
- https://docs.microsoft.com/en-us/aspnet/core/test/razor-pages-tests?view=aspnetcore-3.1
- ttps://docs.microsoft.com/en-us/aspnet/core/test/integration-tests?view=aspnetcore-3.1
- https://www.youtube.com/watch?v=dBCFFZS4ACo

Chapter 13: building a real application

1-* relationship using the Fake repository with a list

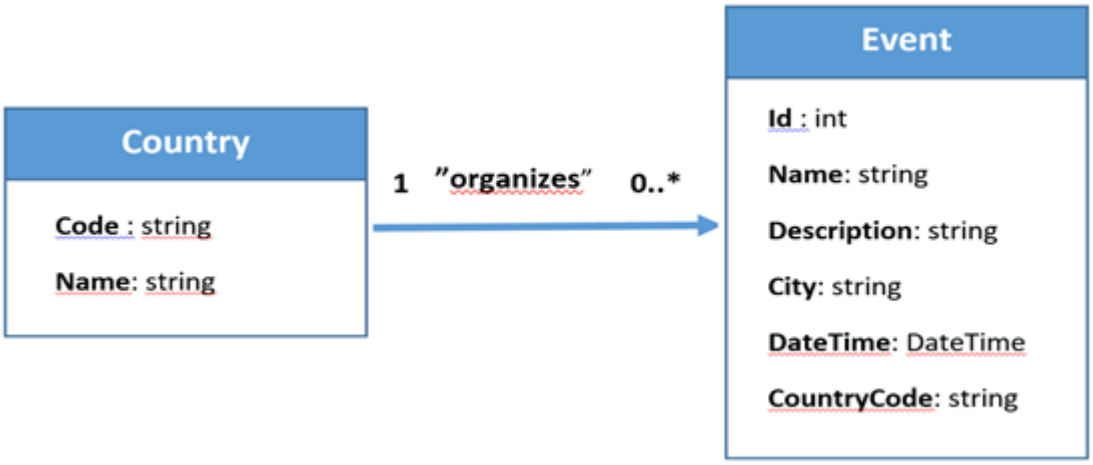
Introduction

As you have noticed, managing Event objects is the main functionality of our application. We were dealing with only one type of objects, which is the Event type. In real life applications that follow the Object Oriented Paradigm, to perform a task many objects should communicate and interact using the services of each other.

Let us go back to our application. Let us say that we are doing well and we gain a good reputation in Europe and we want to extend our application to include organizing events in most of the European cities (Copenhagen, Paris, Madrid, Brussels, …etc.).

One of the new requirements that we impose to our application is to be able to display the events that take place in a specific European country. The user story could be formulated as follows: ***As a traveler, I will be able to view all the events in a specific European country, so I can choose my destination***.

For a good design, our model is no longer going to be composed of a single class Event but will include another class "Country". The domain model is illustrated in the figure below.



No need to explain the relationship between these two conceptual classes. It is a simple one-to-many association. This can be implemented in many ways. The Country class encapsulating a list of event objects is an option. However, in our case, we are going to implement this association by having the CountryCode as a property in the Event class, to refer to the Code property in the Country class. The Code property is unique for each country. This is also how a relational database works. Anyway, the database is the subject of the next semester.

Another thing that deserves attention is that it will make sense to move the "City" property from the Event class into the "Country" class. This is because it is more appropriate to assign City as an attribute to the Country class than to the Event class. Anyway, let us close this debate and suppose that for the sake of the filtering

functionality (based on the city), “City” is still a property in the Event class to make filtering easier.

Let us start coding. Many things are exactly the same as what we did when dealing with the Event class. We start by adding the Country model class. The code below shows the code of the Event and Country classes side by side.

```
9   public class Country
10  {
11      [Required]
        16 references
12      public string Code { get; set; }
13
14      [Required]
        11 references
15      public string Name { get; set; }
16  }
```

```
9   public class Event
10  {
        36 references
11      public int Id { get; set; }
        6 references
12      public string CountryCode { get; set; }
13      [Display(Name = "Event Name")]
14      [Required(ErrorMessage = "Name of the Event is required"), MaxLength(30)]
        29 references
15      public string Name { get; set; }
        20 references
16      public string Description { get; set; }
17      [Required]
18      [StringLength(18, ErrorMessage = "Name of the city can not be longer than 18 chars")]
        21 references
19      public string City { get; set; }
20      [Required(ErrorMessage = "Date required")]
21      [Range(typeof(DateTime), "10/1/2020",  "10/1/2021",
22      ErrorMessage = "Value for {0} must be between {1} and {2}")]
        20 references
23      public DateTime DateTime { get; set; }
24  }
```

Notice the CountryCode property in the Event class to refer to the Code property in the Country class.

We then create a FakeCountryRepository class that encapsulates a list of countries and the CRUD operations related to the Country objects. The figure below shows the Event list along with the Country list.

```
FakeCountryRepository.cs*
10 public class FakeCountryRepository:ICountryRepository
11 {
       8 references
12     private List<Country> countries { get; }
       0 references
13     public FakeCountryRepository()
14     {
15         countries = new List<Country>();
16         countries = new List<Country>();
17         countries.Add(new Country() { Code = "FR", Name = "France" });
18         countries.Add(new Country() { Code = "DK", Name = "Denmark" });
19         countries.Add(new Country() { Code = "SP", Name = "Spain" });
20     }
```

```
FakeEventRepository.cs*
10 public class FakeEventRepository:IEventRepository
11 {
       11 references
12     private List<Event> events { get; }
       0 references
13     public FakeEventRepository()
14     {
15         events = new List<Event>();
16         events.Add(new Event() {Id = 1,Name = "Roskilde Festival",
17             Description = " A lot of music", CountryCode="DK",
18          City="Roskilde",   DateTime = new DateTime(2020, 6, 9, 10, 0, 0) } );
19         events.Add(new Event(){ Id = 2, Name = "Paris Marathon",
20             Description = " Many Marathon runners",
21          City="Paris",   CountryCode = "FR", DateTime = new DateTime(2020, 3, 6, 9, 30, 0) });
22         events.Add(new Event() {Id = 3, Name = "Paris-Dakar",
23             Description = "Car racers",  City="Paris",
24             CountryCode= "FR",DateTime = new DateTime(2020, 10, 3, 16, 0, 0)});
25     }
```

As can be seen, the FakeCountryRepository implements an interface named ICountryRepository. This repository is configured as a service in the ConfigureServices method of the Startup.cs class, as shown below.

```
26   public void ConfigureServices(IServiceCollection services)
27   {
28       services.AddRazorPages();
29       services.AddSingleton<IEventRepository, FakeEventRepository>();
30       services.AddSingleton<ICountryRepository, FakeCountryRepository>();
31   }
```

In the Pages folder, we created a new folder named “**Countries”**. This folder will hold all the Razor Pages to display all countries in the collection, to add a new country and to delete an existing country. At the top menu, we added a link to the page displaying the list of countries. The new file structure looks like the following :

The only code that deserves a lot of attention is the code in the "**CreateEvent**" page and the "**CountryEvents**" page. It is interesting to look at "CreateEvent" because we need to supply the CountryCode as well to create an Event. The "CountryEvents" is also interesting because this page is where we implement displaying the events that occur in a specific country. As I consider visiting France , I would like to see which cities are organizing events.It sounds obvious. Let us explore the code in these 2 files(CreateEvent and CountryEvents)

CreateEvent page

The code below shows the part of the "CreateEventModel" class that changed.

```
CreateEvent.cshtml.cs*
13  public class CreateEventModel : PageModel
14  {
15      IEventRepository repo;
16      [BindProperty]
        15 references
17      public Event Event { get; set; }
        2 references
18      public SelectList CountryCodes { get; set; }
        0 references
19      public CreateEventModel(IEventRepository repository , ICountryRepository  crepo )
20      {
21          repo = repository;
22          List<Country>  countries = crepo.GetAllCountries();
23          CountryCodes = new SelectList( countries, "Code", "Name");
24      }
```

Line 18: We define a **SelectList** property named “CountryCodes”, which represents the list from which the user can select a single item.

Line 22: Notice the use of the “**ICountryRepository**” service to get all countries.

Line 23: We create and initialize our selectList. We are creating the SelectList from the “countries” collection. We specify the “**Code**” property value as the data value (the value binded to the CountryCode property of the Event object). We also specify the “**Name**” property value as the text that appears in the list of options.

Now, let us look at the page code. The code below shows the last part of the “CreateEvent.cshtml” file.

