Poul Klausen

C# 12

WWW the Client Side

Software Development

bookboon
LEARNING

POUL KLAUSEN

C# 12: WWW THE CLIENT SIDE

SOFTWARE DEVELOPMENT

C# 12: WWW the Client Side: Software Development
1st edition

ISBN 978-87-403-3854-6

CONTENTS

FOREWORD

This book is the twelfth in a series of books on software development. The programming language is C#, and the language and its syntax and semantics fills obviously much, but the books have also largely focus on the process and how to develop good and robust applications. This book is similar to the book C# 11 about development of web applications, but focusing on the client side. This means that key topics are style sheets and JavaScript, and in particular, the last part fills. The primary purpose of the book is to show that it is possible to perform complex client-side programming using JavaScript. The book requires knowledge of programming of the server side similar to what has been dealt with in the previous book, and together, the two books provide a relatively basic introduction to web application development.

As the title says this series of books deals with software development, and the goal is to teach the reader how to develop applications in C#. It can be learned by reading about the subject and by studying complete sample programs, but most importantly by yourself to do it, and write your own programs from scratch. Therefore, an important part of the books is exercises and problems, where the reader has to write programs that correspond to the substance being treated in the books. All books in the series is built around the same skeleton and will consist of text and examples, exercises and problems that are placed in the text where they naturally belongs. The difference between exercises and problems is that the exercises largely deals with repetitions of the substance that is presented in the text, and furthermore it is relatively accurately described what to do. Problems are in turn more loosely described, and are typically a little bigger and there is rarely any clear best solution. These are books to be read from start to finish, but the many code examples, including exercises and problems plays a central role, and it is important that the reader predict in detail studying the code for the many examples and also solves the exercises and problems or possibly just studying the recommended solutions.

All books ends with a larger sample program, which focus primarily is on the process and an explanation of how the program is written. On the other hand appears the code only to a limited extent, if at all, and the reader should instead study the finished program code, perhaps while testing the program. In addition to show the development of programs that are larger than the examples, which otherwise is presented, the aim of the concluding examples also is to show program examples from varying fields of application.

All sample programs are developed and tested on a Windows machine, and the development tool is Visual Studio. Therefore, you must have Visual Studio installed and Microsoft provides a free version that is fully adequate. Visual Studio is an integrated development environment that provides all the tools needed to develop programs. I do not want to deal with the use of Visual Studio, which is quite straightforward, and it is a program that you quickly become familiar with.

Finally a little about what the books are not. It is not "a how to write" or for that matter a reference manuals to C#, but it is as the title says books on software development. It is my hope that the reader when reading the books and through the many examples can find inspiration for how to write good programs, but also can be used as a source collection with a number of examples of solutions to concrete everyday programming problems that you regularly face as a software developer.

1 INTRODUCTION

In the book C# 11, I have provided a basic introduction to Web Application Development in C#. I have primarily processed the server side, where a browser sending a request to a web server, that dynamically dependent on the data sent with the request, creates the HTML document to be sent as a response to the client. This client / server principle is also the whole idea of web applications and, in fact, you can stop here as all web applications could in principle be developed according to the pattern outlined in C# 11. However, there are much more to learn and here I would like to mention:

1. The visual and the possibilities for developing web applications (web sites) with an attractive user interface, and especially user interfaces that are easy to maintain.
2. Client programming, where part of the code is executed by the browser for the purpose of preventing a request to the server every time something happens on the client page.

The first is primarily a matter of which components (items) are provided and what options are available to customize them to exactly the look and feel that you may be interested in. It's about *cascading style sheets* and HTML 5. Both of them do not have anything to do with C#, but are subjects that you as a web developer must be aware of. And then the visual, that more than anything else is a question about skill and knowledge about what is a good user interface and how to design web user interfaces. Unfortunately, I only have limited knowledge about what is a good user interface, so I just have to introduce the technique.

The other is primarily about *JavaScript*, which is program code sent as script (text) along with the server's response, and which is code executed by the browser without sending requests to the server. That it's script code means that it's only text and not binary code, why it is generally perceived as harmless and cannot contain malicious code, but also because it's limited what to do with JavaScript and basically you can only manipulate the HTML document's elements. JavaScript has gained interest, among other things, because the browsers interpreters of JavaScript have been optimized, why JavaScript today is effective, but also because the language has evolved so that today you can write advanced JavaScript that also can be maintained. In spite of the name, JavaScript is not Java, and it is important to make it clear that it is a simple script language, but with the same syntax as Java - and hence the name. JavaScript will fill a part in this book.

The book will also contain topics that relate to the server, and in particular I will introduce server side JavaScript. I also will show how to use the Entity Framework to maintenance databases when developing web applications. In fact, the use of the Entity Framework as an API for database programming is so strongly integrated in the development of web applications that you cannot work as a web developer at all without using this API, something that I will elaborate on in the next book.

2 CASCADING STYLE SHEETS

In the previous book I have mentioned and used style sheets and I will start this book with a short introduction to style sheets. The idea is that using HTML you organize a document in elements, and using styles you define how these elements should be shown in the browser. As example I will use the application *HtmlApplication* from the previous book. I have added a new HTML document to the project and called it *styleddoc.html*. The content is the same as *index.html* and the goal is to show how the document's visual presentation can be changed using a style sheet. For that I have added a style sheet named *styles.css*. Initially, it is nothing but an empty file. The document that has to use a style sheet must have a *link* element in the header:

```
<head>
  <title>HTML Application</title>
  <meta charset="UTF-8">
  <link href="styles.css" rel="stylesheet" type="text/css"/>
</head>
```

The syntax should be correct and you can easily drag the file into the document with the mouse, and Visual Studio will automatically insert the correct link.

The idea of style sheets is as mentioned to separate content and presentation. A style sheet consists of styles identified by *selectors*, and as an example, I have added the following style for a *h1* element to the style sheet:

```
h1 {
  font-size: 36pt;
  font-weight: bold;
  color: darkslateblue;
}
```

Such a selector is called a *type selector* or an *element selector* and means that all of the document's *h1* elements will use the style that the selector defines. You can thus define how the headers of the document should be presented, and it will apply to all documents that uses that style sheet. Typically, you also wants to define a style for the body element, such as defining the font to be used as default:

```
body {
  font-family: serif;
  font-size: 10pt;
}
```

You can also define styles for *descendent selectors* that occur with a space between two selectors:

```
section p {
  margin-left: 20px;
}
```

This selector indicates that a paragraph element must have a left indentation of 20, but only for paragraphs that are nested in a *section* element.

You can also define a so-called ID selector, known as it starts with the character #

```
#content-list {
  font-size: 14pt;
  font-weight: bold;
  color: darkgray;
}
```

This selector is used by an element whose *id* is *content-list*, and as the element's *id* attribute should be unique, there is only one element in a document that directly can use this style. In this case, it is the following element:

```
<p id="content-list">Content</p>
```

Finally, you can define *class selectors*, starting with a dot:

```
.default-text {
  width: 800px;
  color: #222222;
}
```

An element can use a *class selector* by directly specifying it with a class attribute:

```
<p class="default-text">In the book Java 11, ... </p>
```

As another example of selectors, there is defined a *class* with the name *forword* which defines the color as green:

```
.forword {
  color: darkgreen;
}

.forword h5 {
  color: darkred;
}
```

The last style means that a *h5* element under an element whose *style* is *forword* appears in red text:

```
<article class="forword">
  <h4 id="forword">Forword</h4>
  <section>
    <h5>About this book</h5>
    <p>Here is the text</p>
  </section>
</article>
```

You can also define selectors that relates to attributes. As an example, the following style:

```
h4[id] {
  color: darkred;
}
```

means that all *h4* elements that have an *id* attribute must be displayed with a dark red text. As another example, the HTML element *abbr* can be used to display a tooltip:

```
<h5>HTML <abbr title="Introduction to HTML syntax">syntax</abbr></h5>
```

Here, the value of the attribute *title* will be displayed as a tooltip if the mouse is pointing to the word *syntax*. Consider the following styles:

```
abbr[title] {
  color: gray;
}

abbr[title]:hover {
  color: red;
}
```

They define that for all *abbr* elements with a *title* attribute, the text should appear gray and holding the mouse over the text (*mouse ower*), the text should appear red.

Attributes as selectors can be especially interesting for custom attributes. Consider the following paragraph:

```
<p data-marked>Here is the text</p>
```

For example, if you want all paragraphs with this attribute to appear with a large font, you can write:

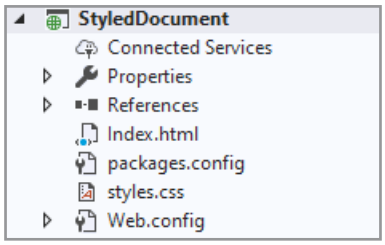
```
p[data-marked] {
  font-size: 48pt;
}
```

You are encouraged to study the final result (the document *styleddoc.html*) and especially the finished style sheet. Note that you from Visual Studio can open the document in the browser by right click om the document in Solution Explorer and select *View in Browser*.

2.1 MORE SELECTORS

The example *StyledDocument* consists of a single HTML document called *Index.html* as well as a style sheet. If you open the document in the browser, the result is as shown below. You must note that the project is an

ASP.NET Web Application (.NET Framework)



project and is created using Visual Studio as an empty application and to this project I have added a html document and a style sheet.

I do not want to show the document here, as there is not much news of explaining. The style sheet is as follows:

```
p {
  margin-left: 30px;
  font-size: 10px;
}

.queens {
  margin-left: 10px;
}

.queens > article > h2 {
  margin-left: 20px;
  font-size: 18px;
  color: red;
}

#regent + p {
  color: blue;
}

p > span {
  font-weight: bold;
  color: darkgreen;
}

#blueking ~ p {
  font-size: 12px;
}

* {
  color: gray;
}

.king {
  margin-left: 20px;
}

.king > * {
  color: orange;
}
```

```
p[data-info="to"] {
  color: magenta;
}

p[data-info="from"] {
  color: maroon;
}

p[data-info*="th"] {
  color: pink;
  font-weight: bold;
}
```

The first two styles do not require further explanation. The third identifies elements with the class *queens* including child elements of the type *article* and again including *h2* child elements, and then *h2* elements under *article* elements under elements with the class *.queens*.

The fourth style defines a paragraph that is next sibling to an element with *id regent*. The fifth style is used by a *span* element, which is child of a paragraph element. The sixth style is used by all elements that are siblings of an element with *id blueking*.

A * is used as a selector for all elements (rarely used directly), but in combination with other selectors it can be used to style all elements in a subtree. In this case, means

```
.king > *
```

all elements under elements with the class *king*. Finally, there are the three last styles that relates to attributes. Here they are used to a paragraph with the attribute *data-info*, and in particular you can select the attributes text:

- `[attr^="text"]` attributes which starts with text
- `[attr$="text"]` attributes which ends with text
- `[attr*="text"]` attributes which contains text

and there are actually a few more options.

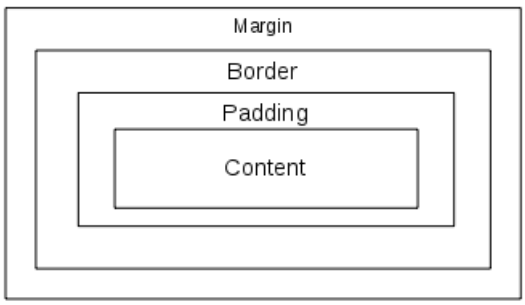
2.2 STYLES

Above I have shown examples of how to refer from a style sheet to the individual elements in the HTML tree, but there are quite a few other areas where styles are available, and where you should know what they means - and there are really many. Styling an HTML document is an extensive topic, and an adequate processing of styles is beyond the scope of this book. In this section I will outline some of the basic principles, but there is much more, and in general, in software development it is an area that requires considerable expertise before designing modern web applications. However, I would like to emphasize once again that the development tool usually supports styling and is a significant help in practice, although it is by no means always easy to understand the effects of the many possibilities.

Basics styling includes:

- how much the elements fill in the browser
- where the elements are located
- how the content of the individual elements is formatted, including fonts and colors

Basically, an HTML element appears as a box:



In the middle you have the content (if there is a content) and without having *padding*, which indicates how much space there should be outside of the content. For an element with a style, the *width* and *height* will in most cases mean the size of the content area inclusive padding, but it can be changed with *box-sizing*. Around the padding there may be a *border*, and finally there may be a *margin* about all of it. For example, if you specify a *background* color, it will relate to the entire *padding* area. You should especially note that the *margin* area is transparent and is thus only used to create spaces between the elements. You should be aware that if elements are vertically aligned and two margins (top and bottom) meet, they are automatically collapsed to one whose height is the largest of the two.

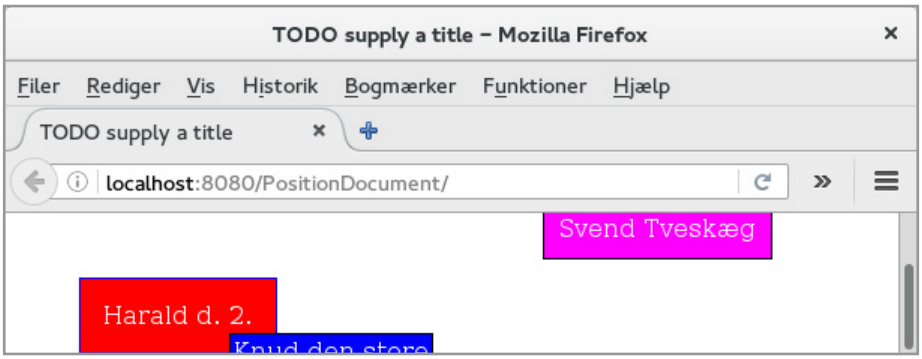
Many elements such as *p*, *h*, and *article* elements behave as described above and show the contents as a block and are therefore called *block boxes*. Others such as *strong* and *span* are called *inline boxes* as they format the content of a line. How the different elements show their content can be defined by *display* and thus override how they as default shows their content. Elements can be laid out by the browser in several ways, but the default is *static*, and for *block boxes*, it means that they are placed vertically underneath each other, whereas for *inline boxes* it means that they are laid out horizontally on the same line and wraps as required.

As mentioned, the position of elements is defined as *static*, which means that the elements float in the window, but you can define other options, as are best illustrated by an example. If you open the application *PositionDocument* in the browser, you get the window:

The elements are all *span* elements that are rendered in the following order:

1. Gorm den gamle
2. Harald Blåtand
3. Svend Tveskæg
4. Harald d. 2.
5. Knud den store
6. Hardeknud

Unless otherwise stated, they would be laid out in one row which would wrap if the line was too long. In this case, the two first and the last are laid out in default (*static*) positions, while the third has *relative* position, which means that it is laid out relative to the previous element, as is *Harald Blåtand*. You should note that the last element (*Hardeknud*) is laid out as if the element *Svend Tveskæg* was laid out in default position and that the position of the previous two is ignored as they are laid out as *absolute* and *fixed* position respectively. *Absolute* means that the element is placed in a fixed position relative to the upper left corner of the document. *Fixed* is a variant, but here the element is also placed in a fixed position, but this time relative to the part of the document that is visible. You can illustrate the difference by changing the size of the window and then scrolling it:

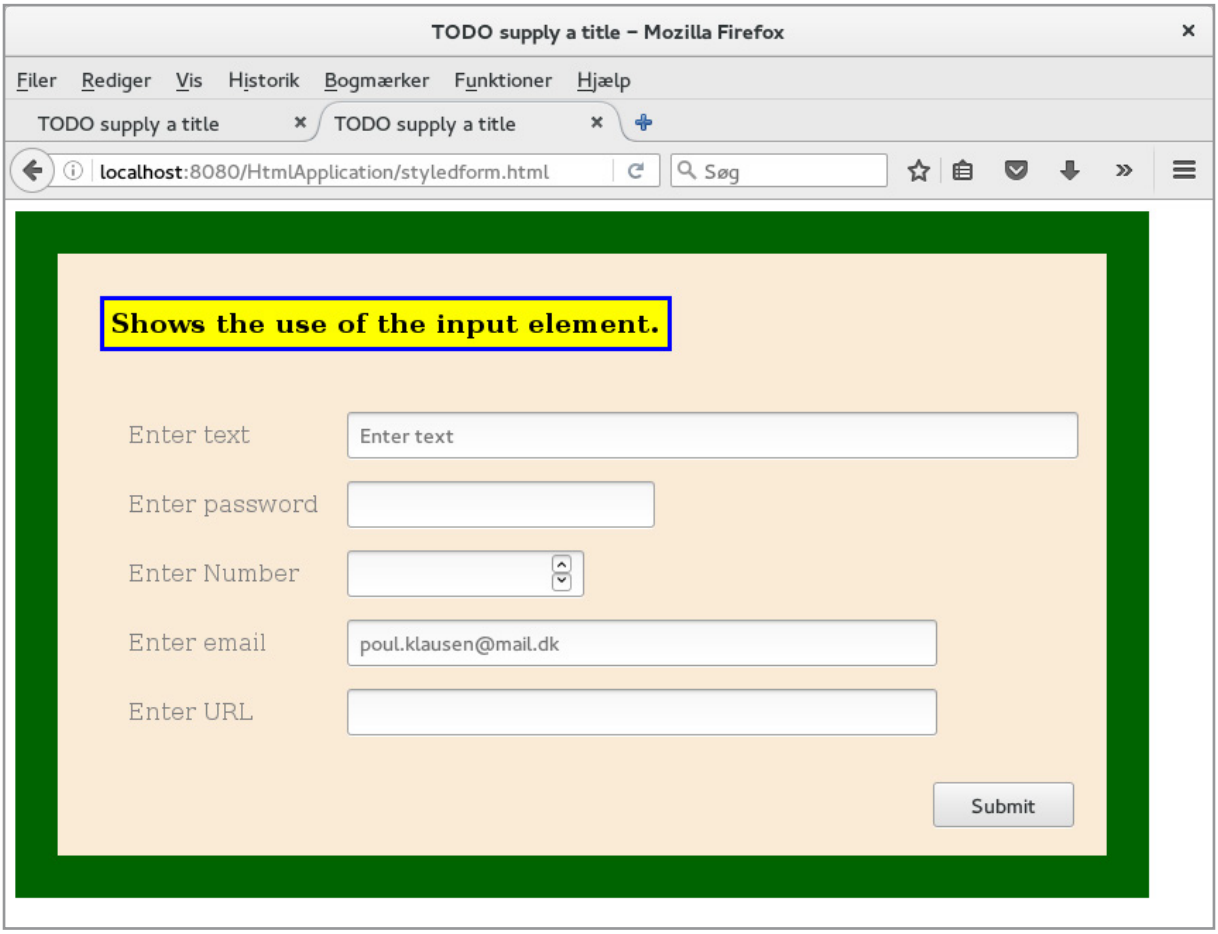


The code is shown below:

```
<!DOCTYPE html>
<html>
  <head>
    <title>TODO supply a title</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <style>
      .normal {
        background: yellow;
        border: 1px solid gray;
        padding: 5px;
      }
      .relative {
        background: magenta;
        color: white;
        border: 1px solid black;
        padding: 10px;
        position: relative;
        left: 50px;
        top: 30px;
      }
      .absolute {
        background: red;
        color: white;
        border: 1px solid blue;
        padding: 15px;
        position: absolute;
        left: 50px;
        top: 80px;
      }
      .fixed {
        background: blue;
        color: white;
        border: 1px solid black;
        padding: 2px;
        position: fixed;
        left: 150px;
        top: 80px;
      }
    </style>
```

```
  </head>
  <body>
    <span class="normal">Gorm den gamle</span>
    <span class="normal">Harald Blåtand</span>
    <span class="relative">Svend Tveskæg</span>
    <span class="absolute">Harald d. 2.</span>
    <span class="fixed">Knud den store</span>
    <span class="normal">Hardeknud</span>
  </body>
</html>
```

Styling includes much more than what has been said in this chapter, but the above should be enough to get an idea of what is possible. Often you are in the situation that you want to achieve a certain effect on an element, but you do not know what to write. Here is the best source actually the Internet, where there are countless examples of how to style different HTML elements. As a conclusion, I have added a document called *styledform.html* to the project *HtmlApplication*. The document shows the same form as previously mentioned, but this time no table is used and the elements are placed only by styling:

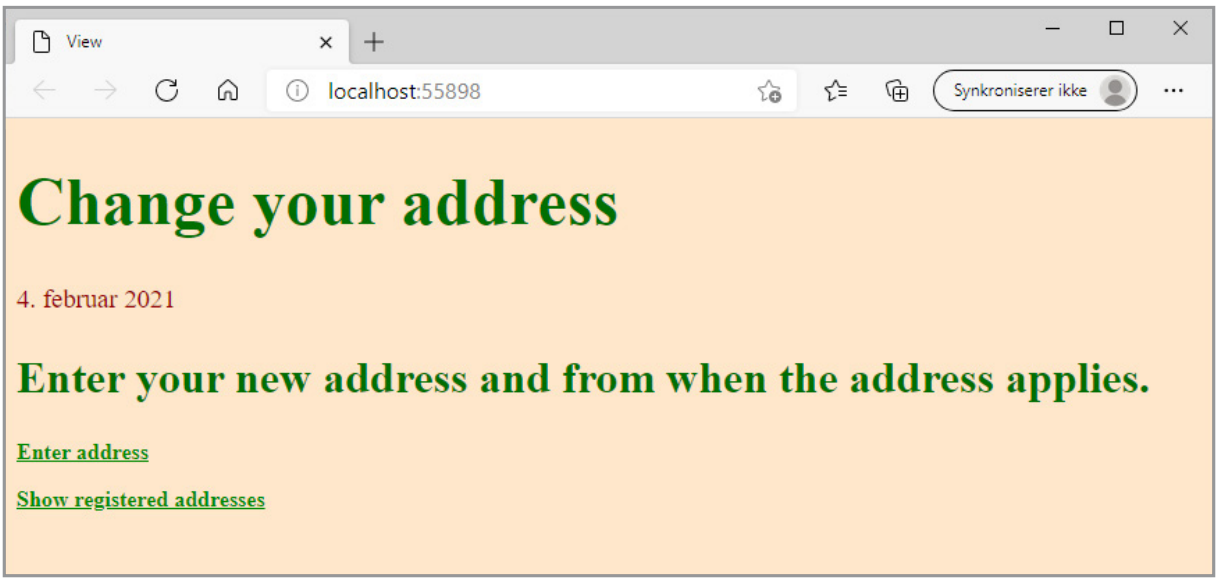


You are encouraged to study the document and especially the corresponding style sheet, and how the individual elements are styled.

2.3 USING A CLIENT-SIDE PACKAGE

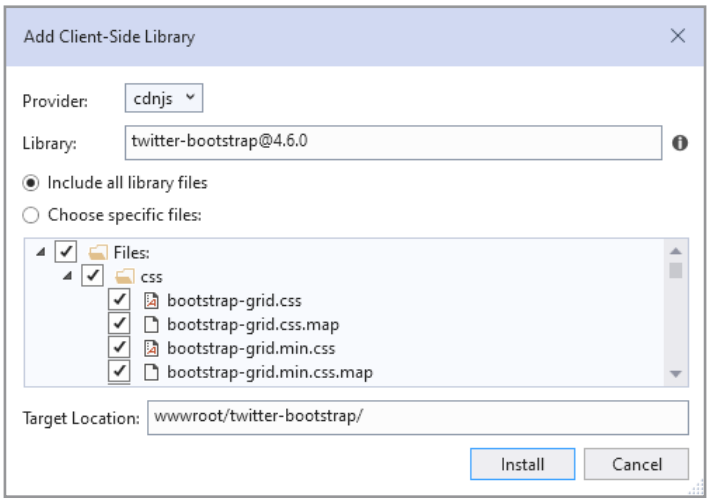
I you have to write a web application and think about styling there is considerable work ahead, but there is help to be found. Many web applications has an alike look and feel, and a number of libraries have been developed with stylesheets and java script that are ready-made and ready for use and in an easy way offer a nice design for the finished application. Such a library is called a client-side library or package. Using such a library has both advantages and disadvantages. The advantages are of course that you get a well-tried and accepted design and used correctly with a nice look and feel as a result. The disadvantage is that the application's user interface is thus pushed into a design that others have developed, but also that in order to get the full benefit you must learn to use the library in question, and it is not necessarily simple.

There are many such libraries and some very comprehensive. As an example, I want to show one called *twitter bootstrap* which is relatively popular, and to show how I will use the project *ChangeAddress1* from the previous book. I have created a copy of the project, and if I run the project the result is:



The above page is the start page and the applications also has three other pages where there are links to two of them from the start page. For the moment the visual appearance of the application is determined by a style sheet which is fine enough, but there is a long way to go in how modern web applications present themselves. As the first I have removed the stylesheet in the folder *wwwroot\css* and if I run the program again all the styles are gone, but the application can still run and works in principle as before.

To add the client-side library I right click on the folder *wwwroot* and select *Client-side Library*:



In this window I enter the name of the library (the windows a list when you enter) and after the library is selected I clicks on *Install* and it is added to the project.

Next I have added a folder *Shared* to the *Views* folder and to this folder a *Razor layout* file:

```
<!DOCTYPE html>

<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>@ViewBag.Title</title>
</head>
<body>
  <div>
    @RenderBody()
  </div>
</body>
</html>
```

and to the *Views* folder I have added a *Razor view start* file. Then I have modified the view *Index.cshtml:*

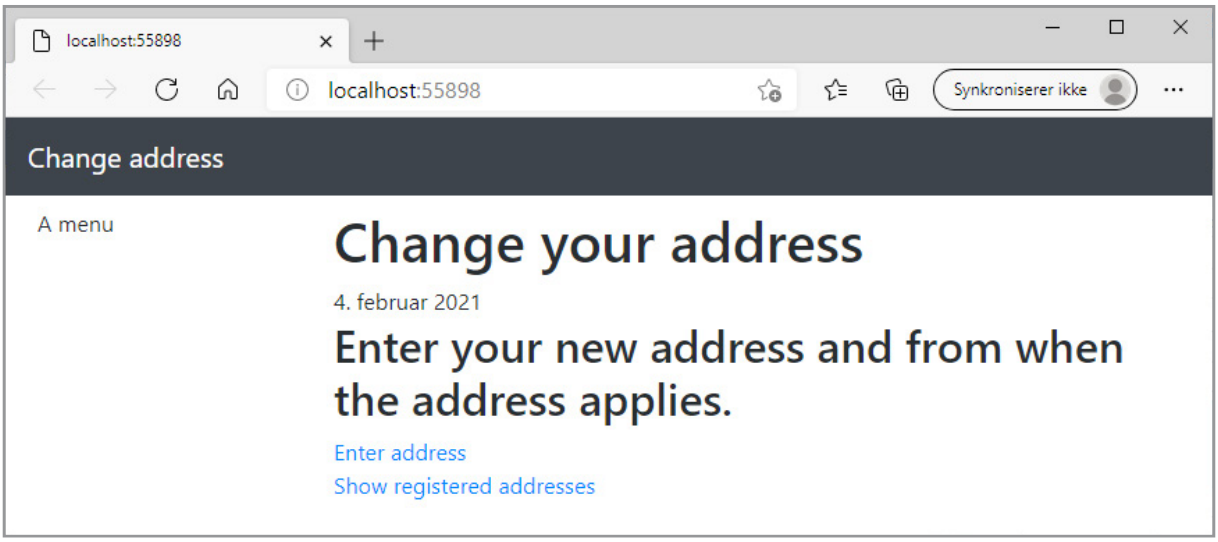
```
@{
}
<div>
  <h1>Change your address</h1>
  <div class="date">@Model.ToLongDateString()</div>
  <h2>Enter your new address and from when the address applies.</h2>
  <a asp-action="AddrForm">Enter address</a>
  <div>
    <p /><a asp-action="AddrList">Show registered addresses</a>
  </div>
</div>
```

That is the view is change to some layout to insert in *_Layout.cshtml*. If I now run the program again it all works, but the start page *Index.cshtml* now uses the layout file.

Next I update *_Layout.cshtml*:

```
<!DOCTYPE html>

<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>@ViewBag.Title</title>
  <link href="~/twitter-bootstrap/css/bootstrap.css" rel="stylesheet" />
</head>
<body>
  <div class="bg-dark text-white p-2">
    <span class="navbar-brand ml-2">Change address</span>
  </div>
  <div class="row m-1 p-1">
    <div id="categories" class="col-3">
      A menu
    </div>
    <div class="col-9">
      @RenderBody()
    </div>
  </div>
</body>
</html>
```

First you must note I have added a reference to a stylesheet in the library. This means that all views using the layout will use this stylesheet. The *body* element starts with a *div* element with a *span* element and here you should note that the *div* element uses three styles and the same does the *span* element, all styles defined in the *bootstrap* stylesheet. The next *div* element has two nested *div* elements which also is styled, and you experience here the challenges with the class library as you have to learn what all these styles mean. If you run the program the result is



and you can see that something has happened. I you look at the above design again and compare with the result you get an idea of what the styles means. You must note that the styles both define the page layout as a top layout and two columns as well as how the different elements are shown. It is a typical design for many web pages where the top panel is used for a header, a navigation bar or something else, while the left column is used for a menu. The right column is then used for content.

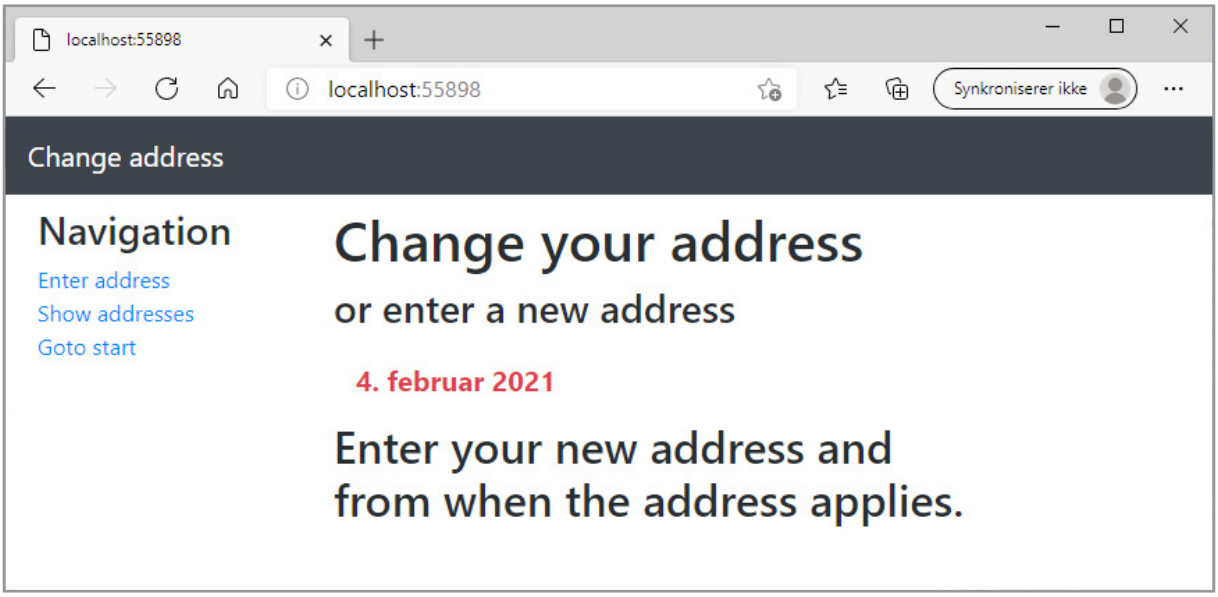
In this case I have modified the layout file:

```
<!DOCTYPE html>

<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>@ViewBag.Title</title>
  <link href="~/twitter-bootstrap/css/bootstrap.css" rel="stylesheet" />
</head>
<body>
  <div class="bg-dark text-white p-2">
    <span class="navbar-brand ml-2">Change address</span>
  </div>
  <div class="row m-1 p-1">
    <div id="categories" class="col-3">
      <h3>Navigation</h3>
      <div><a asp-action="AddrForm">Enter address</a></div>
      <div><a asp-action="AddrList">Show addresses</a></div>
      <div><a asp-action="Index">Goto start</a></div>
    </div>
    <div class="col-9">
      @RenderBody()
    </div>
  </div>
</body>
</html>
```

and the view *Index.cshtml*:

```
@{
}
<div>
  <h1>Change your address</h1>
  <h3>or enter a new address</h3>
  <div class="bg-white text-danger p-2">
    <span class="navbar-brand ml-2 font-weight-bold">
      @Model.ToLongDateString()</span>
  </div>
  <h2>Enter your new address and<br />from when the address applies.</h2>
</div>
```

Note that the only thing that has happened is that I have moved around some elements, and if you open the application in the browser, the result is:



and the three links works as they usually do.

