Synthesis Lectures on Computer Science

Razvan Alexandru Mezei

Introduction to the Development of Web Applications Using ASP .Net (Core) MVC

Synthesis Lectures on Computer Science

The series publishes short books on general computer science topics that will appeal to advanced students, researchers, and practitioners in a variety of areas within computer science.

Razvan Alexandru Mezei

Introduction to the Development of Web Applications Using ASP .Net (Core) MVC

Razvan Alexandru Mezei
Department of Computer Science
Hal and Inge Marcus School of Engineering
Saint Martin's University
Lacey, WA, USA

ISSN 1932-1228 ISSN 1932-1686 (electronic)
Synthesis Lectures on Computer Science
ISBN 978-3-031-30625-9 ISBN 978-3-031-30626-6 (eBook)
https://doi.org/10.1007/978-3-031-30626-6

© The Editor(s) (if applicable) and The Author(s), under exclusive license to Springer Nature Switzerland AG 2024

This work is subject to copyright. All rights are solely and exclusively licensed by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.
The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
The publisher, the authors, and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, expressed or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

I would like to dedicate this work to my fiancée Adriana Cheteles for her great support and encouragement and to my Ph.D. advisor, Prof. George Anastassiou, whose mentorship and guidance opened my professional path.

Razvan Alexandru Mezei

Preface

This work is intended to be used as a (quick) one-semester introduction to Web Applications development using ASP .Net Core MVC. In particular, it briefly introduces some client-side languages and frameworks (HTML, CSS, JavaScript, and Bootstrap), then it focuses primarily on the server-side portion (C#, Entity Framework Core) using the Model-View-Controller (MVC) pattern. Along the way, you will be introduced to many concepts such as routing, services and dependency injection, object relational mapper, model validation, and authentication. By the end of this book, you will create a web application that stores its data in a database and includes some basic account management functions (register user accounts, login, and logout).

Lacey, WA, USA Razvan Alexandru Mezei

Contents

1 Introduction ... 1

2 Prepare the Development Environment ... 5
- 2.1 Choose a Web Browser ... 5
- 2.2 Install Visual Studio Code ... 6
- 2.3 Install Visual Studio ... 6
- 2.4 Install DB Browser for SQLite ... 7
- 2.5 Miscellaneous/Optional ... 8
  - 2.5.1 Show File Name Extensions ... 8
- 2.6 Microsoft SQL Server ... 8
  - 2.6.1 Sample Data Generators ... 8

3 Brief Introduction to Html ... 9
- 3.1 Let's Create Our First HTML Page ... 9
- 3.2 Add Titles, Paragraphs and Headings ... 11
- 3.3 Add a Second Webpage ... 13
- 3.4 Add Links and White Spaces to Our Pages ... 13
- 3.5 Add Images and White Spaces to Our Pages ... 15
- 3.6 Tables and Buttons ... 17
- 3.7 A Few Other HTML Elements We'll Use Later ... 19
  - 3.7.1 Label and Select Elements ... 19
  - 3.7.2 Input Elements ... 20
- 3.8 Form and More on Input Elements and Attributes ... 21
- 3.9 GET Versus POST Request, the Action and the Method Attributes ... 25

4 Brief Introduction to CSS, Javascript, and Bootstrap ... 29
- 4.1 Motivation for Using CSS and JavaScript ... 29
- 4.2 Our First CSS Example ... 30
- 4.3 Introduction to CSS Syntax ... 31
- 4.4 CSS Selectors ... 33
- 4.5 Conflicting CSS Specifications ... 37

4.6 Other CSS Selectors 38
4.7 A Few More Examples of Property-Value Pairs for CSS 38
4.7.1 Text Color in CSS 39
4.7.2 Text Alignment in CSS 39
4.7.3 Fonts in CSS 40
4.8 The Box Model and the Developer Tools 43
4.9 The DIV Element 45
4.10 Ways to Add CSS 45
4.10.1 Internal 45
4.10.2 In-line 45
4.10.3 External 47
4.11 First Encounter with Bootstrap 48
4.11.1 Add Bootstrap 5 .css to Our Webpages 48
4.11.2 Bootstrap 5 Tables 49
4.11.3 Bootstrap 5 Buttons and Links 50
4.11.4 Bootstrap 5 Container, Padding 51
4.11.5 Bootstrap 5 Source Code 51
4.11.6 Center Contents with < DIV> and CSS 52
4.12 Introduction to JavaScript 54
4.13 JavaScript Statements 56
4.14 JavaScript Functions 56
4.15 Add JavaScript to Our Webpages 57
4.16 Introduction to the Document Object Model (DOM) 58
4.17 Add Event Handlers 60
4.18 An Example: Toggle Between Dark/Light Mode 61
4.19 The Back Button 62
4.20 External JavaScript 63
4.21 More Introduction to Bootstrap 64
4.22 Ways to Include Bootstrap in Our Projects 64
4.23 Some CDNs for Bootstrap 5 64
4.24 View Bootstrap 5 Source Files 65
4.25 Bootstrap 5 navbar 65

5 Some C# Fundamentals 69
5.1 Hello World in C# (Console Application) 69
5.2 Top-Level Statements 70
5.3 Namespaces, Using Directive, and Global Using Directive 71
5.3.1 Namespaces 71
5.3.2 Using Directives 71
5.3.3 Implicit Using Directives 73
5.3.4 Global Using Directives 73
5.4 Comments 74

5.5 Existing Data Types 74
5.6 String Interpolation 76
5.7 Enumerations 77
5.8 Classes 78
5.9 References and Objects 79
5.10 Instance Variables/Non-static Fields 80
5.11 Dot Notation 81
5.12 Methods 82
5.13 The this Keyword 84
5.14 Access Modifiers 84
5.15 Properties 85
5.16 Constructors 87
5.17 Method Overloading 88
5.18 Conditionals, Loops, and Lists 88
5.19 Collections and Generic Collections 90
5.20 Inheritance 90
5.21 The base Keyword and the Constructors 92
5.22 Interfaces 95
5.22.1 Some Motivation 96
5.22.2 How to Define an Interface 96
5.22.3 How to Implement an Interface 96
5.23 How to Use an Interface 97
5.24 Lambda Expressions 98
5.25 LINQ 100
5.26 Working with null Values 102
5.27 Solution Files .sln 102
5.28 Other Resources for Learning C# 103

6 Middleware, Services, Intro to Dependency Injection 105
6.1 What Are ASP .Net (Core) MVC Web Applications? 105
6.2 An Introduction to the MVC Pattern 107
6.3 A Quick Dive into an MVC Example (Optional) 108
6.4 Let's Start Our ASP .Net Core Application Project in Here 113
6.4.1 The Empty Web Application Starting Point 113
6.4.2 The MVC Web Application Starting Point 114
6.5 Entry Point to Our Web Application: Program.cs 114
6.6 The Middleware Pipeline 115
6.6.1 The Current Code in Our Project 115
6.6.2 Run, Use, and Map 117
6.6.3 First Example 117
6.6.4 Second Example 119
6.6.5 Third Example 119

6.6.6 Other Middleware Components . . . 121
6.7 Static Files Middleware . . . 122
6.7.1 What Are *Static Files*? . . . 122
6.7.2 Where Do We Store *Static Files*? . . . 122
6.7.3 How Do We Allow Access to *Static Files*? . . . 122
6.7.4 How Can We Access *Static Files*? . . . 123
6.7.5 Default (Static) Page . . . 124
6.8 Introduction to Services (Optional) . . . 128
6.8.1 Example—Step 1: Define a Class and An Interface . . . 128
6.8.2 Example—Step 2: Register a Service . . . 129
6.8.3 Example—Step 3: Use a Service . . . 129

7 Routing, Models, and Controllers . . . 131
7.1 A Little Cleanup Before We Continue . . . 131
7.2 Some Essential MVC Concepts and the HTTP Request Lifecycle . . . 132
7.3 Introduction to Routing . . . 134
7.3.1 Adding MVC to Our ASP .Net Core Application . . . 135
7.3.2 Default Routing, the Home Controller, and Actions . . . 135
7.4 Add a Model, a Controller, and Views . . . 139
7.4.1 Add a Model Class . . . 139
7.4.2 Add a (Second) Controller Class . . . 140
7.4.3 Add a First View . . . 141
7.4.4 Test Our Code so Far . . . 142
7.5 Various Action Result Types . . . 143
7.6 Conventional Versus Attribute Routing . . . 145
7.6.1 Conventional Routing . . . 146
7.6.2 Attribute Routing . . . 148
7.6.3 Mixing Routings . . . 150

8 More on Controllers and Views, Introduction to Razor Syntax . . . 153
8.1 A Little Cleanup Before We Continue . . . 153
8.2 Some Essential MVC Concepts and the HTTP Request Lifecycle (Revisited) . . . 154
8.3 Another Example of Model, Controller, and Views . . . 156
8.3.1 The Instructor Model . . . 156
8.3.2 The `InstructorController` Class . . . 157
8.4 The Index Action and View . . . 161
8.4.1 Add a View for Our `Index` Action . . . 161
8.4.2 Strongly Typed and Weakly Typed Views . . . 163
8.4.3 Introduction to Razor Engine and Razor Syntax . . . 164
8.4.4 Action Using a View with a Different Name . . . 167
8.5 The ShowDetails Action and View . . . 169
8.5.1 The ShowDetails Action . . . 169

8.5.2 The ShowDetails View ... 170
8.6 A First Look at Tag Helpers and HTML Helpers ... 173
8.6.1 A First Example of an HTML Helper ... 173
8.6.2 A First Example of a Tag Helper ... 174
8.6.3 Add Links to the Index View Using Tag Helpers and HTML Helpers ... 175
8.6.4 Add Bootstrap to the Index View ... 175
8.6.5 Add Links to the *ShowDetails* View ... 177

9 More on Views, Data Annotations ... 179
9.1 Introduction to Data Annotations ... 179
9.1.1 Update the ShowDetails View ... 179
9.1.2 Update the Index View (Optional) ... 184
9.2 The Add Action and View ... 186
9.2.1 The Add Action—GET ... 186
9.2.2 The Add View ... 187
9.2.3 The Add Action—POST ... 192
9.2.4 A Few More Details About the Model Binding ... 195
9.2.5 A Few More Details About the GET Versus POST ... 196
9.3 The Edit Action and View ... 198
9.3.1 The Edit Action—GET ... 198
9.3.2 Add Edit Links in the Index View ... 198
9.3.3 The Edit View ... 199
9.3.4 The Edit Action—POST ... 201
9.3.5 An Example of a Service ... 203
9.4 The Delete Action and View ... 207
9.4.1 The Delete Action—GET ... 207
9.4.2 Add Delete Links in the Index View ... 208
9.4.3 The Delete View ... 208
9.4.4 The DeleteConfirmed Action—POST ... 209

10 Model Validation ... 213
10.1 Step 1: Add (Built-in or Custom) Validation Attributes ... 216
10.2 Step 2: Enforce Validation by Making Use of the ModelState ... 217
10.3 Step 3: Display Error Messages via Validation Tag Helpers ... 218
10.3.1 To Display a Summary of All Error Messages ... 219
10.3.2 To Display In-line Error Messages ... 219
10.4 Let's Test Our Model Validation ... 221
10.5 Custom Validation Attributes (Optional) ... 222
10.5.1 Create a Custom Validation Attribute ... 223
10.5.2 Use a Custom Validation Attribute ... 226
10.5.3 Let's Test the Newly Added Custom Validation ... 226

10.6 Validation Text Styling 227

11 Persistent Data: Entity Framework Core 231
11.1 Introduction 231
11.2 Classes Involved: Providers, DbContext, and DbSet 232
11.3 Add Entity Framework Core to Our Web Application 232
11.3.1 Step 1: Create/Choose Your Entity Classes 233
11.3.2 Step 2: Install NuGet Packages 233
11.3.3 Step 3: Create a Class Derived from DbContext 234
11.3.4 Step 4: Data Seeding 235
11.3.5 Step 5: Register Our DbContext as a Service, and Use a Connection String 236
11.3.6 Test Our Database 239
11.4 Use Entity Framework Core in Our Web Application, Dependency Injection Revisited 240
11.4.1 Inject Entity Framework Core in InstructorController 240
11.4.2 Update the Actions to Use Entity Framework Core 240
11.4.3 Important: Automated Id Generation 242
11.4.4 Let's Test That We Have Persistent Data 243
11.4.5 EnsureDeleted 244
11.5 Practice: Update the StudentController Class 245
11.5.1 Inject Entity Framework in StudentController 245
11.5.2 Use Entity Framework Core in StudentController Actions 245
11.6 How to Use Microsoft SQL Server Instead of SQLite (Optional) 247
11.6.1 Install SQL Server Express LocalDB Database on Your Machines 247
11.6.2 Make Changes so Entity Framework Core Now Works with a Microsoft SQL Server Database 248

