Poul Klausen

C# 11

WWW The server side

Software Development

bookboon
LEARNING

POUL KLAUSEN

C# 11: WWW THE SERVER SIDE

SOFTWARE DEVELOPMENT

C# 11: WWW The server side: Software Development
1st edition

ISBN 978-87-403-3785-3

CONTENTS

FOREWORD

This book is the eleventh 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 deals with the development of web applications where the focus is on the server side and how to develop dynamic web pages. The book starts with an introduction to world wide web and the main technologies, followed by a simple example on ASP.NET web application. The rest of the book deals with how to write an ASP.NET Core MVC application, and after reading the book, you should be able to write classic web applications. However, the book contains little about the client side, which is dealt with first in the next book.

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.

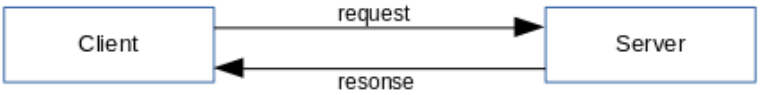
Finally a little about what the books are not. It is not “a how to write” or for that matter a reference manuals to C#, but it is as the title says books on software development. It is my hope that the reader when reading the books and through the many examples can find inspiration for how to write good 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

Everything that in the previous books has been said about system development, programming and C# has been aimed at applications running on a single computer - also called desktop applications. In the following books, I will treat the development of applications that run on multiple machines and exchange data over a network, which usual is the Internet, and in fact, such programs are the most common in practice - at least if you look at programs used in companies or public institutions. Besides running the programs on a network, they are characterized by having many concurrent users and are typically included in an IT solution that includes several programs that work together to solve the desired tasks. Development of such programs requires new technical solutions and, on the other hand, they make new demands for the system development process itself, but fortunately everything so far been said is still valid, but it is necessary to expand with new concepts.

In the literature, such programs are generally referred to as enterprise applications, and without it being precisely defined what it is, it covers programs for large companies and organizations, but perhaps you should think about it in that way that a program is not just a program, but a solution to a work situation or task in a larger company. I will start by looking at the development of web applications, which are applications that run over the Internet. It is not the only kind of enterprise applications, and even maybe it's not even sharp anymore what are web application and what are not. At least it is a talk of some technology and applications that are developed and run in a way other than traditional office programs, and in addition, they are programs that we all meet, whether it's at work or privately.

Web applications have been around for many years and have, as a result of a long development, been more and more widely distributed from simple static websites to actual applications such as web stores and applications such as Google's office programs, but while there's been much happening, the principle is still the same with a server and a client:



The client is often an ordinary workstation with a browser. A browser is a program similar to any other PC program, but using the browser, the user can enter a *request* to a web server by entering a web address in the browser's address bar. A web server is also a program that runs on a machine somewhere. It is a program that is constantly running and listening for a request. A *request* means that the client's browser wants to get a HTML document send back and the web server will then load the document and send it back to the client's

browser as a *response*. Once the client gets the document, the browser will rendering it and display it on the screen. This means that the browser interprets the HTML code and based on the code determines how the document should appear on the screen.

It is at least the basic principle, and as it works in the beginning with *World Wide Web*, but today there is a lot more. A request regarding not only to send an HTML document back, but often data is sent along with the request (that could be data about items to be added to a shopping cart) and it may also be information to the server to find specific data (for example from search criteria for goods). This means that the server has to do a lot of other things like, for example, to save data in databases, retrieve data from a database and dynamically build the response to be sent to the client. In other words, on basis of the client's requests, the web server must perform software, programs that are basically developed and executed like any other program, but only programs that run on the machine hosting the web server.

A web application is thus characterized by, that the program's code being executed on the server, which then sends the result back to the client in the form of a dynamically generated HTML document. However, a *client-side* program can also contains code that is executed by the browser. There are several options, but the most important is *JavaScript*, which is script code (which is just text) that is send as part of the HTML document. Of course, it requires that the browser is able to interpret and execute the *JavaScript* code, but all modern browsers are. Thus, a web application can include code executed on the server side, and code executed on the client side, but such that the largest part of the code is server code. Therefore are web applications called for *client-server* applications.

As described above, a web application builds on network traffic, and every time a user makes a request, there is a delay before getting a response. Web applications therefore run differently than a traditional desktop application. Nevertheless, many of the tasks that previously were solved by programs running on the specific client machine, has been taken over by web applications. There are several reasons for that. The most important thing is the obvious advantage that you can access data regardless of where you are located, and it is not necessary to install the program on the user's machine. Then there is the delay that draws in the other direction, but it does not mean the same as before and, among other things because that today we have a much higher rate on the Internet than before, but also because we have used to the fact that web applications works that way. Finally, technically, a number of actions has been taken to ensure that the delay is not felt so much primarily by ensuring that no more data is sent than necessary. One could say that it has always been the goal of getting a web application to appear to the user, as it was a usual program installed on the user's computer.

The goal of this book is to show how to develop web applications in C#. However, it requires a modest knowledge of HTML that I do not want to touch on, but although it should not be in place, you will probably be able to follow the development of the examples anyway, as it is relatively small what I use, and also the development tools (Visual Studio) provides great help.

Finally, it should be mentioned that web applications are not the solution to everything and there are still programs that should be developed as classical PC applications, primarily because they cannot live with the above delay. Now, a common PC application can also communicate with a server over the Internet and thus be part of an enterprise application, but it requires another technology that I return to in subsequent books. It should also be mentioned that handheld computers such as mobile phones today play a very important role and again set new requirements for the programmer, a topic that is dealt with in later books.

1.1 A LITTLE ABOUT THE TECHNOLOGY

World Wide Web (WWW) has been here for a long time (since 1989) and in all that time it is developed from a technology to download and show HTML documents in a browser to a technology to run applications in the users browser, applications that are hosted on a web server somewhere. It has been a long way, and on that route a lot of technologies and development tools has been developed. At the beginning we talked about web-pages or home-pages, but today we talk about web-applications, applications we all use every day. In the following I will mention some of the most important technologies, but there are many others.

From start WWW was a client / server technology, where the client and the server are connected on the Internet. The server is called a web-server and is a program running as a service on a machine somewhere. The service is listening for requests on a port, and this port is as default port 80. Microsoft's web server is called *Internet Information Server* (and short IIS). The client was the user's machine with a web-browser, and when the user wants to look at webpage, the user must enter the address of the machine for web-server as well as the name of the webpage. The web-server must then find and load the page on the server's hard disk and send the page as a response to the client. For this to works, there are at least three things, that need to be in place:

1. A way to find the address of the web-server's machine on the internet
2. A standard for how to send a request and the same for the server's response and that is how the client and the server should communicate
3. A standard for the format of the webpage such the user's browser can understand and read the page and show it correct om the screen

To solve the first problem every machine on the internet has a unique address called an IP address. I shall not go into details about IP addresses in this book and how these numbers are assigned. For users it is impossible to remember these addresses, and it is all complicated by the fact that many machines often get their IP address changed. To solve these problems the Internet uses Name Servers, which is a family of servers on the Internet, whose task is to find IP addresses from a name. In fact, it is a complicated system and it is the task of a current internet provider to use the system to find an IP from a name of a web server.

If two people has to communicate, they need a language they both understand. For computers to communicate over a network it is the same. The computers send messages to each other, and it means the computers must communicate after some rules such they understand each other. Such rules are called a protocol, and a protocol defines which commands a computer (such as a browser on a client machine) can send to the receiver (such as a web-server). The protocol used for World Wide Web is called HTTP and is a simple text based protocol, which defines the syntax for commands used on WWW. In this book I will not say much about HTTP, as it is rare you as a programmer directly refer to the protocol.

The format of a webpage is HTML, which is a markup language. A web page contains text and references to other elements as well as HTML codes that tells how the browser should show the document. The following will not include a complete presentation of HTML, but I will use HTML in examples, and here I will explain HTML to the extent that it is necessary.

After introduction of webpages on the Internet, the next step was the use of dynamic webpages. It means that the user as usual sends a request to the web-server, but not for a static HTML page, but instead a request where the web-server executes a program that create the page. This technic opens up a lot of new possibilities, as the program on the server could do all what a program can do. In particular, the program can read and update a database table. This in combination with the possibility of using HTTP to send data as a part of a server request make it possible to the client to send data to a web-server and let the server perform some operations on these data and send the result back as a dynamic generated response. This form for client / server communication is the basic principle of web applications both before and since. However, a lot has happened since the first web applications came into the world, but as mentioned above it has all been about getting a web application to behave as if it were a windows application running on the user's machine. The challenges have primarily concerned three issues:

1. Response time where the user in the browser enter data for a request, send them to the server and then has to wait for the server's response.
2. All code in a web-application are performed on the server, and every action, which requires execution of program code, must take place on the web server.

3. The HTTP protocol is stateless, which means when the client send a request it requires a connection to the server over the internet and after the server has send its response, the connection is disconnected and the server has forgotten all about who the client is.

The first problem is basically solved using faster Internet and the future will undoubtedly result in an even faster Internet. However, it does not solve the basic problem that a web application is based on a client / server technique, and good solutions therefore require more than just a fast Internet. Two of the most important measures for this are firstly to move part of the program logic to the browser, and that is to let the browser execute part of the program code and secondly to reduce the data that is send between browser and server. Another technique is to avoid updating the entire browser window for each request, but only the necessary part of the window, a technique called AJAX, which means that we as users do not experience the request-wait-response problem to the same degree.

In a web application it is in principle the web-server that must execute the program's code, but some operations, such as validating users' entries, can advantageously be performed on the browser. It requires that the browser is able to execute code, and here one has tried two ways. One is to directly send a translated program as part of the server's response, and then equip the browser with an interpreter for that program. Probably the most common technology has been the use of Java Applets. However, sending a translated program code to the browser presents a security issue as the code may contain anything including malicious code. Therefore, that solution is today perceived as obsolete and should be avoided. The prevalent technology today is instead to send script code, which is just text, and if the browser is equipped with an interpreter for such script code, one can to some extent, achieve the same. It requires that you agree on a script language and today you use JavaScript, which is a language that all browsers can interpret. The use of JavaScript is very widespread today and is actually part of every web application. This is mainly due to the fact, that the language has been significantly developed and is effective both in terms of development, but also in terms of the browser's execution of a JavaScript program. The next book contains an introduction to JavaScript.

When a web application build on a client / server technology it is a problem, that the HTTP protocol is stateless and the server does not remember the clients requests. To solve this problem web-servers use a technology called sessions, where the server generate and send a unique id as part of its response. The browser can then send this id back together with a new request. It make it possible for the server to store data (objects) for a specific user identified by the session id.

One thing is the basic technology. Another thing is how to, then in practice to develop web applications, and that is what this book is about. Here, too, there has been a long development, and I will not here describe the historical development, but just mention that with the introduction of .NET, ASP.NET was also developed as a framework for the development of web applications using, among other things, C#. In fact, developing web applications is far from simple, and along the way ASP.NET has evolved, very primarily with the aim of making development of web applications simpler and more stable. There has also been a focus on program architecture with the aim of developing programs that are easier to maintain, but also architectures that lead to more robust solutions. Today we are talking about ASP.NET MVC, and although it does not sound like much other than yet another application of the MVC pattern, it is actually a radical change in how to develop and structure a web application. In the following, I will use this framework everywhere, quite simply as an appropriate and efficient architecture for a web application, and all of it is of course supported by the fact, that ASP.NET MVC (ASP.NET Core) is supported by Visual Studio.

1.2 ABOUT HTTP

Before I go on, I will add a few remarks about the HTTP protocol. A web application, as already described, consists of a number of files (web pages) located on a web server. These files, that are webpages, images, etc. are all used by clients, who download the files, and display them in a browser. That is a client is a regular PC with a browser while a web server is a computer (a server) that runs a program (a service) called a web server. The web server constantly listens on a port (for example 80) after requests from clients, and in case of a request (a request on a particular page), the web server sends a response as a HTML document to the address to which the client's request relates. Specifically, the client enters the address of a web page in his browser, for example.

www.torus.dk

and the browser will then convert the name to an IP address using the DNS system, after which the browser opens a socket to the server via a port (the default is 80). Then the web server can reply back with a response in the form of HTML, which the browser can then interpret and display as a web page.

From the server, more things can happen. In simplest cases a client request regarding a static HTML page, and if so, the server should do nothing but load the page from the machine's disk and send it to the client's browser. Today, however, it will often be a dynamic web page, for example a request to a web application and client's request regarding then

in reality a program. The web server loads that application and executes it. Often it will include database operations either because the application requires data from a database or also because the client has sent data with the request, which must be stored in a database. In any case, the program that the web server executes will generate HTML (whose content may be dynamically based on the content of a database) and send this HTML to the client.

In order for this client / server architecture to work, agreements or rules that exactly determine the format of the client requests and server responses are required, and this is where HTTP enters the picture as the name of that protocol (a protocol is a set of rules), which indicates how the communication between client and server should be.

HTTP stands for *HyperText Transfer Protocol* and is a protocol belonging to the application layer, and typically implemented over TCP / IP. It is a stateless protocol that, as mentioned, is based on request and response, where a client application sends a request to a server and where the server responds to the client with a response. That the protocol is stateless means that after the server has sent the response, it has forgotten everything about the client. The server thus stores no information about its clients, and whenever a client requests the server, it is perceived by the server as a new request. That the HTTP protocol is thus stateless, presents many challenges in connection with web applications, and there are several techniques to cope with these challenges, where I have already mentioned session objects, but also cookies can be used. The current version of HTTP is called version 1.1 or it is HTTP/2 which is the newest version of the protocol and is today supported by most browsers.

A URL is a reference to a particular resource (file) on the Internet and generally has the following format:

http://Servernavn/Filenavn

An example of an URL could be

```
http://www.eadania.dk/education/it/index.xhtml
```

Here www.eadania.dk is the server name, while *education/it/index.xhtml* is the name of the file on the server relative to a directory determined by the server. Exactly an URL format is

http://Servernavn:port/Filenavn

where port is the *port number* used by the web server, but if nothing is specified, the browser will use port 80, which is standard on the Internet.

In HTTP, a client request and response from a server consist of three parts:

- request- or response line
- a header
- data

A web-transaction consist of

- a client request
- a server response

and it is initiated by the client by establishing a connection to the server at a predetermined port. Default is, as mentioned, port 80. The client then sends an HTTP command followed by the document address and the version of the protocol that is used. It could, for example, be

GET /index.htm HTTP/1.1

Next the header is send as a number of lines of the form

Keyword: Value

where each line ends with a carriage return and a line feed. As an example, it could be

```
User-Agent:Lynx/2.4 libwww/5.1k
Accept:image/gif, image/x-xbitmap, image/jpeg, */*
```

After the last header line, a blank line is send, after which it may follow data to be send to the server and it all ends with another blank line.

A command associated with a request is also called for a method, and there are seven methods:

Method	Description
OPTIONS	Used to ask a server about what options it offers.
GET	Asks the server to return the document that the document address specifies.
HEAD	In principle, works as GET, but the server does not return the actual content of the document. The command is used to test if the document has been changed since the last request.
POST	Used to send a data block to the server along with a request.
PUT	It is the opposite command to GET and stores the data block on the document address.
DELETE	Deletes the document on the server that the document address specifies.
TRACE	Used to track a request's route through different firewalls and proxy servers and used in connection with debugging of complex network issues.

As mentioned, a response from a server also consists of three parts. The first line has the format

Protocol Statuscode Description

and as an example it could be

```
HTTP/1.1 200 OK
```

Then, the server's header contains information about the server and the document that is being send. The header consists of lines according to the same pattern as a request, and it ends in the same way with a blank line. After the header comes the data block, which may be a document, an output from a program or possibly an error message.

A connection is not terminated, after the server has completed its response. The reason is that many HTML documents contain references to other files such as images, etc., and the client may therefore request such resources without having to re-establish the connection.

The HTTP protocol defines many headers, but the most important is the following where Q stands for request while S is for response:

Header	Q	S	Description
Accept:	x		Specifies what types the client will accept.
Accept-Charset:	x		Specifies which character sets the browser can accept.
Accept-Encoding:	x		Specifies what types of encodings the client knows. If this header is omitted, the client will accept all encodings.
Accept-Language:	x		Specifies which languages the client accepts.
Age:		x	Used in conjunction with cache control.
Allow:		x	Specifies what methods the resource on the document address can respond to.
Authorization:	x		Used in conjunction with digital signatures.
Cache-Control:	x	x	Used by proxy servers and describes how to handle request and response.
Code:	x		Defines an encoding of the data block. Standard is Base64.
Content-Base:		x	Used to solve relative URLs in the returned document. This header overrides Content-Location.
Content-Encodning:		x	Indicates an encoding applied to the document before it is transmitted.
Content-Language:		x	Indicates the natural language for the content.
Content-Length:		x	The length of the data block in bytes.
Content-Location:		x	Specifies the location of the document being send.
Content-MD5:		x	Used for a checksum for the data block.
Content-Type:		x	Specifies the type of data being send.
Expires:		x	Specifies a date for data to be perceived as obsolete.
From:	x		The client's email address.
Host:	x		The host's name from the URL.
Last-Modified:		x	Specifies where the document sent last has been changed.
Location:		x	Used for redirect to another address for example in case of an error.
Referrer:	x		The source for the current request.
User-Agent:	x		The browser's signature that is used to test which browser a request is coming from.
Warning:		x	Used for additional information in connection with a response.

As mentioned above, HTTP defines seven methods, and *GET* and *POST* are the most important. GET is the simplest and simply consists in sending a request to a server that it should send a document with a given document address. That is there is no data block in connection with a GET. The document that GET refers may be a program and you can transfer data to such a program. This happens as part of the URL:

```
GET /index.xhtml?code=7800&date=20030423 HTTP/1.1
```

and you would in the browser's address field types

```
http://server:80/index.xhtml?code=7800&date=20030423
```

Here two data elements are transferred to the page *index.xphtml*. The question mark after the document address indicates that a parameter is send in pairs

keyword=value

If more parameters are sent, each pair must be separated by &.

POST is different as data is encapsulated in the data block, and the method is used when larger and more complex data sizes are sent to the server. With POST it is possible to enclose data of different types in the same request as well as to send binary data to the server. Another important difference between GET and POST is that when GET data is added to the URL, they are also visible to the user in the browser's address bar, which is not always desirable. This is not the case with the POST method. In addition, it should be noted that the server may have limitations regarding the length of an URL, and thus how many data can be sent with GET.

Strictly speaking, what has been said above about the HTTP protocol relates to HTTP/1.1. From the programmer's point of view, it does not make much difference whether the protocol is HTTP/1.1 or HTTP/2. The latter is fully backwards compatible, but of course means improvements. Above I have mentioned that HTTP is text based protocol, but HTTP/2 is actually binary where the browser converts data to a binary format. HTTP/2 is generally more efficient and utilizes, among other things, data compression. You must note that many web-servers still does not support HTTP/2.

1.3 ABOUT HTML

As mentioned I do not want to treat HTML, and the following is only just a very short presentation that merely highlights the most important HTML concepts, but as the browser basically receives HTML, it's useful with a short introduction to the subject. This is especially true after the use of HTML 5, as this standard adds some subjects and principles, and the standard is now supported in most browsers.

HTML is very closely linked to cascading style sheets, also known as CSS, and the goal is that HTML using markup should structure data while CSS defines how the individual elements should be displayed in the browser. Styling of HTML elements should occur in its own document, calling a *Cascading Style Sheet*, while the HTML document alone must take care of structuring the data. That way, web pages are much easier to maintain. I will short mention CSS in the next book, but some of the example in these books use style sheets, just to show what it is.

A HTML document has the basic form:

```
<!DOCTYPE html>
<html>
  <head>
    <title>HTML Application</title>
    <meta charset="UTF-8">
  </head>
  <body>
    ...
  </body>
</html>
```

but many development tools add their own elements to the header. HTML define more structural elements where the most important are:

- *section*, that represents a section in a document, and it can together with H1 - H6 elements be used to indicate the document's structure, and a section must start with a H element
- *article*, that represents an independent piece of content in a document and as the name says an article, and an article element must start with a H element
- *aside*, that represents a content that is only slightly related to the rest of the page
- *header*, that represents a header section
- *footer*, that represents a footer section and will often contain information about the author, copyright information and so on
- *nav*, that represents a section in the document intended for navigation (links)
- *dialog*, that is used to mark up a conversation
- *figure*, that can be used to associate a caption together with some embedded content, such as some graphic or a video

It is not a requirement that a HTML document uses these elements (which are introduced in HTML 5), but it is recommended, among other things, as they can be styled in a style sheet. Similarly, the following structure is recommended for an HTML document, where there may be more sections and where some elements can be omitted completely:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Html5Document</title>
    <meta charset=”UTF-8”>
  </head>
  <body>
    <h1>HTML5 Document</h1>
    <header>
      <h2>Content</h2>
      <nav>
        <ul>
          <li>For example a menu</li>
        </ul>
      </nav>
    </header>
    <section>
      <h2>Section title</h2>
      <article>
        <header>
          <h2>Article title</h2>
          <p>Introdoction to this title</p>
        </header>
        <p>Here is the text.</p>
      </article>
      <article>
        <header>
          <h2>Article title</h2>
          <p>Introdoction to this title</p>
        </header>
        <p>Here is the text.</p>
      </article>
    </section>
    <aside>
      <h2>About this section</h2>
      <p>References or other.</p>
    </aside>
    <footer>
      <p>Copyright and similar</p>
    </footer>
  </body>
</html>
```

There are many other HTML elements, and there are many new elements in HTML 5, but I do not want to mention the elements here. General is the form of a HTML element

```
<elem attr ... >content</elem>
```

where there may be a number of attributes. If there is no content, it is allowed to write:

```
<elem attr ... />
```

even though it is not actually intended to use this notation in HTML 5. A very specific example could be:

```
<h1 class="header1>That is a header</h1>
```

where there is one attribute that refers to a class in a style sheet. As a starting point for this chapter, I have created a web application called *HTMLApplication*. I do not want to display the content here as the page fills a lot, and the page should show only how the above structure elements are used to structure the content of a HTML document. You must ignore the content which is only text, that has nothing to do with this book. When you test / read the document, the file *index.html*, note that the structural elements have no visual effect. They are used solely to structure the text, and the visual effects are solely intended to header elements (H1 - H6) as well as a list of links. In any case, the visual effect is what is default.

It should also be mentioned that it is allowed to define custom attributes and, if necessary, they should start with the word *data-* and the application is that they can be referenced from *JavaScript,* but also from a style sheet.

1.4 HTML FORMS

A HTML form consists of input fields, lists, buttons and so on and is used for data entry. In generally there is not associated with the big challenges, but HTML actually defines a lot about forms. Consider the following document that is part of the *HTMLApplication* project:

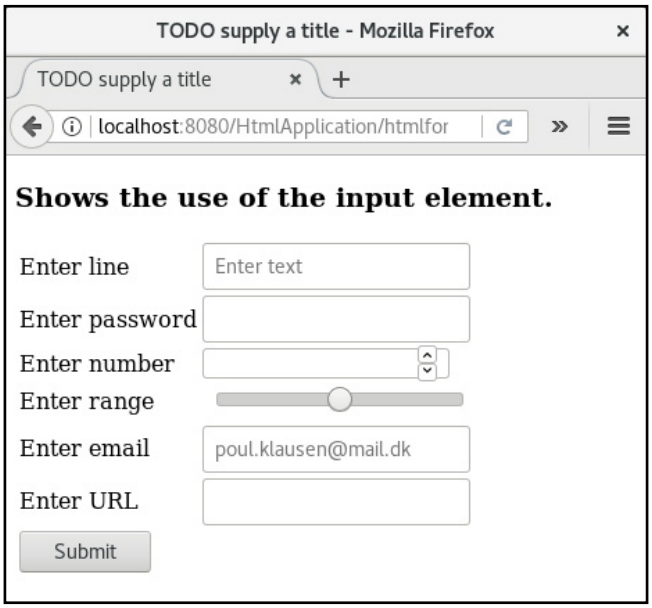
```
<!DOCTYPE html>
<html>
  <head>
    <title>TODO supply a title</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
  </head>
  <body>
    <h1>Shows the use of the input element.</h1>
      <form method="post">
        <table>
          <tr>
            <td>Enter line</td>
            <td>
              <input type="text" id="linefield" required
              placeholder="Enter text" />
            </td>
          </tr>
          <tr>
            <td>Enter password</td>
            <td><input type="password" id="password" /></td>
          </tr>
          <tr>
            <td>Enter number</td>
            <td><input type="number" id="numberfield" step="5" /></td>
          </tr>
          <tr>
            <td>Enter range</td>
            <td><input type="range" id="rangefield" min="100" max="999"
            /></td>
          </tr>
          <tr>
            <td>Enter email</td>
            <td>
              <input type="email" id="mailfield"
              placeholder="poul.klausen@mail.dk" />
            </td>
          </tr>
          <tr>
            <td>Enter URL</td>
            <td><input type="url" id="urlfield" /></td>
```

```
            </tr>
            <tr>
              <td colspan="2"><input type="submit" value="Submit" /></td>
            </tr>
          </table>
        </form>
  </body>
</html>
```

The basic form element is *input*, and you specifies with a *type* attribute how to render the element on the screen and what else it should be able to do with the element. The default *type* is *text*, which means you can enter a single text line. Note (the first input element) that you can specify *required* and that with a placeholder you can specify a hint for the text to be entered. In particular, note the *number* and *range* types that indicate that you can only enter numbers (the last is rendered as a slider), and note the two last ones to enter an email address and a URL, respectively.

You must note how the layout of the form is defined using a HTML table and how a table is defined in HTML. Tables have been a part of HTML all along and although there are other techniques to organize content of a HTML page I will in the following often use HTML tables.

The start page *index.html* in the application *HTMLApplication* has a link *Open form* at the bottom and if you click on this link the form defined above opens



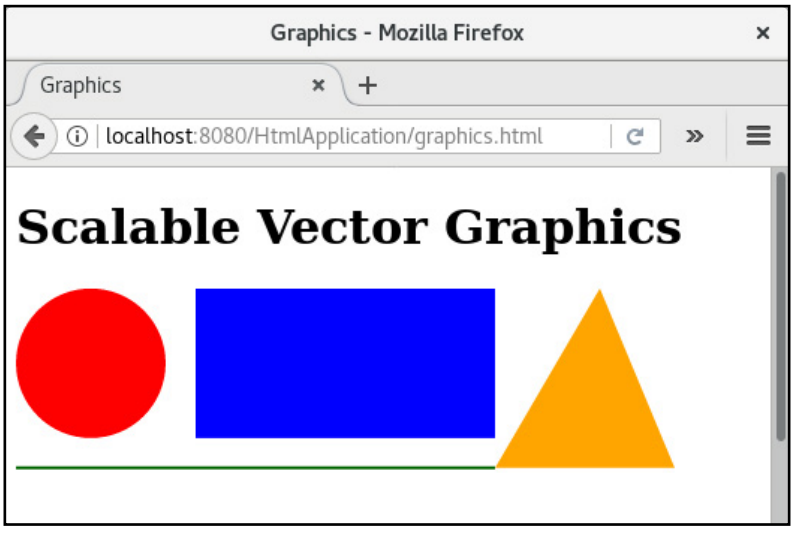
Note that the form is used solely as an example of which *input* elements exists and how to validate data before sending the values to the server.

1.5 SCALABLE VECTOR GRAPHICS

In HTML 5, it is also possible to work with geometric shapes like circles and rectangles. Consider the following page:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Graphics</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
  </head>
  <body>
    <h1>Scalable Vector Graphics</h1>
    <svg height="200" width="500" xmlns="http://www.w3.org/2000/svg">
      <circle cx="50" cy="50" r="50" fill="red" />
      <rect x="120" y="0" width="200" height="100" fill="blue" />
      <line x1="0" y1="120" x2="320" y2="120" stroke-width="2"
        stroke="darkgreen" />
      <polygon points="390,0 440,120, 320,120" fill="orange" />
    </svg>
  </body>
</html>
```

Opening the page in the browser gives you the result:



The code is easy enough to understand and you are encouraged to investigate what else is available.

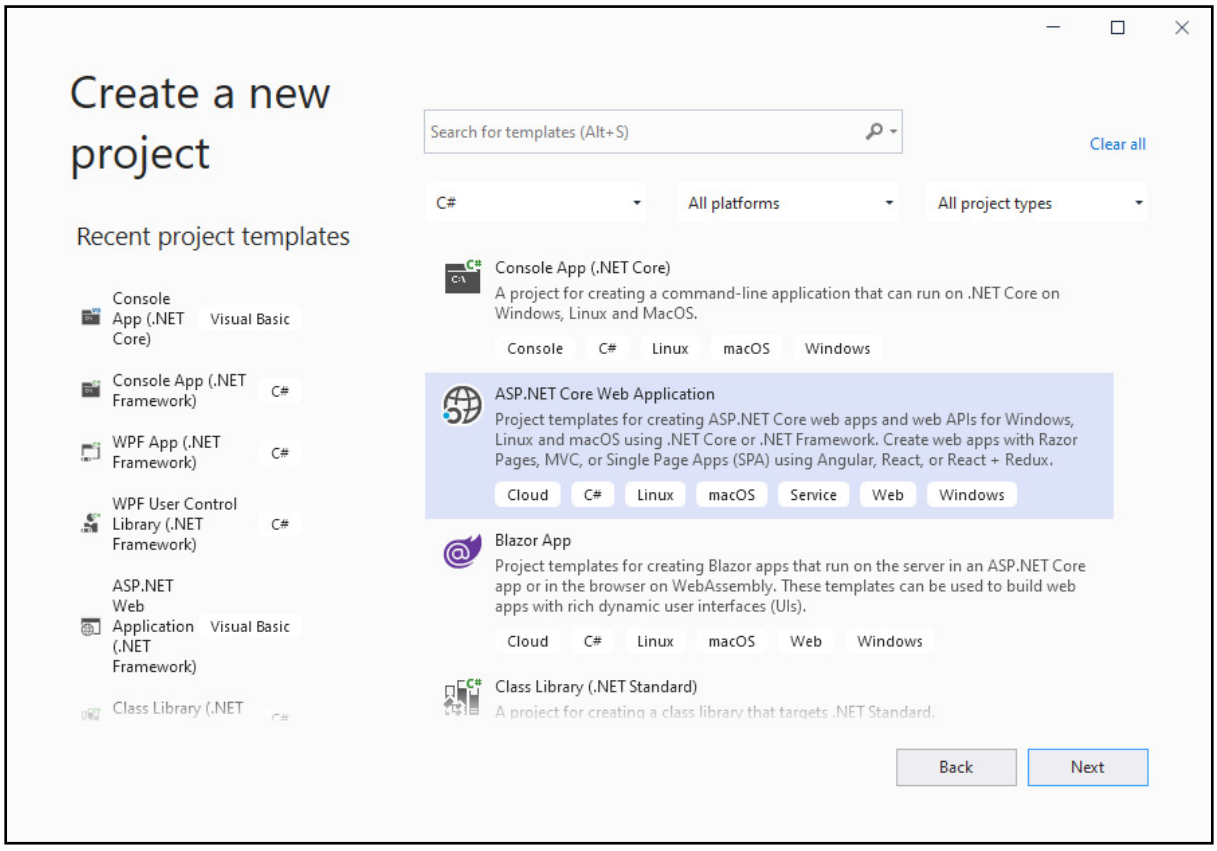
2 CREATE AN ASP.NET CORE PROJECT

In this chapter I will create three web-applications. It is all very simple applications, but conversely applications that are sufficient to show what a web application is and what one does to develop such an application using Visual Studio. The following will primarily focus on how, and can subsequently be used as a guide on how to proceed.

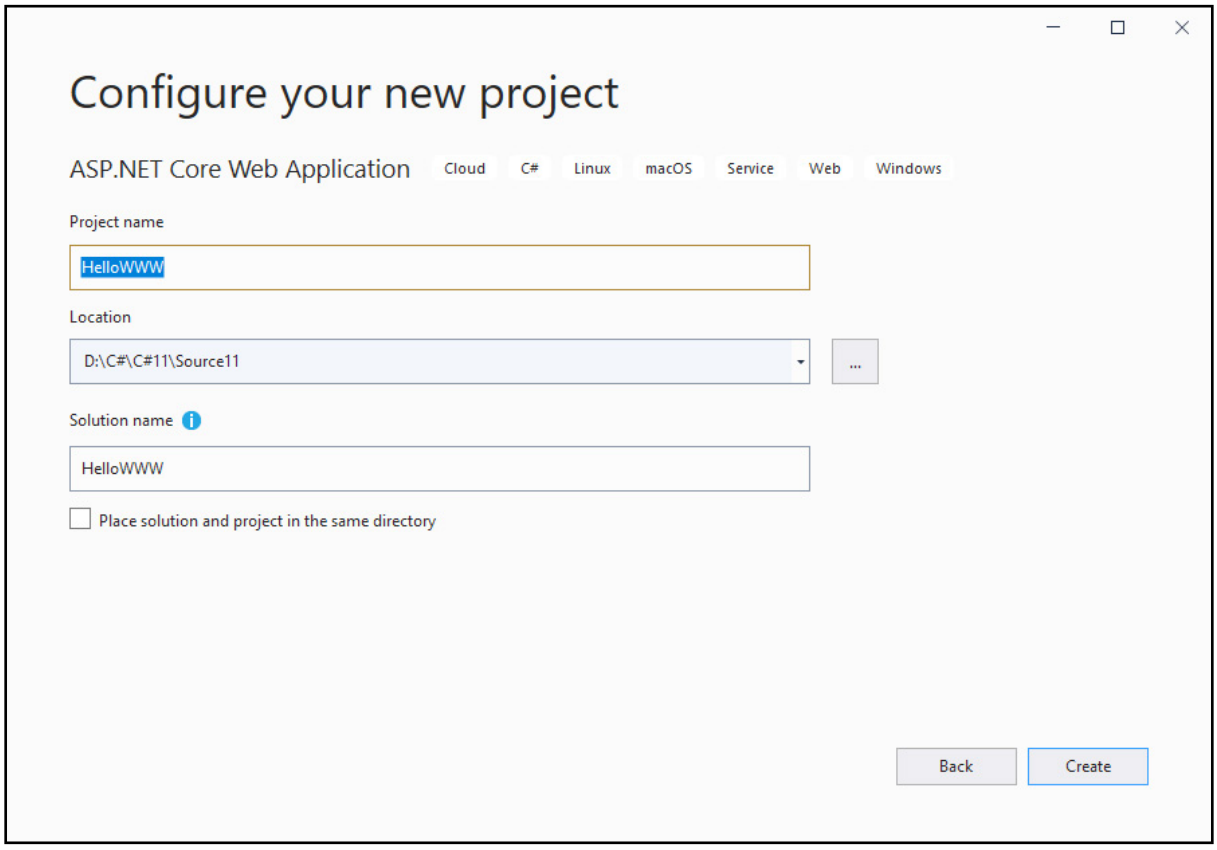
2.1 HELLOWWW

The first example is very simple and is an application which only opens a page showing some text, but the page is dynamic generated on the server before it is send to the client. The sole purpose is to show how to create such an application using Visual Studio.

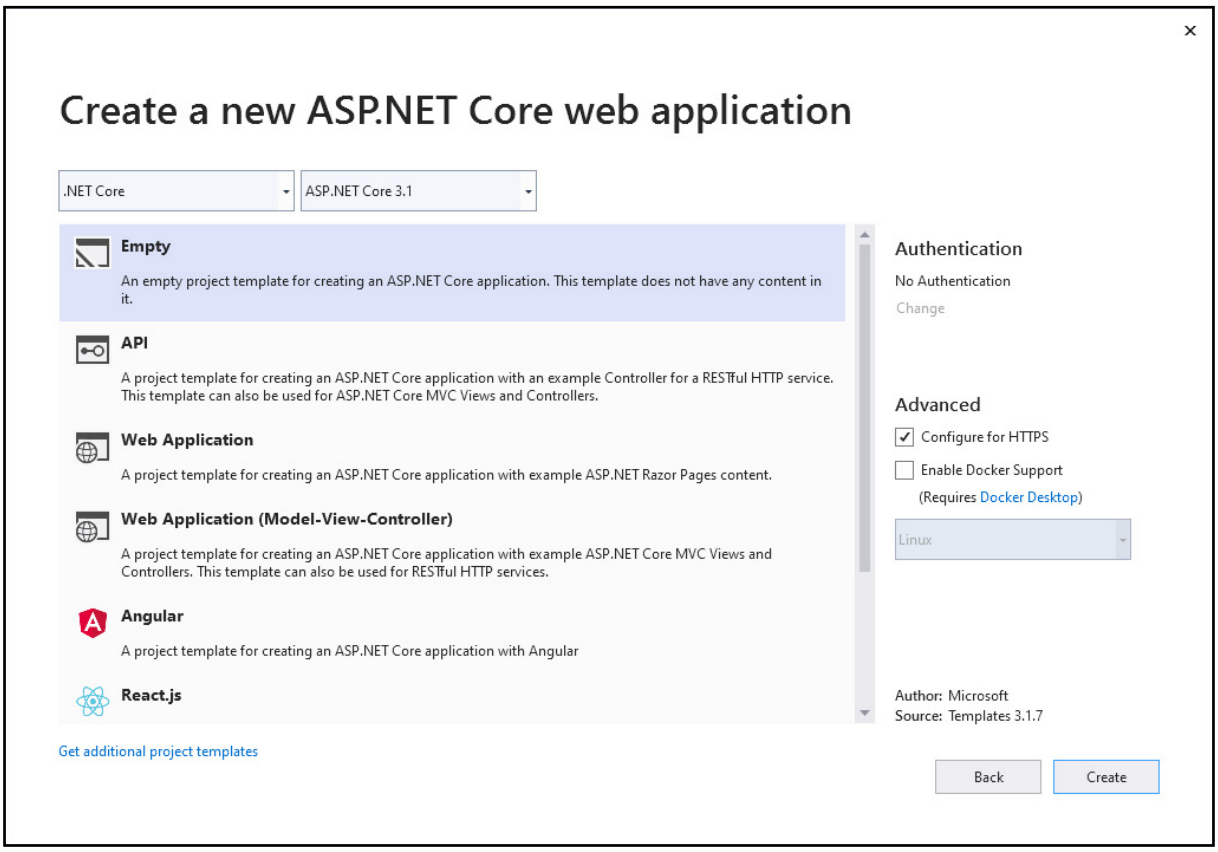
In Visual Studio I create a *ASP.NET Core Web Application* project:



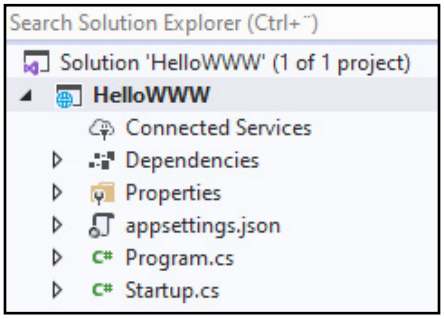
When you click *Next* you get the usual project window. The only thing to do is to select the folder where to store the project and enter the project name:



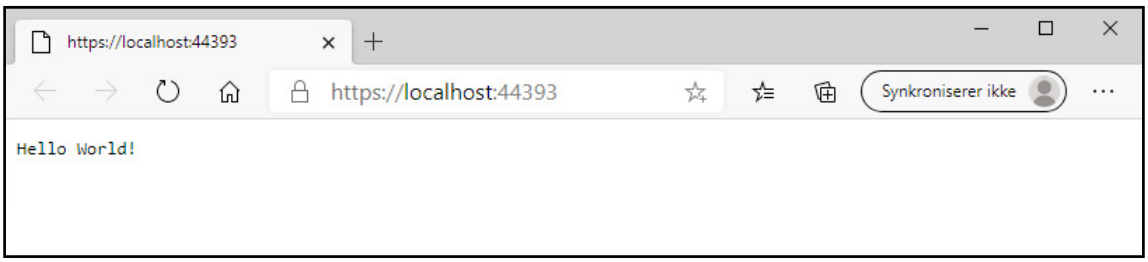
When you clicks *Create* you get a window where to select a project template:



You must note the two combo boxes and there content. A project template tells which directories and files Visual Studio creates and includes in the project, but for the this project and also most of the following I select *Empty*. The result is that I myself have to create many folders and files, but to create simple applications it makes it easier to follow what is happening and why. If you choose another template, Visual Studio will create a ready-made skeleton for an application, which will often contain many things that you do not need, and it can initially make it more difficult to figure out what is needed and what it all means. When I in the above window clicks *Create* Visual Studio creates the following files:



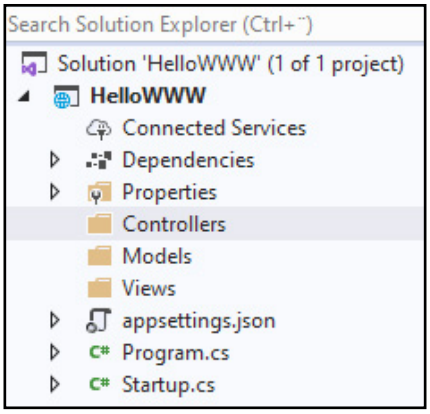
and the most important is the last which is a class that creates some startup code. What Visual Studio has created is an application, and if you in the toolbar click on *IIS Express* Visual Studio opens your browser and shows a page showing the text *Hello World!*



IIS Express is a build in web server in Visual Studio used for development. If a web application is to be used in practice, it must of course be installed on a running web server, but for the time being I will only use *IIS Express*. You should note the port number, which is a number assigned by Visual Studio when the project is created. If you open *Startup.cs* you can see where the test *Hello World!* come from:

```
namespace HelloWWW
{
  public class Startup
  {
    public void ConfigureServices(IServiceCollection services)
    {
    }

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

      app.UseRouting();

      app.UseEndpoints(endpoints =>
      {
        endpoints.MapGet("/", async context =>
            {
          await context.Response.WriteAsync("Hello World!");
        });
      });
    }
  }
}
```

For now it is not a real web application, but only an application which send some text to the web browser. As the first step I have added three folders to the project:



Next I have added a model class to the folder *Models*:

```
namespace HelloWWW.Models
{
  public class Name
  {
    public string Firstname { get; set; }
    public string Lastname { get; set; }

    public static Name[] GetNames()
    {
      return new Name[] {
        new Name { Firstname = "Gorm", Lastname = "Den gamle" },
        new Name { Firstname = "Harald", Lastname = "Blåtand" },
        new Name { Firstname = "Svend", Lastname = "Tveskæg" } };
    }
  }
}
```

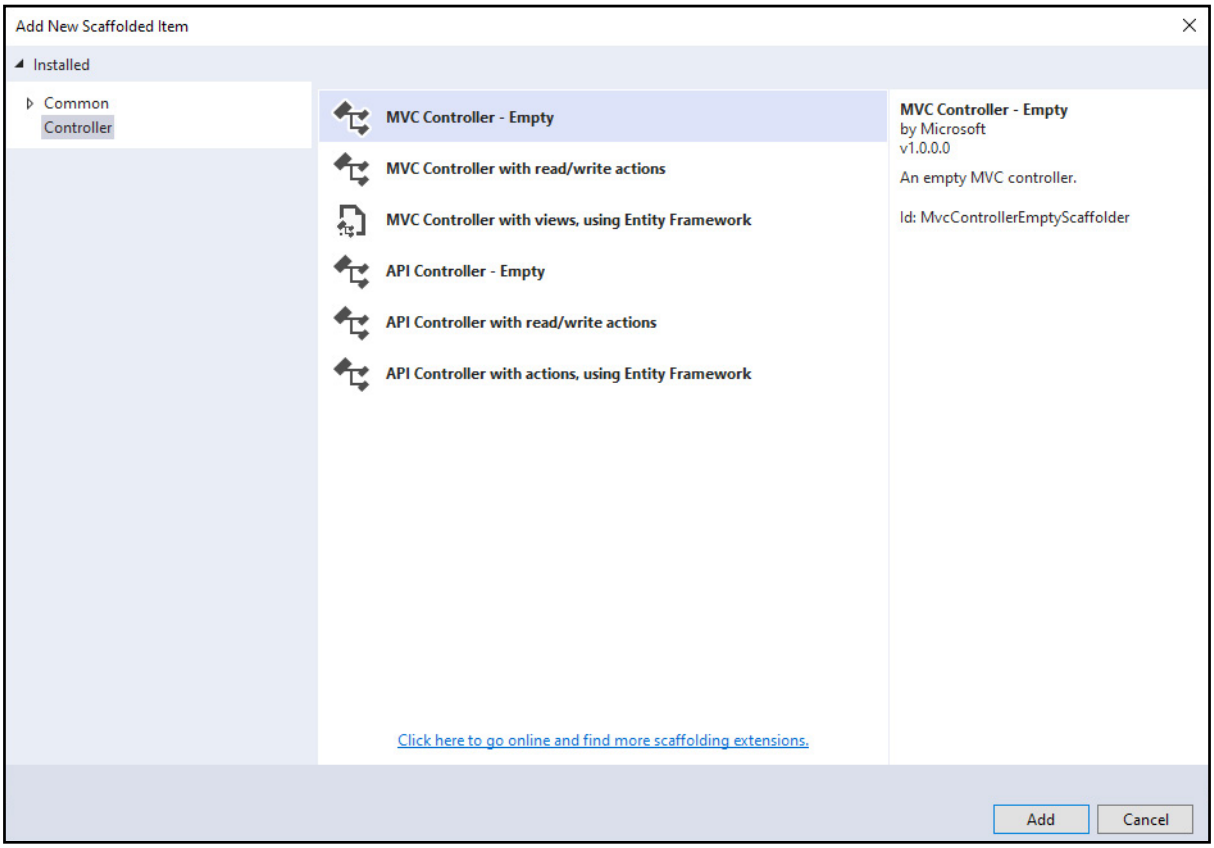
It is a usual model class and there is nothing to note even if it is a class in a web application.