I also have to update the three other views, and the form is updated as:

```
@model ChangeAddress.Models.Address
@{
}
<h1>Enter address</h1>
<form asp-action="AddrForm" method="post">
  <div asp-validation-summary="All"></div>
  <div class="row p-2">
    <label asp-for="Firstname" class="col-2">First name:</label>
    <input asp-for="Firstname" class="col-3" />
    <span asp-validation-for="Firstname" class="text-danger"></span>
  </div>
  <div class="row p-2">
    <label asp-for="Lastname" class="col-2">Last name:</label>
    <input asp-for="Lastname" class="col-3" />
    <span asp-validation-for="Lastname" class="text-danger"></span>
  </div>
  <div class="row p-2">
    <label asp-for="Addrline" class="col-2">Address:</label>
    <input asp-for="Addrline" class="col-3" />
    <span asp-validation-for="Addrline" class="text-danger"></span>
  </div>
  <div class="row p-2">
    <label asp-for="Zipcode" class="col-2">Zip code:</label>
    <input asp-for="Zipcode" class="col-1" />
    <span asp-validation-for="Zipcode" class="text-danger"></span>
  </div>
  <div class="row p-2">
    <label asp-for="Email" class="col-2">Email:</label>
    <input asp-for="Email" class="col-3" />
    <span asp-validation-for="Email" class="text-danger"></span>
  </div>
  <div class="row p-2">
    <label asp-for="Date" class="col-2">Date:</label>
    <input asp-for="Date" class="col-1" />
    <span asp-validation-for="Date" class="text-danger"></span>
  </div>
  <div class="row p-2">
    <label class="col-2">New Member:</label>
    <select asp-for="IsNew" class="col-3">
      <option value="true">Yes, I am a new member</option>
      <option value="false">No, I am an existing member</option>
```

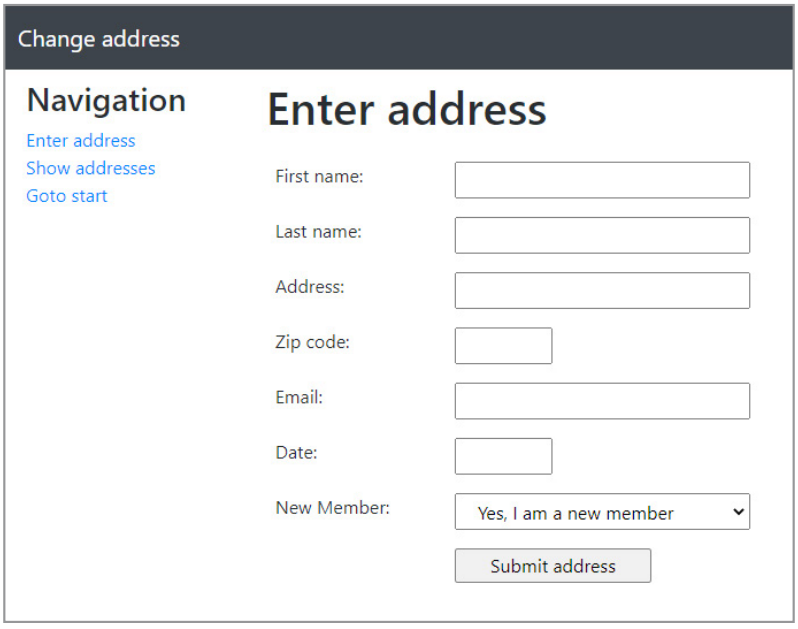
```
    </select>
  </div>
  <div class="row p-2">
    <label class="col-2"></label>
    <input type="submit" class="col-2" value="Submit address" />
  </div>
</form>
```

I should not review how the library works and what each style means here, as it is very comprehensive, but a good place to seek information is

```
https://www.tutorialrepublic.com/twitter-bootstrap-tutorial
```

which is an excellent tutorial. In the above I uses that the library basically divides the window into a grid with 12 columns. This grid can be nested and you can specify with a style how many columns an element should use. The library of course includes much more and including padding, margin and more. Of course, there are also styles for the individual HTML elements as for example tables, and the whole idea is that by using this library you can easily reach a nice and well-functioning user interface. It is certainly worth the work to learn twitter bootstrap, but it should also be mentioned that it is just one of several options, and each client-side package has its proponents.

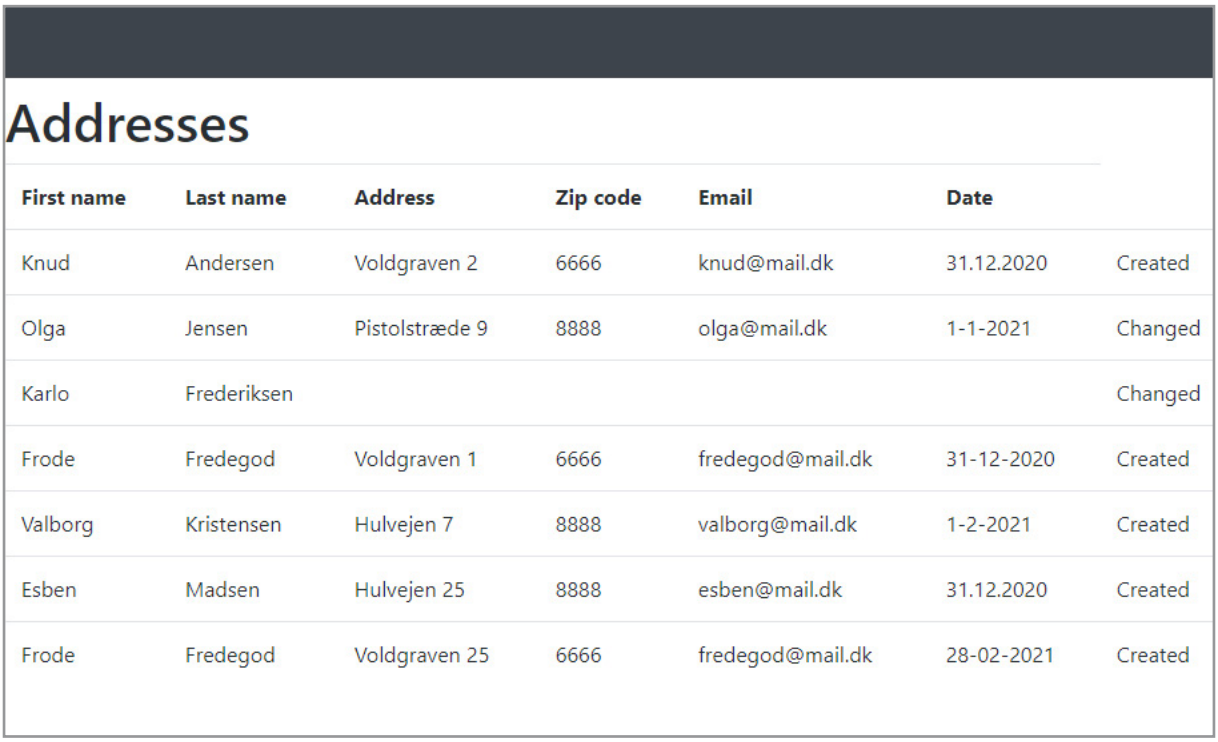
Below is show the result of the form after the above modifications:



Also the view *Receipt.cshtml* is modified but I will not show the result here. The last view which shows the table for all addresses is changed as:

```
@model ChangeAddress.Models.Addresses
@{
}
<h1>Addresses</h1>
<table class="table">
  <tr>
    <th>First name</th>
    <th>Last name</th>
    <th>Address</th>
    <th>Zip code</th>
    <th>Email</th>
    <th>Date</th>
  </tr>
  @foreach (ChangeAddress.Models.Address addr in Model)
  {
    <tr>
      <td>@addr.Firstname</td>
      <td>@addr.Lastname</td>
      <td>@addr.Addrline</td>
      <td>@addr.Zipcode</td>
      <td>@addr.Email</td>
      <td>@addr.Date</td>
      <td>
        @if (addr.IsNew == true)
        {
          @:Created
        }
        else
        {
          @:Changed
        }
      </td>
    </tr>
  }
</table>
```

There are not many changes and in fact only two. The view is change to a HTML stub to be inserted in *_Layout.cshtml* and the *table* element is expanded with a *class*. The result is:



Addresses

First name	Last name	Address	Zip code	Email	Date	
Knud	Andersen	Voldgraven 2	6666	knud@mail.dk	31.12.2020	Created
Olga	Jensen	Pistolstræde 9	8888	olga@mail.dk	1-1-2021	Changed
Karlo	Frederiksen					Changed
Frode	Fredegod	Voldgraven 1	6666	fredegod@mail.dk	31-12-2020	Created
Valborg	Kristensen	Hulvejen 7	8888	valborg@mail.dk	1-2-2021	Created
Esben	Madsen	Hulvejen 25	8888	esben@mail.dk	31.12.2020	Created
Frode	Fredegod	Voldgraven 25	6666	fredegod@mail.dk	28-02-2021	Created

and with a single grip has got a table which is much nicer than before.

The conclusion is that the use of a client-side package is something that can be recommended, even more than that because it is actually necessary to develop modern web applications in practice. However, there are two things to keep in mind. First, one must learn to use that library, its syntax, and the whole idea. Secondly, it is typically either or and if you have chosen a library, then you should use it throughout the application, whatever that is. Finally, it should be mentioned that you lose the opportunity to fine tune the application with your very own unique look and feel, if you are otherwise capable of it.

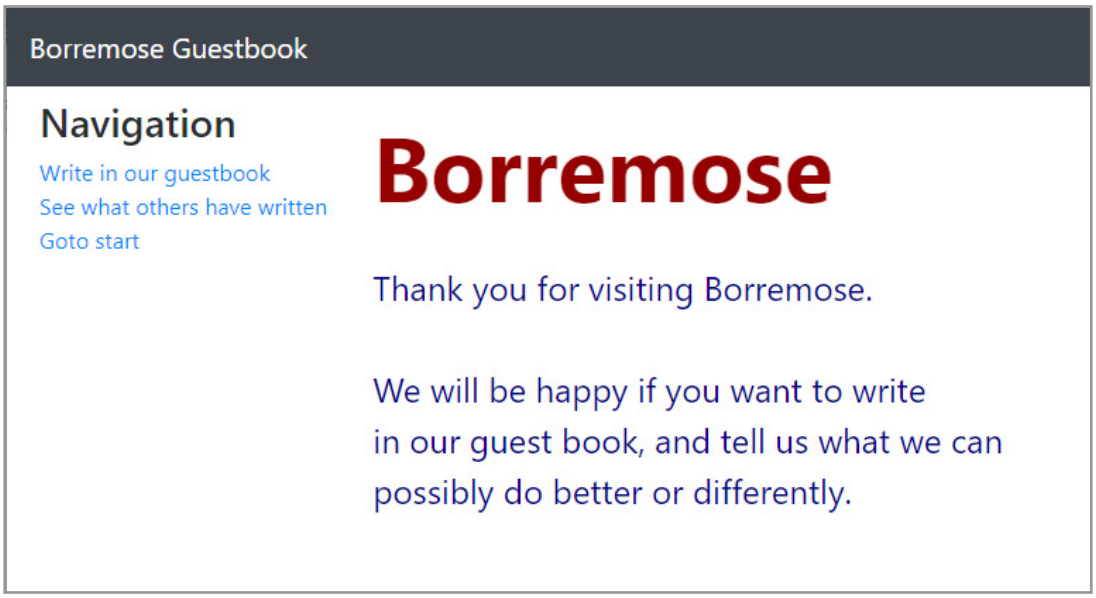
EXERCISE 1: GUESTBOOK

In the previous book (exercise 3) you write an application used to write in a guestbook. Create a copy of the solution for this exercise. You should in the same way as above update the project to use the client-side library *twitter-bootstrap*, but this time you should note remove en existing stylesheet. Start by

1. adding the library to your project
2. add a *Razor Layout* to a subfolder *Shared* in your *Views* folder
3. add a *Razor View Start* to your *Views* folder

Update the layout view *_Layout.cshtml* in the same way as in the previous example, but this time there should be a reference to both the old stylesheet and the stylesheet from the library.

Next you must modify the views such they uses the layout file. If there are styles in your stylesheet which conflicts with styles in the library you must remove them. As example the start page could be:

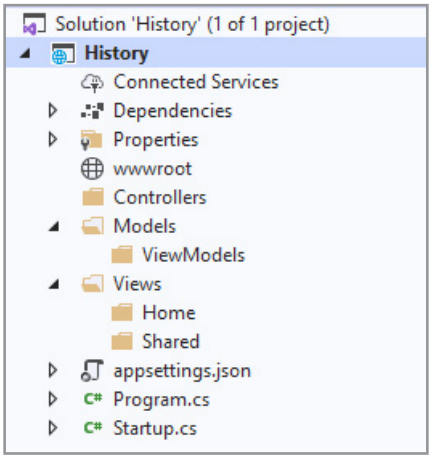


3 THE ENTITY FRAMEWORK

In this chapter I will write a program which show information about historical persons (see eventual C# 6), but where data is stored in a database using the Entity Framework. As implemented in ASP.NET Core it happens in a slightly different way than shown before for a Windows program. The following example also uses twitter-bootstrap library. To show how it works I will show the development step by step.

3.1 CREATE THE PROJECT

The start is a new ASP.NET Core project called *History*. To this project I adds the usual folders for at MVC application:

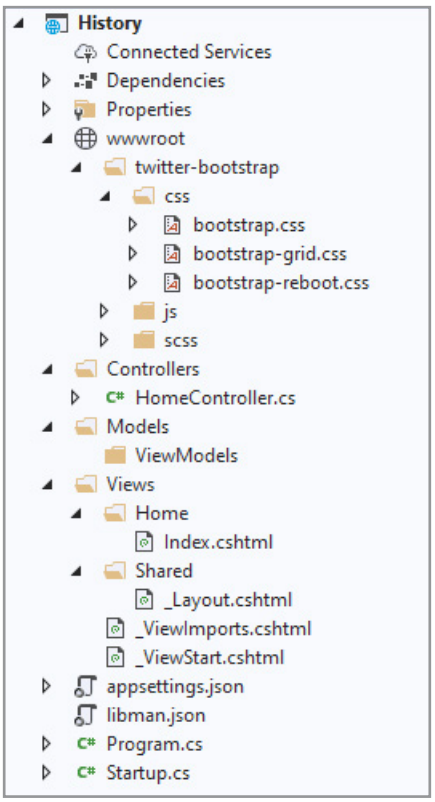


As the next step I adds the usual files:

1. A Razor View Start: *_ViewStart.cshtml*
2. A Razor View Imports: *_ViewImports.cshtml*
3. A Razor Layout: *_Layout.cshtml*
4. A Razor View: *Index.cshtml*
5. A controller: *HomeController*

I also install the *twitter-bootstrap* library, and the result is as shown below. You must to which folders the different files are added. The last three things is to update *_Layout.css* to use the *twitter-bootstrap* library and create the overall layout:

```
<!DOCTYPE html>

<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>@ViewBag.Title</title>
  <link href="~/twitter-bootstrap/css/bootstrap.css" rel="stylesheet" />
</head>
<body>
  <div class="bg-dark text-white p-2">
    <span class="navbar-brand ml-2">HISTORY PEOPLE</span>
  </div>
  <div class="row m-1 p-1">
    <div id="categories" class="col-3">
      Menu
    </div>
    <div class="col-9">
      @RenderBody()
    </div>
  </div>
</body>
</html>
```



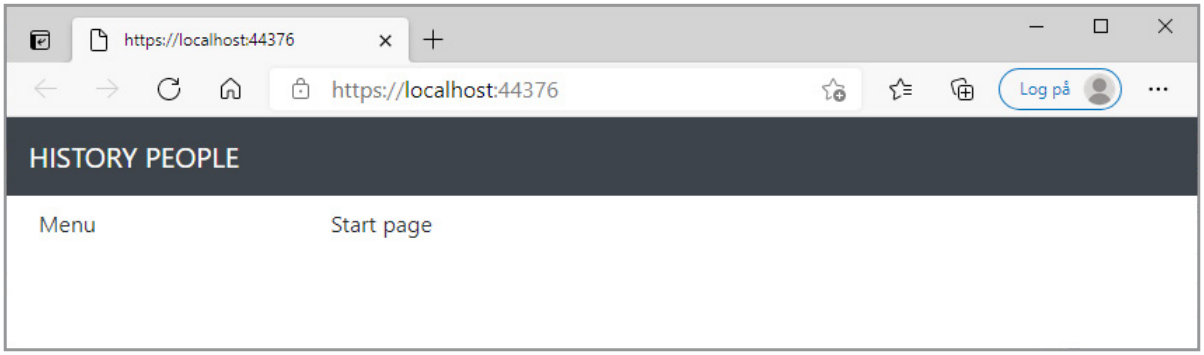
Add some text to *Index.cshtml* to make it visible:

```
@{
}
Start page
```

Modify *Startup.cs*:

```
public class Startup
{
  public void ConfigureServices(IServiceCollection services)
  {
    services.AddControllersWithViews();
  }

  public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
  {
    if (env.IsDevelopment())
    {
      app.UseDeveloperExceptionPage();
    }
    app.UseStaticFiles();
    app.UseRouting();

    app.UseEndpoints(endpoints =>
    {
      endpoints.MapDefaultControllerRoute();
    });
  }
}
```

The the application can run:



and I am ready to start the development.

3.2 THE MODEL

As the next step I will create the database and the show how to use the Entity Framework to create the database. It is not quite simple and can seem a bit overwhelming the first time, but if you have only done it a few times you will find that it is the same thing that has to happen every time. In this case there is only one table and will start creating a model class representing a person and then the a row in the table:

```
using System.ComponentModel.DataAnnotations.Schema;

namespace History.Models
{
  public class Person
  {
    public long PersonId { get; set; }
    public string Name { get; set; }
    public int? From { get; set; }
    public int? To { get; set; }
    public char? Gender { get; set; }
    public string Title { get; set; }
    public string Country { get; set; }

    [Column(TypeName = "ntext")]
    public string Text { get; set; }
  }
}
```

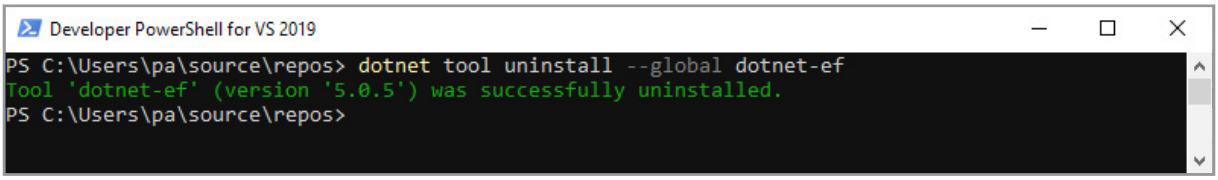
There is not much to note, but note the annotation for last property which tells the entity framework that the type of the database column must be *ntext* and not *varchar*.

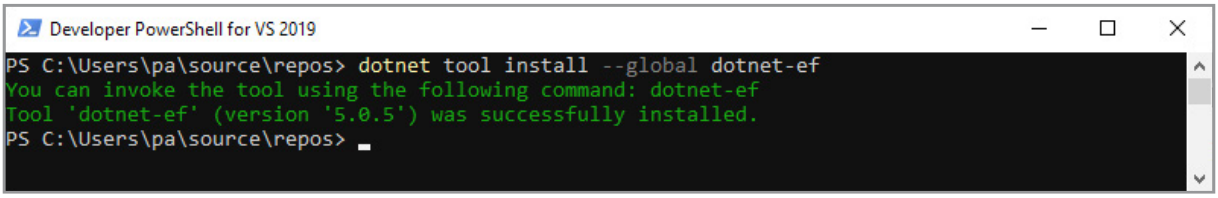
When the folder *Models* now has a class I can update the view imports file *_ViewImports.cshtml*:

```
@using History.Models
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

It has nothing to do with the entity framework, but is needed later.

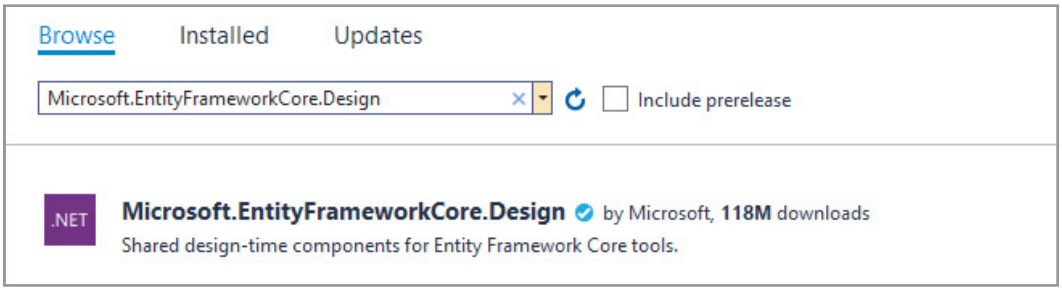
To use the entity framework for ASP.NET Core you must ensure that the framework is installed. You can du that from the *Developer PowerShell* command prompt using the following two commands, where the first remove an already installed framework while the other installs the framework:

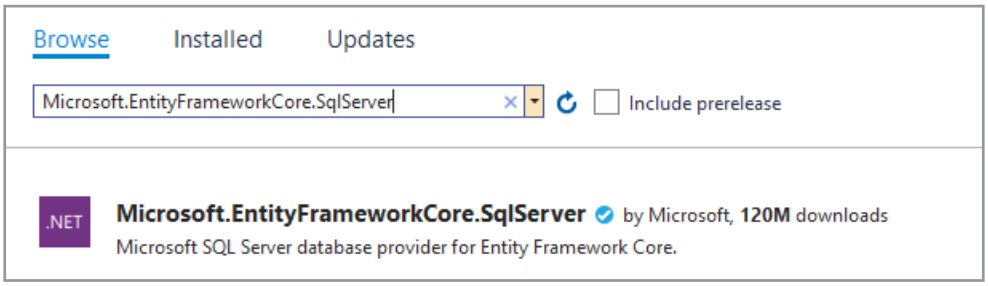




You should note that this is only necessary the first time you use the entity framework in an application.

Next you must using *NuGet* add two packages to your project:





In this project I will use *LocalDb* as database manager. It is a simple local database system like *SQLite*, and as default it is installed on your machine together with *SqlServer*. To reference the database I must create a connection string in *appsettings.json*:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "HistoryConnection": "Data Source=(localdb)\\MSSQLLocalDB;Database=
    History;MultipleActiveResultSets=true"
  }
}
```

As the next step I must create three classes representing the database. All classes are added to the *Models* folder, and the first represents the database and its context as an object:

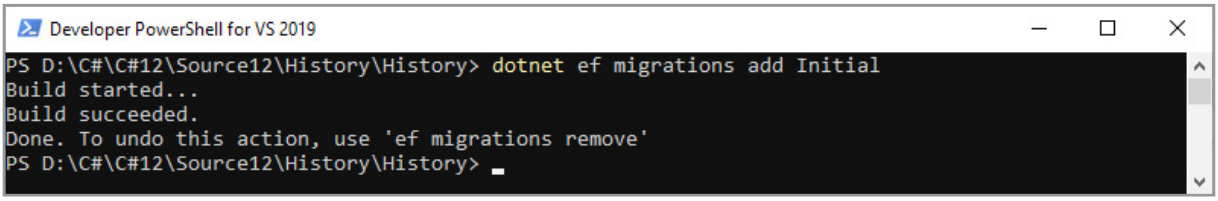
```
public class HistoryDbContext : DbContext
{
  public HistoryDbContext(DbContextOptions<HistoryDbContext> options) :
    base(options)
  {
  }

  public DbSet<Person> Persons { get; set; }
}
```

In general a table is represented as a collection of the type *DbSet* and in this case there is only one table. To simplify the representation (seen from the program) of the database, a repository is often defined in the form of a defining interface and an implementing class:

```
public interface IHistoryRepository
{
  IQueryable<Person> Persons { get; }
}

public class EFHistoryRepository : IHistoryRepository
{
  private HistoryDbContext context;

  public EFHistoryRepository(HistoryDbContext ctx)
  {
    context = ctx;
  }

  public IQueryable<Person> Persons => context.Persons;
}
```

The main reason is to make it easy to use LINQ for queries on the database.

What are shown above look like other database model classes as written in for example the previous book, but there I also has to create a lot of code to create the database and code to perform all the need SQL statements. It is what the entity framework do, but it happens using a tool performed as a command in *Developer PowerShell*. In *PowerShell* I must set current directory to the project directory and then perform the following command:



The command create a directory *Migrations* and here two classes:

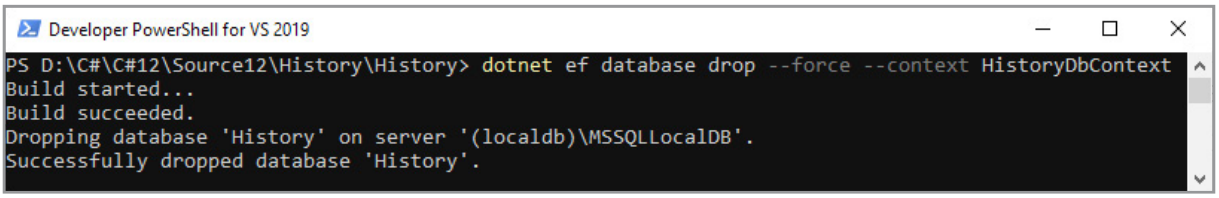
Now the database is ready for use (the application can create the database), but the *StartUp* class must be updated:

```
public class Startup
{
  public Startup(IConfiguration config)
  {
    Configuration = config;
  }

  private IConfiguration Configuration { get; set; }

  public void ConfigureServices(IServiceCollection services)
  {
    services.AddControllersWithViews();
    services.AddDbContext<HistoryDbContext>(opts => { opts.UseSqlServer(
      Configuration["ConnectionStrings:HistoryConnection"]); });
    services.AddScoped<IHistoryRepository, EFHistoryRepository>();
  }

  public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
  {
    if (env.IsDevelopment())
    {
      app.UseDeveloperExceptionPage();
    }
    app.UseStaticFiles();
    app.UseRouting();

    app.UseEndpoints(endpoints =>
    {
      endpoints.MapDefaultControllerRoute();
    });
    SeedData.EnsurePopulated(app);
  }
}
```

The class has a constructor which has an *IConfiguration* object as parameter. The object is used in *ConfigureServices()* to read the connection string in *appsettings.json* and the creating a connection to the database. You should note the last statement in the method *Configure()*. When the database is created it is empty, and if so the method *EnsurePopulated()*¸ inserts 10 rows in the table. You should note that the method only inserts rows if the table is empty.

```
public static class SeedData
{
  public static void EnsurePopulated(IApplicationBuilder app)
  {
    HistoryDbContext context = app.ApplicationServices.CreateScope().
      ServiceProvider.GetRequiredService<HistoryDbContext>();
    if (context.Database.GetPendingMigrations().Any())
    {
      context.Database.Migrate();
    }
    if (!context.Persons.Any())
    {
      context.Persons.AddRange(
        new Person { Name = "Ragnar Lodbrog", Title = "Viking", Gender = 'M',
          Country = "DK", Text = "Danish legendary king who lived in the 800s" },
        new Person { Name = "Gorm den Gamle", Title = "King", Gender = 'M',
          Country = "DK", Text = "Danish king who ruled from around 936 to 958.
          He is considered the first in the Danish royal line." },
        new Person { Name = "Harald Blåtand", Title = "King", Gender = 'M',
          Country = "DK", To = 985 },
        new Person { Name = "Svend Tveskæg", Title = "King", Gender = 'M',
          Country = "DK", From = 960, To = 1014 },
        new Person { Name = "Harald den 2.", Title = "King", Gender = 'M',
          Country = "DK", To = 1018 },
        new Person { Name = "Knud den Store", Title = "King", Gender = 'M',
          Country = "DK", From = 995, To = 1035 },
        new Person { Name = "Hardeknud", Title = "King", Gender = 'M',
          Country = "DK", From = 1018, To = 1042 },
        new Person { Name = "Magnus den Gode", Title = "King", Gender = 'M',
          Country = "DK", From = 1024, To = 1047 },
        new Person { Name = "Svend Estridsen", Title = "King", Gender = 'M',
          Country = "DK", From = 1019, To = 1076 },
        new Person { Name = "Egil Skallegrimson", From = 910, To = 990,
          Gender = 'M', Country = "IS" }
        );
      context.SaveChanges();
    }
  }
}
```

If I now run the application the database is created and 10 rows are added to the table. If I want to test the result I can opens *Microsoft SQL Management Studio*. There should now be a database *History* with a table *Persons* which has 10 rows. If at some point you want to delete the database, you can do so with the following command:



3.3 VIEW THE CONTENT

In this section I will update *Index.cshtml* to show the persons created in the database, and to du so I will introduce a *TagHelper* class.

The view must in the same way as shown in the previous book use page navigation where I will use the country as a filter. To use pagination I have added a class *PageInfo* to the *ViewModels* sub-folder:

```
public class PageInfo
{
  public int TotalItems { get; set; }
  public int ItemsPerPage { get; set; }
  public int CurrentPage { get; set; }
  public int TotalPages => (int)Math.Ceiling((decimal)TotalItems /
ItemsPerPage);
}
```

You should note that the last property is read only and calculated. Next I have written a model for *Index.cshtml*:

```
public class PersonsViewModel
{
  public IEnumerable<Person> Persons { get; set; }
  public PageInfo PageInfo { get; set; }
  public string CurrentCountry { get; set; }
}
```

The model has properties for persons to show, as well as *PageInfo* object and a value for the current country (used later). With this classes I can update the controller:

```
public class HomeController : Controller
{
  private IHistoryRepository repository;
  public int PageSize = 4;

  public HomeController(IHistoryRepository repo)
  {
    repository = repo;
  }

  public ViewResult Index(string country, int personPage = 1) => View(
    new PersonsViewModel { Persons = repository.Persons.Where(
    p => country == null || p.Country == country).OrderBy(p => p.Name).
    Skip((personPage - 1) * PageSize).Take(PageSize),
    PageInfo = new PageInfo { CurrentPage = personPage,
    ItemsPerPage = PageSize, TotalItems = country == null ?
    repository.Persons.Count() :
    repository.Persons.Where(e => e.Country == country).Count() },
    CurrentCountry = country });
}
```

The controller has a constructor with a parameter for the repository, and when the controller is instantiated such an object is transferred using dependency injection. Note that the page size is 4 which of course is too little, but for now I will stick to this value to test pagination as the database only has 10 rows. The method *Index()* has two parameters, where the first is for the filter while the other is the page number. Else the method must create *PersonsViewModel* object using a complex LINQ expression and also the creation of a *PageInfo* object uses a LINQ expression.

To create the view I must

1. create the menu
2. create list showing the persons
3. create the navigations links

and for all three parts I will show some technical details, something that one can live without, but which for more comprehensive web applications are important.

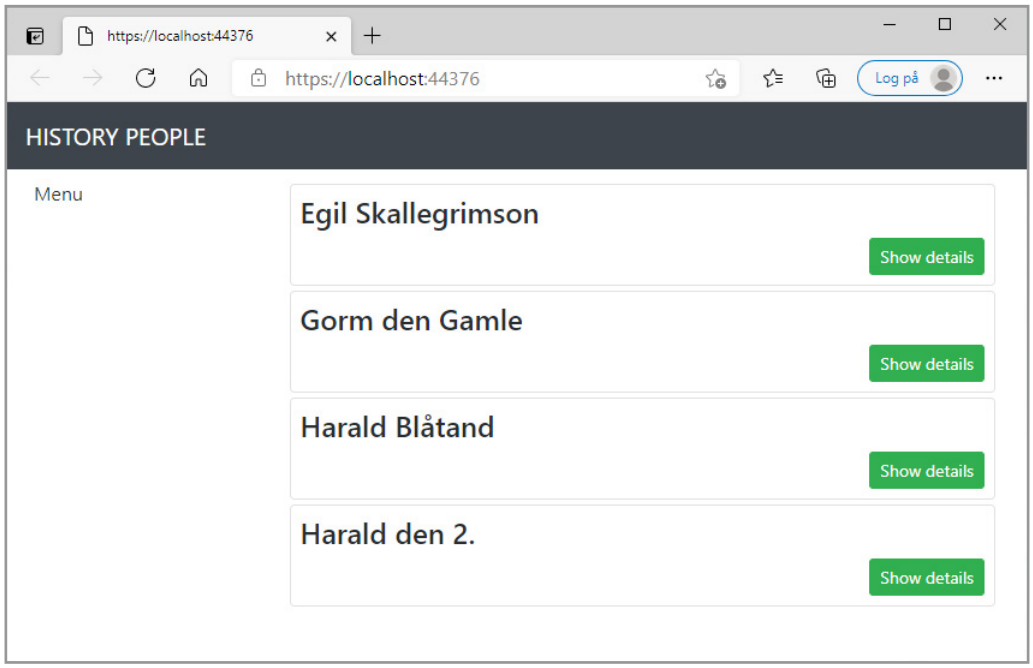
A partial view

To show the list of persons I will use a partial view which is just some content to add in another view. The meaning is that it should be possible to use the same content in more views. It will not be the case in this application and and the goal is then only to show the syntax and what it takes. To create the partial view I add a Razor view called *PersonSummary.cshtml* to *Views\Shared*:

```
@model PersonViewModel

<div class="card card-outline-primary m-1 p-1">
  <div class="bg-faded p-1">
    <h4>
      @Model.Person.Name
    </h4>
    <form id="@Model.Person.PersonId" asp-action="PersonView" method="post">
      <input type="hidden" asp-for="Person.PersonId" />
      <input type="hidden" asp-for="Page" />
      <input type="hidden" asp-for="Country" />
      <span class="card-text p-1">
        <button type="submit" class="btn btn-success btn-sm pull-right"
          style="float:right">Show details</button>
      </span>
    </form>
  </div>
</div>
```

Note that it is a view for a *PersonViewModel* object. The view shows the person's name and else contains a form for three hidden fields and a submit button. It all means that the partial view shows the name and a button. The three hidden fields are used for the person's id, the current page number and the current country used in the menu. Also note which styles from twitter-bootstrap are used to style the submit button. The form tag an id element as there are a form for each person shown on a page. When the user clicks on the button the application should show a view with all information about this person, and the action method *PersonView* is used to open this view.

With this partial view available the list of persons can in *Index.cshtml* be written as following. You should note the design that can all be applied to twitter bootstrap in the partial view.



A TagHelper

As the next I will show the use of a tag helper to create the navigation links. A tag helper is a C# class where you using C# can define how a HTML element should shown and contains. In this case the class is a helper for a *div* element and represents navigations links. The class is called *PageLinkTagHelper* and placed in the folder *Infrastructures*:

```
[HtmlTargetElement("div", Attributes = "page-model")]
public class PageLinkTagHelper : TagHelper
{
  private IUrlHelperFactory urlHelperFactory;

  public PageLinkTagHelper(IUrlHelperFactory helperFactory)
  {
    urlHelperFactory = helperFactory;
  }

  [ViewContext]
  [HtmlAttributeNotBound]
  public ViewContext ViewContext { get; set; }
  public PageInfo PageModel { get; set; }
```

```
  public string PageAction { get; set; }
  public bool PageClassesEnabled { get; set; } = false;
  public string PageClass { get; set; }
  public string PageClassNormal { get; set; }
  public string PageClassSelected { get; set; }

  public override void Process(TagHelperContext context, TagHelperOutput
  output)
  {
    IUrlHelper urlHelper = urlHelperFactory.GetUrlHelper(ViewContext);
    TagBuilder result = new TagBuilder("div");
    for (int i = 1; i <= PageModel.TotalPages; i++)
    {
      TagBuilder tag = new TagBuilder("a");
      tag.Attributes["href"] =
        urlHelper.Action(PageAction, new { personPage = i });
      if (PageClassesEnabled)
      {
        tag.AddCssClass(PageClass);
        tag.AddCssClass(i == PageModel.CurrentPage ?
          PageClassSelected : PageClassNormal);
      }
      tag.InnerHtml.Append(i.ToString());
      result.InnerHtml.AppendHtml(tag);
    }
    TagBuilder tag1 = new TagBuilder("a");
    tag1.Attributes["href"] = urlHelper.Action(PageAction, new {
    personPage = 0 });
    if (PageClassesEnabled)
    {
      tag1.AddCssClass(PageClass);
      tag1.AddCssClass(PageClassNormal);
    }
    tag1.InnerHtml.Append("Create");
    result.InnerHtml.AppendHtml(tag1);
    output.Content.AppendHtml(result.InnerHtml);
  }
}
```

Note first of all that it is a class that is used to dynamic build a HTML element. In this case it is a *DIV* element, and it builds a *div* element which contains some links. The class is a helper class which creates an object that inserts some HTML in a *div* element. The class is decorated which tells it is a helper for a *div* element and it can be used with a page model defined by a *page-model* attribute in a *div* element. In this case it will be a *PageInfo*

object. Note that the class inherits *TagHelper*. The class has a constructor with a parameter of the type *IUrlHelperFactory* which defines an object assigned by the system. There are some properties, but else the most important is the method *Process()*, which dynamic build a HTML tag. The properties is for the *PageModel* (a *PageInfo* object) and for the action method to be used when the user clicks on a link. The other attributes are for styling and instead to hardcode the use of twitter-bootstrap the properties are used to assign the styles to the tag helper when it is used. One property defines where to use the style and the other tree properties defines three styles.