12 Consistent Look: Layouts, Friendly Error Pages, and Environments 251
12.1 Filter Results 251
12.1.1 Update the Index View 251
12.1.2 Update the Index Action 253
12.1.3 Implement the *Clear the Filter* Button 254
12.2 Filter Results Using a Dropdown List (Optional) 255
12.2.1 Create the Dropdown List Items in the Index Action 256
12.2.2 Display the Dropdown List Items in the Index View 256
12.2.3 Use of the Dropdown List to Filter Our Results 256
12.2.4 The Code 258
12.3 Consistent Webpages—Using Razor Layouts 258
12.3.1 Create a Layout 259
12.3.2 Use the Layout in Our Views 259

12.3.3 Add a Bootstrap 5 Navbar to Our Layout 265
12.3.4 Add Navigation Links to Various Actions and Controllers . . . 265
12.4 Layout Sections (Optional) 267
12.4.1 Define a Section 267
12.4.2 Make Use of a Section 269
12.5 Make Use of Bootstrap 5 Buttons 270
12.5.1 The Index View 270
12.5.2 The ShowDetails View 273
12.5.3 Use Bootstrap for Styling Validation Errors 273
12.6 Configure a Friendly Error Page 277
12.6.1 Introduction 277
12.6.2 Work with Multiple Environments 278
12.6.3 The Developer Exception Page 279
12.6.4 The Friendly Error Page 280

13 Work with Images (Optional) 285
13.1 Add a New Property for the Image to the Model/Entity Class 285
13.2 Modify the Add View, so It Allows a User to Upload an Image 286
13.3 Modify the Add Action so the File Uploaded Gets Saved into the Database 287
13.4 Modify the ShowDetails Action to Transform the Byte Array Back into an Image 289
13.5 Modify the ShowDetails View so It Displays the Profile Image 289
13.6 Bootstrap Card Deck for the Index Action and View (Optional) 291

14 Introduction to Authentication. User Login, Logout, and Registration 293
14.1 Introduction to Some Security Concepts 293
14.2 Introduction to ASP .Net (Core) Identity 294
14.2.1 Step 1: Install NuGet Packages 295
14.2.2 Step 2: Define Our User Class (Derived from IdentityUser) 295
14.2.3 Step 3: Update Our DbContext Derived Class to Use Identity 296
14.2.4 Step 4: Register the Identity Services 296
14.2.5 Step 5: Add Authentification and Authorization Middleware Components 297
14.2.6 Test Your Work 298
14.2.7 Step 6: Register, Login, and Logout 298
14.2.8 Step 7: Add Simple Authorization to Our Web Application (Optional) 306

References 311

1 Introduction

This work is intended to be used as an introduction to Web applications development using ASP.Net (Core) MVC, primarily for undergraduate students. It assumes that our readers already have some experience with one programming language (ideally C#, but other similar languages, such as Java or C++, may suffice too). Readers will be introduced to various client-side languages and frameworks (such as HTML, CSS, JavaScript, and Bootstrap), learn about some server-side ones (C#, ASP.Net Core), and make use of an object relational mapper (the Entity Framework Core). The focus of this book is on the server-side development, in particular the MVC pattern.

There are several reasons why we believe students can greatly benefit from this content. First, it allows them to develop potentially medium to large-sized projects (web applications) that use multiple programming languages (both client and server side) all in one project, which can be particularly useful before working on a capstone project. It also provides a great playing ground for applying many of the object-oriented programming concepts (including classes, inheritance, interfaces, dynamic and static polymorphism, and many others), and also exposes students to important concepts (such as responsive design, authentication, object relational mapper, cookies, routing, session information, HTTP requests, CRUD operations, asynchronous programming, and cross-platform development) which should students gain confidence when preparing for software development-related job interviews.

This book starts (in Chap. 2) with preparing the development environment and it goes over the installation of the applications needed for the remaining chapters in this book. Then, in the following two chapters (Chaps. 3 and 4), we provide an introduction to client-side development and briefly go over HTML5, CSS, JavaScript, and Bootstrap 5. In the following chapter (Chap. 5), we attempt to provide a brief introduction to various C# language components. Then, we use the remaining chapters (Chaps. 6–14) to cover several

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024

R. A. Mezei, *Introduction to the Development of Web Applications Using ASP .Net (Core) MVC*, Synthesis Lectures on Computer Science,
https://doi.org/10.1007/978-3-031-30626-6_1

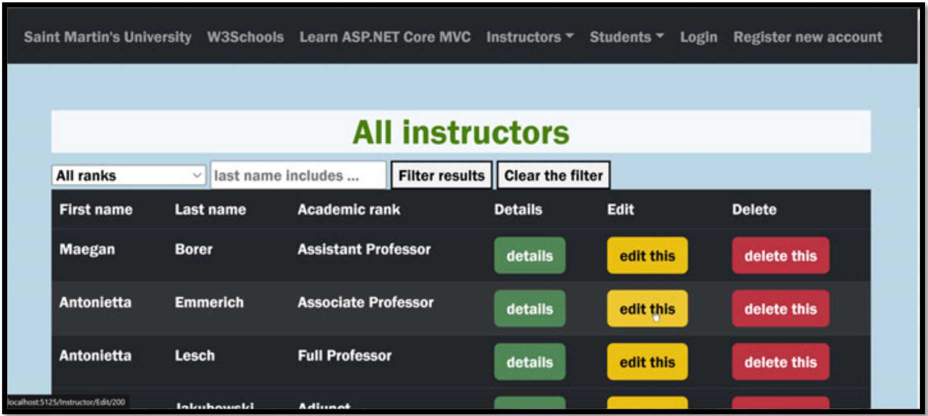


Fig. 1.1 A screenshot of how the final application will look like in a browser

topics related to web applications development using ASP.Net Core MVC such as routing, middleware pipeline, services and dependency injection, models, views, view models, controllers and actions, Razor syntax, model binding, HTML and tag helpers, model validation, layouts, entity framework core, connection strings, identity, authentication, and simple authorization.

In Chap. 6, we build a web application and add to it new functionality for the remaining chapters of the book. By the end of the book, our web application will store its data in a database, will allow us to create user accounts, and include login and logout functionality. We'll mostly use an SQLite database but will also demonstrate how to use a Microsoft SQL Server database.

Figure 1.1 shows how the final application will look like.

The two main sources used for most topics covered in this book and extensively used during class presentations are as follows:

- *W3Schools Online Web Tutorials* ([1])—a great resource for learning about client-side development (and much more!), in particular, used introducing HTML, CSS, JavaScript, and Bootstrap 5.
- *Get started with ASP.NET Core MVC* ([2])—a great resource for learning about ASP.Net Core MVC.

In our course, as part of the course assignments, we asked our students to create their own personal projects. Each week, as we covered new concepts and demonstrated in class how they can be applied to our class project, students were asked to apply the same concepts to their own personal project. It has been documented that individualized assignments can help keep students motivated and engaged throughout the course, give students a

chance to express their own interests, as well as “deter cheating or blindly copying from other students” ([3]). Overall, we were quite impressed with the range of ideas for web applications that our students came up with when they created their own projects. Based on the course evaluations, most students not only enjoyed the class materials, but they also felt more confident and more prepared to apply for software development positions. Some also felt they gained a better understanding of coding and debugging skills in general.

We conclude this chapter with the following statement. This work is not meant to be a complete/comprehensive resource on C# nor ASP.Net (for that please see the list of references), but a one-semester simplified introduction to web development using the ASP.Net (Core) framework and the MVC architectural pattern. There are many great topics and features that we could not include in a one-semester course, so a comprehensive approach is beyond the scope of this book.

2 Prepare the Development Environment

Before we start developing our web applications, we first need to set up our development environment. Here is a list of tools we'll be using throughout this book:

- A(ny) web browser.
- Visual Studio and Visual Studio Code (with related packages).
- DB Browser for SQLite (alternatively, one can use Microsoft SQL Server Management Studio for Microsoft SQL Server).

Let's go over each one of these tools and provide some guidance on how to install them on your computers, and what packages to add.

2.1 Choose a Web Browser

One important reason why *web applications* are so popular is the fact that most desktop computers, laptop computers, tables, and mobile phones that are connected to the Internet will also have a browser installed on them. If you already have a web browser (you probably do!) then use that one. Otherwise, you can install one from the Internet.

In this book, we will use the Google Chrome browser for no particular reasons other than the fact that is it a very popular one, and already have this web browser installed. To download Google Chrome, one can use the following source https://www.google.com/chrome/. Other popular browsers are Mozilla Firefox, Microsoft Edge, and Opera. Choose your favorite one.

The languages "spoken" by virtually all modern web browsers are HTML, CSS, and JavaScript. For this reason, web applications make use of these "client-side" languages.

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024

R. A. Mezei, *Introduction to the Development of Web Applications Using ASP .Net (Core) MVC*, Synthesis Lectures on Computer Science,
https://doi.org/10.1007/978-3-031-30626-6_2

If your clients already have a modern web browser installed, they don't need to install anything else in order to run JavaScript, CSS, and HTML.

Why do we care? While we can have control over our servers and what applications they have installed on them, making assumptions about the clients' systems is not an easy task. Any wrong assumption and we can severely limit our clients' pool. But if our client side of the application uses HTML, CSS, and/or JavaScript, the chances that our code can run on our clients' machines are very high (all they need is a modern web browser). This is not the case if we want our application to use, say Java, or C#. What would our clients need in order to be able to run Java applications? What about C# applications?

2.2 Install Visual Studio Code

To write JavaScript, HTML, and CSS code, we can use any text editor program. One can use any popular text editor such as Vim, Notepad, Notepad++, and Atom. In this book, we will use Visual Studio Code (VS Code). VS Code is a very popular editor that works on Windows, Mac OS, and Linux machines. To download it and then install it, one can go to the following page on the Microsoft's webpage https://visualstudio.microsoft.com/.

Once downloaded, run the installation file (you will need to agree with the license agreement terms) and feel free to use the default settings (press the *Next*>button twice then press the *Install* button).

We will only use Visual Studio Code for the next two chapters. Then, we will use Visual Studio for the remaining of this book. Depending on the operating system on your machine you may use it for the entire book (see below for more details).

2.3 Install Visual Studio

Once we start programming in C# (see Chap. 5) and for the remaining part of the book we will be using Visual Studio. To download it and then install it, one can go to the following page on the Microsoft's webpage https://visualstudio.microsoft.com/.

In here you will find that there are two versions of Visual Studio, one that works on Windows machines, and one that works on Mac OS systems.

Important note: Unfortunately, at the time of writing this book, Visual Studio is not available for Linux machines. If you're using a Linux distribution, you should use Visual Studio Code instead. Note that the Visual Studio Code does not come to include the C# compiler nor the .Net SDK, so you'll need to install them separately. We recommend our readers to check [1] or [2] for guidance on how to use Visual Studio Code for developing ASP.Net Core MVC applications.

Install the version that corresponds to your machine's operating system (note: if you already have Visual Studio 2022 or above installed on your machine, then you can skip

this step and move to the next one). For the slides shown in this book, I will use the one for Windows. The Community edition is free and we will use this version (Visual Studio Community 2022) in our book.

Important note: In order to use .Net 6 (or above), you must get Visual Studio 2022 (or above). For example, if you already have Visual Studio 2019 installed on your machine, you won't be able to use that for .Net 6.0: you'll need to install Visual Studio 2022. You can have multiple versions of Visual Studio installed on your machine—if you already have Visual Studio 2019, you do not have to uninstall it, you can keep it and install Visual Studio 2022 along with Visual Studio 2019.

Once you run the installation application, the *Visual Studio Installer* will open up (note: if you already have Visual Studio installed on your machine, then open Visual Studio Installer and continue with this step). Make sure to select the *ASP.NET and web development* workload.

On the *Individual components* tab, make sure the *.Net 6.0 Runtime* is selected (also select *LINQ to SQL tools* which are down the page, under *Code Tools*).

Then click on the *Install* button. When the installation finishes, open the Visual Studio application and click on the *Create a new project* button. Then, in the search box type MVC and make sure to have the option of creating an *ASP.NET Core Web App (Model-View-Controller).*

If you do, then you're all set. Otherwise, make sure to follow the Visual Studio Installer step described above. If you want, we recommend you to continue and follow the steps shown on the webpage: https://learn.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/start-mvc to create your first ASP.Net Core MVC application and make sure it runs on your machine.

2.4 Install DB Browser for SQLite

One last installation (we'll only use this starting with Chap. 11) is a free open-source application that allows us to interact with SQLite database files. We will use it to check our database values. To download this application, go to https://sqlitebrowser.org/dl/ and download the version that corresponds to your system. For my system, that is *DB Browser for SQLite—Standard installer for 64-bit Windows.* Once downloaded, run the installation file (you will need to agree with the license agreement terms) and feel free to use the default settings (press the *Next* button twice then press the *Install* button).

Once the installation finishes, open the application to make sure you can run it on your machine.

2.5 Miscellaneous/Optional

2.5.1 Show File Name Extensions

We recommend that you set your operating system to *show file extensions*. On my Windows 11 machine, I would open any folder and go to Views > Show > File name extensions (make sure it's checked).