I then right click om the folder *Controllers* and in the popup menu I select *Controller* and get the window below. Here I clicks on *MVC Controller - Empty* and when I clicks *Add* Visual Studio opens the usual window to add a new item. The name is

HomeController.cs

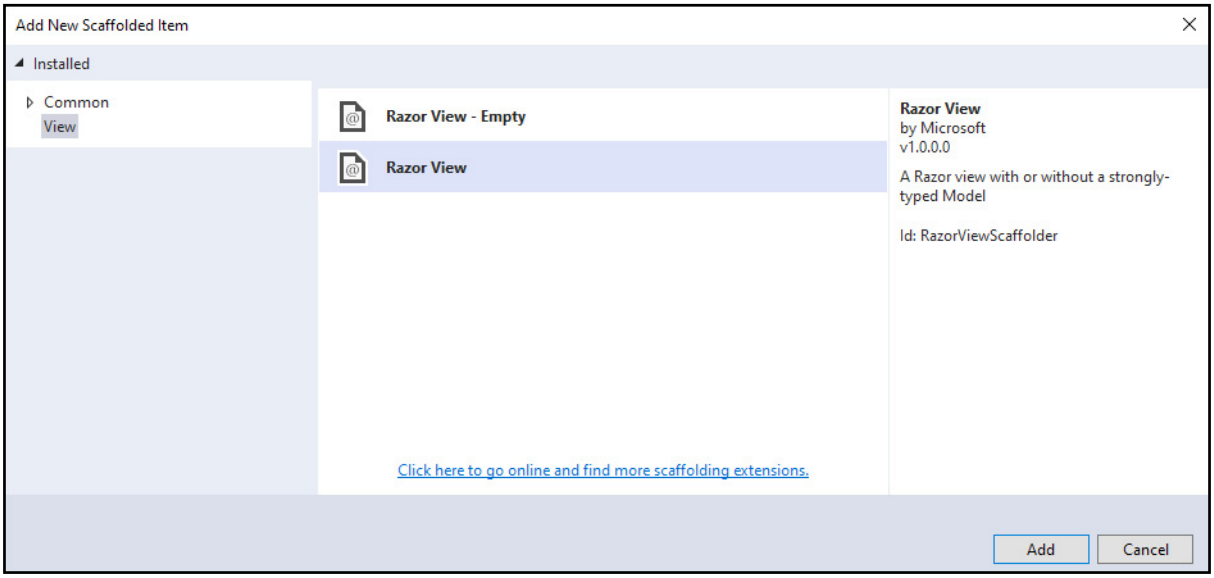
I select this name which is the name Visual Studio uses for the default controller. A controller represents an object to where a user from a browser can send a request, and it is the the controller that must handle the request and typical send a view back to the user's browser as a response. The result is the following class:

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

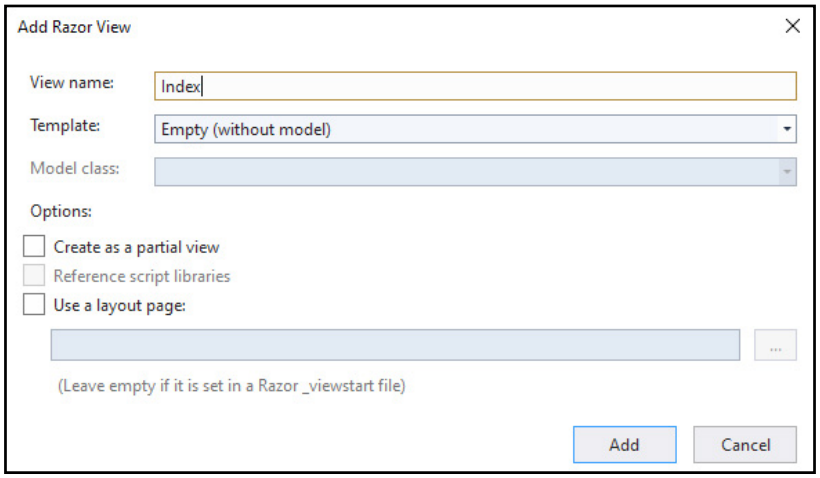


Also here I have selected an empty controller, but it must be a MVC controller.

A controller is a class and contains action methods, in this case the method *Index()*. An action method is the method which is called when a user sends a request to this application and as so the method which starts the execution of the application. Typical will an action send a response to the client for the request, a response which often will be a web page created as a view. In the *Views* folder I have added a subfolder called *Home*. When I right click on the folder and select *View* in the popup menu, Visual Studio opens the following window:



Here I select *Razor View* and click *Add*:



As template I select *Empty (without model)* and enter *Index* for name. When I clicks *Add* Visual Studio creates a view called *Indes.cshtml*:

```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
<body>
</body>
</html>
```

There is not much to note, but the content starts with a razor block, which is code that must be executed on the web server, and in this case there is nothing. Later I will show how to use this block to define a layout for the current view. Else you can see that the view is html, and to have some content I have added a H1 element to the body:

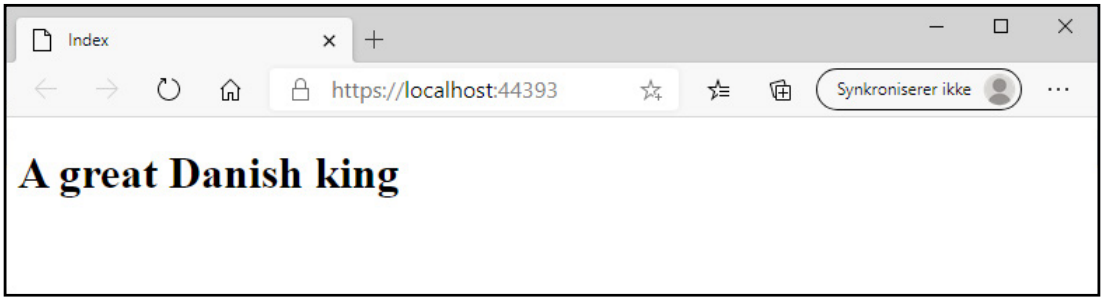
```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
<body>
  <h1>A great Danish king</h1>
</body>
</html>
```

To bind it all together and to have the view sent to the browser, *Startup.cs* must be updated:

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

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

    app.UseRouting();

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

An endpoint represents a start point for a web application and before it was a method which send some text to the browser. Now it is defined that the endpoint should be the controller.

If you now run the program (click on *IIS Express*) Visual Studio opens the following browser:



The result is a web application. A simple one, but an application with a controller and a view. For now there is nothing dynamic in the application as the view is only static HTML, but to show how to dynamic create the view I will make use of the model. I have change the controller such it creates a *Name* object and sends this object as a parameter to the view:

```
public class HomeController : Controller
{
  public ViewResult Index()
  {
    return View(new Name { Firstname = "Christian", Lastname = "The
    fourth" });
  }
}
```

Also note that I have changed the return type as controller should return a view. You should note that the code is all C#, and both the model and the controller are usual C# classes. Then I have updated the view:

```
@model HelloWWW.Models.Name
@{
  Layout = null;
}

<!DOCTYPE html>

<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Index</title>
</head>
<body>
  <h1>A great Danish king</h1>
  <b>@Model.Firstname @Model.Lastname</b>
  <h3>Other old Danish kings</h3>
  <ul>
    @foreach (HelloWWW.Models.Name n in HelloWWW.Models.Name.GetNames())
    {
    <li>@n.Firstname @n.Lastname</li>
    }
  </ul>
</body>
</html>
```

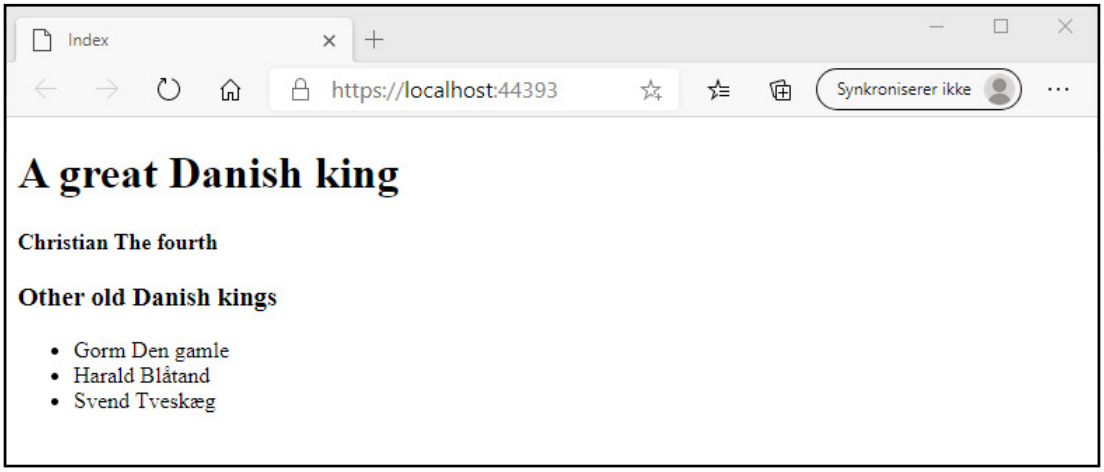
The controller sends a *Name* object to the view, and this object must be defined in the view, what is the purpose of the first line. It means that the parameter in the view is known as the name *Model*, and you should think of the view as a view for a model object. The name *Model* is used in the body to reference the object, and you must note how to reference to properties on the model object. If you write

@Model.Firstname

it means that the value of this property is inserted in the html. You must also note

@foreach

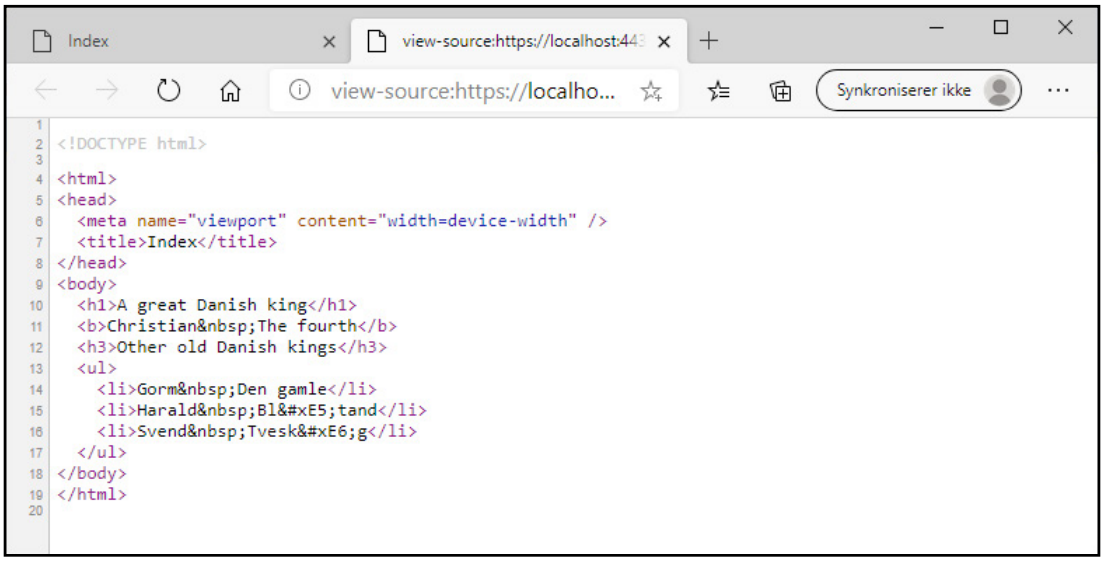
which defines a loop executed on the server to generate html inserted in the view. If you now run the application the result is:



It is of course an extremely simple application, but still an application that dynamically on the server generates an html page that is sent to the browser. If in the browser you select

Show source code

you can see which html the application has sent to the client:



Exercise 1: DanishKings

You must create a web application which in principle is identical as the above and where the application is developed in the same way as above. You can follow the guidelines below.

1) Create a new *ASP.NET Core Web Application* as you can call *DanishKings* when it as above should be an empty project. Add the three folders: *Controllers*, *Models* and *Views*.

Modify the file *Startup.cs* in the same way as above.

Build the project to ensure you have not entered any errors. If you do not have any errors you are ready to develop the application.

2) Create another folder which you can call *Data*. The previous book *C# 10*, the program *PrintProject*, has a XML document called *Kings.xml*. Add this file to the new folder *Data*. The same project has a file *Kings.cs* with two classes. Add this file to the folder *Models*. The file must be modified:

1. Change the namespace to *DanishKings.Models*
2. Remove the method *GetStream()* in the class *Kings*
3. Modify the method *Load()* such it reads the XML document in the *Data* folder:

```
using (Stream stream = new FileStream("Data/Kings.xml", FileMode.Open))
```

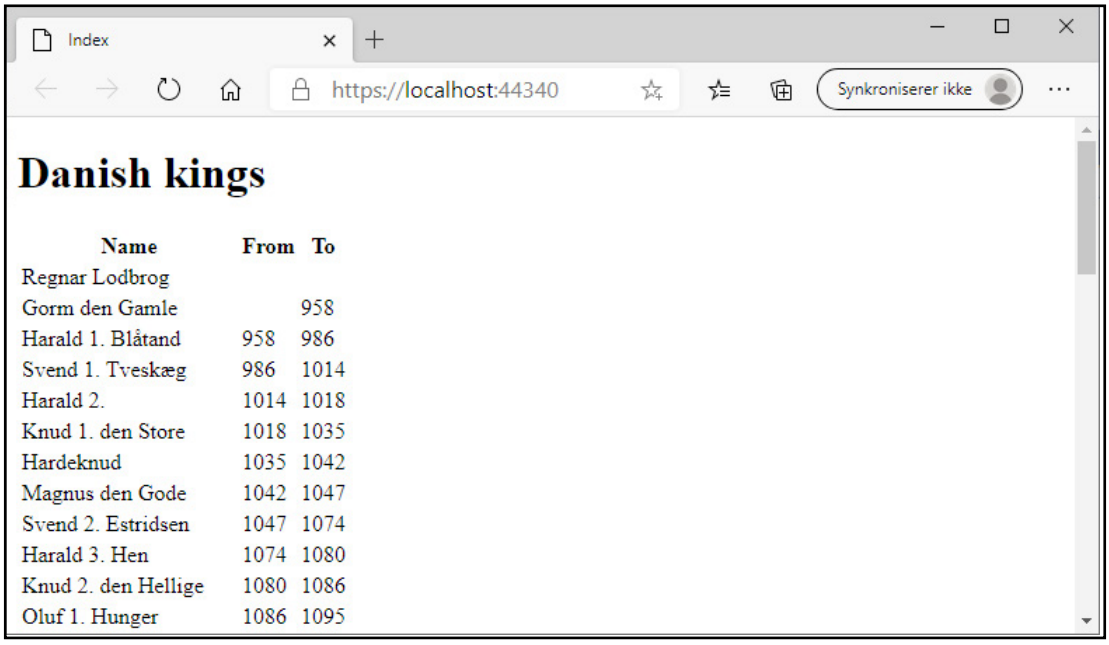
Now the applications model is ready.

3) Add a controller to the *ControllersFolder*. The name should be *HomeController* and and the response should be a view created from a *Kings* object as parameter:

```
public class HomeController : Controller
{
  public ViewResult Index()
  {
    return View(new Kings());
  }
}
```

Remember to change the return type for the method *Index()*.

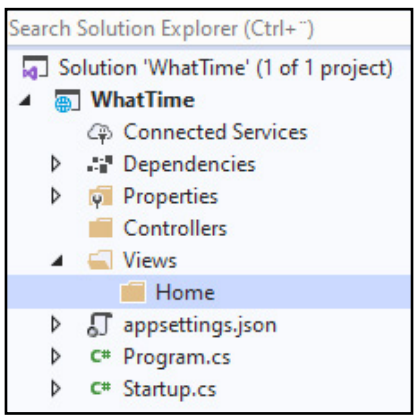
4) The last step is to write the view when the view must show all Danish kings (all names in the file *Kings.xml*) in a html table:



2.2 WHATTIME

The next example is a web application which opens a web page showing the date and time and to be more accurate the date and time for the server. Somehow, this is perhaps the simplest example of a dynamic web application imaginable.

To write the application I creates an ASP.NET Core Web Application project in exactly the same way as above. When the project is created I add three folders:



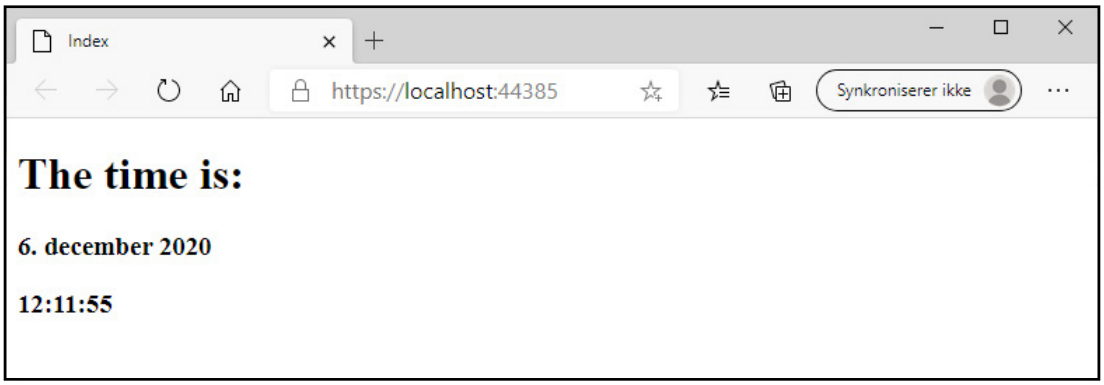
You should note that this time there is no *Models* folder as the application does not use model classes. I then add a controller:

```
namespace WhatTime.Controllers
{
  public class HomeController : Controller
  {
    public ViewResult Index()
    {
      return View(DateTime.Now);
    }
  }
}
```

Note that current date and time is send as a parameter to the view. A view is added to the *Home* folder:

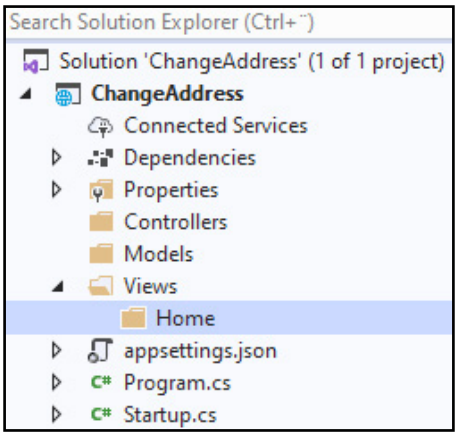
```
@model System.DateTime
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
<body>
  <h1>The time is:</h1>
  <h3>@Model.ToLongDateString()</h3>
  <h3>@Model.ToLongTimeString()</h3>
</body>
</html>
```

As the last thing the class *Startup.cs* must be change as before and the application can run:



2.3 CHANGEADDRESS

I am now ready to write an application where the user can enter address data in a form and submit the data to the server. The application must simulate a situation where a user wants to register an address change. It is an application with more views and as so a more realistic application than the applications shown above. The project start is an ASP.NET Core project as above called *ChangeAddress* and after I have created the project folders the result is:



As the next steps I:

1. Add a *HomeController*
2. A view for the controller
3. Update the controller to return a view
4. Update *Startup.cs*

After these steps I am ready to start the development.

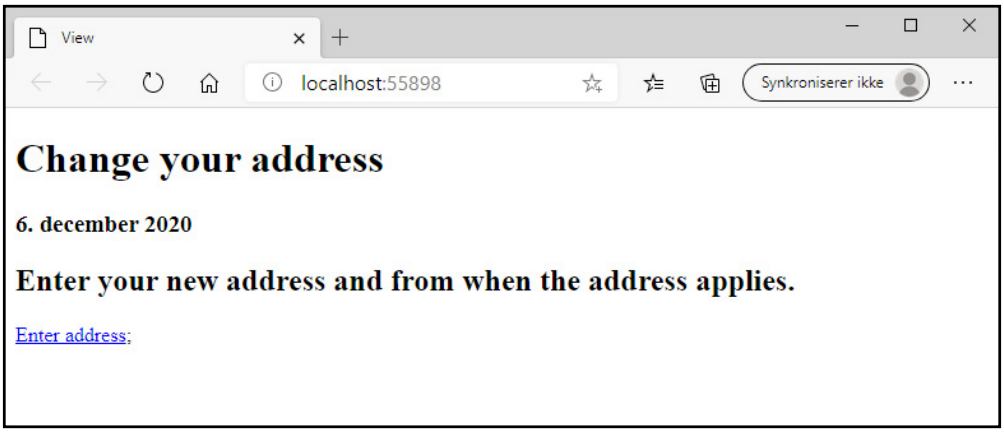
First I have modified the controller to send the current date to the view:

```
public ViewResult Index()
{
  return View(DateTime.Now);
}
```

Then I have changed the view to show a welcome page:

```
@model System.DateTime
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>View</title>
</head>
<body>
  <div>
    <h1>Change your address</h1>
    <h3>@Model.ToLongDateString()</h3>
    <h2>Enter your new address and from when the address applies.</h2>
    @Html.ActionLink("Enter address", "AddrForm");
  </div>
</body>
</html>
```

The view has a link which is created using a method in a class *Html*. It is a class with helper methods to create HTML elements, and the method *ActionLink()* creates a hyperlink. The first parameter is the text shown in the browser, and the other parameter is the name of the action in the controller to be performed when the user clicks on the link. You should note that the line starts with a @ which tells the server to perform a method and substitute the result in the html.



Next I must expand the project with a model class (a class added to the folder *Models*):

```
namespace ChangeAddress.Models
{
  public class Address
  {
    public string Firstname { get; set; }
    public string Lastname { get; set; }
    public string Addrline { get; set; }
    public string Zipcode { get; set; }
    public string Email { get; set; }
    public string Date { get; set; }
    public bool? IsNew { get; set; }
  }
}
```

The class has properties for all the data elements the user can enter. Next I have updated the controller:

```
public class HomeController : Controller
{
  public ViewResult Index()
  {
    return View(DateTime.Now);
  }

  public ViewResult AddrForm()
  {
    return View();
  }
}
```

The new action in the controller requires a new view, and you can the view called *AddrForm* as above. The folder *Home* will then contains two views. The view is empty, but it should contains what is needed to enter a model object:

```
@model ChangeAddress.Models.Address

@{ Layout = null; }

<!DOCTYPE html>

<html>
  <head>
    <meta name="viewport" content="width=device-width" />
    <title>AddrForm</title>
  </head>
  <body>
  <h1>Enter address</h1>
  @using (Html.BeginForm())
  {
    <p>First name: @Html.TextBoxFor(x => x.Firstname) </p>
    <p>Last name: @Html.TextBoxFor(x => x.Lastname) </p>
    <p>Address: @Html.TextBoxFor(x => x.Addrline) </p>
    <p>Zip code: @Html.TextBoxFor(x => x.Zipcode) </p>
    <p>Your email: @Html.TextBoxFor(x => x.Email)</p>
    <p>Address date: @Html.TextBoxFor(x => x.Date)</p>
    <p>
      Is you a new member?
      @Html.DropDownListFor(x => x.IsNew, new[] {
        new SelectListItem() {Text = "Yes, I am a new member",
          Value = bool.TrueString},
        new SelectListItem() {Text = "No, I am an existing member",
          Value = bool.FalseString}
      }, "Choose an option")
    </p>
    <input type="submit" value="Submit address" />
  }
  </body>
</html>
```

The result of the view must be a HTML form, and the form is created using some helper methods, methods in the class *Html* that create HTML elements. The first statement creates a HTML form, and you should note how to create it with the statement

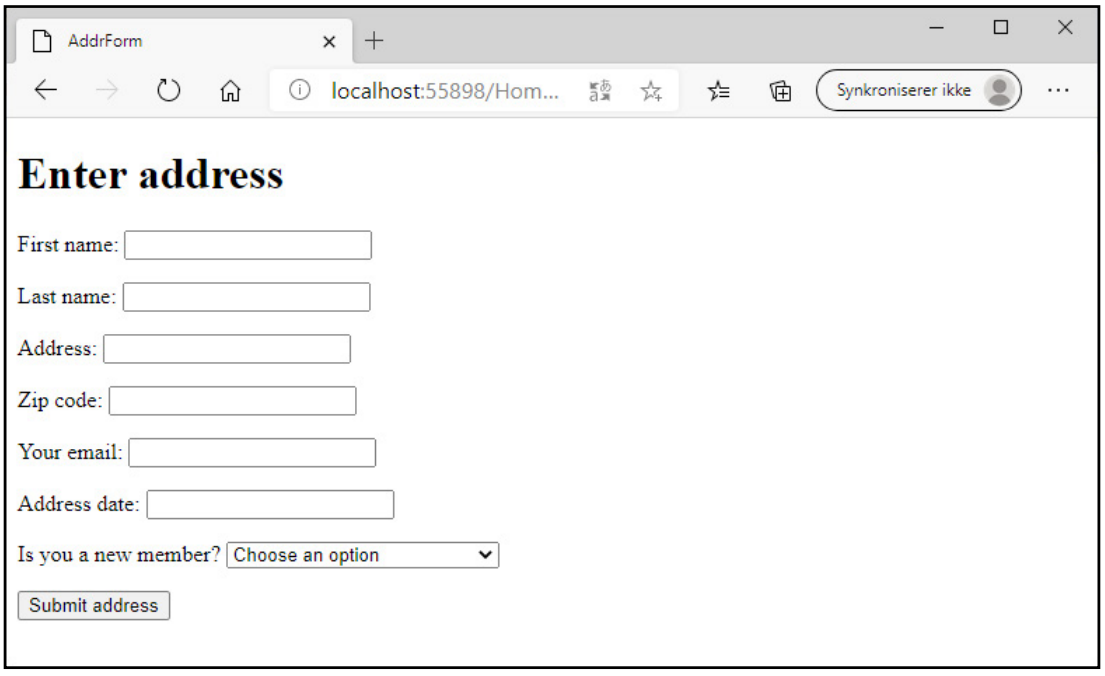
```
@using (Html.BeginForm()) { ... }
```

BeginForm() is a method to create a form, and *@using* means to end the form when the scope ends. A method like *TextBoxFor()* creates a HTML input element, but an element bound to a property in the model for this view. In the same way *DropDownListFor()* is a method which creates a drop down list, and again bound to a property in the model. Here you should note how to define the elements in the drop down list and how it happens in a code block:

```
@Html.DropDownListFor(x => x.IsNew, new[]
{
  new SelectListItem() {Text = "Yes, I am a new member",
    Value = bool.TrueString},
  new SelectListItem() {Text = "No, I am an existing member",
    Value = bool.FalseString}
}, "Choose an option")
```

and this code is not HTML, but a a call of a C# method which creates HTML.

The last line defines a submit button, but defined as usual in HTML. If you run the application and click the link in the start page, the application opens the following page:



When you run the program and enter data in the form and then clicks the submit button nothing happens, except that all fields are blanked. The reason is that unless otherwise stated, a form by submit will perform a submit to itself, which simply means that the form is displayed again, and this without any processing of the form's data. The solution is of course to assign an action to the submit button. When you request a page from the browser, a HTTP GET is performed. When you perform a submit it is instead a HTTP POST command. I will then modify the controller so it has two actions, one for GET and one for POST:

```
public class HomeController : Controller
{
  public ViewResult Index()
  {
    return View(DateTime.Now);
  }

  [HttpGet]
  public ViewResult AddrForm()
  {
    return View();
  }

  [HttpPost]
  public ViewResult AddrForm(Address addr)
  {
    return View();
  }
}
```

The class does now have an override for the method *AddrForm()*, and the two methods are decorated to tell which of them to be used for GET and POST. The method for POST has a parameter of the type *Address* and then a model object.

If you look at the view for the form it is model for an *Address* object, but when the form is opened from the controller no object is send as parameter. The definition of the *Model* object in the view tells which model object to use. The properties for such an object is then bound to the input fields using Razor statements, a technique called *Model Binding*. When the user click the submit button the forms data are send using a POST to the action method (with the same name as the view) in the controller as key / value pairs. When the view is view for an *Address* object such an object is created and corresponding to the model binding initialized with values and it is the object used as parameter in the action method.

It is then the action method that has to use the object to something. In this case the object with address information must be saved in a repository, and typical it will be an database. To make it simple I will store the objects in a file, and I have added the following class to the model:

```
namespace ChangeAddress.Models
{
  public class Addresses : IEnumerable<Address>
  {
    private static string filename = "D:\\Temp\\Adresses.dat";
    private List<Address> list;

    public Addresses()
    {
      Deserialize();
    }

    public void Add(Address addr)
    {
      list.Add(addr);
      Serialize();
    }

    public IEnumerator<Address> GetEnumerator()
    {
      return list.GetEnumerator();
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
      return (IEnumerator)GetEnumerator();
    }

    private void Serialize()
    {
      try
      {
        FileStream file = File.Create(filename);
        BinaryFormatter bf = new BinaryFormatter();
        bf.Serialize(file, list);
        file.Close();
      }
      catch
      {
      }
    }
```

```
    private void Deserialize()
    {
      try
      {
        FileStream file = File.OpenRead(filename);
        BinaryFormatter bf = new BinaryFormatter();
        List<Address> list = (List<Address>)bf.Deserialize(file);
        file.Close();
        this.list = list;
      }
      catch
      {
        list = new List<Address>();
      }
    }
  }
}
```

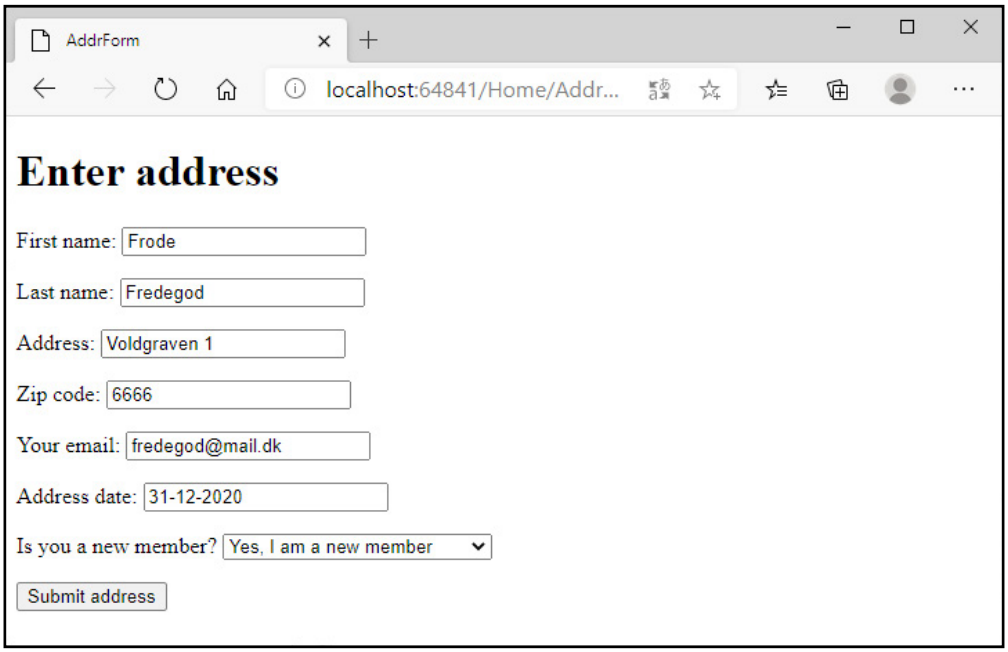
The class is simple and store *Address* objects using simple object serializing, and you should ignore the problem that all the repository is load and stored for each address. To save an address the action method must be changed:

```
public ViewResult AddrForm(Address addr)
{
  (new Addresses()).Add(addr);
  return View("Receipt", addr);
}
```

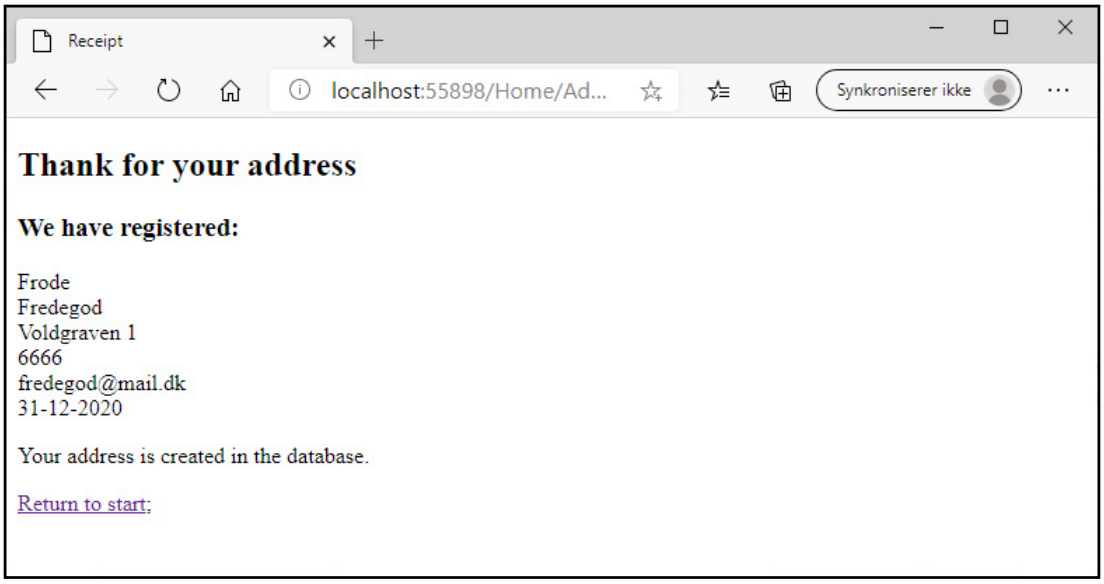
Now the new address is saved, and the method returns a view with an receipt. The view is called *Receipt* and must be a view for *Address* objects:

```
@model ChangeAddress.Models.Address

@{ Layout = null; }

<!DOCTYPE html>

<html>
<head>
  <meta name=”viewport” content=”width=device-width” />
  <title>Receipt</title>
</head>
  <body>
    <div>
      <h2>Thank for your address</h2>
      <h3>We have registered:</h3>
      @Model.Firstname<br/>
      @Model.Lastname<br />
      @Model.Addrline<br />
      @Model.Zipcode<br />
      @Model.Email<br />
      @Model.Date<br />
      <p>
        @if (Model.IsNew == true)
        {
          @:Your address is created in the database.
        }
        else
        {
          @:Your address is updated in the database.
        }
      </p>
      @Html.ActionLink(“Return to start”, “Index”);
    </div>
  </body>
</html>
```

Note the directive for the model class. Else, I think the code is easy enough to understand, but note how to reference the properties in the model object. Also note how to write an if statement, and the view has a link which sends the user back to the start page. If you enter data in the form:



and clicks the submit button you get the window below. The submit button performs a POST, and when the submit button is in a form for an *Address* object the submit will POST this object to an action for this parameter, and the result is that the action for an object of this type as parameter is performed. Then the view for this action is used to create the response. This view is *Receipt* and is a view for an *Address* object which generate the HTML for the window below.



It is a simple view without anything new:

```
@model ChangeAddress.Models.Address
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name=”viewport” content=”width=device-width” />
    <title>Receipt</title>
</head>
<body>
  <div>
    <h2>Thank for your address</h2>
    <h3>We have registered:</h3>
    @Model.Firstname<br />
    @Model.Lastname<br />
    @Model.Addrline<br />
    @Model.Zipcode<br />
    @Model.Email<br />
    @Model.Date<br />
    <p>
      @if (Model.IsNew == true)
      {
        @:Your address is created in the database.
      }
      else
      {
        @:Your address is updated in the database.
      }
    </p>
    @Html.ActionLink(“Return to start”, “Index”);
  </div>
</body>
</html>
```

To finish the application the start page *Index.cshtml* is expanded with a new link which send the user to page that shows all addresses registered in the repository. To open this page the controller must have another action method:

```
public ViewResult AddrList()
{
  return View(new Addresses());
}
```

The view for this controller is:

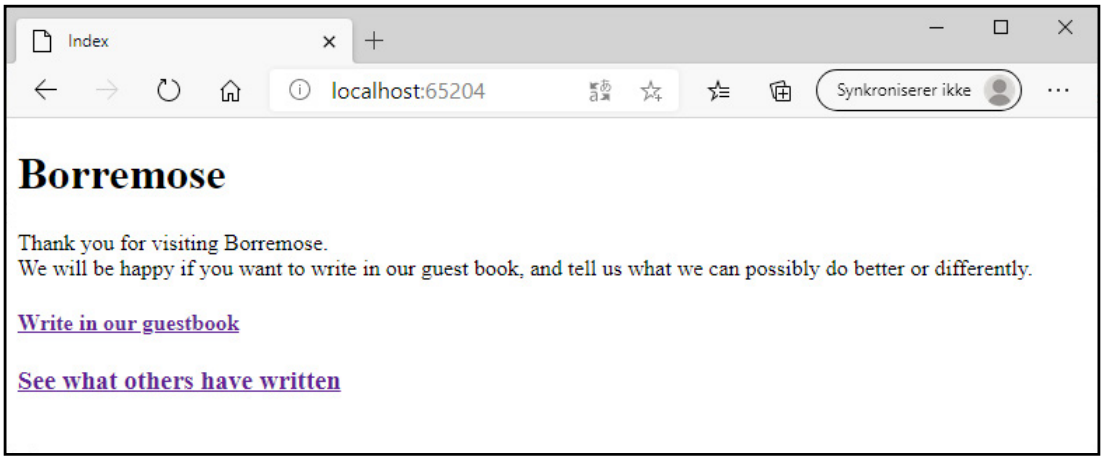
```
@model ChangeAddress.Models.Addresses
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name=”viewport” content=”width=device-width” />
    <title>AddrList</title>
</head>
<body>
  <h1>Addresses</h1>
  <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>
</body>
</html>
```

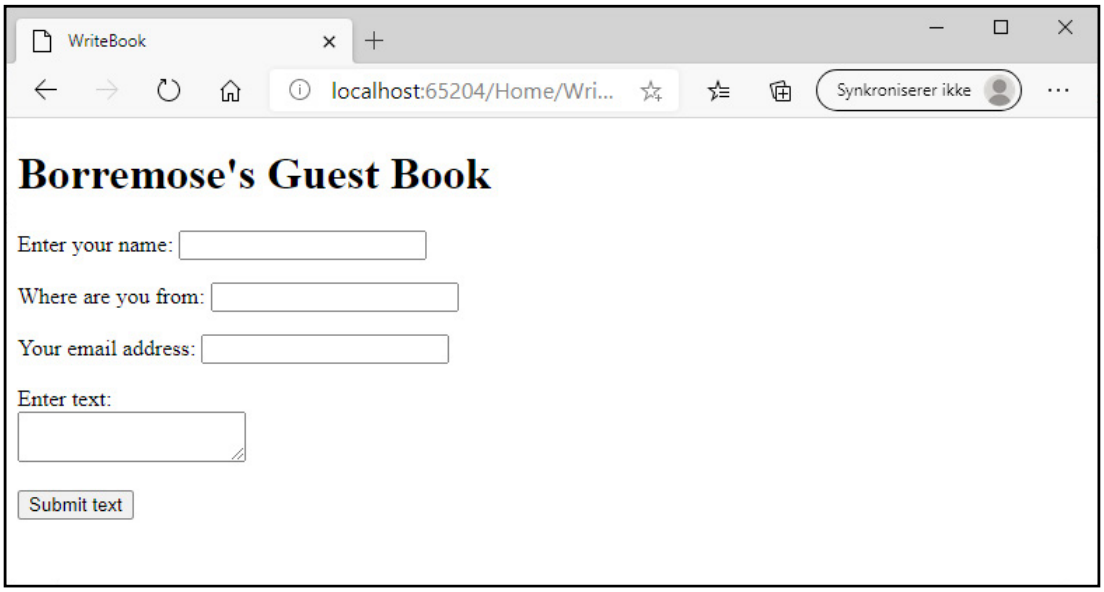
and is a view which shows the addresses in a HTML table.