CreateEvent.cshtml

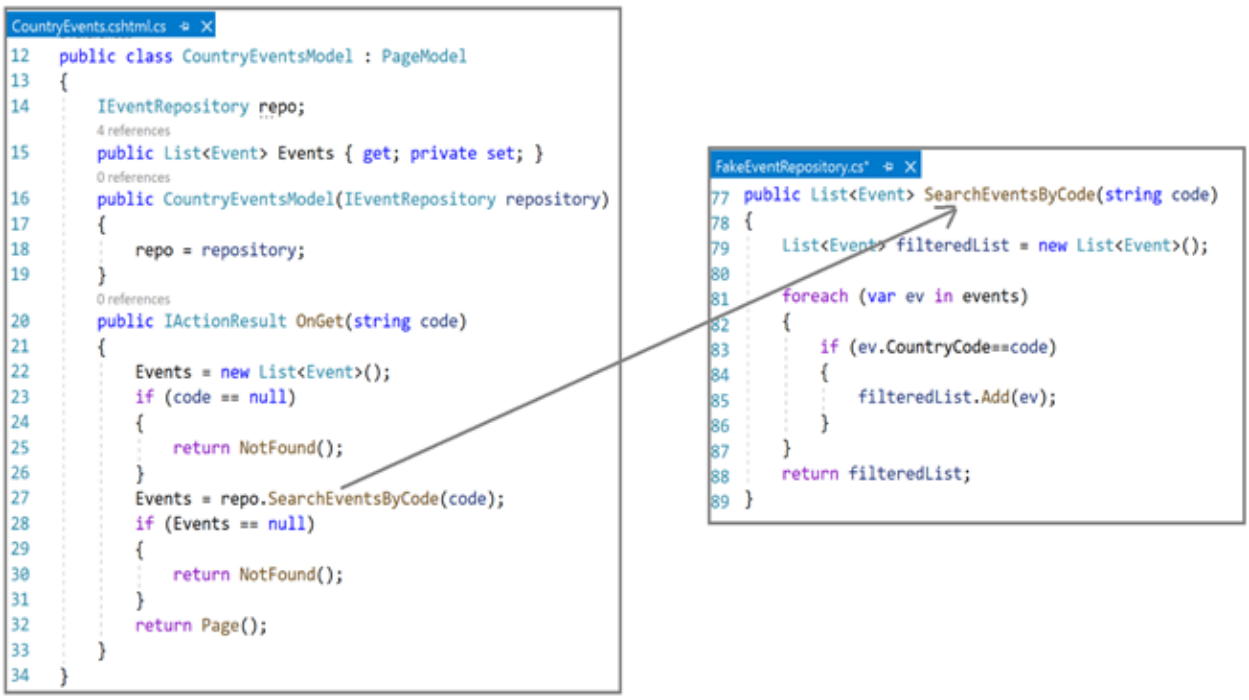
```
27              <div class="form-group">
28                  <label asp-for="@Model.Event.DateTime" class="control-label"></label>
29                  <input asp-for="@Model.Event.DateTime" class="form-control" />
30                  <span asp-validation-for="@Model.Event.DateTime" class="text-danger"></span>
31              </div>
32              <div class="form-group">
33                  <label asp-for="@Model.Event.CountryCode" class="control-label"></label>
34                  <select asp-for="@Model.Event.CountryCode" class="form-control"
35                          asp-items="@Model.CountryCodes">
36                      <option value="">-- Select Code --</option>
37                  </select>
38                  <span asp-validation-for="@Model.Event.CountryCode" class="text-danger" />
39              </div>
40              <div class="form-group">
41                  <input type="submit" value="Create" class="btn btn-primary" />
42              </div>
43          </form>
44      </div>
45  </div>
46  <div>
47      <a asp-page="Index">Back to List</a>
48  </div>
```

Line 34: In the <**select**> element, we used the **asp-for** attribute to specify the property to which the selected item is bound to. In our case, the selected item is bound to the **CountryCode** property.

Line 35: We also used the **asp-items** attribute to specify the collection to which the selectList is binded to. In this case, the collection is binded to the "**CountryCodes**" property defined earlier in the PageModel. It represents the list of items from which the user can select a single item.

CountryEvents

The code below shows the OnGet method. We have passed the code as a parameter to the OnGet method from the CountryIndex page using the QueryString method that we have covered in chapter 9.

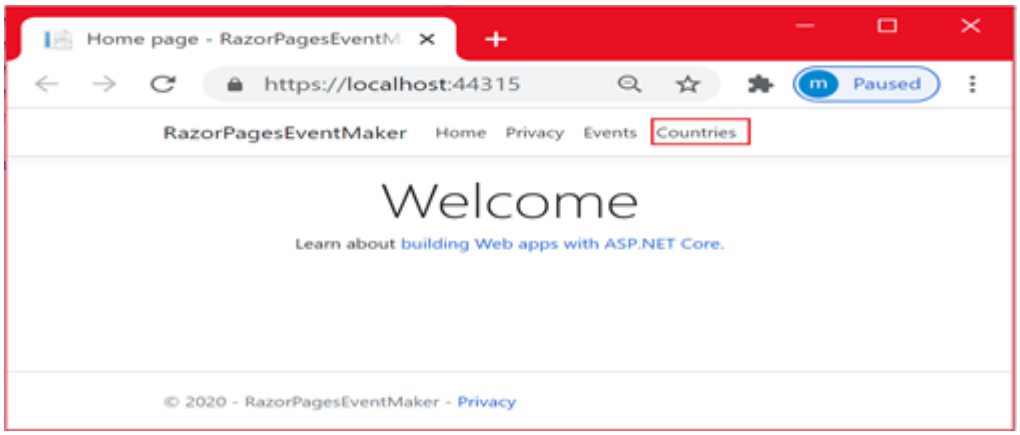


When navigating to this page, the OnGet method is invoked and the Event property is initialized with the list of Event objects that match the passed code parameter. The ***SearchEventsByCode*** method (on the right hand side) is called to filter out the events that match the search criteria.

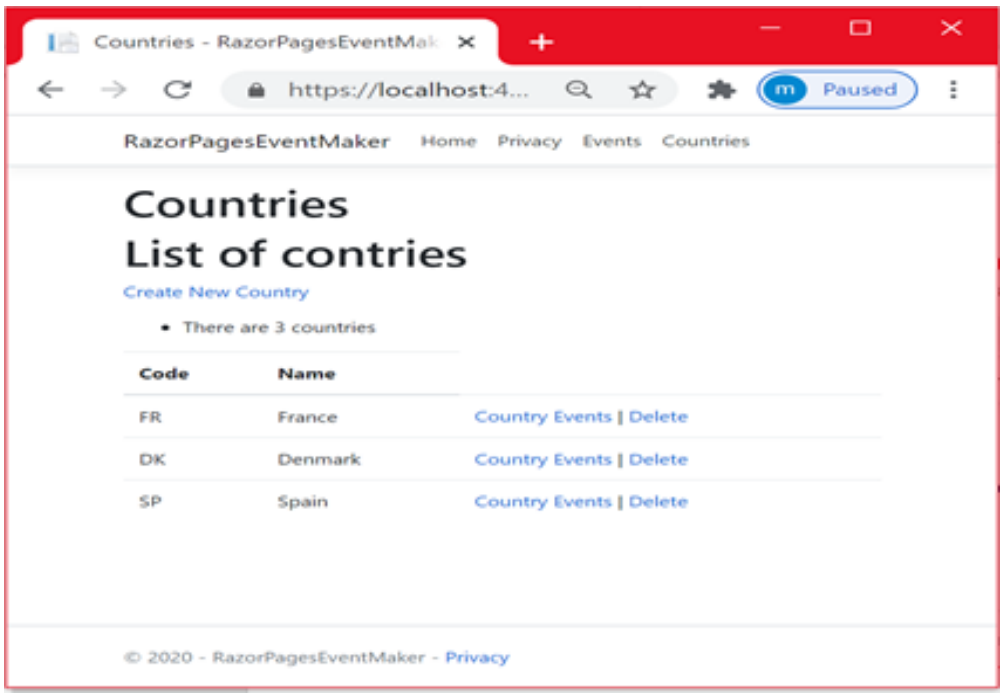
The code below shows how we passed the code from the “CountryIndex” page to the “CountryEvents”page as part of the URL (QueryString method as mentioned earlier). We choose this method simply because there is no risk to pass the code of a country as part of the URL. Remember that you can also pass this data as the route data.

```
CountryIndex.cshtml*
32          @foreach (var item in Model.Countries)
33          {
34              <tr>
35                  <td>
36                      @item.Code
37                  </td>
38                  <td>
39                      @item.Name
40                  </td>
41                  <td>
42                      <a asp-page="CountryEvents" asp-route-Code="@item.Code">Country Events</a> |
43                      <a asp-page="DeleteCountry" asp-route-Code="@item.Code">Delete</a>
44                  </td>
45              </tr>
46          }
47      </tbody>
48  </table>
```