2.6 Microsoft SQL Server

In this book, we will include an optional section that shows you how to use Microsoft SQL Server instead of SQLite. If you want to install the Microsoft SQL Server Management Studio. Here is a link for it: https://aka.ms/ssmsfullsetup.

In Chap. 11, we'll go over details on how to install the SQL Server Express 2019 LocalDB database, using Visual Studio Installer.

2.6.1 Sample Data Generators

We will occasionally need to create some names as sample data. In class, we typically like to ask our students to give us some ideas. In this book, we used a name generator instead, namely we used https://commentpicker.com/fake-name-generator.php to help us create some random names.

3 Brief Introduction to Html

The purpose of this chapter is to introduce some fundamental HTML concepts. We'll use this knowledge quite a bit in subsequent chapters, especially when we create *Razor views* and *layouts*. One great resource that we often used in the class as a source of information as well as for quick demonstrations is the following: [1].

HTML stands for *Hyper Text Markup Language* and it is used to describe the structure of a Webpage. Using various HTML *elements,* we can tell a web browser how to display different parts of a given content. HTML files typically have the extension .html or .htm.

The way the author likes to think of the HTML is as follows: we have content that we would like to display in a browser. Using HTML (tags), we can let a browser know how to display various pieces of this content. Some parts will show up as a paragraph, others as tables, others as headings, and so on. So, in general, we use HTML to describe the structure of the page. Note that for styling (that describes whether the contents should show up left aligned or centered, bold or italic, red or green, etc.) we will use CSS (which is the subject of the next chapter).

3.1 Let's Create Our First HTML Page

Open the *Visual Studio Code* application. First, go to *File* > *Open Folder* and create a new folder for our HTML files.

We named our folder *HTML FILES*. Then, inside that folder, let's create our first HTML file. We named our file *firstwebpage.html.*

In this file, let's add the following contents (do not include the line numbers on the left side, we included them for easier reference):

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024

R. A. Mezei, *Introduction to the Development of Web Applications Using ASP .Net (Core) MVC*, Synthesis Lectures on Computer Science,
https://doi.org/10.1007/978-3-031-30626-6_3

Display this as a heading

Display this as a paragraph.

Display this as another paragraph.

Fig. 3.1 A screenshot of how the H1 and P elements will show in a browser

```
1.<!DOCTYPE html>
2.<HTML>
3.    <HEAD>
4.        <TITLE>Title: our first webpage</TITLE>
5.    </HEAD>
6.    <BODY>
7.        <H1>Display this as a heading</H1>
8.        <P>Display this as a paragraph.</P>
9.        <P>Display this as another paragraph.</P>
10.    </BODY>
11.</HTML>
```

Save the changes and open the *firstwebpage.html* file in a browser. Here (Fig. 3.1) is what we got in our browser:

Now let’s go over each line of the HTML code above.

The first line `<!DOCTYPE html>` is a declaration that specifies that the file is an HTML5 document (HTML5 is the latest version of HTML). We’ll use this in all our HTML files.

In line 2, `<HTML>` represents a **tag** (in this case, the HTML tag). This tag has a corresponding **end tag** in line 11 `</HTML>`. The tags and all content between them together represent an **element**. In this case, lines 2–11 represent the **root element** of the page. This element typically contains two nested elements: the *HEAD* element and the *BODY* element.

The `<HEAD>` element (represented by the start tag: `<HEAD>` from line 3, the end tag: `</HEAD>` from line 5, and the contents between these two tags) typically contains meta information about a page (we’ll see more about this soon). One such meta information is represented by the `<TITLE>` element, which can be seen displayed in a browser, in the page’s tab (top part of the page).

The `<BODY>` element (represented by the start tag: `<BODY>` from line 6, the end tag: `</BODY>` from line 10, and the contents between these two tags) defines the *body of the page*, which contains **all the visible parts of the page**, such as headings, paragraphs,

images, tables, and links. An example of such content is represented by the `<P>` element, which defines a paragraph.

We close this section with a few remarks:

- HTML is NOT case sensitive. That means that we could have used either of the following to represent the HTML tag: `<HTML>`, `<html>`, or even `<HtMl>`. This applies to all tags.
- HTML elements can be nested (they often are). As seen above, we have the `<P>` element nested inside (it is a part of) the `<BODY>` element.
- There are HTML elements (for example, `<BR>`) that have no content. Such elements are called **empty elements** and they have *no end tag*.
- If no title is provided for a webpage, a browser will typically use the filename as its title. This is not always very elegant. So please make sure to always provide a `<TITLE>` element for your webpages. Also, the contents of the title are used by search engines [2], so it is important to provide a meaningful title.
- "HTML largely ignores whitespace" (see more details in [3]).

3.2 Add Titles, Paragraphs and Headings

First, let's change the title of the webpage so it displays: `Info for <Put Your Name>`.

Then, remove the contents of the `<BODY>` element and replace them with the following:

- Add an `<H1>` element with the content: `Student Information`.
- Add an `<H2>` element with the content: `<Put Your Name>`.
- Add at least three `<P>` elements containing any text. Feel free to use https://loremipsum.io/ to generate some random text (or use the Visual Studio Code's ***lorem*** or ***lorem100*** snippet).

Before we show you the solution, here is what your page should look like (see Fig. 3.2):

After you make the changes to your HTML file inside Visual Studio Code, make sure to refresh your web browser, so you see the changes. Try to generate the same webpage, then look at the solution below:

Student Information

Razvan Mezei

Lorem ipsum dolor sit amet consectetur, adipisicing elit. Deleniti voluptatum excepturi tempora eius impedit cum nemo nesciunt tempore maxime corporis error non perspiciatis necessitatibus et, earum labore nihil vel? Quibusdam.

Lorem ipsum dolor sit amet consectetur adipisicing elit. Facilis, quisquam numquam! Laborum minima qui repellat repudiandae deleniti at dolorum sint inventore, rerum quidem nam? Magni quis mollitia accusantium non vero.

Lorem ipsum dolor sit amet consectetur adipisicing elit. Necessitatibus, id quisquam velit accusamus omnis dolores modi, iusto incidunt tempore magnam aliquid neque ducimus eum consectetur, sunt facere perferendis architecto veritatis.

Fig. 3.2 Shows how the page should be displayed in a browser

```
<!DOCTYPE html>
<html>
    <head>
        <title>Info for Razvan Mezei</title>
    </head>
    <body>
        <h1>Student Information</h1>
        <h2>Razvan Mezei</h2>
        <p>Lorem ipsum dolor sit amet consectetur, adipisicing elit. Deleniti
voluptatum excepturi tempora eius impedit cum nemo nesciunt tempore maxime corporis
error non perspiciatis necessitatibus et, earum labore nihil vel? Quibusdam. </p>
        <p>Lorem ipsum dolor sit amet consectetur adipisicing elit. Facilis, quisquam
numquam! Laborum minima qui repellat repudiandae deleniti at dolorum sint inventore,
rerum quidem nam? Magni quis mollitia accusantium non vero.</p>
        <p>Lorem ipsum dolor sit amet consectetur adipisicing elit. Necessitatibus,
id quisquam velit accusamus omnis dolores modi, iusto incidunt tempore magnam aliquid
neque ducimus eum consectetur, sunt facere perferendis architecto veritatis.</p>
    </body>
</html>
```

Note: Our page may not look very pretty just yet. For that, we'll add styling (CSS) during our next chapter. So please have patience. Again, the purpose of HTML is to describe the structure of a Webpage.

Above, we used `H1` and `H2` elements. These are used to define **HTML headings**, with `H1` being the most important, `H2` less important, and so on. You should not skip levels: this means, your page should not contain a `H2` element, if the `H1` is not included anywhere on the page.

Display this as another heading

Fig. 3.3 Shows how the page should be displayed in a browser. Currently, it only contains one H1 element

3.3 Add a Second Webpage

Let's add a second webpage. On the left side of Visual Studio Code (see the screenshot below) click on the `+ New File...` icon and add a new file with the name `secondwebpage.html`.

To this file, add the following lines:

```
1.<!DOCTYPE html>
2.<HTML>
3.    <HEAD>
4.        <TITLE>Title: our second webpage</TITLE>
5.    </HEAD>
6.    <BODY>
7.        <H1>Display this as another heading</H1>
8.    </BODY>
9.</HTML>
```

Open this file in a browser and check that it looks like the screenshot below (Fig. 3.3).

3.4 Add Links and White Spaces to Our Pages

To this second page, let's add three links: one that links to www.w3schools.com, one that links to www.stmartin.edu, and one that links to our first webpage created above.

After line 7 (shown above), after the `<H1>` element, add the following five lines (notice there are two empty lines between the second and third `<A>` elements):

```
<A href="https://www.w3schools.com/">Go to W3Schools</A>
<A href="https://www.stmartin.edu/">Go to St. Martin University</A>


<A href="firstwebpage.html">Go to the first page</A>
```

Fig. 3.4 Shows the page displayed in a browser. It contains an H1 element and three A elements

Then go to your browser, refresh your page, and check how the second page looks like (see Fig. 3.4).

Is this what you expected? Let's talk about these results.

In general, we use the `<A>` elements to add **links** (**anchors**) in our HTML pages. The *content* of these elements (the text between the `<A>` and `</A>` tags) is the text that shows up in as the link. These elements made use of **attributes** to specify where the links should point to. In particular, we used `href="https://www.w3schools.com/"` to point to the w3schools' website.

Attributes, in general, are used to provide additional information needed for HTML elements. They usually come in pairs that look like `attributename="attributevalue"` (above we used: `href="https://www.w3schools.com/"`) and are specified in the *start tag* of an element. As we will see below, one can use multiple attributes in one element.

The whitespaces (space, tabs, newline, and empty lines) are mostly ignored by the browser. If you want your links to

- show on **different lines**, use the tag: `<BR>` BR stands for **break**);
- show just **spaces away** from each other, use the following: ` `

Let's replace the lines above with the following, then refresh your page in the browser and check the results:

```
<A href="https://www.w3schools.com/">Go to W3Schools</A>          
<A href="https://www.stmartin.edu/">Go to St. Martin University</A>
<BR>
<BR>
<A href="firstwebpage.html">Go to the first page</A>
```

Check the results in your browsers.

Student Information

Razvan Mezei

Lorem ipsum dolor sit amet consectetur, adipisicing elit. Deleniti voluptatum excepturi tempora eius impedit cum nemo nesciunt tempore maxime corporis error non perspiciatis necessitatibus et, earum labore nihil vel? Quibusdam.

Lorem ipsum dolor sit amet consectetur adipisicing elit. Facilis, quisquam numquam! Laborum minima qui repellat repudiandae deleniti at dolorum sint inventore, rerum quidem nam? Magni quis mollitia accusantium non vero.

Lorem ipsum dolor sit amet consectetur adipisicing elit. Necessitatibus, id quisquam velit accusamus omnis dolores modi, iusto incidunt tempore magnam aliquid neque ducimus eum consectetur, sunt facere perferendis architecto veritatis.

Go to W3Schools   Go to St. Martin University Go to the first page

Fig. 3.5 Shows how the page should show up in a browser

As an exercise, please add the same three links to the first webpage. Have them displayed on the same line, with five spaces between each other, and modify the third link to point to your *secondwebpage.html*.

Here is what you should get (see Fig. 3.5).

3.5 Add Images and White Spaces to Our Pages

To the first page, let's add an image, right after the first paragraph. For this, we will make use of the <IMG> element (note: it only has a *start* tag, there is no *end* tag for it) and use multiple attributes.

We'll use the SRC attribute to specify which image to load into the browser. This can use a relative path (for images on local machine for example) or an absolute path (typically used to link to images stored somewhere online, but you can also use it for images found on your local machines). Add the following line after the first <P> element:

```
<IMG SRC ="image01.jpg">
```

Now reload your HTML file in the browser to see the changes. The image may be displayed too large. Therefore, let's add another attribute to specify a desired height and/or width. We'll add the HEIGHT="300" attribute.

```
<IMG SRC ="image01.jpg" HEIGHT="300">
```

Lastly, let's add one more attribute, the `TITLE="This is me, resting"` attribute, which will provide a tooltip for our image:

```
<IMG SRC ="image01.jpg" HEIGHT="300" TITLE="This is me, resting">
```

We obtained the following (Fig. 3.6):

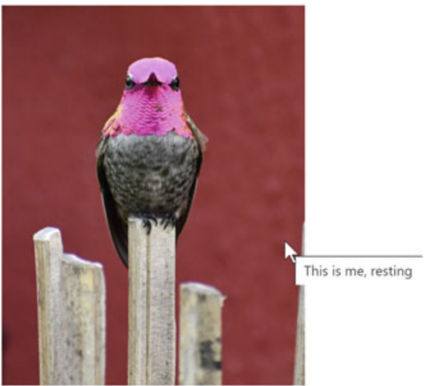


Fig. 3.6 Shows how the page should show up in a browser after we embedded an image. Note the tooltip text shown as the mouse hovers over the image (this was set by the TITLE attribute of the IMG element)

Student Information

Razvan Mezei

Lorem ipsum dolor sit amet consectetur, adipisicing elit. Deleniti voluptatum excepturi tempora eius impedit cum nemo nesciunt tempore maxime corporis error non perspiciatis necessitatibus et, earum labore nihil vel? Quibusdam.