In this case it is a *DIV* element which contains a number of link elements, one link for each page (of persons) and one link use to create a new person. The links are created in the method *Process()* and even though it takes up a lot of space, it's easy enough to follow what's happening. However, the syntax is not completely obvious everywhere, but it is because the code has to manipulate HTML elements and here you must of course know how to do it.

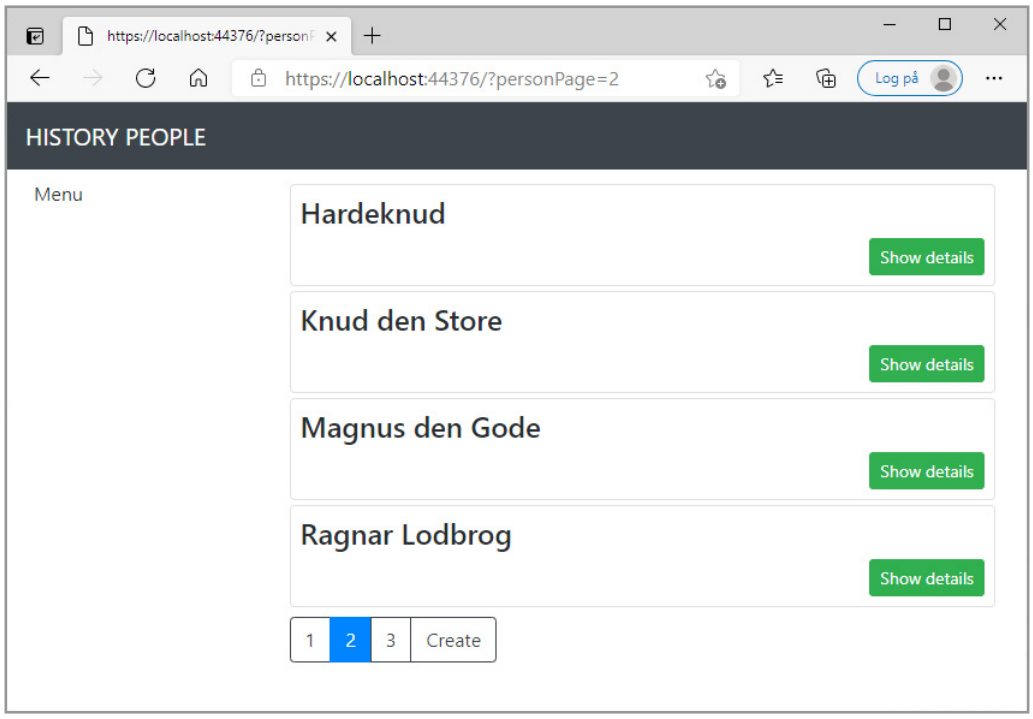
The tag helper is used in the view *Index.cshtml* to create the last line with the navigation buttons and the create link:

```
@model PersonsViewModel

@foreach (var p in Model.Persons)
{
  <partial name="PersonSummary" model="p" />
}
<div page-model="@Model.PageInfo" page-action="Index"
  page-classes-enabled="true" page-class="btn"
  page-class-normal="btn-outline-dark"page-class-selected="btn-primary"
  page-url-counytry="@Model.CurrentCountry" class="btn-group pull-right m-1">
</div>
```

To use the tag helper I must update the *_ViewImports.cshtml* file:

```
@using History.Models
@using History.Models.ViewModels
@using History.Infrastructure
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@addTagHelper *, History
```

If you run the application the result is is where I have clicked on page link 2:



The problem could be solved in other ways, but although the above seems complex, tag helpers are a widely used technique.

The menu

To finish the view I must add a menu or filter in the left part of the view where the user can filters persons on country. The action method already has a parameter for that, but for now it do not have a value. Also note that the class *PersonsViewModel* class has a property for the current country. To implement the menu I have created a new folder called *Components* and to this folder a class

```
namespace History.Components
{
  public class NavigationMenuViewComponent : ViewComponent
  {
    private IHistoryRepository repository;

    public NavigationMenuViewComponent(IHistoryRepository repo)
    {
      repository = repo;
    }

    public IViewComponentResult Invoke()
    {
      return View(repository.Persons.Select(
        x => x.Country).Distinct().OrderBy(x => x));
    }
  }
}
```

Next I must create a view for the menu in a folder

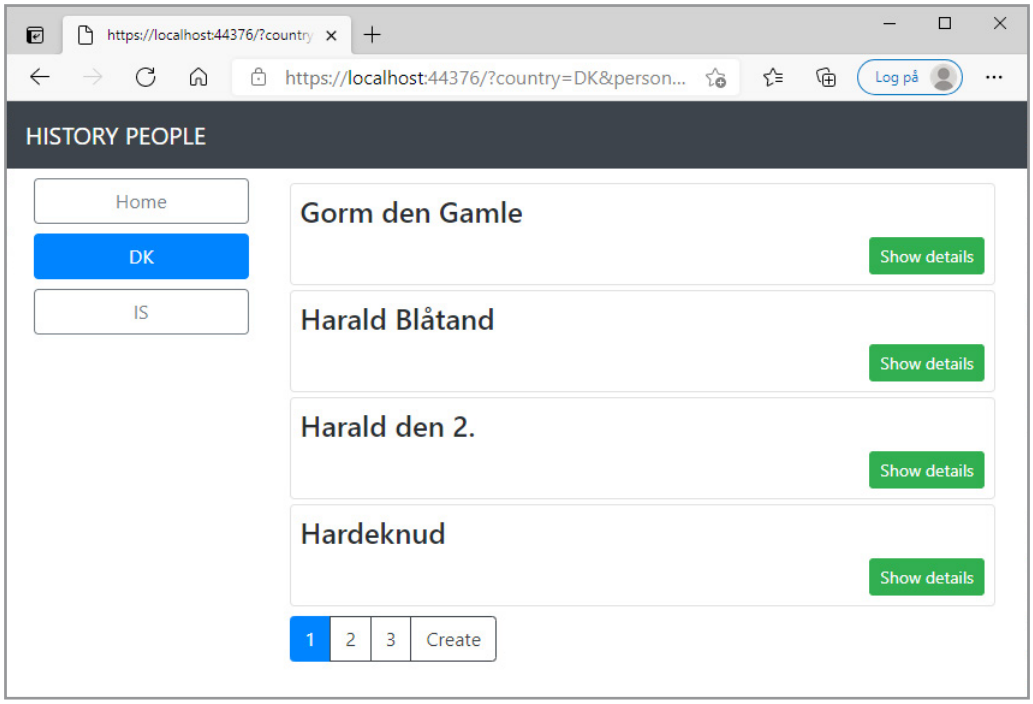
```
Views/Shared/Components/NavigationMenu
```

and the view is called *Default.cshtml*:

```
@model IEnumerable<string>

<a class="btn btn-block btn-outline-secondary" asp-action="Index"
asp-controller="Home" asp-route-country="">Home</a>
@foreach (string country in Model)
{
  <a class="btn btn-block @(country == ViewBag.SelectedCountry ?
    "btn-primary": "btn-outline-secondary")" asp-action="Index"
    asp-controller="Home" asp-route-country="@country"
    asp-route-personPage="1">@country</a>
}
```

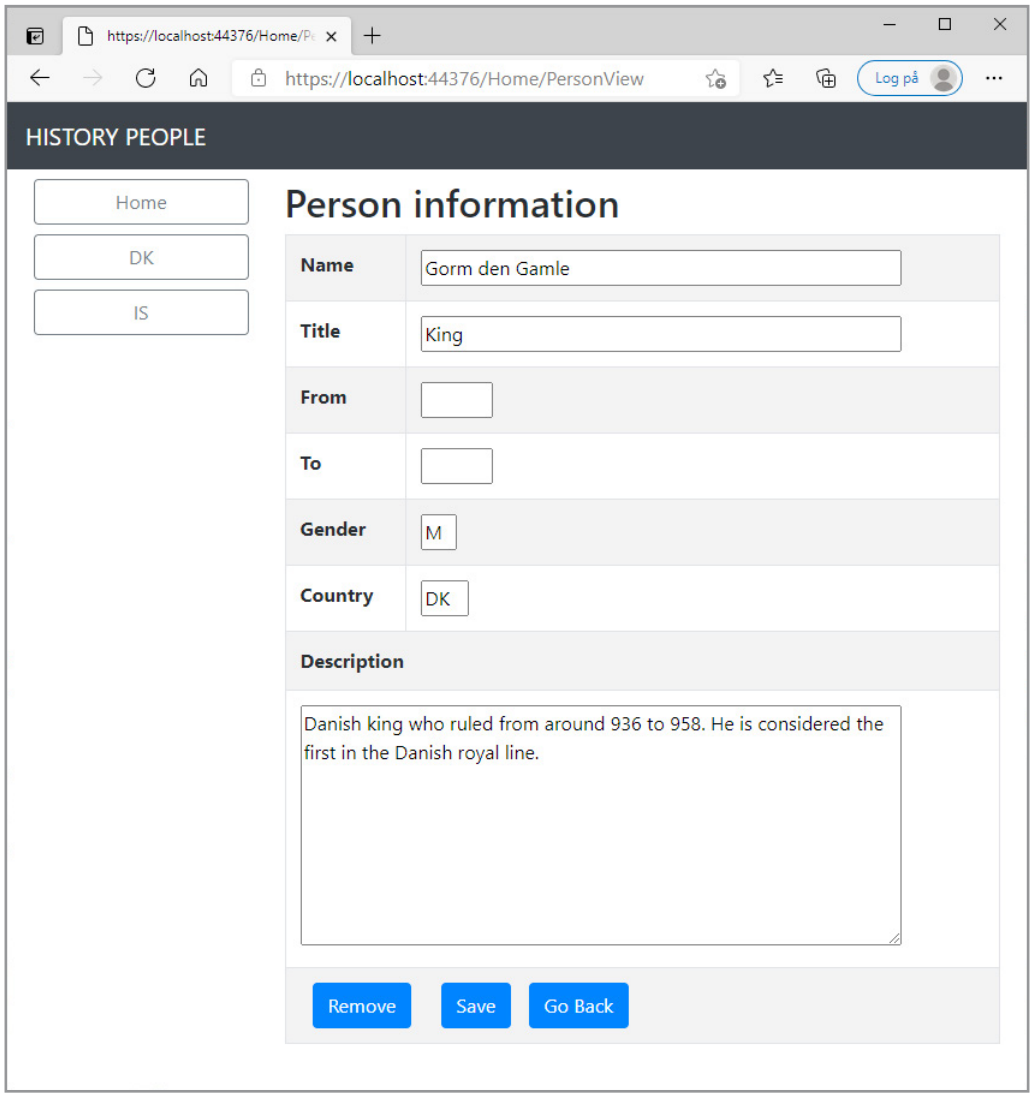
Note that the model is an enumerable of strings which is created in the above component. Else the view creates a link for all countries and a link for each country. To use the menu two things are needed

1. and update of *_Layout.cshtml*
2. a new property for the current country in the tag helper for the navigation links



3.4 SHOW / CREATE A PERSON

When the user clicks on one of the green buttons or the *Create* button the application should opens the following view where an existing person is selected:



I will not show the code for this view since there is nothing new, so jge will instead look at the controller:

```
public class HomeController : Controller
{
  private IHistoryRepository repository;
  private HistoryDbContext context;
  public int PageSize = 4;

  public HomeController(IHistoryRepository repository, HistoryDbContext
  context)
  {
    this.repository = repository;
    this.context = context;
  }

  [HttpGet]
  public IActionResult Index(string country, int personPage = 1)
  {
    if (personPage == 0) return RedirectToAction("PersonView", "Home");
    var persons = repository.Persons.Where(p => country == null ||
      p.Country == country).OrderBy(p => p.Name).Skip((personPage - 1) *
      PageSize).Take(PageSize);
    int items = country == null ? repository.Persons.Count() :
      repository.Persons.Where(e => e.Country == country).Count();
    PageInfo pageInfo = new PageInfo { CurrentPage = personPage,
      ItemsPerPage = PageSize, TotalItems = items };
    ViewBag.SelectedCountry = country;
    return View(new PersonsViewModel { Persons = persons,
      PageInfo = pageInfo, CurrentCountry = country });
  }

  [HttpGet]
  public ViewResult PersonView()
  {
    PersonViewModel model =
     new PersonViewModel { Person = new Person(), Country = "", Page = 1 };
    return View("PersonView", model);
  }

  [HttpPost]
  public ViewResult PersonView(PersonViewModel model)
```

```
  {
    model.Person = repository.Persons.Where(
      p => p.PersonId == model.Person.PersonId).First();
    return View("PersonView", model);
  }

  [HttpPost]
  public IActionResult Save(PersonViewModel model)
  {
    if (model.Person.Name != null && model.Person.Name.Length > 0)
    {
      if (model.Person.PersonId == 0) context.Persons.Add(model.Person);
      else context.Persons.Update(model.Person);
      if (context.SaveChanges() > 0) return RedirectToAction("Index",
        new { country = model.Country, personPage = model.Page });
      model.Error = "Unable to update database...";
    }
    else model.Error = "Person must have a name";
    return View("PersonView", model);
  }

  [HttpPost]
  public IActionResult Remove(PersonViewModel model)
  {
    model.Warning = "Warning: Are you sure you want to remove this person?";
    return View("PersonView", model);
  }

  [HttpPost]
  public IActionResult Accept(PersonViewModel model)
  {
    context.Persons.Remove(model.Person);
    if (context.SaveChanges() > 0) return RedirectToAction("Index",
      new { country = model.Country, personPage = model.Page });
    model.Error = "Unable to remove person...";
    return View("PersonView", model);
  }
}
```

The constructor now has two parameters as it should be possible to update the database. The parameters are initialized using dependency injection when a controller is created and for that the *Startup* class must be updated:

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddControllersWithViews();
  services.AddDbContext<HistoryDbContext>(opts => { opts.UseSqlServer(
    Configuration["ConnectionStrings:HistoryConnection"]); });
  services.AddScoped<IHistoryRepository, EFHistoryRepository>();
  services.AddScoped<HistoryDbContext, HistoryDbContext>();
}
```

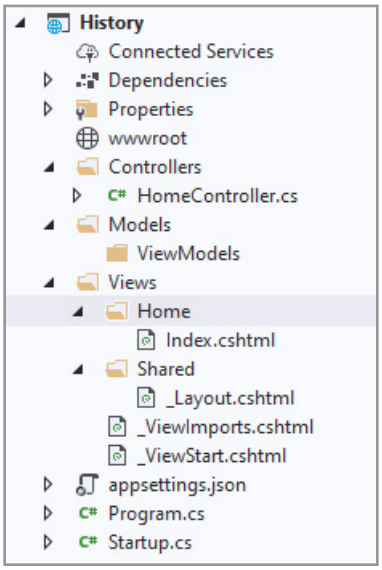
Otherwise, the controller does not contain much new and you should primarily note that there are several action methods primarily due to the three submit buttons in the form.

Then the application is finished and you can maintenance a database for historical people.

PROBLEM 1: HISTORY

In this problem you should write the same program as above, but such the database must have two table where the table *Person* has a foreign key to the other. Instead of starting with a copy, I would recommend that you start from scratch and follow the guidelines below.

1) Create new ASP.NET Core project called *History* and create the same folders and files as shown in the above example:



Add the *twitter-bootstrap* client library and modify the class *Startup*:

```
public class Startup
{
  public void ConfigureServices(IServiceCollection services)
  {
    services.AddControllersWithViews();
  }

  public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
  {
    if (env.IsDevelopment())
    {
      app.UseDeveloperExceptionPage();
    }
    app.UseStaticFiles();
    app.UseRouting();

    app.UseEndpoints(endpoints =>
    {
      endpoints.MapDefaultControllerRoute();
    });
  }
}
```

Update the default view *_Layout.cshtml* such it uses *twitter-bootstrap*:

```
<!DOCTYPE html>

<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>@ViewBag.Title</title>
  <link href="~/twitter-bootstrap/css/bootstrap.css" rel="stylesheet" />
</head>
<body>
  <div class="bg-dark text-white p-2">
    <span class="navbar-brand ml-2">HISTORY PEOPLE</span>
  </div>
  <div class="row m-1 p-1">
    <div id="categories" class="col-3">
      Menue
    </div>
    <div class="col-9">
      @RenderBody()
    </div>
  </div>
</body>
</html>
```

Now it should be possible to the application showing the same start page as above. What is done in this step is a typical start of an ASP.NET Core MVC application.

2) Next you should write the model. Create as above a *Person* class:

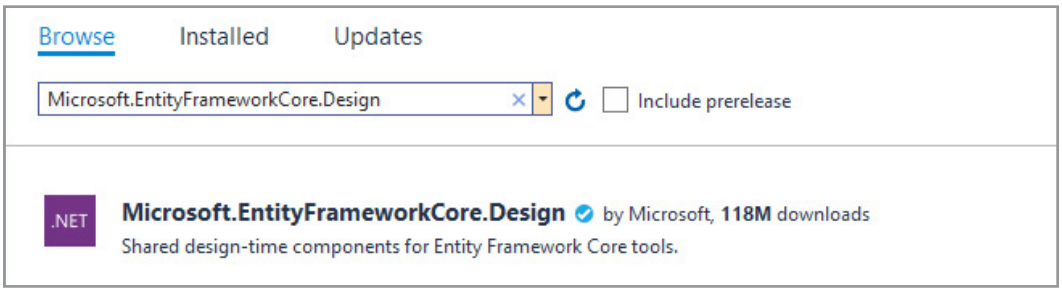
```
public class Person
{
  public long PersonId { get; set; }
  public string Name { get; set; }
  public int? From { get; set; }
  public int? To { get; set; }
  public char? Gender { get; set; }
  public string Title { get; set; }
  public Country Country { get; set; }

  [ForeignKey("CountryFK")]
  public int CountryFK { get; set; }

  [Column(TypeName = "ntext")]
  public string Text { get; set; }
}
```

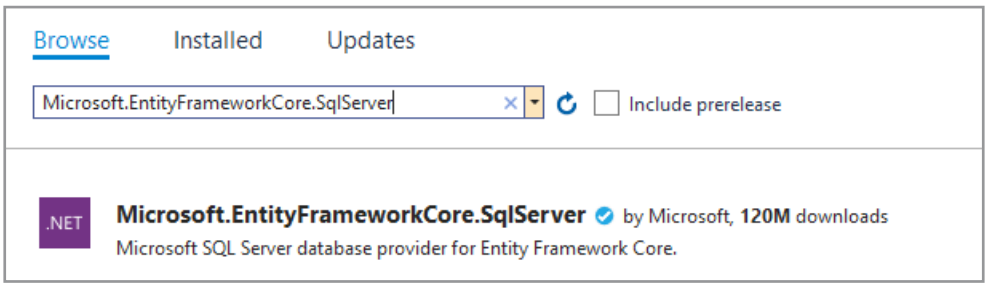
It's basically the same class as before, but the property *Country* has the type *Country* and there is defined another property with the name *CountryFK* that is decorated as a foreign key. You must also define an other model class:

```
public class Country
{
  public int CountryId { get; set; }
  public string Code { get; set; }
  public string Name { get; set; }

  [ForeignKey("CountryFK")]
  public ICollection<Person> Persons { get; set; }
}
```

which define objects that *Person* objects are related to.

You must then in the same way as in the above version of the program use NuGet to install packages for the entity framework for this project:





Next you should update the view imports file *_ViewImports.cshtml*:

```
@using History.Models
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

Update *appsettings.json* with a connection string for a LocalDb database when you this time calls the database *History1*.

In the *Models* folder you must write *HistoryDbContext* class, but this time there must be a D*bSet* also for countries:

```
public class HistoryDbContext : DbContext
{
  public HistoryDbContext(DbContextOptions<HistoryDbContext> options) :
    base(options)
  {
  }

  public DbSet<Country> Countries { get; set; }
  public DbSet<Person> Persons { get; set; }
}
```

Write the repository interface and class:

```
public interface IHistoryRepository
{
  IQueryable<Country> Countries { get; }
  IQueryable<Person> Persons { get; }
}

public class EFHistoryRepository : IHistoryRepository
{
  private HistoryDbContext context;

  public EFHistoryRepository(HistoryDbContext ctx)
  {
    context = ctx;
  }

  public IQueryable<Country> Countries => context.Countries;
  public IQueryable<Person> Persons => context.Persons;
}
```

Open a *Developer PowerShell*, set the current directory to the project directory and create a migration for the database using the same command as above.

You should then create a class *SeedData* in the *Models* folder to add some data to the database. The class should look like the class from the above project and you can start to copy the class from this project, but this time the class must also add data for countries:

```
public static class SeedData
{
  public static void EnsurePopulated(IApplicationBuilder app)
  {
    HistoryDbContext context =
      app.ApplicationServices.CreateScope().ServiceProvider.
      GetRequiredService<HistoryDbContext>();
    if (context.Database.GetPendingMigrations().Any())
    {
      context.Database.Migrate();
    }
    if (!context.Persons.Any())
    {
      Country country1 = new Country { Code = "DK", Name = "Denmark" };
      Country country2 = new Country { Code = "IS", Name = "Iceland" };
      context.Countries.Add(country1);
      context.Countries.Add(country2);
      context.Persons.AddRange(
       new Person { Name = "Ragnar Lodbrog", Title = "Viking", Gender = 'M',
         Country = country1,
         Text = "Danish legendary king who lived in the 800s" },
        ....
    }
  }
}
```

As the last step you must update the class *Startup*:

```
public class Startup
{
  public Startup(IConfiguration config)
  {
    Configuration = config;
  }

  private IConfiguration Configuration { get; set; }

  public void ConfigureServices(IServiceCollection services)
  {
    services.AddControllersWithViews();
    services.AddDbContext<HistoryDbContext>(opts => { opts.UseSqlServer(
      Configuration["ConnectionStrings:HistoryConnection"]); });
    services.AddScoped<IHistoryRepository, EFHistoryRepository>();
  }

  public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
  {
    if (env.IsDevelopment())
    {
      app.UseDeveloperExceptionPage();
    }
    app.UseStaticFiles();
    app.UseRouting();

    app.UseEndpoints(endpoints =>
    {
      endpoints.MapDefaultControllerRoute();
    });
    SeedData.EnsurePopulated(app);
  }
}
```

Now the application can be performed and it will create the database and and data to the two tables. You can open *Microsoft SQL Server Management Studio* to test whether the database has been created and has the correct tables with the expected content.

3) In this step you should write the code for *Index.cshtml*, when it will largely consist of copying from the previous project.

Add the classes *PageInfo*, *PersonViewModel* and *PersonsViewModel* to the *Models\ViewModels* folder.

Update the controller:

```
public class HomeController : Controller
{
  private IHistoryRepository repository;
  public int PageSize = 4;

  public HomeController(IHistoryRepository repository)
  {
    this.repository = repository;
  }

  [HttpGet]
  public IActionResult Index(string country, int personPage = 1)
  {
    var persons = repository.Persons.Where.OrderBy(p => p.Name).
      Skip((personPage - 1) * PageSize).Take(PageSize);
    int items = repository.Persons.Count();
    PageInfo pageInfo = new PageInfo
      { CurrentPage = personPage, ItemsPerPage = PageSize, TotalItems = items };
    return View(new PersonsViewModel { Persons = persons, PageInfo = pageInfo);
  }
}
```

Add the class *PersonSummary* to the *Views\Shared* folder.

Update *Index.cshtml*:

```
@model PersonsViewModel

@foreach (var p in Model.Persons)
{
  PersonViewModel m = new PersonViewModel { Person = p,
    Page = Model.PageInfo.CurrentPage, Country = Model.CurrentCountry };
  <partial name="PersonSummary" model="m" />
}
```

Note that the property *Model.CurrentCountry* still is used even if it has no effect. Try to run the program and it should show the first four persons.

Create a folder *Infrastructure* and add the class *PageLinkTagHelper* to this folder. Add the line for the page links to *Index.cshtml*, and if you run the program you should see the page links and they should all works - not the *Create* link.

As the last there is the menu where some modifications are needed.

Create a folder *Shared\Components\NavigationMenu* and the view *Default.cshtml* to this folder. You do not need to change any thing.

Add a folder *Components* and add the class *NavigationMenuViewComponent* to this folder when the method *Invoke()* must be updated as shown:

```
public class NavigationMenuViewComponent : ViewComponent
{
  private IHistoryRepository repository;

  public NavigationMenuViewComponent(IHistoryRepository repo)
  {
    repository = repo;
  }

  public IViewComponentResult Invoke()
  {
    return View(repository.Persons.Where(p => p.Country != null).
      Select(x => x.Country.Name).Distinct().OrderBy(x => x));
  }
}
```

Update *_Layout.cshtml* to use the menu.

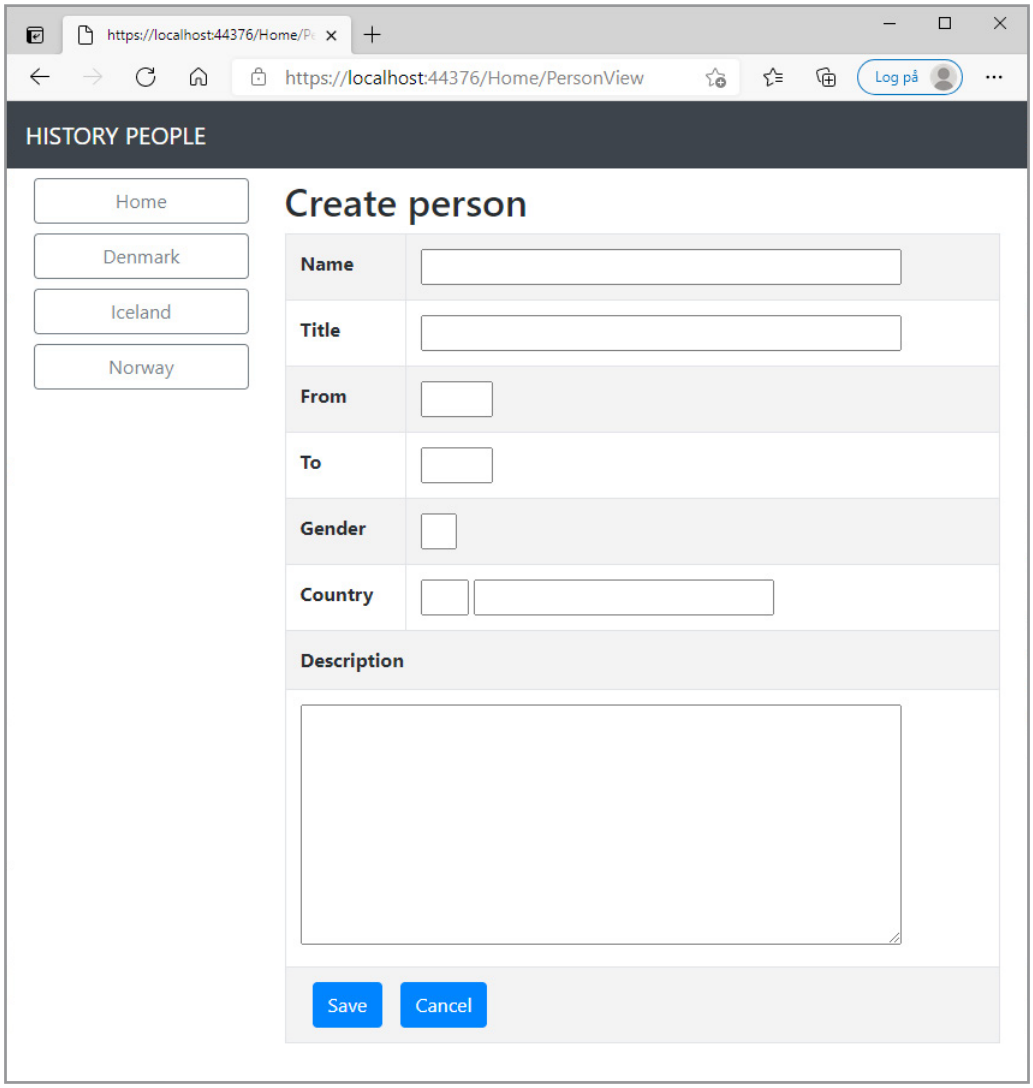
Update the action method *Index()*:

```
public IActionResult Index(string country, int personPage = 1)
{
  if (personPage == 0) return RedirectToAction("PersonView", "Home");
  var persons = repository.Persons.Where(p => country == null ||
    p.Country.Name == country).OrderBy(p => p.Name).Skip(
    (personPage - 1) * PageSize).Take(PageSize);
  int items = country == null ? repository.Persons.Count() :
    repository.Persons.Where(e => e.Country.Name == country).Count();
  PageInfo pageInfo = new PageInfo { CurrentPage = personPage,
    ItemsPerPage = PageSize, TotalItems = items };
  ViewBag.SelectedCountry = country;
  return View(new PersonsViewModel { Persons = persons, PageInfo = pageInfo,
    CurrentCountry = country });
}
```

Run the program and it should all works again.

4) To implement maintenance of persons you should add and expand the form *PersonView* with a new field for country name (see below). The user must enter a value for both country code and country name, and when the user clicks on *Save* the action method must test if there already is a country with this code. If so the method must test if the name is change and is this case update the country. Else the the method must create a new country in the country table.

For maintenance of persons the controller must be expanded with action methods as in the above example, but the action method for *Save* must be changed. Also remember to update *Startup* to transfer a *HistoryDbContext* object to the controller.



4 JAVASCRIPT

I will then give an introduction to JavaScript, which is the preferred language for code on the client side. The language has essentially the same syntax as C#, but otherwise the two languages do not have anything in common, and JavaScript is even a language that is very different from C#, but if you want to work professional as a web developer, you has to know JavaScript. Therefore, this chapter.

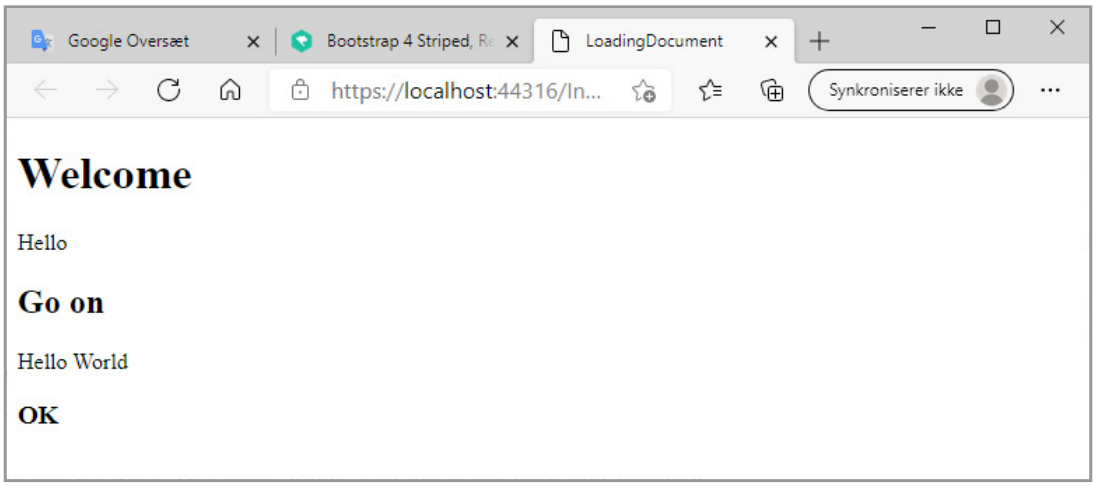
JavaScript is an interpreted programming language, and the source code are as C# just text, but unlike C#, the code is not translated, but is sent as text to the browser, which has a built-in interpreter that interprets the JavaScript code. It is a real interpreter who interprets the code statement for statement and execute each statement before the next is interpreted and performed. To show what JavaScript is, I want to start with a very simple example, called *LoadingDocument*. There is only one file *Index.html* whose content is:

```
<!DOCTYPE html>
<html>
  <head>
    <title>LoadingDocument</title>
    <meta charset="UTF-8">
    <script>
      hello = 'Hello';

      function show() {
        document.write(hello);
        hello += ' World';
      }
    </script>
    </head>
  <body>
    <h1>Welcome</h1>
    <script>show();</script>
    <h2>Go on</h2>
    <script>document.write(hello);</script>
    <h3>OK</h3>
  </body>
</html>
```

In the *head* section there is a *script* block and it is an element that can contain *JavaScript* code. In this case, a variable and a function are defined. The variable is called *hello*, and it is initialized with the text *"Hello"*. In principle, JavaScript uses variables as other programming

languages, including C#, but you should primarily note that the variable has no type. The script block also has a function that basically works as a method in C# and can be executed by calling it. In this case the function is called *show()* and it has two statements. The first write a text (the value of the variable *hello*) on the screen while the other modify the variable.