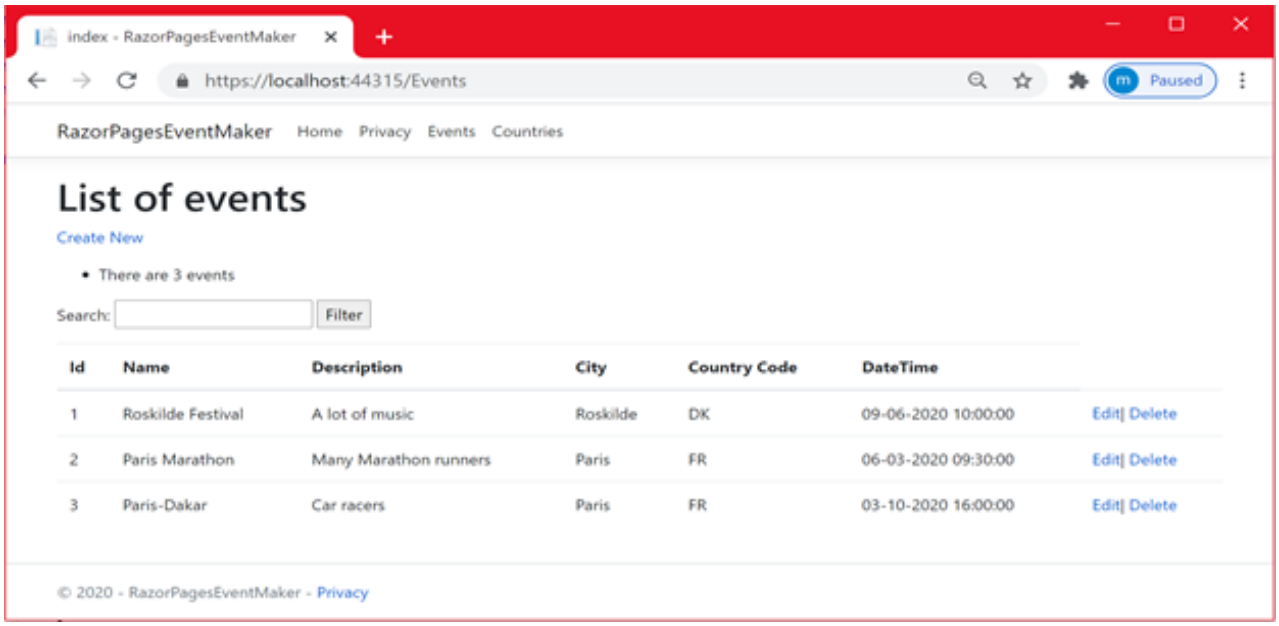
I think we are done . **Let us run the application**. The Front page (index.cshtml) is shown below



Notice the "**Countries**" link added to the top menu. Let us first click on this link to get an overview of the present countries in the list. There are 3 countries at the moment.



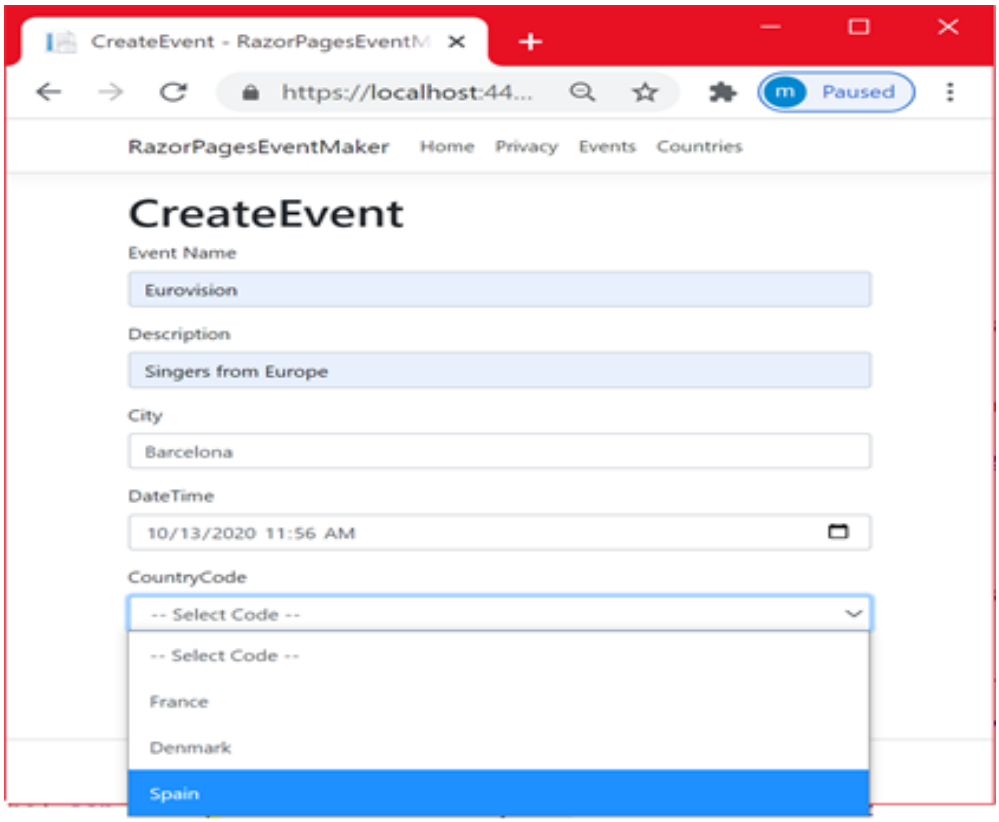
Let us shift to the "**Events**" link and click on it. The list of events is displayed. The output is shown below.



As can be seen , there are 2 events in France and 1 event in Denmark and no event in Spain

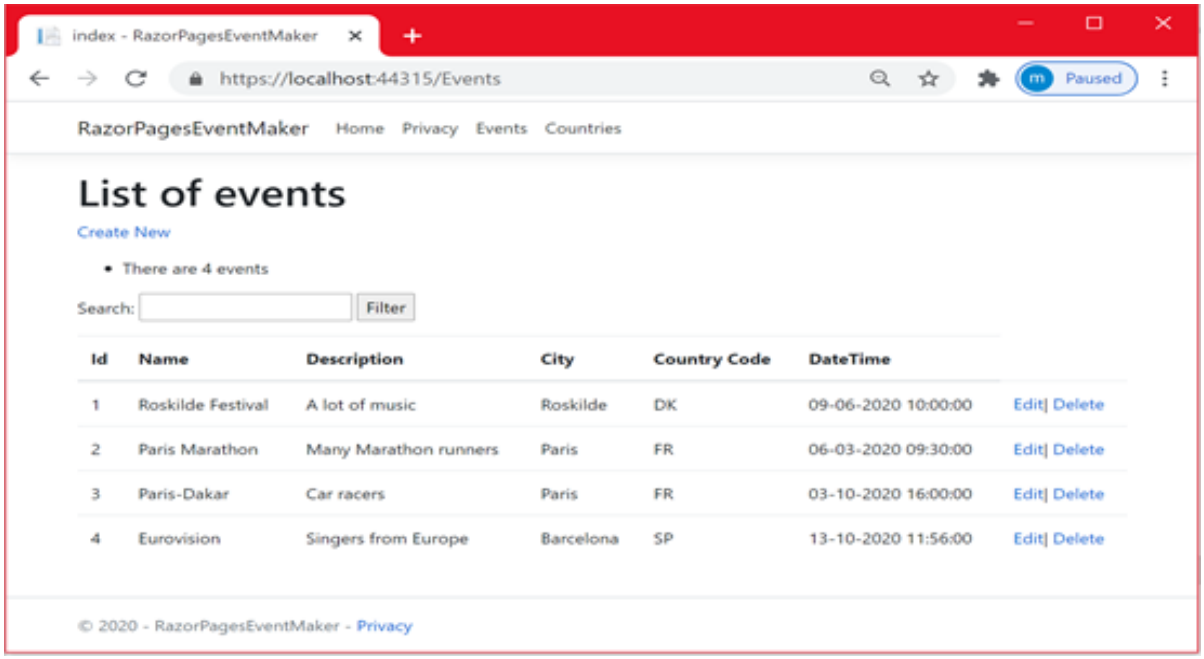
We can perform **CRUD operations** with no problems. With edit and delete, you need to add html elements that correspond to the “CountryCode” property. However, it is interesting to look at how we create a new event because this time, it requires a country code.

On the **Events page** , let us click on the “**Create New**” link.

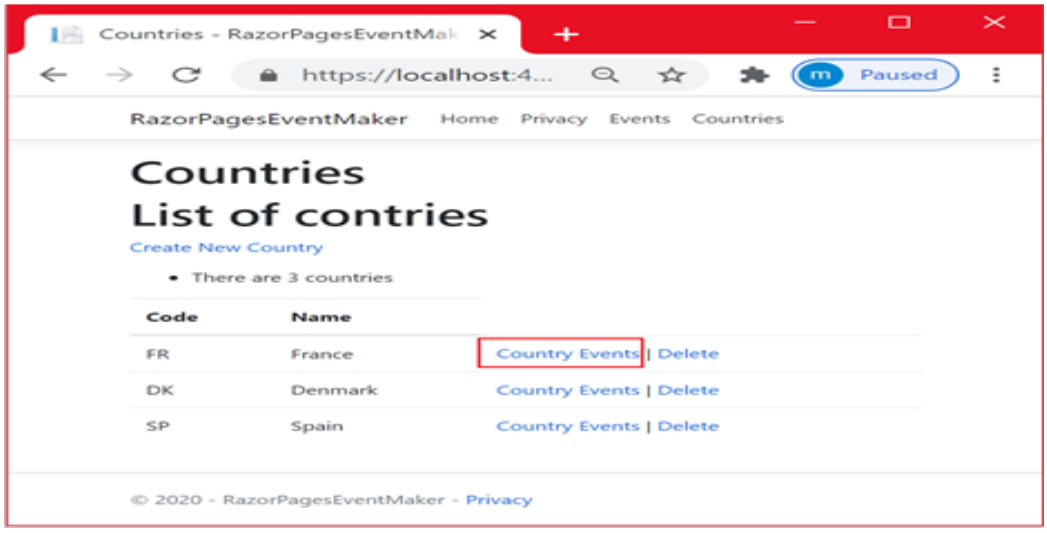


As can be seen , I did fill up all the data. The city is hard coded as “Barcelona”. So I need to select “Spain”as the country. The country is selected from a list using the Select element we covered earlier. I know that it is not the great way to do things but we will live with it.