Exercise 2: Guestbook

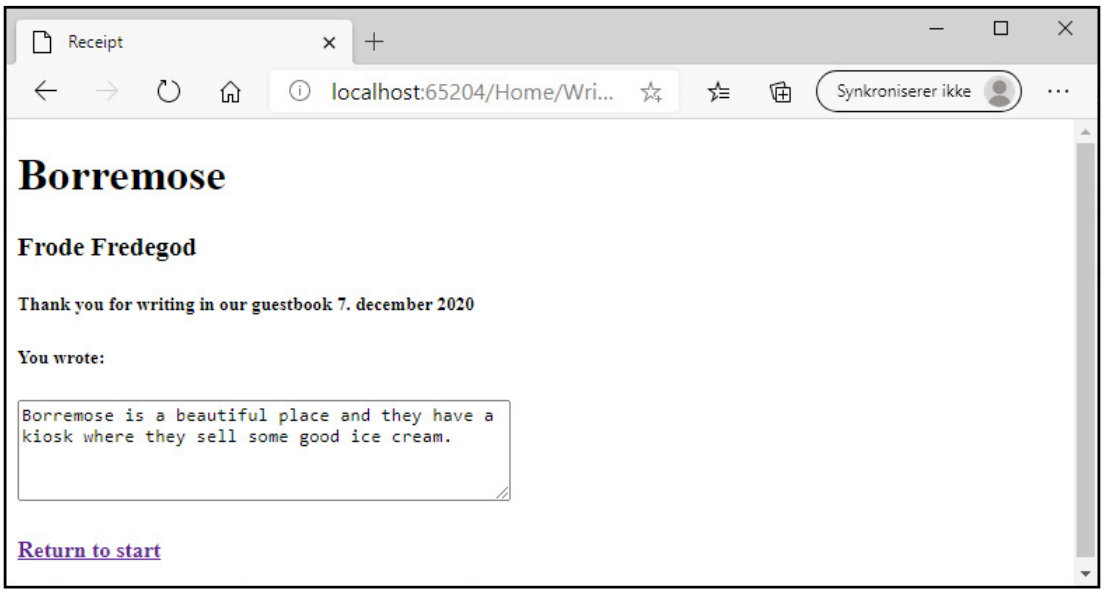
You must write a web application similar to the *ChangeAddress* example. The application should this time simulates an electronic guestbook. When the application starts, it must display the following start page:



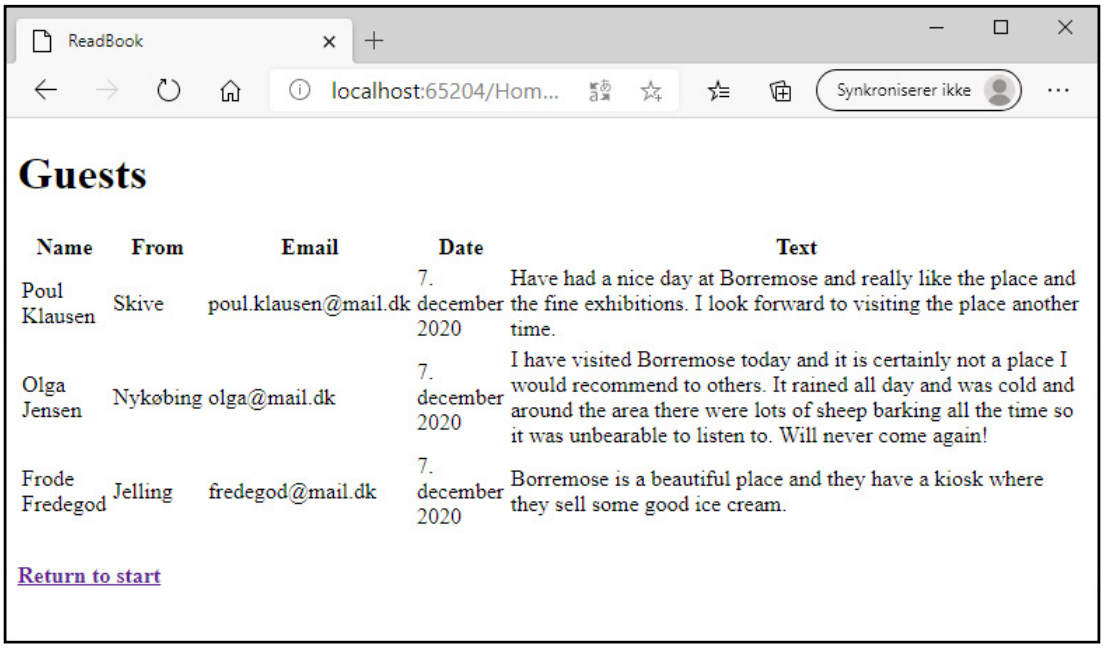
When the user clicks on the link to write in the guestbook the application must open the following form:



When the user submit the form an object for the quest must be saved and you can use a repository as in the application *ChangeAddress* which serializes objects in a file. When information for a guest is saved the application should show a receipt page:



The start page has a link which must open a page to show what others has written. When a user clicks the link the result could be:



Guests

Name	From	Email	Date	Text
Poul Klausen	Skive	poul.klausen@mail.dk	7. december 2020	Have had a nice day at Borremose and really like the place and the fine exhibitions. I look forward to visiting the place another time.
Olga Jensen	Nykøbing	olga@mail.dk	7. december 2020	I have visited Borremose today and it is certainly not a place I would recommend to others. It rained all day and was cold and around the area there were lots of sheep barking all the time so it was unbearable to listen to. Will never come again!
Frode Fredegod	Jelling	fredegod@mail.dk	7. december 2020	Borremose is a beautiful place and they have a kiosk where they sell some good ice cream.

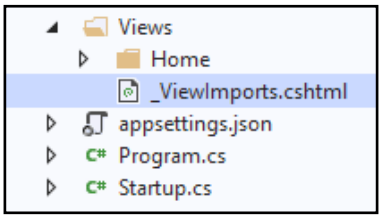
Return to start

3 CHANGEADDRESS IMPROVED

The application from the previous chapter is in principle finish, but if you open the application in the browser, it is clear that a lot is missing. In this chapter I will try to improve the application. I start with a copy of the project from the previous chapter which I have called *ChangeAddress1*. I will improve the application in three directions:

1. use a standard standard tag-library
2. use a style sheet to improve the user interface
3. use validation of the form fields

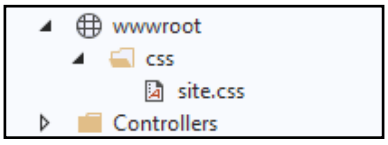
A tag-library is a library of tools which can be used in a view to create HTML elements as alternatives to methods in the class *Html*. There are not immediately the big advantages, but the code looks more like HTML which might make it easier to read the code, and then it is standard in ASP.NET Core. When you create a new project in Visual Studio the library is not immediately available and to use this facility, you must add a file to your Views folder:



with the following content:

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

A style sheet is a text file which is send to the browser together with the server's response, and where you define how the browser should render the different HTML elements. A style sheet must be stored somewhere and it is recommended to do that in a sub-directory in a directory called *wwwroot*:



In this case there is one style sheet called *site.css*. I should not explain style sheets in details in this book. There are many rules and a lot of syntax to learn, but when you see a style sheet is it easy enough to understand the meaning:

```
body { background-color:bisque
}

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

h2 {
  color: darkgreen;
  font-size: 24pt;
  font-weight: bold
}

a:link {
  color: darkgreen;
  font-weight:bold;
}

a:visited {
  color: green;
}

a:hover {
  color: darkgray;
}

a:active {
  color: gray;
}

.date {
  color: darkred;
  font-size: 14pt
}
```

The style sheet defines 8 styles. The first 7 are styles for an HTML element, and if a view for example use a *H1* element and use the above style sheet all *H1* elements will be rendered using the style and draw the text with a dark green color and use a font on 36 points. Note how there are defined 4 styles for a link element. The last style where the name starts with a dot defines a class which is a style that can be assigned to all HTML elements in the view. To use the style two things are needed. The class *Startup.cs* must be changed:

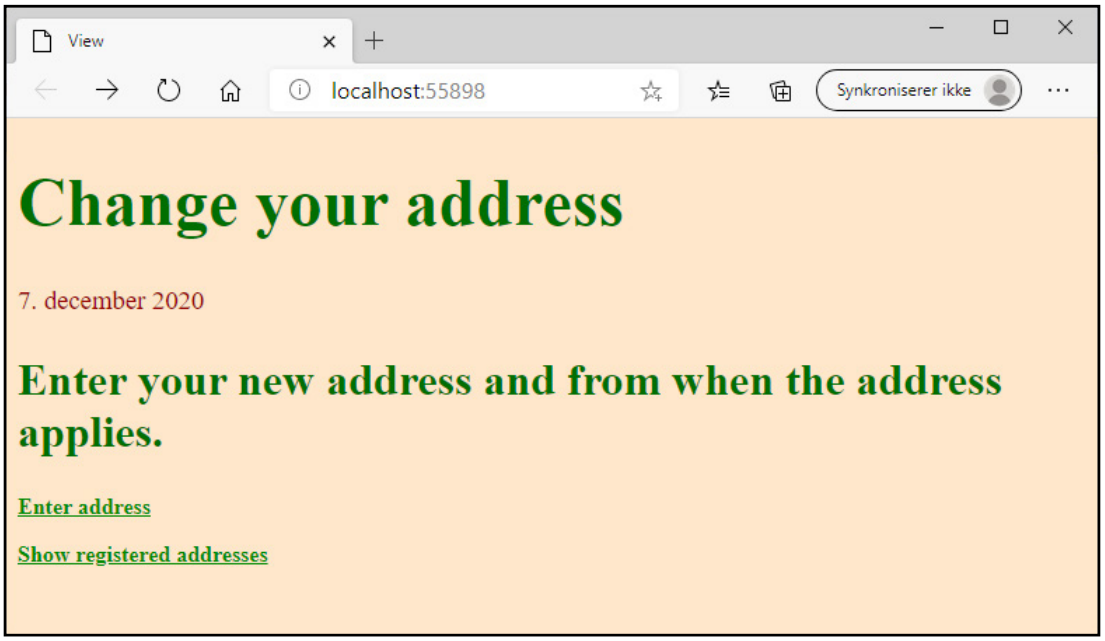
```
public class Startup
{
  public void ConfigureServices(IServiceCollection services)
  {
    services.AddControllersWithViews();
  }

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

    app.UseStaticFiles();
    app.UseRouting();

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

and the style sheet must be reference in the view:

```
@model System.DateTime
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>View</title>
  <link href="~/css/site.css" rel="stylesheet" />
</head>
<body>
  <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>
</body>
</html>
```

Note first how to reference the style sheet and also how in the *div* element to reference the *.date* class in the style sheet. In Visual Studio you can set the reference by dragging the name from Solution Explorer.

Then you must note how the two links are replaced using tag-elements. *asp-action* is a tag-element and the server will from this element generate a HTML link element. If you run the application it shows the following start page:



As the next I will modify the form such it uses the tag-library and the window is styled, but also to validate the form data before they are stored in the repository. First the style sheet is expanded with new classes and in particular five classes are defined

```
.field-validation-error {
  color: #f00;
}

.field-validation-valid {
  display: none;
}

.input-validation-error {
  border: 1px solid #f00;
  background-color: #fee;
}

.validation-summary-errors {
  font-weight: bold;
  color: #f00;
}

.validation-summary-valid {
  display: none;
}
```

These 5 styles have names not defined by me, but it are styles defined by the HTML generated by elements in the tag-library used to show error messages when the input fields are validated. The styles are also used by the input fields in the form, when the form is validated and there are errors. The form is updated as:

```
@model ChangeAddress.Models.Address

@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>AddrForm</title>
  <link href="~/css/site.css" rel="stylesheet" />
</head>
<body>
  <h1>Enter address</h1>
  <form asp-action="AddrForm" method="post">
    <div asp-validation-summary="All"></div>
    <div class="input-line">
      <label asp-for="Firstname" class="label-text">First name:</label>
      <input asp-for="Firstname" class="fieldwidth-1" />
      <span asp-validation-for="Firstname" class="text-danger"></span>
  </div>
  <div class="input-line">
      <label asp-for="Lastname" class="label-text">Last name:</label>
      <input asp-for="Lastname" class="fieldwidth-1" />
      <span asp-validation-for="Lastname" class="text-danger"></span>
    </div>
    <div class="input-line">
      <label asp-for="Addrline" class="label-text">Address:</label>
      <input asp-for="Addrline" class="fieldwidth-1" />
      <span asp-validation-for="Addrline" class="text-danger"></span>
    </div>
    <div class="input-line">
      <label asp-for="Zipcode" class="label-text">Zip code:</label>
      <input asp-for="Zipcode" class="fieldwidth-2" />
      <span asp-validation-for="Zipcode" class="text-danger"></span>
    </div>
    <div class="input-line">
      <label asp-for="Email" class="label-text">Amail address:</label>
      <input asp-for="Email" class="fieldwidth-1" />
```

```
      <span asp-validation-for="Email" class="text-danger"></span>
    </div>
    <div class="input-line">
      <label asp-for="Date" class="label-text">Address date:</label>
      <input asp-for="Date" class="fieldwidth-1" />
      <span asp-validation-for="Date" class="text-danger"></span>
    </div>

    <div class="input-line">
      Is you a new member?<br /><br />
      <select asp-for="IsNew">
        <option value="true">Yes, I am a new member</option>
        <option value="false">No, I am an existing member</option>
      </select>
    </div>
    <input type="submit" value="Submit address" />
  </form>>
</body>
</html>
```

When the user submit the form model binding is used as before, but the action method in the controller is modified to validate the model object:

```
[HttpPost]
public ViewResult AddrForm(Address addr)
{
  if (ModelState.IsValid)
  {
    (new Addresses()).Add(addr);
    return View("Receipt", addr);
  }
  return View();
}
```

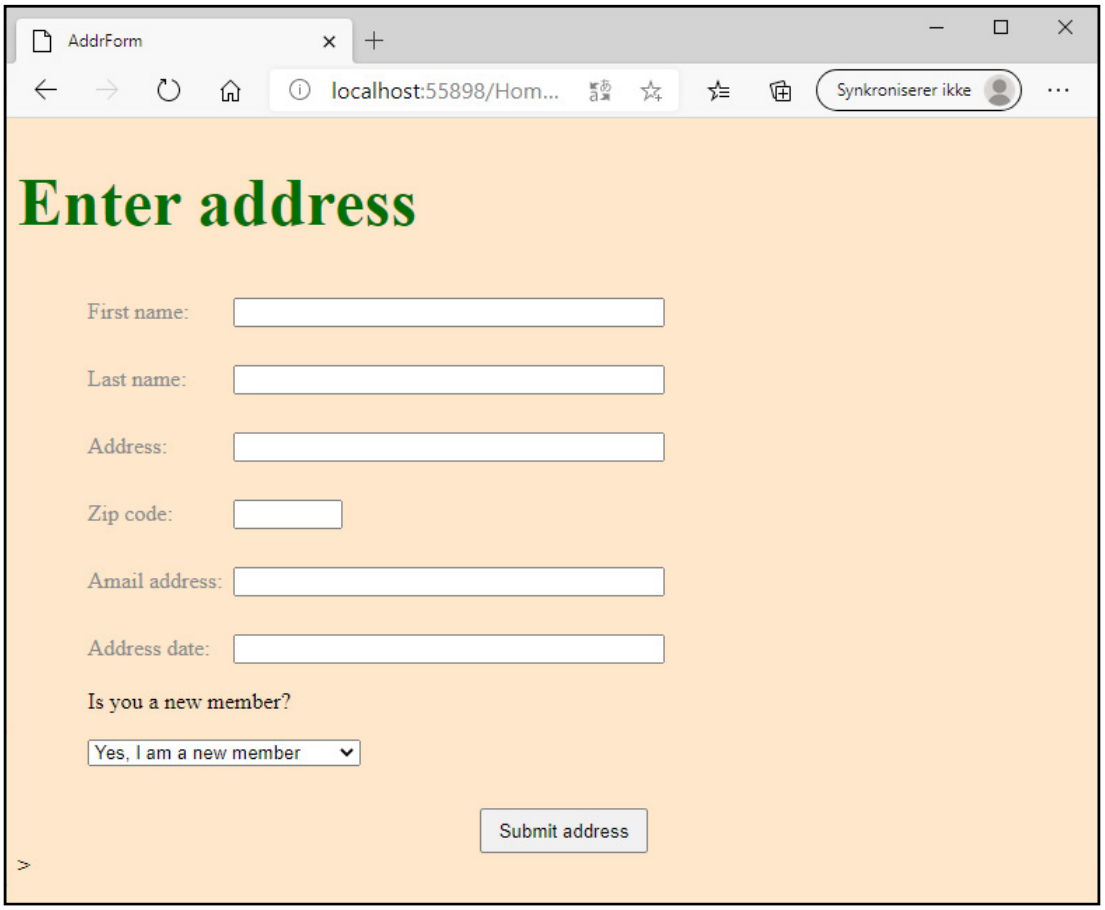
Validating of a form is built-in to ASP.NET, and you do not have to do much. Primarily you have to decorate the properties in the model class:

```
public class Address
{
  [Required(ErrorMessage = "You must enter first name")]
  public string Firstname { get; set; }
  [Required(ErrorMessage = "You must enter last name")]
  public string Lastname { get; set; }
  [Required(ErrorMessage = "You must enter address line")]
  public string Addrline { get; set; }
  [Required(ErrorMessage = "You must enter zip code")]
  [RegularExpression("[1234567890]{4}",
    ErrorMessage = "A zip code must be 4 digits")]
  public string Zipcode { get; set; }
  [RegularExpression(".+\\@.+\\..+",
    ErrorMessage = "You must enter a valid email address")]
  public string Email { get; set; }
  [DisplayFormat(DataFormatString = "{0:dd-MM-yyyy}")]
  [Required(ErrorMessage = "You must date for your address change")]
  public DateTime Date { get; set; }
  [Required(ErrorMessage =
    "You must select where it is a new address of an address change")]
  public bool? IsNew { get; set; }
}
```

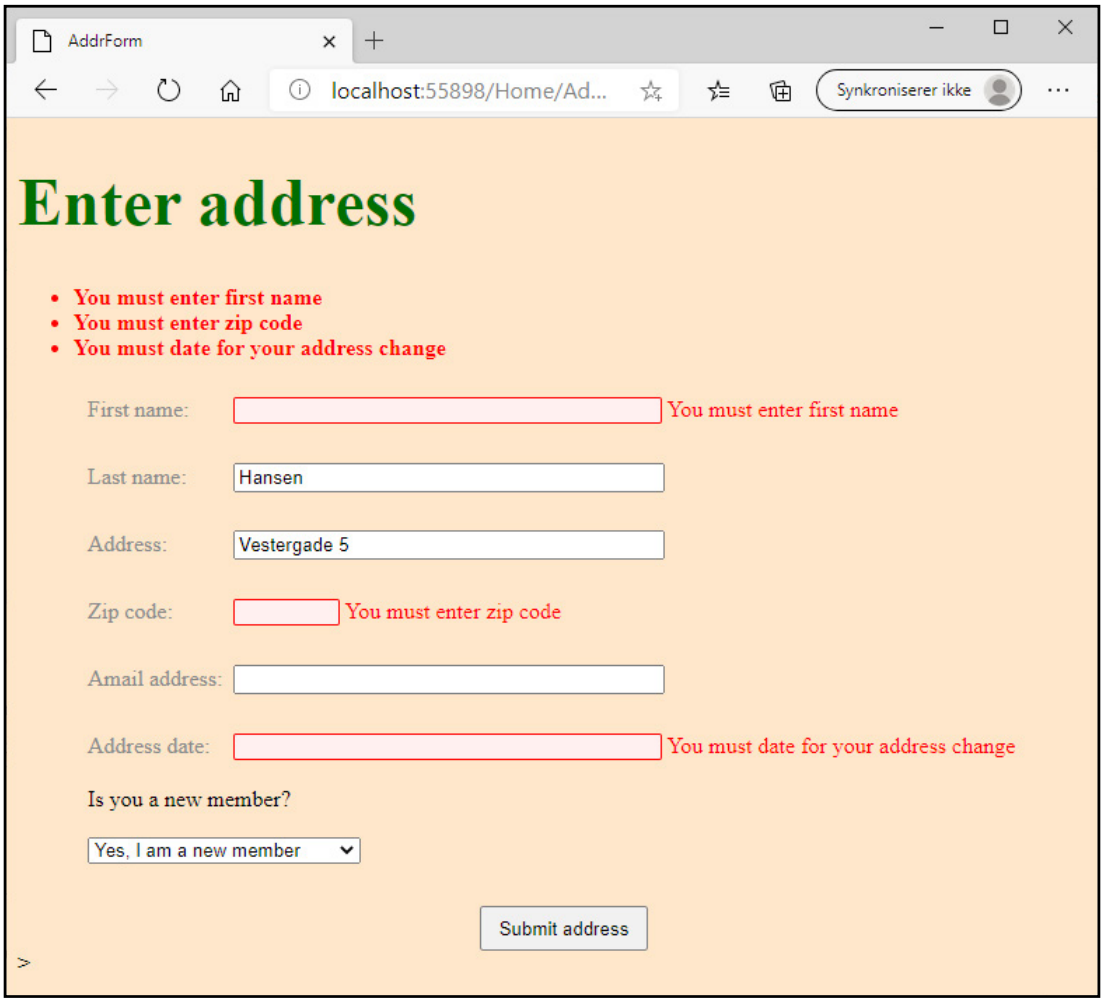
The attributes defines that a property must have a value and if not the error message to show on the screen. Two of the properties are decorated with two attributes. For example the *Zipcode*, which is required, but there is also an attribute for a regular expression that says that the value of a zip code must be four digits. There is also a (very simple) regular expression to validate the email address, but there is no *Required* attribute and you can then create an address without entering a value for email.

Also the *Date* property is decorated with two attributes, but note first that the type is changed from *string* to *DateTime*. The first attribute defines the format of a date, and if the entered value does not have this format there is an error.

If you look at the view it contains fields to show error messages if the validation rules are not met. If you run the application and open the form the result is:

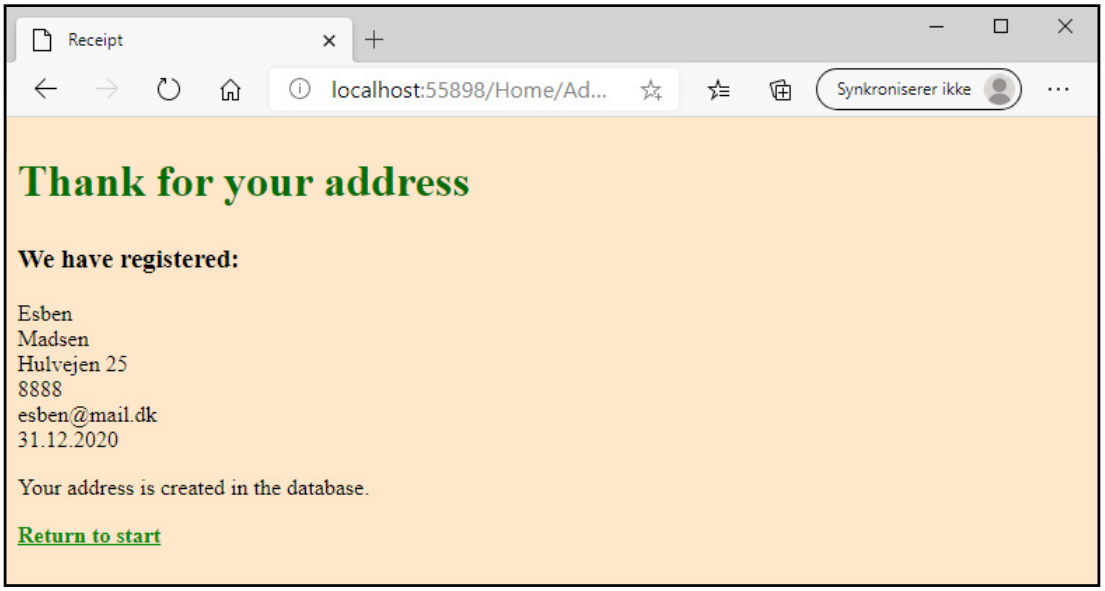


If you enter a value for last name and address and submit the form you get some error messages:

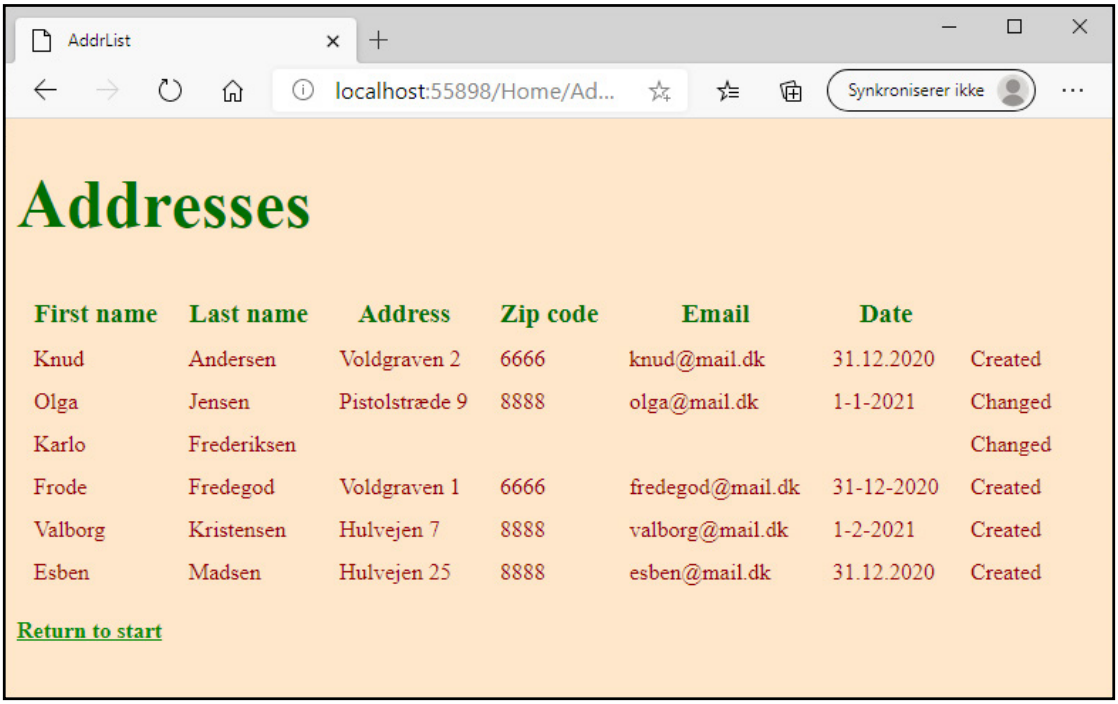


and that is an error message for the fields which can not be validated.

The *Receipt* view also use the style sheet and the result could be:



The last update is the page with the table for the registered addresses. There is not much to do, but the link for return to the start page is changed to an *asp-action* element, and the style sheet is expanded with styles for *TH* and *TD* elements:



Exercise 3: Guestbook improved

In this exercise you must change the solution of exercise 2 in the same way as *ChangeAddress*. I would recommend that you follow the guidelines below.

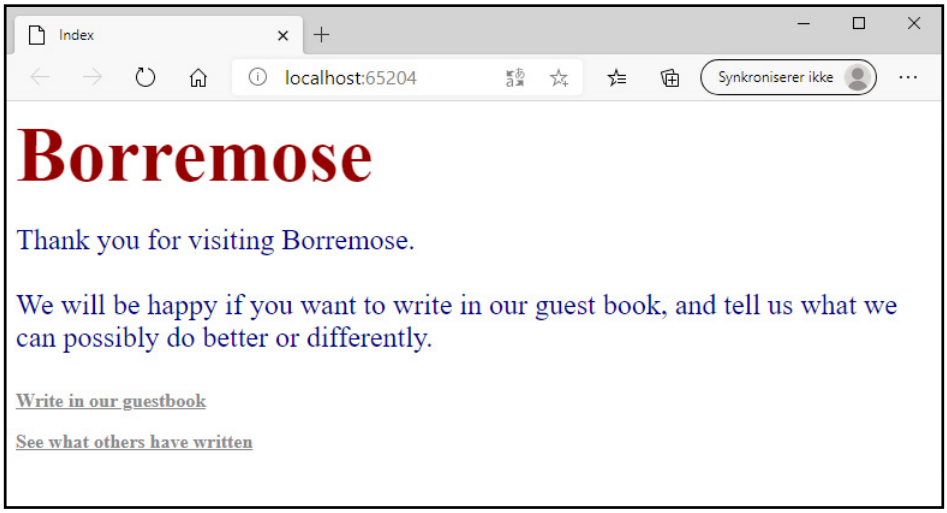
1) Create a copy of your solution from exercise 2 and call the copy *Guestbook1*. Open the copy in Visual Studio. To prepare the changes, you must:

1. Add the file *_ViewImports.cshtml* to your *Views* folder, remember to change the project name in the two first lines in the file.
2. Add a *app.UseStaticFiles();* to the project's *Startup.cs* file.
3. Add a *wwwroot* folder to the project and in this folder a *css* sub folder.
4. Add a style sheet which you can call *site.css* to the folder *wwwroot\css*.

2) Modify the start page when the two links must be changed to *asp-action* elements. To update the page's look and feel you should use styles defined in the above style sheet. The result should be some thing as shown below. Instead of using HTML header elements (H1, H2, ...) you should use DIV elements and define style classes in the style sheet. An example could be:

```
.header-text {
  color: darkred;
  ont-size: 48pt;
  font-weight: bold
}
```

When you have defined and used the styles you must remember to set a reference to the style sheet in the view.

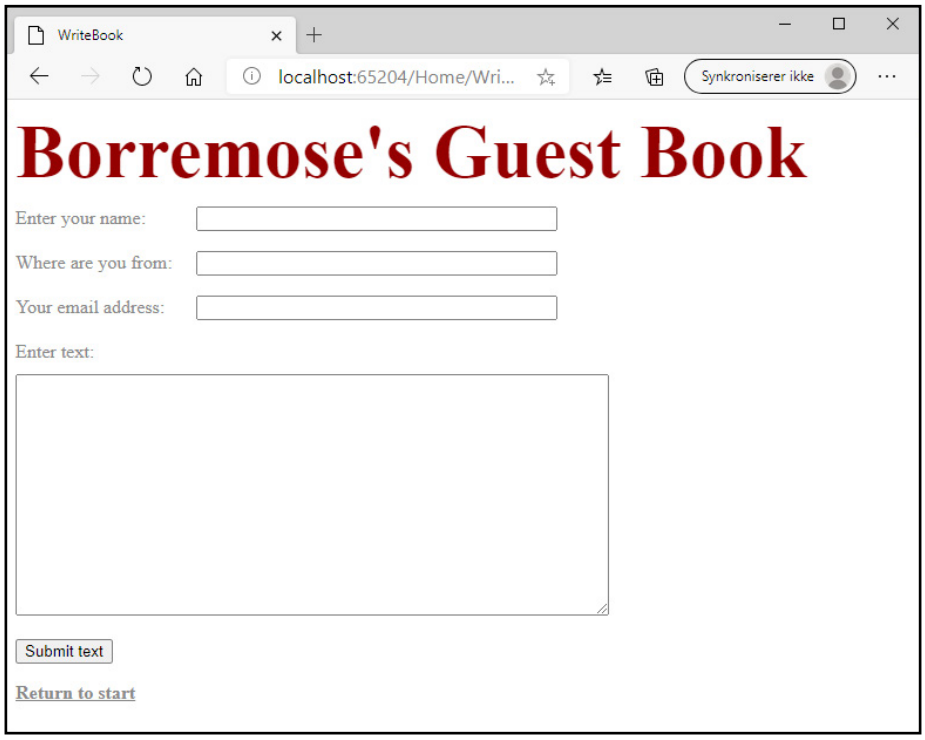


3) You must, using attributes, prepare the model class *Guest* for data validation, when a person who write in the guestbook must

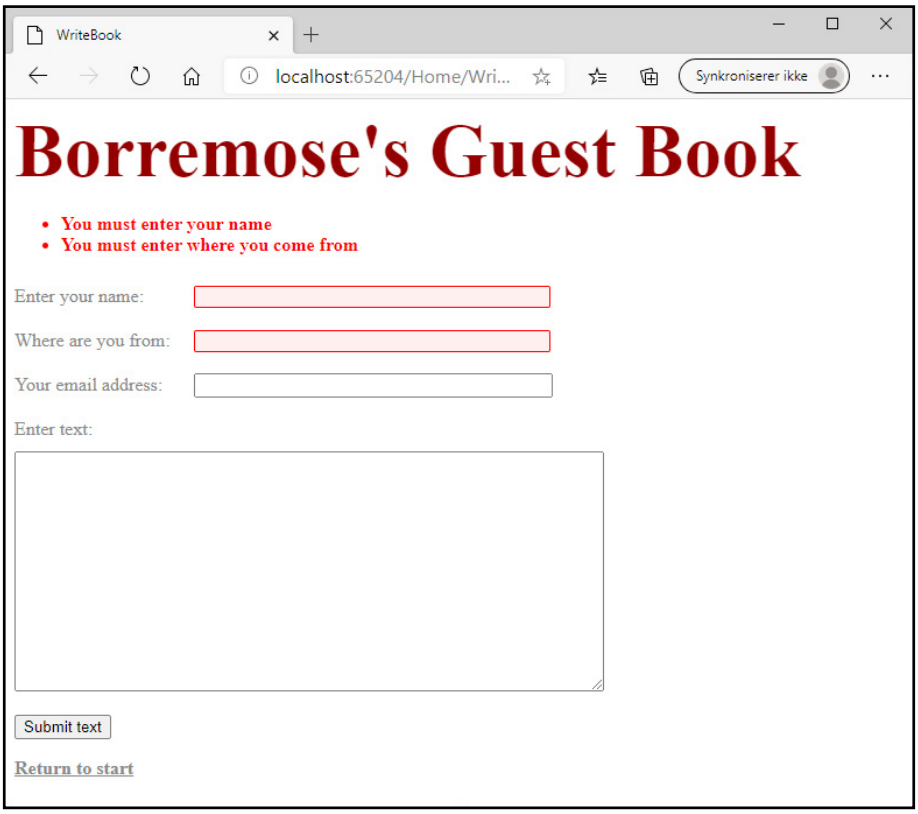
1. write the person's name
2. write where the person come from
3. write a legal email address, where the email address is not required and you can use the same simple validation as used in the application *ChangeAddress1*

The style sheet for the program *ChangeAddress1* has five styles used for data validation. Copy these styles to your style sheet.

4) As the next step you must update the form to something like the following:



where the page is expanded with a link to the start page and cancel action. The look and feel must be updated using styles and all input elements must be changed to asp-tag elements. The form should have a validation summary at the start of the page, but not for each field, and if you submit an empty form the result should be:



Remember also to update the action method in the controller to validate the *Guest* object.

5) You must also update the *Receipt* page with a nicer look and feel.

6) As the last step you must update the view which shows the guestbook. The result could be as shown below, where the column with the email address is removed. Instead is added a new column width a link for each row which should be used to remove a row in the table and as so a user entry in the guestbook. You should ignore that such a function does not make sense in relation to real guestbook. To implement the function more steps are needed:

Add a method to the class *Guests* which remove the object (guest) with index *n*. Note that the method must remove the object from the list and then serialize the list.

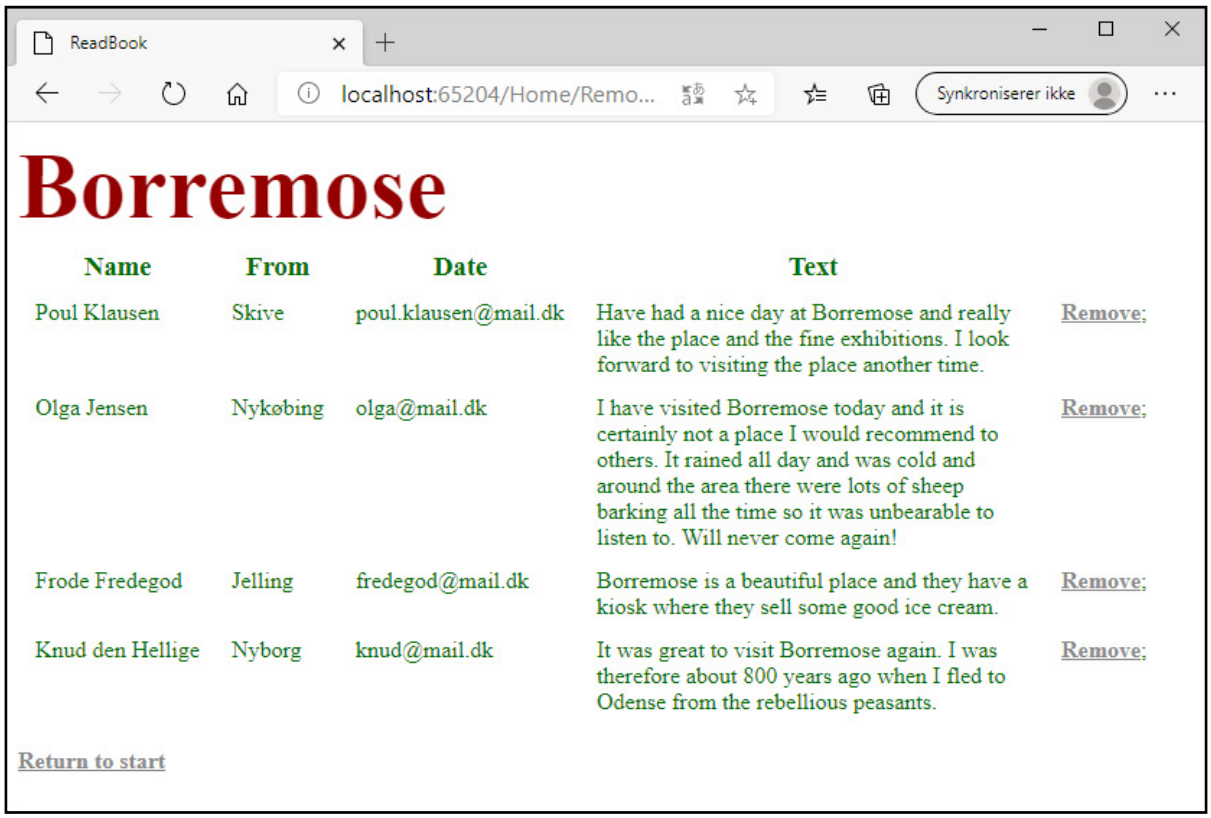
Add a new action method to the controller

```
public ViewResult RemoveBook(int id)
{
  Guests guests = new Guests();
  guests.Remove(id);
  return View("ReadBook", guests);
}
```

Then there is the view where the code to create the rows in the table can be written as:

```
@{int id = 0;
  @foreach (Guestbook.Models.Guest guest in Model)
  {
    <tr>
      <td>@guest.Name</td>
      <td>@guest.From</td>
      <td>@guest.Mail</td>
      <td class="table-field">@guest.Text</td>
      <td>
        @Html.ActionLink("Remove", "RemoveBook", "Home", new { id = @id },
          new { @class = "link-text" });
      </td>
    </tr>
    ++id;
  }
}
```

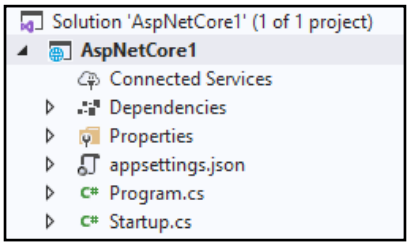
Note that the code defines a local variable, but else just accept the code as it is, when it will be explained later.



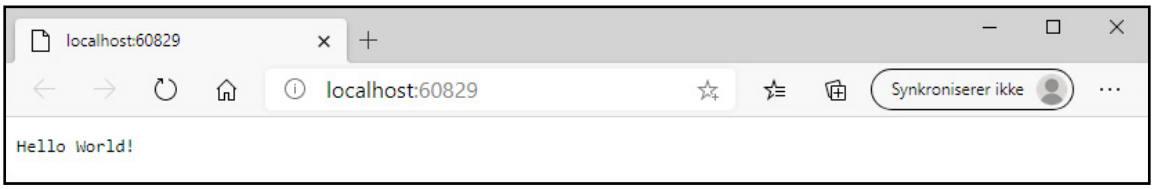
4 ABOUT ASP.NET CORE

In this chapter I will focus on the architecture of ASP.NET Core and what happens from the server get a request for an application and send a response to the user's browser. There are some details, and they are not all necessary to know for developing web applications, but conversely, an overall knowledge can be important, especially if one is developing an application and not everything is working as expected.

If you in Visual Studio creates a new (empty) web application called *AspNetCore1* you get an application with the following content:



The project contains only the most basic files, but sufficient for the project to be compiled and run:

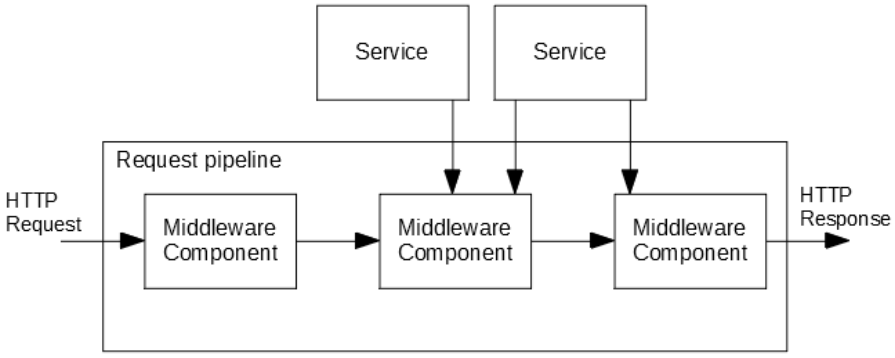


The user's request send from Visual Studio is send to a build-in web server running on the local machine, a server listening to port 60829. The response is quite simple and is only a string, but before the browser get the response a lot has actually happened.

The project has a class *Program.cs* and if you open the class it contains a usual *Main()* method and is as so the entry point. You will probably never need to change this class, but it is a regular main class and therefore you also has the option to change the code. The *Main()* method calls a static method which initializes and starts the web application. You should note that this is done using a generic method parameterized with the type *Startup*, and it is this method that uses the *Startup* class. It is this class that competes with the application and it is extremely central class in a web application and a class you often need to change.

When a web application receive a request the request is handled by one or more modules called a *middleware component*. These middleware components are arranged in a chain called the *request pipeline*, and from the request an object is created representing the request and

the response which is send to the first middleware component. Each component modify the response (or have the opportunity to do so) and send the object to the next middleware component in the chain, and the last component send the response to the browser. If the chain of middleware components does not generate a response ASP.NET Core sends a response with a HTTP Error 404 message. ASP.NET Core creates and manage objects which can be used by middleware components. Each of these objects are created on the basis of a class, and they often uses other classes called services. For now it is sufficient to think of a service as an object that can be used of middleware components.



The *Startup* class as created by Visual Studio has two methods:

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

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

    app.UseRouting();

    app.UseEndpoints(endpoints =>
    {
      endpoints.MapGet("/", async context => {
          await context.Response.WriteAsync("Hello World!");
        });
    });
  }
}
```

The first method is used to define the services used by the application. As default the method is empty and only default services are available. The other method is used to register middleware components, and there are three. The first middleware component handles unhandled exceptions, and the component should only be used when the application is developed. The next component is for URL routing, and the last defines the endpoints for the routing system. As default there is only one endpoint which send the text shown to the browser.