If you open *Index.html* in a browser the script is not immediately used as it is placed in the header. When the browser parses the document it starts by rendering elements in the *body* and the first is a *h1* element and the result is the header *Welcome* on the screen. The next element is a *script* element which contains some JavaScript, and the browser will then perform the script, and in this case it means perform the function *show()* in the script block in the header. It writes a text to the browser window showing the value of the variable *hello*. Then the browser continue and render the next *h2* element after which there is a new script element. The element contains JavaScript which prints the value of the variable *hello* on the screen, this time directly without using the function *show()*. As the last the browser render the *h3* element. The result is:



The above example shows a bit about what JavaScript is and how to add JavaScript to an HTML document, but not what JavaScript can be used for, what is the goal of the following. First, however, I would like to see a little more on the syntax of the language.

4.1 NATURE OF LANGUAGES

When you start with JavaScript and come from other languages such as C#, there are primarily three things that may seem very different:

- Variables' scope
- Types
- Objects and inheritance

and it's actually the three things that make programming in JavaScript much different than, for example, C# programming. As mentioned, JavaScript is an interpreted language and JavaScript code is sent to the browser along with the HTML code. The browser then has a built-in interpreter that interprets and executes the code's statements. The execution ends when all the code is executed correctly or when the interpreter comes to a statement that fails. It is not, therefore, as a C# program, where the entire program is translated to an internal binary language before the program can be run by the C# runtime system. JavaScript is a *type-weak* language, where variables should not be declared and having a particular type. It is not the same as that the language is type less, but it means that a variable can change type and that the current type is determined by the value of the variable and how the variable is used. There are a number of rules for the language's type compatibility, rules that are quite complex and some of the things that I will explain below. JavaScript is not object-oriented in classical sense, but conversely, objects play a major role in the language. However, compared to C#, for example, there is a very big difference in how to create and inherit objects. Perhaps the biggest difference between JavaScript and other languages are functions, since functions are actually objects that may have properties and methods.

The aim of this section is to illustrate some of these many concepts without having all the details, but to such an extent that you can use JavaScript in connection with web application development. When the browser parses a HTML document, it builds a tree consisting of the document's elements. The tree is called DOM (for *Document Object Model*), and the purpose of JavaScript is accurately to modify this tree.

The DOM tree

JavaScript can be placed anywhere in the document, but when the document is loaded, the browser parses the document and builds the DOM tree. If the browser meets any JavaScript, it will also perform it, and therefore it may only work if the element that the script refers to is already part of the DOM tree. Consider the following example:

```
<!DOCTYPE html>
<html>
<head>
  <title>Simple HTML5 document</title>
  <style type="text/css">
    h1.blueStyle {
      color: blue;
    }
    span.boldStyle {
      font-weight: bold;
    }
    p.redStyle {
      color: red;
    }
  </style>

  <script>
    function print(name) {
      document.write("<p>" + name + "</p>");
    }
  </script>
</head>
<body>
  <h1 id="idHeader" class="blueStyle">DOM</h1>
  <p id="idParagraph">This article deals with <span class="boldStyle">
    Regnar Lodbrog</span>
  </p>
  <p>A <span class="boldStyle">Danish</span> vikings</p>
  <script>
    document.write(document.title + "<br/>");
    document.write(document.getElementById("idParagraph").innerText);
    var pars = document.getElementsByTagName("p");
    for (var i = 0; i < pars.length; ++i) {
      pars[i].className = "redStyle";
    }
    document.write(pars.length);
    var spans = document.getElementsByTagName("span");
    for (var i = 0; i < spans.length; ++i) {
      spans[i].style.color = "green";
    }
    var classes = document.getElementsByClassName("boldStyle");
```

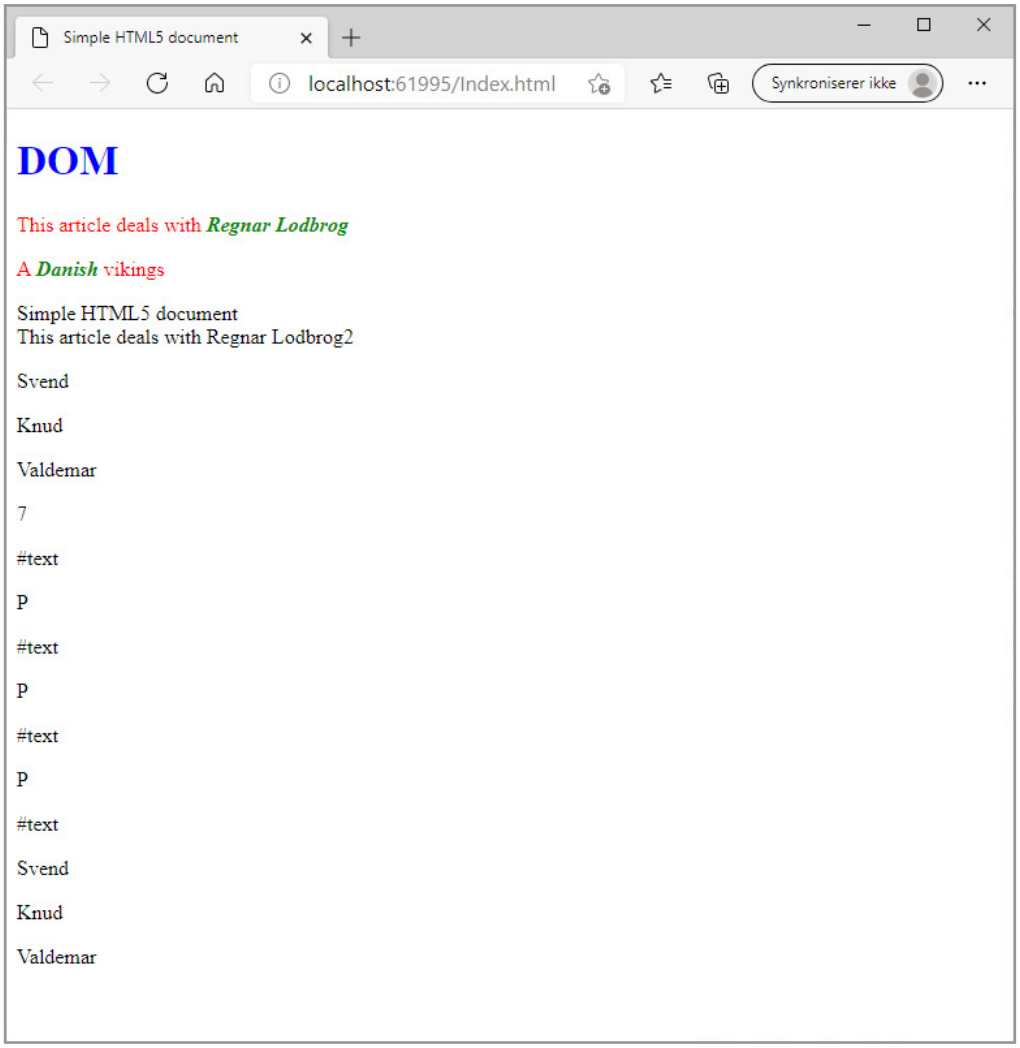
```
    for (var i = 0; i < classes.length; ++i) {
      classes[i].style.fontStyle = "italic";
    }
  </script>
  <div id="names">
    <p>Svend</p>
    <p>Knud</p>
    <p>Valdemar</p>
  </div>
  <script>
    var nodes = document.getElementById("names").childNodes;
    print(nodes.length);
    for (var i = 0; i < nodes.length; ++i) {
      print(nodes[i].nodeName);
    }
    var parent = document.getElementById("names");
    for (var node = parent.firstChild.nextSibling; ; node = node.nextSibling) {
      print(node.innerText);
      node = node.nextSibling;
      if (node == parent.lastChild) {
        break;
      }
    }
  </script>
</body>
</html>
```

The example (called *SimpleDocument*) should show what happens when the browser loads a document with JavaScript and how to refer to an item in the DOM hierarchy. Note first that a style sheet has been defined in the header. In addition, a script element has been defined that defines a single JavaScript function called *print()*. The function prints a name as a paragraph, which means that the function inserts a paragraph element in the place where the function is called. The browser will not perform the function at this location, it is only a definition of a function. The *body* part starts with a *h1* element and two paragraphs, and the browser starts displaying these items. It's not that much mystery in it, but after the browser has interpreted the above elements, a *script* part comes. The first JavaScript statement prints the documents title and the next prints the text in the first paragraph. You should note how to refer to an element with a specific ID and how to refer to the element's text with *innerText*. The statement prints the entire text, including the text in the *span* element, but the last part is not bold, as JavaScript does not interpret HTML. Then the browser continues to interpret and execute JavaScript statements. Here is *pars* an array of all *p* elements - at that time there are 2 - and the subsequent loop sets their *class* to *redStyle*, after which the text becomes red - also the text in the 2 *span* elements. Once that happens, a new print statement shows the length of the array *pars*. The browser continues

to perform JavaScript. *spans* is similarly an array with all *span* elements (there are 2) and the following loop sets the text to green. The last part of the script starts by determining an array *classes* that contain all elements with the class *boldStyle*. Next, the font for these elements is italicized.

The three names of Danish kings are simple HTML paragraphs, and they help to show that the browser first inserts these elements into the DOM tree after the first script block is completed. Then follows another script block. Here, *nodes* are an array with all child nodes to a *div* element. There are 7 child nodes that were not what you would expect, but the reason is that the plain text between the individual paragraphs also counts as a node. Next, a variable *parent* is created that refers to the *div* element. The subsequent loop iterates over all child nodes to *parent* starting from the first node. The loop skips the blank text elements and prints the text for paragraph elements. Please note that the loop stops with a break when you get to *lastChild*.

You should notice that JavaScript uses the dot notation to refer to objects' properties and methods - just as it is known from C#. If you opens the document in the browser, the result is:



Objects in JavaScript

JavaScript is an object-oriented language, but there is nothing similar to a class, and the only method of constructing new objects is by prototyping, which is a form of copying an existing object. In this way, a new object is created, which one can best think of as a copy of another object. A bit more precisely, any object in JavaScript has a property called *prototype*, and this property refers to all properties (properties and methods) inherited from the parent object. This means that if you try to refer to a property or method of an object, the interpreter will first check if that property is defined for the object in question. If this is not the case, the object to which *prototype* refers is checked, and neither is the property found in the hierarchy until it reaches an object where the property is defined or the interpreter can conclude that the object is *undefined*. Properties and methods can be overridden, and any overrides will affect the object where the override is defined and down in the inheritance hierarchy. It is best to think of this form of inheritance as a simple list, where *prototype* refers to the previous (parent) element. An important consequence of this inheritance is that a change of a property of an object is important for all objects down in the hierarchy.

The following example (called *ObjectsDocument*) should illustrate some of these concepts, and especially the syntax for how to create objects:

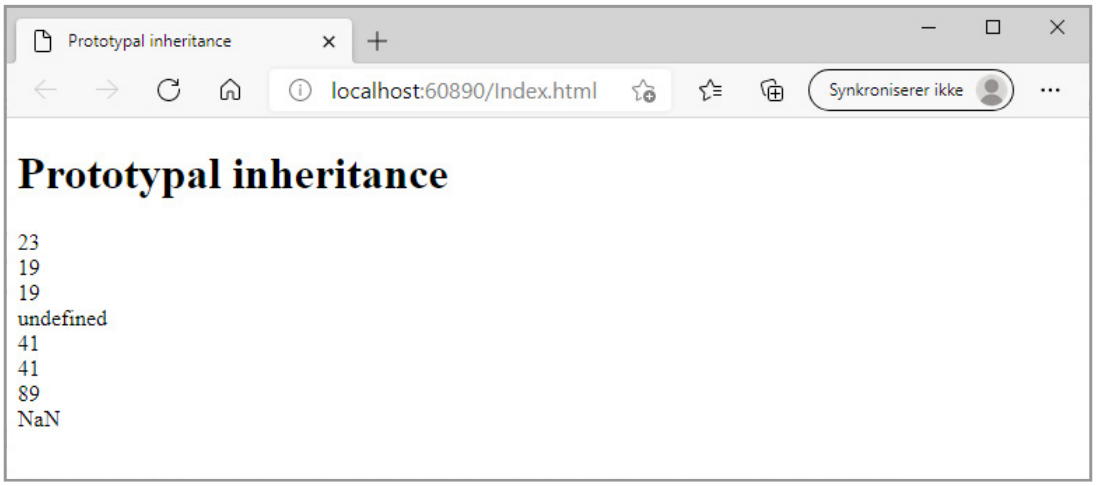
```
<!DOCTYPE html>
<html>
<head>
  <title>Prototypal inheritance</title>
  <script>
    var obj0 = {
      a: 23,
      b: 29
    }

    var obj1 = Object.create(obj0);
    var obj2 = Object.create(obj1);
  </script>
</head>
<body>
  <h1>Prototypal inheritance</h1>
  <script>
    document.write(obj2.a + "<br/>");
    obj0.a = 19;
    document.write(obj1.a + "<br/>");
```

```
    document.write(obj2.a + "<br/>");
    obj1.c = 41;
    document.write(obj0.c + "<br/>");
    document.write(obj1.c + "<br/>");
    document.write(obj2.c + "<br/>");
    obj0.toString = function () { return this.a + this.b + this.c; }
    document.write(obj1 + "<br/>");
    document.write(obj0 + "<br/>");
  </script>
</body>
</html>
```

In the document's header is a *script* block that creates three objects. The first is *obj0* and has two properties with the values 23 and 29 respectively. Next, two objects *obj1* and *obj2* are created, both of which are extensions of *obj0*, but at that time the three objects have the same properties.

The *body* part starts rendering an *h1* element, but otherwise the rest is a *script* block. Next, an *write()* statement is used to show the value of the property *a* in *obj0* and the result is a line that displays the value 23. Next, *a* is changed to 19 and then the value of *a* for *obj1* and *obj2* are shown, which in both cases is 19, corresponding to that *obj1* and *obj2* inherit the value from *obj0*. As a next step is assigned a property *c* to *obj1* with the value 41. Here you should note that you can immediately create a property for an object and assign it a value, and this property is not known from the parent object. Therefore, the three next *write()* statements will show *undefined*, 41 and 41, since *c* is not known from *obj0*, but from *obj1* and *obj2* (*obj2* inherit *obj1*).

obj0 inherits an empty top object called *Object* and thus particular has a *toString()* method. Therefore, both *obj1* and *obj2* have a *toString()* method and the next statement overrides *toString()* for *obj0*. The two last *write()* statements will show 89 (the sum of the three properties *a*, *b* and *c*, known from *obj1*) and *NaN*, since *toString()* can not perform the function from *obj0*, where *c* is not known. If you opens the document in a browser the result is:



Scope

In JavaScript, there are slightly different rules for the scope of variables than in C#, since variables have *functional scope*. That is to say, a variable defined in a function is known only in the function, but in return throughout the function wherever it is defined. The variable is also known in any nested functions. You should also be aware that JavaScript uses *hoisting*, which means that a variable can be used anywhere in the function where it is created - even before it is defined. The reason is that the interpreter starts creating variables as the variables all were defined at the start of the function. However, you must note that a variable at that time is not necessarily assigned a value.

A variable can also have *global scope*, which is the case if it is defined outside of a function, and such a variable can be used anywhere in the document. Normally, you write *var* in front of a variable when created, but it is actually not necessary (but can be recommended). A variable defined without *var* has automatic global scope wherever it is created.

In most programming languages, a variable is removed when the program leave the scope where the variable is defined, and it is also the case in JavaScript. That is, variables are created when the function in which they are defined is performed. It is at least the basic rule, but it does not necessarily apply. The reason is that in JavaScript a function is itself an object, and a function can thus return a function, and for example it can return an internal function. This inner function can refer to a variable in the enclosing function, and thus there may be a reference to a local variable after a function has been completed. If so, the variable will not be removed, a fact called *closures*.

Basically, in JavaScript, there are not so many challenges regarding variables' scope (maybe just except closures), and the following document, called *ScopeDocument*, should illustrate the most important scope rules:

```
<!DOCTYPE html>

<html>
<head>
  <title>Scope</title>
  <script>
    var c = 4;

    function test1() {
      var a = 1;
      var b = 3;
      function test2() {
        var a = 2;
        document.write(a + "<br/>");
        document.write(b + "<br/>");
        c = 5;
        document.write(c + "<br/>");
      }
      test2();
      document.write(a + "<br/>");
    }

    function power2() {
      var n = 1;
      function next() {
        n *= 2;
        return n;
      }
      return next;
    }
  </script>
</head>
<body>
  <h1>Test scope</h1>
  <script>
    test1();
    document.write(c + "<br/>");
//    document.Write(a + "<br/>");

    var f = power2();
    var g = power2();
```

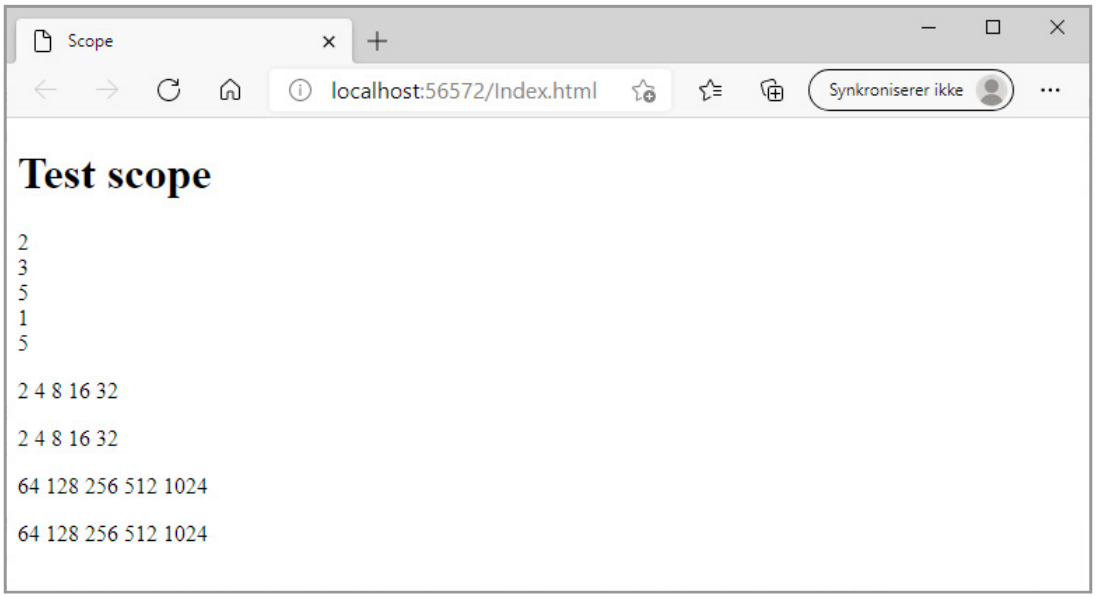
```
    document.write("<p>");
    for (var i = 0; i < 5; ++i) document.write(f() + " ");
    document.write("</p>");
    document.write("<p>");
    for (var i = 0; i < 5; ++i) document.write(g() + " ");
    document.write("</p>");
    document.write("<p>");
    for (var i = 0; i < 5; ++i) document.write(f() + " ");
    document.write("</p>");
    document.write("<p>");
    for (var i = 0; i < 5; ++i) document.write(g() + " ");
    document.write("</p>");
  </script>
</body>
</html>
```

If you start in the *script* block in the header of the document, a variable *c* is defined. It is defined outside of all functions and has global scope and is known everywhere. Next, a function *test1()* is defined with two local variables *a* and *b*. They are known only in the function *test1()* and are thus also known in principle in the nested function *test2()*. However, only *b* applies, since *test2()* also defines a local variable *a* that hides the variable *a* defined in *test1()*. The function *test2()* shows the value of its local variable *a* and the local variable *b* from *test1()*. In addition, *test2()* initializes the global variable *c* and shows its value. The function *test2()* is called from the external function *test1()*, which finally shows the value of its local variable *a*.

There is also defined a function *power2()*, which will illustrate the term *closure*. *power2()* has a local variable *n*, and it returns a function *next()* that uses (refers) to this variable. This means that there is a reference to the variable *n* after *prowe2()* terminates and hence *n* is not removed - that is called *closure*.

The *body* part starts with a *h1* element, after which *test1()* is called. Note that this means that the internal function *test2()* is also performed. Next, the value of the global variable *c* is displayed. If you try to show the value of the variable *a*, it is not defined in this location, and rendering of the document would be interrupted.

As the next step, two references *f* and *g* are defined for *power2()* and the result is that the function *power2()* is performed twice, and partly that *f* and *g* refer to a function with each their copy of the local variable *n*. It is illustrated by the following loops that perform the functions that *f* and *g* refers to, and thus each works on their copy of the local variable *n*. If the page is opened in the browser, the result is:



Data types

When you meet JavaScript, it may seem that the language is type less, but it is not the case. It is only a matter of the interpreter deciding the type of variables from the context, which means, among other things, that a variable can change its type. In the next section on syntax, I will describe the type concept in JavaScript in more detail, but basically there are four types:

1. *Boolean*, which are variables or expressions whose value is *false* or *true*
2. *Number*, which is numbers and everywhere is 64 bits floating numbers
3. *String*, to strings
4. *Object*, which are collections consisting of properties and methods, which cover anything other than the above three types

This type system involves several things, including, among other things, that there constantly must occurs type conversions. It's far from simple and something that I want to look at in the next section, but below is an example (*TypeDocument*) that can illustrate a bit about types and JavaScript:

```
<!DOCTYPE html>
<html>
  <head>
    <title>TypeDocument</title>
    <meta charset="UTF-8">
    <script>
      function add(a, b) {
        return a + b;
      }
    </script>
  </head>
  <body>
    <h1>Types</h1>
    <script>
      var arr = ["Knud", 3.14, 3 == "3", 1024, {}];
      for (var i = 0; i < arr.length; ++i)
        document.write(arr[i] + ", " + (typeof arr[i]) + "<br/>");
      document.write(arr + ", " + (typeof arr) + "<br/>");
      document.write(add(2, 3) + ", " + (typeof add) + "<br/>");
    </script>
  </body>
</html>
```

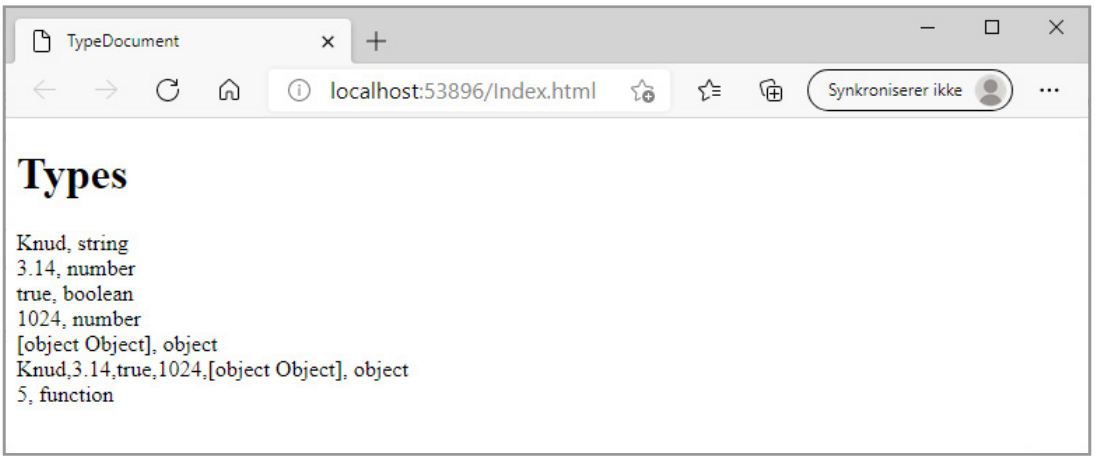
Note that in the header is defined a simple function with two parameters. The parameters have no type and all you can see is that the function performs the plus operator on the two parameters and returns the value. The script block in the body section creates an array of 5 elements of the following types:

1. the first is a *String*
2. the second is a *Number*
3. the third is a *Boolean* (that is *true* due to special conversion rules in JavaScript)
4. the fourth is a *Number*
5. the last is an *Object*

You should note that the array *arr* itself is an *Object*.

The next statement, which is a *for* loop, prints the array's elements as well as their types, and the next statement again does the same for the array. Note the *typeof* operator, which determines the type of a variable. Finally, there is the last statement that performs the function *add()* and prints its type. The type of a function is *Object*, but you should note that *typeof* displays it as a *function*. You should also note the *for* statement which has the same syntax as you know from C#.

If the page is shown in the browser the result is:



4.2 PATTERNS

In the previous section I have shown some basic concepts regarding JavaScript, and of course there are many more details, but the above should be sufficient to illustrate that JavaScript is a much different language than C#. In the time that JavaScript has existed, the language has developed a lot, and at the same time there have been several patterns for how to write JavaScript code. In particular, it appears that a function is an object, and when a function defines a scope, it allows functions to implement a form of data encapsulation. I would like to illustrate that with the following example called *PatternsDocument*.

```
<!DOCTYPE html>
<html>
<head>
  <title>Patterns</title>
  <script>
    var triple = (function () {
      var t = 0;

      function swap1() {
        t = public.a;
        public.a = public.b;
        public.b = t;
      }

      function swap2() {
        t = public.b;
        public.b = public.c;
        public.c = t;
      }

      public = {};

      public.a = 0;
      public.b = 0;
      public.c = 0;

      function build() {
       return "<p>" + public.a + ", " + public.b + ", " + public.c + "</p>";
      }
      public.print = function () {
        document.writeln(build());
      }

      public.sort = function () {
        if (public.a > public.b) swap1();
        if (public.b > public.c) swap2();
        if (public.a > public.b) swap1();
      }
```

```
      return public;
    })()
  </script>
</head>
<body>
  <h1>Patterns</h1>
  <h3>The least number</h3>
  <p>
    <script>
      document.write((function (a, b) { return a < b ? a : b })(5, 3));
    </script>
  </p>
  <h3>Is it a prime?</h3>
  <p>
    <script>
      document.write((function (n) {
        var m = Math.sqrt(n);
        return n + " is " + ((function() {
          if (n === 2 || n === 3 || n === 5 || n === 7) return true;
          if (n < 11 || n % 2 === 0) return false;
          for (var t = 3; t <= m; t += 2) if (n % t === 0) return false;
          return true;
        })() ? "a prime number" : "not a prime number");
      })(123));
    </script>
  </p>
  <p>
    <script>
      document.write((function (n) {
        var t1 = 0;
        var t2 = 1;
        while (n-- > 1) {
          var t3 = t1 + t2;
          t1 = t2;
          t2 = t3;
        }
        return t2;
      })(30));
```

```
      triple.print();
      triple.a = 7;
      triple.b = 3;
      triple.c = 5;
      triple.sort();
      triple.print();

      triple = (function (tpl) {
        tpl.sum = function () {
          return tpl.a + tpl.b + tpl.c;
        }
        return tpl;
      })(triple);

      document.writeln("<p>" + triple.sum() + "</p>");

      triple.sub = (function () {
        public = {};

        public.max = function () {
          var m = triple.a;
          if (triple.b > m) m = triple.b;
          if (triple.c > m) m = triple.c;
          return m;
        }

        return public;
      })();

      triple.a = 19;
      triple.b = 13;
      triple.c = 17;
      document.writeln("<p>" + triple.sub.max() + "</p>");
    </script>
  </p>
</body>
</html>
```

There are many details, and I will start with the script block in the body section. Generally, a function encapsulates a code that is executed when the function is executed, but puts parentheses without the entire function as

```
(function() { ... })();
```

the function, as here an anonymous function, will be performed immediately. As an example, the statement

```
document.write((function(a, b) { return a < b ? a : b })(5, 3));
```

insert the least of numbers the 3 and 5 into the document by performing an anonymous function. Of course, that does not make much sense, but if the parameters instead were variables, it could make sense. Thus, it is very common to insert the value of an expression by evaluating a function. Immediately it seems a bit strange to write a method in this way, but it is quite useful, although it may result in code that is difficult to read what the next expression shows:

```
<script>
  document.write((function(n) {
    var m = Math.sqrt(n);
    return n + " is " + ((function() {
      if (n === 2 || n === 3 || n === 5 || n === 7) return true;
      if (n < 11 || n % 2 === 0) return false;
      for (var t = 3; t <= m; t += 2) if (n % t === 0) return false;
      return true;
    })() ? "a prime number" : "not a prime number");
  })(123));
</script>
```

The script determines whether 123 is a prime number or not and inserts a corresponding text in the document. The example should show that it can be done by evaluating an anonymous function, and partly that JavaScript can use loops and conditions with the same syntax as in C#, which is further elaborated in the following section. The code is not so easy to read, but the text that is inserted is the value of an anonymous method with one parameter *n*. This method has a local variable *m*, which is initialized with the square root of *n*. Also note that it is written in the same way as in C# . The anonymous function returns a value, which is a text, but the value is determined by an inner anonymous function without parameters, a function that, using the usual prime algorithm, determines whether *n* is a prime. In this case, the use of an anonymous function is a bit exaggerated, but it is in some way common when writing JavaScript.

If you look at the header section then it uses an anonymous function to embed variables and functions in a namespace, and the variables and functions will become local to this namespace. An anonymous method used in that way is usually called a *module*, and *triple* is thus an example of a module. Modules can be perceived as a pattern to implement a form of data encapsulation in JavaScript, and among other things, names are local to the module, so you are independent of which names others may have used.

If you examine the module *triple*, *t* is a local variable in the anonymous function and thus in the module, and it will thus work as a private variable. Similarly, the two functions *swap1()* and *swap2()* are private functions for the module *triple*. The module further defines a variable called *public*. Here, you should note that this variable is an *Object* (which has no properties at present), and it has a global scope and is thus known outside of the module. After the object has been created, three properties are defined, which from the outside act as public properties by the object *triple* (explanation follows below). Next, another private function *build()* is defined, and then two functions that are functions of the object *public*, which will thus act as public functions of the object *triple*. When they do, it is in the same way as for the three properties of the object *public*, because the anonymous function is returning the object *public*.

Then there is the last script block in the *body* section, which should primarily show how the module *triple* is used, but first the block shows another use of an anonymous function that is performed without being called explicit. The function determines the 30th Fibonacci number. The remaining part of the block uses the object (module) *triple* and its public properties in terms of properties and function. First, the *triple* object's *print()* method is called that just prints three 0-values (the three properties). Next, the three properties are assigned a value and the method *sort()* is executed, after which the method *print()* is executed again.

You can use the syntax with an anonymous function to expand a module with a new feature. Below is how to expand *triple* with a new function:

```
triple = (function (tpl) {
  tpl.sum = function () {
    return tpl.a + tpl.b + tpl.c;
  }
  return tpl;
})(triple);
```

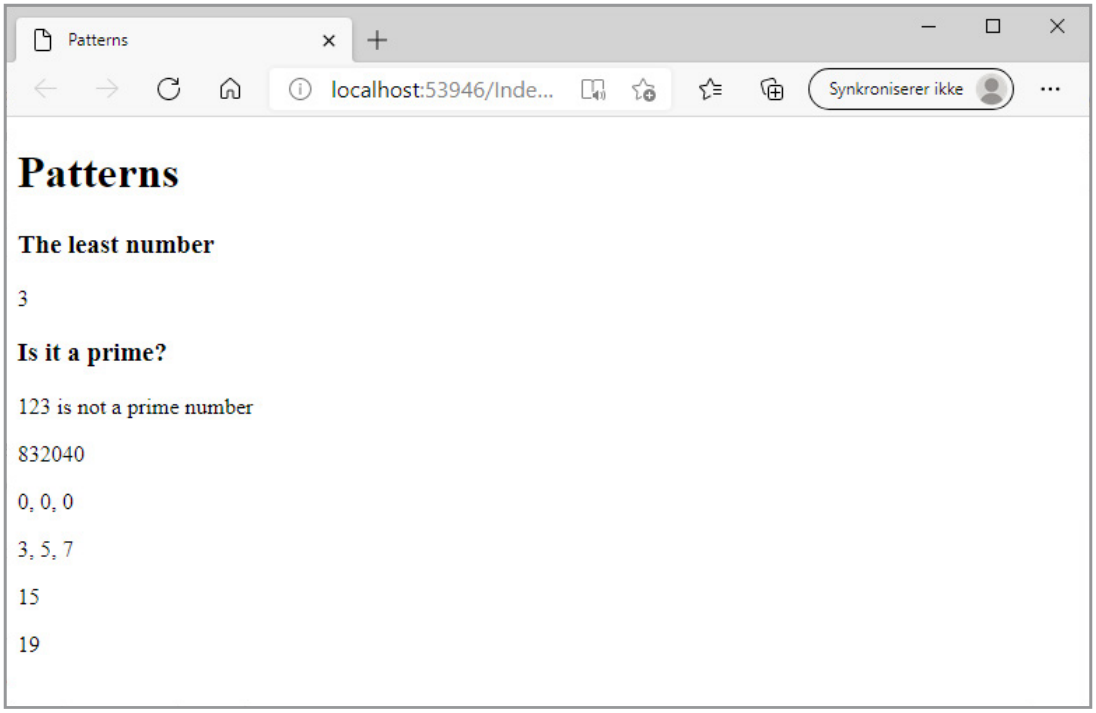
An anonymous function has a parameter *tpl*, and the function interprets it as an object and expands it with a new property, which is a function (returning the sum of three properties). Finally, the anonymous function returns the parameter *tpl* (which has now got a new

function), and is the anonymous function called with *triple* as the actual parameter and *triple* is assigned the return value, the result is that *triple* is expanded with a new function called *sum()*.

You can also by the same pattern expand *triple* with a sub-module (*triple* is an object). Here it is a sub module *sub*, which expands with a new function.

```
triple.sub = (function () {
  public = {};

  public.max = function () {
    var m = triple.a;
    if (triple.b > m) m = triple.b;
    if (triple.c > m) m = triple.c;
    return m;
  }
  return public;
})();
```

The result of all is that functions can be used to define a module term which is a namespace that can have private properties and provide public properties and methods available. If you perform the program, the result is as shown below:



4.3 BASIC SYNTAX

The following deals with the basic JavaScript syntax, and although the syntax is illustrated with examples, the review has to some extent character of a reference. Much of the following has already been mentioned in the previous section.

Basically, JavaScript uses the same syntax as the language C# and as so a language inspired by C. Therefore, I would like to write JavaScript in much the same way as I have written C# code and thus apply the same rules for blocks ({ and }) and the same rules for indentation. Regarding blocks, many choose (and also many development tools) to write

```
function f() {
  // code
}
```

instead of

```
function f()
{
  // code
}
```

I will follow that convention - primarily because my development tool does. There is also a tradition for (inspired by Java) that let function names start with a lowercase letter, and I will also adhere to that.