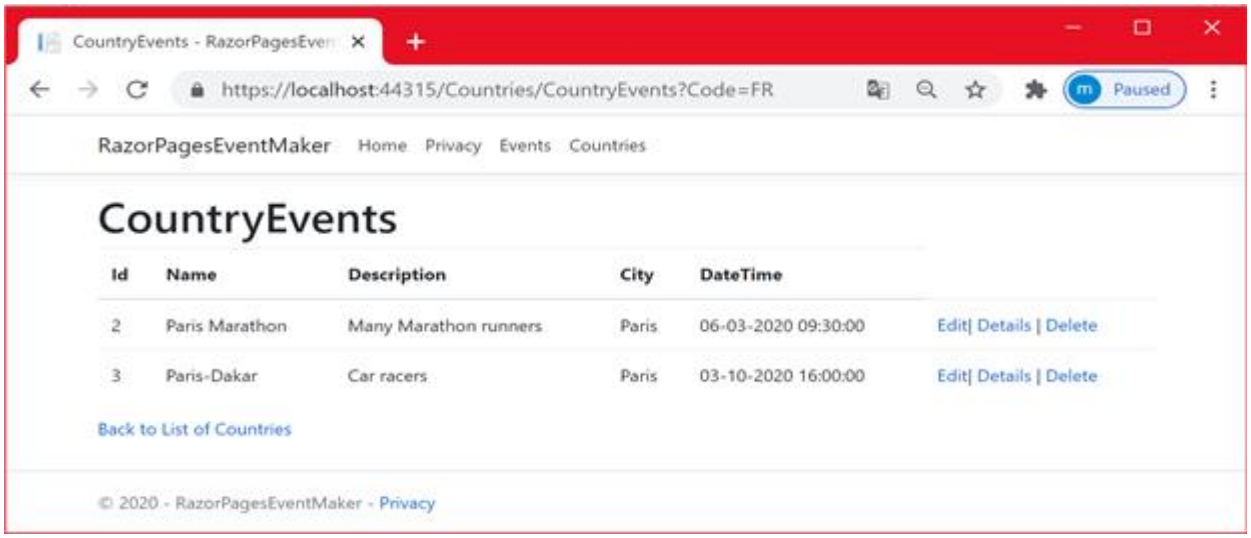
We then click on the “**Save**” button. The output is shown below. As you can see, the new event is created.



Now, for each country, we want to display the organized events and in which city they occur. We know that France organizes 2 events, Spain organizes 1 event and Denmark organizes 1 event. Let us click on the “**Countries**'' link.



To display the events organized in France, click on the "**Country Event**s" link. The output is shown below.



As you can see, only the events organized in France are displayed along with which city the event is taking place. That's it we are done. I hope that you get the hang of everything. Let us draw some interesting conclusions:

- Note that the 1-many relationship is the most commonly encountered and used relationship. Indeed, any many-many relationship can be resolved in two 1-many relationships.
- You may notice that most of the code that is applied to the Event objects is also applied to the Country objects. Having similar code is not efficient. We can achieve an efficient design by making this code generic so that it can be applied to any type. It is unfortunately part of the second semester curriculum.
- The implementation in this chapter considers the Fake Services as data access layers. The implementation, using json file, is not going to be a challenge.

References

- https://docs.microsoft.com/en-us/aspnet/core/data/ef-rp/update-related-data?view=aspnetcore-3.1
- https://www.learnrazorpages.com/razor-pages/tag-helpers/select-tag-helper

Chapter 14: building a real application 1-* relationship (one single json file using Dictionary as the data structure)

Introduction

In the previous chapter , we implemented the 1-* relationship using the Fake data access layer service using a list as the data structure. In this chapter, we are going to use a **Dictionary** instead of a list. We are going to save data in a json file.

Let us start coding. We are going to consider **one single json** file. We created the following json file . Notice the structure of such a json file. The json file encapsulates a **dictionary structure** .

```
{
  "FR": {
    "Code": "FR",
    "Name": "France",
    "EventList": {
      "1": {
        "Id": 1,
        "CountryCode": "FR",
        "Name": "Marathon paris",
        "Description": "Runners",
        "City": "Lille",
        "DateTime": "2020-10-01T00:00:00"
      }
    }
  },
  "DK": {
    "Code": "DK",
    "Name": "Denmark",
    "EventList": {
      "2": {
        "Id": 2,
        "CountryCode": "DK",
        "Name": "Distortion",
        "Description": "a lot of people",
        "City": "Copenhagen",
        "DateTime": "2020-11-24T09:49:00"
      }
    }
  }
}
```

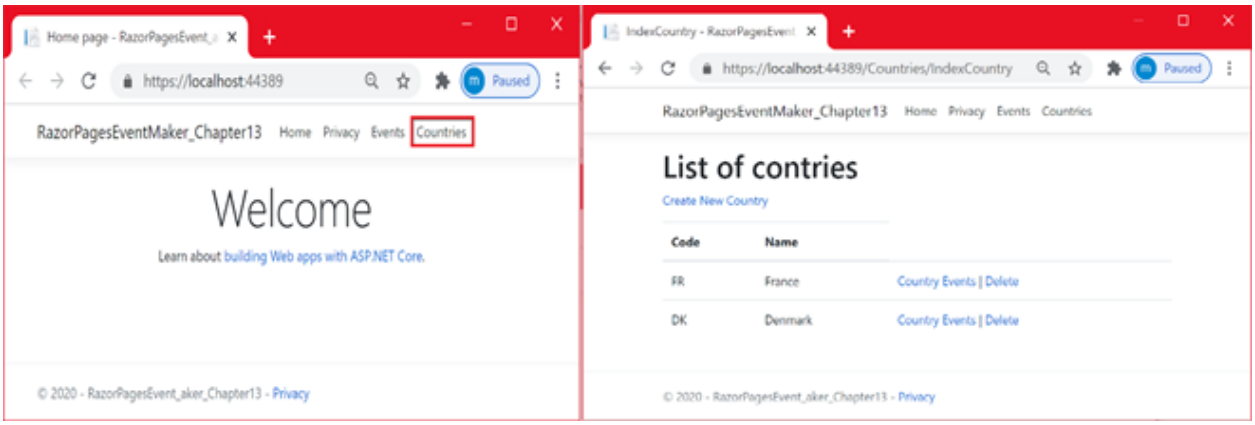
The only code that we are going through is the data access layer code **using dictionarie**s. The other code is 99.999% the same as the one seen in the previous chapter.

Managing Countries

Displaying the list of Countries

Should I explain the code ? **No**. You are very familiar with this code . Try to figure out what the code is doing. I am not either explaining the code in the following sections . Figure out yourself what the code is doing

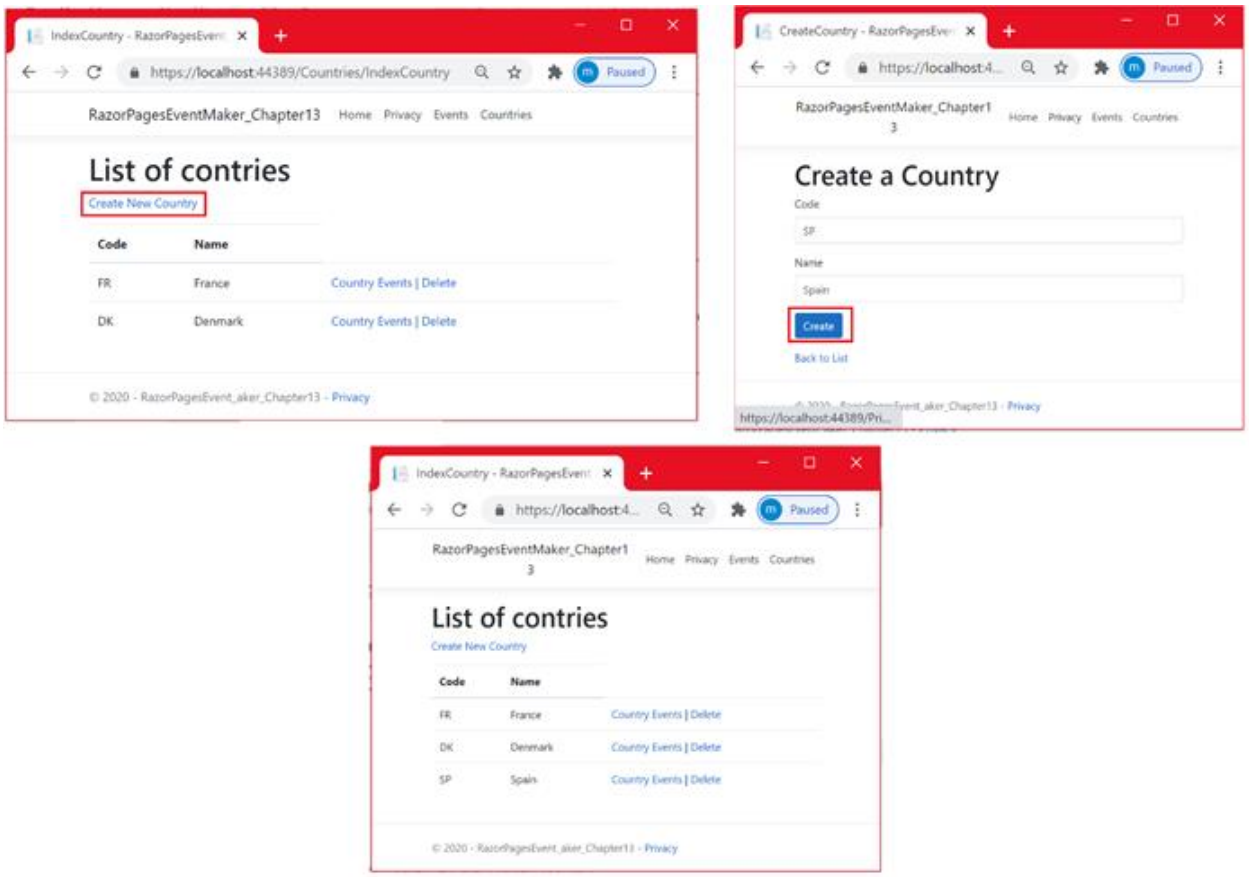
Let us run the application. The figure below shows the output when we click on the "**Countries**" link.