If you look at the class *Startup* in for example the project *ChangeAddress1* the code is:

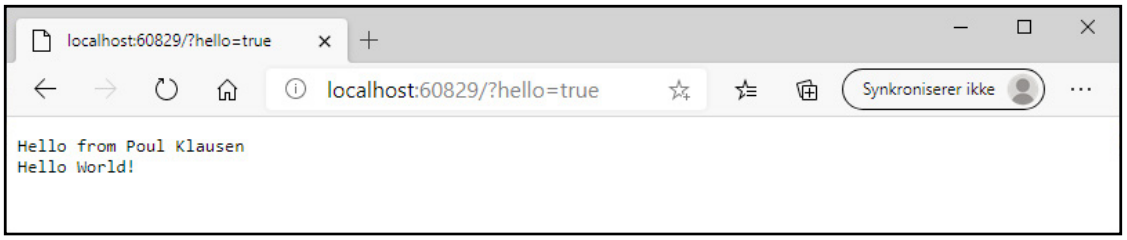
```
public class Startup
{
  public void ConfigureServices(IServiceCollection services)
  {
    services.AddControllersWithViews();
  }

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

    app.UseStaticFiles();
    app.UseRouting();

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

The method *ConfigureServices()* create one service. The project is a MVC project, and the service add services used by controllers. If you write an MVC web application this service is needed. The method *Configure()* configures four middleware components, where the first is used as mentioned above. The next makes it possible to use static elements such as images, stylesheets and more stored under the *wwwroot* folder. The last middleware component define the endpoint and tell that a request should be routed to the default controller which is the *HomeController*.

As an example I have added a new middleware component to *Startup* in the project *AspNetCore1*. It is a method with two parameters where the first is the object created for the request, while the other is a delegate used to send the request object *context* to the next component in the chain. The new middleware component is inserted into the request pipeline with the method *Use()*, and the parameter is the method representing the middleware component defined with a lambda expression. It all happens in the *Startup* class. In this case the method test if the *context* object is for a HTTP GET and if the request has a parameter called *hello* with the value *true*. If so the method add a text to the response.

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

  app.Use(async (context, next) => { if (context.Request.Method ==
  HttpMethods.Get
    && context.Request.Query["hello"] == "true")
    {
      await context.Response.WriteAsync("Hello from Poul Klausen\n");
    }
    await next();
  });

  app.UseRouting();

  app.UseEndpoints(endpoints =>
  {
    endpoints.MapGet("/", async context =>
         {
       await context.Response.WriteAsync("Hello World!");
     });
  });
}
```

If you execute the application there is nothing new, but if you in the browser's address line enter the parameter after the address and hit the *Enter* key you see that the new middleware component is used:



Of course, the example does not have many practical applications, but it shows a little about what a middleware component is and when it is performed, and here you should note that it is performed before the last one. Also you most note the *context* object that has the type *HttpContext*. The object has two properties with the types *HttpRequest* and *HttpResponse* which represent the request and the response, respectively.

The middleware component above is added using the method *Use()* and a lambda expression. Lambda is widely used in ASP.NET Core, but the result can easy be code which is unreadable. If you must write your own middleware components which are not entirely trivial you should probably instead write a class. In this case, I have added the following class:

```
public class MiddleClass
{
  private RequestDelegate next;

  public MiddleClass(RequestDelegate nextDelegate)
  {
    next = nextDelegate;
  }

  public async Task Invoke(HttpContext context)
  {
    if (context.Request.Method == HttpMethods.Get &&
      context.Request.Query["addr"] == "true")
    {
      await context.Response.WriteAsync("Knud den Hellige\n");
      await context.Response.WriteAsync("Voldgraven 3\n");
      await context.Response.WriteAsync("6666 Borremose\n");
    }
    await next(context);
  }
}
```

Note the syntax. The middleware component must then be added to *Startup* as follows where I have added it after the above component:

```
app.UseMiddleware<MiddleClass>();
```

If you then run the application and enter parameters in the address line in the browser the result is:

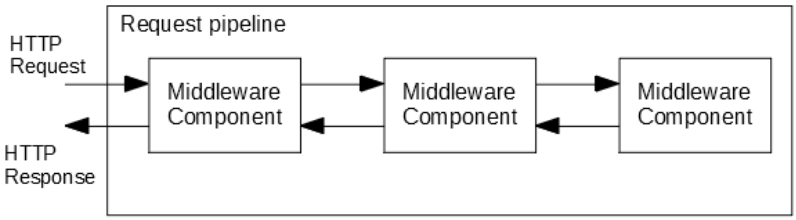
I have then added a statement to the first of the two middleware components:

```
app.Use(async (context, next) => { if (context.Request.Method ==
HttpMethods.Get
  && context.Request.Query["hello"] == "true")
  {
    await context.Response.WriteAsync("Hello from Poul Klausen\n");
  }
  await next();
  await context.Response.WriteAsync("\nDo you hear me?");
});
```

and here you must note that is after the *next()* statement. If you now run the application the result is:

Note that the new line is written after response from last middleware component. The reason is that the last component in the chain of middleware components sends its response back to the previous component and it is repeated for all components so that it is in fact the

response from the first component that is sent to the browser, but only after all components in the chain have had the opportunity to modify the result. A middleware component can thus update the context object on both the way out and the way back.



If you in the above browser window add parameters, the result is:

In particular, a middleware component can completely refrain to execute *next()*, thus interrupting the chain. If I set a comment in the above middleware component:

```
app.Use(async (context, next) => { if (context.Request.Method ==
HttpMethods.Get && context.Request.Query["hello"] == "true")
  {
    await context.Response.WriteAsync("Hello from Poul Klausen\n");
  }
//        await next();
  await context.Response.WriteAsync("\nDo you hear me?");
});
```

and run the application as shown:

you can see the request is not forwarded to the last two middleware components in the pipeline.

As shown above middleware components is a chain of components and they are used in the order determined by their position in the *Startup* class. However, it is possible to insert branches in the chain as you do with the *Map()* method. In *Startup* the following code inserted before the first middleware component defines a branch:

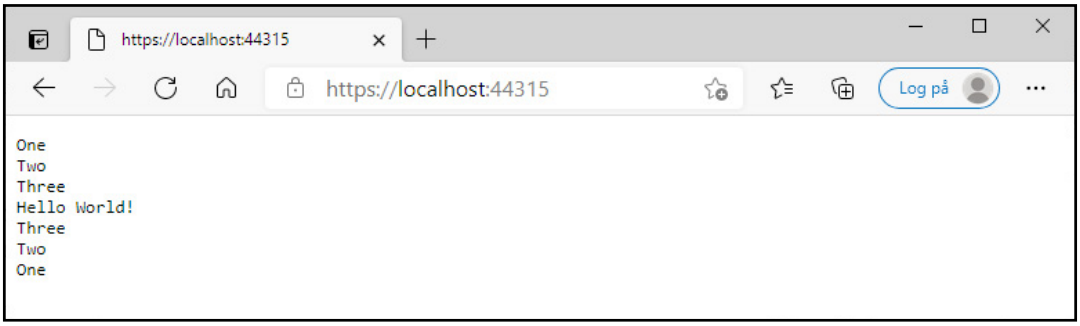
```
app.Map("/king", branch => {
  branch.UseMiddleware<MiddleClass>();
  branch.Use(async (context, next) => {
    await context.Response.WriteAsync($"Danish king");
  });
});
```

The first parameter is a string used to match an URL and in that case the chain of middleware components defined in the map is used, that is the middleware component defined in the class *MiddleClass* followed by the component defined by the lambda expression. If you run the application and enter the parameter *addr* the result is:

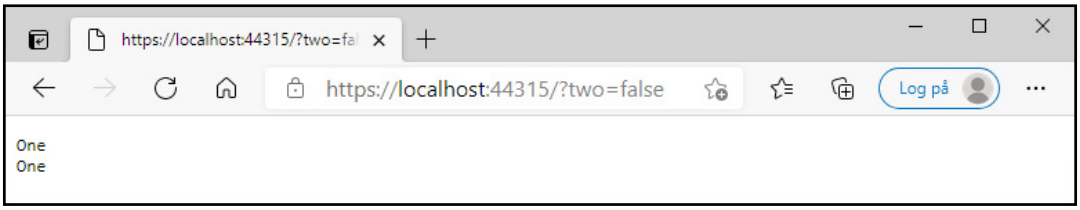
and it all looks at expected and it is the usual chain of components that are used, but if you enter *king* as URL it is the other branch of components which is used.

Exercise 4

Create a new ASP.NET core web application. If you run the application is must write the following text om the screen:



The word *One* and the word *Three* should be written by two middleware components written as lambda expressions in the class *Startup*. The word *Two* should be written by a middleware component written as a class. If the request has a parameter *two* with the value *false* this component should break the request pipeline and the result should be:



4.1 URL ROUTING

As shown in the above section a request is handle through a chain of middleware components, and the last of these components is called an endpoint and is the component which is responsible to handle the request to the browser. When an application receive a request the application must send the request to the right request pipeline which end with the right endpoint and this chain is called a route. The routing system defines patterns for how incoming request should be dispatched to endpoints, and the system has so two main functions:

1. Maps incoming URLs to controllers and action methods
2. Generate outgoing URLs such a specific action methods is called when the user clicks on a link

An application must then have a need to support many routes, and the system to map URLs to endpoints is what is called URL Routing. This mapping happens in the *Startup* class, and is not quite simple. Even if you do not need to know the technical details described below in practice, it can still be good to know a little about the details, partly because the routing in practice does not always go as expected.

To show some of the details I start with a new project *AspNetCore2*. To this project I have added a class which for five countries that maps the country name to the number of inhabitants:

```
public class Inhabitants
{
  private RequestDelegate next;

  public Inhabitants() { }

  public Inhabitants(RequestDelegate nextDelegate)
  {
    next = nextDelegate;
  }

  public async Task Invoke(HttpContext context)
  {
    string[] parts = context.Request.Path.ToString().ToLower().
      Split("/", StringSplitOptions.RemoveEmptyEntries);
    if (parts.Length == 2 && parts[0].Equals("inhabitants"))
    {
      string country = parts[1];
      int? number = null;
      switch (country)
      {
        case "norway": number = 5367580; break;
        case "sweden": number = 10065389; break;
        case "finland": number = 5509717; break;
        case "iceland": number = 364260; break;
        case "denmark": number = 5822763; break;
      }
      if (number.HasValue)
      {
        await context.Response.WriteAsync(
          $"Country: {country}, Inhabitants: {number}");
        return;
      }
    }
    if (next != null) await next(context);
  }
}
```

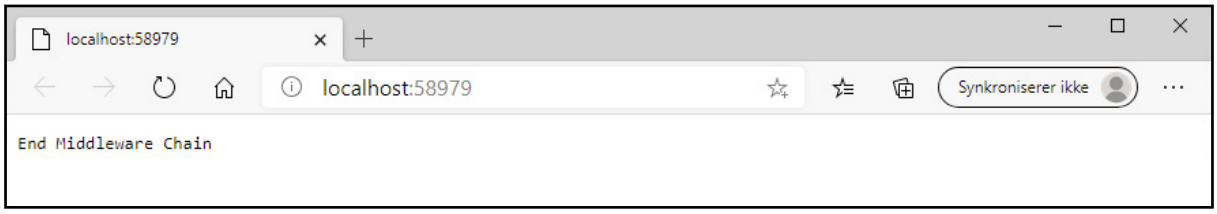
The class is in principle simple, but you must note that it defines a middleware component. The method *Invoke()* splits the path of the URL address into parts and if there are two parts and if the first has the value *inhabitants* the method try to match the other part with one of the five country names. If there is a match the method write a message to the response and return. That is the component acts as an endpoint. Else the method forwards the request to next component in the chain if there is a *next*.

The program also has a class *Area* which works in exactly the same way with only one difference, when the class maps the five country names to there areas. I will not show the code for this class here.

Next I have updated *Startup*:

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

  public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
  {
    app.UseDeveloperExceptionPage();
    app.UseMiddleware<Inhabitants>();
    app.UseMiddleware<Area>();
    app.Use(async (context, next) => {
      await context.Response.WriteAsync("End Middleware Chain");
    });
  }
}
```

The are three middleware components in the chain, one for the class *Inhabitants*, one for the class *Area* and a last one which write a message as a response. If you run the application the result is:



You see no effect of the two first components as they both do there default action and forwards the request to the next component where it end with the component defined in *Startup* using the lambda expression.

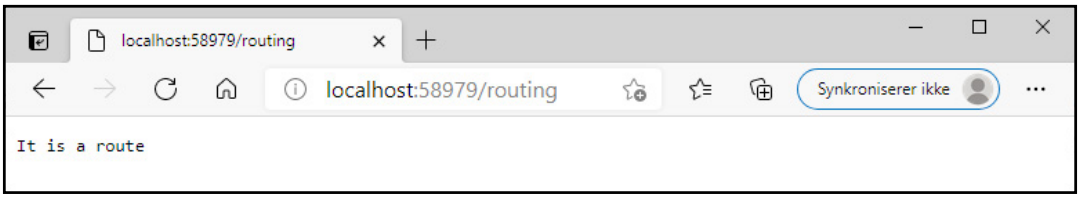
If you enter the URL as shown below the middleware component *Inhabitants* results in a match and write its message as response, and as the method does not forward the request the chain ends with this component.

If, on the other hand, you enter the URL as below there is no match in the first two components and it is again the last component which ends the chain

Each middleware component decides whether to act on a request. The component may look for a specific header or value of a query string, but most components try to match an URL. Each middleware component has to repeat the same set of steps as the request works its way along the pipeline and both middleware components in this example do. It is inefficient and is difficult to maintain, and URL routing solves these problems by introducing middleware that takes care of matching request URLs so that components, called *endpoints*, can focus on responses. The mapping between endpoints and the URLs they require is expressed in a *route*. The routing middleware processes the URL, inspects the set of routes and finds the endpoint to handle the request, a process known as *routing*. The used middleware is added using the methods *UseRouting()* and *UseEndpoints()*. The first method adds the middleware responsible for processing requests to the pipeline while the other is used to define the routes that match URLs to endpoints. URLs are matched using patterns that are compared to the path of the requested URL, and each route creates a relationship between one URL pattern and one endpoint. An example could be:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
  app.UseDeveloperExceptionPage();
  app.UseMiddleware<Inhabitants>();
  app.UseMiddleware<Area>();

  app.UseRouting();
  app.UseEndpoints(endpoints =>
  {
    endpoints.MapGet("routing", async context =>
    {
      await context.Response.WriteAsync("It is a route);
    });
  });

  app.Use(async (context, next) => {
    await context.Response.WriteAsync("End Middleware Chain");
  });
}
```

The configuration now tells to use URL routing and there is defined one endpoint which is mapped to the URL path *routing* for a HTTP GET. If you run the application and enter the the URL you can see that it is the mapped endpoint that write the response:



If you enter something else which not map to *routing* the default endpoint is reached:

Also note that if you enter an URL for one of the two first components they are still reached as they are before the routing middleware component:

The two middleware components can also be added as endpoints:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
  app.UseDeveloperExceptionPage();

  app.UseRouting();
  app.UseEndpoints(endpoints =>
  {
    endpoints.MapGet("routing", async context =>
    {
      await context.Response.WriteAsync("It is a route");
    });
    endpoints.MapGet("area/iceland", new Area().Invoke);
    endpoints.MapGet("inhabitants/norway", new Inhabitants().Invoke);
  });

  app.Use(async (context, next) => {
    await context.Response.WriteAsync("End Middleware Chain");
  });
}
```

If it should be possible to match all patterns defined in the two middleware classes that way 10 endpoints are needed and there is therefore a need for a more flexible system. For that *segment variables* or *route parameters* are defined:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
  app.UseDeveloperExceptionPage();

  app.UseRouting();
  app.UseEndpoints(endpoints =>
  {
    endpoints.MapGet("{one}/{two}/{three}", async context =>
    {
      await context.Response.WriteAsync("Rout found\n");
      foreach (var part in context.Request.RouteValues)
      {
        await context.Response.WriteAsync($"{part.Key}: {part.Value}\n");
      }
    });
    endpoints.MapGet("routing", async context =>
    {
      await context.Response.WriteAsync("It is a route");
    });
    endpoints.MapGet("area/iceland", new Area().Invoke);
    endpoints.MapGet("inhabitants/norway", new Inhabitants().Invoke);
  });

  app.Use(async (context, next) => {
    await context.Response.WriteAsync("End Middleware Chain");
  });
}
```

The new endpoint has a pattern which consists of three variables. The pattern matches URL's with three segments (parts separated by /) and that regardless of the content of the three segments. These values are available through a property *RouteValues* of the *HttpRequest* object which is a dictionary, and if you run the application and enter a URL with three segments the result is:

If I add another endpoint

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
  app.UseDeveloperExceptionPage();

  app.UseRouting();
  app.UseEndpoints(endpoints =>
  {
    endpoints.MapGet("{one}/{two}/{three}", async context =>
    {
      await context.Response.WriteAsync("Rout found\n");
      foreach (var part in context.Request.RouteValues)
      {
        await context.Response.WriteAsync($"{part.Key}: {part.Value}\n");
      }
    });
    endpoints.MapGet("there/are/three", async context =>
    {
      await context.Response.WriteAsync("Three Danisk kings");
    });
    endpoints.MapGet("routing", async context =>
    {
      await context.Response.WriteAsync("It is a route");
    });
    endpoints.MapGet("area/iceland", new Area().Invoke);
    endpoints.MapGet("inhabitants/norway", new Inhabitants().Invoke);
  });

  app.Use(async (context, next) => {
    await context.Response.WriteAsync("End Middleware Chain");
  });
}
```

and you should note that the new endpoint is added after the one with the three variables, and if I enter:

it is the new endpoint that is reached. It tells that patterns based on constant strings have higher priority than patterns based on variables. In fact, the choice of route is quite complex and is based on a system where the individual routes are given a priority and where the route with the highest priority wins.

If you look at the above classes *Inhabitants* and *Area* they are middleware classes, but they are in fact not used as such and only as endpoints. When a middleware class should only be used as an endpoint class it can be written simpler and as an example is shown a class

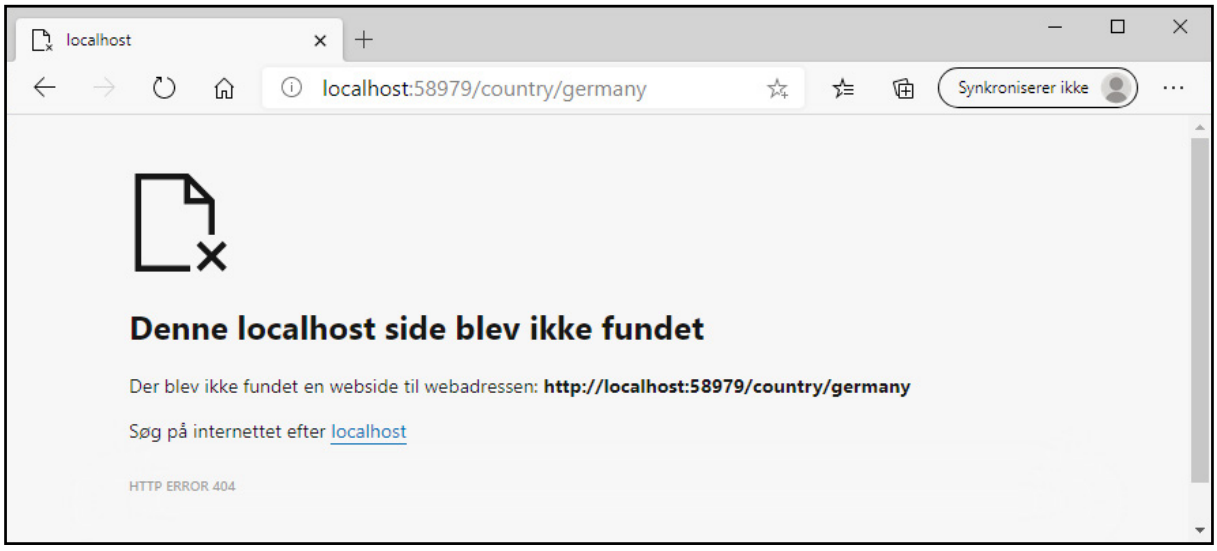
```
public class Country
{
  public static async Task Endpoint(HttpContext context)
  {
    string country = (string)context.Request.RouteValues["country"];
    int? t1 = null;
    int? t2 = null;
    switch (country)
    {
      case "norway":
        t1 = 5367580;
        t2 = 385208;
        break;
      case "sweden":
        t1 = 10065389;
        t2 = 450295;
        break;
      case "finland":
        t1 = 5509717;
        t2 = 338424;
        break;
      case "iceland":
        t1 = 364260;
        t2 = 103001;
        break;
      case "denmark":
        t1 = 5822763;
        t2 = 43094;
        break;
    }
    if (t1.HasValue) await context.Response.WriteAsync(
      $"<table><tr><td>Country</td><td>{country}</td></
      tr><tr><td>Inhabitants</td>
      <td>{t1}</td></tr><tr><td>Area</td><td>{t2}</td></tr></table>");
    else context.Response.StatusCode = StatusCodes.Status404NotFound;
  }
}
```

The class has only one static method which has a *HttpContext* object as parameter. Note how the class use the dictionary *RouteValues* to find a value for a segment *country*. If so it initialize two variables and set the response to a HTML table with the result. If there is no match the methods returns a response with an error message.

You can insert an endpoint in *Startup* as

```
endpoints.MapGet("country/{country}", Country.Endpoint);
```

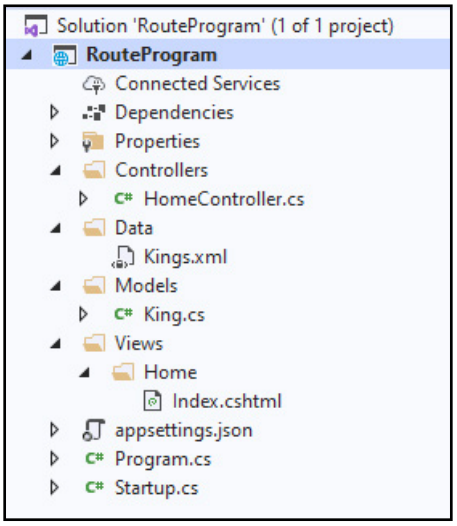
and if you run the application and enter an URL for a country the result could be:



Exercise 5

In this exercise you should write a very simple MVC application with four views and two controllers and it must be possible to navigate between the four views. The purpose is to show how the navigation can be moved from the code to routes in the *Startup* class.

Start with a new *ASP.NET Core Web Application* project which I have called *RouteProgram*. Creates a default MVC architecture as shown below, where the XML document *Kings.xml* and the class file *King.cs* are from exercise 1. In the last file you must remember to change the *namespace*.



You must also update *Startup.cs* for a MVC application:

```
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();
    });
  }
}
```

You can consider this version of *Startup* as the default version for an MVC application. You should solve the exercise through the following steps.

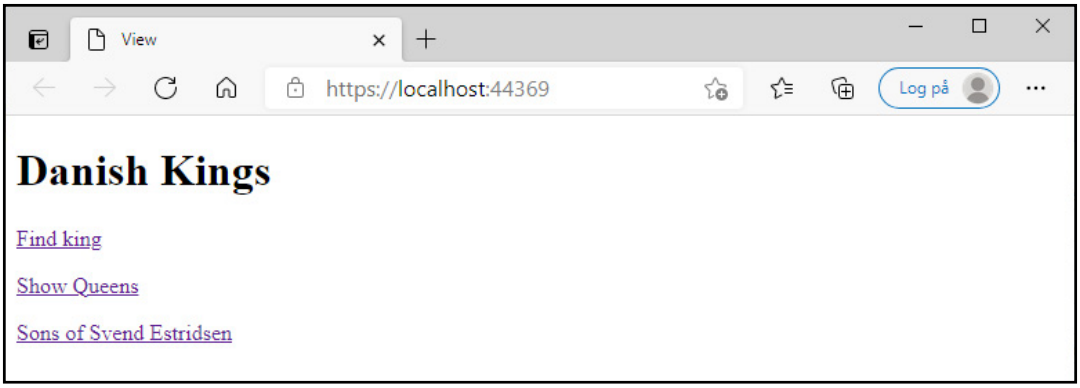
1) Add a *_ViewImports.cshtml* to the *Views* folder with the following content

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

2) Add a new controller, which you can call *ShowController*. Add also a sub folder to *Views* which you must call *Show*.

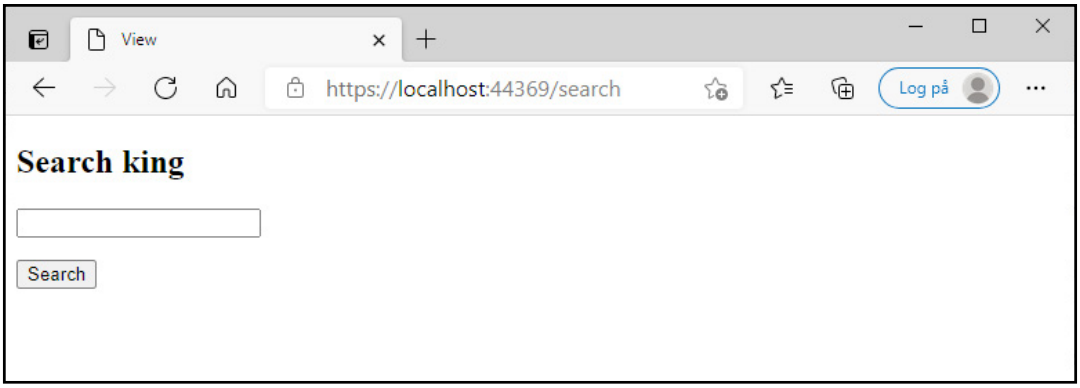
3) When the program start it should open the following browser showing the default view:

```
/Views/Home/Index.cshtml
```



The view shows a header and three links.

4) If the user clicks on the link *Find king* the program shows a simple form:



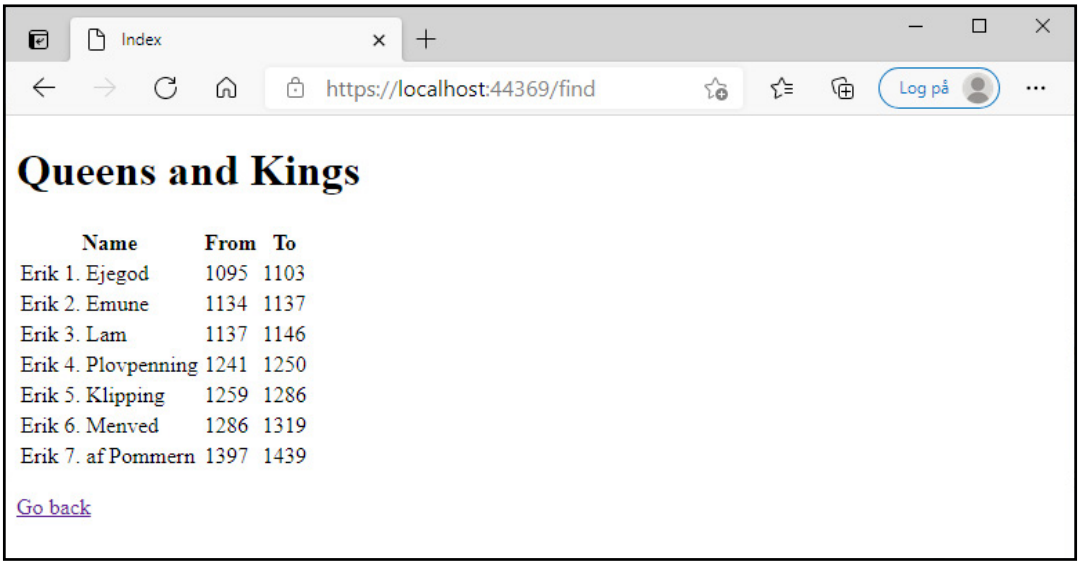
You can write the link as:

```
<p><a asp-action="Search" asp-controller="Home">Find king</a></p>
```

It means that the *Home* controller must have an action method for the link and you must add a view for the form:

```
/Views/Home/Search.cshtml
```

5) If the user enter a text and submit the form the program must open a view showing all kings where the name starts with the entered test. An example could be where the user has entered *Erik*:



The view should be called *Index* and be a view for an action in the *Show* controller:

```
/Views/Show/Index.cshtml
```

The action can be written as:

```
[HttpPost]
public ViewResult Index(SearchViewModel model)
{
  Kings kings = new Kings();
  IEnumerable<King> list = kings.Where(k => k.Name.StartsWith(model.
  Name));
  return View(list);
}
```

This action must be called from the form to enter the search text:

```
<form asp-action="Index" asp-controller="Show" method="post">
```

To return the search text I have created a simple model class called *SearchViewModel* (se the parameter for the action method) which has only one property for a string.

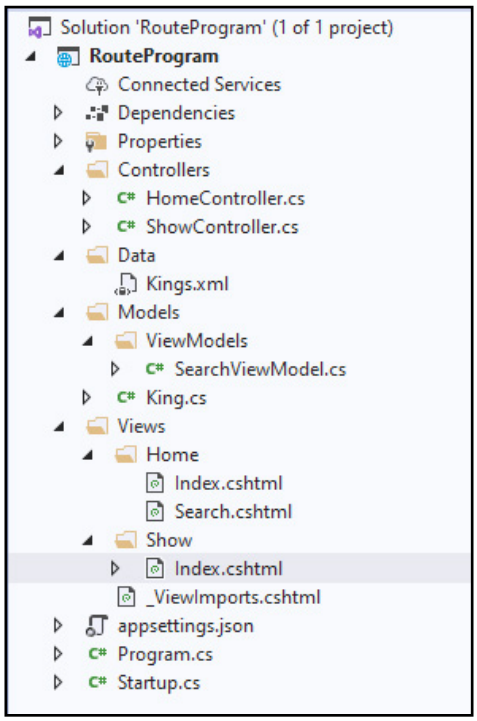
6) The second link on the start page should open a view with the Danish queens (there are two: *Margrethe 1.* and *Margrethe 2.*). To implement this feature the *Show* controller must have another action method:

```
public ViewResult Queens()
{
  Kings kings = new Kings();
  IEnumerable<King> list = kings.Where(k => k.Name.StartsWith("Margre"));
  return View("Index", list);
}
```

The link on the start view can be written as:

```
<a asp-action="Queens" asp-controller="Show">Show Queens</a></p>
```

7) The function for the last link can be implemented in exactly the same way and the the application is in principle finish, and solution has the following files:



8) To navigate the application it has five links:

- three links in *Views/Home/Index.cshtml*
- one link in *Views/Show/Index.cshtml* to go back to the start page
- one link in *Views/Home/Search.cshtml* in the form tag

All links are hardcoded which is fine enough, but it makes it harder to maintain the program. The problem can be solved replacing the links with patterns:

```
<a href="/search">Find king</a>
<a href="/queens">Show Queens</a>
<a href="/sons">Sons of Svend Estridsen</a>
<a href="/">Go back</a>
<form action="/find" method="post">
```

Then *Startup* must be updated with routes for this endpoints:

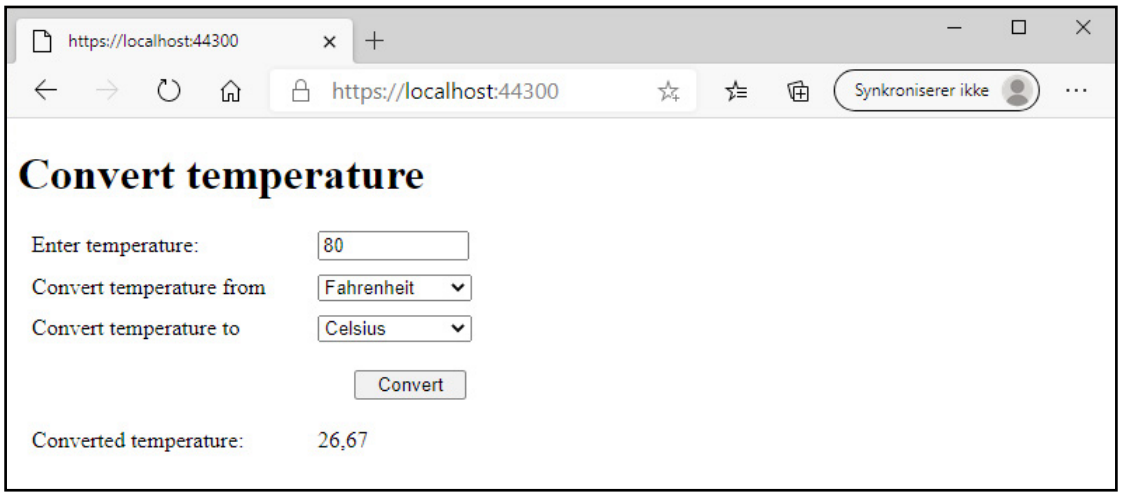
```
app.UseEndpoints(endpoints =>
{
  endpoints.MapControllerRoute(name: "search", pattern: "/search",
    defaults: new { controller = "Home", action = "Search" });
  endpoints.MapControllerRoute(name: "queens", pattern: "/queens",
    defaults: new { controller = "Show", action = "Queens" });
  endpoints.MapControllerRoute(name: "sons", pattern: "/sons",
    defaults: new { controller = "Show", action = "Sons" });
  endpoints.MapControllerRoute(name: "find", pattern: "/find",
    defaults: new { controller = "Show", action = "Index" });
  endpoints.MapDefaultControllerRoute();
});
```

Change your project as describe above and test at it all works.

4.2 A RAZOR PAGE

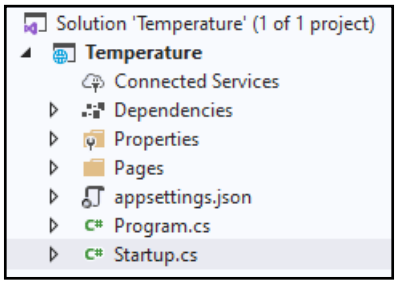
The examples in chapter 2 and 3 and also the last exercise have all followed an MVC pattern and in most cases it will be the right path, but there are other ways, and for a simple application with a single page or two can the MVC pattern seems more complicated than necessary. This section will therefore show you how to write an application that consists of a Razor page.

As an example I will show an application with only a start page where the user can enter a temperature and select where to convert from and to:



The user can enter a temperature in the upper field and convert the value shown below the button.

To write the program I create an empty ASP.NET core Web Application. The page should this time be a Razor page and I add a folder *Pages* to the project



where Razor pages as default are stored. Before I continue I must prepare the application to use Razor pages and modify the *Startup* class:

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

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

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

That is the application must configure a service for Razor pages, add the middleware component for URL routing and define an endpoint for a default Razor page. It is a page with the name *Index* in the folder *Pages*.

Next I add three files to the *Pages* folder

1. a *Razor View Imports* file: *_ViewImports.cshtml*
2. a *Razor View Start* file: *_ViewStart.cshtml*
3. a *Razor Layout* file: *_Layout.cshtml*

Note the file names are the same as are used for an MVC application. The content of the first file is:

```
@namespace Temperature.Pages
@using Microsoft.AspNetCore.Mvc.RazorPages
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

where *Temperature* is the name of the project. The content of the second file is:

```
@{
    Layout = "_Layout";
}
```

and then the same as for a MVC application (see the next chapter). Also the last file is the same as for a MVC application and is used to define the overall design:

```
<!DOCTYPE html>

<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>@ViewBag.Title</title>
</head>
<body>
  <div><h1>Convert temperature</h1></div>
    <div>
      @RenderBody()
    </div>
</body>
</html>
```

In this case I have added a single *H1* element just to do something. To create the page itself I add a *Razor Page* and select the name *Index.cshtml* (which Visual Studio selects as default). When I clicks *Add* and Visual Studio creates a page with the name *Index.cshtml*, but also a C# class file *Index.cshtml.cs* which is the model for the Razor page. I must then write the model:

```
public class IndexModel : PageModel
{
  Required(ErrorMessage = "You must enter a temperature")]
  [RegularExpression("...", ErrorMessage = "...")]
  [BindProperty]
  public string Value { get; set; }
  public string Result { get; private set; }
  [BindProperty] public Unit ConvertFrom { get; set; }
  [BindProperty] public Unit ConvertTo { get; set; }
  public List<SelectListItem> Units { get; } = SelectList.
  Create<Unit>();

  public void OnPost()
  {
    ...
  }
}

public enum Unit { Celsius, Fahrenheit, Kelvin }
```

The model class inherits a class *PageModel* which is the base class for a Razor Page. In this case the model define 5 properties. The first property called *Value* is decorated for validation where it must have a value that must be a legal *double* number (the regular expression is not shown above). The property is also decorated as *BindProperty* which means that the property is updated with values from the view. Other two properties are also decorated with *BindProperty*, and the type of these properties is an *enum* defined last in the file. The last property defines a list used to define a *selection* element (a dropdown list) in the view. The type of the elements are *SelectListItem* and the elements for the dropdown list must be converted to that type. This is done using a static method in a class *SelectList* which is not shown.

With the model class ready the view can be written as:

```
@page
@model Temperature.Pages.IndexModel

<form method="post">
  <div style="margin: 10px 0px 10px 10px">
    <label asp-for="Value" style="width: 200px;display:inline-block">
      Enter temperature:</label>
    <input type="text" asp-for="Value" style="width:100px" />
    <span asp-validation-for="Value" style="color:red"></span>
  </div>
  <div style="margin: 10px 0px 10px 10px">
    <label asp-for="ConvertFrom" style="width: 200px;display:inline-
    block">
      Convert temperature from</label>
    <select asp-for="ConvertFrom" asp-items="Model.Units" style="width:
    110px">
    </select>
  </div>
  <div style="margin: 10px 0px 10px 10px">
    <label asp-for="ConvertTo" style="width: 200px;display:inline-
    block">
      Convert temperature to</label>
    <select asp-for="ConvertTo" asp-items="Model.Units" style="width:
    110px">
    </select>
  </div>
  <div style="margin: 20px 0px 20px 10px">
    <button type="submit" style="width:80px;margin-left:
    230px">Convert</button>
  </div>
  <div style="margin: 10px 0px 10px 10px">
    <label asp-for="Result" style="width: 200px;display:inline-block">
      Converted temperature:</label>
    @Model.Result
  </div>
</form>
```

There is not much new, but you must note the first two directives, which defines that this view is a view for a Razor Page and the model for this view. You should also note that I do not use a style sheet, but the styles are written directly as an attribute in the elements, just to show how.

As the last note that the view is not a full HTML page, but just some HTML. A request for this application is in *Startup* routed to the default Razor Page, that is to */Pages/Index.*

cshtml. When this file is not a full HTML page the Razor engine will search a default layout page with the name defined in *_ViewStart.cshtml* which here is *_Layout.cshtml*. The code in *Index.cshtml* is then inserted for the placeholder *RenderBody()* in the layout file.

In general, I prefer the MVC pattern over just a Razor Page as above, but this pattern certainly has its uses as well. You should note that in an application you can combine the use of both patterns if the necessary services are configured in the *Startup* class.

Exercise 6: Calculation

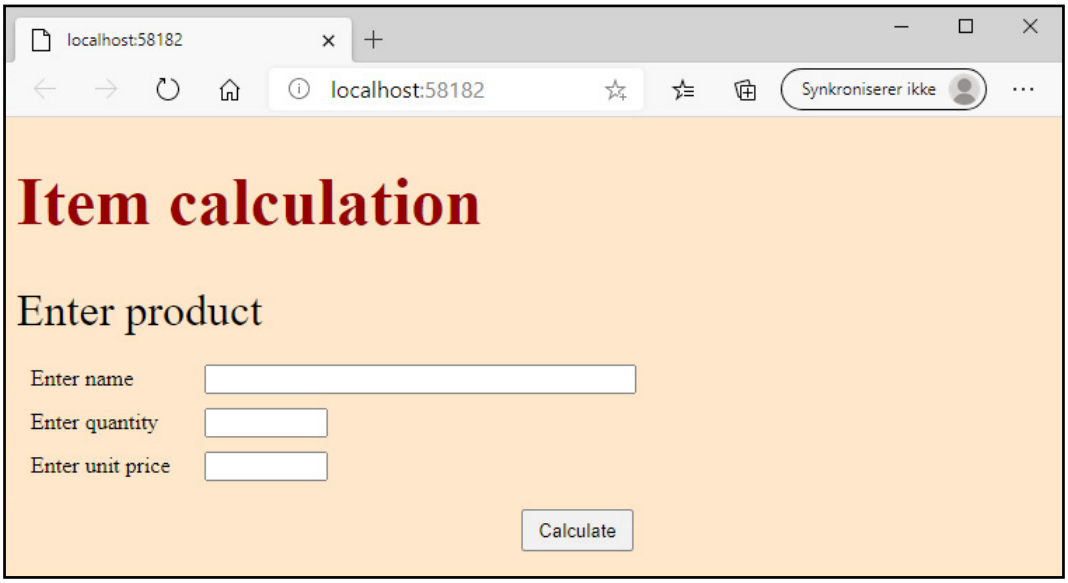
You must write an application where the user must enter a name, number of units and unit price for a product. When the user submit the page the program must show another page as an order confirmation and showing the total price.

Both pages must be Razor pages as in the above example and you should style the individual elements using inline styles.

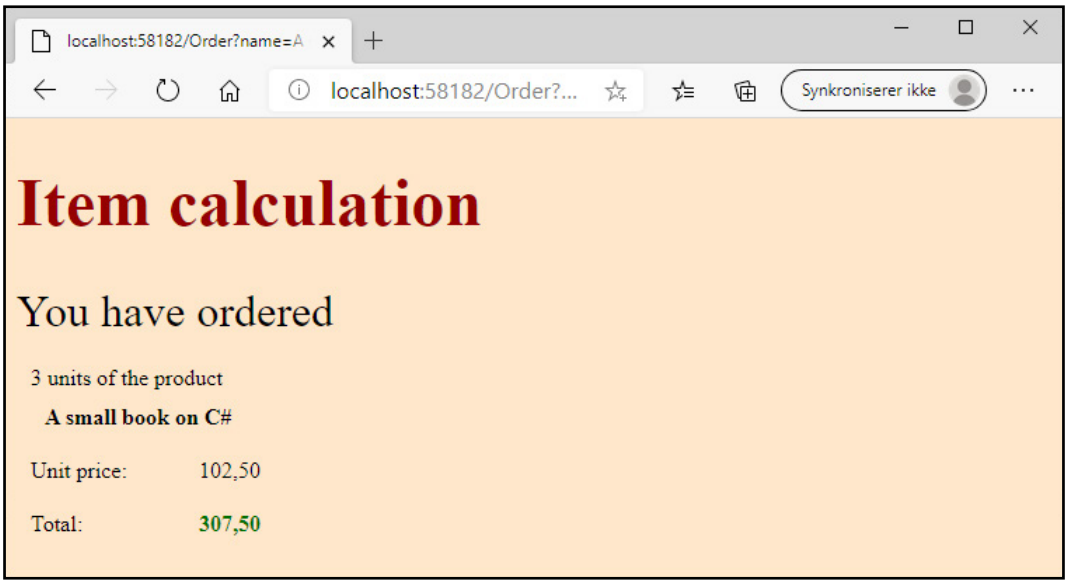
When the application starts it should show a page like shown below. The three fields must be validated in the model such

1. the name field must have a value
2. the quantity field must have a value which must be a positive *int*
3. the price field must have a value which must be a positive *double*

If the one or more of the fields are invalid and the user clicks *Calculate* the program must show error messages to inform the user what is wrong.



If the user enter legal values and clicks *Calculate* the program most open a result page:



To write the application you should follow the guidelines from the above example, but this times there must be two Razor pages where you can call the other for *Order*. There are two challenges:

1. the data entered in the page *Index* must be transferred to the page *Order*
2. the page *Order* must show the transferred data

When the user perform a submit the program performs an *OnPost()* method in the model. This time you should write the method as:

```
public IActionResult OnPost()
{
  if (ModelState.IsValid) return RedirectToPage("Order", "Order",
    new { name = Name, number = Quantity, price = Price });
  else return Page();
}
```

If the model is legal (the properties has legal values) the method performs a *redirect* which means it makes a HTTP GET to a page and in this case to the page *Order*. Else it returns to the current page. The method *RedirectToPage()* has three parameters where the first is the name of the page to redirect to, the second is the name of a *DoGet()* in the model for *Order* while the last is an object with three properties for the values which should be transferred to *Order*. The get method in *OrderModel* could be written as:

```
public void OnGetOrder(string name, string number, string price)
{
  string ds = NumberFormatInfo.CurrentInfo.CurrencyDecimalSeparator;
  int quantity = int.Parse(number);
  double amount = double.Parse(price.Trim().Replace(".", ds));
  ViewData["order"] = new Order { Name = name, Quantity = quantity.
  ToString(),
    Price = string.Format("{0:F2}", amount),
    Total = string.Format("{0:F2}", amount * quantity) };
}
```

ViewData is a dictionary which can be used to transfer data from the model to then content of a Razor page. In this is transferred an object of the type *Order* which is an inner (in this case) type in the model. The dictionary can be used in the content as:

```
@page
@model Calculation.Pages.OrderModel
@{
  var order = (OrderModel.Order)ViewData["order"];
}
```

and you have an object *order* representing the transferred data.