In JavaScript, a statement is terminated as in C# with a semicolon, but it is actually not necessary, as most interpreters are able to set the missing semicolon itself. Of course, if you do not terminate statements with a semicolon, it requires that you follow a number of rules for how the code is written, but newer interpreters are very good to automatically determining where statements end - they simply inserts a semicolon indirectly where it is necessary for the code to makes sense. However, I want to put semicolons everywhere, partly because I am used to it from other languages and partly because I think it increases readability, but even more importantly, missing semicolons may in some cases result in the interpreter's misunderstanding of the code.

Expressions and statements

An expression is simply a code that has a value. Examples include:

```
23
"Hello World"
33 + 29
(13 + 17) / (5 + 5)
Math.sqrt(100)
(str === "Svend") && (tal > 10)
```

A statement, on the other hand, is a sequence of expressions that result in some action, for example an assignment statement. A statement can particular be composed as a block of statements, which is an important term in cases of control statements. Note in particular that a block is not necessarily a statement. For example is an object

```
{
  t1: 23,
  t2: 29
}
```

not a statement but an expression.

Operators works on expressions and JavaScript has a good deal of the way the same operators as C#. The main operators are assignment, calculating operators and comparison operators. Operators also use the same precedence rules, known from other languages, including C#. Below is a table of all operators in JavaScript and their precedence. L / R means that operators evaluate from left to right, while R / L means that operators evaluate from right to left.

Operator	Precedence	Associativity	Remark
`()`	0		Parenthesis
`. []`	1	L/R	Member and index
`new`		R/L	
`()`	2	L/R	Function call
`++ --`	3		
`! ~`	4	R/L	
`+ -`		R/L	Sign
`typeof void delete`		R/L	
`* / %`	5	L/R	
`+ -`	6	L/R	Addtion and subtraktion
`<< >> >>>`	7	L/R	
`< <= > >= in instanceof`	8	L/R	
`== != === !==`	9	L/R	
`&`	10	L/R	
`^`	11	L/R	
`|`	12	L/R	
`&&`	13	L/R	
`||`	14	L/R	
`?`	15	R/L	
`yield`	16	R/L	
`= += -= *= /= %= <<= >>= >>>= &= ^= !=`	17	R/L	
`,`	18	L/R	

Variables

You declare variables with or without the use of the word *var*:

```
var t1 = 23;
t2 = 29;
```

In both cases, a variable has been created, and the difference is important for the variable's scope. It is allowed to declare several variables in the same statement

```
var obj = {}, counter = 0, str = "Knud", flag = true;
```

JavaScript has functional scope. This means that if you create a variable with *var*, its scope is limited to the function where the variable is created or for internal functions. It can be illustrated with the following function:

```
function f()
{
  var t = "Here"; // t's scope is the function f and other scopes in f

  function g() // g() creates a new acope in the function f()
  {
    document.write(t); // t is known here
  }
  g();
}

f();              // will alert "Here"
document.write (t);      // results in an error, when t is not known
```

The scope where a variable is defined is often called for its *local scope*. When referring to a variable, the runtime system will search for the variable in the current scope. If the variable is not found here, the system will search for the variable in the containing scope, and this will continue until the system finds the variable or comes to the top scope, commonly called the *global scope*. The process is sometimes called *scope chain lookup*. Variables that are not declared in a function will always have global scope, and the same applies to variables that are declared without the use of *var*.

Variables are always created at the start of a scope - even if the declaration itself is first written later in the code:

```
function f()
{
  document.write(t);
  var t = 2;
}
```

If you perform this function, it will write *undefined*. The variable *t* is thus created, but it is not initialized. However, if you remove the word *var*, you will get an error as you refer to a variable that has not been created. This corresponds to the interpreter's perception of the above function as:

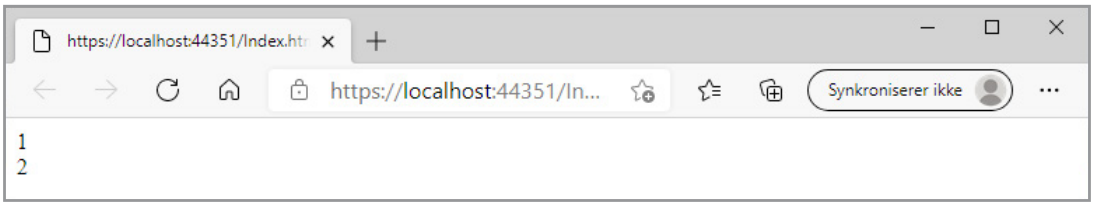
```
function f()
{
  var t;
  document.write(t);
  t = 2;
}
```

As it can sometimes lead to hard-to-see results, it is recommended that you consistently create and initialize variables at the beginning of functions.

When a function terminates, its scope will disappear and at the same time, the runtime system will remove all local variables that the function has created. If the function is called again, its local variables will be re-created. It is at least the normal behavior, but consider the following document:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title></title>
  <script>
    function f()
    {
      var t = 1;
      function g()
      {
        document.write(t++ + "<br/>");
      }
      return g;
    }
  </script>
</head>
<body>
  <script>
    var h = f();
    h();
    h();
  </script>
</body>
</html>
```

If it opens in the browser, the first *write()* will show 1 and the next 2. If you consider the function *f()*, it has a local variable *t* that is assigned the value 1. The internal function *g()* shows the value of this variable, after which it is counted up with 1. The function *f()* returns the internal function *g()*. Note that it is legal, since a function is specifically an object. In the body section, the function *f()* is performed and its return value is stored in the variable *h*, which now refers to a function and the function *f()* terminates. Next, *h()* is performed, which means that it is the function *g()* that is performed as with a *write()* shows the value 1, which is the value of the local variable *t* in *f()*. When *h()* is executed the second time, it will display 2. There it is still the value of *t* and you can see that *t* is not initialized again and thus not removed after *f()* is terminated. The reason is that there is an indirect reference to *t* via *g()*. When *f()* terminates, *t* is not removed since *h* refers to *g()* which refers to *t*. That a local variable in this way will not be removed after the function that creates the variable terminates is called closures.



Types

In JavaScript, you should not specify any data type when creating a variable. It is said that it is a type-weak language. However, it is not the same as JavaScript does not have types and there are basically 4 types:

- *Boolean*, which are variables or expressions that are either *false* or *true*
- *Number*, which are numbers, that are 64 bits floating points
- *String*, which are random sequences of characters
- *Object*, which are families of properties and methods

That the language is type-weak means that the runtime system based on variables and expression values automatically determines the type. This means, in particular, that the type of a variable can be changed over its lifetime and is always determined by the value of the variable. For example the following code

```
var arr = [“Knud”, 47, true, {}, f];
for (var i = 0; i < arr.length; i++) alert(typeof arr[i]);
alert(typeof arr);
```

where the function *f()* is as above, will alert the following type names (the operator *typeof*)

- *string*
- *number*
- *boolean*
- *object*
- *function*
- *object*

Here you should note that for *f*, the result is *function*, even if the type is formally *object*, and that the type of an array is an *object*.

The value of a variable can be a *primitive* where there are the following options

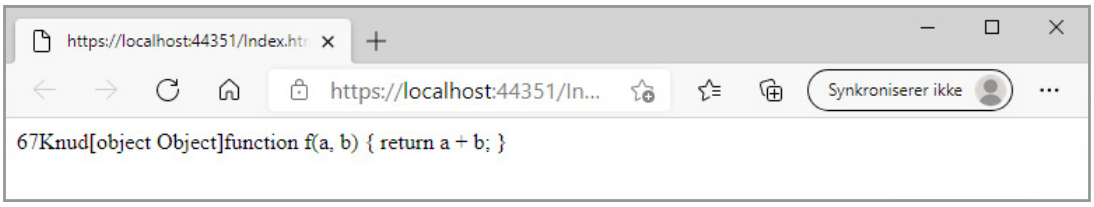
- *true* or *false* and then a *boolean*
- a *number* and then all possible numbers
- a *string*
- *null*, that means that the variable has no value
- *undefined*, that means that the variable is defined, but not initialized

Everything that is not a primitive is an *object*, and it therefore special means arrays and functions.

Consider the following code (*TypeDocument1*):

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <title></title>
  <script>
    function f(a, b)
    {
      return a + b;
    }
  </script>
</head>
<body>
  <script>
    var arr = [47, true, "Knud", {}, f];
    var v = 19;
    for (var i = 0; i < arr.length; i++) v += arr[i];
    document.write(v);
  </script>
</body>
</html>
```

If you perform this code, you get the result:



The variable *v* has the value 19, which has the type *number*. When you loop over the array, you first determines the sum of *v* and the value 47. They are two numbers and the result is 66. Next, add *true*, which is a *boolean*, and here occurs an automatic type conversion to a number where the value *true* is converted to 1 and the result of the sum is 67. In the third iteration, the plus is performed on a *number* and a *string*, and this time occurs an automatic conversion to a string for both operands. In the fourth iteration, the operator is executed on a *string* and an *object*, and the result is a string concatenation. The same goes for the last time, since *f()* is also an *object*, but the string - the result of *toString()* - is this time the code of the function itself.

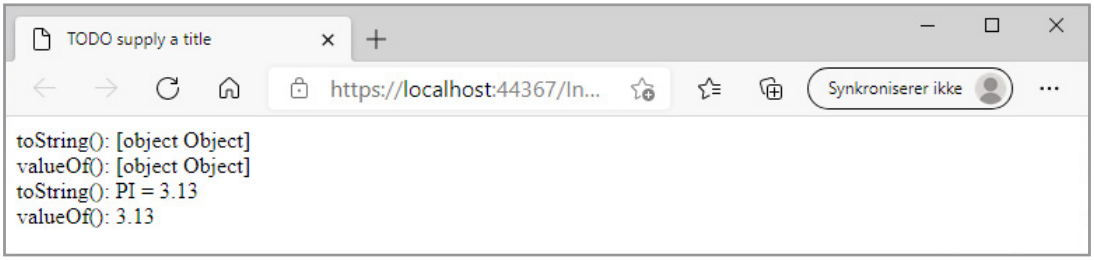
The example should show that as JavaScript is type-weak, an expression may well consist of different types of operands, and consequently implicit type conversions are always ongoing. This can naturally lead to unexpected results, and the rules are also complex. Basically, the following applies.

Note first that a variable of the type *object* has two methods *toString()* and *valueOf()* (which can be overridden), where the first returns the value of the object as a *string*, while the other returns the value as a *primitive*. For example, if you consider the following code:

```
<!DOCTYPE html>
<html>
  <head>
    <title>TODO supply a title</title>
    <meta charset="UTF-8">
    <script>
      var obj1 = {
        a: 2,
        b: 3
      };
      var obj2 = {
        x: 3.13,

        toString: function() {
          return "PI = " + this.x;
        },
```

```
        valueOf: function() {
          return this.x;
        }
      }
    </script>
  </head>
  <body>
    <script>
      document.write("toString(): " + obj1.toString() + "<br/>");
      document.write("valueOf(): " + obj1.valueOf() + "<br/>");
      document.write("toString(): " + obj2.toString() + "<br/>");
      document.write("valueOf(): " + obj2.valueOf() + "<br/>");
    </script>
  </body>
</html>
```

there are defined two objects where the latter overrides *toString()* and *valueOf()*. Opening the code in the browser is the result:



Here you can see that *toString()* and *valueOf()* as default return the string *[object object]*, but otherwise the value of the overridden methods.

For automatic type conversion, JavaScript uses three internal methods called *toPrimitive()*, *toNumber()* and *toBoolean()*. *toPrimitive()* has an argument and returns a *primitive* following the algorithm:

1. if the argument is an *object* returns *valueOf()* if it is a *primitive*, and else returns *toString()* if it returns a primitive and else reports an error
2. if the argument is a primitive return the argument

toNumber() also has an argument and returns a number according to the following algorithm:

1. if the argument has the type *Number* returns the argument
2. if the argument has the type *Boolean* returns 1, if the value is *true* and else 0
3. if the argument is *Null* returns 0
4. if the argument has the type *object* returns *toNumber(toPrimitive(argument))*
5. it the argument is *undefined* returns *NaN*
6. if the argument is a *string* returns the value of *parseInt()*, if the argument can be parsed to a *Number* and else *NaN*

Allso *toBoolean()* has an argument and returns a *boolean* according to the following algorithm:

1. it the argument has the type *boolean* returns the argument
2. if the argument is *Null* returns *false*
3. if the argument is *undefined* returns *false*
4. if the argument has the type *object* returns *true*
5. if the argument has the type *number* returns *false* if the value is 0 or *NaN* and else *true*
6. if the argument has the type *string* returns *false* if the string is empty and else *true*

As can be seen from these rules, it may be difficult to figure out the value of an expression, but in practice it rarely presents the big problems, since the value will usually be the expected, almost the value of a condition (an expression of the type *Boolean*) as well sometimes may results in a value other than expected, and here it is especially the operator == which causes problems. For this operator, the following table is used:

arg1	arg2	result
Null	Undefined	true
Undefined	Null	true
Number	String	arg1 == toNumber(arg2)
String	Number	toNumber(arg1) == arg2
Boolean	any	toNumber(arg1) == ang2
any	Boolean	arg1 == toNumber(arg2)
String or Number	Object	arg1 == toPrimitive(arg2)
Object	String or Number	toPrimitive(arg1) == arg2

A classic example of the problems that comparison may cause are the following code:

```
<script>
  if (“Hello”) {
    alert(“Hello” == true);
    alert(“Hello” == false);
  }
</script>
```

If you try the code, you will get two *alert()*, both of which will show *false* and it is not entirely obvious. The condition for *if* has the value *true*, which immediately follows from the *toBoolean()* algorithm. The first *alert()* will apply the third last row in the above table and will thus evaluate

```
“Hello” == toNumber(true)
“Hello” == 1
toNumber(“Hello”) == 1
NaN == 1
```

that is *false*. The last *alert()* essentially makes the same thing:

```
“Hello” == toNumber(true)
“Hello” == 0
toNumber(“Hello”) == 0
NaN == 0
```

The result is that it is not always easy to find out the result of a comparison with == (or !=). Therefore, the operator === has been introduced (and !==) which simply means comparison without type conversion, and many prefer to use === rather than ==.

Another thing that can cause problems is *null* and *undefined*. *null* means that a variable has no value and you can assign a variable the value *null*. *undefined* means that a variable is not yet assigned a value, and for example, the code will

```
<script>
  var a;
  document.write(a);
  document.write("<br/>");
  a = null;
  document.write(a);
</script>
```

results in

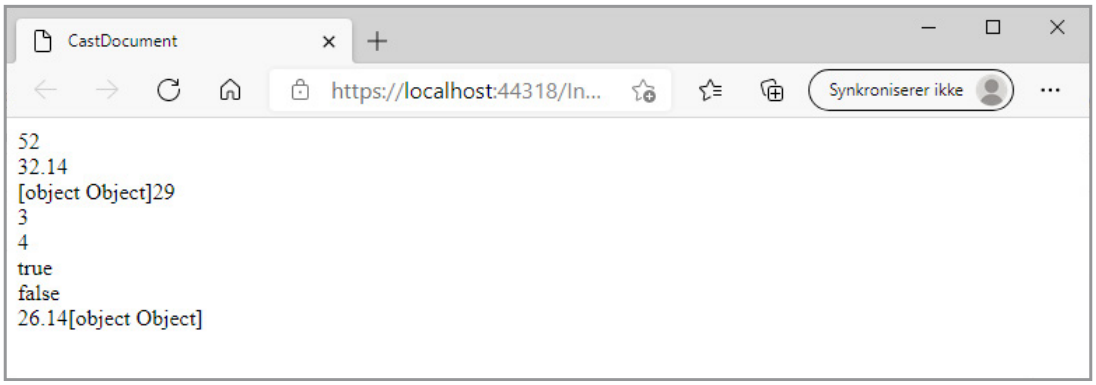
```
undefined
null
```

As an example of some of the above conversion rules, you can consider the document:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>CastDocument</title>
    <script>
      var obj1 = {
        a: 23,
        b: 3.14,
        toString: function () {
          return this.a;
        }
      }
      var obj2 = {
        a: 23,
        b: 3.14,
        valueOf: function () {
          return this.b;
        },
```

```
        toString: function () {
          return this.a;
        }
      }
      var obj3 = {
        a: 23,
        b: 3.14
      }
    </script>
  </head>
  <body>
    <script>
      document.write(obj1 + 29);
      document.write("<br/>");
      document.write(obj2 + 29);
      document.write("<br/>");
      document.write(obj3 + 29);
      document.write("<br/>");
      var t = 3;
      var u;
      document.write(3 + (t == 2));
      document.write("<br/>");
      document.write(3 + (t == 3));
      document.write("<br/>");
      document.write(3 == "3");
      document.write("<br/>");
      document.write(3 === "3");
      document.write("<br/>");
      (function () {
        document.write(obj1 + obj2 + obj3);
      })();

    </script>
  </body>
</html>
```

If the document is opened in the browser, you get the result:



Objects

An object is simply a collection of properties that may be of some value. The type of a property can be anything and especially an object, and since a function is an object, an object can also contain functions. You creates an object as follows:

```
var obj1 =
{
  a: 2,
  b: 3,
  c: 5
}
var obj2 = {}
```

where two objects have been created. The first has three properties, while the other is an empty object. In JavaScript, you do not have classes, as you know it from C#, and you can not inherit the same way you know from this language, but in JavaScript you use a form of inheritance called *prototyping* that in short means that you can create objects by expanding existing objects. For example, if you write

```
var obj3 = Object.create(obj1);
obj3.d = 7;
```

an object has been created that inherits *obj1* and expands this object with an additional property. For example, if you performs the following statements:

```
document.write(obj3.a + obj3.b + obj3.c + obj3.d);
obj1.a = 11;
document.write (obj3.a + obj3.b + obj3.c + obj3.d);
```

the first statement will show 17 and the other 26. Here, you should note that changing the value of the property *a* on the object *obj1* also has an effect on the object *obj3*, which justifies the relationship between *obj1* and *obj3* being a form of inheritance. Technically, it is implemented by the fact that each object has a property called *prototype*. This property refers to properties inherited from the parent object (the object from which the current object is created based on). When you try to refer a property on an object, the interpreter will first search for this property in the current object and if it not find the property the interpreter will search in its parent via *prototype*. This continues until you either find the desired property or can not find it. This chain may be interrupted by overriding a property. As an example is shown another object that expands *obj1* using a method:

```
var obj4 = Object.create(obj1);
obj4.f = function (t) { return t * (this.a + this.b + this.c); }
```

Note that although the method *f()* is a property in the object *obj4* which extends *obj1*, the method can not immediately refer to the object's other properties, but it can be solved with *this* as referring to the current object. The following statement

```
document.write (obj4.f(2));
```

will write the value 38. As another example of how to define an object using a method, you can consider the following:

```
var obj5 =
{
  x: 3.14,
  y: 1.41,
  g: function (z) { return (this.x + this.y) / z; }
}
```

If you perform the following statement

```
document.write (obj5.g(3));
```

you get the result `1.5166666666666`.

Since *obj1* is as above, you usually refer to the individual properties using the dot notation, but you can also use array notation, which uses the name of the property as an index:

```
document.write (obj1.a);
document.write (obj1["a"]);
```

Both of these statements will write the value 11. The main application of the last notation is that it allows to loop over an object's properties without knowing their names:

```
for (var t in obj1)
{
  document.write("<br/>");
  document.write(obj1[t]);
}
```

This statement will alert 11, 3 and 5.

You can create objects in three ways. You can create objects directly as shown above with *obj1* and *obj5*, that is, you directly write the properties that the object should have. You can also create an object with *Object.create()* such as *obj3* and *obj4*, where an object is created as an extension of another object. Finally, you can create objects using a constructor. Consider the following function:

```
function pointConstructor(x, y)
{
  this.x = x;
  this.y = y;
  this.dist = function (p) {
    return Math.sqrt((this.x - p.x) * (this.x - p.x) +
      (this.y - p.y) * (this.y - p.y));
  }
  this.toString = function() { return "( " + this.x + ", " + this.y + " )"; }
}
```

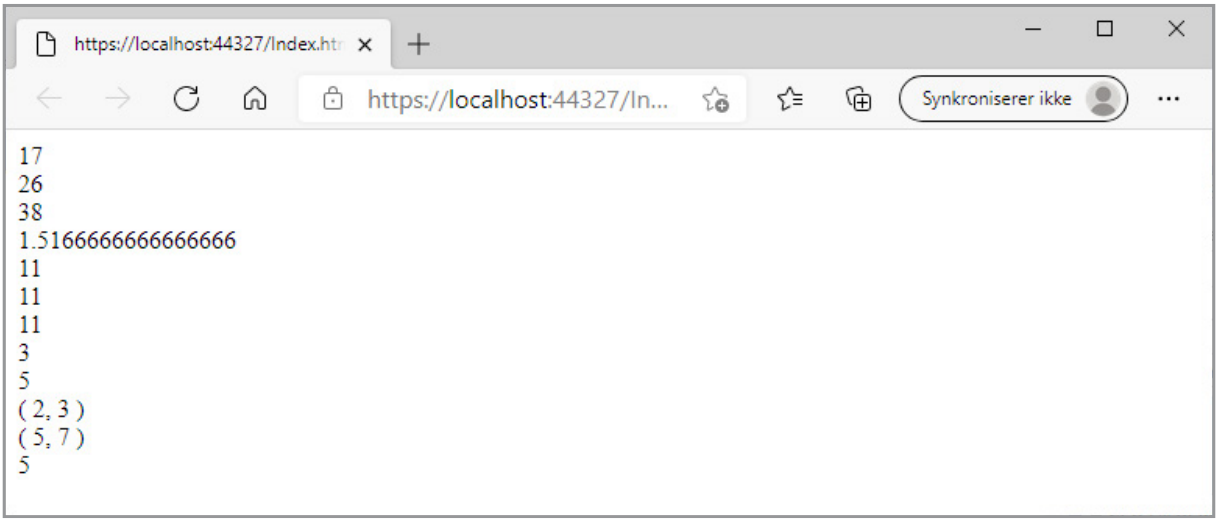
The function creates an object with 4 properties, where the last two are methods. An object is perceived as a point in a coordinate system, and the method *dist()* has a parameter that is to be interpreted as a point. The method returns the distance between the current point and the parameter *p*. The last method is an override of *toString()*. You should note that the method has parameters, but it is not necessary. Below is how to use the constructor method to create two objects:

```
var p1 = new pointConstructor(2, 3);
var p2 = new pointConstructor(5, 7);
```

The following statement use the objects *p1* and *p2*:

```
document.write(p1 + "<br/>" + p2 + "<br/>" + p1.dist(p2));
```

If you opens the program (*CreateDocument*) in the browser the result is:



In principle, it does not matter whether an object is created in one way or another and you use the most appropriate way. However, you must note that the constructor way resembles how to create objects in C#.

I will finish this section on objects with an example of a *cup* object representing a 5-cube cup where a cube should be represented by a *cube* object. The example applies in addition more control statements (loops) and also arrays that are dealt with in the next section. The example will also use the module concept described in the previous section. The code is as follows:

```
<!DOCTYPE html>
<html>
  <head>
    <title>CubesProgram</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <script>
      var cup = (function() {
        function cube() {
          this.eyes = 1;
          this.roll = function () {
            this.eyes = Math.floor(Math.random() * 6) + 1;
          };
          this.valueOf = function () {
            return this.eyes;
          }
        };

        public = {
          cubes: [new cube(), new cube(), new cube(), new cube(), new
          cube()],

          toss: function () {
           for (var i = 0; i < this.cubes.length; ++i) this.cubes[i].roll();
          },

          yatzy: function () {
            var t = this.cubes[0].eyes;
            for (var i = 1; i < this.cubes.length; ++i)
              if (this.cubes[i] != t) return false;
            return true;
          },

          toString: function () {
            var str = this.cubes[0].eyes;
            for (var i = 1; i < this.cubes.length; ++i)
              str += " " + this.cubes[i].valueOf();
            return str;
          }
```

```
        };

        return public;
      })();
    </script>
  </head>
  <body>
    <h1>Play Yatzy</h1>
    <script>
      var count = 0;
      do {
        cup.toss();
        document.write(cup + "<br/>");
        ++count;
      }
      while (!cup.yatzy());
    </script>
    <h3>You've got yatzy after <script>document.write(count)</script>
    attempts</h3>
  </body>
</html>
```

In the header section is defined a module called *cup*, which represented a cup with 5 cubes and three *public* attributes:

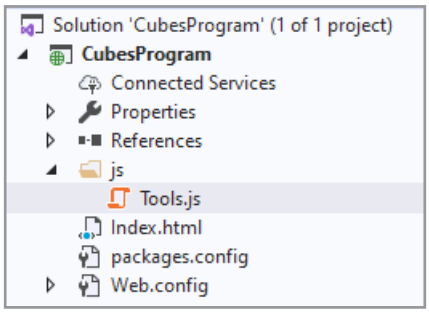
1. *toss()*, that is a method, which simulates the use of the cup
2. *yatzy()*, which is a method, that tests where all 5 cube values are the same
3. *toString()*, that is an override of *toString()*

Initially, a constructor method is defined that creates a *cube* object. An object *public* is defined which uses the constructor method to create an array of 5 *cube* objects. The public object has three functions that implement the above methods. The body part simulates using the cup until all cubes have the same value.

If you run the program (opens it in the browser) it simulates that you turn the cup until you get yatzy.

EXERCISE 2

Create a copy of the program *CubesProgram*. You must then create a folder *js* and add a JavaScript file called *Tools.js* to that folder:



You must then move all your script code from *Index.html* to *Tools.js* - but without the script elements. *Index.xhtml* must have a link in the header to the JavaScript file (drag the filename in Visual Studio). The result of *Index.xhtml* should be:

```
<!DOCTYPE html>
<html>
  <head>
    <title>CubesProgram</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <script src="js/Tools.js"></script>
  </head>
  <body>
    <h1>Play Yatzy</h1>
    <script>
      var count = 0;
      do {
        cup.toss();
        document.write(cup + "<br/>");
        ++count;
      }
      while (!cup.yatzy());
    </script>
    <h3>You've got yatzy after <script>document.write(count)</script>
    attempts</h3>
  </body>
</html>
```

Test the program, at it should works as before.

When using JavaScript, you will usually place the code in its own file, as shown in this exercise.

4.4 ARRAYS

Arrays works immediately in the same way as in other programming languages and with the same syntax known from C#, but there are also important differences. In JavaScript arrays are dynamic and should not be allocated for a certain size. The following function writes the number of elements in an array as well as the array's elements:

```
function show(arr) {
  var str = arr.length + "<br/>";
  for (var i = 0; i < arr.length; ++i) str += "<br/>" + arr[i];
  document.write(str + "<br/><br/>");
}
```

If you consider this function, note that syntactically, an array is used in the same way as in C#. An array has a length property, and the individual elements are referenced via an 0-based index.

```
var arr1 = ["Svend", "Knud", "Valdemar"];
show(arr1);
```

The statement creates an array of three elements and sends this array to the function *show()*, you will get the result:

```
3

Svend
Knud
Valdemar
```

Since an array has no type, an array can contain elements of different types:

```
var arr2 = [”Gorm”, 3.13, { x: 2, y: 3 }, obj1];
show(arr2);
```

where the array contains a string, a number and two objects. The last object is:

```
var obj1 = {
  x: 2,
  y: Math.sqrt(2),
  toString: function () {
    return this.x + ‟ -> ‟ + this.y;
  }
}
```

and if you the array print the array using the method *show()* the result is:

```
4

Gorm
3.13
[object Object]
2 -> 1.4142135623730951
```

You can also create an array with a constructor function:

```
var arr3 = new Array();
```

and the result is an empty array, and if you print the array the method prints an empty array. If you have an array, you can immediately add items, and the array will dynamically expand:

```
arr3[0] = 11;
arr3[1] = 13;
arr3[2] = 17;
arr3[3] = 19;
show(arr3);
```

and the array now have 4 elements:

```
4

11
13
17
19
```

An array may also have "holes", and thus elements that are not defined:

```
arr1[5] = "Erik";
show(arr1);
```

and the result is:

```
6

Svend
Knud
Valdemar
undefined
undefined
Erik
```

Constructor functions can have parameters:

```
var arr4 = new Array(4);
var arr5 = new Array(23, 29, 31, 37);
```

and here the first creates an array of length 4 with 4 undefined elements, while the other creates an array with 4 elements. Finally, you should note that you can change the value of the property *length* and thus delete items in an array. As mentioned, you reference the elements in an array using a numeric index, but indexing is actually more flexible than you would expect. Consider the following code:

```
var arr6 = new Array();
arr6[0] = 2;
arr6[1] = 3;
arr6["3"] = 5;
arr6["abc"] = 7;
show(arr6);
```

First, note that the code does not fail and it will display the following alert:

```
4

2
3
undefined
5
```

If the index is not a number, the interpreter will try to convert it to an integer and the result of

```
arr["3"] = 5;
```

is therefore the element with index 3. On the other hand, it is not immediately clear what

```
arr["abc"] = 7;
```

means when "abc" can not be converted to a number. An array is an object, and therefore you can specifically define its own properties. This is exactly what happens here as *arr["abc"]* is interpreted as a property named *abc*, which then gets the value 7. There is reason to be aware of this interpretation as it is easy to accidentally add properties to an array. In this case the statement

```
document.write(arr.abc);
```

will show 7. The elements in an array can be anything and hence especially other arrays. It allows to simulate multidimensional arrays. For example will the following code write the value 130:

```
var v1 = [2, 3, 6, 7];
var v2 = [11, 13, 17, 19];
var v3 = [23, 29];
var v = [v1, v2, v3];
var s = 0;
for (var i = 0; i < v.length; ++i) for (var j = 0; j < v[i].length;
++j) s += v[i][j];
document.write(s);
```

In particular, note how to iterates the array *v* as a 2-dimensional array, but it is the programmer's responsibility that the elements in *v* actually are arrays. Otherwise, you will get an error. Also note that *v* illustrates a heterogeneous array and thus an array where the rows do not have the same length.

In fact, JavaScript is quite flexible as to how to define an array. Below is defined an array consisting of two arrays, and these arrays have elements of different types. The one even has an element that is not defined.

```
var kings = [["Gorm", , 958], ["Harald Blåtand", 958, 986]];
for (var i = 0; i < kings.length; ++i) show(konger[i]);

3

Gorm
undefined
958

3

Harald Blåtand
958
986
```

You should note that there are a number of standard functions for arrays that I mentions in a later section.

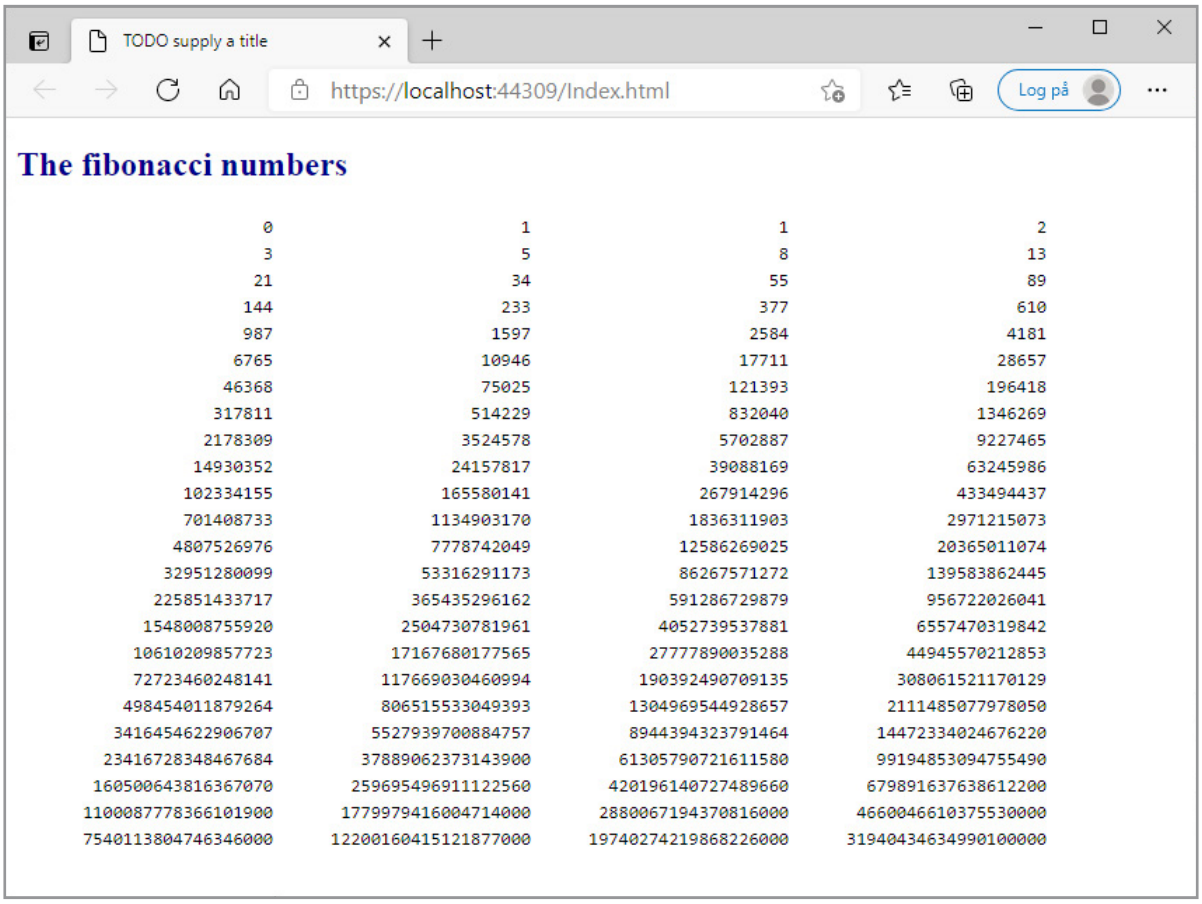
EXERCISE 3

The Fibonacci numbers are, as you know, the following sequence:

```
0, 1, 1, 2, 3, 5, 8, 13, 21, ....
```

Create a new project, that you can call *FiboProgram*. The project must have a start page *Index.html* that shows the first 95 fibonacci numbers in a table (se the window below). The program should be written as follows:

1. The project must have a JavaScript file, that contains a module called *fibonacci*, which automatically creates a private array, that contains the Fibonacci numbers. The module must have two public properties, where the first is called *length*, and is a simple property whose value is the length of the array, while the other property is a function *get(n)*, that returns the *n*th Fibonacci number.
2. The table and the header text should be styled by a simple style sheet.
3. The start page *Index.html* must dynamic generate the table in JavaScript.



4.5 FUNCTIONS

In JavaScript is a function as mentioned an object. That means more things, for example that a function can be return value from another function and that a function can be a parameter to another function. It also means that a function may have properties and methods like any other object. These relationships means that functions in many ways are different from functions in other languages like C#.

A function is usually created with the reserved word function:

```
function factorial(n)
{
}
```

and the function may have an arbitrary number of parameters and especially none. Since a function can refer to all variables in its own scope, the function can specifically refer to itself and you can therefore write recursive functions:

```
function factorial(n)
{
  if (n < 1) return 1;
  return n * factorial(n - 1);
}
```

It is allowed to declare a function multiple times in the same scope, which simply means that the function name is given a different value. It can also give unexpected results. For example, if you consider the following code:

```
function f()
{
  return 23;
}
document.write(f());
function f()
{
  return 29;
}
```

it will write 29. The reason is that JavaScript collects all definitions at the start of a scope and the above is thus equivalent to:

```
function f()
{
  return 23;
}
function f()
{
  return 29;
}
document.write(f());
```

In JavaScript, you can also define a function as an expression:

```
var show = function (arr) {
  var str = arr.length + "<br/>";
  for (var i = 0; i < arr.length; ++i) str += "<br/>" + arr[i];
  document.write(str);
}
show([2, 3, 5, 7]);
```

The variable *show* refers to an anonymous function, and the use of anonymous functions is widely used in JavaScript. Consider the following example:

```
function validate(x, ok) {
  if (ok(x)) return true;
  document.write("<br/>Error: " + x + " is an illegal value<br/>");
  return false;
}
```

The function has two parameters, the latter being interpreted as a function that will validate the first parameter. The function could, for example, be used as follows:

```
validate(1234, function (t) { return t >= 100 && t <= 999; });
```

where the function should validate if the first argument is a 3-digit number. The validation function is sent as an anonymous function.

When a function is performed, it occurs in a given context, and the function has access to variables, objects, and other functions in this context through its scope. Internally, this context is referenced by the *this* pointer, which is indirectly transferred to the function when it is called. In this context, it is necessary to distinguish between a function and a method where a method is a function defined as a property in an object. If you perform a usual function, its context will be the window object, while the context of a method is the parent object. It can be illustrated by the following code, where *m* is a method in the object *obj*, while *h()* is a global function with an inner function *g()*.