Add new Country

Figure out yourself what the code is doing

Let us run the application.



Display the list of events in a specific country: I want to display the events that take place in France.

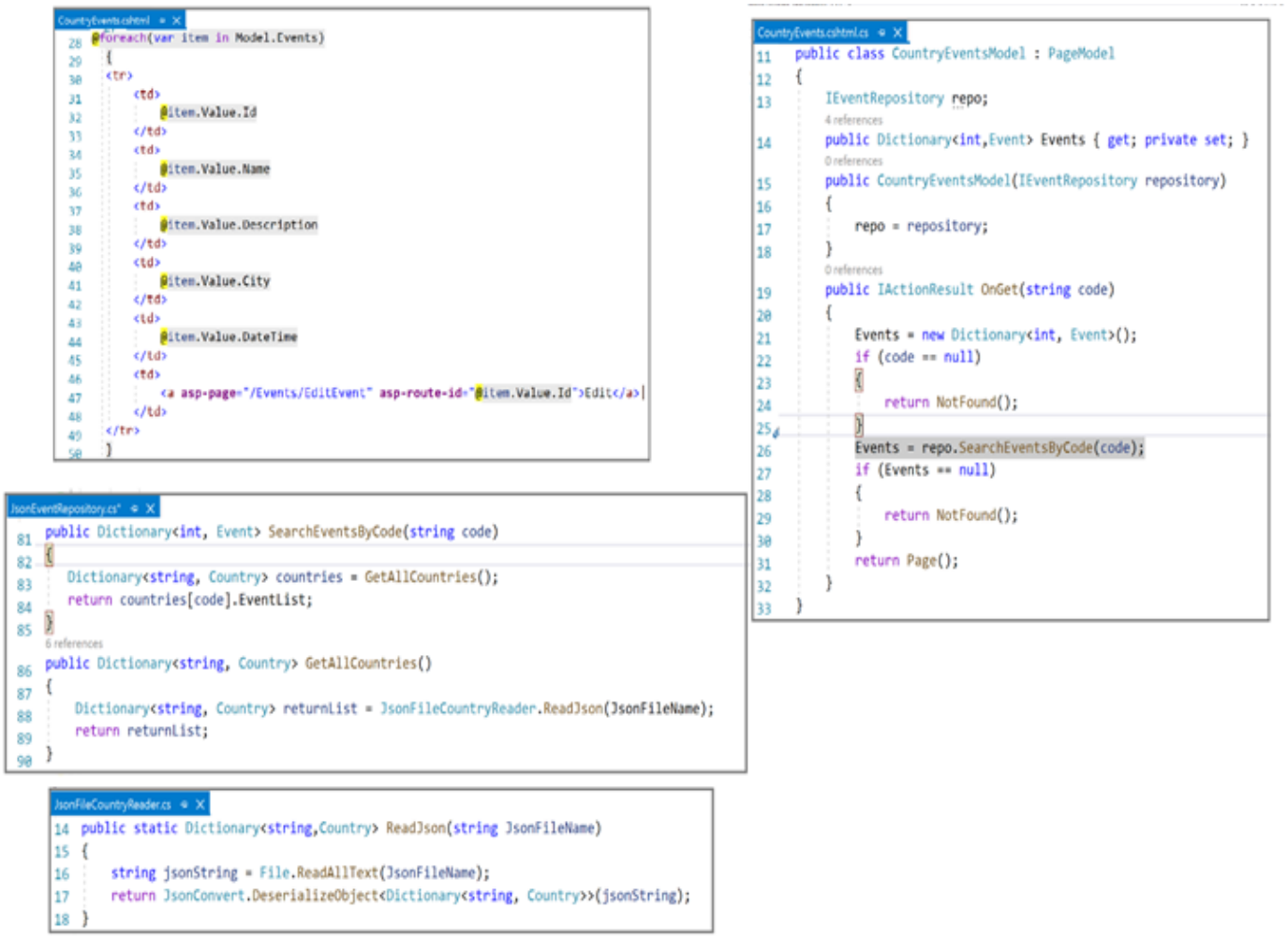
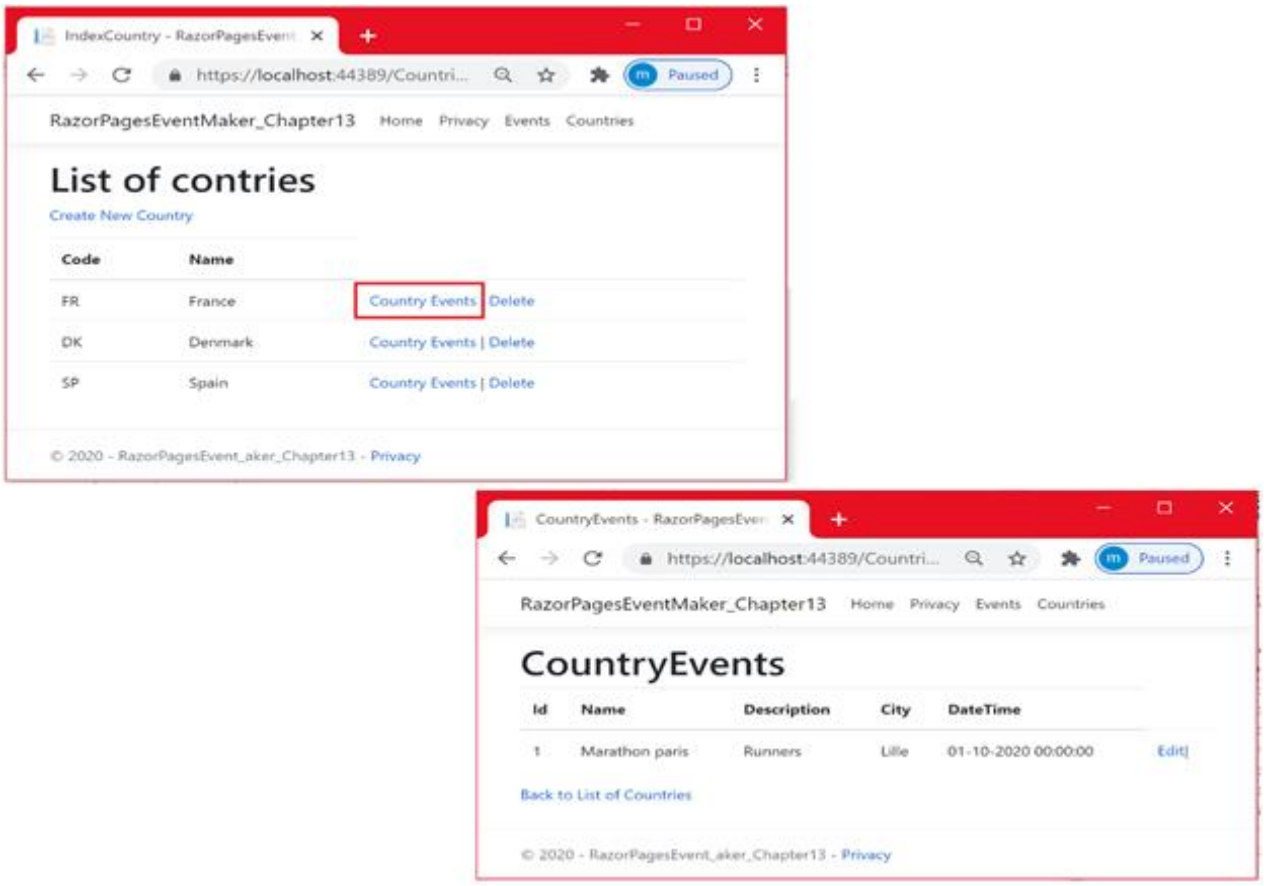