This is me, resting

Lorem ipsum dolor sit amet consectetur adipisicing elit. Facilis, quisquam numquam! Laborum minima qui repellat repudiandae deleniti at dolorum sint inventore, rerum quidem nam? Magni quis mollitia accusantium non vero.

Lorem ipsum dolor sit amet consectetur adipisicing elit. Necessitatibus, id quisquam velit accusamus omnis dolores modi, iusto incidunt tempore magnam aliquid neque ducimus eum consectetur, sunt facere perferendis architecto veritatis.

Go to W3Schools Go to St. Martin University Go to the first page

Fig. 3.7 This is the same as the above figure, but the underlying image was removed. Note that the text "This is me, resting" is still included/displayed in the page

We should also add the `alt` attribute in case the clients seeing our page are using a screen reader, have a poor Internet connection, or the image has not been found (this attribute is also helping make the page more accessible):

```
<IMG SRC ="image01.jpg" HEIGHT="300" TITLE="This is me, resting" ALT="Profile photo
for Razvan Mezei">
```

Here (see Fig. 3.7) is how this will look if we remove our image (so that it is not found by the browser).

3.6 Tables and Buttons

Next, let's add a table and a few buttons to a webpage. For this, let's make changes to our webpage: *secondwebpage.html*. In here, let's change the TITLE and the H1 elements so they both say `Current enrollment for Razvan Mezei`.

Before we create our own table, we encourage you to visit the following page to see an example of table https://www.w3schools.com/tags/tryit.asp?filename=tryhtml_tbody_css. That example also contains CSS, which you can ignore for now. You should note the following:

- To create a table, we can make use of the `<TABLE>` element.
- Inside the `<TABLE>` elements, two nested elements are used: the `<THEAD>` element is used to create the *table headers*, and the `<TBODY>` element is used to contain the table rows.
- The `<TR>` element defines a *table row.* In a row, typically multiple `<TH>` (for table header) or `<TD>` (for table cell) elements are being used.

Please get comfortable with this code. We'll use this again when we learn about ASP .Net Core MVC—Views.

After the `<H1>` element, let's add our own `<TABLE>` element. We'll create a table that contains the courses our person is enrolled in. For example, let's assume *Razvan Mezei* is enrolled in the following courses (`CSC200: Object Oriented Programming`, `CSC495: ST ASP .Net Core MVC`, and `CSC340: Data Structures and Algorithms`). Let's also provide some imaginary links for them (for now they should just point to https://www.stmartin.edu/).

Here is how the output should look like. What HTML code can be added to accomplish this (Fig. 3.8)?

Here is a possible solution (note that we also added two `<BR>` elements at the end of the table):

```
<TABLE>
    <THEAD>
        <TR>
            <TH>Course ID</TH>
            <TH>Course name</TH>
            <TH>Course link</TH>
        </TR>
    </THEAD>
    <TBODY>
        <TR>
            <TD>CSC200</TD>
            <TD>Object Oriented Programming</TD>
            <TD><a href="https://www.stmartin.edu/">Course link</a></TD>
        </TR>
        <TR>
            <TD>CSC340</TD>
            <TD>Data Structures and Algorithms</TD>
            <TD><a href="https://www.stmartin.edu/">Course link</a></TD>
        </TR>
        <TR>
            <TD>CSC495</TD>
            <TD>ST ASP .Net Core MVC</TD>
            <TD><a href="https://www.stmartin.edu/">Course link</a></TD>
        </TR>
    </TBODY>
</TABLE>
<BR>
<BR>
```

Current Enrollment for Razvan Mezei

Course ID	Course name	Course link
CSC200	Object Oriented Programming	Course link
CSC340	Data Structures and Algorithms	Course link
CSC495	ST ASP .Net Core MVC	Course link

Go to W3Schools Go to St. Martin University Go to the first page

Fig. 3.8 Shows how an HTML table will look like in a browser

The table isn't the prettiest it could be. We'll make it look better once we see what CSS is. Please have patience.

3.7 A Few Other HTML Elements We'll Use Later

3.7.1 Label and Select Elements

The following HTML elements will show up later in the course, so we thought it would be useful to briefly introduce them in here.

Below our table let's add a <LABEL> element, which can be used to display a text. Labels can provide very useful information to screen readers (see [4]) which will read out loud the label when a user clicks on an associated element.

```
<LABEL> Choose a major </LABEL>
```

Let us associate this with a <SELECT> element (see [5]), which can be used to provide the user with a dropdown list of options to choose from.

```
<SELECT id="majors">
    <OPTION value="Art">Art</OPTION>
    <OPTION value="Math">Mathematics</OPTION>
    <OPTION value="CS">Computer Science</OPTION>
    <OPTION value="Undecided">Undecided</OPTION>
</SELECT>
```

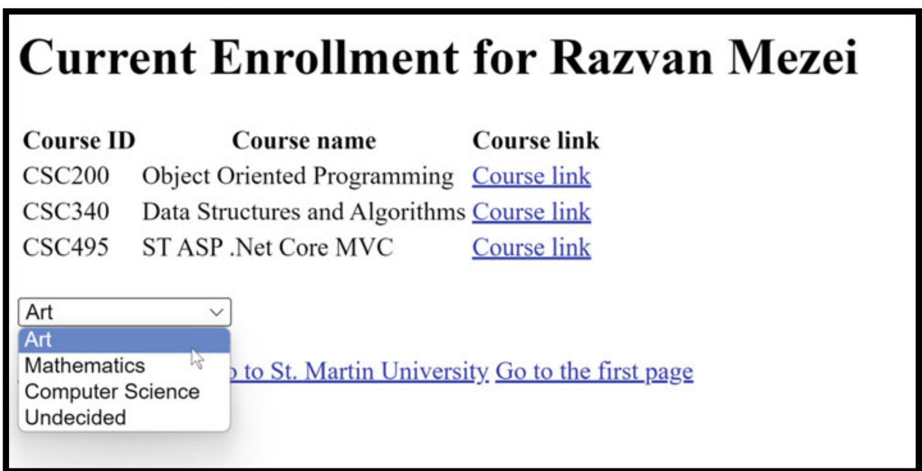


Fig. 3.9 Shows the same table as seen in Fig. 3.8, but with a newly added dropdown list

To associate a <LABEL> element with a <SELECT> you need to set a unique identifier in the <SELECT> element (we chose id="majors"), then in the <LABEL> tag add the attribute for="majors".

Here is the outcome (Fig. 3.9).

3.7.2 Input Elements

Labels also work (can be associated via the for attribute) with <INPUT> elements. There are many types of <INPUT> elements: color, password, date, number, text, and submit, to name a few (see more in here [4]). Let's briefly see them by example. Below the <SELECT> element of the *secondwebpage.html* file, add the following lines::

```
<LABEL for="passw">Password: </LABEL> <INPUT type="password" id="passw"> <BR>
<LABEL for="user">Username: </LABEL> <INPUT type="text" id="user"> <BR>
<LABEL for="color">Favorite color: </LABEL> <INPUT type="color" id="color"> <BR>
<LABEL for="cv">Upload your CV: </LABEL> <INPUT type="file" id="cv"> <BR>
<LABEL for="gd">Expected graduation date: </LABEL> <INPUT type="date" id="gd"> <BR>
```

Reload the webpage in the browser to see the changes. Then, try to interact with these <INPUT> elements. They should look similar to (Fig. 3.10).

We encourage you to interact with each of these <INPUT> elements.

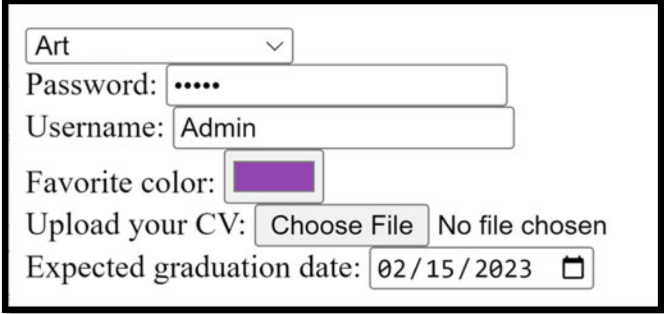


Fig. 3.10 Shows several labels and input elements of various types (including password, text, color, and file)

3.8 Form and More on Input Elements and Attributes

For this part, let's create a third webpage, call it *register.html*, and add the following HTML code in it as a starting point:

```
<!DOCTYPE html>
<HTML>
    <HEAD>
        <TITLE>Register a new account</TITLE>
    </HEAD>
    <BODY>
        <H1>Register a new account</H1>

    </BODY>
</HTML>
```

There will be times when we want to allow the user to submit multiple values at once. For example, when a user logs in, we want to collect the user's *username* and *password* and submit them together. Similarly, when we want to allow a user to *register a new account*, we typically want to collect multiple information, such as *username*, *password*, *email address*, and maybe a *phone number*.

A <FORM> element can be used to collect user information (typically from multiple <INPUT> elements). See more in [6]. Below the <H1> element from *register.html*, add a <FORM> element. If you refresh your browser, you will see that the <FORM> element by itself is not visible; it's a container for other elements that we'll add below.

Let's add a few <INPUT> elements to collect the user's information (also add <LABEL> elements, so the user knows what's expected in the form). Here is how our *register.html* file looks like after adding four <INPUT> elements nested inside the <FORM> element:

```
<!DOCTYPE html>
<HTML>
    <HEAD>
        <TITLE>Register a new account</TITLE>
    </HEAD>
    <BODY>
        <H1>Register a new account</H1>
        <FORM>
            <LABEL for="passw">Password: </LABEL> <INPUT type="password" id="passw"> <BR>
            <LABEL for="user">Username: </LABEL> <INPUT type="text" id="user"> <BR>
            <LABEL for="email">Email address: </LABEL> <INPUT type="email" id="email"> <BR>
            <LABEL for="phone">Phone number: </LABEL> <INPUT type="tel" id="phone"> <BR>
        </FORM>
    </BODY>
</HTML>
```

Refresh the browser to see the results. It should look similar to (Fig. 3.11).

Note: If you click on any <LABEL> element, the associated <INPUT> element gets focus. Try it!

To be able to send all information entered by the user in these fields inside one <FORM> element, we can add the following:

- An <INPUT> element with the attribute: type="submit". This will show up as a button!
- Each <INPUT> element whose value you want to be sent to the server must use a *name* attribute.

Let's modify our <FORM> to satisfy the two requirements above.

Register a new account
Password:
Username:
Email address:
Phone number:

Fig. 3.11 Shows a H1 element and a form containing four labels and input elements

```
<FORM>
    <LABEL for="passw">Password: </LABEL> <INPUT type="password" id="passw" name="password"> <BR>
    <LABEL for="user">Username: </LABEL> <INPUT type="text" id="user"  name="username"> <BR>
    <LABEL for="email">Email address: </LABEL> <INPUT type="email" id="email" name="email"> <BR>
    <LABEL for="phone">Phone number: </LABEL> <INPUT type="tel" id="phone" name="phone"> <BR>
    <BR>
    <INPUT type="submit" value="REGISTER NEW USER">
</FORM>
```

Note: The <INPUT type="submit" value="REGISTER NEW USER">provided us with a *button*, ready to submit all named values (see Fig. 3.12).

Important note: The name attribute used in each of our <INPUT> elements can be used by the server (when processing the request, we're sending to the server). If you omitted this for an <INPUT> element, the value of that <INPUT> element will not be sent at all to the server.

Note: We already mentioned this (using different words) but would like to emphasize the for attribute of a <LABEL> element must be equal to the id attribute of the associated <INPUT> element in order to bind them together.

Next, before the *submit button*, let's add a *checkbox* (and an associated *label*):

```
<LABEL for="veteran">Are you a veteran? </LABEL>
<INPUT type="checkbox" id="veteran"  name="isVeteran"> <BR>
```

This added a checkbox to our form (see Fig. 3.13).

We finish this subsection with a quick introduction to various important *attributes* used for <INPUT> elements:

Fig. 3.12 Shows the same elements as seen in Fig. 3.11, but with a newly added input element (displayed as a button)

Register a new account

Password:
Username:
Email address:
Phone number:
Are you a veteran?

REGISTER NEW USER

Fig. 3.13 To Fig. 3.12 we now added a label and an input element of type checkbox

- Use the *value* attribute to specify a default value for an <INPUT> element.
- Use the *placeholder* attribute to specify a hint for an <INPUT> element. When the user clicks on the element, the placeholder text/hint disappears.
- Use the *required* attribute to specify that an <INPUT> element must contain a value before the <FORM> is submitted (and the values sent to the server).
- There are many more attributes that can be used with <INPUT> elements. Here is one source to check: [7].