```
var obj =
{
  m : function() { alert(this === obj); }
}
function h()
{
  alert(this === window);
  function g()
  {
    alert(this === window);
  };
  g();
}
obj.m();
h();
```

If you execute the code, it will write *true* three times. First, the method *m()* is executed and its context is the object *obj*. Next, *f()* is executed and it starts performing the internal function *g()* whose context is the window object in the same way as the external function *h()*.

The most frequent use of *this* is in connection with methods and constructor functions. If you have a function and type *new* in front of the function, it means creating an empty object, and this will then refer to the context for this object. As an example, below is shown a constructor function that creates a point:

```
function createPoint(x, y)
{
  this.x = x;
  this.y = y;

  this.move = function (p) { this.x = p.x; this.y = p.y; };
  this.add = function (p) { this.x += p.x; this.y += p.y; };
  this.scale = function (t) { this.x *= t, this.y *= t; };
  this.dist = function (p) {
   return Math.sqrt((this.x - p.x) * (this.x - p.x) +
    (this.y - p.y) * (this.y - p.y)); };
  this.toString = function() { return "( " + this.x + ", " + this.y + " )"; }
}
```

The following code uses the function to create two objects, and you should note in particular that the objects do not inherit each other and each have there own variables on which they work on:

```
var p1 = new createPoint(0, 0);
var str = p1.toString();
var p2 = new createPoint(2, 3);
str += "<br/>" + p2;
p1.move(p2);
str += "<br/>" + p1;
p1.add(p2);
str += "<br/>" + p1;
p1.scale(2);
str += "<br/>" + p1;
str += "<br/>" + p1.dist(p2);
document.write(str);
```

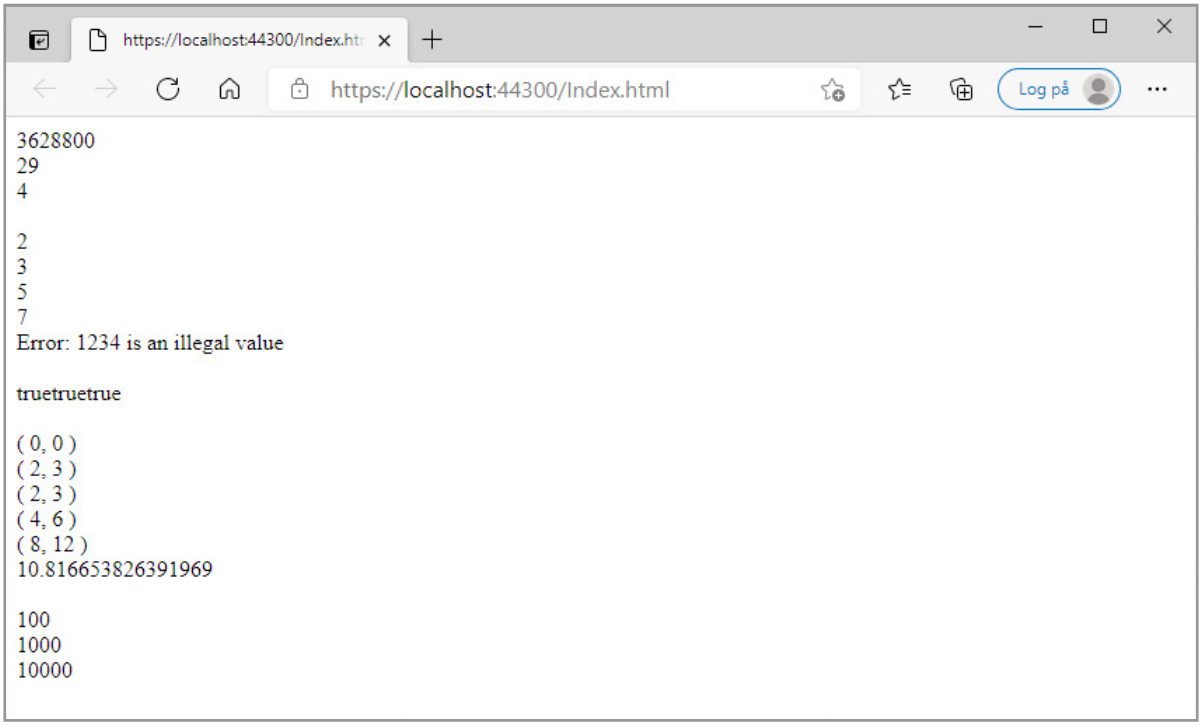
Also note that the above recalls the class concept from an object-oriented programming language, but it has nothing to do with classes - JavaScript has only objects.

Usually, a function is performed by specifying parentheses after the function name, if any, the actual parameters are written between parentheses. The parentheses () are in this context called for the *function invokes operator*. However, there is an alternative to calling a function. You can use *apply()* or *call()*, where you can specify the context in which the function should work within. Consider the following code:

```
var scale = 10;
var obj1 =
{
  scale: 100
}
var obj2 =
{
  scale: 1000
}
function sum(x, y, z)
{
  return this.scale * (x + y + z);
}
document.write(sum(2, 3, 5));
document.write (sum.apply(obj1, [ 2, 3, 5]));
document.write (sum.call(obj2, 2, 3, 5));
```

It will write the values 100, 1000 and 10000 respectively. At the top is defined a variable called scale that has the value 10. Next, two objects are defined which each have a property called scale and their values are 100 and 1000 respectively. These objects each define their context (scope). The function *sum()* has three parameters and returns the sum of these parameters after multiplying the sum with the value of a variable *scale* from the current context. The first write statement shows the value of this function by calling it in the normal way, and since it is a global function, its context will be the window object, and that is why the global *scale* variable is used. The next write statement performs the function again, but this time using *apply()*. The first parameter is the context in which the function should work, and here it is *obj1*. That is that *scale* this time is the property *scale* from *obj1* that has the value 100. Note that the parameters are transmitted as an array. The last write statement works in principle in the same way, but this time the function *call()* is used and *obj2* is used as a context. The difference between *apply()* and *call()* is that in the first one you specify parameters for the function as an array, while the other indicates the parameters as a list.

If you opens the program *FunctionDocument* in the browser, the result is:



4.6 PROGRAM CONTROL

As the last section regarding the basic syntax and semantics I want to show the language's control statements, and here is not much to explain. Secondly, I have already used most of the control statements several times, and basically, there are the same statements as in C#.

Regarding conditions, JavaScript has an *if* statement, which has the same syntax as in C#, and the semantics are also the same. The only difference is the condition, which should be an expression that evaluates a *boolean*, and here corresponding to the conversion rules described above, there is somewhat greater flexibility (and possibilities of errors) than the case is in C#.

JavaScript also has a *switch* statement that also works in the same way as in C#. The same goes for *while* and *do* loops. The most commonly used loop is as in C# the *for* loop, and the classic *for* loop works exactly the same as in C#:

```
for (var i = 0; i < arr.length; ++i) s += arr[i];
```

In C#, you also have a *foreach* loop, which is actually an iterator. There is a corresponding variant of a *for* loop in JavaScript, sometimes called *for-in*. It can be used to iterate all properties in an object. Below is an object with 4 properties and the following loop determines the sums of these four properties with an *for-in* loop:

```
var obj =
{
  a: 2,
  b: 3,
  c: 5,
  d: 7
}
var s = 0;
for (var t in obj) s += obj[t];
document.write(s);
```

4.7 GLOBAL OBJECTS AND FUNCTIONS

JavaScript is born with several global objects and some global functions that are immediately available and in terms of methods offers a variety of services. The global objects are

1. Array
2. Boolean
3. Date
4. Math
5. Number
6. RegExp
7. String

I do not want to review these objects and their methods here, but the names tell you a bit about what you can do with them and you are encouraged to investigate the properties of the objects (there are many reefs on the internet describing these objects and methods). Finally, there are the following global functions, whose behavior you are also encouraged to investigate:

1. eval()
2. isFinite()
3. isNaN()
4. parseFloat()
5. parseInt()

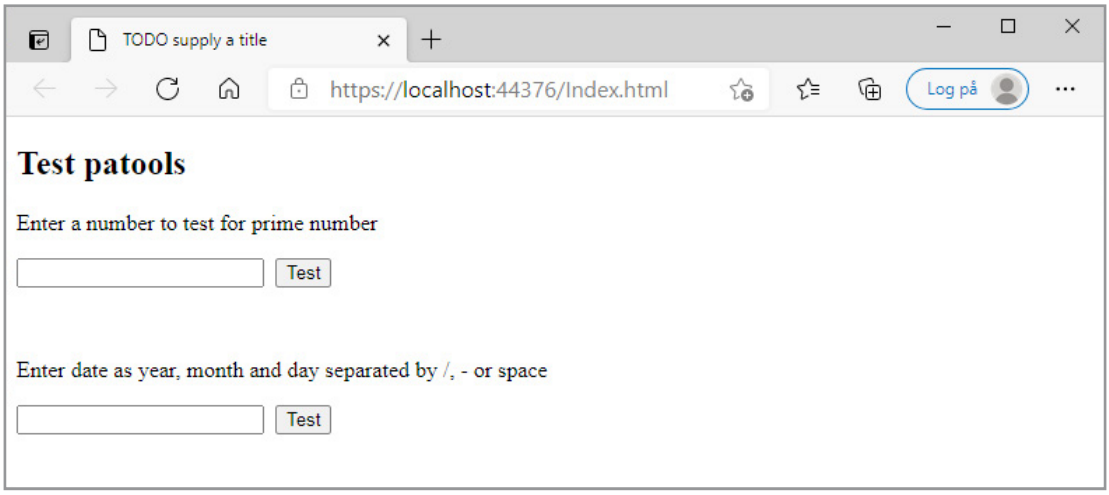
EXERCISE 4

Create a web application, that you can call *ScriptLibrary*. Create a folder *js* and add a JavaScript file with the name *Tools.js*. The file should implements a module called *patools* (or whatever name you want), and you should think of the module as a JavaScript library. The library should have four properties:

1. *isInt*, that is a function with one parameter, and the function should returns *true*, if the parameter is a legal integer.
2. *isPrime*, that is a function with one parameter, and the function should returns *true*, if the parameter is a prime number.
3. *isLeapyear*, that is a function with one parameter, and the function should returns *true*, if the parameter represents a leap year as a year between 1700 and 9999.
4. *isDate*, that is a function with one parameter, and the function should returns *true*, if the parameter represents a legal date between 1700 and 9999. The parameter must be a string of the form year, month and day (that is YYYY_MM_DD), where the separation character must be -, /, space or nothing.

When you have written the library methods, you should test the module from *Index.html* (see below). The html code could be:

```
<body>
  <h2>Test patools</h2>
  <form>
    <p>Enter a number to test for prime number</p>
    <input type="text" id="txtnumber"/>  
    <input type="button" value="Test" onclick="validate1();" />
    <p><span id="result1"></span></p>
    <br/>
    <p>Enter date as year, month and day separated by /, - or space</p>
    <input type="text" id="txtdate"/>  
    <input type="button" value="Test" onclick="validate2();" />
    <p><span id="result2"></span></p>
  </form>
</body>
```



You must write event handlers for the two buttons. The first must test where the value in the first input field is a prime number and update the first *span* element with a message. The second must do the same, but for the other elements and test where the value is a legal date.

The event handlers should be written in the script block in the header. Note that it is also necessary to add a link element for your JavaScript library.

4.8 DOM

DOM stands for *Document Object Model* and is a W3C standard for how browsers must build an HTML document as a hierarchy of objects. DOM is not part of JavaScript, but the DOM objects allow JavaScript to manipulate the objects and thus refer to the contents

of the current document from script code. In combination with JavaScript, DOM is the key to developing websites where something is happening on the client side. DOM is a standard, but it is up to the browser vendors to implement this standard and the standard is no better than to the extent that the browser vendors live up to it. This means that there may be (and are) differences between different browsers and different versions of the same browser, but the most basic objects are, however, implemented roughly standard.

The root of the DOM tree is called *window* and is an object consisting of several properties and collections. The main property is *document* that represents the document that the browser shows. Consider, for example, the following document (*DOMTree* project):

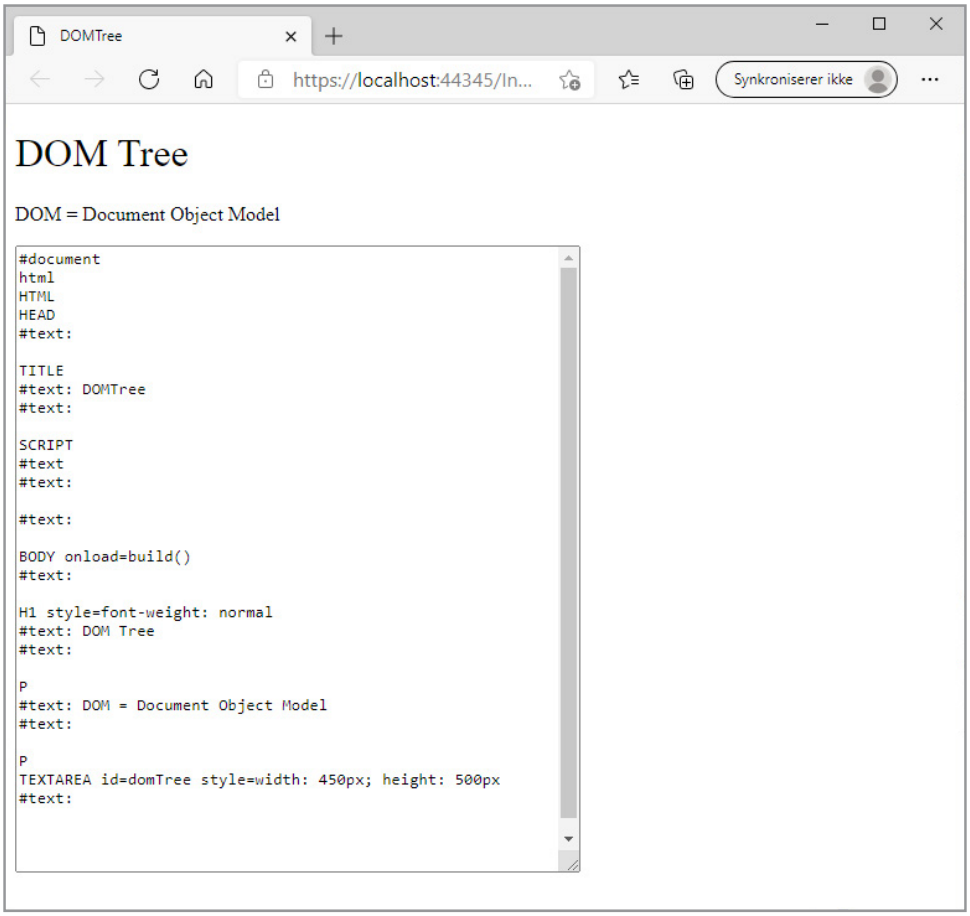
```
<!DOCTYPE html>
<html>
  <head>
    <title>DOMTree</title>
    <script>
      tree = "";
      function parse(node)
      {
        tree += node.nodeName;
        if (node.nodeName === "#text" && node.parentNode.nodeName !==
        "SCRIPT")
          tree += ": " + node.textContent;
        if (node.attributes != null)
          for (var i = 0; i < node.attributes.length; ++i)
            tree += " " + node.attributes[i].name + "=" + node.
            attributes[i].value;
        tree += "\n";
        if (node.childNodes != null)
          for (var j = 0; j < node.childNodes.length; ++j)
            parse(node.childNodes[j]);
      }
      function build()
      {
        parse(document);
        document.getElementById('domTree').value = tree;
      } </script>
  </head>
  <body onload="build()">
    <h1 style="font-weight: normal">DOM Tree</h1>
    <p>DOM = Document Object Model</p>
    <p><textarea id="domTree" style="width: 450px; height: 500px"></
    textarea></p>
  </body>
</html>
```

In the DOM tree, each element in the document is represented by a node, and it also applies to elements that are not visible, such as a *script* element. The typical use of JavaScript is to modify the DOM tree, including specifically changing the individual nodes. In this case, the script element begins by defining a global variable. The function *parse()* has a node as a parameter, and it starts by adding this node's name to the global variable *tree*. If attributes are attached to the node, they are also added to the variable as key / value pairs. Finally, if there are child nodes, the function *parse()* is called recursively for each child node.

You must note the syntax for the function *parse()*. Of course, it is not so easy to guess what the individual properties are called, but when you see the result, it's easy to follow what happens.

There is another function called *build()* which calls the above recursive function with the *document* object as parameter, that is the root of the DOM tree. When the function *parse()* is performed, the function *build()* will with a reference to the element with id *domTree* (which is a text area) insert the value of the global variable *tree*. The *build()* function is performed as an event handler for *onload* that occurs after the document is loaded and displayed in the browser, and after that the DOM tree is built.

If you opens the document in a browser, you get the window below, and here you can see how each element in the document is represented by a node in the DOM tree.



Refer elements in DOM

With regard to the use of DOM, the first step is to learn how to refer to the individual elements of the DOM tree and then do something with these elements. This is basically done by referring to the elements *id* attributes or the elements *class* attribute, which is illustrated in the introduction to this chapter with the example *SimpleDocument*. The example shows a little about what options are available to find a particular element or array of elements in the DOM tree. The basic methods are:

- *getElementById()*
- *getElementsByTagName()*
- *getElementsByClassName()*
- *querySelector()*
- *querySelectorAll()*

Here, the last two are the most advanced, as they use CSS selectors as explained in chapter 2 of this book, and the difference is that the first returns the first element that matches the selector, while the other returns all the elements that match. In addition, a node has the following properties:

- *firstChild*
- *lastChild*
- *nextSibling*
- *previousSibling*

that can be used to traverse the DOM tree.

4.9 MODIFY THE DOM TREE

It is also possible to modify the DOM tree using JavaScript, and although already done in the above examples there are many more options. You can modify the elements properties and change their content, and you can even move an element in the tree. You can also delete an element, and you can insert new elements. Again, I will illustrate some of the possibilities with an example:

```
<!DOCTYPE html>
<html>
<head>
  <title>Modify document elements</title>
  <style>
    .blueClass {
      background-color: lightsteelblue;
    }

    .plainClass {
      font-weight: normal;
    }
  </style>
</head>
<body>
  <h1>Header</h1>
  <h3 id="txt">Modifies DOM elements using Javascript</h3>
  <p id="adr"><a>Torus data</a></p>
  <p id="par">
    More text about <span style="font-weight:bold">
      DOM
    </span> objects
  </p>
  <script>
    var link = document.querySelector("#adr a");
    link.href = "http://bookboon.com";
    link.style.backgroundColor = "#ff0000";
    link.style.color = "#ffffff";
    link.target = "_blank";
    var txt = document.getElementById("txt");
    txt.classList.add("blueClass");
    txt.classList.add("plainClass");
    document.getElementsByTagName("h1")[0].innerText = "Hello World";
    par.innerHTML = "<ul><li>Svend</li><li>Knud</li><li>Valdemar</li></ul>";
    var head = document.head;
    var elem = document.createElement("script");
    elem.innerText =
"function factorial(n) { if (n < 2) return 1; return n * factorial(n - 1); }";
    head.appendChild(elem);
    var body = document.body;
    var text = document.createElement("p");
```

```
    text.id = "fact";
    body.appendChild(text);
    document.getElementById(
      "fact").innerHTML = "10! = <b>" + factorial(10) + "</b";
    document.write("<br/>" + par.innerText);
    document.write("<br/>" + par.innerHTML);
  </script>
</body>
</html>
```

The page defines in the header two classes, called respectively *blueClass* and *plainClass*. The document has 4 elements as a *h1* element, a *h3* element and two paragraphs. The first paragraph contains a *link* and the other a *span* element. Then follows a script block.

The first paragraph has an *id* named *adr*. This paragraph has a *link* (an element) as a child element, and the variable *link* is set to refer to this element. Here you should especially note the method *querySelector()* and the syntax to refer to a child element for the element with *id adr*:

```
var link = document.querySelector("#adr a");
```

The following statements are used to modify properties of this element.

The variable *txt* refers to the *h3* element. An HTML element has (in HTML5) a list for *class* objects. The reference *txt* is used to add *class* objects to the element *h3*. The result is that *h3* gets a blue background and the text is displayed as normal. The next statement:

```
document.getElementsByTagName("h1")[0].innerText = "Hello World";
```

changes the text in the element *h1* by changing the element's *innerText*. You must primarily note the reference (the *h1* element has no *id*), and *getElementsByTagName()* is an array with all *h1* elements, and you get the first (and in this case only) by referring the item with index 0.

par are *id* for a paragraph containing some HTML and the following statement replaces this HTML with a list of 3 elements:

```
par.innerHTML = "<ul><li>Svend</li><li>Knud</li><li>Valdemar</li></ul>";
```

You can therefore specifically change a DOM object to something completely different. Next, two variables *head* and *elem* are defined, which refer respectively to the document's header as well as a new element that is a *script* element. You can in that way create new HTML elements:

```
var elem = document.createElement("script");
```

Next, this element's *innerText* is defined as a JavaScript function. The new script element is added to the *head* variable as a child node. This means that in the header part of the document there dynamically is added a script element with a JavaScript function. Next, two more variables are defined. *body* is a reference to the body of the document, and *txt* is a reference to a paragraph element. The new (and empty) paragraph is assigned an *id* and is added to the body of the page, meaning it is inserted as an empty node in the DOM tree. Finally, the content of the new paragraph is changed to some HTML. Note that it includes the dynamically added JavaScript function being executed and the value is inserted in the new paragraph:

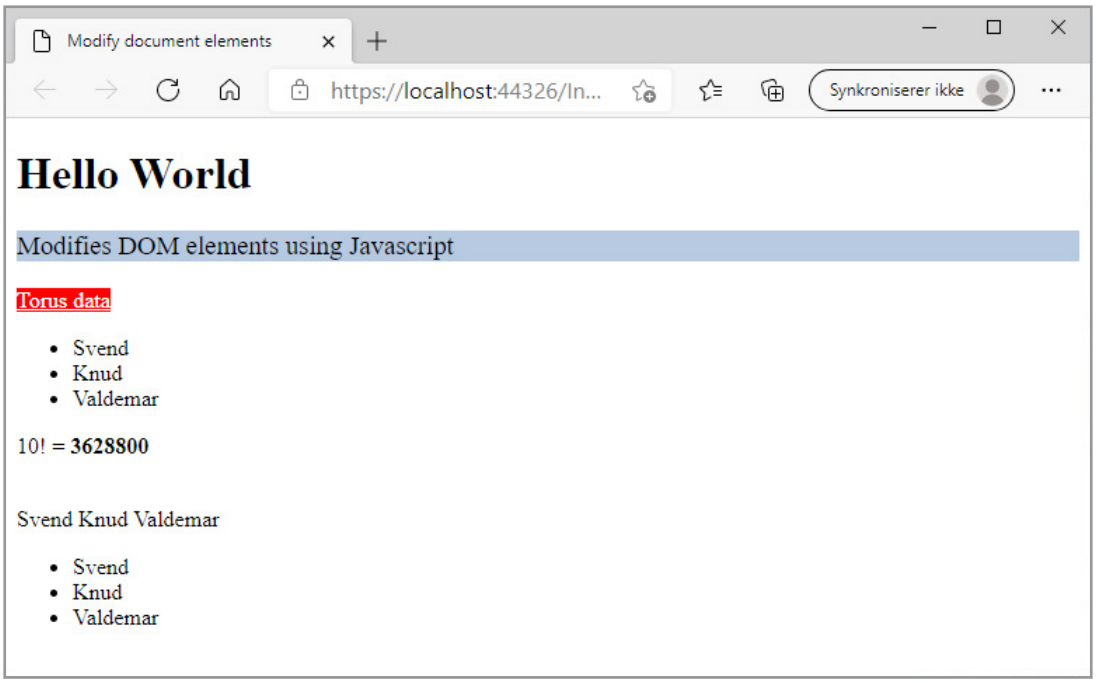
```
document.getElementById("fact").innerHTML = "10! = <b>" + factorial(10)
+ "</b";
```

As the last is written the *innerText* of the node referenced by *par*:

```
document.write(par.innerText);
```

which shows the text in the last paragraph. The paragraph is referred to by its ID, and you should note that you can do it directly. You should also note that the HTML element *span* is not displayed as part of *innerText*. Unlike *innerText, innerHTML* shows the entire text including HTML elements.

Running the application (*ModifyDocument*) opens the following browser:



Events

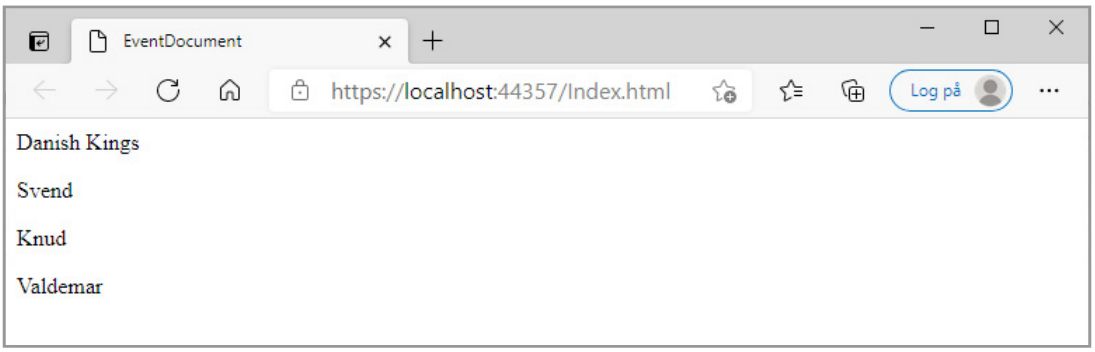
DOM defines exactly as events how to interact with the individual elements. Most events relates to the mouse, and for a particular event you can associate an event handler with an element in the document. When a user interacts with an element, the browser checks whether an event handler has been registered for that element, and if so, the handler is executed. The elements are in a document part of a hierarchy and an event will bubble up through the tree of all parent elements, and for all parent elements where an event handler has been registered for that event, this will be executed. When the event reaches the top - and that is, the body element - an event will follow the same path back until it returns to the element that originally raised the event. The two phases are called the *bubbling phase* and the *capturing phase* respectively, and you can specify the phase in which you wish to process the event.

When an event handler is performed, it is done in the same way as any other JavaScript function within a context, and DOM defines that it is the context of the element to which that particular handler is related. You can therefore refer to the element that the event concerns with the *this* reference.

DOM defines the following events:

- events for the mouse: *click*, *mousedown*, *mouseup*, *mousemove*, *mouseover*, *mouseout* and more
- events for the keyboard: *keypress*, *keydown*, *keyup*
- events for objects: *load*, *error*, *scroll*
- events for forms: *select*, *change*, *submit*, *reset*, *focus*
- events for user interaction: *focusin*, *focusout*

Consider, for example, a page with the following content (the project *EventDocument*):



Here the upper text is a *div* element, while the three bottom are paragraphs. The code is as follows:

```
<!DOCTYPE html>
<html>
  <head>
    <title>EventDocument</title>
    <script>
      var text;

      function kingCapture()
      {
        text += "\n" + this.innerText + " capture";
      }

      function kingBubble()
      {
        text += "\n" + this.innerText + " bubble";
      }
```

```
      function kingsCapture()
      {
        text = "Kings capture";
      }

      function kingsBubble()
      {
        text += "\nKings bubble";
        alert(text);
      }
    </script>
  </head>
  <body>
    <div>Danish Kings</div>
    <div>
      <p id="paragraf1">Svend</p>
      <p id="paragraf2">Knud</p>
      <p id="paragraf3">Valdemar</p>
    </div>
    <script>
      var p1 = document.getElementById("paragraf1");
      var p2 = document.getElementById("paragraf2");
      var p3 = document.getElementById("paragraf3");
      p1.addEventListener("click", kingBubble);
      p2.addEventListener("click", kingBubble);
      p3.addEventListener("click", kingBubble);
      p1.addEventListener("click", kingCapture, true);
      p2.addEventListener("click", kingCapture, true);
      p3.addEventListener("click", kingCapture, true);
      document.getElementsByTagName("div")[1].
        addEventListener("click", kingsCapture, true);
      document.getElementsByTagName("div")[1].
        addEventListener("click", kingsBubble);
    </script>
  </body>
</html>
```

In the header are defined two simple event handlers. You should note the use of the *this* reference, which will refer to the element that has raised the event. The HTML code requires no explanations, but it is the subsequent script block that assign event handlers to the HTML elements. First, there are references to the three paragraphs, and then two event handlers are assigned for a click event for each paragraph. The one is a handler for the bubbling phase (*false*), and the other is a handler for capturing the phase (*true*). You should note how to associate an event handler with the method *addEventListener()*. As a result, clicking on one of the three paragraphs will cause an alert to show which events

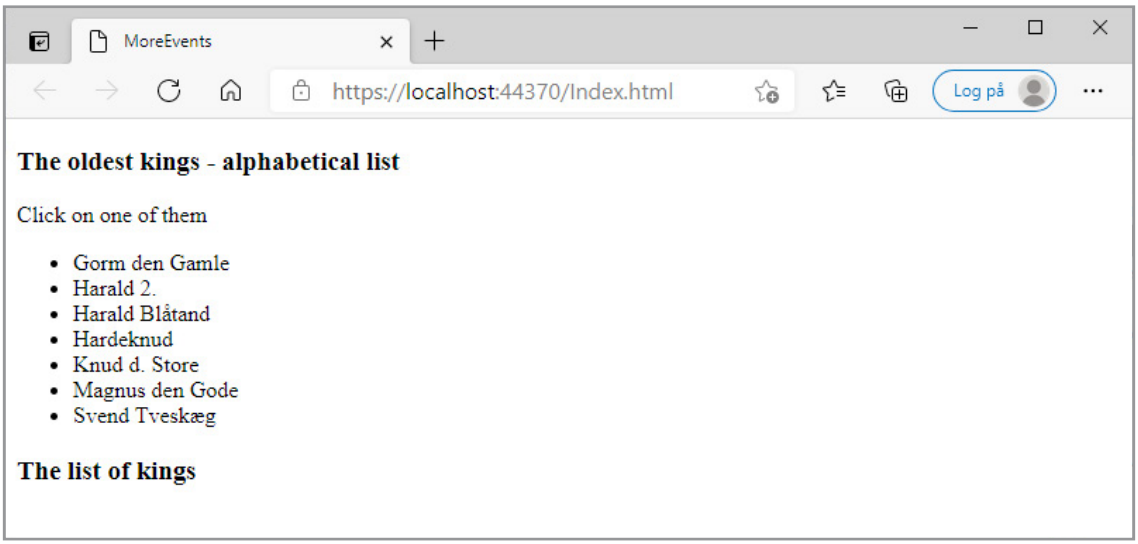
are fired. The last statement in the script block assigns an event handler to the first *div* element. Here you should notice how to find the item and how to set the event handler as an anonymous function.

There is also a method called *removeEventListerner()* and that has the same parameters as *addEventListener()*. This function is used to remove an event handler from an element.

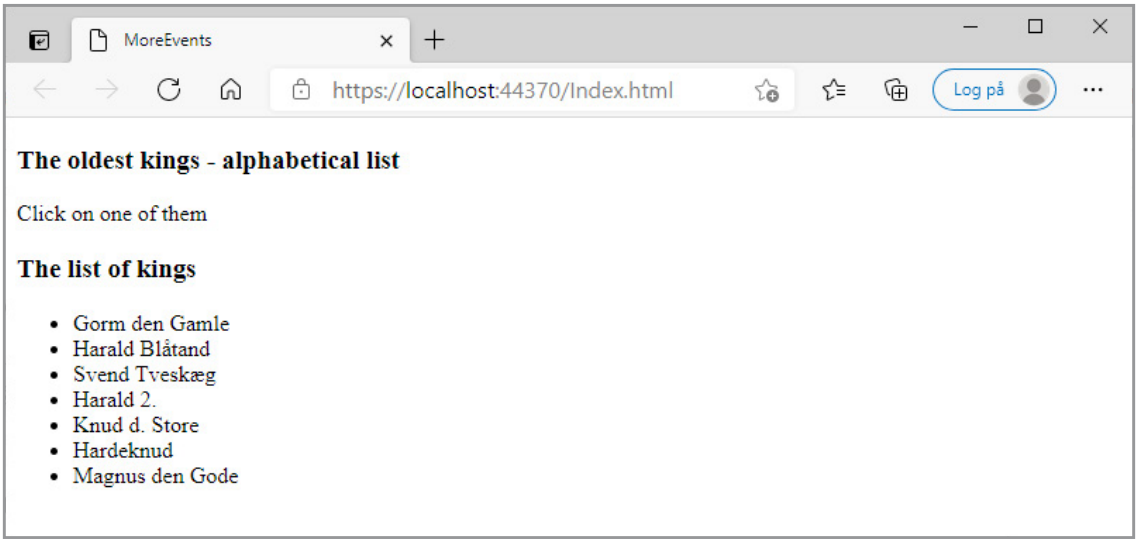
When an event handler is performed, an event object is transferred to the handler. Consider the following code as example (*MoreEvents*):