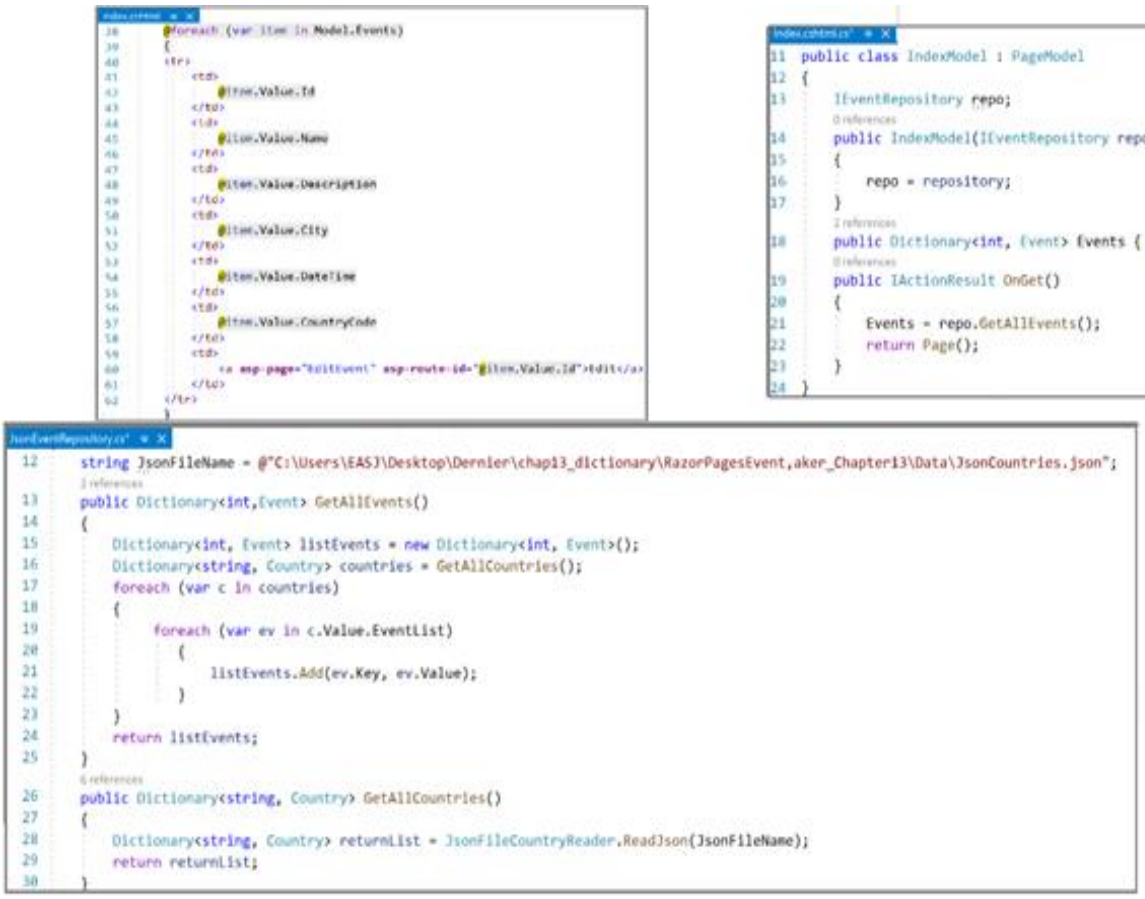
Figure out yourself what the code is doing

Let us run the application. The figure below shows the output of searching for the events that take place in France along with the city in which the event is taking place.



Managing Events

Displaying all the events



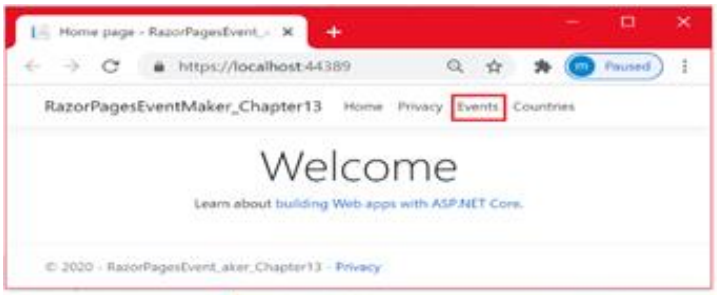
```
@foreach (var item in Model.Events)
{
<tr>
    <td>
        @item.Value.Id
    </td>
    <td>
        @item.Value.Name
    </td>
    <td>
        @item.Value.Description
    </td>
    <td>
        @item.Value.City
    </td>
    <td>
        @item.Value.DateTime
    </td>
    <td>
        @item.Value.CountryCode
    </td>
    <td>
        <a asp-page="EditEvent" asp-route-id="@item.Value.Id">Edit</a>
    </td>
</tr>
}
```

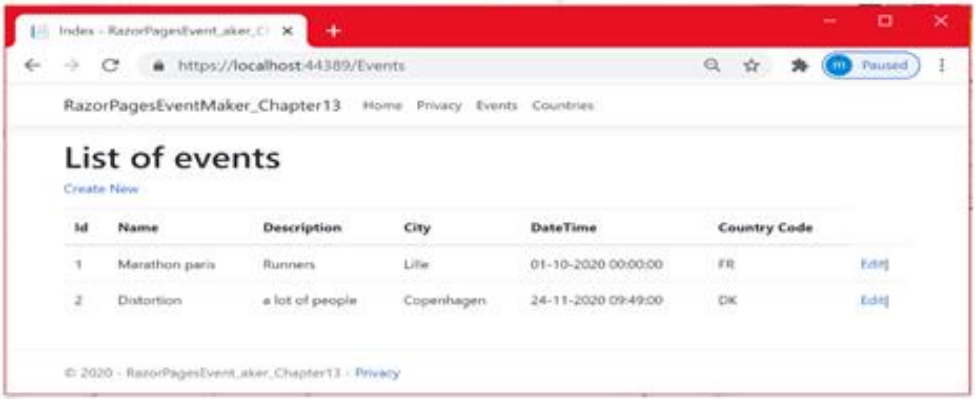
```
public class IndexModel : PageModel
{
    IEventRepository repo;
    0 references
    public IndexModel(IEventRepository repository)
    {
        repo = repository;
    }
    2 references
    public Dictionary<int, Event> Events { get; private set; }
    0 references
    public IActionResult OnGet()
    {
        Events = repo.GetAllEvents();
        return Page();
    }
}
```

```
string JsonFileName = @"C:\Users\EASJ\Desktop\Dernier\chap13_dictionary\RazorPagesEvent,aker_Chapter13\Data\JsonCountries.json";
1 reference
public Dictionary<int,Event> GetAllEvents()
{
    Dictionary<int, Event> listEvents = new Dictionary<int, Event>();
    Dictionary<string, Country> countries = GetAllCountries();
    foreach (var c in countries)
    {
        foreach (var ev in c.Value.EventList)
        {
            listEvents.Add(ev.Key, ev.Value);
        }
    }
    return listEvents;
}
6 references
public Dictionary<string, Country> GetAllCountries()
{
    Dictionary<string, Country> returnList = JsonFileCountryReader.ReadJson(JsonFileName);
    return returnList;
}
```

Figure out yourself what the code is doing.

Let us run the application. The figure below shows the output of displaying all events