Let's see some examples. Change the <INPUT> elements above to match the code below:

```
<FORM>
    <LABEL for="passw">Password: </LABEL>
    <INPUT type="password" id="passw" name="password" required>
    <BR>
    <LABEL for="user">Username: </LABEL>
    <INPUT type="text" id="user"  name="username" placeholder="choose a username">
    <BR>
    <LABEL for="email">Email address: </LABEL>
    <INPUT type="email" id="email"  name="email">
    <BR>
    <LABEL for="phone">Phone number: </LABEL>
    <INPUT type="tel" id="phone"  name="phone" value="000-000-0000">
    <BR>
    <LABEL for="veteran">Are you a veteran? </LABEL>
    <INPUT type="checkbox" id="veteran"  name="isVeteran">
    <BR>
    <BR>
    <INPUT type="submit" value="REGISTER NEW USER">
</FORM>
```

Fig. 3.14 If the user attempt to submit a form without a text in the password field (which was set as required), then an error ("Please fill out this field") will be displayed

Refresh the webpage in the browser, and check the results.

Then, if you click on submit, without entering anything in the Password field (which was set as required!), you should get an error message (see Fig. 3.14).

3.9 GET Versus POST Request, the Action and the Method Attributes

There are just a few more things to know about the <FORM> elements (we'll make use of these in future chapters!). We'll talk next about the action attribute and the method attribute for <FORM> elements.

We use the action attribute of the <FORM> element to specify where to send your request. Who/which code on the server side should process your request? Here is an example of how to use this attribute. Change the <FORM> tag so it matches the code below:

```
<FORM action="/register.php">
```

In the browser, if you fill out the form, and click on the SUBMIT button, you will probably get an error message (we did not set up the server side yet—please have patience).

The next attribute we want to discuss about is the method. We review it in more details in a future chapter and make extensive use of it, but we would like to briefly introduce it in here.

GET and POST are two (there are several others, and you can also create your own) **HTTP verbs.** We can use either of them when we send requests to servers. But the difference between them is very important.

- Use the GET method to inform the browser to send the request data as part of the URL.
  - o This is very useful if we want to allow users to bookmark searches so that it contains field values.
  - o An example: https://www.amazon.com/s?k=lenovo+laptops.
  - o The data is appended to the URL request, after the ? (the portion of the URL that comes after the ? is called a **query string**).
  - o There are limitations on how much data can be sent using GET requests, in particular one can only send text data.
- Use the POST method to inform the browser to send the request data as part of the *request body*, not part of the URL.
  - o This is useful if you are sending sensitive information that should not be bookmarked.
  - o An example: when you login, you probably don't want your password to show up in the browser's history.
  - o Imagine seeing this in a browser: https://www.amazon.com/s?username=alex&password=Password123.
  - o Another example: you can't send a file in the URL portion of the request. So, if you want to upload files to a server, you will want to use the POST method.
  - o There are fewer limitations on what data you can send via this method. You are not limited to text data, you can also send binary (for example, images, pdfs, etc.).
- See more on this here: [8].

Let's see this in practice. For this part, please set the action attribute to "#" (this will send the request back to the current page):

```
<FORM action="#">
```

Example 1: add the attribute method="get" to your form.

```
<FORM action="#" method="get">
```

Then fill in some values in the form, click on the submit button, and observe the URL.

Notice that the URL contains the following **query string** (the part of the URL that comes after the question mark):

?password=secretpassword&username=admin&email=admin%40admin.com&phone=000-000-0000&isVeteran=on#

The query string contains **name=value** pairs, where **name** is the attribute we used for our `<INPUT>` elements, and **value** is the actual value entered in the `<INPUT>` element.

Now change the attribute to method="post" and submit the same data as before (make sure to first remove the query string from the URL). You will see that there is no query string being sent this time:

If you want, check out the following two examples from W3Schools:

- For the GET method: https://www.w3schools.com/html/tryit.asp?filename=tryhtml_form_get.
- For the POST method: https://www.w3schools.com/html/tryit.asp?filename=tryhtml_form_post.

We'll see more about GET versus POST in future chapters, but we wanted to quickly introduce them in here.

There are many more HTML elements (and topics in general) to cover, but for the purposes of this book, and what we'll need when we cover the ASP.Net (Core) MVC, we believe we covered sufficient material. We'll learn more HTML elements as we need them. In the meantime, feel free to explore the following great resource [1].

Check for updates

4 Brief Introduction to CSS, Javascript, and Bootstrap

4.1 Motivation for Using CSS and JavaScript

In the previous chapter, we've seen how one can use **HTML** (Hypertext Markup Language) to define the structure of a webpage. When we have information to display in a browser, we make use of various HTML tags/elements to tell the browser that certain parts of this information represent various headings, while others represent paragraphs, tables, hyperlinks, images, and so on.

In this chapter, we will make use of **CSS** (Cascading Style Sheets) which can help us specify styling for our webpages. For example, we can specify if we want our headings to be colored in blue, display our paragraphs as centered/left/right aligned, and many other styling options.

Lastly, we will also make use of **JavaScript** to make our webpages more interactive. We can program responses to various events. For example, when a user clicks on a button, what should the webpage do?

Using *CSS* and *JavaScript* (or better yet, a library such as **Bootstrap**) we'll be able to make the basic HTML table (and other elements) from our *secondwebpage.html* look much better. Right now, the table with no style looks like (Fig. 4.1).

By the end of this chapter, the very same table (seen in Fig. 4.1), but with styling added, will look like (Fig. 4.2).

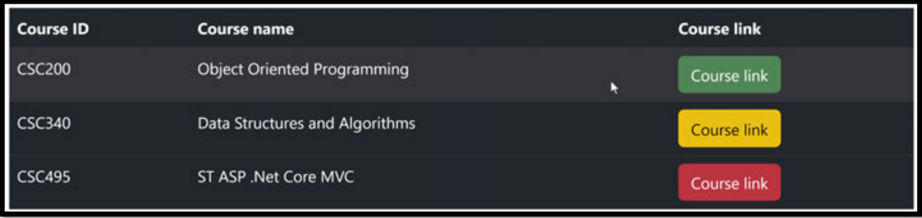
We hope this sparked your interest. Before we dive in, we would like to mention that this chapter is only meant to be an introduction to CSS and JavaScript. Enough for what we'll need for the other chapters in this book, those that focus on the ASP .Net Core MVC. But there is much more out there to learn, and we encourage you to explore it. Some useful resources (and sources of inspiration for this book) are as follows:

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024

R. A. Mezei, *Introduction to the Development of Web Applications Using ASP .Net (Core) MVC*, Synthesis Lectures on Computer Science,
https://doi.org/10.1007/978-3-031-30626-6_4

Course ID	Course name	Course link
CSC200	Object Oriented Programming	Course link
CSC340	Data Structures and Algorithms	Course link
CSC495	ST ASP .Net Core MVC	Course link

Fig. 4.1 Shows a table with no styling



Course ID	Course name	Course link
CSC200	Object Oriented Programming	Course link
CSC340	Data Structures and Algorithms	Course link
CSC495	ST ASP .Net Core MVC	Course link

Fig. 4.2 Shows the same table as shown in Fig. 4.1, but with styling (in particular, Bootstrap) added

- CSS Tutorial: https://www.w3schools.com/css/default.asp.
- JavaScript Tutorial: https://www.w3schools.com/js/default.asp.
- Bootstrap 5 Tutorial: https://www.w3schools.com/bootstrap5/index.php.

Important note: Just like HTML, CSS is NOT case sensitive. But JavaScript **IS** case sensitive. So be very careful with this.

4.2 Our First CSS Example

Let's start with an example. Let's say that we would like our *firstwebpage.html* to have:

- a light blue background for the entire webpage,
- a green and centered h1 element, and
- a light gray background color for each paragraph.

For this, we'll make use of CSS. One way to add CSS to a webpage is to add a `<STYLE>` element nested inside the `<HEAD>` element of the page. Add the following CSS code to your *firstwebpage.html* source file:

```
<STYLE>
    BODY{background-color:lightblue;}
    H1{color:green; text-align: center;}
    P{background-color: lightgray;}
</STYLE>
```

Let's see how the CSS code above styled our page. You are given below a screenshot without (Fig. 4.3) and one with (Fig. 4.4) the above given CSS code (which we'll explain below).

4.3 Introduction to CSS Syntax

Now let's talk about the CSS syntax. To add CSS, we can use the `<STYLE>` element (we'll see other ways below). The CSS syntax typically follows the following format:

```
selector {property1:value1; property2:value2; …}
```

The **selector** determines where to apply the styling specifications between the curly braces (`{…}`). Inside these curly braces, we can include one or more *property-value* pairs, separated by semicolons (**;**). Inside each **property-value** pair, we use the colon (**:**). Spacing is largely ignored by CSS, so we make use of it to make our code easier to read.

For example,

```
P{background-color: lightgray;}
```

In here we used:

- the (*element*) *selector* `P` (meaning the styling specifications inside the curly braces `{…}` will be applied to all paragraph `<P>` elements of the page), and
- the property-value pair: `background-color:lightgray` (meaning we want these paragraphs to have a gray background color).

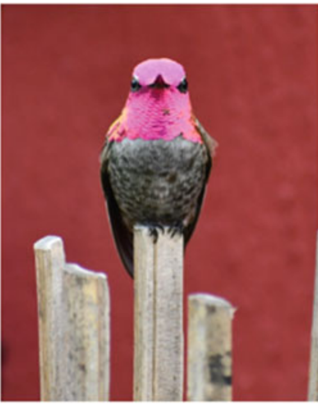
Here is another example:

```
H1{color:green; text-align: center;}
```

Student Information

Razvan Mezei

Lorem ipsum dolor sit amet consectetur, adipisicing elit. Deleniti voluptatum excepturi tempora eius impedit cum nemo nesciunt tempore maxime corporis error non perspiciatis necessitatibus et, earum labore nihil vel? Quibusdam.



Lorem ipsum dolor sit amet consectetur adipisicing elit. Facilis, quisquam numquam! Laborum minima qui repellat repudiandae deleniti at dolorum sint inventore, rerum quidem nam? Magni quis mollitia accusantium non vero.

Lorem ipsum dolor sit amet consectetur adipisicing elit. Necessitatibus, id quisquam velit accusamus omnis dolores modi, iusto incidunt tempore magnam aliquid neque ducimus eum consectetur, sunt facere perferendis architecto veritatis.

Go to W3Schools Go to St. Martin University Go to the second page

Fig. 4.3 **(Left)** Shows the firstwebpage.html file displayed in a browser, before adding CSS styling

In here we used.

- another element selector: `H1` (meaning this is applied to all heading `H1` elements from the page), and
- multiple property-value pairs:
  - o `color:green` (meaning we want the `H1` elements to have a green text color) and
  - o `text-align:center` (meaning we want the `H1` elements to have a centered text alignment).

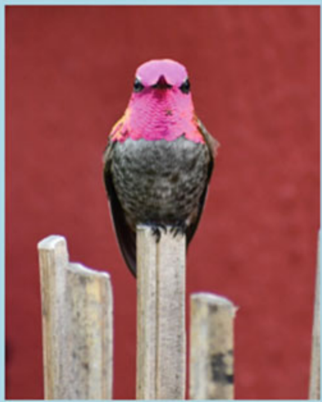


Fig. 4.4 **(Right)** Shows the firstwebpage.html file displayed in a browser, after adding CSS styling

4.4 CSS Selectors

There are many types of CSS selectors, but in this book, we'll only focus on a few of them, namely the *element selectors*, the *id selectors*, and the *class selectors.*

In the example given above, we used the following element selectors: `BODY`, `H1`, and `P`. They are called **element selectors** because they specify to which HTML *elements* should the styling specifications apply. The example `P{background-color: lightgray;}` applies the given styling to **all** *paragraph* elements.

What if we only want to apply certain styling to only one of the paragraphs? To apply CSS styling to only one element (paragraph, table, heading1, etc.), we can use **id selectors.** For this, we can follow the two steps shown as follows:

- We define/add an **id** *attribute* (with a unique identifier of our choice) to our selected element.

  o For example, let's change our first `<P>` tag into

```
<P ID="thisisspecial">
```

- Then use an **id** *selector* in our CSS specifications (make sure to use #).

  o For example, we can add the following CSS specification inside the `<STYLE>` element:

```
#thisisspecial{background-color: rgb(222,157,210);}
```

The result is that we uniquely identified an element (one paragraph in our example) and applied certain CSS styling to it. In our example, the paragraph will have a pink background (see Fig. 4.5).