```
<!DOCTYPE html>
<html>
  <head>
    <title>MoreEvents</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
  </head>
  <body>
    <h3>The oldest kings - alphabetical list</h3>
    <p>Click on one of them</p>
    <ul id="list1">
      <li>Gorm den Gamle</li>
      <li>Harald 2.</li>
      <li>Harald Blåtand</li>
      <li>Hardeknud</li>
      <li>Knud d. Store</li>
      <li>Magnus den Gode</li>
      <li>Svend Tveskæg</li>
    </ul>
    <h3>The list of kings</h3>
    <ul class="list2"></ul>
    <script>
      var list = document.querySelector(".list2");
      var kings = document.querySelectorAll("li");
      for (var i = 0; i < kings.length; i++)
      {
        kings[i].addEventListener("click",
          function (e) { list.appendChild(e.target); }, false);
      }
    </script>
  </body>
</html>
```

If the code appears in the browser, the result is:



The list with *id list1* has 7 names arranged alphabetically. Next, an empty list is identified by a *class* attribute. The following script block sets a reference to the empty list using the *class* attribute and then a reference to an array of all *li* elements. The last loop associates an anonymous event handler to these *li* elements. This handler uses the event object - here called *e* - to move the element that is clicked to the empty list. Below is the result after clicking on all names (in the correct order) in the top list:



The most important thing about the example is how to transfer a parameter to the event handlers. It is an object called *event* and, as illustrated below, it is an advanced object with many properties and methods.

Properties of the *event* object are:

- *event.clientX* and *event.clientY* indicates in the case of a mouse event the coordinates for the mouse relative to the browser window.
- *event.offsetX* and *event.offsetY* indicates in the case of a mouse event the coordinates for the mouse relative to the element that has raised the event.
- *event.keyCode* indicates in case of a key event code for key pressed.
- *event.target* is a pointer to the node in the DOM tree, which has raised the event.
- *event.currentTarget* is a pointer to the node in the DOM tree, which is bubbled or captured
- *event.eventPhase* indicates the event phase as 1 for capture, 2 for target, 3 for bubbling.
- *event.type* indicates event type such as click, key press, etc.
- *event.relatedTarget* indicates for some events (for example *mouseout*) the element that originally raised the event.
- *event.stopPropagation()* is a method that can be called to stop an event propagation up or down in the DOM tree, but other registered event handlers are still performing.
- *event.stopImmediatePropagation()* is a method that as *event.stopPropagation()*, stops an event propagation up or down in the DOM tree, but it does not performs other registered event handlers.
- *event.preventDefault()* is a method that stops a default event handling if possible.

Looking at the above code nothing happens when you point to the individual elements in the top list. This can be solved by assigning an event handler for *mouseover* and *mouseout*. I have added the following event handlers:

```
function blue()
{
  this.setAttribute("style", "color:blue;");
}
function black()
{
  this.setAttribute("style", "color:black;");
}
function moveKing(e)
{
  e.target.removeEventListener("mouseover", blue, false);
  e.target.removeEventListener("mouseout", black, false);
  e.target.setAttribute("style", "color:black;");
  list.appendChild(e.target);
}
```

The first two handlers must be used for respectively *mouseover* and *mouseout* for the elements in the top list. You should note that the latter has a parameter that will be a reference to an event object. The event handler must be used to move an element. When an element is moved, it will no longer be highlighted when the mouse points to it. Therefore, the two event handlers for *mouseover* and *mouseout* are removed, and you should note that *e.target* refers to the element that has raised the event. In addition, the color must be set to black, and then the element is moved to the bottom list in the same way as above.

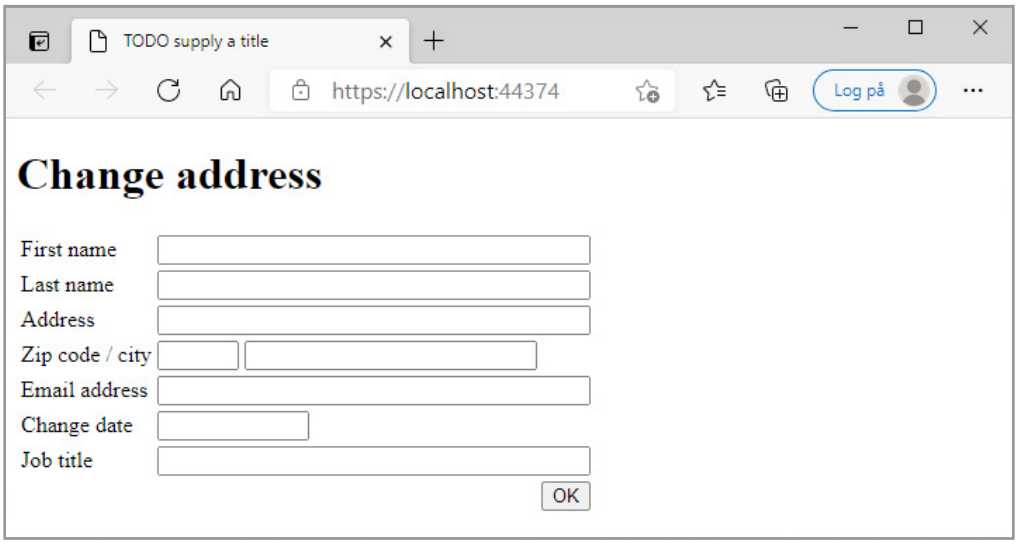
The HTML code is the same as above, including the script that associates event handlers:

```
<script>
  var list = document.querySelector(".list2");
  document.getElementById("list1").addEventListener("click", moveKing, false);
  var kings = document.querySelectorAll("li");
  for (var i = 0; i < kings.length; i++)
  {
    kings[i].addEventListener("mouseover", blue, false);
    kings[i].addEventListener("mouseout", black, false);
  }
</script>
```

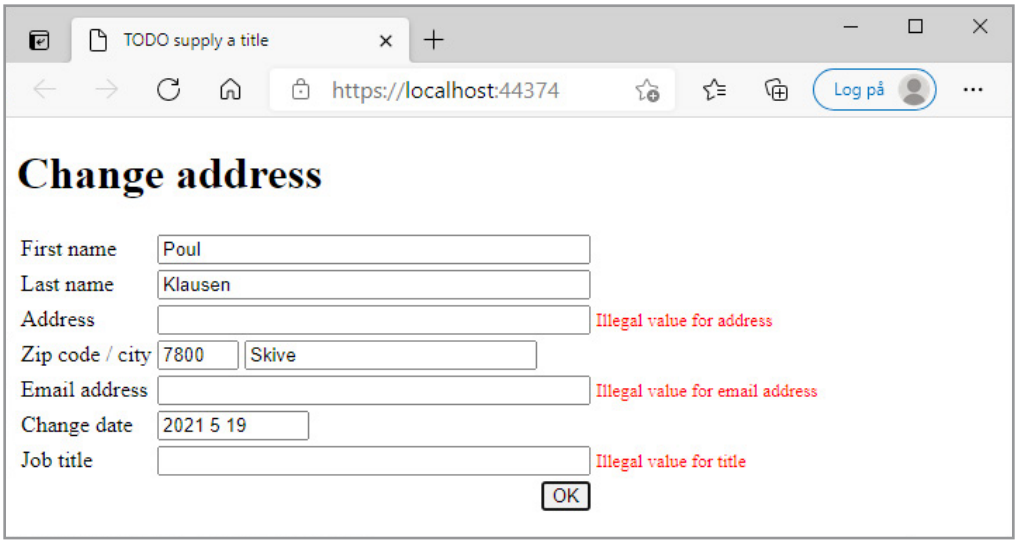
Here you should first note that the event handler *moveKing()* for click events is assigned to the list itself and not to the individual list elements. Here are used that events bubbles up and clicking on a list element, the event then bubbles up to the element's parent, which is the *ul* element, that has attached an event handler, and in the handler, *e.target* will refer to the list element that have fired the event.

PROBLEM 2

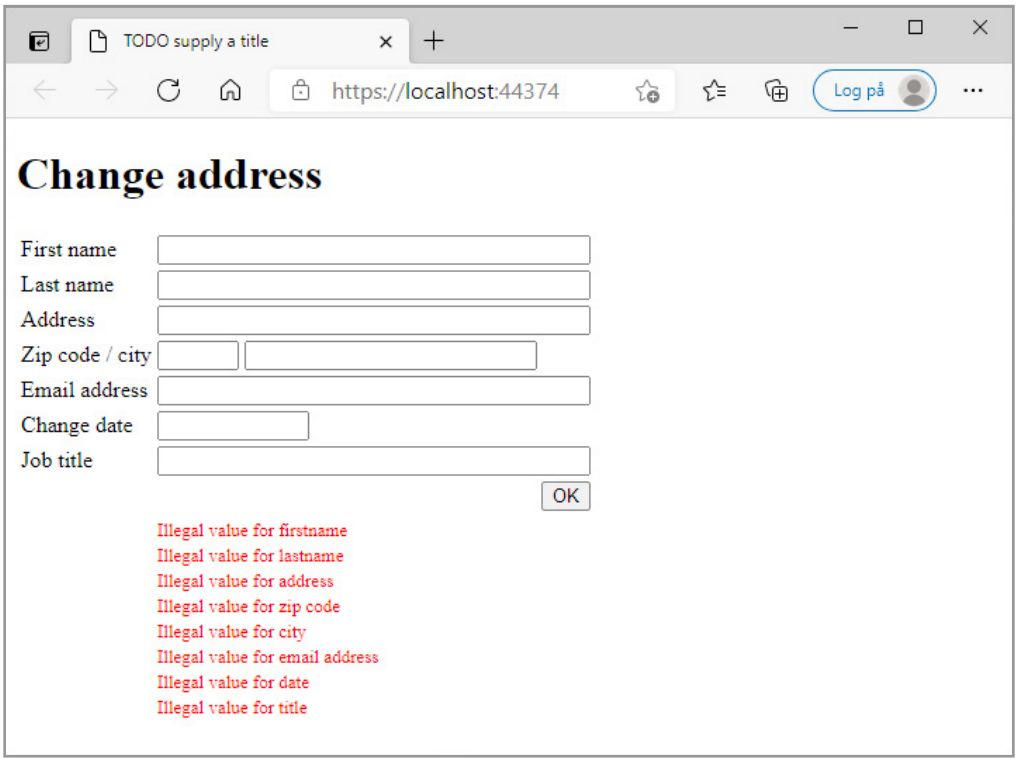
Create a new project that you can call *ChangeAddress*. The project should only have one page *Index.html*, and it should basic be a form with component arranged in a table:



When the user enters data, they should be validated, and the validation must take place every time a field lost focus. Al fields must be validated and for first name, last name, address, city and title the only requirement is that the fields must not be empty. Zip code must be 4 digits while email address and date must be a legal email address and a legal date, respectively. All validation must happen in JavaScript on he client side, and below is a window where has entered values for first name, last name, zip code, city name and date:



And below a window where the OK button is clicked without data entered:



To validate an email address you should use a regular expression. It is part of the task to find out how and how the syntax is. To validate a date, you should use the JavaScript library from exercise 4, and it must be possible to enter a date in the same format as described in this exercise. The button should be a submit button, and you can with an *onsubmit* event for the *form* element test if a click on the button must result in a submit (it should not be if not all fields are filled out correctly).

If all fields are correctly filled and clicking on the button, nothing else than an submit must be done with the result that all fields are blank.

5 THE JAVASCRIPT WORLD

The previous chapter is an introduction to JavaScript that today plays a crucial role in the development of web applications. As mentioned and probably also to some extent illustrated above, JavaScript today is both efficient and flexible and in terms of what is possible when programming the client side, there are virtually no limits.

However, there are a few challenges with JavaScript. Firstly, there are many details and especially regarding modifying the DOM tree, and secondly, the fact that the language is interpreted can make it difficult to test and correct JavaScript code. For these reasons, several (many) alternative products have been developed, all of which can be considered as superstructures or additions to JavaScript. In this chapter, I will mention a few and very important of these products, just to point out what it is all about. These are technologies that you will most likely encounter if you work as a web developer, but the following is by no means an introduction to the application of the technologies, but merely a hint of what it is.

I will to some extent explain what jQuery is, while the other products are pretty much just mentioned by name. All the mentioned products are supported by Visual Studio, but for practical use they must be learned, which of course requires effort and a book could be written about each of the products without any problems, but fortunately there are many on the market, so it is easy to find tutorials there which explains how the products are used.

5.1 JQUERY

jQuery is a large library of JavaScript functions, and if you work as a web developer, you cannot avoid jQuery and, if nothing else, jQuery provides so many functions that it is worth the effort to investigate what it is. So, therefore, a brief introduction, but also because I have actually used jQuery in the final example in the last chapter.

First, you have to grab the product and that is the latest version, as new versions are constantly being added. You can go to the page

```
https://jquery.com/download/
```

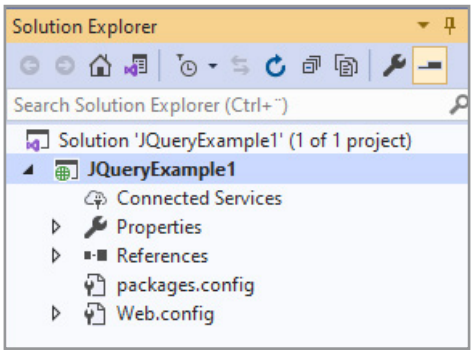
from where you can download the product as a JavaScript file, and thus as a plain text file with the code (right click on the link and select *Save link as*). In fact, you download two files:

1. *jquery-3.2.1.js*
2. *jquery-3.2.1.min.js*

where the version number can of course differ. Both files contains the same JavaScript code, and the difference is, that the latter is compressed, so it is significantly smaller (about one third). The goal is that while developing, you can use the first one where the code is written in readable form and with comments, and when the program is finished, you can replace it with the last one. Here you must remember that JavaScript is text that is sent along with the HTML document, and you are therefore for the sake of bandwidth interested in sending as few data as possible.

For a web project in Visual Studio you can also use *Add* | *Client-side Library* to add jQuery to your project.

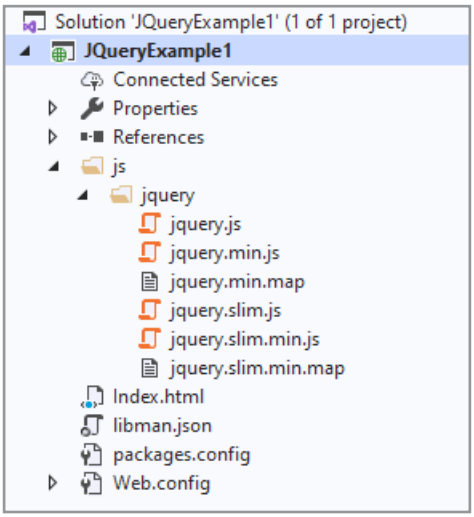
As example I create a new empty *ASP.NET Web Application (.NET Framework)* project called *JQueryExample1*:



Next I and a HTML page called *Index.html* and a folder called *js*. On this folder I right click an select

Add | Client-side Library

and add jQuery to my project:



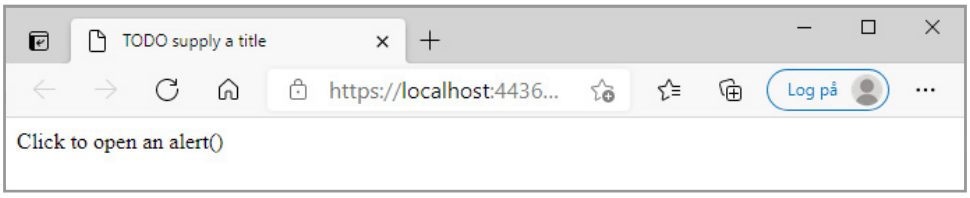
The content of *index.html* is as follows:

```
<!DOCTYPE html>
<html>
  <head>
    <title>TODO supply a title</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <script src="js/jquery/jquery.js"></script>
    <script>
      $(document).ready(function() {
        $("div").click(function() {
          alert("Hello world!");
        });
      });
    </script>
  </head>
  <body>
    <div>Click to open an alert()</div>
  </body>
</html>
```

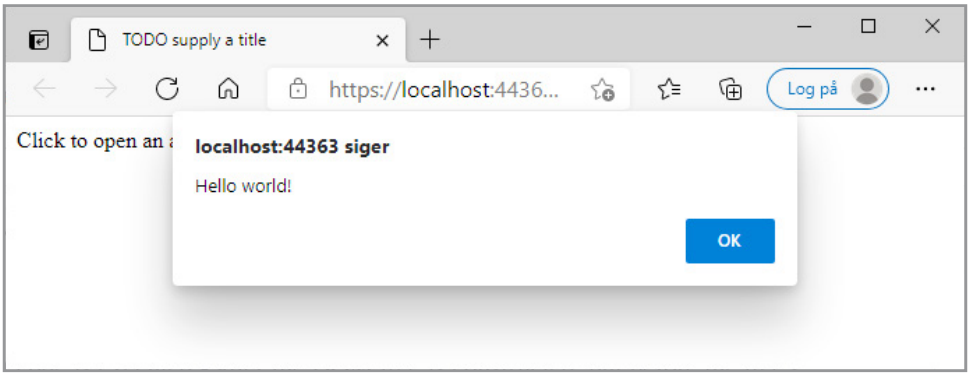
First, note that in the header there is a script element that refers to the jQuery file. In addition, there is a script element with some JavaScript code that uses jQuery. Here means

```
$(document).ready
```

that the following JavaScript code is executed - after the DOM tree is constructed, but before the tree's elements are rendered by the browser. In this case, it means performing an anonymous function that has one statement that associates a *click* event handler with an *alert()* to all *div* elements. Specifically, note the syntax *$()*, which is a jQuery function, which refers to a collection of elements in the DOM tree, and you can then perform a JavaScript function on these items. In this case, the *body* part has only a single *div* element and if you opens the document in the browser, the result is:



What is not very mysterious, but if you click on the text, you get a popup:



The example shows two things. First of all, what is technically necessary to use jQuery, and second, what is the idea of jQuery, that is, in an easy way to select items in the DOM tree and do something with them. The first goes quite easily and you simply copy the library to the project (in fact there is an option since multiple browser vendors make the library available and you can just link to the current library). As for the other, it all builds on the *$()* function, which is just an alias for *jQuery()*, and the above script block could be written as follows:

```
<script>
  jQuery(document).ready(function() {
    jQuery("div").click(function() {
      alert("Hello World!");
    });
  });
</script>
```

Viewed from the program, it does not matter what you write, but it is standard to write *$()*, what can be recommended. The use of jQuery is so widespread that most developers perceives *$()* as jQuery code.

A whole different thing is what you can use as an argument for the function *$()* and which functions you can specify should work on the elements (*click()* is an example) and here there are simply many options, a lot more than can be accommodated in this book so it is necessary to read the documentation:

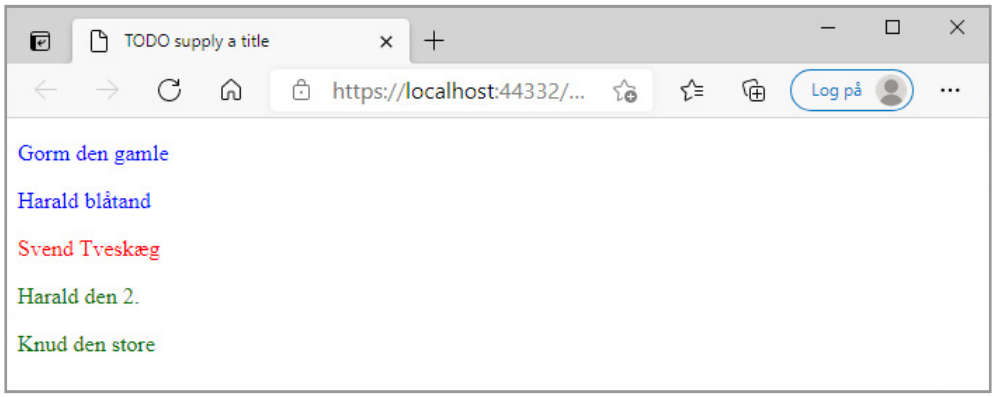
```
http://www.tutorialspoint.com/jquery/jquery_tutorial.pdf
```

which provides an adequate and easily readable documentation. I would like to justify, with a few examples of what you can with jQuery.

Generally, the parameter of *$()* may be any selector corresponding to what is discussed in chapter 2 on style sheets. Consider, for example, the following HTML document (*JQueryExample2*):

```
<!DOCTYPE html>
<html>
  <head>
    <title>TODO supply a title</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <script src="js/jquery/jquery.js"></script>
    <script>
      $(document).ready(function() {
        var e = $("p");
        for (var i = 0; i < e.length; ++i) e[i].style.color = 'darkgreen';
        var c = $(".first");
        for (var i = 0; i < c.length; ++i) c[i].style.color = 'blue';
        var v = $("#viking");
        for (var i = 0; i < v.length; ++i) v[i].style.color = 'red';
      });
    </script>
  </head>
  <body>
    <div>
      <p class="first">Gorm den gamle</p>
      <p class="first">Harald blåtand</p>
      <p id="viking" class="first">Svend Tveskæg</p>
      <p>Harald den 2.</p>
      <p>Knud den store</p>
    </div>
  </body>
</html>
```

The *body* part is simple and consists of 5 paragraph elements in a *div* element. You should note that the first three defines a class attribute (although no equivalent css class is defined in any place), and the third element has an ID. The JavaScript code (the jQuery function *$().ready()*) performs a function that starts by selecting all paragraph elements, and the result is a collection of 5 elements. The next *for* loop iterates over these elements and turns the color to green. Next, the same is done where the color is set to blue, but only for the elements whose class is *first*. Finally, the color is set to red, but this time only for those elements that have a specific *id* (and there is only one). If the document is opened in the browser, you get the result:



In particular, note that the elements in the DOM tree are processed by the JavaScript code before the document appears in the browser.

The above is easy enough to understand, but it can be written easier using a jQuery function:

```
<!DOCTYPE html>
<html>
  <head>
    <title>TODO supply a title</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <script src="js/jquery/jquery.js"></script>
    <script>
      $(document).ready(function() {
        $("*").css("font-size", "24pt");
        $("p").css("color", "darkgreen");
        $(".first").css("color", "blue");
        $("#viking").css("color", "red");
        $(".last, #viking").css("font-size", "36pt");
      });
    </script>
  </head>
  <body>
    <div>
      <p class="first">Gorm den gamle</p>
      <p class="first">Harald blåtand</p>
      <p id="viking" class="first">Svend Tveskæg</p>
      <p class="last">Harald den 2.</p>
      <p class="last">Knud den store</p>
    </div>
  </body>
</html>
```

Note that the last two paragraph elements this time has a class *last*. If you consider the first select in the JavaScript function, it selects all elements and sets the font size to 24 points, but it is done using the function *css()*, which is a jQuery function that works on all elements that are selected. The same applies to the next three statements, which also use the function *css()* instead of writing the necessary loops. This is where you meet the strength of jQuery, and there are simply a large number of functions that you can read all about in the documentation mentioned above. In particular, note the last statement that sets the font to 36 points for all elements that either have id *viking* or class *last*.

As another example (*JQueryExample4*), a document similar to the above is shown below, but where the example shows the use of multiple functions:

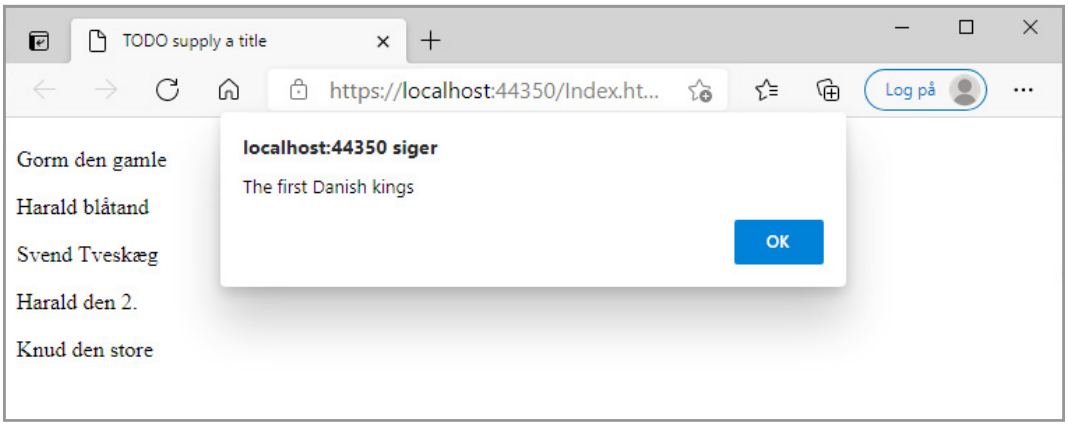
```
<!DOCTYPE html>
<html>
  <head>
    <title>TODO supply a title</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <script src="js/jquery/jquery.js"></script>
    <style>
      .red-class {
        color: red;
      }
      .blue-class {
        color: blue;
      }
      .green-class {
        color: darkgreen;
      }
    </style>
    <script>
      $(document).ready(function() {
        var text = $("p").attr("title");
        $("h1").text(text + "s");
        $("#viking").attr("title", "Last Danish viking King")
        $(".first").addClass("red-class");
        $(".last").addClass("blue-class");
        $("#viking").click(function () {
          $(this).toggleClass("green-class");
        });
```

```
        alert($("h1").html());
        $("h2").html("but there are more Kings and Queens")
      });
    </script>
  </head>
  <body>
    <div>
      <h1></h1>
      <p class="first" title="The first Danish king">Gorm den gamle</p>
      <p class="first">Harald blåtand</p>
      <p id="viking" class="first">Svend Tveskæg</p>
      <p class="last">Harald den 2.</p>
      <p class="last">Knud den store</p>
      <h2></h2>
    </div>
  </body>
</html>
```

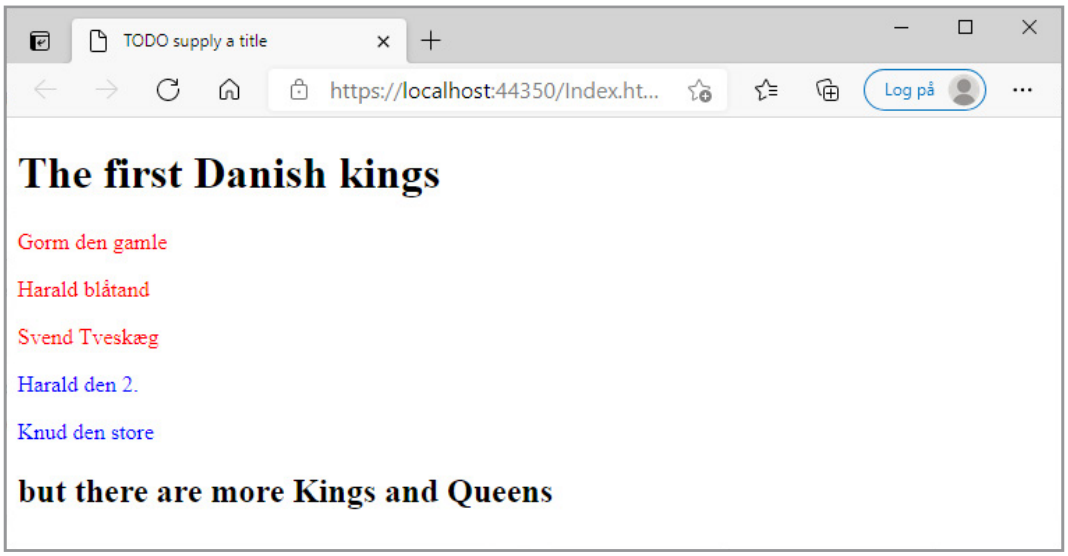
Note first that the *body* part is essentially the same as in the previous example but it is expanded with a *h1* element and a *h2* element, both of which are empty. Also note that a *title* attribute has been added to the first paragraph element. Then note that three simple styles are defined with each their class name.

Then the jQuery block (the function). First, the value of the *title* attribute is saved in a variable. This is done by selecting all paragraphs and then using the *attr()* function. It returns the value of *title* for the first element, that has a *title* attribute, and in this case there is only one element (which is the first). The value of the variable is then used to associate a text with the *h1* element, which occurs with the function *text()*. The third statement defines a *title* attribute of the element with id *viking*, which again occurs with the *attr()* function in an override, which specifies the attribute as well as the value. Then is attached a class *red-class* to all elements whose class is *first*. Please note that an element may have more values for the class - an element can be of more classes. The next statement performs the same for all elements whose class is *last*, but this time the class *blue-class* is added.

As a next step is added an event handler for click events to the element with id *viking*. The result is that if you click on the element, the color will change from red to green and click again, the color will change back to red. The next last statement shows an *alert()* with the value of *innerHtml* for the element *h1* while the last initializes the *h2* element. If you open the document, you first get the following window with a popup:

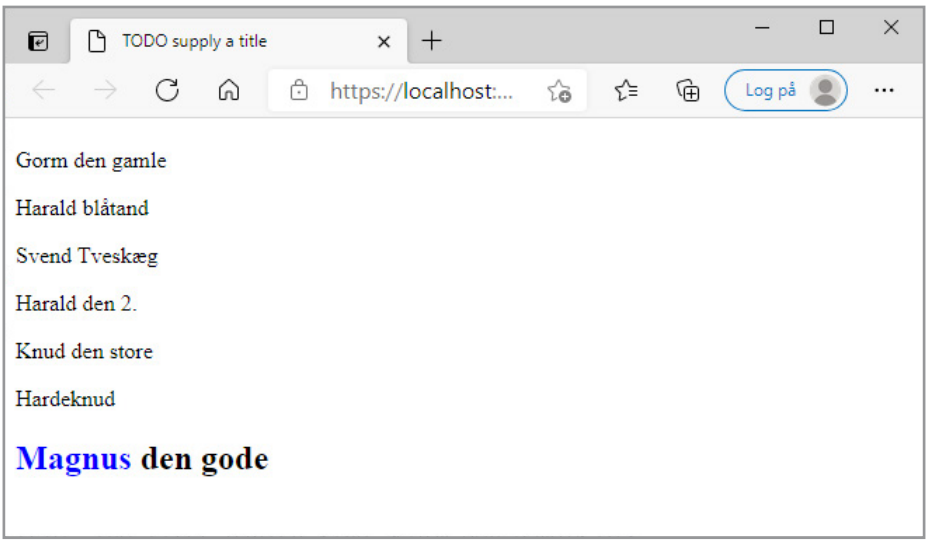


and when you click *OK*:



Finally, I will show an example that modifies the DOM tree (*JQueryExample5*):

```
<!DOCTYPE html>
<html>
  <head>
    <title>TODO supply a title</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <script src="js/jquery/jquery.js"></script>
    <script>
      jQuery(document).ready(function() {
        $("div").append("<p id='norwegian'>Magnus</p>");
        $("#norwegian").before("<p>Hardeknud</p>");
        var text = $("#norwegian").text();
        $("#norwegian").replaceWith("<h2><span>" + text + "</span> den
        gode</h2>");
        $("span").css("color", "blue");
      });
    </script>
  </head>
  <body>
    <div>
      <h1></h1>
      <p class="first" title="The first Danish king">Gorm den gamle</p>
      <p class="first">Harald blåtand</p>
      <p id="viking" class="first">Svend Tveskæg</p>
      <p class="last">Harald den 2.</p>
      <p class="last">Knud den store</p>
    </div>
  </body>
</html>
```



As for the *body* part, there is nothing new compared to the previous element, only that the *h2* element is removed. The jQuery function starts by adding a *p* element to the *div* element. This happens with the function *append()*, which adds the element as the last child to the *div* element. The next statement uses the function *before()* to add another *p* element, but this time an element that is also a child to the *div* element, but in front of the element just inserted (the element with the value *Magnus*). The third statement stores the value of the element with id *norwegian* in a variable, and the next statement replaces the element with id *norwegian* with another element (an *h2* element containing a *span* element). As the last changes the color of the element is changed to blue.

The above is only a hint of what is possible with jQuery and you are encouraged to investigate what else is. In particular, I would like to mention that the library also contains functions that use ajax (see the next section). This means that you using JavaScript can communicate asynchronously with the server, which gives a whole new dimension regarding the use of JavaScript.

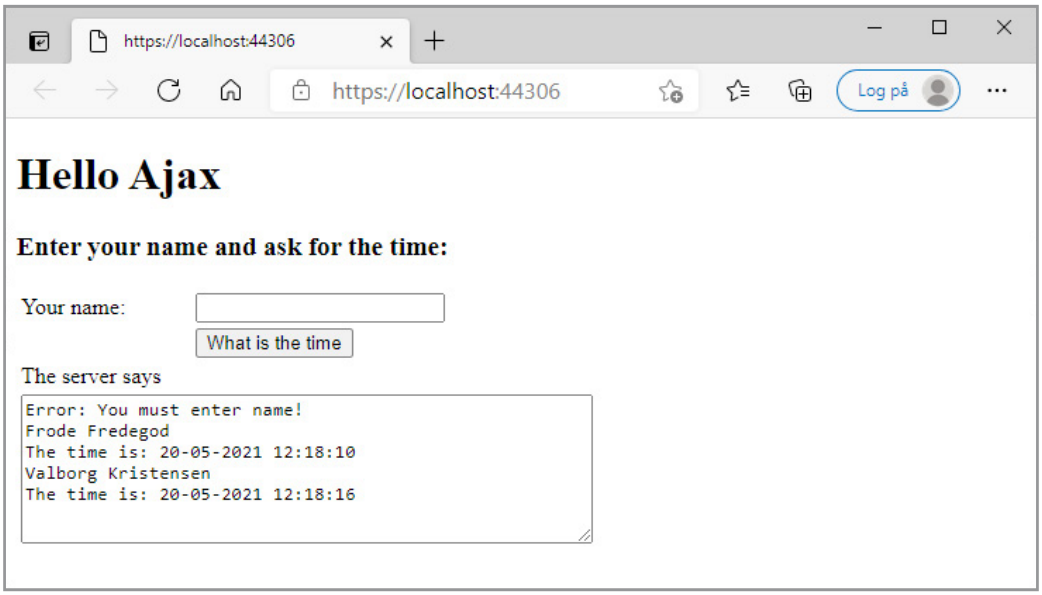
In addition to what is mentioned above, there are also more extensions of jQuery, and in particular I want to mention *jQuery UI*, which can be downloaded from:

```
https://jqueryui.com/
```

It is an extension that defines a number of elements for the user interface such as dialogs and the ability to select elements using the mouse - and more. When you download the product, you will get a JavaScript file (*jquery-ui.js*) and an associated style sheet (*jquery-ui.css*), and a project that illustrates the application. I will use the product in the example in the next chapter.

5.2 AJAX

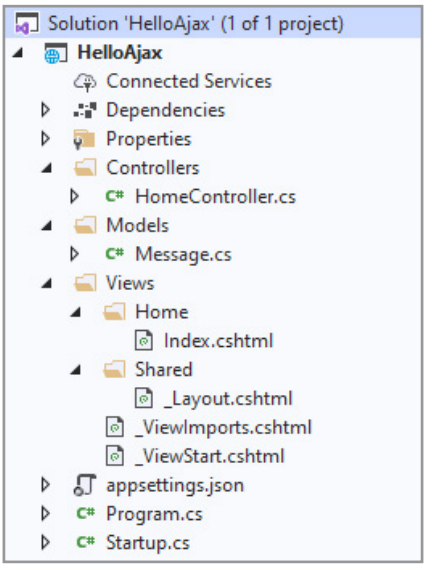
As next, I want to show how you using JavaScript can send data to the server, but without reloading the page. The benefits are not so big, but it gives a quieter window, as it all happens asynchronously, and the user thus, to a lesser extent, notes that data being sent to the server, and since the entire document do not has be rendered again. I will show you how using a simple example:



When the user enter a name and clicks on the button the name is validated client-side using JavaScript, and the name is empty an error message is inserted in the text area (the first line above). However, if the user has entered a name the name is send to the server which return a response consisting of the name and the time. The response is added to the text area as two lines, but it all happens without a reload of the page, and what is modified on the client-side is the content of the of the text area and the input field is cleared.

The technique is called *ajax* and is based on JavaScript and is part of jQuery and actually a jQuery function.

The application is a MVC application consisting of a default controller, a default view and a simple model:



The model only has two properties:

```
public class Message
{
  public string Name { get; set; }
  public string Date { get; set; }
}
```

The class is used to represents the data which the server should send as a response. Next you should note the layout file:

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>@ViewBag.Title</title>
  <script type="text/javascript" src=
    "https://ajax.googleapis.com/ajax/libs/jquery/1.8.3/jquery.min.js">
  </script>
  <script>
  </script>
</head>
<body>
  <h1>Hello Ajax</h1>
  <div>
    @RenderBody()
  </div>
</body>
</html>
```

Ajax build on jQuery, but instead of added the library to the project as a client-side library I this time add a reference to Google site from where to download the library. There is not much difference, but it can be useful to know that it is also a possibility.