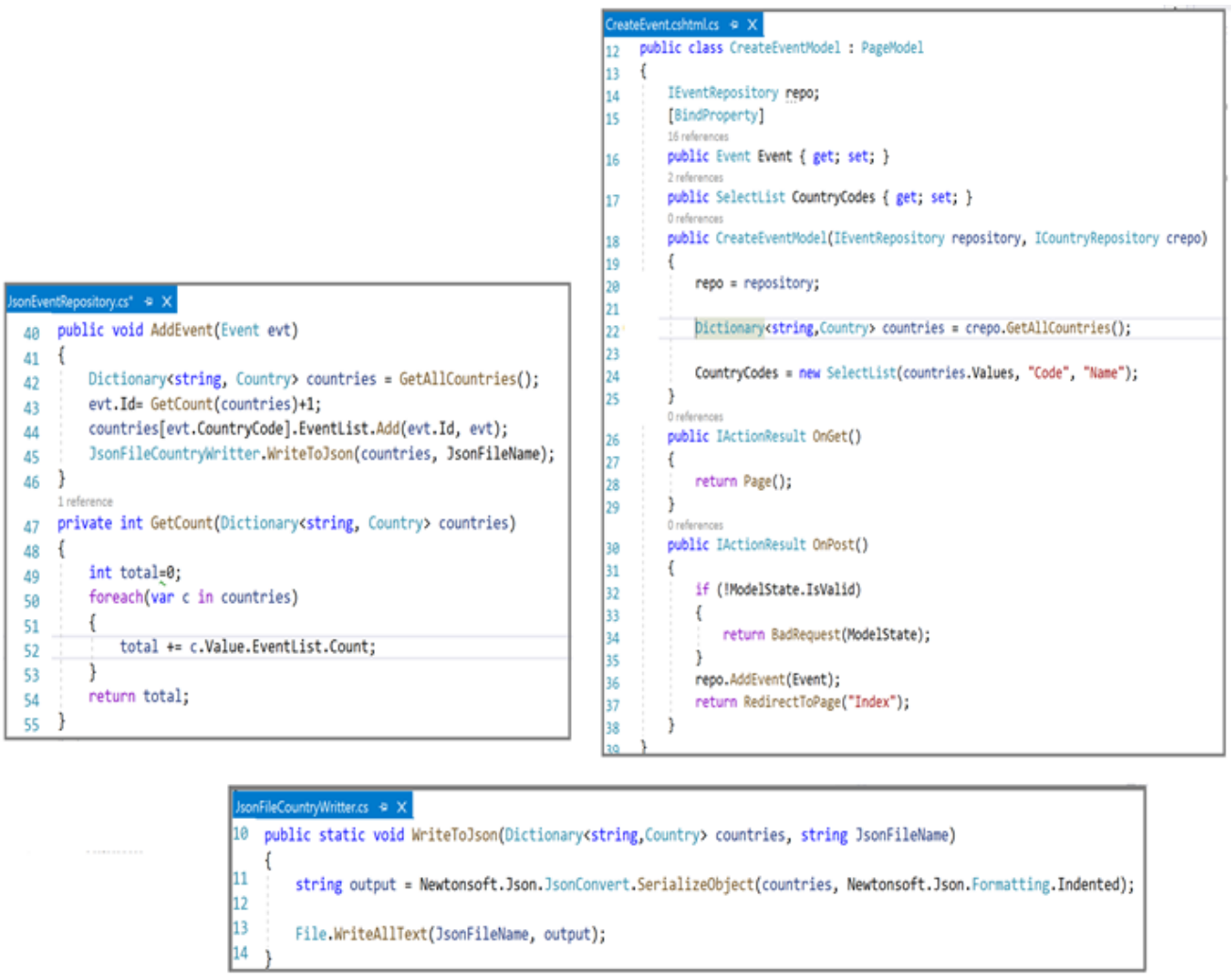




Add a specific event to a specific country

CreateEvent.cshtml*

```
@page
@model RazorPagesEventMaker_Chapter13.CreateEventModel
@{
    ViewData["Title"] = "CreateEvent";
}
<h1>CreateEvent</h1>
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="None " class="text-danger">
            </div>
            <div class="form-group">
                <label asp-for="@Model.Event.Name" class="control-label"></label>
                <input asp-for="@Model.Event.Name" class="form-control" />
                <span asp-validation-for="@Model.Event.Name" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="@Model.Event.Description" class="control-label"></label>
                <input asp-for="@Model.Event.Description" class="form-control" />
                <span asp-validation-for="@Model.Event.Description" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="@Model.Event.City" class="control-label"></label>
                <input asp-for="@Model.Event.City" class="form-control" />
                <span asp-validation-for="@Model.Event.City" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="@Model.Event.DateTime" class="control-label"></label>
                <input asp-for="@Model.Event.DateTime" class="form-control" />
                <span asp-validation-for="@Model.Event.DateTime" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="@Model.Event.CountryCode" class="control-label"></label>
                <select asp-for="@Model.Event.CountryCode" class="form-control"
                        asp-items="@Model.CountryCodes">
                    <option value="">-- Select Code --</option>
                </select>
                <span asp-validation-for="@Model.Event.CountryCode" class="text-danger" />
            </div>
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>
<div>
    <a asp-page="Index">Back to List</a>
</div>
```



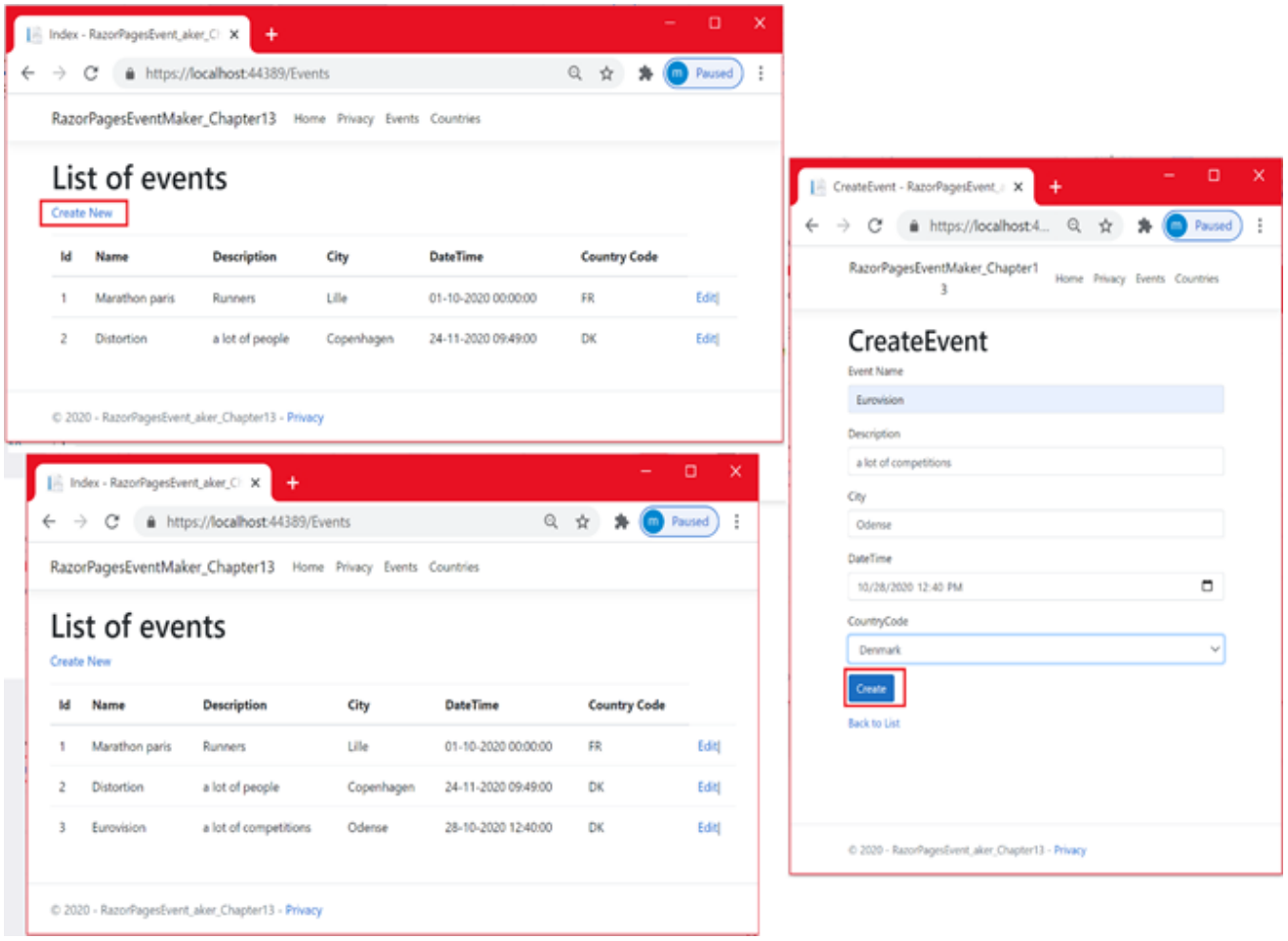
```
JsonEventRepository.cs*
40  public void AddEvent(Event evt)
41  {
42      Dictionary<string, Country> countries = GetAllCountries();
43      evt.Id= GetCount(countries)+1;
44      countries[evt.CountryCode].EventList.Add(evt.Id, evt);
45      JsonFileCountryWritter.WriteToJson(countries, JsonFileName);
46  }
    1 reference
47  private int GetCount(Dictionary<string, Country> countries)
48  {
49      int total=0;
50      foreach(var c in countries)
51      {
52          total += c.Value.EventList.Count;
53      }
54      return total;
55  }
```

```
CreateEvent.cshtml.cs
12  public class CreateEventModel : PageModel
13  {
14      IEventRepository repo;
15      [BindProperty]
        16 references
16      public Event Event { get; set; }
        2 references
17      public SelectList CountryCodes { get; set; }
        0 references
18      public CreateEventModel(IEventRepository repository, ICountryRepository crepo)
19      {
20          repo = repository;
21
22          Dictionary<string,Country> countries = crepo.GetAllCountries();
23
24          CountryCodes = new SelectList(countries.Values, "Code", "Name");
25      }
        0 references
26      public IActionResult OnGet()
27      {
28          return Page();
29      }
        0 references
30      public IActionResult OnPost()
31      {
32          if (!ModelState.IsValid)
33          {
34              return BadRequest(ModelState);
35          }
36          repo.AddEvent(Event);
37          return RedirectToPage("Index");
38      }
39  }
```

```
JsonFileCountryWritter.cs
10  public static void WriteToJson(Dictionary<string,Country> countries, string JsonFileName)
    {
11      string output = Newtonsoft.Json.JsonConvert.SerializeObject(countries, Newtonsoft.Json.Formatting.Indented);
12
13      File.WriteAllText(JsonFileName, output);
14  }
```

Figure out yourself what the code is doing

Let us run the application and try to create a new event in Denmark , then display all events.