4.3 SERVICES

In this section, I will address the concept of services. A service is just a class and overall, it is about how to create and consume services, which are objects that can be shared by the various middleware components. This means that a service object must be created and then be available to all middleware components and also that components can find the service objects.

One way is to create service objects as singletons, and if the middleware components know where the service is stored it is immediately available. This approach definitely has its uses, but also has drawbacks as a singleton is relatively static and it can be difficult to replace one service with another. This is the challenge that dependency injection has to solve and is the subject for the next section, but I want to start with singletons as it is a very simple and familiar pattern.

I start with a project *AspNetCore3*, and here I creates a folder *Services* which is the place to store services. Next I add two services. The first is a simple service representing a random generator, and you can think of the service as an object to generate random numbers and an object which is available throughout the application. The service is just a class which encapsulates a *Random* object, but written as a singleton:

```
public class RandomService
{
  private static RandomService instance = null;

  private Random rand = new Random();

  private RandomService()
  {
  }

  public static RandomService Instance
  {
    get
    {
      if (instance == null) instance = new RandomService();
      return instance;
    }
  }

  public Random Rand { get => rand; }
}
```

The next service represents a simple counter and is defined by an interface:

```
public interface ICounter
{
  public string Value { get; }
  public void Up();
  public void Down();
  public void Square();
}
```

The following class implements the interface and as such a concrete service:

```
public class SmallCounter : ICounter
{
  private static SmallCounter instance = null;

  private long counter = 0;

  private SmallCounter()
  {
  }

  public static SmallCounter Instance
  {
    get
    {
      if (instance == null) instance = new SmallCounter();
      return instance;
    }
  }

  public string Value { get => "" + counter; }

  public void Up()
  {
    ++counter;
  }

  public void Down()
  {
    --counter;
  }

  public void Square()
  {
    counter *= counter;
  }
}
```

The class is quite simple and there is nothing to explain. Next I will use the services in a middleware component. You must note the services are implemented as a singletons and can then be used from all middleware components. The class *SmallCounter* implements an interface, but when the class is a singleton it can not be referenced through this interface, but you must everywhere use the class name to reference the service. If you want to use another implementation of the interface you have to change everywhere where the service is used. Some times these problems are solved using a broker, which is just a class that using a static method returns an *ICounter* object:

```
public static class CountBroker
{
  public static ICounter Counter => BigCounter.Instance;
}
```

The middleware component can then be written as:

```
public class MiddleUp
{
  private RequestDelegate next;

  public MiddleUp(RequestDelegate next)
  {
    this.next = next;
  }

  public async Task Invoke(HttpContext context)
  {
    int n = RandomService.Instance.Rand.Next(1, 10);
    while (n-- > 0) CountBroker.Counter.Up();
    await context.Response.WriteAsync($"{CountBroker.Counter.Value}\n");
    await next(context);
  }
}
```

The component is absolutely trivial, but it uses both services and increase the counter a random numbers of times after which it write the value of the counter to the response. You should note that the *SmallCounter* service is referenced using the broker. The program also has another middleware component which works in the same way, but decreases the counter:

```
public class MiddleDown
{
  private RequestDelegate next;

  public MiddleDown(RequestDelegate next)
  {
    this.next = next;
  }

  public async Task Invoke(HttpContext context)
  {
    int n = RandomService.Instance.Rand.Next(1, 10);
    while (n-- > 0) CountBroker.Counter.Down();
    await context.Response.WriteAsync($"{CountBroker.Counter.Value}\n");
    await next(context);
  }
}
```

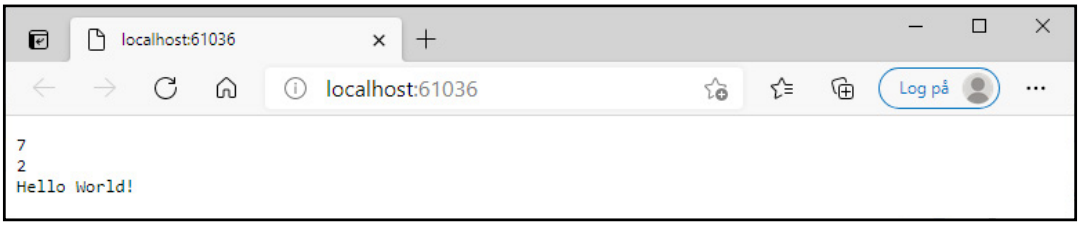
The next class also represents a middleware component, but this time implemented as an endpoint component:

```
public class EndSquare
{
  public static async Task Endpoint(HttpContext context)
  {
    CountBroker.Counter.Square();
    await context.Response.WriteAsync($"{CountBroker.Counter.
    Value}\n");
  }
}
```

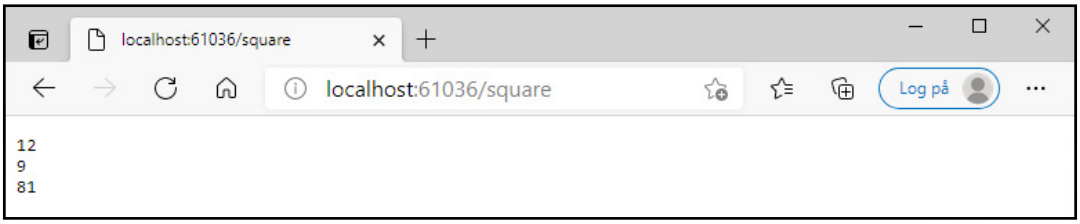
The next is to try it all from *StartUp*:

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

  public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
  {
    app.UseDeveloperExceptionPage();
    app.UseRouting();
    app.UseMiddleware<MiddleUp>();
    app.UseMiddleware<MiddleDown>();
    app.UseEndpoints(endpoints =>
    {
      endpoints.MapGet("/square", EndSquare.Endpoint);
      endpoints.MapGet("/", async context =>
      {
        await context.Response.WriteAsync("Hello World!");
      });
    });
  }
}
```

If you run the application the result could be:



You can see that the two middleware components are used as they are on the request pipeline. The first has increased the counter seven times where the other has decreased the timer five times. The endpoint *EndSquare* is not used as there is no route for the endpoint. Instead, the default endpoint is used. If you enter the route in the browser address line and push *Enter* the result could be:



Here you should note two things: The endpoint *EndSquare* is used, but also that it is the same *SmallCounter* object that is used. You can see this as the middleware component *MiddleUp* only counts to max 9. If you refresh your browser several times, you will notify the same. It means that implementing a service as a singleton the object is created the first time the property *Instance* is referenced, but the object lives until a new request for the application, and if you implement services as a singleton it is something to be aware of.

Above the service implements the interface *ICounter* and using the broker it means that the service everywhere is known as an *ICounter*. If you implement another counter:

```
public class BigCounter : ICounter
{
  private static BigCounter instance = null;

  private BigInteger counter = 0;

  private BigCounter()
  {
  }

  public static BigCounter Instance
  {
    get
    {
      if (instance == null) instance = new BigCounter();
      return instance;
    }
  }

  public string Value { get => counter.ToString(); }

  public void Up()
  {
    ++counter;
  }

  public void Down()
  {
    --counter;
  }

  public void Square()
  {
    counter *= counter;
  }
}
```

the only thing you need to use this counter instead of the *SmallCounter* service is change the *CountBroker* class.

Exercise 7

Create copy of the project from exercise 6, and you can call the project for *RouteProgram1*. In the file *Kings.cs* you must

1. Create an interface *IKings* which define the class *Kings*. The interface should inherits *IEnumerable<King>* and define two properties.
2. The class *Kings* should implement this interface.
3. Modify the class *Kings* such it is a singleton.
4. Create a broker class that has a property for a *IKings* object.

Build the application. The result should be three errors in the *ShowController*. Remove the errors using the broker, at the application should run again.

4.4 DEPENDENCY INJECTION

Instead of writing services as singletons you can use dependency injection, a technique that is more flexible, but dependency is much more and is a technique built-in to ASP.NET Core. Basically, it is a matter of ASP.NET Core being able to instantiate an object for a service based on constructors parameters. It makes it more flexible to set up services that way than to define singletons and broker classes.

I start with a new project called *AspNetCore4* and create a folder *Services*. To this folder I add an interface that defines a service:

```
public interface ITimeFormatter
{
  Task Format(HttpContext context, string content);
}
```

It should be a very simple service which just add some text to the response. An example could be:

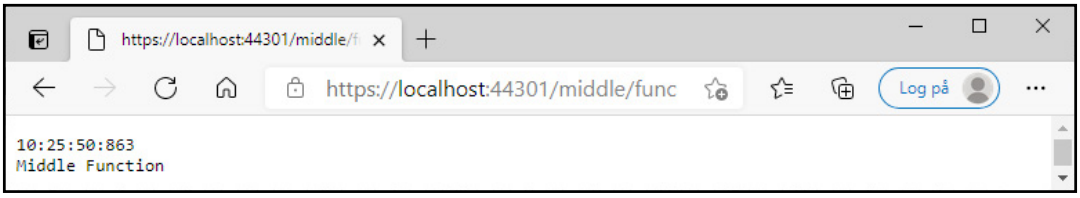
```
public class TimeFormatter : ITimeFormatter
{
  private DateTime dt = DateTime.Now;

  public async Task Format(HttpContext context, string content)
  {
    await context.Response.WriteAsync($"{string.Format(
      "{0,2:D2}:{1,2:D2}:{2,2:D2}:{3,3:D3}",
      dt.Hour, dt.Minute, dt.Second, dt.Millisecond)}\n{content}");
  }
}
```

Note that the service is not written as a singleton.

The service can be used in *Startup* as, where the service is used in a middleware component defined as a function:

```
public class Startup
{
  public void ConfigureServices(IServiceCollection services)
  {
    services.AddSingleton<ITimeFormatter, TimeFormatter>();
  }

  public void Configure(IApplicationBuilder app, IWebHostEnvironment env,
    ITimeFormatter formatter)
  {
    app.UseDeveloperExceptionPage();
    app.UseRouting();
    app.Use(async (context, next) =>
    {
      if (context.Request.Path == "/middle/func")
      {
        await formatter.Format(context, "Middle Function");
      }
      else
      {
        await next();
      }
    });
    app.UseEndpoints(endpoints => {
      endpoints.MapGet("/", async context =>
      {
        await context.Response.WriteAsync("Hello World!");
      });
    });
  }
}
```

A service must be configured or registered which happens in the method *ConfigureServices()*. Here the service is registered as an *ITimeFormatter* service of the type *TimeFormatter*. To use the service in the middleware function it must be known in the method *Configure()*, and the method is expanded with a new parameter of the type *ITimeFormatter*. When the runtime try to perform the method *Configute()* it see that the method depends on an *ITimeFormatter* object, and it it has registered such an object it *inject* such an object to the method. The object (the service) can then be used in the middleware function as normally. If you run the application as shown below you can see the service in action:



A service created using dependency injection can also be used in a middleware component written as a class:

```
public class TimeMiddle
{
  private RequestDelegate next;
  private ITimeFormatter formatter;

  public TimeMiddle(RequestDelegate next, ITimeFormatter formatter)
  {
    this.next = next;
    this.formatter = formatter;
  }

  public async Task Invoke(HttpContext context)
  {
    if (context.Request.Path == "/middle/class")
    {
      await formatter.Format(context, "Middle Class");
    }
    else
    {
      await next(context);
    }
  }
}
```

and the solution is to give the constructor a parameter for the service. If you use the middleware class in *Startup*:

```
app.UseMiddleware<TimeMiddle>();
```

the runtime will se that the constructor for the middleware class need a *ITimeFormatter* object, and when it has registered such an object as a service, it will use is as a parameter.

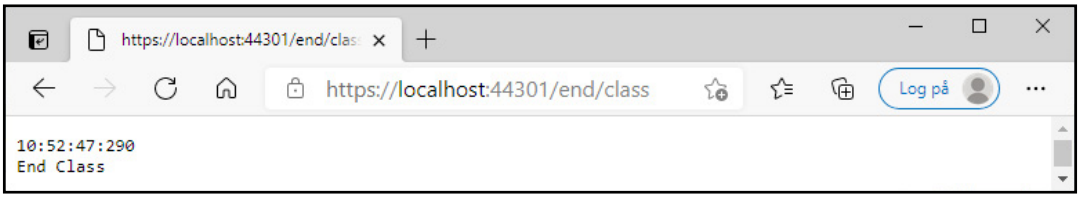
Dependency injection can also be used to transfer services to endpoints, but here dependency injection can not immediately be used as the endpoint class has no constructor:

```
public class TimeEnd
{
  public static async Task Endpoint(HttpContext context)
  {
    ITimeFormatter formatter =
      context.RequestServices.GetRequiredService<ITimeFormatter>();
    await formatter.Format(context, "End Class");
  }
}
```

The solution is to use an extension method for the class *HttpContext* which has references to registered services. You can use the endpoint in *Startup* as:

```
endpoints.MapGet("/end/class", TimeEnd.Endpoint);
```

and an example of running the application could be:



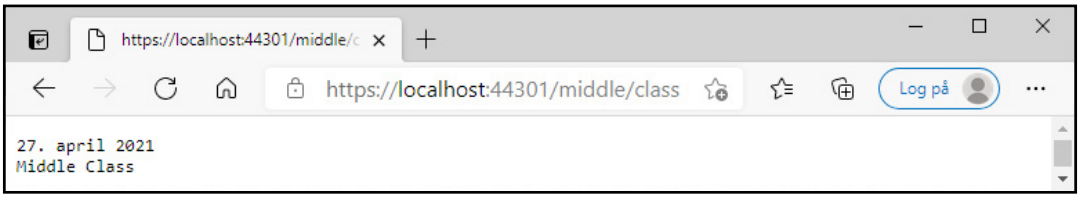
If I need another formatter (another version of the service), for example:

```
public class DateFormatter : ITimeFormatter
{
  private DateTime dt = DateTime.Now;

  public async Task Format(HttpContext context, string content)
  {
    await context.Response.WriteAsync($"{dt.ToLongDateString()}\
    n{content}");
  }
}
```

I only have to update *ConfigureService():*

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddSingleton<ITimeFormatter, DateFormatter>();
}
```

and then it all works again, but using the new formatter:



Using dependency injection to instantiate services is flexible and simple, but compared to the use of custom singletons and broker classes, there is basically not much difference, but dependency injection actually offers a lot more. As used in the example above, the objects are created as singleton objects and thus live throughout the application and the same object is available for all routes. It is not always appropriate and objects created for dependency injection can be created in other ways that define their life cycle, something that one may not need so often but can still be important. It requires a lot of detail and is something I will not go into in this book.

4.5 COOKIES

As mentioned HTTP is a stateless protocol which means that the server does not remember clients. If a client sends a request and together with the request some data the server handles the data and send a response back, but has not registered anything about the request or the data that was sent. Everything regarding the previous request is gone if the client requests the server again. In relation to today's web applications, it gives problems and think as an example of a web store where users put items in a shopping cart, or that you have no identification at all, that it is the same client who sends a request again. All objects created as part of the last request are gone.

There are several solutions on these problems. One is the use of hidden fields in a form, which are fields whose content are not shown in the browser. Such a field can be bound to a value in the model in the same way as other fields, and the value is then send to the client as part of the response as the values of other properties and again send back to the server when the user submits the form. In this way the state of properties is preserved between two requests.

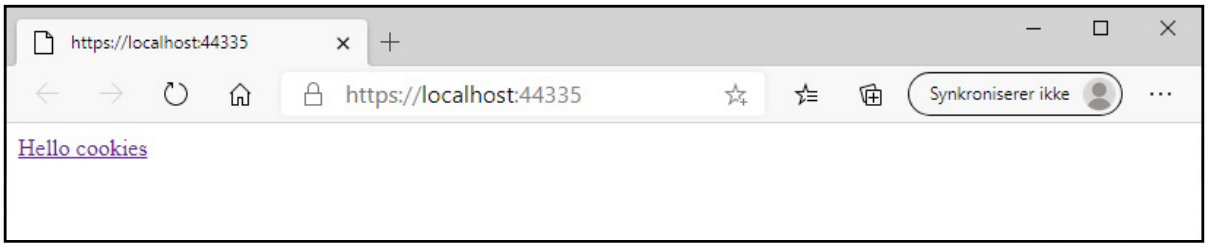
This approach has its uses, but is far from always appropriate. First, only simple data can (without special encoding) be sent back and forth that way, and since data is constantly sent between client and server, it can make unnecessary demands on bandwidth (although it may not be such a big problem today). Finally, it should be noted that this is often data that should only live on the server side and that data that is sent between client and server can in principle always be attacked. The conclusion is that hidden fields are certainly techniques that can and are used to preserve the value of simple data, but are not the solution to preserve more complex objects.

In this section I will look at cookies which is another technique used to send data between the server and a client, a technique which is very popular and modern web applications can actually not exist without. A cookie is not else than a small text which the server adds to the response where it is stored by the browser and send back to the server as part of a subsequent request. In this way cookies can be used to span a series of HTTP requests which each can be identified with the cookie.

To show how I have created a new project *AspNetCore5*. I have change the *Startup* class:

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

  public void Configure(IApplicationBuilder app)
  {
    app.UseDeveloperExceptionPage();
    app.UseRouting();
    app.UseEndpoints(endpoints =>
    {
      endpoints.MapGet("/cookie", async context =>
      {
        int counter = int.Parse(context.Request.Cookies["counter"] ??
        "0") + 1;
        context.Response.Cookies.Append("counter", counter.ToString(),
          new CookieOptions { MaxAge = TimeSpan.FromMinutes(30) });
        await context.Response.WriteAsync($"<h1>Counter: {counter}</h1>
          <div><a href =\"/cookie\">Again</a></div>
          <div><a href =\"/clear\">Clear</a></div>");
      });
      endpoints.MapGet("/clear", context =>
      {
        context.Response.Cookies.Delete("counter");
        context.Response.Redirect("/");
        return Task.CompletedTask;
      });
      endpoints.MapFallback(async context => await context.Response.
      WriteAsync(
        "<a href=\"/cookie\">Hello cookies</a>"));
    });
  }
}
```

There are three endpoints, where the last is a fallback and is an endpoint used if there is no match for one of the other two routes. If there is a match for the first route the endpoint initialize a *counter* when it try to read a cookie send as part of the request. If there is no cookie the value is set 0 and else the value is the value of the cookie. Then 1 is added to the counter. Next the value of the counter is converted to a *string* and added to the response as a cookie. Note that the endpoint also define how long the cookie can live on the client. Note the response from the end point is the value of *counter* (and thus the value of the cookie) and two links. The second endpoint clear the cookie and remove it and then make a redirect to the root pattern. If you run the application the result is:



If you clicks the link you get a match for the first route:

If I here clicks 5 times on the first link:

and when I the clicks on the second link I return the start page. If I here clicks the link again the counter has the value 1, when the cookie is removed.

Much has been said over the years for and against cookies. Firstly, it means that the browser stores data on the client's machine (if the browser is otherwise set to do so), and secondly, cookies are used to collect information about clients. However, many of today's web applications cannot work without the use of cookies, and therefore in some countries a law has been introduced where a web application, if it uses cookies, must ask the user to accept it. For these reasons, a cookie can be defined as essential or not essential, where an essential cookie is a cookie that is always sent together with the server's response. If a cookie is defined as not essential it means that server must check if the client accept the use of cookies before the cookie is sent. Below I have updated the above program to check where to send a cookie.

First I have added a middleware component:

```
public class ConsentMiddleware
{
  private RequestDelegate next;

  public ConsentMiddleware(RequestDelegate next)
  {
    this.next = next;
  }

  public async Task Invoke(HttpContext context)
  {
    if (context.Request.Path == "/granted")
    {
      context.Features.Get<ITrackingConsentFeature>().GrantConsent();
      await context.Response.WriteAsync("<div>Granted</div>");
    }
    else if (context.Request.Path == "/withdrawn")
    {
      context.Features.Get<ITrackingConsentFeature>().WithdrawConsent();
      await context.Response.WriteAsync("<div>Withdrawn</div>");
    }
    await next(context);
  }
}
```

If the URL pattern is */granted* the *context* object is used to grant consent for cookies, and if the pattern is */withdrawn* it revokes the permit. Then I have updated *Startup*:

```
public class Startup
{
  private bool isEssential = false;

  public void ConfigureServices(IServiceCollection services)
  {
    services.Configure<CookiePolicyOptions>(
      opts => { opts.CheckConsentNeeded = context => true; });
  }

  public void Configure(IApplicationBuilder app)
  {
    app.UseDeveloperExceptionPage();
    app.UseCookiePolicy();
    app.UseMiddleware<ConsentMiddleware>();
    app.UseRouting();
    app.UseEndpoints(endpoints =>
    {
      endpoints.MapGet("/cookie", async context =>
      {
        int counter = int.Parse(context.Request.Cookies["counter"] ??
        "0") + 1;
        context.Response.Cookies.Append("counter", counter.ToString(),
          new CookieOptions { MaxAge = TimeSpan.FromMinutes(30),
          IsEssential = isEssential });
        await context.Response.WriteAsync($"<h1>Counter: {counter}</h1>
          <div><a href =\"/cookie\">Again</a></div>
          <div><a href =\"/clear\">Clear</a></div>
          <div><a href =\"/essential\">Essential: " + isEssential + "</
          a></div>
          <div><a href =\"/granted\">Granted</a></div>
          <div><a href =\"/withdrawn\">Withdrawn</a></div>");
      });
      endpoints.MapGet("/clear", context =>
      {
        context.Response.Cookies.Delete("counter");
        context.Response.Redirect("/");
        return Task.CompletedTask;
      });
      endpoints.MapGet("/essential", context =>
      {
```

```
        isEssential = !isEssential;
        context.Response.Redirect("/");
        return Task.CompletedTask;
      });
      endpoints.MapFallback(async context => await context.Response.
      WriteAsync(
        "<a href=\"/cookie\">Hello cookies</a>"));
    });
  }
}
```

The class is expanded with an instance variable which denotes where a cookie is essential or not. The class installs a service which indicates whether the application should check for consent, and in this case the service always return *true*. The check is performed using a middleware component and my own middleware component *ConsentMiddleware* is also added to the request pipeline. The first endpoint is expanded with three more links and an endpoint for the URL pattern */essential* is added, and endpoint which just change the value of the instance variable *isEssential*. If you now run the application and clicks on the link there are five links:

You can now experiment and investigate when the counter is counted up and thus where the cookie is sent to the client.

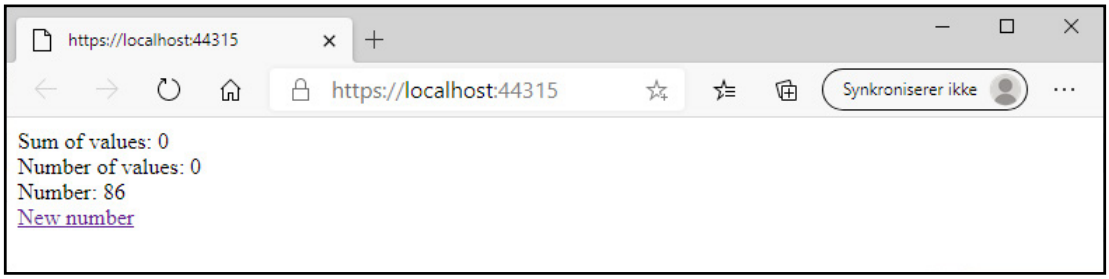
4.6 SESSIONS

A session variable is a variable which lives alone on the server, but which retains its value over a sequence of requests. In principle it is not possible as HTTP is a stateless protocol, but using a cookie the server can bypass this restriction. When a client send a request for

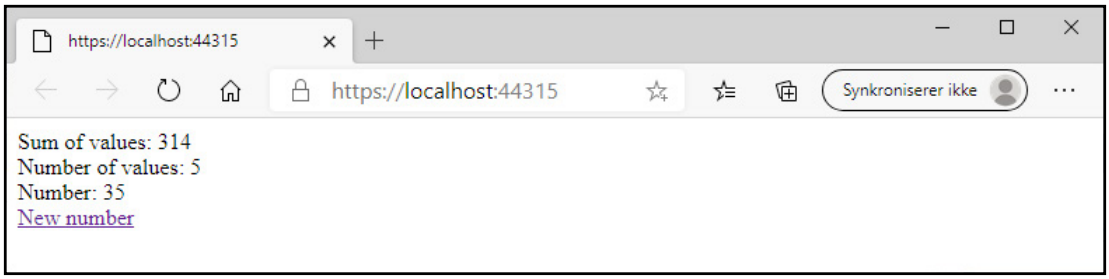
an application the web server looks for a session id, which is a unique id send as a cookie. Which it is the first time the browser request the application no session id exists, and the browser has no session id to send. If so the server creates a session id with a random value and attach it to the server's response to the browser. Next time the browser request the application it sends the session id back, and this continue as long the browser requests the same application. In this way there is established and identification of an actual browser somewhere and an actual application on the web server, and it is what is called a session. When the web server receives a request for a specific application from the person from whom this request comes and the server can therefore create objects that exactly relate to the session in question and let them live on the server between several requests. It is this kind of objects which is called a session variable.

There are several limitations to session variables. As the first they has a timeout, which as default is 20 minutes and this means if there has been no reference to the variable in a period length than the timeout, the server automatic remove the variable. The application can also remove the variable itself, and it can set the timeout, and the meaning is of course to avoid the server's memory gradually filled up by unused session variables. Another limitation is the type of a session variable which must be *int* or *string*. If you want to store other objects as session variables they must first be serialized to a string. It is important to note that the value of a session variable never is sent to the browser and session variables are only known by the server.

As an example the project *AspNetCore6* uses two session variables. If you run the application it opens the following browser:



The first two lines shows the values of two session variables, both of the type *int*. The third line shows a random generated 2-digits integer, which is added to the first session variable where the other is incremented by 1. If you clicks on the link the browser send a request the application again and is then a request within the same session. Below is the browser window after the link has been clicked 5 times:



The code is:

```
public class Startup
{
  private static Random rand = new Random();

  public void ConfigureServices(IServiceCollection services)
  {
    services.AddDistributedMemoryCache();
    services.AddSession(options => {
      options.IdleTimeout = TimeSpan.FromMinutes(30);
      options.Cookie.IsEssential = true;
    });
  }

  public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
  {
    if (env.IsDevelopment())
    {
      app.UseDeveloperExceptionPage();
    }
    app.UseSession();
    app.UseRouting();
    app.UseEndpoints(endpoints =>
    {
      endpoints.MapGet("/", async context =>
      {
        int cou = context.Session.GetInt32("counter") ?? 0;
        int sum = context.Session.GetInt32("sum") ?? 0;
        int num = rand.Next(10, 100);
        context.Session.SetInt32("sum", sum + num);
        context.Session.SetInt32("counter", cou + 1);
        await context.Response.WriteAsync("<div>Sum of values: " + sum +
          "</div><div>Number of values: " + cou + "</div><div>Number: " +
          num + "</div><div><a href =\"/\">New number</a></div>");
        });
    });
  }
}
```

First you must note how to install a service to use session variables. You must install two services in *ConfigureServices()*. The first tells that session variables should be stored in memory, and there are other possibilities, for example to store the values in a database. The other define options for session variables and here the timeout and that session id is an essential cookie. Next you must note the definition of a middleware in *Configure()* which is the service for use of sessions.

Else the application has only one endpoint which starts to determine the values of two session variables. If they not exists they are set to 0. Next a random number is generated and the two session variables are updated.

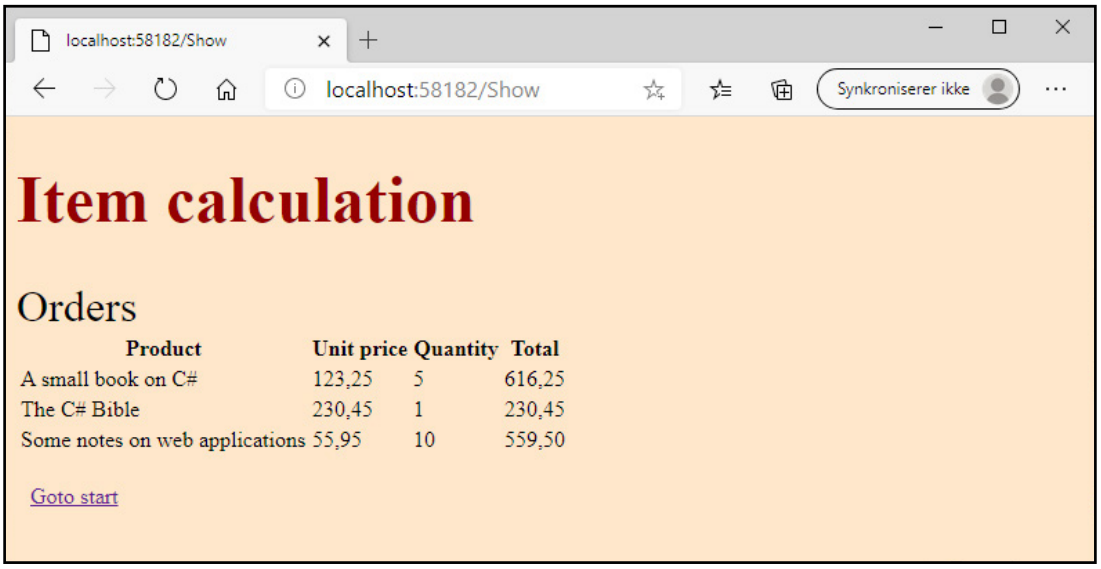
Exercise 8: An improved Calculation

In this exercise you should write another version of the application from exercise 6. Create a copy of the solution for exercise 6 and calls the copy *Calculation1*. You should then add two adjustments:

1. The Razor page *Order.cshtml* should have a link which sends the user back to the start page *Index.cshtml*.
2. If the class *Order* in the file *Order.cshtml.cs* is defined as an inner class you should move the class such it is an ordinary class. If so, you also have to update the reference in *Order.cshtml*.

Build and run the application to ensure that all works.

You must then improve the program such the result of each calculation is added to a list of *Order* objects. The start page *Index.cshtml* must have a link which opens a page showing all stored calculations. The result could be as shown below where there are three calculations:



The list of order objects should be stored as a session variable.

1) Update *Startup.cs* such the program is ready for use of session variables.

2) Add a class with two extension methods for the type *Session*, methods which can be used to store an object as a session variable when the object is serialized to a JSON string:

```
public static class SessionExtensions
{
  public static void SetJson(this ISession session, string key, object
  value)
  {
    session.SetString(key, JsonSerializer.Serialize(value));
  }

  public static T GetJson<T>(this ISession session, string key)
  {
    var data = session.GetString(key);
    return data == null ? default(T) : JsonSerializer.
    Deserialize<T>(data);
  }
}
```

3) In the model class *Order.cshtml.cs* you must update the handler for post to update the list for *Order* objects:

```
List<Order> orders =
  HttpContext.Session.GetJson<List<Order>>("orders") ?? new
  List<Order>();
orders.Add(order);
HttpContext.Session.SetJson("orders", orders);
```

That is all results for a calculation are added to a list which is stored as a session variable.

4) Add a new Razor page which you can call *Show*. You can implement the model as:

```
public class ShowModel : PageModel
{
  public void OnGet()
  {
    Orders =
      HttpContext.Session.GetJson<List<Order>>("orders") ?? new
      List<Order>();
  }

  public List<Order> Orders { get; private set; }
}
```

and you can create the page as a simple HTML table:

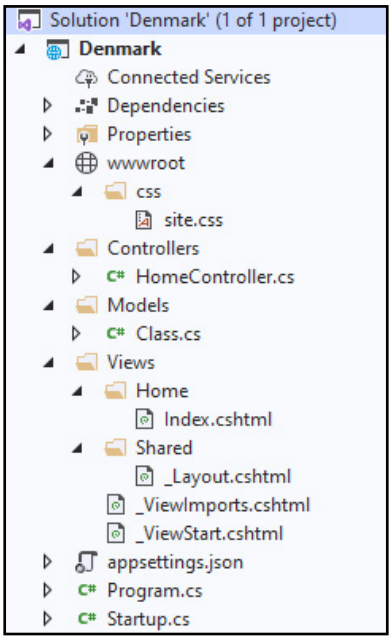
5 A ASP.NET CORE MVC APPLICATION

The subject for this chapter is to show how to create a web application using the MVC pattern and the chapter should then show some guidelines to follow to create a simple web applications. The example is an application used to manage a simple database and is as so a typical web application where the user can search data in a database and maintain the content of the database. As database I will use the database *Denmark* created in the book C# 6, problem 1. If you do not remember the database or you not have the database installed you should read problem 1 in C# 6. I will describe the development through the following sections:

1. Project start
2. The model and connecting the database
3. Show municipalities
4. Show zip codes
5. Database maintenance

5.1 PROJECT START

I start the development by creating a new *ASP.NET Core Web Application* in Visual Studio, a project I have called *Denmark* and is created using an empty template. The first step is to create 7 folders and add 7 standard files:



The files are

1. a stylesheet
2. a home controller
3. a temporary file to the *Models* folders as it should not be empty
4. a view for the start page
5. a Razor Layout for views
6. a Razor View Imports
7. a Razor View Start

You must note in which folders the files are created. Also note the names both for folders and files, which are default names. You can choose others, but you should use the default names as an MVC application is largely based on these names and you make life cumbersome by choosing other names. Three of the files must be modified. The one is *_ViewImports.cshtml* to name the standard tag helpers:

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

This is why the folder *Models* must contains a class as Visual Studio else report an error. The other is the file *_Layout.cshtml*:

```
<!DOCTYPE html>

<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>@ViewBag.Title</title>
  <link href="~/css/site.css" rel="stylesheet" />
</head>
<body>
  <div>Denmark</div>
    <div>
        @RenderBody()
    </div>
</body>
</html>
```

and the important is to add a reference to the stylesheet. The div element with the text *Denmark* is just to se an effect of the layout file. Finally I also update the view *Index.cshtml* and the purpose is alone to show that view is used:

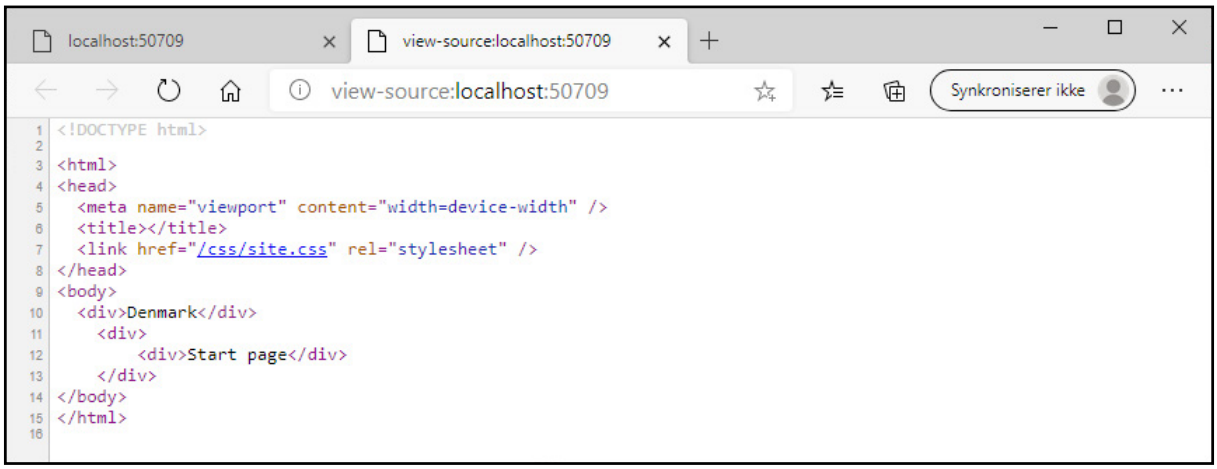
```
@{
}
<div>Start page</div>
```

As the last step to initialize a MVC application I must update the *Startup* class:

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

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

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

Then it should be possible to build an run the application:

and if you view the code the result is:



The result is the default view *Index.cshtml* inserted in the layout file and the purpose is that all the application's views should have the same layout and use the same stylesheet.

5.2 THE MODEL AND CONNECTING THE DATABASE

To implement the model I have to write four classes:

1. A class representing a region
2. A class representing a zip code
3. A class representing a municipality
4. A class representing the database

The first is trivial as it not should be possible to modify a region, and as an example I will show the second:

```
public class Zipcode : IComparable<Zipcode>
{
  public Zipcode()
  {
    Municipalities = new List<Municipality>();
  }

  [Required(ErrorMessage = "You must enter code")]
  [RegularExpression("[1234567890]{4}", ErrorMessage = "...")]
  public string Code { get; set; }

  [StringLength(30)]
  [Required(ErrorMessage = "You must enter city, max 30 characters")]
  public string City { get; set; }

  public List<Municipality> Municipalities { get; private set; }

  public override bool Equals(object obj)
  {
    if (obj == null) return false;
    if (GetType() == obj.GetType())
    {
      Zipcode zipcode = (Zipcode)obj;
      if (Code == null) return zipcode == null;
      return Code.Equals(zipcode.Code);
    }
    return false;
  }

  public override int GetHashCode()
  {
    return Code == null ? 0 : Code.GetHashCode();
  }

  public override string ToString()
  {
    return string.Format("{0} {1}", Code, City);
  }

  public int CompareTo(Zipcode zipcode)
  {
    return Code == null ? -1 : Code.CompareTo(zipcode.Code);
  }
}
```

It is a typical model class and there is not much to note, but you must note how to decorate the properties for code and city, such they can be validated in a web form. Also note that the class implements *Equals()* and is defined comparable. Note the list for *Municipality* objects when a *Zipcode* object has references for all municipalities that uses this zip code. That is the one side of the many to many relation in the database and thus the foreign key.

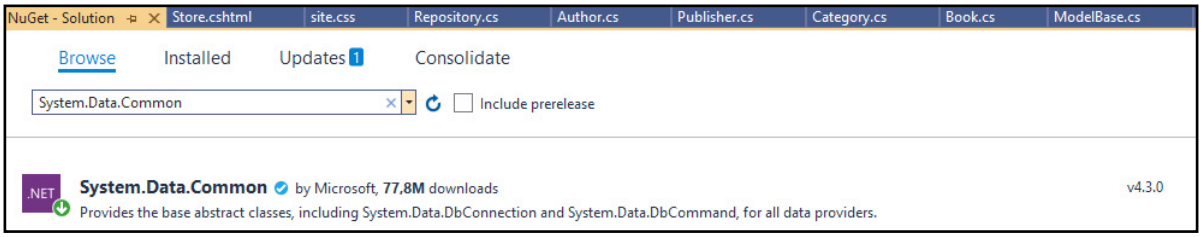
The class *Repository* represents the database and I should not show the full code here as it is extensive:

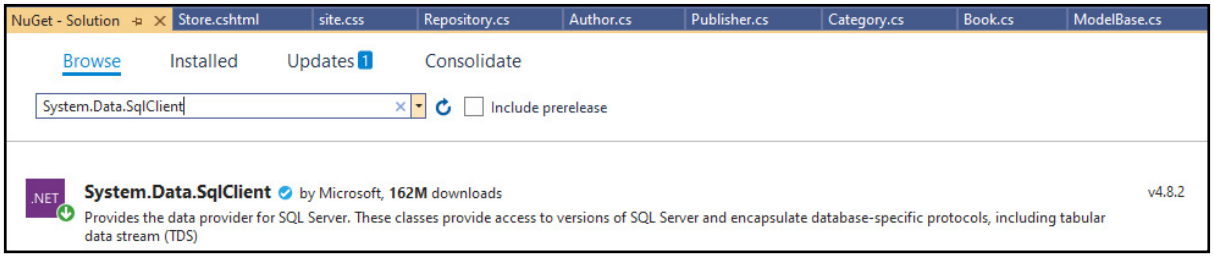
```
public class Repository : RepositoryBase
{
  private static readonly Repository instance = new Repository();

  static Repository()
  {
  }

  private Repository()
  {
    try
    {
      Initialize();
    }
    catch
    {
      throw new Exception("Unable to connect to database");
    }
  }

  public static Repository Instance
  {
    get { return instance; }
  }

  public List<Region> Regions { get; private set; }
  public List<Zipcode> Zipcodes { get; private set; }
  public List<Municipality> Municipalities { get; private set; }