Id selectors should only be applied to one/unique element in a webpage. If we want to apply certain styling to multiple elements, we can make use of the *class selectors. Class selectors* can be applied to multiple elements of the same type (say multiple paragraphs) or of different types (say a table and a paragraph). To use **class selectors,** we again use two steps:

- We add a **class** *attribute* (choose a unique identifier) to our selected elements.

  o For example, let's change our first `<P>` tag into

```
<P ID="thisisspecial" CLASS="myclass">
```

  o Also add CLASS="myclass" inside the `<H2>` tag.

- Then use a **class** *selector* in our CSS specifications (make sure to use a.).

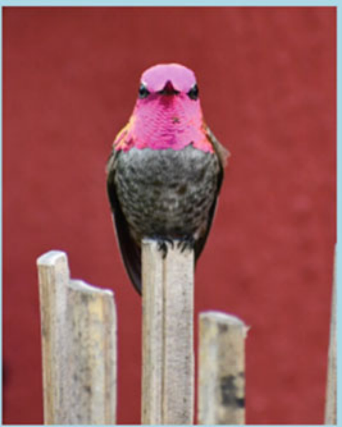
  o For example, we can add the following CSS specification inside the `<STYLE>` element:

```
.myclass{color:blue; text-align: center;}
```

Student Information

Razvan Mezei

Lorem ipsum dolor sit amet consectetur, adipisicing elit. Deleniti voluptatum excepturi tempora eius impedit cum nemo nesciunt tempore maxime corporis error non perspiciatis necessitatibus et, earum labore nihil vel? Quibusdam.



Lorem ipsum dolor sit amet consectetur adipisicing elit. Facilis, quisquam numquam! Laborum minima qui repellat repudiandae deleniti at dolorum sint inventore, rerum quidem nam? Magni quis mollitia accusantium non vero.

Lorem ipsum dolor sit amet consectetur adipisicing elit. Necessitatibus, id quisquam velit accusamus omnis dolores modi, iusto incidunt tempore magnam aliquid neque ducimus eum consectetur, sunt facere perferendis architecto veritatis.

Go to W3Schools Go to St. Martin University Go to the second page

Fig. 4.5 Shows the result of applying specific CSS styling to an element (first paragraph) selected using an ID selector

Looking at the outcome below (Fig. 4.6), can you guess which elements were applied the *class selector* defined above?

We give below the entire code for *firstwebpage.html*:

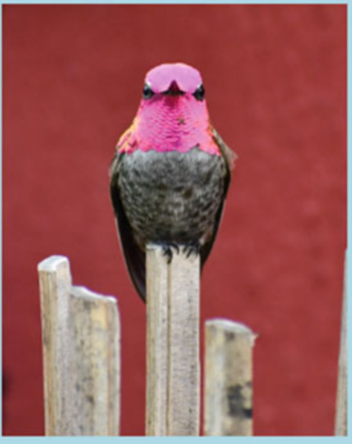


Fig. 4.6 This is similar to Fig. 4.5, but certain elements were applied styling via class selectors. Can you figure out which elements?

```
<!DOCTYPE html>
<HTML>
    <HEAD>
        <TITLE>Info for Razvan Mezei</TITLE>
        <STYLE>
            BODY{background-color:lightblue;}
            H1{color:green; text-align: center;}
            P{background-color: lightgray;}

            #thisisspecial{background-color: rgb(222,157,210);}
            .myclass{color:blue; text-align: center;}
        </STYLE>

    </HEAD>
    <BODY>
        <H1 >Student Information</H1>
        <H2 CLASS="myclass">Razvan Mezei</H2>



        <P ID="thisisspecial" CLASS="myclass"> Lorem ipsum dolor sit amet consectetur, adipisicing
elit. Deleniti voluptatum excepturi tempora eius impedit cum nemo nesciunt tempore maxime corporis
error non perspiciatis necessitatibus et, earum labore nihil vel? Quibusdam. </P>
        <IMG SRC ="image01.jpg" HEIGHT="300" TITLE="This is me, resting" ALT="Profile photo for
Razvan Mezei">
        <P>Lorem ipsum dolor sit amet consectetur adipisicing elit. Facilis, quisquam numquam!
Laborum minima qui repellat repudiandae deleniti at dolorum sint inventore, rerum quidem nam? Magni
quis mollitia accusantium non vero.</P>
        <P>Lorem ipsum dolor sit amet consectetur adipisicing elit. Necessitatibus, id quisquam
velit accusamus omnis dolores modi, iusto incidunt tempore magnam aliquid neque ducimus eum
consectetur, sunt facere perferendis architecto veritatis.</P>

        <A href="https://www.w3schools.com/">Go to W3Schools</A>            
 
        <A href="https://www.stmartin.edu/">Go to St. Martin University</A>      
   
        <A href="secondwebpage.html">Go to the second page</A>
    </BODY>
</HTML>
```

Important observation: From the example above, you should note that we can apply multiple *class* and/or *id selectors* to the same element.

4.5 Conflicting CSS Specifications

A related and important question may be what happens if two selectors have conflicting specifications? For example, one chooses a color of red, while the other chooses blue. Which one wins?

The answer is more complex (see the links below for a more in-depth explanation), but here is a simplification that is sufficient for this book:

- The most specific wins, regardless of the order in which they are specified.
  In particular, *id selector* > *class selector* > *element selector*.
- For the same specificity, the last one wins.

We direct the reader to check out the following two resources that will go into more depth:

- CSS Specificity: https://www.w3schools.com/css/css_specificity.asp.
- Cascade, specificity, and inheritance:
  https://developer.mozilla.org/en-US/docs/Learn/CSS/Building_blocks/Cascade_and_inheritance.

4.6 Other CSS Selectors

There are other types of selectors, for example, *universal selectors*, *grouping selectors*, *child selectors*, *descendant selector*s, *attribute selectors*, *pseudo-selectors,* and so on. We won't go over them in this book, but we do want to encourage you to research about them as you need them.

To provoke your interest, add the following line to your CSS specifications (add it right before the </STYLE> tag):

```
P::first-letter{color:red; font-weight: bold;}
```

What did this CSS line do?

4.7 A Few More Examples of Property-Value Pairs for CSS

Earlier, we've seen the CSS syntax:

```
selector {property1:value1; property2:value2; …}
```

and covered some of the most important CSS selectors and included some important property-value pairs to use in our CSS code. We focused primarily on understanding the CSS syntax. Below we will introduce a few more property-value pairs we can make use of in later chapters. Since we skipped many useful CSS properties (that we may not use in this course), we refer the readers to check out [1], a resource we often use in our in-class demonstrations.

Fig. 4.7 This screenshot shows how the H1 element will be displayed after adding the text color styling described above

4.7.1 Text Color in CSS

As seen above, to set the color of a given text, one can use the `color` property. Similarly, to set the background color for a given text, one can use the `background-color` property.

For example, let's add CSS specifications for `H1` elements so the `H1` selector looks as shown below:

```
H1{color:green; background-color: aliceblue; text-align: center;}
```

The `H1` element from our *firstwebpage.html* now looks as follows (see Fig. 4.7).

Besides specifying colors by name (for example, `background-color: aliceblue;`) one can also use

- hexadecimal values, representing Red, Green, Blue
  (the equivalent example: `background-color: #f0f8ff`) or
- the three whole numbers between 0 and 255 sent to the rgb function
  (the equivalent example: `background-color: rgb(240, 248, 255)`).

Use the following tool to help you pick a great color for your design: https://www.w3schools.com/colors/colors_picker.asp.

4.7.2 Text Alignment in CSS

To specify the horizontal text alignment, one can use the `text-alignment` property and make use of the following values: center, left, right.

Earlier we have already seen:

```
H1{color:green; text-align: center;}
```

On your own, modify the value of text-align property to make use of other values.

4.7.3 Fonts in CSS

To choose a font for our text, we can make use of the `font-family` property. As you start typing.

```
BODY{background-color: ligthblue; font-family: }
```

IntelliSense from Visual Studio Code has some suggestions ready to use, for example,

```
'Courier New', Courier, monospace
'Franklin Gothic Medium', 'Arial Narrow', Arial,…
```

In our example (*firstwebpage.html*), we added the following to our BODY selector:

```
font-family:'Franklin Gothic Medium', 'Arial Narrow', Arial, sans-serif
```

and obtained (Fig. 4.8).

Notes: We specified four values, `'Franklin Gothic Medium'`, `'Arial Narrow'`, `Arial`, and `sans-serif`:

- If a font name includes spaces, then you need to make use of quotes around that name.
- The order is important: the browser will use the first available font from the list (or use a default font if none from the given list are available). So, start with the one you prefer, but provide "fallback" options afterward.

  o We encourage you to read more about this (including the "fallback" system) at [2].

You may want to try the following `font-family: Papyrus, Helvetica, sans-serif;` see if you like this better (Fig. 4.9).

To use `italic` text, we can specify the `font-style` property. For bold, use the `font-weight` property.

Fig. 4.8 A font styling has been added to the BODY element (via the BODY element selector)

To specify text size, one can use the `font-size` property. There are multiple ways to specify a value for the font-size property, including the following:

- Use **viewport width** to get text whose size will depend on the browser window.

Fig. 4.9 This is similar to Fig. 4.8, but a different font-family is being used

o Note: `1 vw` = 1% of the viewport width.
Example: `font-size: 10vw;`

- Use px to specify a text size dependent on the **number of pixels**.
  Example: `font-size: 40px;`

Add the following to your CSS code and see what happens if you resize the width of the webpage:

```
H2{font-size:10vw;}
```

Challenge:

- Add the following to your CSS code and try to find out what it does (hint: select some text inside the browser):

```
::selection { background-color: lightgreen; }
```

4.8 The Box Model and the Developer Tools

Each HTML element is considered a **box**. That means that each element has the following:

- **Content**—this is where text/images appear.
- **Padding**—a transparent area around the *content*.
- **Border**—an area that goes around the *padding*.
- **Margin**—a transparent area outside the *border*.

One can view this in a browser, by pressing the **F12** key, which will open the **Developer Tools**. There are lots of great things to say about the Dev. Tools, but we will probably not make much use of it in this book, due to time constraints. We strongly encourage you to find out more (on your own) about the *Developer Tools*.

Using the *Developer Tools*, one can select an HTML element (under the **Elements** tab) and inspect or temporarily modify various values of your webpage (do not worry, your changes will not affect/change the underlying webpage!).

Under **Styles** tab (see the bottom left side of the *Dev tools* page), one can find the *box model* of the selected element. Currently, this is similar to the following (see Fig. 4.10).

Let's change various values of the *padding*, *border*, and *margin*. Double click on the "-" to give them some values. For example (see Fig. 4.11),

will change the appearance of the corresponding HTML element (a paragraph, <P> element, for us) into (Fig. 4.12).

Note: Changes made inside the Dev tools are not permanent. Let's make such changes inside our CSS code defined inside the *firstwebpage.html* file.

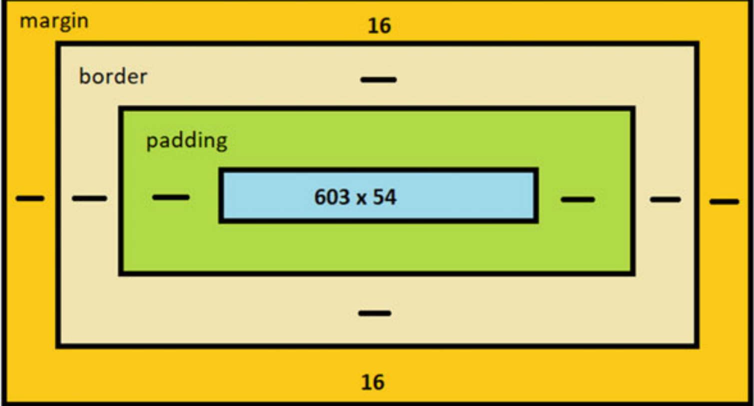


Fig. 4.10 Shows the box model. Notice how the content is in the middle. Then, comes the padding. Then the border, and finally the margin

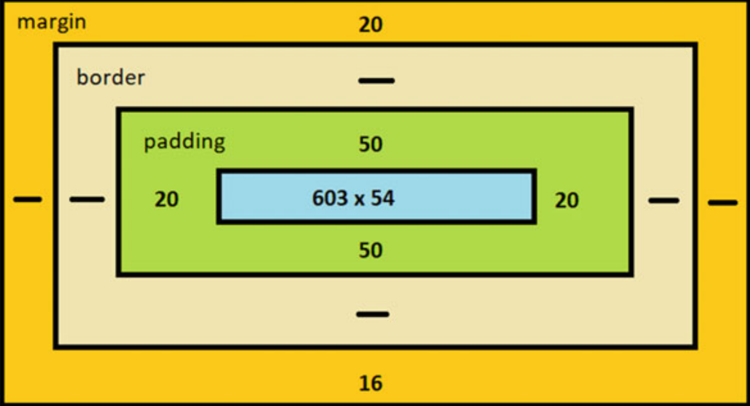


Fig. 4.11 Shows the box model seen in Fig. 4.10, but some of the values (for margin and padding) have been changed

Lorem ipsum dolor sit amet consectetur adipisicing elit. Facilis, quisquam numquam! Laborum minima qui repellat repudiandae deleniti at dolorum sint inventore, rerum quidem nam? Magni quis mollitia accusantium non vero.