The controller has two action methods:

```
public class HomeController : Controller
{
  public IActionResult Index()
  {
    return View();
  }

  [HttpPost]
  public JsonResult Ask(string name)
  {
    Message message = new Message { Name = name, Date = DateTime.Now.
    ToString() };
    return Json(message);
  }
}
```

The first is the default action method, while the other is an action method for a HTTP POST. Note the return type which tells that the method must return its response as JSON. The method has a parameter which is the name the user has entered, and the method creates a model object initialized by this value as well as the current time. The object is then serialized as JSON and send as response.

As the next there is the view as where it all takes place:

```
<script>
  function updateResult(text) {
    var field = document.getElementById("txtText");
    field.value = field.value + text + "\n";
  }

  function buttonClicked(e) {
    var name = document.getElementById("txtName").value;
    if (name == null || name.length == 0) {
      updateResult("Error: You must enter name!");
    }
    else {
      $.ajax({
        type: "POST",
        url: "/Home/Ask",
        data: { "name": $("#txtName").val() },
        success: function (response) {
          updateResult(response.Name + "\nThe time is: " + response.Date);
          document.getElementById("txtName").value = "";
        },
```

```
        failure: function (response) {
          updateResult(response.responseText);
        },
        error: function (response) {
          updateResult(response.responseText);
        }
      });
    }
  }

  $(document).ready(function () {
    document.getElementById("cmdSend").
      addEventListener("mousedown", buttonClicked);
    });
</script>

<h2>Enter your name and ask for the time:</h2>
```

The view starts with a script block and is followed by a html table which defines the content of the view. There is nothing to note, but it is the script that uses ajax. First note the call of the jQuery function which is an event handler to the button. The handler starts to extract the name from the input field, and if the name is empty the handler calls a function *updateResult()* with an error message. The function add the message to the text area field, and you should note that it all happens on the client side. If the user has entered a name the handler calls an jQuery function *ajax()* initialized with more parameters:

1. the type of the command and here it should be a HTTP POST
2. the action method to be performed
3. the data send to the server encoded as JSON
4. a callback method as a JavaScript function with the response from the server
5. two other callback methods used if there is an error

The callback method for success is an anonymous method which call *updateResult()* with the response. Here you must note the model object from server is automatically JSON deserialized and you can reference it as an object.

What is back is *Startup* where you have to configure a service for JSON serializing:

```
public class Startup
{
  public void ConfigureServices(IServiceCollection services)
  {
    services.AddControllersWithViews().AddJsonOptions(
      options => options.JsonSerializerOptions.PropertyNamingPolicy = null);
  }

  public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
  {
    if (env.IsDevelopment())
    {
      app.UseDeveloperExceptionPage();
    }
    app.UseStaticFiles();
    app.UseRouting();
    app.UseEndpoints(endpoints =>
    {
      endpoints.MapDefaultControllerRoute();
    });
  }
}
```

PROBLEM 3

In this exercise you must write another version of the program written in *problem 2*. The program should open the same window and test the field values in the same way, but with three differences:

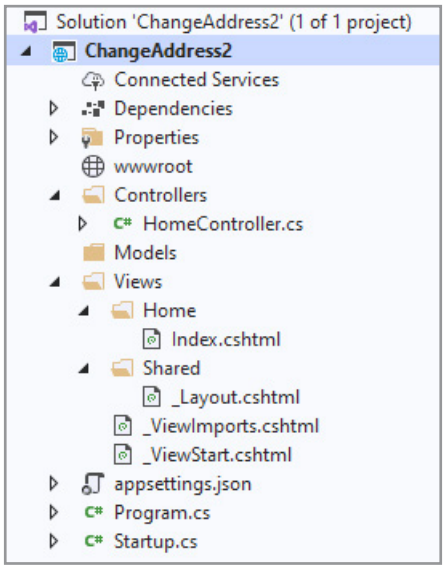
1. The application must be an ASP.NET Core MVC program.
2. When the user has entered an address it must be stored in a database.
3. The program must use ajax to send data to the server.

As database you should use a LocalDb database and the entity framework.

The code from program 2 can be reused (and should), but I think you should start from scratch and follow the guidelines below.

1) Create the project

Create a new ASP.NET Core project as you can call *ChangeAddress2*. Create the standard folders as well as standard files for a MVC application:



Update *Startup* such you have an running program (which only show an empty browser window).

2) Create the model

Add a model class called *Person* for an address when the class must have a property for each value the user must enter (there are 8) and the type of all properties must be *string*. The class must also have a property (the first property) of the type *long* called *PersonId* which should be used for the primary key in the database.

Update *_ViewImports.cshtml* with a reference for the model:

```
@using ChangeAddress2.Models
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

Use NuGet to add two references for the entity framework (you can see how in chapter 3).

Add a connection string for a LocalDb database to *appsettings.json* when you can call the database *Addresses*.

Add a context class called *AddressesDbContext* and a repository called *IAddressesRepository* and *EFAddressesRepository* to the *Models* folder. You can see in chapter 3 how to write this types (and they should look the same as this the database also has only one table).

Add the following class to the model which should be used to create the database:

```
public class CreateDb
{
  public static void EnsurePopulated(IApplicationBuilder app)
  {
    AddressesDbContext context = app.ApplicationServices.CreateScope().
      ServiceProvider.GetRequiredService<AddressesDbContext>();
    if (context.Database.GetPendingMigrations().Any())
    {
      context.Database.Migrate();
    }
  }
}
```

Update *Startup.cs* such the program is ready for entity framework and create a database:

```
public class Startup
{
  public Startup(IConfiguration config)
  {
    Configuration = config;
  }

  private IConfiguration Configuration { get; set; }

  public void ConfigureServices(IServiceCollection services)
  {
    services.AddControllersWithViews();
    services.AddDbContext<AddressesDbContext>(opts =>
{ opts.UseSqlServer(Configuration["ConnectionStrings:AddressesConnection"]); });
    services.AddScoped<IAddressesRepository, EFAddressesRepository>();
    services.AddScoped<AddressesDbContext, AddressesDbContext>();
  }

  public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
  {
    if (env.IsDevelopment())
    {
      app.UseDeveloperExceptionPage();
    }
    app.UseStaticFiles();
    app.UseRouting();
    app.UseEndpoints(endpoints =>
    {
      endpoints.MapDefaultControllerRoute();
    });
    CreateDb.EnsurePopulated(app);
  }
}
```

Make sure you use the correct name for your connection string.

Build the project without running the program.

Open a *Developer PowerShell* prompt and creates a migrate for the database (see possible chapter 3 for how to do).

Now run the program and it should run without errors, but still alone with a blank browser window.

You can open Microsoft SQL Server Management Studio to ensure that the database is created with one table.

3) Add the user interface

Create a folder *css* and *js* as subfolders in *wwwroot*. Add the stylesheet and JavaScript files from the project for problem 2 to this directories. Add references for stylesheets and JavaScript files to the start of *Index.cshtml*. Then add the code for the form from problem 2 to *Index.cshtml*.

Then the program should run again and work in the same way as in problem 2.

4) Store data in the database

Add a constructor to the controller for the database context and the repository.

Add an action method for a HTTP POST with a *Person* object as parameter. The method must store the object in the database and return a *JsonResult* as response telling where the object is stored.

The JavaScript used to validate the form must call the above action method using ajax with the form data as parameter.

Update the class *Startup* for JSON serializing.

The it should all works and you can store addresses in the database. Remember you can use Microsoft SQL Server Management Studio to check that the addresses is actually stored in the database.

5.3 TYPESCRIPT

TypeScript can be characterized as a language based on JavaScript and with the same syntax as JavaScript. Basically it is a language which translates to JavaScript, and you can use all you know from JavaScript, but it expands with new features which means that programming in JavaScript happens much in the same way as programming in C#. Programming in JavaScript has some challenges where the biggest is that it is difficult to debug and correct JavaScript code. JavaScript gives the programmer many degrees of freedom which gives both advantages and disadvantages and one of the disadvantages is that it is difficult to write the code correctly. This is exactly what TypeScript seeks to solve and with the prevalence that JavaScript has today, there is good reason to be interested in TypeScript.

5.4 ANGULAR

Angular is a development platform, built on *TypeScript*. The platform is a component-based framework for building complex web applications that consists of a collection of integrated libraries that cover a wide variety of features. Angular is supported by Visual Studio. Angular is designed to make updating as easy as possible, so you can take advantage of the latest developments with a minimum of effort.

5.5 REACT

React is another framework for developing web applications and is often considered an alternative to Angular, as it is in many ways the same you want to achieve, but otherwise the two products are very different. React is a JavaScript library and does not use TypeScript, and one can therefore quite rightly perceive React as an alternative to jQuery, but there are also significant differences here. jQuery is a library of JavaScript functions that can be used in HTML pages. React is also a library with JavaScript functions, but it is instead a library with functions that make it possible to embed HTML.

5.6 NODE

As the last I will mentioned *Node* which is a cross-platform runtime environment that executes JavaScript code on a server or a desktop machine. The interesting thing about node is that it also makes it possible to use JavaScript on the server side.

5.7 JSON

Above and also in previous books I have used JSON. It has nothing directly to do with programming, but it is an encoding of data which plays an increasing role and has done so for many years and especially it is widespread in connection with web applications. For that reason, I will add to this book a few remarks on what JSON is.

JSON is a standard for data exchange primarily between client and server in web applications, but in principle, JSON can be used in many other contexts and will be too. In relation to data exchange, JSON can to some extent be seen as an alternative to XML. JSON is characterized by being simple and easy to write and read (there are very few and simple rules), being text based and being platform independent, but compared to XML one can also say that it is encoding of text for programmers..

A JSON document has in principle the same syntax as an object in JavaScript, and an example could be:

```
{
  "danish": "Danish kings",
  "kings": [

    {
      "name": "Gorm den Gamle",
      "to": 958
    },
    {
      "name": "Harald Blåtand",
      "from": 958,
      "to": 987
    },
    {
      "name": "Svend Tveskæg",
      "from": 987,
      "to": 1014
    }
  ]
}
```

The basic syntax is that data are represented as key / value pair separated by colon. Several data elements (key / value pairs) can be gathered as objects in a collection where the elements are separated by commas. Finally, the value of a data element may be an array. The above example defines a data structure with two elements where the first is a key / value pair and the other is an array with three elements. Each element in the array is an object where the first has two elements and the two others three elements.

In general, JSON supports the following data types:

1. *Number* as in JavaScript
2. *String* that is double-quoted Unicode with backslash as escaping
3. *Boolean* that is *true* or *false*
4. *Array* which is an ordered sequence of values in square brackets separated by comma
5. *Object* that is an unordered collection of *key : value* pairs

With regard to escaping of characters in strings, the same symbols are used as in C# and JavaScript. Put a little differently, a JSON data structure has the same syntax as a JavaScript object consisting solely of properties.

As another example, is shown a JSON structure that defines a 2-dimensional array:

```
numbers = {
  "primes" : [
    [2, 3, 5, 7],
    [11, 13, 17, 19],
    [23, 29],
    [31, 37],
    [41, 43, 47]
  ]
};
```

As an example, the following program (*JsonProgram*) prints the above JSON data structures:

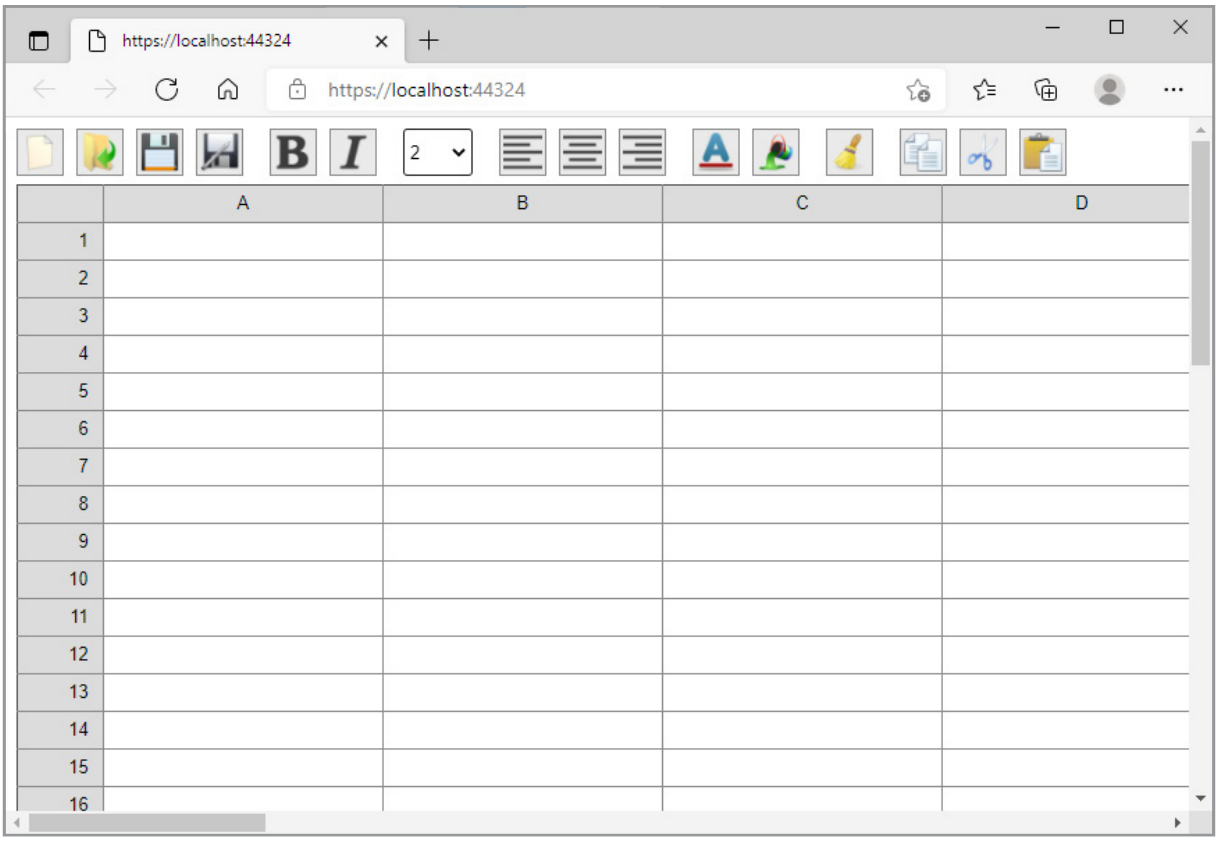
```
<!DOCTYPE html>
<html>
<head>
  <title>TODO supply a title</title>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <script>
    danish = {
        "kings": [
        {
          "name": "Gorn den Gamle",
          "to": 958
        },
        {
          "name": "Harald Blåtand",
          "from" :958,
          "to": 987
        },
        {
          "name": "Svend Tverskæg",
          "from": 987,
          "to": 1014
        }
        ]
      };
      numbers = {
        "primes" : [
          [2, 3, 5, 7],
          [11, 13, 17, 19],
          [23, 29],
          [31, 37],
          [41, 43, 47]
        ]
      };</script>
</head>
<body>
  <h1>Danish Kings</h1>
  <script>for (var i = 0; i < danish.kings.length; ++i)
      {
        var king = danish.kings[i];
        document.write(king.name);
```

```
        document.write("<br/>");
        if ("" + king.from !== 'undefined' && "" + king.to !== 'undefined')
          document.write("From: " + king.from + " to: " + king.to);
        else if ("" + king.from !== 'undefined')
          document.write("From: " + king.from);
        else if ("" + king.to !== 'undefined') document.write("To: " +
        king.to);
        document.write("<br/>");
        document.write("<br/>");
      }
  </script>
  <script>for (var i = 0; i < numbers.primes.length; ++i)
      {
        for (var j = 0; j < numbers.primes[i].length; ++j)
          document.write(numbers.primes[i][j] + " ");
      }
      document.write("<br/>");
  </script>
</body>
</html>
```

As long as you only have to manipulate JSON data on the client page, there is actually not much else to tell than what appears from the above, but if data has to be sent to or received from a server, they must be parsed to be used. For example, if you want to send JSON data from the client to an action method on the server side, and afterwards it should be able to use that data, they must be parsed or decoded to a Java object, using standard parsers that can be downloaded and used by the server code. Similarly, on the server side, you can encode beans as JSON data before they are sent to the client, and on the client side, correspondingly, you must download JavaScript parsers that can decode the data sent from the server. I do not want to review in this place, but I can mention that it is quite simple and requires only some browsing on the Internet.

6 WEBSHEET

As the final example of this book, I will write a web application, which is a simple spreadsheet. The application is called *WebSheet*, and if you open it in the browser, you get the following window:



A spreadsheet requires a high user interaction, where the user must be able to enter numbers and formulas in the individual cells, and in the case of a formula, the expression must be evaluated. The client has to do a lot, and the goal of the project is to show how it can be implemented in JavaScript, thus showing something about what is possible in JavaScript. To facilitate the work, I have used jQuery, which is a JavaScript library that provides a wide range of features available, including specially finished elements for the user interface such as dialog boxes, context menus, and more.

As mentioned above, the goal is to show how to use JavaScript to program the client side of a web application, but it is not a fully functional spreadsheet. Note that the program can be used, but there is a lot that is missing or things that could be better.

6.1 THE PROGRAM'S FUNCTIONS

To enter in a cell, first the cell must be opened that occurs as *CTRL* + *SHIFT* and click with the mouse. Then you can enter a number, a formula or a text. The entry is accepted either by pressing *Tab* or *Enter*.

You can select one or more cells using the mouse and proceed in the usual way by holding down the left button and drag the mouse, and the CTRL and SHIFT keys have the usual functionality.

The program has a toolbar with 16 features, as mentioned from the left has the following functions:

1. Create a new spreadsheet and enter the number of rows (default 50) and the number of columns (default 20).
2. Open an existing spreadsheet where you can search among the spreadsheets that have been created and saved.
3. Save the spreadsheet that you work on - if it has been saved. Otherwise, the *Saveas* function is called.
4. Save the spreadsheet as a new spreadsheet, which means entering a name.
5. Sets the text in the selected cells to bold.
6. Sets the text in the selected cells to italic.
7. Sets the number of decimals for the selected cells.
8. Sets the text in the selected cells to left adjusted.
9. Sets the text in the selected cells to centered.
10. Sets the text in the selected cells to right adjusted.
11. Sets the text color for the selected cells.
12. Sets the background color for the selected cells.
13. Deletes all formatting for the selected cells or for the entire spreadsheet, if no cells are selected.
14. Copies the selected cells.
15. Deletes the contents of the selected cells and stores the content.
16. Inserts cells that are saved or deleted.

In addition, right-clicking on a row header you get a context menu with four functions:

1. Select all cells in the row.
2. Insert an empty row above the current row.
3. Insert an empty row under the current row.
4. Remove the row.

The same goes for columns:

1. Select all cells in the column.
2. Insert an empty column before the current column.
3. Insert an empty column after the current column.
4. Remove the column.

6.2 DESIGN

The content of a spreadsheet must be saved and that have to be done on the server side. The content must thus be sent to and from the server. To facilitate this transport, a spreadsheet is represented as a JSON object:

```
var datadefs = (function() {
  public = {};
  public.values = [];
  public.options = {
    bold: [],
    italic: [],
    align: [],
    background: [],
    foreground: [],
    decimals: []
  };

  return public;
})();
```

values is an array with the spreadsheet's data elements (one element for each data cell). That is, the array does not contain values for row and column headers. The array has one element for each row, where each element is an array with data elements of the form

```
{
  value:
  type:
}
```

Here *value* is the cell's value, and *type* can be *n* for number, *s* for string and *e* for expression. As an example, the data presentation for a blank spreadsheet with 3 rows and 2 columns is shown below.

```
{
  "values":[
    [{"value":"","type":"s"},{"value":"","type":"s"}],
    [{"value":"","type":"s"},{"value":"","type":"s"}],
    [{"value":"","type":"s"},{"value":"","type":"s"}]
  ],
  "options":
    {
      "bold": [],
      "italic": [],
      "align": [],
      "background": [],
      "foreground": [],
      "decimals": []
    }
}
```

The object *options* is used for formatting, and *bold* and *italic* contains objects of the form

```
{
  row:
  col:
}
```

that defines a cell in the table. *align* contains objects of the form:

```
{
  row:
  col:
  align:
}
```

where the value of the last attribute is *left*, *center* or *right*. The two arrays *background* and *foreground* contains objects of the form

```
{
  row:
  col:
  val:
}
```

where the last attribute is a color value. Finally, the last array *decimals* contains objects of the form:

```
{
  row:
  col:
  dec:
}
```

where the last attribute is the number of decimals.

A spreadsheet must be saved somewhere, and a database is created using the following model class:

```
public class Sheet
{
  public long SheetId { get; set; }
  public string Name { get; set; }
  public DateTime Created { get; set; }
  public DateTime Modified { get; set; }
  public int Rows { get; set; }
  public int Cols { get; set; }

  [Column(TypeName = "ntext")]
  public string Content { get; set; }
}
```

It is a simple database with only one table. The names tells you what the individual columns are to be used for, but you should especially note the two columns *rows* and *cols*, which contains the number of rows and the number of columns respectively. Finally, note the last column *content* that is for the spreadsheet's content.

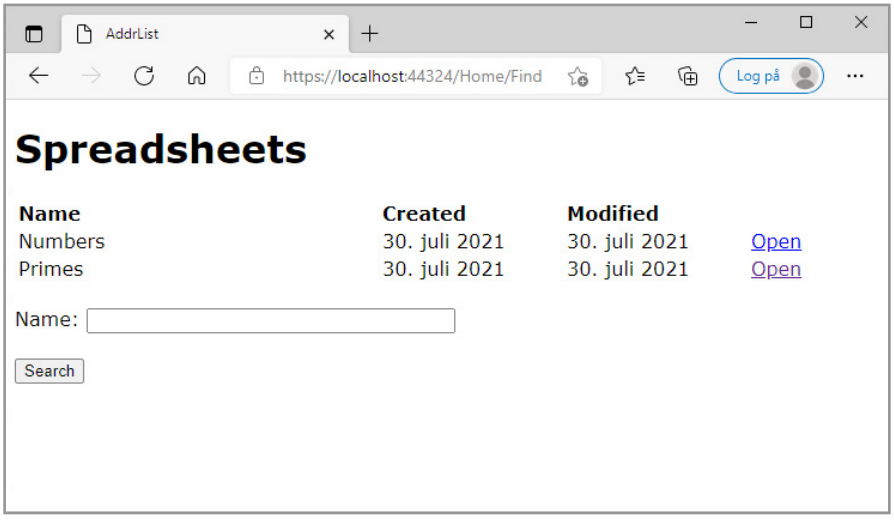
6.3 PROGRAMMING

The project is MVC application and is in principle a simple application with only one controller and two views. The project uses jQuery and libraries for jQuery are found under *wwwroot*. Here you also finds a simple stylesheet called *styles* and three JavaScript files mentioned below.

The *Model* folder has several model classes. The one is called *SheetName* and is a class that represents 4 columns from the database table (*id*, *name*, *created* and *modified*). The important thing is that it does not represent the content of a spreadsheet as it can fill a part. The class is used when searching for spreadsheets. Another model class is called *SheetModel* and is derived from *SheetName* and expands with the last three columns. The class is used to represent the spreadsheet you are working on. The database is a LocalDb database and is created using the Entity Framework. As mentioned above the class *Sheet* defines the table and the *Model* folder has the usual classes to create the database if it is not there.

The subfolder *ViewModels* has five classes where two are models for the two views while the last three are used in the view *Index* to dynamic build the spreadsheet. Objects of the classes are created in *IndexViewModel*.

The *Search* view is a simple view used to open a saved spreadsheet:



The rest of the code is client code, primarily as JavaScript and style sheets. Regarding the latter, there are 4 style sheets, the two being part of jQuery and are not modified. The other two are relatively simple and are used by *Index.xhtml* (*styles.css*) and for dialogs respectively. There are 8 JavaScript files, but only the 3 are written for this project. The four belongs to jQuery, while the fifth is a standard parser for JSON.

The application largely uses jQuery and in addition to the ability to easily select elements in the DOM tree, jQuery are used for the following:

- to select cells in the table using the mouse, for example, by dragging the mouse
- to open a dialog, either as a simple message box or where the user can enter data
- to a context menu that opens if the user right-clicking on either a row or column header
- to open a simple color selector where the user can choose a color for either text or background

It requires that several files relating to jQuery are downloaded and added to the project. It should be noted that jQuery offers far more than what is used in this example.

For the sake of the current project, 3 JavaScript files have been added:

1. Expressions.js
2. Websheet.js
3. ColorSelector.js

The first contains a single module *expression*, which includes all the functions necessary for formulas. In the book C# 3, in the final example, I have shown a class that has methods that can parse and evaluate a mathematical expression. The module *expression* is primarily a converting of this class to JavaScript, but with two changes:

1. *expression* supports only two mathematical functions, namely the square root function and a sum function. However, also the power function is supported by implementing the operator ^.
2. Instead of common variables, an expression works on cell references, which means that the parser must be changed a bit.

For the *sum* function, it may sum up a number of values (constants or cell references), which are indicated as a comma separated list. You can also specify two cell references in the same row or column and separated by semicolon, and the function summarizes all values between these references.

The implementation of expressions (formulas) is thus relatively simple as one can directly apply the algorithms from C# 3. However, the challenges relate to three situations:

1. When inserting rows or columns, expression's references to cells may change. For example, if you insert a row, all references in expressions that are larger than the index for the new row are counted by 1. The same applies to columns.
2. By copying / pasting formulas, the formula's cell references must be changed in relation to how far a formula is moved.
3. Formulas can refer to other formulas via cell references, and thus there is the possibility of circular references. This problem is partially solved by the fact that after each entry of a formula, the program checks whether the formula results in a circular reference.

Some of the module's methods are used to handle these situations.

Several of the methods may raise an exception and, if it happens, the exception action is an *alert()*. For a finished application, of course, it makes no sense, but I have maintained them, as they may be useful if others want to work on the project. In fact, it is relatively difficult to troubleshoot JavaScript code, and one of the difficulties with JavaScript is that there are very many possibilities for errors, and it is difficult to test the code for errors. In an operational situation, you could therefore replace the relevant alert's with a request to the server and, using ajax, send the appropriate error messages to the server that could save them in a log file or a database table. This allows you to check on a regular basis whether the code results in errors, what has caused the error, that can make the maintenance of the program considerably easier.

Then there is the JavaScript file *websheet.js*, which contains all the script that will be used to handle the user interaction. The file is extensive and consists of three modules:

1. *datadefs*, which is the data presentation of the current spreadsheet
2. *sheet,* containing all methods to manipulate the spreadsheet and then all functions used from the toolbar
3. *handlers,* that contain event handlers to edit the content of cells

The first I mentioned during the design and the last one is relatively simple. In order to edit the content of a cell, a cell in the table contains two components, one being a *div* element with a *span* element, which generally shows the text. It is the content of this element that is formatted according to the selected formats. The other element is an input component that is generally invisible. Holding down the CTRL and SHIFT keys and clicking the mouse will change the visibility of these two components, and you can edit the content. When

you click again or enter *Tab* or *Enter* switches on visibility again and the user interface is updated. The latter is not quite simple. First, it may mean that formulas elsewhere in the spreadsheet must be updated if they depend on the cell that has been changed. Secondly, in case it is a formula that is entered, it must be validated that it not results in a circular reference. These are things that are handled by functions in the *handlers* module.

There is also a function *initCells()* that is not part of the above modules. The task of the function is to associate the event handlers in the module *Handlers* to the table's cells. This function is used when the window is initialized by the browser.

An important part of the module *sheet* relates to formatting, in which you must be able to format cells, ensure that the formatting are saved and used again when a saved spreadsheet is loaded. In addition, formatting may be modified if you insert / delete rows and columns. You should be aware that formatting is not copied in connection with copy / paste.

Another part concerns the toolbar's buttons to open and save spreadsheets, where requests are sent to the server. In particular, be aware that a spreadsheet is saved, asynchronously, using ajax.

Finally, there are the buttons to copy / paste, and especially the event handler for paste is not simple. The functions do not use the machine's clipboard, but instead an internal array. The reason is to make it simple, but you should note that there is actually an API so that you from JavaScript can use the system clipboard.

Then there is the file *ColorSelector.js*, which makes nothing but defining a simple object for color selection. Only a few colors can be chosen, but jQuery offers a more advanced color selection dialog that would be a good option. When I have chosen to write my own, it is alone to make the program simple.

6.4 CONCLUSION

As mentioned, the purpose of this project is to show a little about what is possible with JavaScript, and in fact, it is possible to program complex logic that is performed on the client side alone. JavaScript is in fact a fairly complex programming language, and there is a lot more to learn than what this book is about. However, it is not more difficult to learn JavaScript than another programming language, and if you need to use JavaScript to a large extent, there is nothing more than going systematically, for example, reading a book exclusively related to JavaScript. However, there is one thing that should be noted that since JavaScript is not translated by a compiler, and is not type-strong, there is a high risk

that the code contains errors and, worse, it is both difficult and time consuming to find the mistakes. The conclusion is, therefore, that JavaScript is a fully-fledged programming language, but it is far from an effective language for developing major programs. If you need to write larger programs in JavaScript, you should be interested in some of the technologies mentioned in the previous chapter.

The current project is not a complete spreadsheet, which can be used as an alternative to, for example, *Excel* - nor does it make sense to try to develop such an alternative. However, the project may have applications, and if you think of a web application that needs a spreadsheet-like view, the project or parts of it may be used. Should the project be expanded, so it looks like a real spreadsheet, there are some missing and I will mention some features that could significantly improve the program. In fact, each feature could be a task or project in itself.

1. Data entry should be improved as you do not always have to click the mouse to open a cell. It is primarily a question about to handle events related to the keyboard, such as arrow keys.
2. The program should apply the system clipboard so that you can copy data to other programs. It is primarily a matter of getting into the corresponding JavaScript API.
3. The expression module should be expanded with more mathematical functions. It's actually a relatively simple task, and the algorithms can be found in the book C# 3.
4. A spreadsheet is stored in a database which may not be the best solution. Alternatively, a spreadsheet could be saved as a text file in the file system, which in many contexts would be preferable.
5. However, the biggest lack is performance where the application becomes ineffective on large spreadsheets. An option to solve the problem could be to show only a part of a spreadsheet on the client side and then place some navigation buttons in the toolbar. This improvement is not quite simple.