  ...
}
```

The class is written as a singleton and when the object is created it initializes three collections with the content of the database. It is simple and is done in a private method *Initialize()*. Then the class must have methods to manipulate the content of the database, three for

each of the two tables *Zipcode* and *Municipality*. These methods are in principle all simple, but are written as transactions as all methods must update both the actual table and the relation table *Post*. It's all complicated by the fact that all methods also must maintenance the memory model of the database.

However, there is a single challenge as the project does not have references to the assemblies for database operations. These references are added with NuGet:





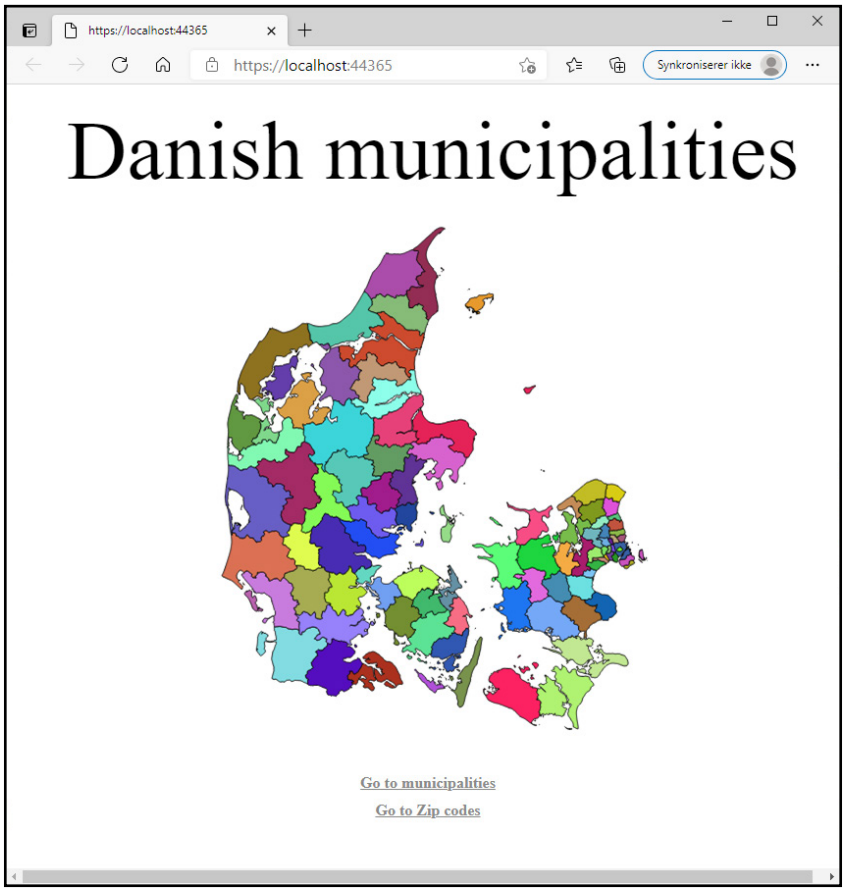
and then the application can be compiled. When writing a web application like this one could instead uses the Entity Framework which is tightly integrated with ASP.NET Core. It can provide significant benefits as one has to write much less code than is the case in this case but everything is not is not on the plus side and in many cases I actually prefer to write my own repository class as in this example. At least I will not in this book use Entity Framework, but the following books shows how.

5.3 SHOW MUNICIPALITIES

In this section I will show the implementations of two views:

1. A start page and that is *Index.cshtml*
2. A view which shows all municipalities

Most web applications starts with a welcome page, where the site is present and for this application it is the view *Index*:

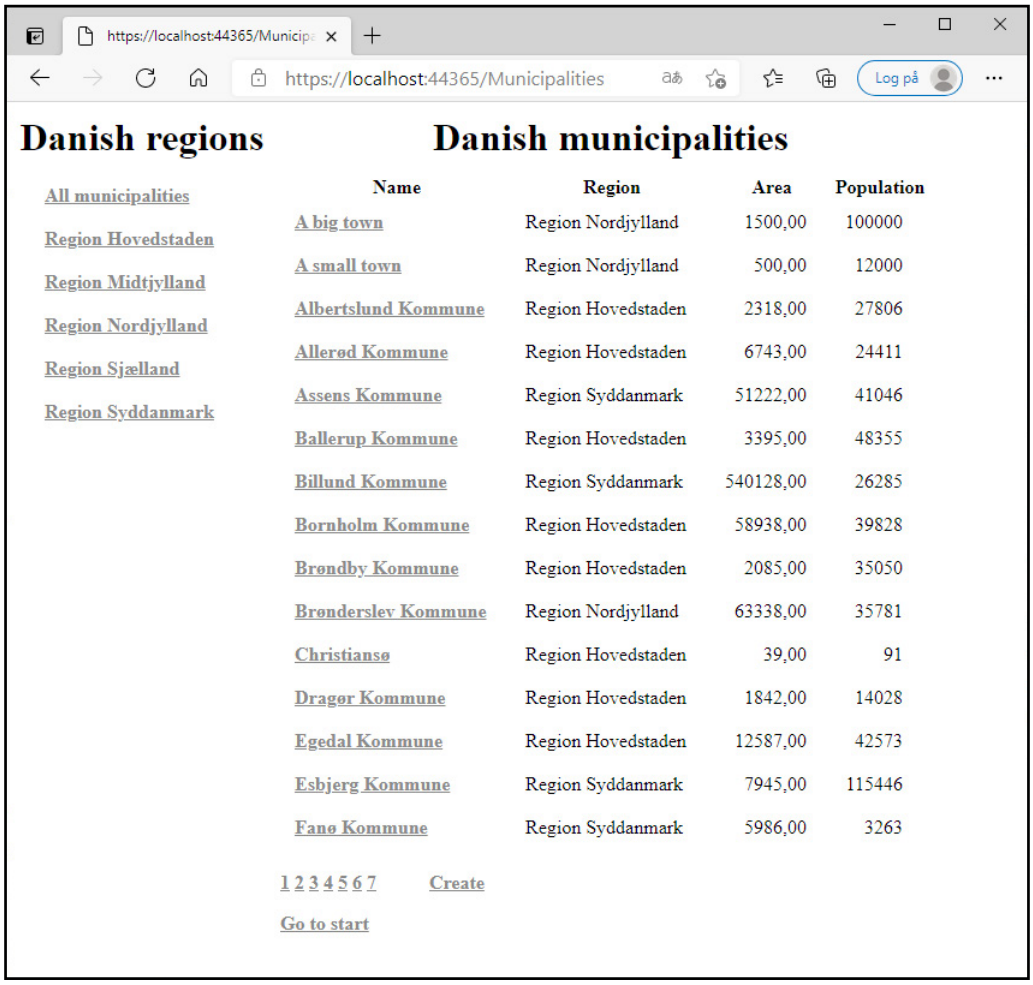


It's nothing more than a static HTML page showing an image and two links:

```
<div class="index-head">Danish municipalities</div>
<div class="index-head"><img src="~/images/municipalities.png" /></div>
<div class="center-text"><a asp-action="Municipalities"
  asp-controller="Home" class="link-text">Go to municipalities</a></div>
<div class="center-text"><a asp-action="Zipcodes"
  asp-controller="Home" class="link-text">Go to Zip codes</a></div>
```

The image is stored in a sub-folder *images* in *wwwroot*. The most important is the two links where the links references an action method in the controller. Note the links also references the controller which is not necessary as *Home* is the default controller. To navigates to the two views referenced by the links I must add two action methods called respectively *Municipalities* and *Zipcodes* to the controller and correspondingly two views to the *Home* folder. In the rest of this section I will look at the view *Municipalities* which should show all municipalities.

If you click on the first link you get a page which shows two columns, where the first column shows all regions while the other shows 15 municipalities:



When you look at the page you must note:

1. The region names are links and are used to filter the municipalities such only municipalities for this region are shown.
2. The municipality name is a link and is used to open a form to edit the municipality. This function is not implemented yet.
3. Below the municipalities are some links shown as numbers which are used to navigate to an other page for municipalities.
4. The navigation line also has a link *Create* which be used to create a new municipality. This feature is also not implemented yet.
5. Last there is a link which send the user back to the start page.

The implementation contains many details and it can actually be quite a challenge to keep track of how it all fits together, so therefore a relatively detailed review. In all views and view components I use styles defined in the style sheet, and I will not generally mention these styles. They are all simple, but it is recommended that you examine the style sheet yourself and notice the effect of the individual styles.

When you clicks on the link for municipalities on the start page the following action method in the *HomeController* is performed:

```
public IActionResult Municipalities(string region, int page = 1)
{
  IEnumerable<Municipality> municipalies =
    Repository.Instance.AllMunicipalities.Where(m => region == null ||
    region.Length == 0 || (m.Region != null && m.Region.Name.
    Equals(region))).Skip
    ((page - 1) * pageSize).Take(pageSize).OrderBy(m => m.Name);
  int items = region == null || region.Length == 0 ?
    Repository.Instance.Municipalities.Count :
    Repository.Instance.Municipalities.Where
    (m => m.Region.Name.Equals(region)).Count();
  Paging paging =
    new Paging { PageSize = pageSize, Current = page, TotalItems =
    items };
  return View(new MunicipalitiesModel { Municipalities = municipalies,
    Regions = Repository.Instance.Regions, Paging = paging,
    CurrentRegion = region });
}
```

First note that the method has two parameters where the first is used for the region clicked on in the left column in the above view. The other denotes the navigation page number when the user clicks on a page number. When you clicks on the link on the start page both parameters are undefined which means that *region* is *null* while *page* has the default value 1. The action method uses the view to create the response and must send some data to the view. It is data about the municipalities to show, the regions and data for the paging, that is a model for the view. This model is defined in a class in a sub-folder *ViewModels* in the folder *Models* and is called *MunicipalitiesModel*:

```
public class MunicipalitiesModel
{
  public IEnumerable<Region> Regions { get; set; }
  public IEnumerable<Municipality> Municipalities { get; set; }
  public Paging Paging { get; set; }
  public string CurrentRegion { get; set; }
}
```

It is a very simple class, but typical you defines such a class for each view. The class has four properties where the first is the regions, the next the municipalities to show, the third is for the navigation page links while the last is the selected region. This values are initialized in the action method in the controller, and the method creates a *MunicipalitiesModel* object and use it for a model for the view. The municipalities to show are selected used a complex LINQ expression and the same goes for the number of matching municipalities used to create a *Paging* object:

```
public class Paging
{
  public int TotalItems { get; set; }
  public int PageSize { get; set; }
  public int Current { get; set; }
  public int TotalPages
  {
    get { return (int)Math.Ceiling((double)TotalItems / PageSize); }
  }
}
```

It also contains four properties where the last is calculated and is used for the number of page links.

Then there is the view:

```
@model MunicipalitiesModel
<table>
  <tr style="font-size:24pt">
    <th>Danish regions</th>
    <th>Danish municipalities</th>
  </tr>
  <tr>
    <td style="display:inline-block;vertical-align:top">
      <table>
        <tr>
          <td>
            <a class="link-text" asp-action="Municipalities" asp-
            controller="Home"
              asp-route-region="">All municipalities</a>
          </td>
        </tr>
        @foreach (Region region in Model.Regions)
        {
          <tr>
            <td>
              <a class="link-text" asp-action="Municipalities"
                asp-controller="Home" asp-route-region="@region.Name"
                asp-route-page="1">@region.Name</a>
            </td>
          </tr>
        }
      </table>
    </td>
    <td style="vertical-align:top">
      <table>
        <tr>
          <th>Name</th>
          <th>Region</th>
          <th>Area</th>
          <th>Population</th>
        </tr>
        @foreach (Municipality municipality in Model.Municipalities)
        {
          <tr>
            <td><a class="link-text" asp-action="Municipality"
              asp-controller="Edit" asp-route-munber="@municipality.Id"
```

```
            asp-route-region="@Model.CurrentRegion"
            asp-route-page="@Model.Paging.Current">
            @municipality.Name</a></td>
          <td>@municipality.Region.Name</td>
          <td style="text-align:right">@municipality.Area</td>
          <td style="text-align:right">@municipality.Population</td>
        </tr>
      }
    </table>
    <p/>
    <div>
      @for (int i = 1; i <= Model.Paging.TotalPages; ++i)
      {
        <span style="width:20px"><a class="link-text"
          asp-action="Municipalities" asp-controller="Home"
          asp-route-page="@i" asp-route-region="@Model.CurrentRegion">
          @i</a></span>
      }
      <span style="margin-left:40px"><a class="link-text"
        asp-action="Municipality" asp-controller="Edit"
        asp-route-region="@Model.CurrentRegion"
        asp-route-page="@Model.Paging.Current">Create</a></span>
    </div>
    <p/>
    <div><a asp-action="Index" class="link-text">Go to start</a></div>
  </td>
 </tr>
</table>
```

First you should note that the design of the user interface which build on nested table elements. When you see the design, it is easy enough to understand and the only thing that can be difficult to figure out is the links. There are six links. The first link is for all regions and link to the action method *Municipalities* in the controller *Home*. *asp-route-* is a prefix which set the parameter *region* to blank (a name convention), a value used for URL routing. It means that the action method receive a blank value for the parameter *region*. The next link is for the other menu items and thus a link for all region names. There is not the big difference, but this time the value of *region* is the region name and the value of *page* is set to 1 (not needed as it is the default value). Each municipality name is a link, but here the action method is *Municipality* in a controller called *Edit*. This controller is not defined yet, but you should note the parameters where there are three: The municipality number, the current region and the page number. The last two are used to ensure to return from the *Municipality* view to the current page.

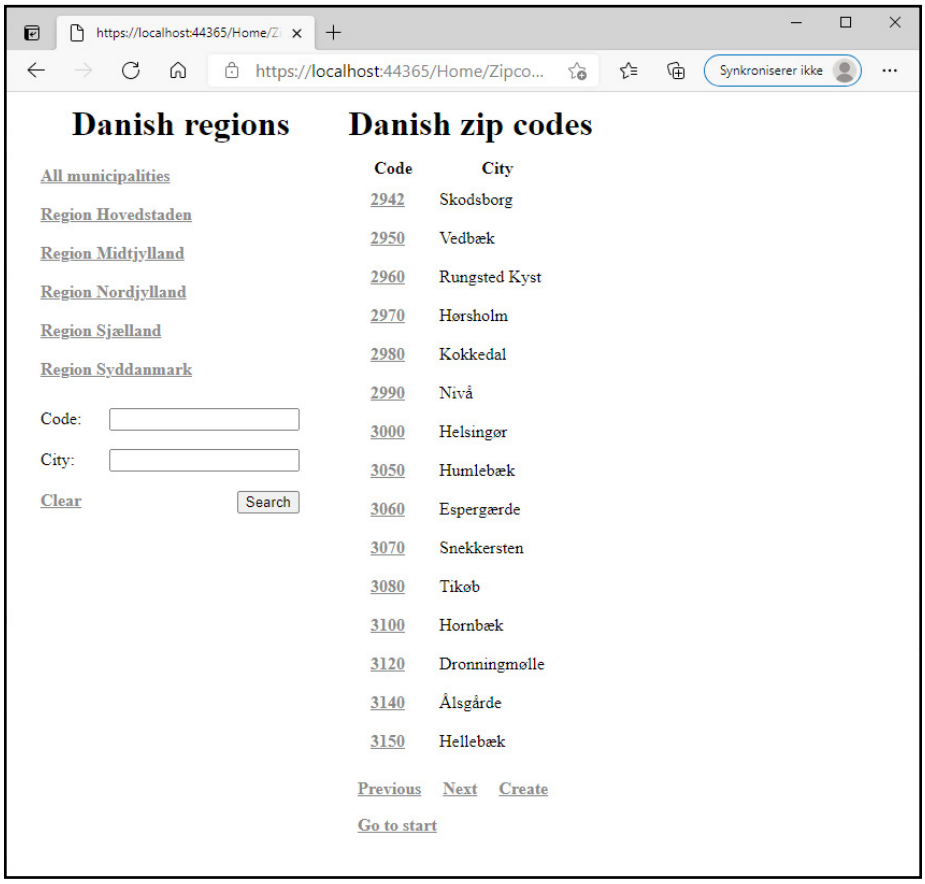
The there is the navigation links which are created in a *for* loop. You should note the syntax as well as the possibility to use C# code inside a HTML part to dynamic create HTML. The links does not contain anything new and you must primarily note the parameters which are the new page number and the current region.

The last two links are for creating a new municipality and to return to the start page. The link to create a new municipality is in principle identical to the link for the municipality name, just is there no value for municipality number.

In the URL routing these values are used to select the right endpoint and as parameters for the action method. The third link is the same as the first, but the last look different, and it is.

5.4 SHOW ZIP CODES

If you on the start page clicks on the link for zip codes you get the page shown below. It is in principle identical to the page for municipalities, but this time there are more items (about 1100) and instead using a navigation page link for each page I uses two links to step back and forth. As before the zip codes can be filtered by regions (the zip codes which are used by municipalities in that region), but there may still be many zip codes and as so there also is a possibility to search for a zip code.



First I have to add a class to the folder *Models/ViewModels* used for search values:

```
public class SearchValues
{
  public string Code { get; set; }
  public string City { get; set; }
}
```

The class is trivial and should be used to transfer search data to and from the page. I have also create another model class (in the same folder) for the new view:

```
public class ZipcodesModel
{
  public IEnumerable<Region> Regions { get; set; }
  public IEnumerable<Zipcode> Zipcodes { get; set; }
  public Paging Paging { get; set; }
  public string CurrentCategory { get; set; }
  public SearchValues Search { get; set; }
}
```

It is also a simple class and look likes the class for municipalities, but the type of the collections for data items is changed and an object for the search criteria is added.

The controller is expanded with two new action methods as there must be an action method for a GET request as well as an action method to POST the form:

```
[HttpGet]
public IActionResult Zipcodes(string region, string code, string city,
  int page = 1)
{
  SearchValues search = new SearchValues { Code = code, City = city };
  IEnumerable<Zipcode> zipcodes =
    Repository.Instance.AllZipcodes.Where(z => UseZipcode(z, region) &&
    UseZipcode(z, search)).Skip((page - 1) * pageSize).
    Take(pageSize).OrderBy(z => z.Code);
  int items = region == null || region.Length == 0 ?
    Repository.Instance.Zipcodes.Where(z => UseZipcode(z, search)).
    Count() :
    Repository.Instance.Zipcodes.Where(z => UseZipcode(z, region) &&
    UseZipcode(z, search)).Count();
  Paging paging = new Paging { PageSize = pageSize, Current = page,
    TotalItems = items };
  return View(new ZipcodesModel { Zipcodes = zipcodes,
    Regions = Repository.Instance.Regions, Paging = paging,
    CurrentRegion = region, Search = search });
}

[HttpPost]
public ViewResult Zipcodes(ZipcodesModel model)
{
  IEnumerable<Zipcode> zipcodes =
    Repository.Instance.AllZipcodes.Where(
    z => UseZipcode(z, model.Search)).OrderBy(z => z.Code);
  Paging paging = new Paging { PageSize = pageSize, Current = 1,
    TotalItems = zipcodes.Count() };
  return View(new ZipcodesModel { Zipcodes = zipcodes.Take(pageSize),
    Regions = Repository.Instance.Regions, Paging = paging,
    CurrentRegion = model.CurrentRegion, Search = model.Search });
}
```

In principle the action method should works in the same way as before, but this time it is more complicated to find the zip codes used by a region and if the zip codes matches the search values. For that I have added two helper methods (not shown above) to test where a zip code should be used. You should note that the action method for GET this time has four parameters. The reason why there are parameters for the search values is that the result of a search may well include several pages, and therefore the search values must be preserved if you navigate back and forth.

The view look likes the view for municipalities, but the links are a bit more complex as there are several parameters and finally there is the form:

```
@model ZipcodesModel
@{
  int prev = Model.Paging.Current - 1;
  int next = Model.Paging.Current + 1;
}
<table>
  <tr style="font-size:24pt">
    <th>Danish regions</th>
    <th>Danish zip codes</th>
  </tr>
  <tr>
    <td style="display:inline-block;vertical-align:top">
      <table>
        <tr>
          <td>
            <a class="link-text" asp-action="Zipcodes" asp-
            controller="Home"
              asp-route-region="">All municipalities</a>
          </td>
        </tr>
        @foreach (Region region in Model.Regions)
        {
          <tr>
            <td>
              <a class="link-text" asp-action="Zipcodes" asp-
              controller="Home"
                asp-route-region="@region.Name" asp-route-page="1">@
                region.Name</a>
            </td>
          </tr>
        }
        <tr>
          <td>
            <form asp-action="Zipcodes" asp-controller="Home"
            method="post">
              <input asp-for="CurrentRegion" type="hidden" />
              <div>
                <label asp-for="Search.Code" class="search-text">Code:</
                label>
                <input asp-for="Search.Code" class="search-field" />
              </div>
```

```
            <div>
              <label asp-for="Search.City" class="search-text">City:</
              label>
              <input asp-for="Search.City" class="search-field" />
            </div>
            <div>
              <button type="submit"
                style="margin-top: 10px;float: right">Search</button>
            </div>
            <div>
              <span class="search-text"><a class="link-text"
                asp-action="Zipcodes" asp-controller="Home" asp-route-
                page="1"
                asp-route-region="@Model.CurrentRegion">Clear</a></
                span>
            </div>
          </form>
        </td>
      </tr>
    </table>
  </td>
  <td style="vertical-align:top">
    <table>
      <tr>
        <th>Code</th>
        <th>City</th>
      </tr>
      @foreach (Zipcode zipcode in Model.Zipcodes)
      {
        <tr>
          <td><a class="link-text" asp-action="Zipcode" asp-
          controller="Edit"
            asp-route-code="@zipcode.Code"
            asp-route-region="@Model.CurrentRegion"
            asp-route-page="@Model.Paging.Current">@zipcode.Code</a></td>
          <td>@zipcode.City</td>
        </tr>
      }
    </table>
    <p />
    <div>
```

```
        @if (Model.Paging.Current > 1)
        {
          <span><a class="link-text" asp-action="Zipcodes"
            asp-controller="Home" asp-route-page="@prev"
            asp-route-region="@Model.CurrentRegion"
            asp-route-code="@Model.Search.Code"
            asp-route-city="@Model.Search.City">
            Previous</a>    </span>
        }
        @if (Model.Paging.Current < Model.Paging.TotalPages)
        {
          <span><a class="link-text" asp-action="Zipcodes"
            asp-controller="Home" asp-route-page="@next"
            asp-route-region="@Model.CurrentRegion"
            asp-route-code="@Model.Search.Code"
            asp-route-city="@Model.Search.City">
            Next</a>    </span>
        }
        <span><a class="link-text" asp-action="Zipcode" asp-
        controller="Edit"
          asp-route-region="@Model.CurrentRegion"
          asp-route-page="@Model.Paging.Current">Create</a></span>
      </div>
      <p />
      <div><a asp-action="Index" class="link-text">Go to start</a></div>
    </td>
  </tr>
</table>
```

First note that the view this time creates two variables. It is normally C# variables used for the page number if the user steps forward or backward. The design is as before using two tables inside each other. Also the links in the menu (the regions) are as before. This time the menu also has the form with the search fields. The *form* tag defines the form should submit to an action method *Zipcodes* in the *Home* controller. The form has three fields which are all bound to the model using *asp-for*. The first field is a hidden field used for the selected region when this value else will be lost. The form also has a *Clear* link used to clear the search values. You should note that it references the *Zipcodes* action method, but using GET and does not submit the form.

The table with zip codes is as before and each zip code is a link for an action method in a controller *Edit*. The difference is the navigation line where there this time only are two link for navigation. Note that these links are only shown if it is possible to go forward or backward.

5.5 DATABASE MAINTENANCE

In the last step I want to implement maintenance of the database, and it should be possible to maintenance two tables, and as so the application must be expanded with two new views. Actually, it is not quite simple, as it must be possible to create, change and delete rows in the two tables, and it is all complicated by the fact that there is a many to many relationship between *Municipalities* and *Zipcodes*.

The program is written so that action methods for maintaining the database must be placed in their own controller. There is no special reason for this other than to show how and what is needed is:

1. Add a new controller called *EditController* to the *Controllers* folder
2. Add a sub-folder to the *Views* folder called *Edit*
3. Add two views to the new view folder called respectively *Municipality* and *Zipcode*
4. Update the new controller as:

```
public class EditController : Controller
{
  public IActionResult Municipality()
  {
    return View();
  }

  public IActionResult Zipcode()
  {
    return View();
  }
}
```

Then you can run the application and all links should works, of course with trivial result.

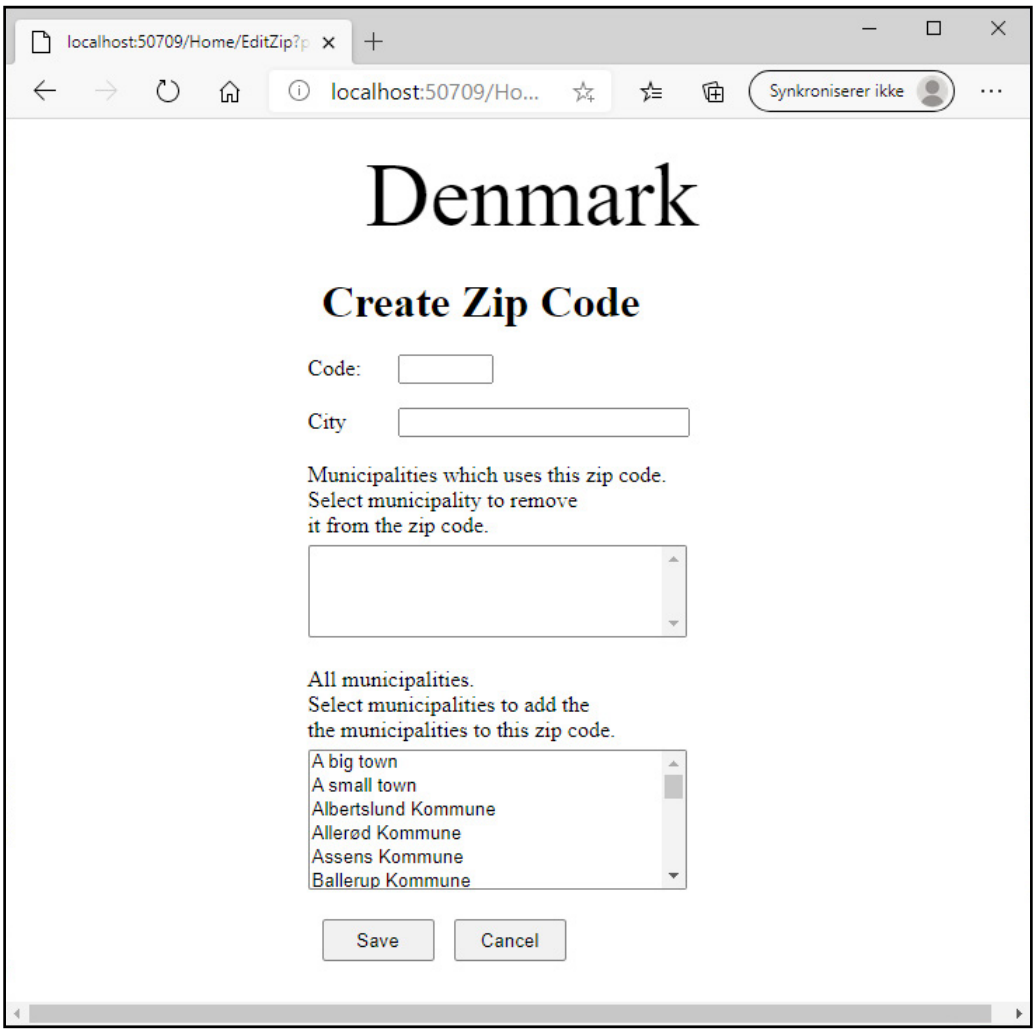
All problems regarding the maintenance of the database tables and including keeping the memory representation of the tables are solved in the *Repository* class, so what is left has only to do with the communication between the controller and the individual views. Here the challenges are that HTTP is a stateless protocol, and that after a view is generated and sent to the browser, the controller's objects no longer exist. As mentioned above there are several ways to solve these problems, and one can, for example, use hidden fields and session variables. I will use both and to be able to use session variables I must as shown in the previous chapter enable the use of session variable in the *Startup* class:

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

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

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

I want to save a list as a session variable and for that I have added the following class to the *Tools* folder:

```
public static class SessionExtensions
{
  public static void SetJson(this ISession session, string key, object
  value)
  {
    session.SetString(key, JsonSerializer.Serialize(value));
  }

  public static T GetJson<T>(this ISession session, string key)
  {
    var sessionData = session.GetString(key);
    return sessionData == null ? default(T) :
      JsonSerializer.Deserialize<T>(sessionData);
  }
}
```

I will start with maintenance of zip codes, and if you select zip codes from the start page and then clicks on the link *Create* you get the following page:



The user must enter the zip code as well as name of the city. The lower list (*select* element) shows all municipalities and if the user select one or more municipalities and clicks *Save* the selected municipalities are added as the municipalities which uses the zip code, and if you again opens the form for the new zip code, the municipalities are shown in the upper list. If you edit an existing zip code, and you want to remove a municipality that uses the zip code you must just select the municipality in the upper list and clicks *Save*. If the form is opened for an existing zip code the form also has a *Remove* button.

It is how it works, and to implements the function I must

1. add a view model class to the *Models\ViewModels* folder
2. update the action method in the controller
3. add three new action methods to the controller
4. add a view

The view model class is:

```
public class ZipcodeModel
{
  public int Page { get; set; }
  public string Region { get; set; }
  public bool IsNew { get; set; }
  public Zipcode Zipcode { get; set; }
  public int[] SelectedTags1 { get; set; }
  public SelectList Municipalities1 { get; set; }
  public int[] SelectedTags2 { get; set; }
  public SelectList Municipalities2 { get; set; }
  public string Error { get; set; }
}
```

and represents the data which the controller must send to the view. The two first properties is used to ensure that one can return to the same page from which the form was called. The next is used to indicate where the form is used to create a new zip code or to update an existing zip code. The next is the model for the zip code. The next four properties are used for the two *select* lists. A *SelectList* is a list of key / value pairs, where value is the value shown in the list, and key is the key for that value. The *SelectedTags* arrays contains the keys for selected values when the form is submitted. The last property is used for an error message and is explained below.

When four action methods are needed it is an action method for HTTP GET and three actions methods for HTTP POST, one for each button, and you should note that a form can have more submit buttons, just they have different names. The first action method is performed when the user in the page *Zipcodes* click on a zip code or clicks on the *Create* link:

```
[HttpGet]
public ViewResult Zipcode(string code, string region, int page)
{
  Zipcode zipcode = code == null || code.Length == 0 ? new Zipcode { } :
    Repository.Instance.Zipcodes.Where(z => z.Code.Equals(code)).First();
  List<int> refs = new List<int>();
  foreach (Municipality m in zipcode.Municipalities) refs.Add(m.Id);
    HttpContext.Session.SetJson("zipcode", refs);
  return View(new ZipcodeModel { Page = page, Region = region,
    IsNew = code == null, Zipcode = zipcode, Municipalities1 =
    new SelectList(zipcode.Municipalities, nameof(Models.Municipality.Id),
    nameof(Models.Municipality.Name)), Municipalities2 =
    new SelectList(Repository.Instance.Municipalities,
    nameof(Models.Municipality.Id), nameof(Models.Municipality.Name)) });
}
```

It look likes the action methods in the *Home* controller and has three parameters, where the first is the key for *Zipcode* object to edit. The two others is used are used to return to the same page from where this view was requested (if this return should work exactly, any search criteria should also be sent as parameters). If the parameter *code* has a value the *Zipcode* with this value as key is retrieved from the repository and the user can edit this object. Else a new *Zipcode* object is created and the user may enter values for the object. Before create a *ZipcodeModel* object and as so a model for the view the *id* for all municipalities using this zip code are stored in a list and JSON serialized and stored in a session variable. I explain why below.

When you submit a form the server creates an object which type is the model. Key / values pairs initialized in the form (*asp-for*) are send to the server and are used to initialize properties at the model object, but only this properties can be initialized. This means that the list for municipalities in the *Zipcode* object not is initialized and the *Municipality* objects it contained before are lost. To solve this problem the action method creates a list containing the municipality numbers for these municipalities and store this object as a session variable. Note, that it would be simpler to store the *Zipcode* object, but the object contains a list of *Municipality* objects which again contains a list of *Zipcode* objects. This leads to circular references and thus to an object which can not be JSON serialized and therefore this work around. Note the parameters for the action method, where the first indicates whether to instantiate a new *Zipcode* object or whether the operation involves editing an existing object. In addition to creating the session object, the only thing that happens is that the method instantiates a model object for the view.

To use session variables I must update the *Startup* class, and to serialize and deserialize JSON objects I add the following class to the *Models* folder:

```
public static class SessionExtensions
{
  public static void SetJson(this ISession session, string key, object
  value)
  {
    session.SetString(key, JsonSerializer.Serialize(value));
  }

  public static T GetJson<T>(this ISession session, string key)
  {
    var sessionData = session.GetString(key);
    return sessionData == null ? default(T) :
      JsonSerializer.Deserialize<T>(sessionData);
  }
}
```

that defines two extension methods to do the job. The view takes up some space, but I have chosen to show most of it as there are several things you should notice.

```
@model ZipcodeModel
@{
}
<div style="max-width:300px;margin:auto">
  @if (Model.Zipcode.Code == null)
  {
    <div class="head-text">Create Zip Code</div>
  }
  else
  {
    <div class="head-text">Edit Zip Code</div>
  }
  <form method="post">
    <input asp-for="IsNew" type="hidden" />
    <input asp-for="Page" type="hidden" />
    <input asp-for="Category" type="hidden" />
    <div class="input-line">
      <label asp-for="Zipcode.Code" class="search-text">Code:</label>
      <input asp-for="Zipcode.Code" style="width:60px" />
      <span asp-validation-for="Zipcode.Code" class="text-danger"></
      span>
    </div>
    <div class="input-line">
      <label asp-for="Zipcode.City" class="search-text">City</label>
      <input asp-for="Zipcode.City" style="width:200px" />
      <span asp-validation-for="Zipcode.City" class="text-danger"></
      span>
    </div>
    <div style="margin-top:10px;margin-bottom:5px">...</div>
    <div><select asp-items="Model.Municipalities1" asp-
    for="SelectedTags1"
      multiple="multiple" style="width:270px"></select></div>
    <div style=" margin-top: 20px;margin-bottom: 5px">...</div>
    <div><select asp-items="Model.Municipalities2" asp-
    for="SelectedTags2"
      multiple="multiple" style="width:270px;height:100px"></select></
      div>
    @if (Model.Error != null && Model.Error.Length > 0)
    {
      <div style="margin:10px;font-weight:bold;color:red">@Model.
      Error</div>
```

```
    }
    <div style="margin-top:20px">
      @if (!Model.IsNew)
      {
        <input type="submit" value="Remove"
        formaction="RemoveZipcode".../>
      }
      <input type="submit" value="Save" formaction="UpdateZipcode".../>
      <input type="submit" value="Cancel"
      formaction="CancelZipcode".../>
    </div>
  </form>
</div>
```

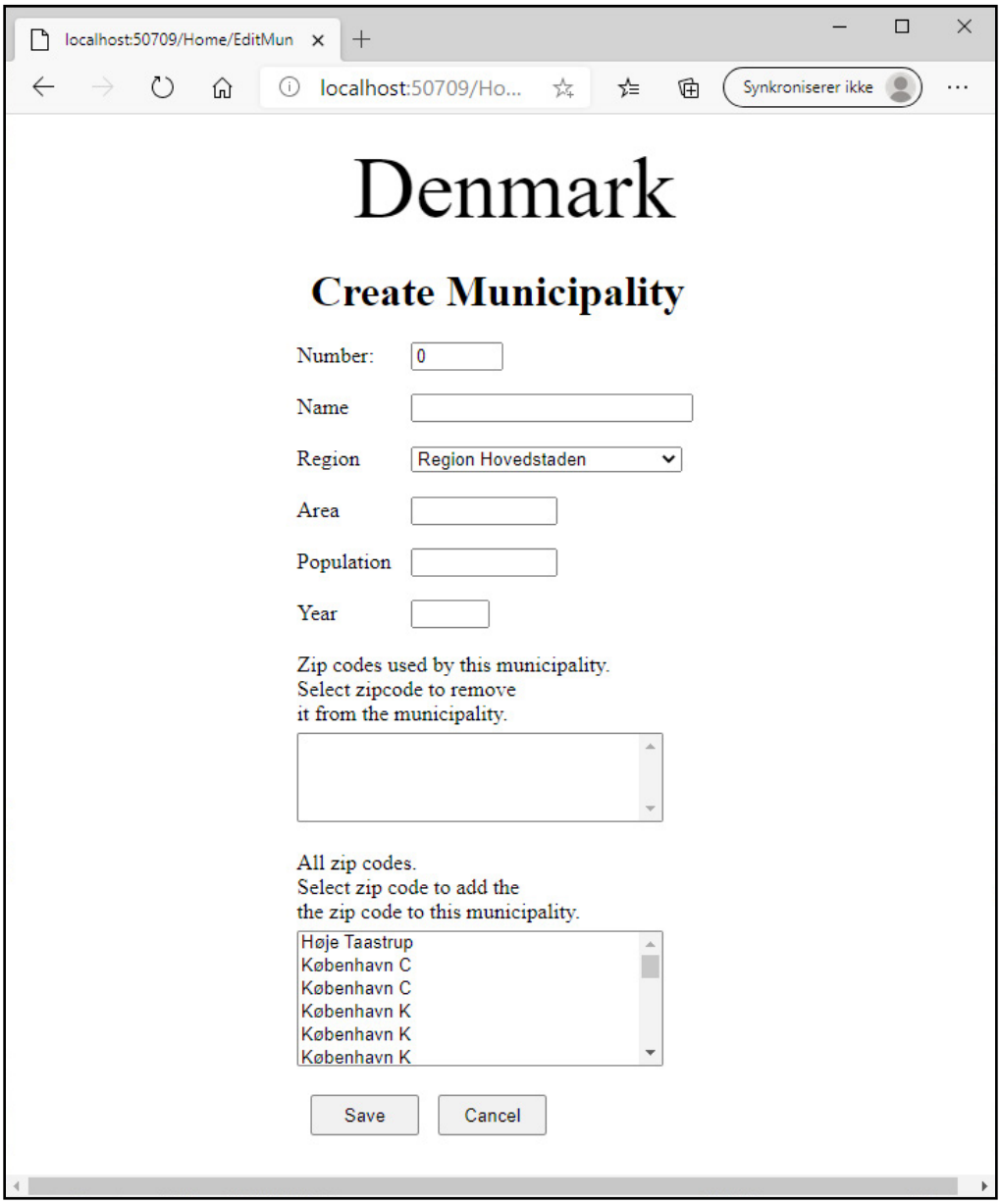
First note the use of an *if*-statement to determine which header to show and at the end of code to determine where to show the *Remove* button. An *if*-statement is also used to determine whether to show an error message. Also note that the form starts with three hidden fields. It is fields which are not shown in the browser, but the field is initialized with the value of a model property defined with *asp-for*. This value is again send to the server when the form is submitted. Also note how to define a HTML *select* element and then a list. The attribute *asp-items* indicate the collection which should be used to initialize the list with elements, and *asp-for* reference the array which will contains the keys for all selected elements. Finally denote that a form may have more submit buttons, when they references to action methods with different names. As an example, I will show the method for the *Save* button, as it is not quite simple:

```
[HttpPost]
public IActionResult UpdateZipcode(ZipcodeModel model)
{
  Zipcode zipcode = model.Zipcode;
  if (ModelState.IsValid)
  {
    List<int> refs = HttpContext.Session.GetJson<List<int>>("zipcode");
    foreach (int id in refs)
      zipcode.Municipalities.Add(Repository.Instance.Municipalities.
      Single(
      m => m.Id == id));
    if (model.SelectedTags1 != null)
    {
      foreach (int id in model.SelectedTags1)
      zipcode.Municipalities.RemoveAll(m => m.Id == id);
    }
    if (model.SelectedTags2 != null)
    {
      foreach (int id in model.SelectedTags2)
      {
        Municipality municipality =
          Repository.Instance.Municipalities.Single(m => m.Id == id);
        if (!model.Zipcode.Municipalities.Contains(municipality))
          zipcode.Municipalities.Add(municipality);
      }
    }
    bool ok;
    if (model.IsNew) ok = Repository.Instance.Insert(zipcode);
    else ok = Repository.Instance.Update(zipcode);
    if (!ok)
    {
      refs = new List<int>();
      foreach (Municipality m in zipcode.Municipalities) refs.Add(m.Id);
      HttpContext.Session.SetJson("zipcode", refs);
      return View("Zipcode", new ZipcodeModel {
        Error = "The database could not be updated", Zipcode = zipcode,
        Municipalities1 = new SelectList(zipcode.Municipalities,
        nameof(Models.Municipality.Id), nameof(Models.Municipality.
        Name)),
        Municipalities2 = new SelectList(Repository.Instance.
        Municipalities,
```

```
        nameof(Models.Municipality.Id), nameof(Models.Municipality.
        Name)) });
    }
    HttpContext.Session.Clear();
    return RedirectToAction("Zipcodes", "Home", new { region = model.
    Region,
      page = model.Page });
  }
  else return View("Zipcode", new ZipcodeModel { Zipcode = zipcode,
    Municipalities1 = new SelectList(zipcode.Municipalities,
    nameof(Models.Municipality.Id), nameof(Models.Municipality.Name)),
    Municipalities2 = new SelectList(Repository.Instance.
    Municipalities,
    nameof(Models.Municipality.Id), nameof(Models.Municipality.Name)) });
}
```

The method starts to set a reference to the *Zipcode* object, and you must note that it is a new object created when the view is submitted and not the same object which values was used to create the view. If the state of the form is legal (the two fields for code and city) the session variable is used to initialize the *Zipcode* object with the *Municipality* objects it had before. The array *SelectedTags1* contains the key for these municipalities which must be removed, and the array *SelectedTags2* the key of these municipalities which must be added. After that the object is saved in the database, either by inserting a new row or by updating an existing row. The operation can fail, for example if you try to insert a new row with an existing zip code. If so the method must post back the form, but before the session variable must be updated. If the database operation is performed correct the session variable is removed and the application returns to the page showing the zip codes. The last two action methods are simpler and I will not show the code here.

Maintenance of municipalities happens in the same way and using a view that look like the above. If you open the page for municipalities and clicks *Create* you get the following form to create a new municipality (see below). I will not show the code for this view or the code for action methods in the controller as the code in principle is identical to code used to maintenance zip codes.



After that the application is finish and ready for use, if it was otherwise installed on a real web server.

Problem 1 The World

In C# 6 problem 2 you write a program used to maintenance a database with information about countries. The database has two tables, where the one contains information about the continents (only there names) where the other contains information about countries. May be you should read the problem formulation in C# 6. In this problem you should write another version of the program, but this time it must be a web application.

Before you start you must change the database and expand it with a new table:

```
use world;

create table Currency (
  Curr_Code NCHAR(3) PRIMARY Key,
  Curr_Name NVARCHAR(30) NOT NULL,
  Curr_Rate DECIMAL(8,2)
);
```

The online code for this book has a semicolon separated text file *Currency.txt* which contains lines for currencies with three fields:

1. Currency name
2. Currency code
3. Currency exchange rate relative to Danish crowns

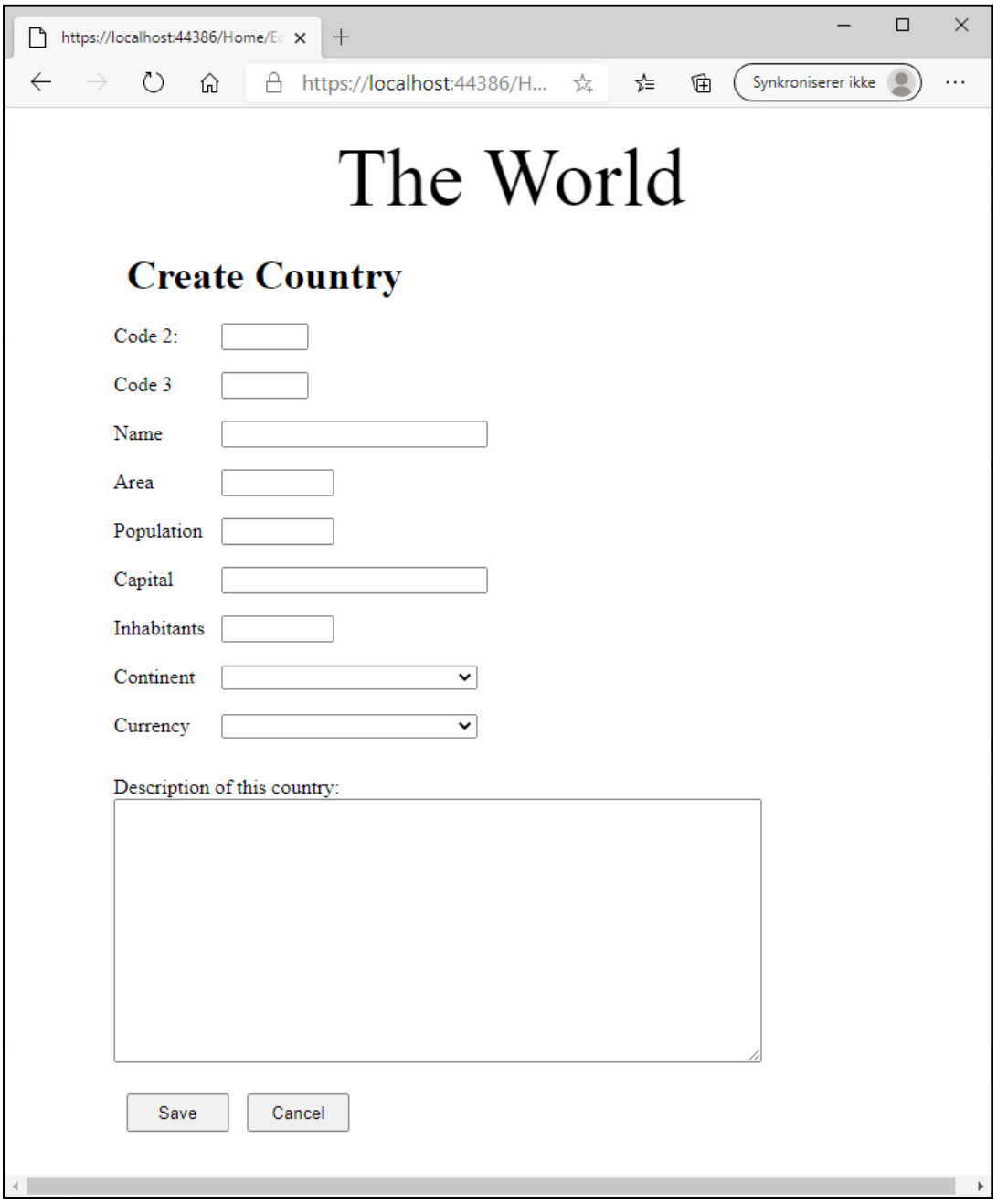
The exchange rates are not completely new and you can update them if you like or you can find the right exchange rates for your currency.

Write a console application which uses the text file and updates the table *Currency*.

Next you must write the web application, and you can proceed in the same way as in the example *Denmark* above. The current program is actually simpler as the database does not have many-to-many relationships, which is why the methods for maintaining the database become simpler. In the same way as the application *Denmark* the application should have five views:

1. a start page
2. a view which in a table shows all countries
3. a view which in a table shows all currencies
4. a view to show / modify a country
5. a view to show / modify a currency

As an example the view to create / modify a country could be something like the following:



In order to be used, the program lacks (at least) one important feature, namely an easy way to periodically update the exchange rates. This issue has been postponed to a sequel to this book.

6 THE BOOKSTORE

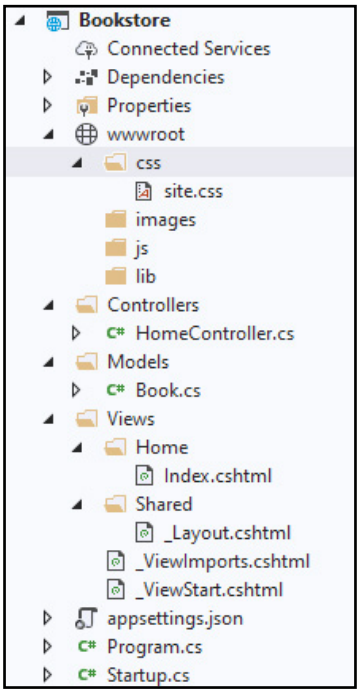
As the last example in this book I will develop a web application for a bookstore. It is a greatly simplified example of a web store where users can put books from an antique bookstore in a shopping cart. The goal, of course, is to show the development of a web application using ASP.NET Core to serve as a typical example of a web application. When I have chosen a bookstore it is only because I in C# 6 have created a book database with about 140 books and thus have a database with some data available. The task is solved through the following iterations:

1. Project start
2. The models and the repository
3. The store
4. The shopping cart
5. Check out
6. The maintenance of the database
7. The last things

In relation to a real online store, there are many shortcomings and limitations where the most important thing is of course everything regarding payment, but there are also a number of other factors such as security and finally the user interface that is far from what you could wish for.

6.1 PROJECT START

I start the development by creating a new *ASP.NET Core Web Application* in Visual Studio, a project I have called *Bookstore* and is created using an empty template and I have added the following folders and files:



Then I have modified the following files:

1. *Startup.cs*
2. *_Layout.cshtml*
3. *_ViewImports.cshtml*
4. *site.css*
5. *Index.cshtml*

and added an image to the folder *wwwroot\images*. The *Startup* class is modified to support a MVC template for a web application:

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

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

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

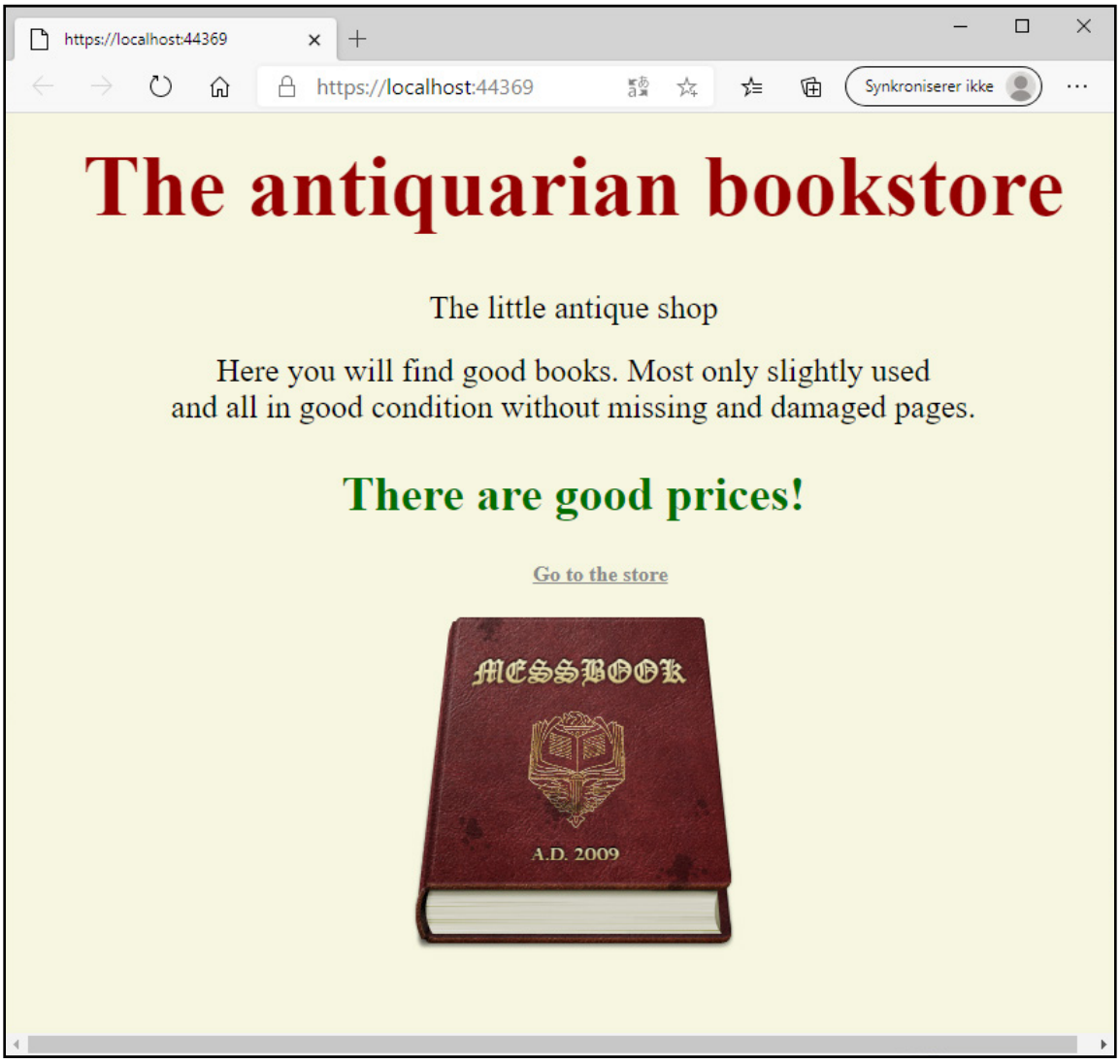
_Layout is updated with a header and a reference for the style sheet, and *_ViewImports.cshtml* is updated as:

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

Some styles are added to the style sheet, and finally is the view *Index.cshtml* modified as:

```
<div class="centered">
  <div style="font-size:24px;margin:20px">The little antique shop</div>
  <div style="font-size:24px">Here ... </div>
  <div style="font-size:36px;margin:30px;font-
  weight:bold;color:darkgreen">
    There are good prices!</div>
  <dir class="start-4"><a asp-action="Store" class="link-text">
    Go to the store</a></dir>
  <div class="start-5">
    <img src="~/images/Book.png" />
  </div>
</div>
```

If you the run the application the start page is a shown below and the result is a web application project which opens a start page in the browser:



6.2 THE MODELS AND THE REPOSITORY

As mentioned above I will use the database *Library* from the book C# 6, but instead of using the database directly I have created a copy called *Bookstore* (in SQL Server Management Studio I have created a backup and restored the backup to an empty database *Bookstore*). The database has five tables:

1. Book
2. Publisher
3. Category
4. Author
5. Written

where the last is a many-to-many relationship between *Book* and *Author*. A book must have a price and for that and for later to administrate customers I have added two new tables:

```
create table Price (
  Book_id int PRIMARY KEY,
  Price DECIMAL(8,2)
);

create table Zipcode (
  Code NCHAR(4) PRIMARY KEY,
  City NVARCHAR(30) NOT NULL
);
```

To add data to the two tables I have written a console application called *CreateDB* which initializes the two tables, the first with a random price for each book and the other with the Danish zip codes.

Next I have written four model classes, one for each of the first four tables and as example is shown the class *Book*, where I have not shown validations and overrides of the standard methods:

```
public class Book : IComparable<Book>
{
  public Book()
  {
    Authors = new List<Author>();
  }

  public int Id { get; set; }
  public string Isbn10 { get; set; }
  public string Isbn13 { get; set; }
  public string Title { get; set; }
  public int? Year { get; set; }
  public int? Edition { get; set; }
  public int? Pages { get; set; }
  public int? Count { get; set; }
  public string Text { get; set; }
  public Decimal? Price { get; set; }
  public Category Category { get; set; }
  public Publisher Publisher { get; set; }
  public List<Author> Authors { get; private set; }
}
```

As long the program does not handle customers and the shopping card only these four model classes are needed. The folder *Models* also has a class *Repository* written as a singleton and the class consists preliminary of four properties which are collections of model objects:

```
public class Repository : RepositoryBase
{
  private static readonly Repository instance = new Repository();

  static Repository()
  {
  }

  private Repository()
  {
    try
    {
      Initialize();
    }
    catch
    {
      throw new Exception("Unable to connect to database");
    }
  }

  public static Repository Instance
  {
    get { return instance; }
  }

  public List<Book> Books { get; private set; }
  public List<Author> Authors { get; private set; }
  public List<Publisher> Publishers { get; private set; }
  public List<Category> Categories { get; private set; }
  ...
}
```

These properties are initialized in the method *Initialize()* called from the constructor, where the method reads the content of the database and creates the model objects.

The repository must also contains methods used to update the database tables. These methods are not added yet and is postponed until I need to.

Note that I must use NuGet to set references for the assemblies for SQL Server.

With the above data preparations, I am ready to write the first views.

If I in the action method for the controller add the following statement

```
Bookstore.Models.Repository rep = Bookstore.Models.Repository.Instance;
```

and set a breakpoint it is possible the test the class *Repository* such I can be sure the database is read and the model objects are created. When done the statement must be removed as it should not be there for the next iterations.

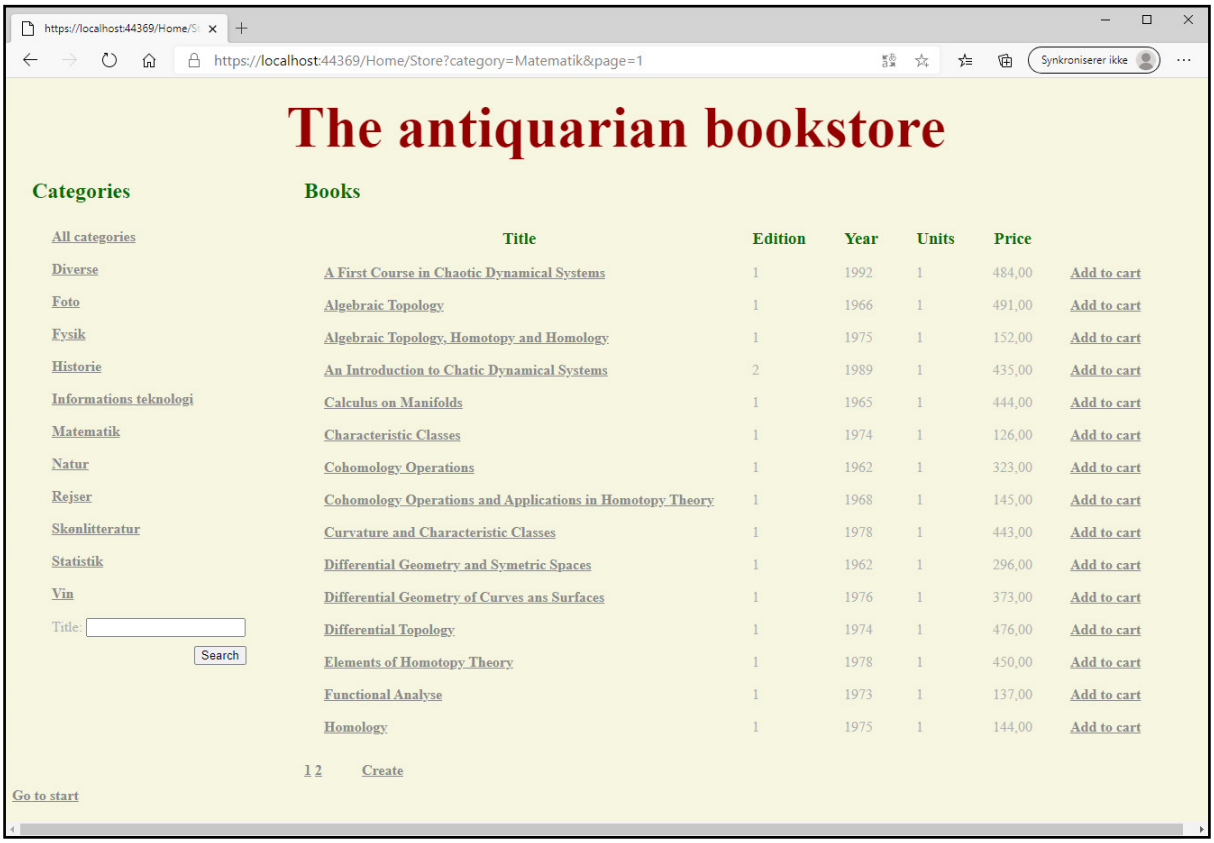
6.3 THE STORE

In principle a web store is a page with a list of products where the user can scroll through the list and add products to a shopping cart. In this iteration I will implement a view which shows a product list and where the user can select a product and opens a details page.

When the user om the start page clicks on *Go to Store* the program opens the page shown below. The view is implemented in the same way as in the example in the previous chapter:

1. a class *Paging* used to implement page navigation
2. a class *BooksModel* in a subfolder *Models\ViewModels* used as model for the view
3. two action methods in the controller called *Store*, one for HTTP GET and one for HTTP POST
4. the view which in the same way as in the previous chapter is a table with two nested tables

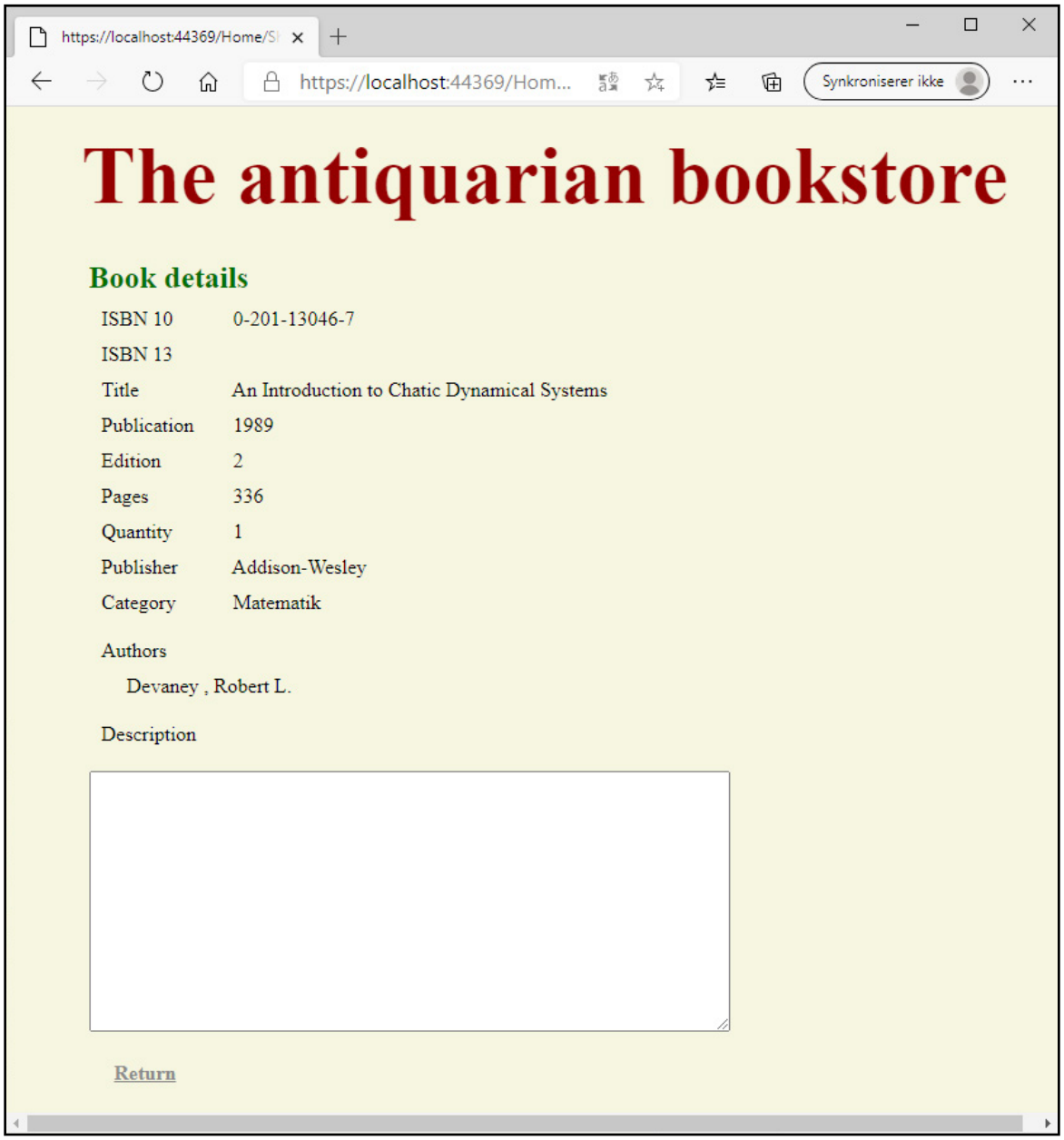
The view uses book categories as a filter, and the view below shows the result after the user has click on *Mathematics*. There is also a search field where to search books for the title. All books, which contains the search text in the title matches the search.



The left column in the book table is a link used to add the book to the shopping cart and is not used for the moment, but it is the subject for the next iteration. The same goes for the bottom link *Create*, and in fact this link should not be there at all, as a user should not be able to create new books. It should only be possible for an administrator and is something that will be implemented later.

If the user clicks on a book title the result is the page shown below. It is a view which shows details for a book, and is a simple view and is simple to implement. It requires a model for the view called *BookModel* and one action method in the controller called *ShowBook*. The action method has the book's *id* as a parameter, and otherwise the only challenge is that if the user clicks on the *Return* link, the program must return to the same place (same page and search) as it was before. This problem is solved with parameters for the action method.

After this iteration a user can go to the bookstore and navigates the books in the store. There is also a possibility for a simple search and the user can open a page to show information for a book in the store.



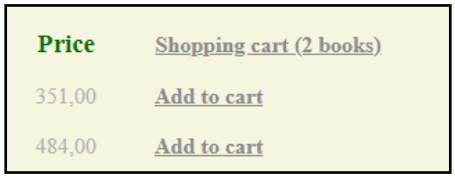
6.4 THE SHOPPING CART

In this section I will implement the shopping cart, and the pattern is the product page (the book list) with a link to add items to the shopping cart. The shopping cart is a page which shows the content of the cart and is open when the user clicks on the link for add an item to the cart. From this page the user can go back to buy more items or the user can go to check out to pay for the items in the cart:

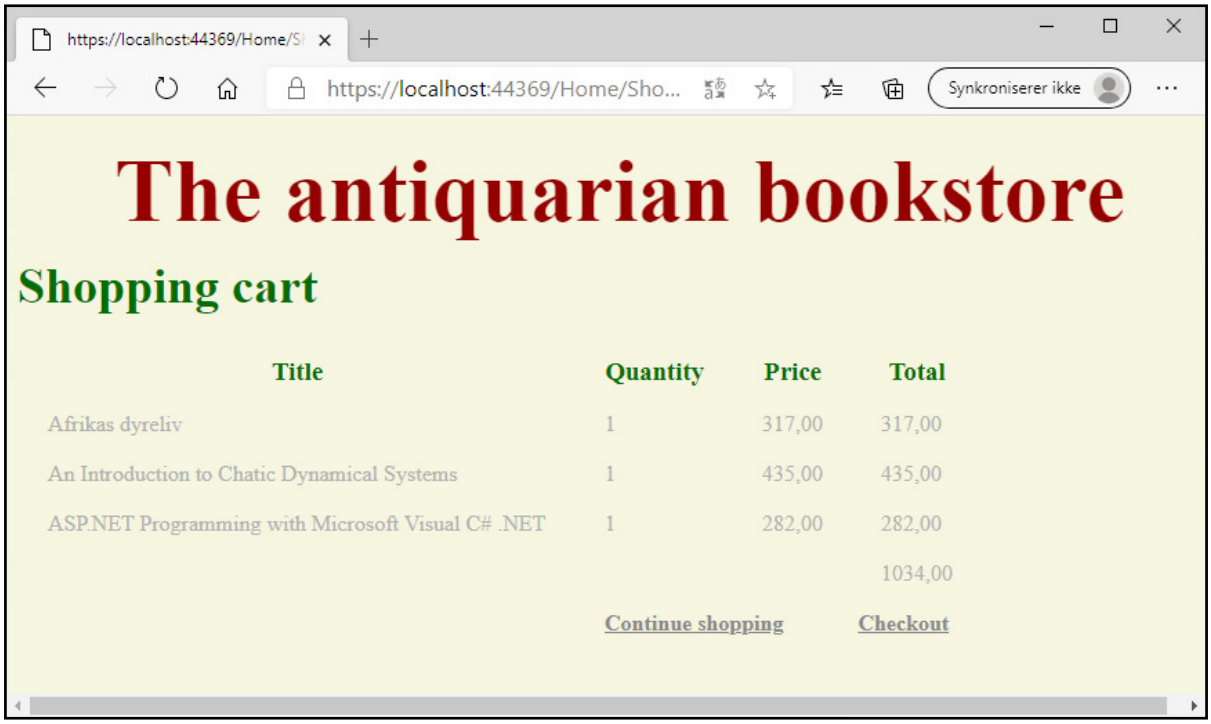
This means that I must update the links in the book list and add a new page for the shopping cart.

The shopping cart must be stored somewhere and I will use a session variable. A session variable means that the contents of the shopping cart are only stored in memory and it automatically disappears if the user leaves the page. Conversely, it also means that if the user pauses and leaves the application for more than 20 minutes (or whatever may have been selected) then the content is gone and the shopping cart is empty. Thus, there are both advantages and disadvantages, and there are actually several web stores where the content is retained even if the user has left the application.

To implement the shopping cart the page for the store is expanded with a link:



and in this case it shows that two books are added to the cart. If the user clicks *Add to cart* for a book or clicks on the new link the shopping cart is shown:



It is quite simple to implement the function and most of the work lies in the controller. Both action methods for the *Store* view starts with the following statement which try to read a session variable for the shopping cart:

```
List<CartItem> cart =
  HttpContext.Session.GetJson<List<CartItem>>("cart") ?? new
  List<CartItem>();
```

It is a list with elements of the type *CartItem*, which is the key for a book added to the cart and the number of books if the same book is added several times:

```
public class CartItem
{
  public int Bookid { get; set; }
  public int Count { get; set; }
}
```

Note that it is not possible to enter a quantity when the use puts a book in the cart, since for an antique bookstore you typically only buy one copy of the book and typically the bookstore will only have the same copy. If you want to buy more copies (and the bookstore has several copies), you must put the same book in the cart several times.

In reality, it does not make sense that in an antique bookstore you can have several units of the same book. If you have several examples of a book, they will appear as different products. You should ignore this detail and not think so much about the fact that the books are used books.

Then there is the action method for the shopping page which is not entirely simple:

```
public IActionResult Shopping(int bookid, string search, string category,
  int page = 1)
{
  if (bookid != 0)
  {
    Book book = Repository.Instance.Books.Single(b => b.Id == bookid);
    if (book != null)
    {
      List<CartItem> cart = HttpContext.Session.
      GetJson<List<CartItem>>("cart")
        ?? new List<CartItem>();
      bool found = false;
      for (int i = 0; i < cart.Count && !found; ++i)
        if (cart[i].Bookid == bookid)
        {
          ++cart[i].Count;
          found = true;
        }
      if (!found) cart.Add(new CartItem { Bookid = bookid, Count = 1 });
      HttpContext.Session.SetJson("cart", cart);
      List<Book> books = new List<Book>();
      foreach (CartItem item in cart)
        books.Add(Repository.Instance.Books.Single(b => b.Id == item.
        Bookid));
      return View(new CartModel { Books = books, Items = cart,
        Search = search, Category = category, Page = page });
    }
  }
  else if (bookid == 0)
  {
    List<CartItem> cart = HttpContext.Session.
    GetJson<List<CartItem>>("cart")
      ?? new List<CartItem>();
    if (cart.Count > 0)
    {
      List<Book> books = new List<Book>();
      foreach (CartItem item in cart)
        books.Add(Repository.Instance.Books.Single(b => b.Id == item.
        Bookid));
      return View(new CartModel { Books = books, Items = cart,
        Search = search, Category = category, Page = page });
    }
  }
  return RedirectToAction("Store", new { search = search,
    category = category, page = page });
}
```

The parameters is the id for the selected book or 0 if no book is selected (the user has clicked on the link for the shopping cart) and the parameters are needed to return to the store with the same selections. If *bookid* is different from 0 the method try to read the book in the database, and if the book is found (and it should be) the method tests where the book is already in the cart. If so the quantity in the cart for that book is incremented and else the book must be added to the cart. Next a book list for the books in the cart is determined and the view is created using the following view model:

```
public class CartModel
{
  public List<Book> Books { get; set; }
  public List<CartItem> Items { get; set; }
  public string Search { get; set; }
  public string Category { get; set; }
  public int Page { get; set; }
}
```

The code for the view starts with:

```
@model CartModel
@{
  decimal total = 0;
}
  <div style="max-width:1000px;margin:auto">
    <div style="...">Shopping cart</div>
    <table>
      <tr>
        <th>Title</th>
        <th>Quantity</th>
        <th>Price</th>
        <th>Total</th>
      </tr>
      @for (int i = 0; i < Model.Items.Count;
        total += Model.Books[i].Price.Value * Model.Items[i].Count, ++i)
      {
        <tr>
          <td>@Model.Books[i].Title</td>
          <td>@Model.Items[i].Count</td>
          <td>@Model.Books[i].Price</td>
          <td>@(Model.Books[i].Price.Value * Model.Items[i].Count)</td>
        </tr>
      }
```

and here you should note how to define a local variable and how to use the variable to calculate the total. Also note how to insert a Razor expression for each line in the table.

There is three things left. I must update the *Startup* class to support session variables and the class *BooksModel* must be expanded with a property for shopping cart. This property is necessary in the view *Store* to show the number of books in the cart. The last thing is that it should only be possible to add a book to the shopping cart if the number of books is positive and if the book has a price. To solve that I have added a method to the *Store* view:

```
@model BooksModel
@{
  bool HasItems(int id)
  {
    Book book = Model.Books.Single(b => b.Id == id);
    if (book == null) return false;
    int count = 0;
    foreach (CartItem item in Model.Cart)
      if (item.Bookid == id)
      {
        count = item.Count;
        break;
      }
    return book.Count - count > 0 && book.Price != null;
  }
}
```

When I generate the table with books I can then test where there should be an *Add to cart* link.

You should note that it does not completely solve the problem as there is a concurrency problem. If a user wants to put a book in the cart, you can not see (not test) whether another user has put the same book in the cart. You see here another problem with session variables, that they are linked to the session and not the application, so solving the problem requires more than using a simple session variable. In the current case, you will first be able to find out that the book has actually been sold to another user when you go to checkout.

6.5 CHECKOUT

When the program should ignore payment it should be simple to implement this function, and the following must happen:

1. From the page showing the shopping cart the user clicks the link *Checkout*.
2. It requires the user to login as a customer.
3. The program shows an order page which the user must accept.
4. If so the database must be updated and the program sends the user an email.

As so the implementation requires:

1. Update the database with tables for customers
2. Implement the model classes for customers and update the class *Repository*
3. Login
4. Create a new to show the order
5. Send a mail to the user

The database

The database is expanded with two new tables, one for customers and one for orders:

```
use Bookstore;

create table Customer (
  Mail NVARCHAR(50) PRIMARY KEY,
  Salt NVARCHAR(50) NOT NULL,
  Pass NVARCHAR(200) NOT NULL,
  Name NVARCHAR(100) NOT NULL,
  Addr NVARCHAR(100) NOT NULL,
  Code NCHAR(4) NOT NULL,
  Phon NVARCHAR(30),
  FOREIGN KEY (Code) REFERENCES Zipcode ON DELETE CASCADE
);

create table Orders (
  Ord_Id INT NOT NULL IDENTITY(1,1) PRIMARY KEY,
  Book_id INT,
  Mail NVARCHAR(50),
  Date DATETIME,
  Quan INT DEFAULT 1,
  Price DECIMAL(8, 2),
  FOREIGN KEY (Book_id) REFERENCES Book ON DELETE CASCADE,
  FOREIGN KEY (Mail) REFERENCES Customer ON DELETE CASCADE
);
```

The latter is in fact a many-to-many relationship between *Book* and *Customer* and shows which books a customer has purchased. When there is a surrogate key it is because a customer in principle can buy a copy of the same book several times the same day. The price is saved as it could later be changed. Also note the customer table which has a column for the password. The password is saved as one way hash in the same way as shown in previous books.

The models and the repository

The folder *Models* is expanded with three new types. The first defines a customer, where I have not shown the validation rules and the standard overrides:

```
public class Customer : IComparable<Customer>
{
  public Customer()
  {
    Orders = new List<Order>();
  }

  public string Mail { get; set; }
  public string Name { get; set; }
  public string Address { get; set; }
  public Zipcode Zipcode { get; set; }
  public string Phone { get; set; }
  public List<Order> Orders { get; private set; }

  ....
}
```

Note that the type not have properties for password and salt as these values should only be used when a customer logs in. The class also has a list with references for orders for this customer. You should note that there is an order (an *Order* object) for each book which a customer has purchased and that the class should therefore perhaps be called something like *OrderLines*.

The other two new model classes are called *Order* and *Zipcode* and are not shown here as they are trivial.

Then there is the *Repository* class which is expanded with a new property representing all zip codes:

```
public List<Zipcode> Zipcodes { get; private set; }
```

The method *Initialize()* is changed accordingly so that the list of zip codes is initialized. You should note that the class does not have similar properties for customers and orders as information about customers must only be read in the database when a customer logs in.

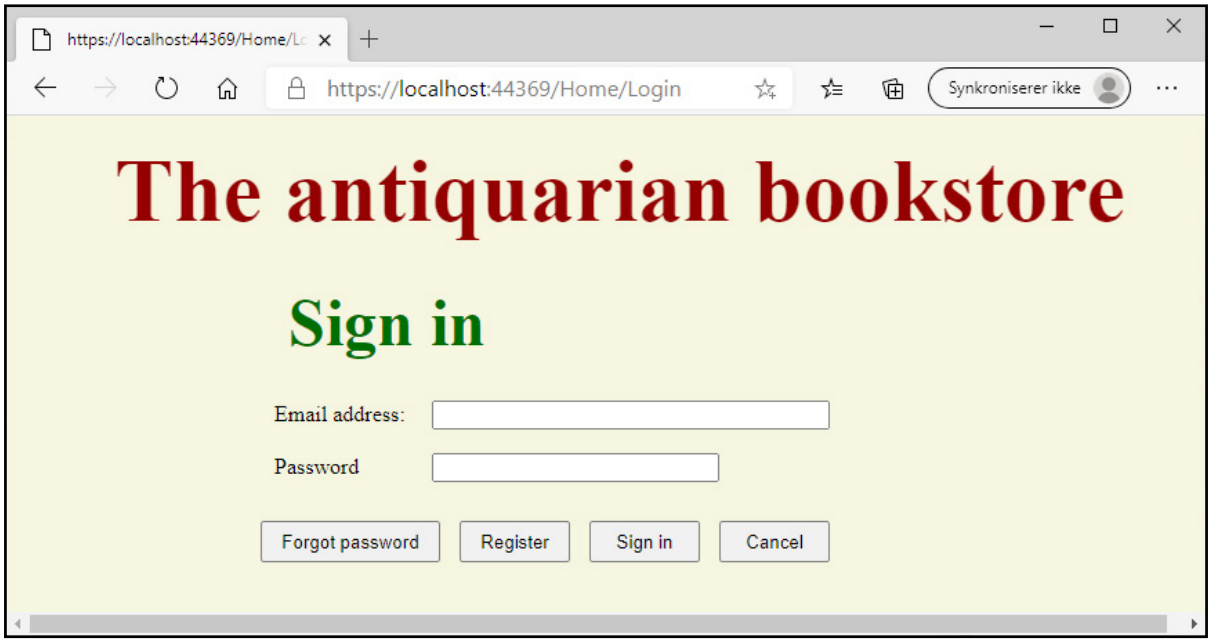
The class is expanded with 4 methods:

```
public Customer Login(string mail, string pass) { ... }
public bool Insert(Customer customer, string passwd) { ... }
public bool Update(Customer customer, string passwd = null) { ... }
public bool Insert(Customer customer, List<Order> orders) { ... }
```

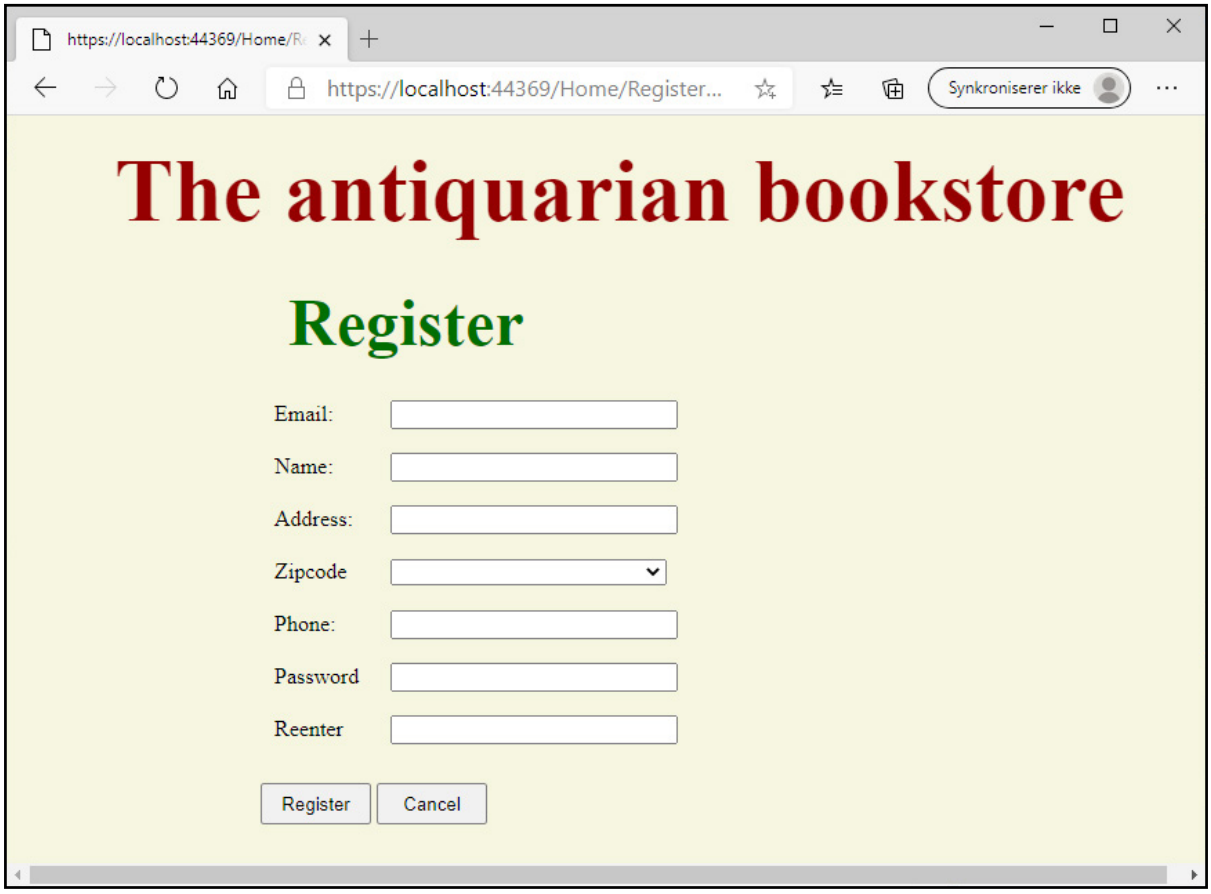
The first is for a customer to login and return and object representing the customer. The second creates a new customer and is used when a customer checks out and not yet is created as a customer. The third is used to update the customers data and can specially be used to change the customer's password. The last is used to add order lines for this customer. None of the methods are particularly complex. However, the latter must also update the *Book* table and decrements the number of copies for the current book.

Login

The start page is expanded with a new link used to login:



If the user is created in the database and enter user id (email address) and password the user can login as a customer, and the program returns to the start page, and the login link is change to a logout link and shows the users name. If the user has forgotten the password the user may clicks the button for that (is explained below), and if the user is not created as a customer the user may clicks on *Register* to be created as a new customer (se below). When the user has signed in or is registered as a customer a *Customer* object is created and stored in a session variable. In this way all actions methods in the controller can easily check where a user is signed in.

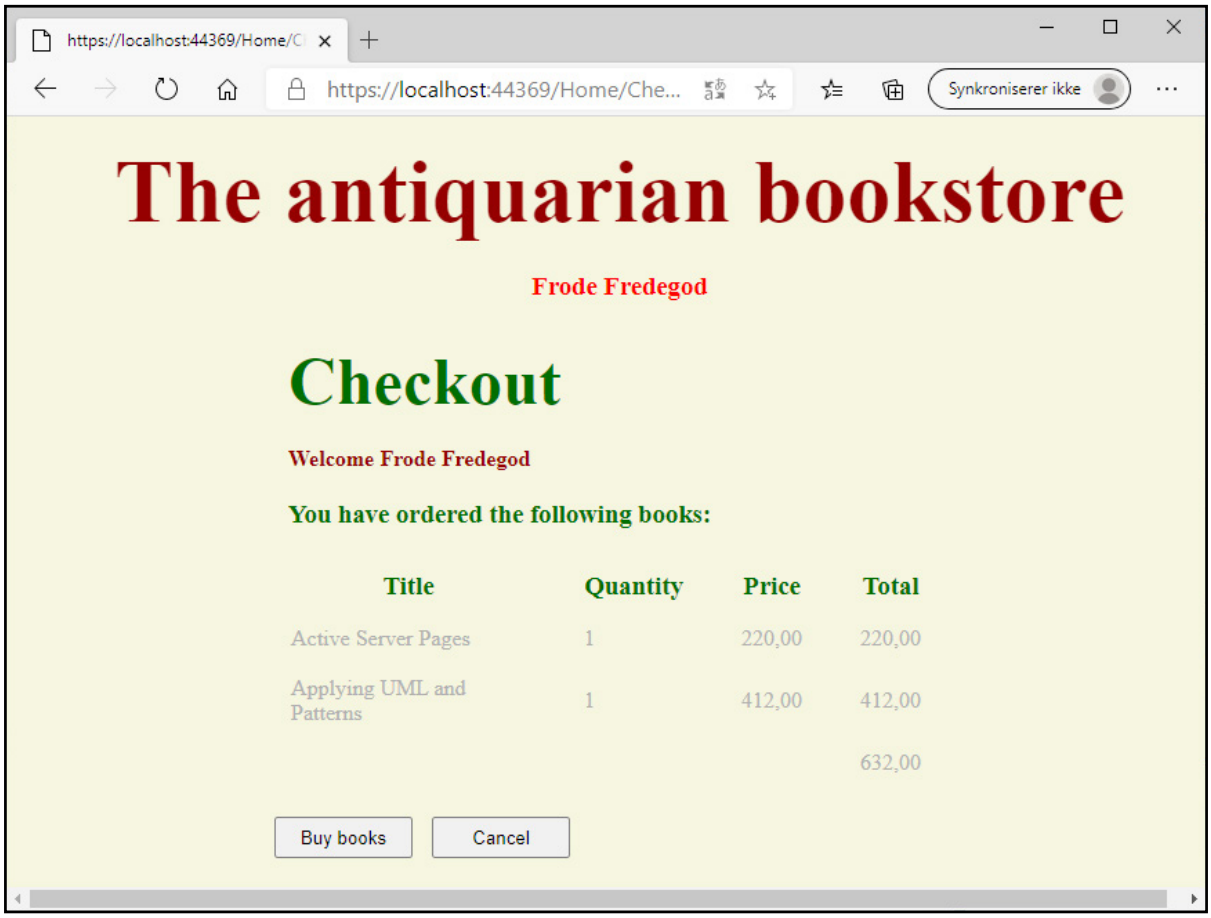


It should also be possible to show if a user is signed in on other pages than the start page. For that the *_Layout.cshtml* is updated to reference the session variable:

```
<!DOCTYPE html>
@using Microsoft.AspNetCore.Http
@using Bookstore.Tools;
@{
  Customer customer = Context.Session.GetJson<Customer>("Customer");
  string name = customer == null ? "" : customer.Name;
}
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>@ViewBag.Title</title>
  <link href="~/css/site.css" rel="stylesheet" />
</head>
<body>
  <div class="layout-head">The antiquarian bookstore</div>
  @if (name != null)
  {
    <div class="layout-name">@name</div>
  }
  <div>
    @RenderBody()
  </div>
</body>
</html>
```

Checkout view

If the user is on the page for the shopping cart and want to check out there is two possibilities. If the user is signed in the result is the page shown below showing the books added to the shopping cart. Else the user must sign in as explained above.

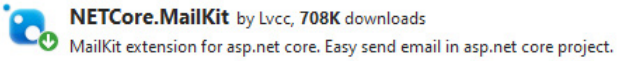


The page is quite simple and is almost the same as the page for the shopping cart, but in a real web store this is where the payment takes place. In this case, all that needs to happen is if the customer accepts and clicks the button *Buy books*.

When the user clicks the button the action method must

1. update the database and remove the session variable for shopping cart
2. send an email to the customer which shows the purchased books

I will start with the last and some classes are added to the folder *Tools*. To send an email I have to use NuGet to add a tool to the project:



Sending emails requires configuration of the tool and for that I have added the following class:

```
public class EmailConfiguration
{
  public string From { get; set; }
  public string SmtpServer { get; set; }
  public int Port { get; set; }
  public string UserName { get; set; }
  public string Password { get; set; }
}
```

and you can se that you need to tell:

1. where the email come from
2. which SMTP server to user
3. the port number for this server
4. the username in the server
5. the password for the server

The values for these properties must be stored somewhere and here the configuration file *appsettings.json* is used:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "EmailConfiguration": {
    "From": "...",
    "SmtpServer": "...",
    "Port": ...,
    "Username": "...",
    "Password": "..."
  },
  "AllowedHosts": "*"
}
```

where you must enter values for your configuration. As the last thing concerning the configuration the *Startup* class must be updated width a configuration and a new service

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

  public IConfiguration Configuration { get; }
  public void ConfigureServices(IServiceCollection services)
  {
    services.AddControllersWithViews();
    services.AddDistributedMemoryCache();
    services.AddSession();
    services.AddSingleton(Configuration.GetSection("EmailConfiguration").
      Get<EmailConfiguration>());
    services.AddScoped<IEmailSender, EmailSender>();
  }