Fig. 4.12 Shows how the P element will look once we change some of the box-model values (for margin and padding)

```
p {
  background-color: lightyellow;
  width: 70%;
  border: 15px solid green;
  padding: 50px;
  margin: 20px;
}
```

Your page will now look as follows (see Fig. 4.13).

4.9 The DIV Element

<DIV> elements can be used to define a **division**, or better named a **section**, in an HTML document. We will use it as a container of elements (nest various elements inside a <DIV> element and style it using CSS).

4.10 Ways to Add CSS

4.10.1 Internal

There are multiple ways of applying CSS to our webpages. What we have seen so far is what's called *internal CSS*. For **internal CSS**, one adds a <STYLE> element in the <HEAD> element of a webpage and adds the desired CSS code in there.

4.10.2 In-line

Another way to add CSS is called *in-line CSS*. For **in-line CSS**, one would add a style attribute directly inside the desired HTML element. For example,

```
<A href="firstwebpage.html" style="color:red">Go to the second page</A>
```

Fig. 4.13 This is how the firstwebpage.html file will look after applying all the styling described above

4.10.3 External

Probably the best way to apply CSS to your webpages is by using external .css files. This way, one could reuse the same one (or more) file across multiple webpages. If changes need to be made, you only need to modify one file (and affect all webpages that use that .css file). To move our CSS code to **external CSS** files, we'll need to:

- create a file, with extension .css. In this file, we put the code that is inside the <STYLE> element (not including the <STYLE> tags);
- to apply the CSS code from the file created above, we then need to link it inside the webpage (see below).

Let's modify our webpage, so it uses an external CSS file. Then, let's link this file to all three webpages we created so far.

Step 1: Move all the CSS code between (not including) the <STYLE> tags into a newly created file, let's name it *personal.css*, and (for simplicity) put it inside the same directory as your webpages. Here are the contents we added to this file:

```
BODY{background-color:lightblue; font-family:'Franklin Gothic Medium', 'Arial Narrow', Arial, sans-serif}
H1{ color:green;
    background-color: aliceblue;
    text-align: center; }
/* P{background-color: lightgray;} */

#thisisspecial{background-color: rgb(222,157,210);}
.myclass{color:blue; text-align: center;}

P::first-letter{color:red; font-weight: bold;}
H2{font-size:10vw;}

::selection { background-color: lightgreen; }

p {
    background-color: lightyellow;
    width: 70%;
    border: 15px solid green;
    padding: 50px;
    margin: 20px;
}
```

Note: Refresh your webpage in the browser, and check that your styling is now gone.

Step 2: Link this external file into the *firstwebpage.html* (add the line below in place of the <STYLE> element).

```
<LINK rel="stylesheet" href="personal.css">
```

Copy the link above to all other webpages and see how the CSS file we just created above affects them.

Note: In any given webpage, one can make use of any or all three ways to add CSS seen above.

4.11 First Encounter with Bootstrap

We hope that we have convinced you that styling is a great way to improve the look and feel of a webpage. But there is more to it. We've seen above how you can link a .css file and use it to multiple webpages. In particular, one can use the same .css file for an entire website, helping create a consistent look and feel. More importantly, you can use .css files created by others. There are free and open-source .css files that you can just learn how to use them and utilize in your web application.

One such file can be found at https://cdn.jsdelivr.net/npm/bootstrap@5.2.3/dist/css/boo tstrap.min.css. This is part of the **Bootstrap 5** framework, "which is the most popular HTML, CSS, and JavaScript framework for creating responsive, mobile-first websites" ([3]). Bootstrap 5 is free and open source.

To link this .css file to our webpages, we just need to add.

```
<LINK href="https://cdn.jsdelivr.net/npm/bootstrap@5.2.3/dist/css/bootstrap.min.css"
rel="stylesheet">
```

4.11.1 Add Bootstrap 5 .css to Our Webpages

Add the following two <LINK> elements to all three webpages we created so far, and add them right after the <TITLE> element:

```
<LINK href="https://cdn.jsdelivr.net/npm/bootstrap@5.2.3/dist/css/bootstrap.min.css"
rel="stylesheet">
<LINK rel="stylesheet" href="personal.css">
```

As of right now, after we linked these .css files into the *secondwebpage.html* file, our table looks like this (see Fig. 4.14).

Next, let's make use of the CSS classes defined in Bootstrap 5. For more details, please refer to [3].

Course ID	Course name	Course link
CSC200	Object Oriented Programming	Course link
CSC340	Data Structures and Algorithms	Course link
CSC495	ST ASP .Net Core MVC	Course link

Fig. 4.14 Shows the table from the *secondwebpage.html* file, after the external CSS styling was applied to this file

4.11.2 Bootstrap 5 Tables

Tables are something we see quite often on webpages. So, it should not come as a surprise to find that there are CSS libraries that can help us make our tables look more professional. Bootstrap 5 contains CSS class definitions that are very useful for tables. One such class is the .table class. Let's use it with our table. To the `<TABLE>` element, add the .table class (replace `<TABLE>` with `<TABLE class="table">`). Our table will now look a little better (Fig. 4.15).

There are more classes defined in Bootstrap 5. In particular, we'll use the following (please add this to the `<TABLE>` element): `class="table table-dark table-hover"`. Here is the outcome (Fig. 4.16).

Course ID	Course name	Course link
CSC200	Object Oriented Programming	Course link
CSC340	Data Structures and Algorithms	Course link
CSC495	ST ASP .Net Core MVC	Course link

Fig. 4.15 Shows the same table as the one from Fig. 4.14, after we added the .table class selector (defined by Bootstrap 5)

Course ID	Course name	Course link
CSC200	Object Oriented Programming	Course link
CSC340	Data Structures and Algorithms	Course link
CSC495	ST ASP .Net Core MVC	Course link

Fig. 4.16 Shows the same table as the one from Fig. 4.14, after we added the CSS selector `class="table table-dark table-hover"` (defined in Bootstrap 5)

4.11.3 Bootstrap 5 Buttons and Links

Using Bootstrap 5 we can also make our `<A>` elements/links look much better. In particular, we can make the links shown above to look like buttons. Also, there are different colors we can use for our buttons, for example,

- btn-primary gives us a blue button;
- btn-success gives us a green button;
- btn-warning gives us a yellow button;
- btn-danger gives us a red button.

These classes must be used in conjunction with the `.btn` class. Read more about this in [4]. In there, you'll find other classes that can be used, for example, ones that allow you to change the size, the outline, and so on.

In our *secondwebpage.html*, we added the classes mentioned above to make our page look better. For example, we replaced:

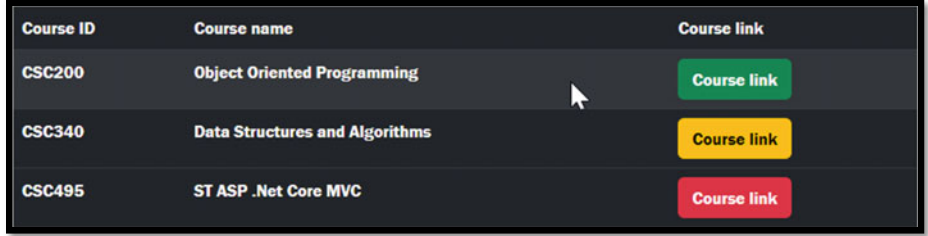
```
<TD><a href="https://www.stmartin.edu/">Course link</a></TD>
With:

<TD><a href="https://www.stmartin.edu/"  class="btn btn-success" >Course link</a></TD>
```

Here is the outcome (Fig. 4.17).

As an exercise, make the necessary changes to your *secondwebpage.html* so it now looks as follows (Fig. 4.18).

On your own, also improve the other webpages.



Course ID	Course name	Course link
CSC200	Object Oriented Programming	Course link
CSC340	Data Structures and Algorithms	Course link
CSC495	ST ASP .Net Core MVC	Course link

Fig. 4.17 Shows the same table as the one from Fig. 4.16, after we added CSS selectors such as `class="btn btn-success"` and `class="btn btn-warning"` (defined in Bootstrap 5) to the A elements in the table

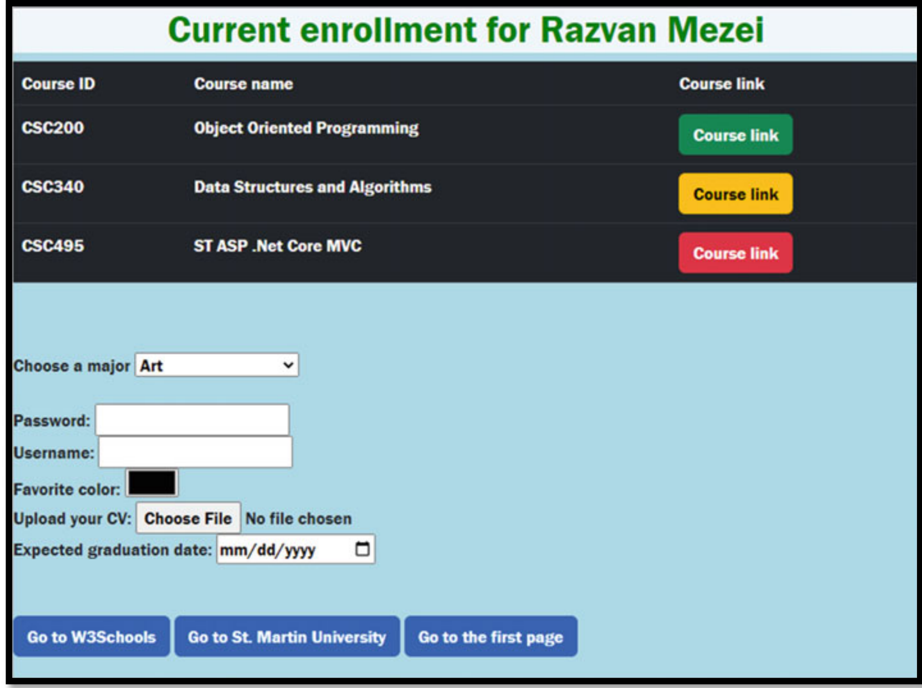


Fig. 4.18 This image is left as an exercise for the readers to figure out what styling to add, in order to get their pages to look like this one

4.11.4 Bootstrap 5 Container, Padding

One last improvement we'll add to our webpages. For each webpage, add all its contents inside a `<DIV>` element with the CSS class of `class="container-fluid p-5"`. This will provide some padding around our webpages. Here is how our webpage looks right now (see Fig. 4.19).

4.11.5 Bootstrap 5 Source Code

There is much more about Bootstrap, but we won't use more details right now (except for the following). Besides the sources we recommended above that should be great for learning Bootstrap 5 in more details, we want to encourage you to directly open the link used above in a web browser. You should see the actual CSS code we made use of above:

```
https://cdn.jsdelivr.net/npm/bootstrap@5.2.3/dist/css/bootstrap.min.css
```

Fig. 4.19 This figure is similar to Fig. 4.18, but it has more padding added because of the styling described above

If you remove the.min from the link (**min** stands for **minified**), you can see a more reader friendly version of the same source code, but this version now contains comments and more spacing, making it easier for a developer to read it:

```
https://cdn.jsdelivr.net/npm/bootstrap@5.2.3/dist/css/bootstrap.css
```

4.11.6 Center Contents with <DIV> and CSS

We want to finish this part of the chapter that focuses on CSS with the following example. We would like the three buttons at the bottom of the webpage to be nicely centered. For this, we can put them inside a `<DIV>` element and apply CSS to this element.

In our example, we will create a class selector, named `CenterThoseLinks`, and then apply styling to it.

The `<DIV>` element now looks like.

```
<DIV class="CenterThoseLinks">
    <A href="https://www.w3schools.com/" class="btn btn-primary">Go to W3Schools</A>
    <A href="https://www.stmartin.edu/"  class="btn btn-primary">Go to St. Martin University</A>
    <A href="firstwebpage.html" class="btn btn-primary">Go to the first page</A>
</DIV>
```

And, inside the *personal.css* file we included:

```
.CenterThoseLinks{
    text-align: center;
}
```

Now the page has nicely aligned buttons at the bottom of the page (Fig. 4.20).