Edit an event

Step 1: Getting the selected event

EditEvent.cshtml*

```
@page "{id}"
@model RazorPagesEventMaker_Chapter13.EditEventModel
@{
    ViewData["Title"] = "EditEvent";
}
<h1>EditEvent</h1>
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <input type="hidden" asp-for="@Model.Event.Id" />
            <div class="form-group">
                <label asp-for="@Model.Event.Name" class="control-label"></label>
                <input asp-for="@Model.Event.Name" class="form-control" />
                <span asp-validation-for="@Model.Event.Name" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="@Model.Event.Description" class="control-label"></label>
                <input asp-for="@Model.Event.Description" class="form-control" />
                <span asp-validation-for="@Model.Event.Description" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="@Model.Event.City" class="control-label"></label>
                <input asp-for="@Model.Event.City" class="form-control" />
                <span asp-validation-for="@Model.Event.City" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="@Model.Event.DateTime" class="control-label"></label>
                <input asp-for="@Model.Event.DateTime" class="form-control" />
                <span asp-validation-for="@Model.Event.DateTime" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="@Model.Event.CountryCode" class="control-label"></label>
                <input asp-for="@Model.Event.CountryCode" class="form-control" />
                <span asp-validation-for="@Model.Event.CountryCode" class="text-danger"></span>
            </div>

            <div class="form-group">
                <input type="submit" value="Save" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>
<div>
    <a asp-page="index">Back to List</a>
</div>
```

Index.cshtml

```
@foreach (var item in Model.Events)
{
<tr>
    <td>
        @item.Value.Id
    </td>
    <td>
        @item.Value.Name
    </td>
    <td>
        @item.Value.Description
    </td>
    <td>
        @item.Value.City
    </td>
    <td>
        @item.Value.DateTime
    </td>
    <td>
        @item.Value.CountryCode
    </td>
    <td>
        <a asp-page="EditEvent" asp-route-id="@item.Value.Id">Edit</a>
    </td>
</tr>
}
```

JsonEventRepository.cs*

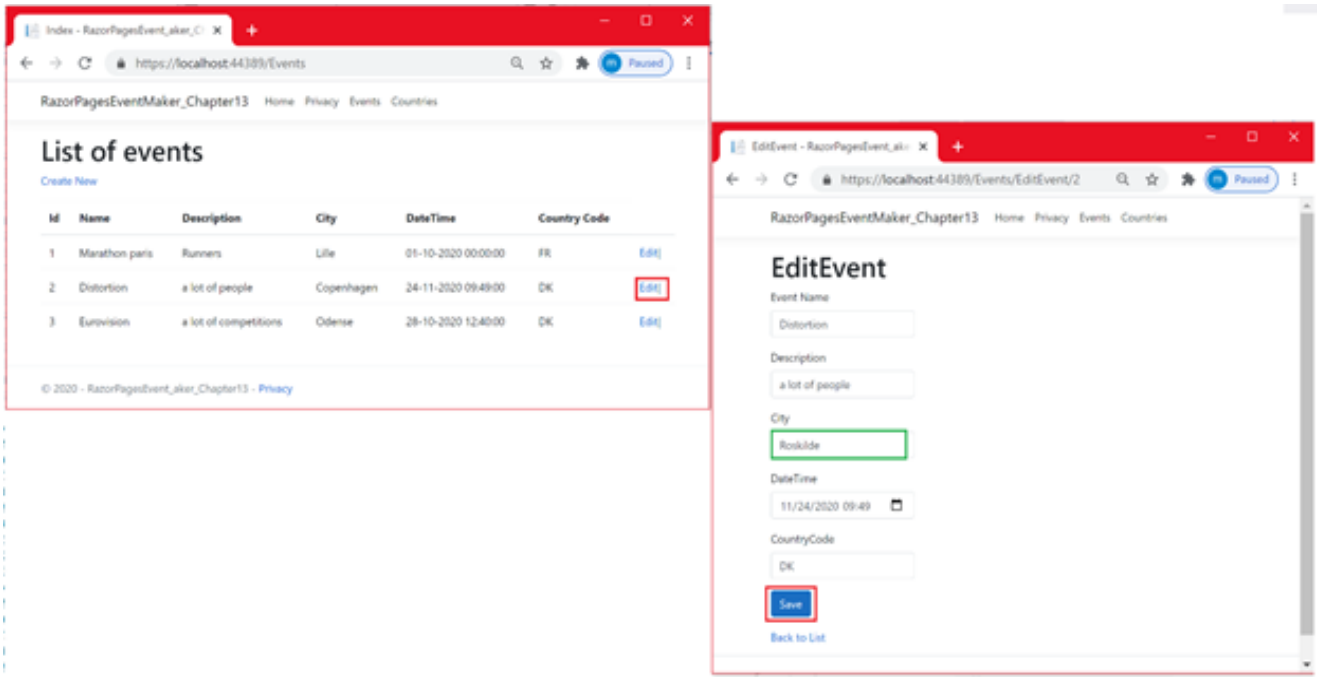
```
public Event GetEvent(int id)
{
    foreach(var c in GetAllCountries())
    {
        foreach (var v in c.Value.EventList)
        {
            if (v.Key == id)
            return v.Value;
        }
    }
    return new Event();
}
0 references
public Dictionary<string, Country> GetAllCountries()
{
    Dictionary<string, Country> returnList = JsonFileCountryReader.ReadJson(JsonFileName);
    return returnList;
}
```

EditEvent.cshtml.cs*

```
public class EditEventModel : PageModel
{
    IEventRepository repo;
    [BindProperty (SupportsGet =true)]
    19 references
    public Event Event { get; set; }
    0 references
    public EditEventModel(IEventRepository repository)
    {
        repo = repository;
    }
    0 references
    public IActionResult OnGet(int id)
    {
        Event = repo.GetEvent(id);
        if (Event == null)
        {
            return null;
        }
        return Page();
    }
```

Figure out yourself what the code is doing.

Run the application



Step 2: Updating the event

EditEvent.cshtml.cs*

```
public class EditEventModel : PageModel
{
    IEventRepository repo;
    [BindProperty (SupportsGet =true)]
    19 references
    public Event Event { get; set; }
    0 references
    public EditEventModel(IEventRepository repository)
    {
        repo = repository;
    }
    0 references
    public IActionResult OnGet(int id)
    {
        Event = repo.GetEvent(id);
        if (Event == null)
        {
            return null;
        }
        return Page();
    }
    0 references
    public IActionResult OnPost()
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }
        repo.UpdateEvent(Event);

        return RedirectToPage("Index");
    }
}
```

JsonEventRepository.cs

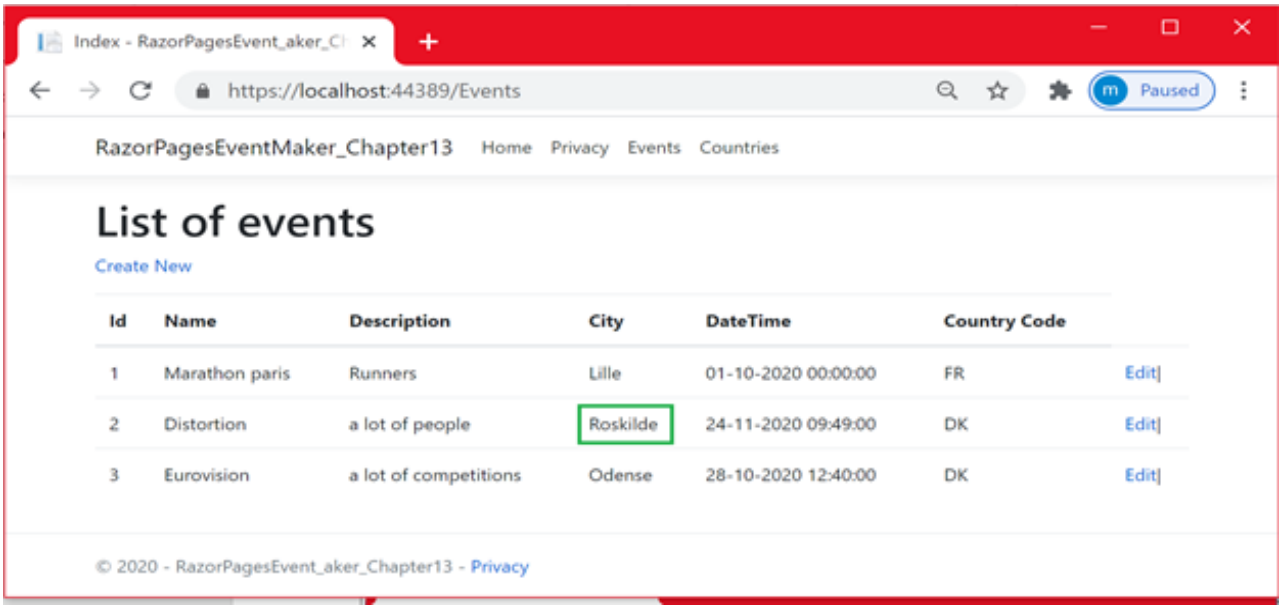
```
public void UpdateEvent(Event @evt)
{
    Dictionary<string, Country> countries = GetAllCountries();
    Event ev = countries[@evt.CountryCode].EventList[@evt.Id];
    ev.Name = evt.Name;
    ev.City = evt.City;
    ev.Description = evt.Description;
    ev.DateTime = evt.DateTime;
    ev.CountryCode = evt.CountryCode;
    JsonFileCountryWritter.WriteToJson(countries, JsonFileName);
}
```

JsonFileCountryWritter.cs

```
4 references
public static void WriteToJson(Dictionary<string,Country> countries, string JsonFileName)
{
    string output = Newtonsoft.Json.JsonConvert.SerializeObject(countries, Newtonsoft.Json.Formatting.Indented);

    File.WriteAllText(JsonFileName, output);
}
```

Figure out yourself what the code is doing.

Run the application



As you can see, we change the city from Copenhagen into Roskilde.

THE END OF PART I