  ...
}
```

Next I need a class representing a mail with a list of recipients of mail (mail addresses), a subject and the message:

```
using System.Collections.Generic;
using System.Linq;
using MimeKit;

namespace Bookstore.Tools
{
  public class Message
  {
    public List<MailboxAddress> To { get; set; }
    public string Subject { get; set; }
    public string Content { get; set; }
    public Message(IEnumerable<string> to, string subject, string content)
    {
      To = new List<MailboxAddress>();
      To.AddRange(to.Select(e => new MailboxAddress(e)));
      Subject = subject;
      Content = content;
    }
  }
}
```

and last a class used to send the mail. This class is usually defined by an interface:

```
public interface IEmailSender
{
  void SendEmail(Message message);
}
```

and can be implemented as:

```
using MailKit.Net.Smtp;
using MimeKit;

namespace Bookstore.Tools
{
  public class EmailSender : IEmailSender
  {
    private readonly EmailConfiguration config;

    public EmailSender(EmailConfiguration emailConfig)
    {
      config = emailConfig;
    }

    public void SendEmail(Message message)
    {
      var emailMessage = CreateEmailMessage(message);
      Send(emailMessage);
    }

    private MimeMessage CreateEmailMessage(Message message)
    {
      var emailMessage = new MimeMessage();
      emailMessage.From.Add(new MailboxAddress(config.From));
      emailMessage.To.AddRange(message.To);
      emailMessage.Subject = message.Subject;
      emailMessage.Body = new TextPart(MimeKit.Text.TextFormat.Html)
        { Text = message.Content };
      return emailMessage;
    }

    private void Send(MimeMessage mailMessage)
    {
      using (var client = new SmtpClient())
      {
        try
```

```
        {
          client.Connect(config.SmtpServer, config.Port, true);
          client.Authenticate(config.UserName, config.Password);
          client.Send(mailMessage);
        }
        catch
        {
          throw;
        }
        finally
        {
          client.Disconnect(true);
          client.Dispose();
        }
      }
    }
  }
}
```

You should note that the method *CreateMailMessage()* define the message as HTML. With all this the action method in the controller can be written:

```
[HttpPost]
public IActionResult Checkout(CheckoutViewModel model)
{
  Customer customer = HttpContext.Session.GetJson<Customer>("Customer");
  List<CartItem> cart = HttpContext.Session.GetJson<List<CartItem>>("c
  art");
  if (customer != null && cart != null)
  {
    List<Order> orders = new List<Order>();
    foreach (CartItem item in cart)
    {
      Book book = Repository.Instance.Books.Single(b => b.Id == item.
      Bookid);
      orders.Add(new Order { BookId = book.Id, Date = DateTime.Now,
        Quantity = item.Count, Price = book.Price.Value });
    }
    try
    {
```

```
      Message message = new Message(new string[] { customer.Mail },
        "The antiquarian bookstore", BuildMessage(customer, orders));
      emailSender.SendEmail(message);
      if (Repository.Instance.Insert(customer, orders))
      {
        HttpContext.Session.Remove(("cart"));
        return RedirectToAction("Index");
      }
      model.Error = "Unable to update database";
    }
    catch
    {
      model.Error = "Unable send mail to customer";
    }
  }
  model.Error = "You must sign in or the cart is empty";
  return View(model);
}
```

The method takes up a lot of space, but is otherwise simple enough. You should primarily notice how the method sends an email and how the method forms the message with the *BuildMessage()* method. I have not shown the method here, but created the text as an HTML text.

Other things

Two things are missing:

1. send a mail to a user if the user has forgotten the password
2. edit customer information

When I have already implemented then functionality to send a mail the first is quite simple and consists solely in implementing a new action method which sends the user a new random generated password and update the database:

```
public IActionResult LoginForgot(LoginViewModel model)
{
  if (model.Mail != null && model.Mail.Length > 0 && IsMail(model.Mail))
  {
    try
    {
      string passwd = NewPassword();
      if (Repository.Instance.ChangePassword(model.Mail, passwd))
      {
        Message message = new Message(new string[] { model.Mail },
          "The antiquarian bookstore", NewPassword());
        emailSender.SendEmail(message);
        model.Error = "We have sent you a new password";
      }
      else model.Error = "Unable to change password";
    }
    catch
    {
      model.Error = "Could not sent your new password";
    }
  }
  model.Passwd = "";
  return View("Login", model);
}
```

To update the database the *Repository* class is expanded with a new method.

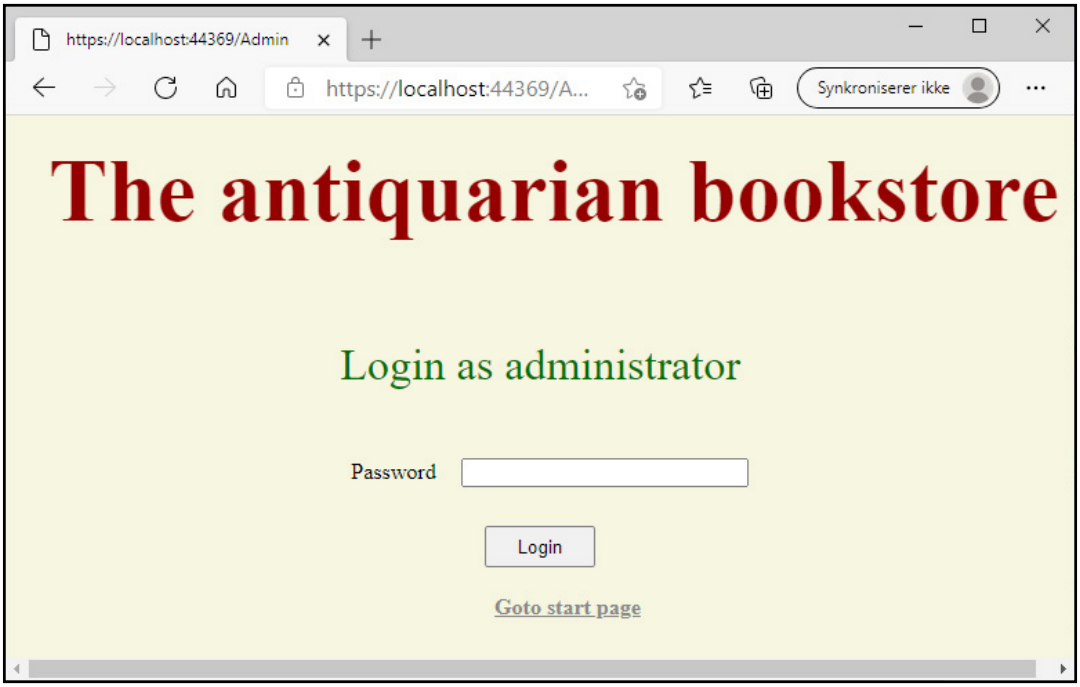
When a user sign in the login link on the start page is changed to two other links, one for logout and one to show and edit customer information:

The field for email is read only as it should not be possible to change the email as it is the key. The page also shows books which this customer has purchased. I do not have to explain the implementation here as there is nothing new.

6.6 MAINTENANCE THE DATABASE

After the above iteration, everything regarding the users' use of the application is in principle finished where the user can go to the web store, find books and put them in the shopping cart. The user can log in and go to checkout to buy the books. What is left is the administration where the store must be able to maintain the warehouse and add new books, and with IT terms it means maintaining the database.

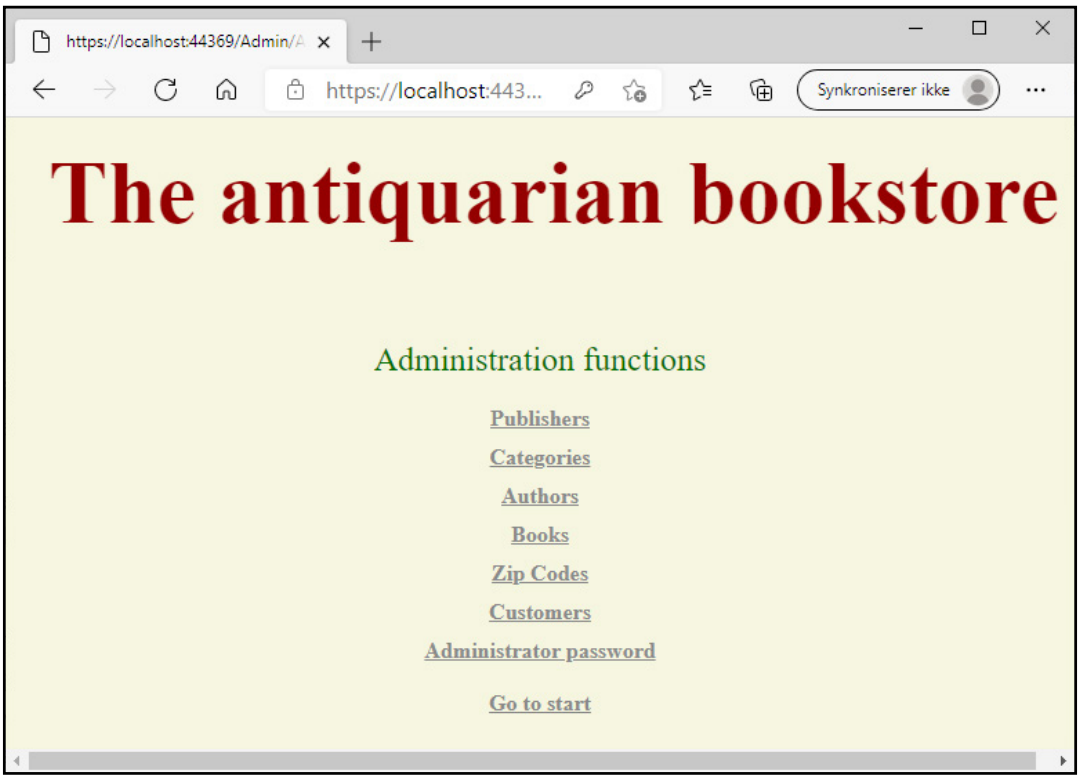
You should note that this is very typical for a web application, where there is a front end which is the part of the application the users sees, and a back end which is only visible for and can be used of the application's administrator. Often it is two applications separated from each other (and I think it should be), but sometimes, for the sake of simplicity, you will leave it as the same application and where you have to log in as an administrator to perform administrator tasks. I will apply this approach. To log in as administrator there must be a way to log in, and I have chosen to place a link on the home page (not absolutely the best solution) so that by clicking on this link you get to an administrator page for login:



If you here clicks on *Go to start page* you returns to the front end, but else you must enter a password for the administrator. The database has a table *Admin* with three columns, where the first column is for an email address while the two others are for password and salt. The meaning with the first column is that there can be several administrators, but in this case only one administrator must be possible so that the first column could actually be deleted. I have kept it if for no other reason then, so that it could be used in a later change of the program. The administrator is created the first time a user tries to log in as an administrator, and the password entered is then the administrator's password.

You should note that this solution is simple and even too simple and the same applies to customers and the administration of their logins. The administration of users of a website is by no means simple and among other things it will often be the case that users must have different rights, that for a website typically means that which pages you can open depends on who you are. Therefore, all that concerning user administration is actually built into an API in ASP.NET and requires a something to get acquainted with it, and more than there is room for in this book, but for practical web development it is a part that belongs.

If you in this case enter a password and clicks login you get the following page which is the administrator menu with 7 functions:

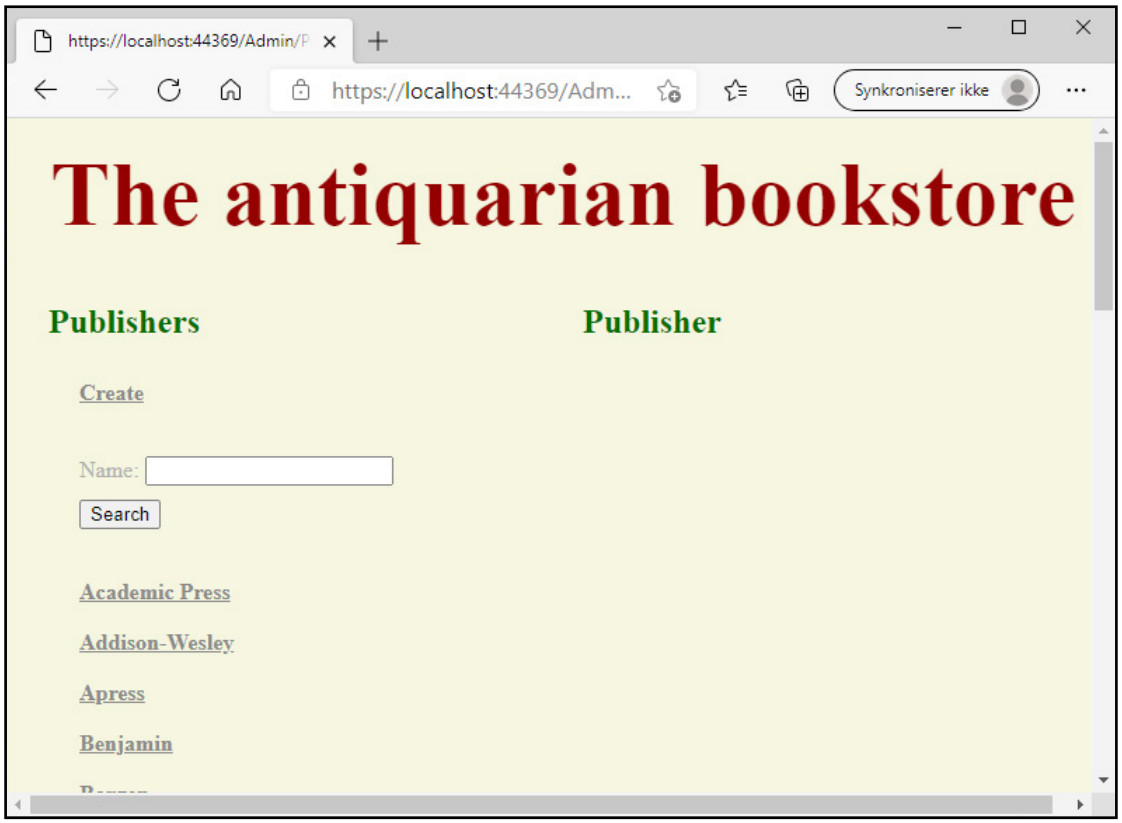


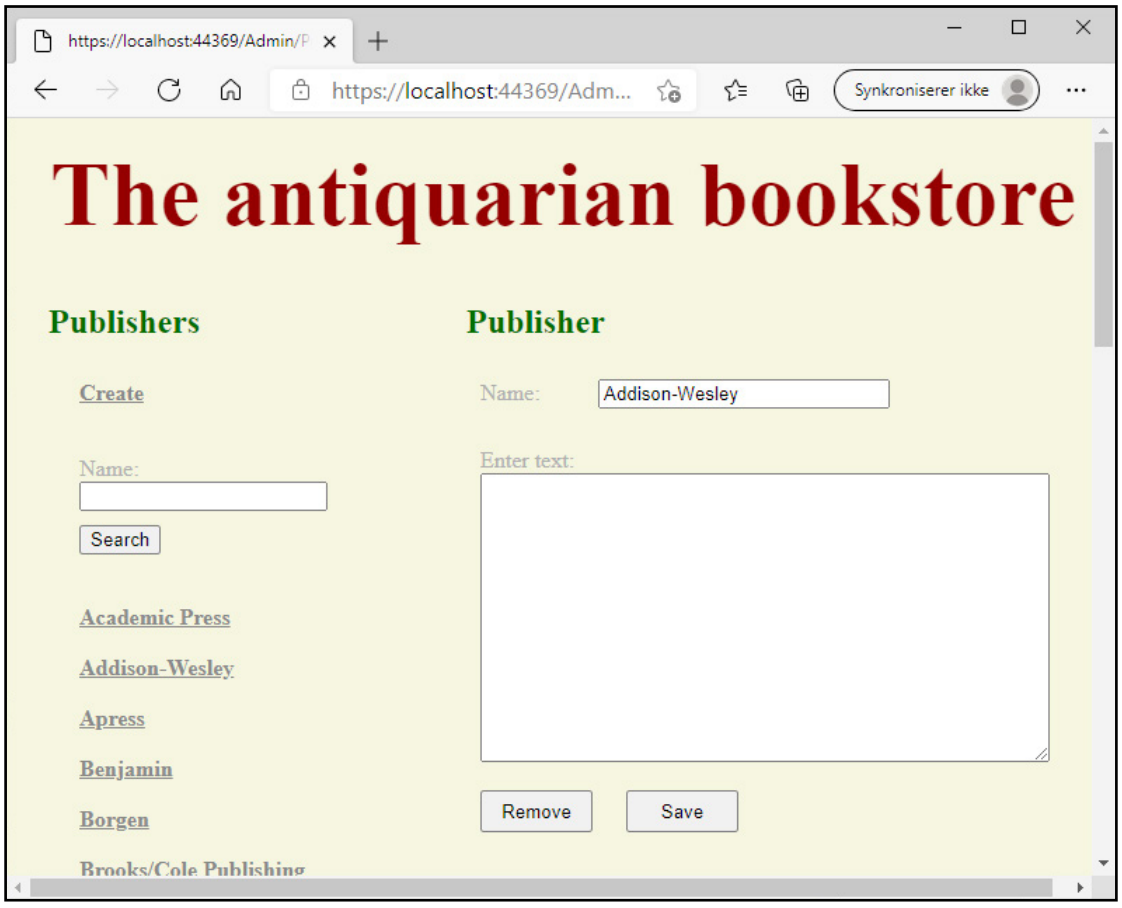
The implementation of the above two pages are simple and is not mentioned here, but the first requires an update of the class *Repository* with a method for login. All views for administration are added to a subfolder *Admin* and the folder *Controllers* has a *AdminController* which contains the action methods for administration.

The implementation of the first 6 functions all looks like each other while the last is trivial. As an example, I will review the implementation function *Publishers* which relates to the maintenance of publishers. If you select the function the result is as shown below. At the left is a list of all publishers in the database, and the users clicks on a link for at publisher the program opens a page which shows information for the publisher and where the user can edit the information. The first link (the upper link) in the list opens the same page and is used to create a new publisher. The left part of the window also has a field to enter a search text used to search a publisher after the name.

If you click on a publisher in the list the result could be as shown below and you can edit the publisher. You can just edit the name and enter a description. You can also remove the publisher, and you can thus in principle perform the usual database operations. As mentioned the same page is used to create a publisher and the only difference is that in this case the *Remove* button is not shown.

The implementation of the feature requires the following:





1. Update the repository with three methods *Insert()*, *Update()* and *Delete()*
2. Add a *ViewModel* class as a model for the view for this function
3. Create the view
4. Add action methods in the controller
5. Update the menu link in the view *Admin*

To update the *Repository* class requires some code. For publishers it is relatively simple, but in general it can mean that several database tables must be maintained and in particular it means that the collections that represents the database tables must be maintained. However, in all cases these are standard database operations, so I will not mention the code here.

The view model classes are all simple, and especially in this case. A view model class is used to bind data to a form in a view and for publishers the class is:

```
public class Publishers
{
  public Publisher Publisher { get; set; }
  public IQueryable<Publisher> Publishers { get; set; }
  public string Name { get; set; }
  public string Error { get; set; }
  public string Warning { get; set; }
}
```

The class has an object of the type *Publisher* and the form elements are bound to properties in the *Publisher* class. You should note that it means that field validations rules defined in this class are used. The property *Publishers* is a collection with the publishers to show in the left part of the view, and the property *name* is used for the search text. The last two properties are used for error messages, and here the last is used when the user delete a publisher to give a warning message.

The there is the view, and you should note that it is the same view which is used for both the above pages. The view is called *Publishers.cshtml*:

```
@model PublishersViewModel
@{
}
<table>
  <tr style="font-size:24pt">
    <th style="text-align:left;font-size:24px">Publishers</th>
    <th style="text-align:left;font-size:24px">Publisher</th>
  </tr>
  <tr>
    <td style="display:inline-block;vertical-align:top">
      <table>
        <tr>
          <td>
            <a class="link-text" asp-action="Publishers"
              asp-route-pubid="0">Create</a><br /><br />
          </td>
        </tr>
        <tr>
          <td>
            <form asp-action="Publishers" method="post">
              <div>
                <label asp-for="Name">Name:</label>
                <input asp-for="Name" />
              </div>
              <div>
                <button type="submit"
                  style="margin-top: 10px">Search</button><br/><br />
              </div>
            </form>
          </td>
        </tr>
        @foreach (Publisher publisher in Model.Publishers)
        {
          <tr>
            <td>
              <a class="link-text" asp-action="Publishers"
                asp-route-pubid="@publisher.Id" asp-route-name="@Model.
                Name">
                @publisher.Name</a>
            </td>
          </tr>
```

```
        }
        <tr>
          <td>
            <br /><a class="link-text" asp-action="Admin">Goto
            administration</a>
          </td>
        </tr>
      </table>
    </td>
    <td style="vertical-align:top">
      @if (Model.Publisher != null)
      {
      <form method="post">
        <input asp-for="Publisher.Id" type="hidden" />
        <input asp-for="Name" type="hidden" />
        <div class="input-line">
          <label asp-for="Publisher.Name"
            style="display:inline-block;width:70px;margin:10px">
            Name:</label>
          <input asp-for="Publisher.Name" style="width:200px" />
          <span asp-validation-for="Publisher.Name" class="text-
          danger"></span>
        </div>
        <div class="input-line" style="margin-top:20px;margin-
        left:10px">
          <label asp-for="Publisher.Text" class="label-text">
            Enter text:</label><br />
          <textarea asp-for="Publisher.Text"
            style="width:400px;height:200px"></textarea>
        </div>
        <p />

        @if (Model.Error != null && Model.Error.Length > 0)
        {
          <div style="margin:10px;font-weight:bold;color:red">@Model.
          Error</div>
          <p />
        }
        else if (Model.Warning != null && Model.Warning.Length > 0)
        {
          <div style="margin:10px;font-weight:bold;color:red">
```

```
            @Model.Warning</div>
          <p />
          <input type="submit" value="OK" formaction="PublisherAccept"
            style="width:80px;height:30px;margin-left:10px;margin-
            right:10px" />
        }
        else
        {
          @if (Model.Publisher.Id > 0)
          {
            <input type="submit" value="Remove"
            formaction="PublisherRemove"
            style="width:80px;height:30px;margin-left:10px;margin-
            right:10px" />
          }
          <input type="submit" value="Save" formaction="PublisherUpdate"
            style="width:80px;height:30px;margin-left:10px;margin-
            right:10px" />
        }
      </form>
      }
    </td>
  </tr>
</table>
```

It takes up a lot of space, but I still chose to show the whole code. Basically, it is an HTML table with two columns, where the first column shows the list of links for publishers, while the second shows the form for editing a publisher.

The first column is itself a table, but with only one column. The is a row for the *Create* link, a row to enter the search text and the a row for each publisher determined by the view model property *Publishers*. Finally, there is a row with a link for return to the *Admin* page. You must note that the row for enter the search text in fact is a form with an input field and a submit button. The input field is bound to the *Name* property in the model, and the submit button post the form to controller to select the publishers to show in the list.

The content of the second column is only shown if a publisher is selected or there is clicked on *Create* to create a new publisher. If it is the case the column contains a form with a input field and a *textarea* field and one or two buttons. There is not much new to add regarding the form, but you should note the submit buttons and how to determine which buttons to show and which action methods the buttons calls.

You should also note the many styles attributes and that they take up a lot of space and make the code difficult to read. These styles should instead be moved to the stylesheet and it tells that there is work left in connection with a code review.

The form has three different submit buttons and there is a submit button in the menu, and then there is a need for 5 action methods in the controller:

```
[HttpGet]
public IActionResult Publishers(string pubid, string name)
{
  if (HttpContext.Session.GetInt32("admin") != 1)
    return RedirectToAction("Index");
  if (name == null) name = "";
  PublishersViewModel model = new PublishersViewModel {
    Publishers = Repository.Instance.AllPublishers.Where(
      p => p.Name.Contains(name)) };
  if (pubid != null)
  {
    int id = int.Parse(pubid);
    if (id == 0) model.Publisher = new Publisher();
    else model.Publisher = Repository.Instance.Publishers.Find(p =>
    p.Id == id);
  }
  return View(model);
}

[HttpPost]
public IActionResult Publishers(PublishersViewModel model)
{
  if (HttpContext.Session.GetInt32("admin") != 1)
    return RedirectToAction("Index");
  string name = model.Name == null ? "" : model.Name;
  model.Publishers = Repository.Instance.AllPublishers.Where(
    p => p.Name.Contains(name));
  return View(model);
}

[HttpPost]
public IActionResult PublisherUpdate(PublishersViewModel model)
{
  if (HttpContext.Session.GetInt32("admin") != 1)
    return RedirectToAction("Index");
  if (ModelState.IsValid)
  {
    bool ok = model.Publisher.Id == 0 ?
      Repository.Instance.Insert(model.Publisher) :
        Repository.Instance.Update(model.Publisher);
    if (ok) return RedirectToAction("Publishers");
```

```
    model.Error = "Unable to update database...";
  }
  string name = model.Name == null ? "" : model.Name;
  model.Publishers = Repository.Instance.AllPublishers.Where(
    p => p.Name.Contains(name));
  return View("Publishers", model);
}

[HttpPost]
public IActionResult PublisherRemove(PublishersViewModel model)
{
  if (HttpContext.Session.GetInt32("admin") != 1)
    return RedirectToAction("Index");
  model.Warning = "Warning: Are you sure you want to remove this
  publisher?";
  string name = model.Name == null ? "" : model.Name;
  model.Publishers = Repository.Instance.AllPublishers.Where(
    p => p.Name.Contains(name));
  return View("Publishers", model);
}

[HttpPost]
public IActionResult PublisherAccept(PublishersViewModel model)
{
  if (HttpContext.Session.GetInt32("admin") != 1)
    return RedirectToAction("Index");
  bool ok = Repository.Instance.Delete(model.Publisher);
  if (ok) return RedirectToAction("Publishers");
  model.Error = "Unable to update database...";
  string name = model.Name == null ? "" : model.Name;
  model.Publishers = Repository.Instance.AllPublishers.Where(
    p => p.Name.Contains(name));
  return View("Publishers", model);
}
```

The first action method is for a HTTP GET and is performed when the user select the function from the *Admin* menu, or clicks on a publisher or clicks *Create*. You must note the first statement. It must only be possible to open this page if the administrator is logged in. When this happens a session variable is created and initialized to 1, and the action method (and all other action methods) test if this variable exists and has the value 1. If not the user is redirected to the login page. In this way a user can not navigate to the page using the address line in the browser unless you are part of an administrator session and the administrator is logged in. The a *PublishersViewModel* object is created initialized with

the publishers which matches the current search text. The parameter *pubid* is the used to determine where to instantiate a *Publisher* object which may either be for a new *Publisher* or an existing *Publisher*. Then the view is created for the current view model.

The other action methods are all for a HTTP POST and the next is used when the user clicks the submit button for a search text. The method is simple and only has to filter the publishers for current search text.

The third action method is used when the user clicks *Save*. The method validates the data (the result of the model binding) and if the data are legal methods in the repository are calked to insert a new publisher or update an existing publisher. If it is performed correct a redirection to the *Publishers* action is performed. Else the view model is post back to the form with a message telling about an error.

The action method *PublisherRemove()* is called when the user clicks on the *Remove* button. The method only initialize a warning message and post back the model. If the user then clicks *OK* the last action method is called, which is the method that deletes the publisher from the database.

The implementation of the other functions in the administrator menu happens in the same way. However, maintaining books and customers is more complicated. Books, because it is a more complex form, and customers, because you also have to be able to maintain customers' orders, something that you can discuss the sensible in.

6.7 THE LAST THINGS

As other programs a web application must be tested and you must do a review of the implementation to clean up. I will not mention this process here as it is not much different than for desktop applications, but a web application must be published to a web server before it can be used in practice. Before I look at that I will mentioned the connection string for the database.

Until now, connection strings have been hardcoded in the class *RepositoryBase*. In the same way as for other applications, it should be possible to configure where the database is hosted and its connection string should therefore be moved to a configuration file. For a web application, this means *appsettings.json*:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "EmailConfiguration": {
    "From": "...",
    "SmtpServer": "...",
    "Port": 465,
    "Username": "...",
    "Password": "..."
  },
  "AllowedHosts": "*",
  "ConnectionString": {
    "Constr": "Data Source=localhost\\SQLEXPRESS;Initial
    Catalog=Bookstore;
    Integrated Security = True" }
}
```

The program must read the connection string from the configuration file and for that I have added the following class to the *Tools* folder:

```
public class ConnectionString
{
  private static ConnectionString instance = null;

  public string Constr { get; set; }

  private ConnectionString()
  {
  }

  public static ConnectionString Instance
  {
    get
    {
      if (instance == null) instance = new ConnectionString();
      return instance;
    }
  }
}
```

which is a singleton to represent the connection string. I have also added a class:

```
public class DatabaseConfiguration
{
  public string Constr { get; set; }
}
```

and here you must note that the class has one property which has the same name as the key for the connection string in the configuration file. With this type you can read the connection string in *Startup* as:

```
ConnectionString.Instance.Constr =
  Configuration.GetSection("ConnectionString").
  Get<DatabaseConfiguration>().Constr;
```

The only thing left is to update the *Repository* class so that it uses the class *ConnectionString* to initialize the variable *constr*.

All examples in this book uses the build in web server in Visual Studio which is fine when developing an application, but for practical use the application must be hosted on a running web server somewhere. However, how depends on the current web server and it is necessary to examine the documentation for that server for how to do, and first one must of course have access to a running server. As an example, I will show how the current application can be installed on Microsoft's Information Server, which is usually briefly called IIS.

It is quite simple to install ISS on a Windows machine as the server is simply part of windows. Instead of showing how I would refer to articles on the internet and an example is

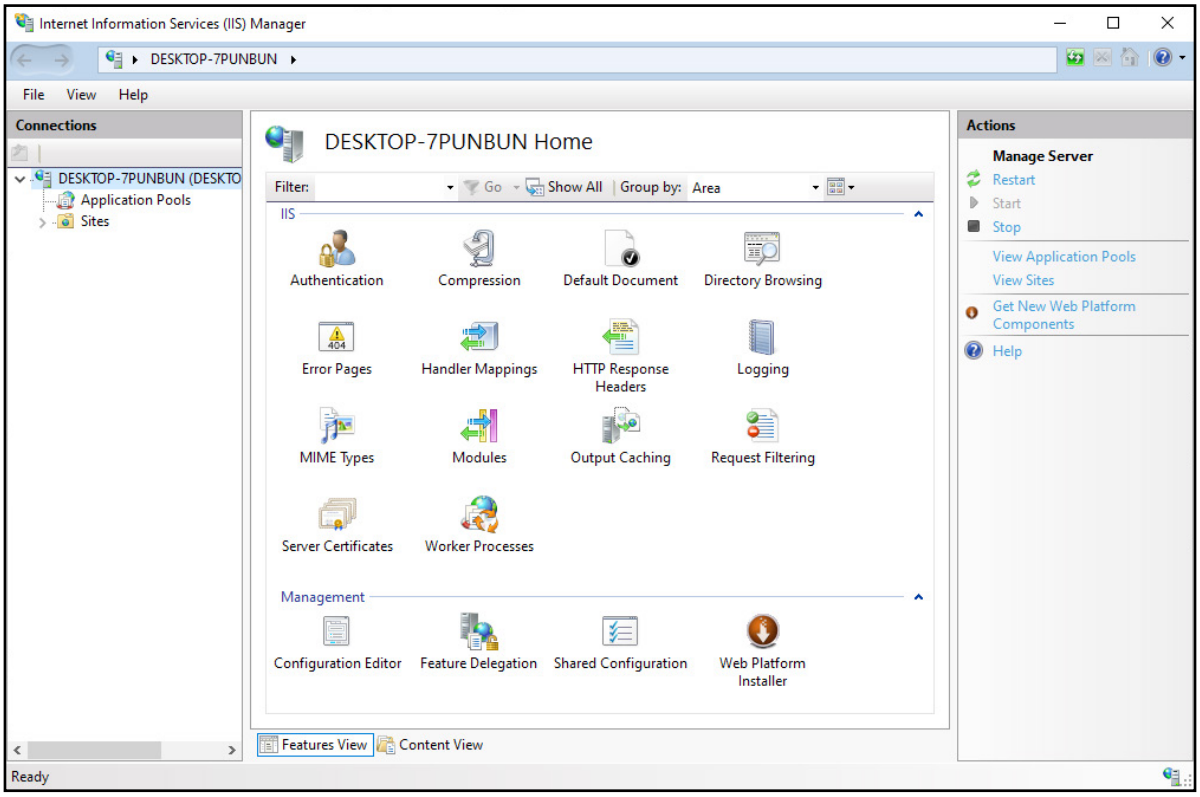
```
https://helpdeskgeek.com/windows-10/install-and-setup-a-website-in-
iis-on-windows-10/
```

When the server is installed, running and tested it must be updated for ASP.NET Core 5. You can do that from the following link:

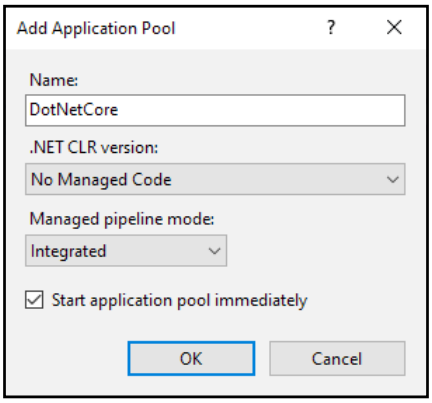
https://dotnet.microsoft.com/download/dotnet/5.0

After the server is running you must open

Internet Information Services (IIS) Manager

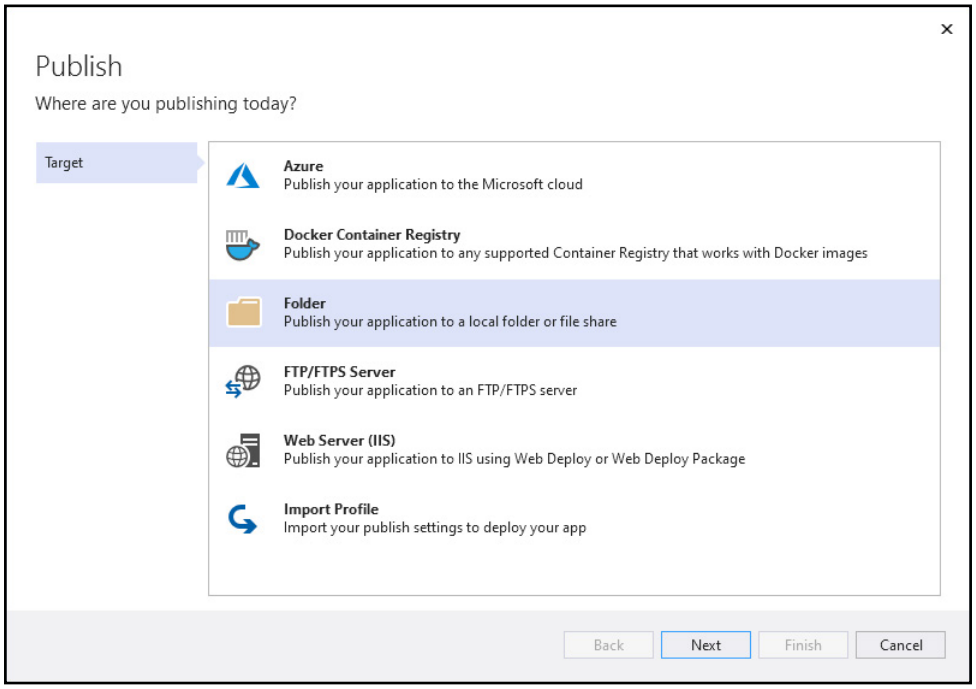


The you must right click on *Application Pools* and select *Add Application Pool*:

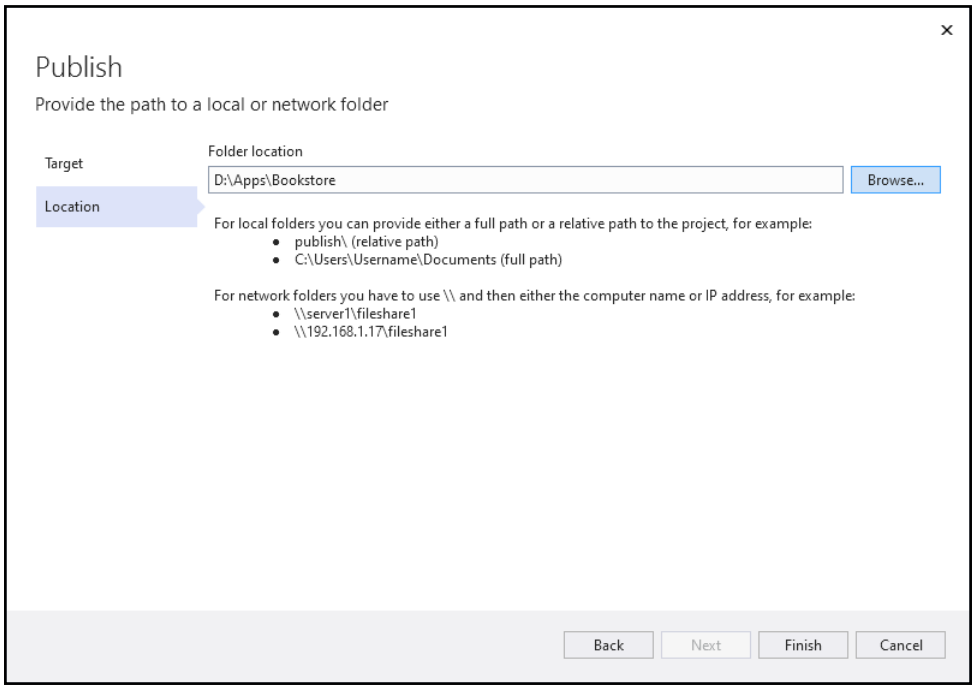


You must choose a name, but the important thing is the *.NET CLR version* which must be *No Managed Code*.

Then you should be ready to publish the application. In the following, I assume that IIS is installed on the local machine and thus the same machine where the program is developed. Start to build the application as release version. Then right click on the project name in Solution Explorer in Visual Studio and select *Publish*:

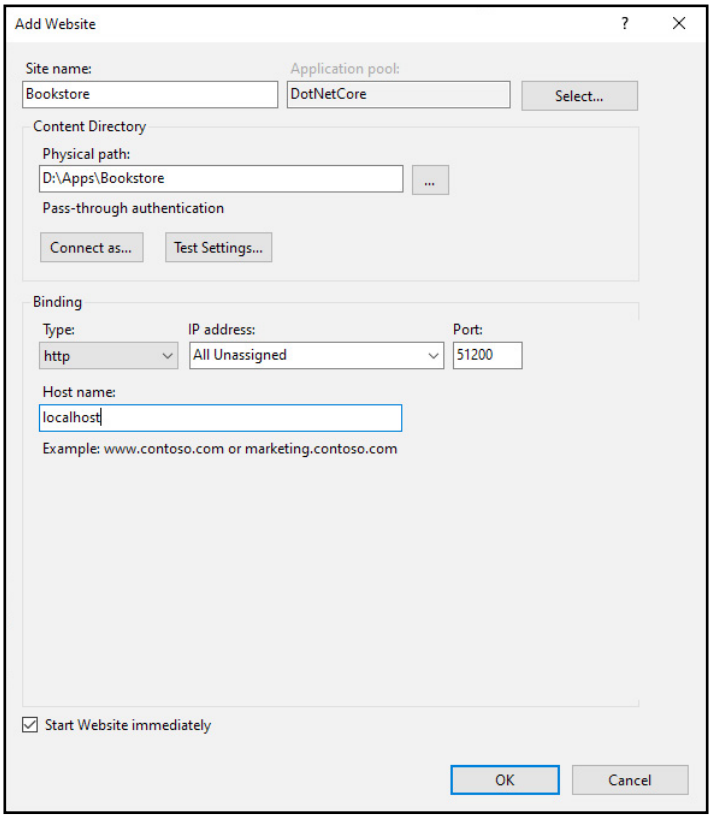


Here I select *Folder* and browse to a directory where I want to place the application:

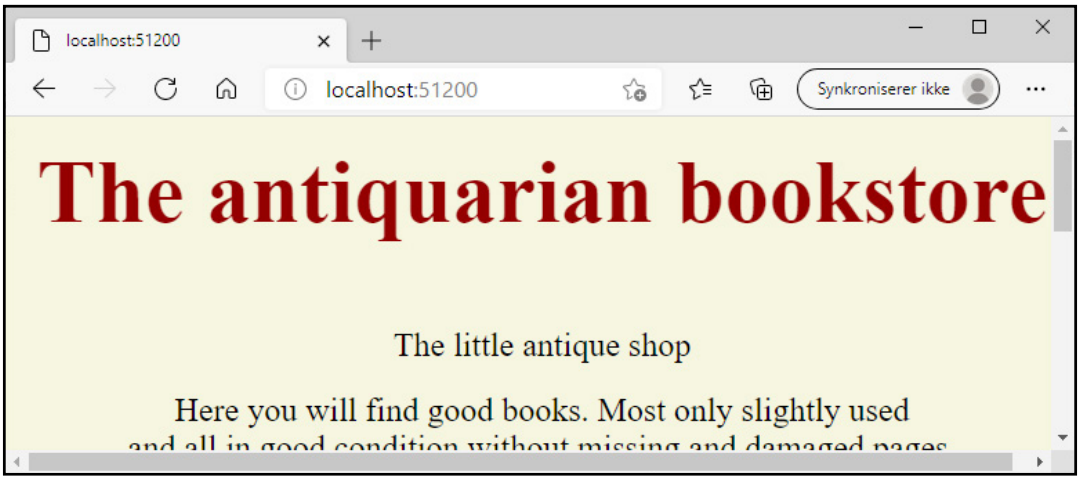


When I click *Finish* Visual Studio shows a summary window with a *Publish* button, and when I clicks the button Visual Studio copy all the necessary files to the selected directory. This is the file which must be hosted on the web server, and when my web server is running on the same machine it means that the server must know the directory with the files as a web application.

In *Internet Information Services (IIS) Manager* I right click on *Website* and select *Add Website*:



For *Site name* I have entered *Bookstore*. Then I have selected the application pool and browsed to the directory where the application is published. You must also note the port number which must be an unused port number, and when I clicks *OK* the application is hosted and ready for use. You can then run the application by entering the following address in your browser:



The result is a complete web application that is ready to use. Perhaps not the world's most advanced web application, but still an application that contains many of the features that characterizes a typical web application. The question then is what is missing and that is a part. Firstly, there is the user interface which far from lives up to what one expects from the web applications of today. That's the topic of the next book, which is probably not exactly a guide to developing user interfaces, but then at least explains some of the important technologies.

Another thing is security, which is something that plays a crucial role in web applications. Web applications are used from a network and the users come from all possible locations and for some users even with dubious intentions. Therefore, web applications must be secured against unwanted use, which to some extent happens with use of passwords and so on. The question then is to what extent the current application is secured against outside intervention, and there is a long way to go, someone that the book C# 13 deals with. The most serious shortcoming of the current application is that data is sent unencrypted and this also applies to users' passwords, for example. This means that in principle everyone can get hold of these passwords, and then there is not much security left.

The conclusion is that there is a lot left regarding web applications and I do not reach all the way, but the next two books will add a lot so you have a good starting point if you have to work as a developer of web applications.