For completeness, and for you to be able to verify your work, we provide here the entire source code for *secondwebpage.html*:

```
<!DOCTYPE html>
<HTML>
    <HEAD>
        <TITLE>Current enrollment for Razvan Mezei</TITLE>
        <LINK href="https://cdn.jsdelivr.net/npm/bootstrap@5.2.3/dist/css/bootstrap.min.css" rel="stylesheet">
        <LINK rel="stylesheet" href="personal.css">
</HEAD>
<BODY>
    <DIV class="container-fluid p-5">
        <H1>Current enrollment for Razvan Mezei</H1>

        <TABLE class="table table-dark table-hover">
            <THEAD>
                <TR>
                    <TH>Course ID</TH>
                    <TH>Course name</TH>
                    <TH>Course link</TH>
                </TR>
            </THEAD>
            <TBODY>
                <TR>
                    <TD>CSC200</TD>
                    <TD>Object Oriented Programming</TD>
                    <TD><a href="https://www.stmartin.edu/"  class="btn btn-success" >Course link</a></TD>
                </TR>
                <TR>
                    <TD>CSC340</TD>
                    <TD>Data Structures and Algorithms</TD>
                    <TD><a href="https://www.stmartin.edu/" class="btn btn-warning">Course link</a></TD>
                </TR>
```

```
                <TR>
                    <TD>CSC495</TD>
                    <TD>ST ASP .Net Core MVC</TD>
                    <TD><a href="https://www.stmartin.edu/" class="btn btn-danger">Course link</a></TD>
                </TR>
            </TBODY>
        </TABLE>
        <BR>
        <BR>
        <LABEL> Choose a major </LABEL>
        <SELECT id="majors">
            <OPTION value="Art">Art</OPTION>
            <OPTION value="Math">Mathematics</OPTION>
            <OPTION value="CS">Computer Science</OPTION>
            <OPTION value="Undecided">Undecided</OPTION>
        </SELECT>

        <BR>
            <BR>
        <LABEL for="passw">Password: </LABEL> <INPUT type="password" id="passw"> <BR>
        <LABEL for="user">Username: </LABEL> <INPUT type="text" id="user"> <BR>
        <LABEL for="color">Favorite color: </LABEL> <INPUT type="color" id="color"> <BR>
        <LABEL for="cv">Upload your CV: </LABEL> <INPUT type="file" id="cv"> <BR>
        <LABEL for="gd">Expected graduation date: </LABEL> <INPUT type="date" id="gd"> <BR>

        <BR>
       <BR>

       <DIV class="CenterThoseLinks">
           <A href="https://www.w3schools.com/" class="btn btn-primary">Go to W3Schools</A>
           <A href="https://www.stmartin.edu/"  class="btn btn-primary">Go to St. Martin University</A>
           <A href="firstwebpage.html" class="btn btn-primary">Go to the first page</A>
       </DIV>
    </DIV>
  </BODY>
</HTML>
```

4.12 Introduction to JavaScript

To ***program the behavior*** of our webpages, and make them more dynamic and more interactive, we can use JavaScript. Although we won't directly write a lot of JavaScript code in our book, we can use it to perform computations, create conditional and repetitive actions, change contents in pages and elements, apply CSS styling dynamically, validate forms, alert users, and (most importantly for this book) respond to various events (such as mouse move events, mouse click events, and so on).

NOTE: There are a lot of great details to learn about JavaScript, but we'll only focus on the parts that will be useful for our upcoming chapters. We only include this very brief introduction to JavaScript in order to have a more complete discussion about client-side technologies. To learn more about JavaScript, we invite the reader to look into [5].

VERY IMPORTANT: JavaScript **IS** case sensitive. So please be very careful.

Fig. 4.20 This figure is similar to Fig. 4.19, but the buttons at the bottom of the page are now centered

This leads to an interesting situation: we can combine HTML, CSS, and JavaScript into our webpages, but some languages (HTML, CSS) are not case sensitive, while others (JavaScript) are. So please be extra careful. We'll see (below) examples on how to use all three languages together.

To add JavaScript to a webpage, we can use external .js files, or we can embed JavaScript directly inside our webpages, using the `<SCRIPT>` element. For simplicity, we'll first embed them inside our webpages, then we'll see how to link external .js files into a webpage.

One can embed the `<SCRIPT>` element inside the `<HEAD>` element. For better performance reasons, many sources recommend adding the `<SCRIPT>` element right before the end tag of the `<BODY>` element.

4.13 JavaScript Statements

Just like in other languages (such as C# and Java), a JavaScript consists of a list of instructions called **statements**. These statements are run by the browser and optionally end with a semicolon.

To define blocks of code (statements that are meant to be run together), we make use of curly braces.

For example,

```
<SCRIPT>
    var count = 0;
    count = count + 1;
</SCRIPT>
```

JavaScript has **keywords**, which are reserved words that cannot be used as *identifiers* (for example, they cannot be used as *variable* names).

To declare variables, one can use the `var` keyword or the `let` keyword. There are very important differences between the two, but we'll skip them because we won't make much use of them in this book.

Lastly, to use comments, one can use any of the following (just like in C# or Java):

- Single-line comments, for example.
- Multi-line comments, for example.

For example,

```
<SCRIPT>
    var count = 0;  //this is a single line comment
    count = count + 1; /*this is a multi-line comment*/
</SCRIPT>
```

4.14 JavaScript Functions

We'll make some use of JavaScript function, so please look into them more carefully. A **function** is essentially a named and reusable block of code, designed to do a given task.

Here is an example of **function definition**:

```
function Max(num1, num2)
{
    let answer = num1;
    if(num2>num1)
    {
        answer = num2;
    }
    return answer;
}
```

To use this function, we call it by name as follows:

```
document.title = Max(4, 2022);
```

In the example above, please note the keyword **function** being used when defining a function. Most of this code should be very familiar to you (if you've taken an introductory course in C# or Java). Can you answer the following questions?

- What is the *name of the function* declared above?
- What are the *parameters* of the function above?
- Where do we *call/invoke* that function?
- What *arguments* did we pass to this function?
- What is the name of the *local variable* used in the code above?
- What does it mean for a function to *return a value*?

We'll learn more about `document.title` below (see the Document Object Model - DOM).

4.15 Add JavaScript to Our Webpages

In the *register.html* let's add the following HTML and JavaScript code, right after the `</FORM>` tag:

```
<BR>
<BR>
<BUTTON onclick="ToGreenBackground()">Green background</BUTTON>
<BUTTON onclick="ToRedBackground()">Red background</BUTTON>

<SCRIPT>
    function ToGreenBackground(){
        document.body.style.backgroundColor="GREEN";
    }
    function ToRedBackground(){
        document.body.style.backgroundColor="RED";
    }
</SCRIPT>
```

Fig. 4.21 The webpage has a blue background

Inside the <SCRIPT> element we defined two functions. One that would change the background color of the page's BODY into GREEN, and the other that would change it to RED.

Just defining two JavaScript methods won't do much, we also need to call these methods in order to carry on the task they were programmed to do. For this, we added two buttons and programmed them so that when you click on a button, one of the functions will be called.

When you refresh your webpage inside a browser, it should look like (Fig. 4.21).

Next, if you click on the *Green background* button, you should get (Fig. 4.22).

Lastly, by clicking on the *Red background* button, you should obtain (Fig. 4.23).

4.16 Introduction to the Document Object Model (DOM)

When a webpage is loaded into a browser, the browser creates an object, called the **Document Object Model**, for the webpage ([6]). Using this object (which has a logical tree structure), one can manipulate the webpage, for example,

- add/modify/remove HTML *elements* in a page;
- add/ modify/remove HTML *attributes* in a page;
- add/modify/remove *CSS styles* in a page;
- call JavaScript functions to react to various events in a page.

Above we've seen the following examples:

```
document.body.style.backgroundColor="RED";
```

Fig. 4.22 The webpage has a green background after the user clicks on the “Green background” button

Fig. 4.23 The webpage has a red background after the user clicks on the “Red background” button

This code would change the background color (which is styling/CSS) of the `<BODY>` element to `RED`.

Similarly, the example below would change the `TITLE` element to have the value 2022 (which is the value returned by `Max(4, 2022)`):

```
document.title = Max(4, 2022);
```

We can use the DOM object, document, to search for elements that have various characteristics. For example (there are many more than the ones shown below, see more in [7]),

- `document .body` will return the `<BODY>` element;
- `document .cookie` will return the webpage's cookie;
- `document .forms` will return a collection with all `<FORM>` elements;
- `document .head` will return the `<HEAD>` element;
- `document .images` will return a collection of all `<IMG>` elements;
- `document .links` will return a collection of all `<A>` elements that have an `HREF` attribute;
- `document .getElementById(id)` will find and return an element, using a given CSS `id` selector;
- `document .getElementsByClassName(name)` will find and return a collection of elements, using a CSS class `name`.

We can then use these in combination with the following:

- The `innerHTML` property, in order to modify the **content** of an HTML element.

  o Example:
  ```
  document.getElementById("myCSSid").innerHTML = "Paragraph contents changed!";
  ```

- The `.attribute` syntax, in order to modify the value of an **attribute** for an HTML element.

  o Example:
  ```
  document.getElementById("myCSSid").src = "OlympiaLogo.jpg";
  ```

- The `.style.property` syntax, in order to modify the **style** of an HTML element.

  o Example:
  ```
  document.getElementById("myCSSid").style.color = 'red';
  ```

4.17 Add Event Handlers

There are various **events** that JavaScript can capture and therefore we can program a **response** (called **event handler**) when they occur. Some examples of **events**:

- the user clicks on a button,
- the user enters some text in a text field,
- the user chooses some value from a dropdown list or menu,
- a webpage has finished loading its contents,
- … and many others.

The typical way to associate JavaScript response (event handler) with a specific event is as follows:

```
<element eventName="JavaScript code or function call">
```

There are several ways to associate events and handler; see [5] for more examples. Below, let's see an example that demonstrates some of these concepts. We'll see more as we go through this book.

4.18 An Example: Toggle Between Dark/Light Mode

In the *register.html* replace all the code between the </FORM> tag and </BODY> tag with the following code:

```
<BR>
<BR>
<BUTTON onclick=" ToggleLightAndDark()>Toggle light/dark mode</BUTTON>

<SCRIPT>
    function ToggleLightAndDark(){
        document.body.classList.toggle("my-dark-mode");
    }
</SCRIPT>
```

Also, add the following CSS code inside the <HEAD> element:

```
<STYLE>
    .my-dark-mode{
        background-color: black;
        color:white;
    }
</STYLE>
```

In the JavaScript code above, we have a <BUTTON> element, which displays as a button (not surprisingly).

Fig. 4.24 **(Left)** The webpage has a default blue background

To this element, we added onclick=" ToggleLightAndDark()". The effect of it is that now, each time you click on the button (the click event), the ToggleLightAndDark function (which we defined) will be called.

The function ToggleLightAndDark is defined to toggle between the current style of the <BODY> element and a CSS *class selector* which we defined as my-dark-mode. Lastly, the class selector my-dark-mode was defined to change the background color of the element (<BODY> in our example) to black, and text color to white.

Here is how it behaves (see Figs. 4.24 and 4.25).

We challenge you to improve this example and also make use of Bootstrap 5. Read more about this in [8].

4.19 The Back Button

Here is how one can add a button that will take you back to the previously visited webpage, using the browser's history:

```
<input type="button" value="GO BACK" onclick="history.back()"/>
```

We'll make use of it when we start developing our ASP .Net Core MVC application. What you should note in here is the following (see more [9]):

Fig. 4.25 **(Right)** The webpage has a black background after the user clicks on the "Toggle light/dark mode" button

- The `value` attribute is used to display the text on the button.
- The `onclick` attribute is used to specify what method to be called when the user clicks on the button. In particular, it has the same effect as calling `history.go(-1)` and it will move the browser to the previous page. If there is no previous page, then calling this method will do nothing.

If you prefer the `<BUTTON>` element, you can use the following instead:

```
<BUTTON onclick="history.back()">  GO BACK  </BUTTON>
```

4.20 External JavaScript

Lastly, let's move our JavaScript code into an external file and link that file to our webpage. For this, create a file with the extension .js (we used *myScript.js*) and move all contents of the `<SCRIPT>` element (not including the `<SCRIPT>` tags!) into this file. Then, replace the `<SCRIPT>` tags with the following:

```
<SCRIPT src="myScript.js"></SCRIPT>
```

That's it. Now, you can reuse this script in other/multiple webpages if you so desire.

4.21 More Introduction to Bootstrap

To learn Bootstrap in more depth, we recommend the following two sources: [10] and [11]. In here we would like to add a little more introduction to Bootstrap 5. First, you should note that Bootstrap 5 contains not only CSS but also JavaScript code (see below for more details).

Bootstrap 5 “is the most popular HTML, CSS, and JavaScript framework for creating responsive, mobile-first websites. […] is completely free to download and use!” ([10]). In particular, it allows us to quickly create responsive webpages without “reinventing the wheel”. By **responsive** we mean the layout automatically adapts/responds to the device’s layout. As an example, as we’ll see below, if the browser’s window is too narrow, the navbar collapses the menu options (to create a more user-friendly environment). We get this by making use of pre-built (i.e., developed into Bootstrap) CSS styles and JavaScript functionality for buttons, tables, navbars, and so on.

4.22 Ways to Include Bootstrap in Our Projects

There are multiple ways to include Bootstrap in our projects. One can:

- use various package managers (such as npm, yarn, **NuGet**, and so on),
- **download** and use it as your own .css and .js files, or
- (the way we’ll use it in this book) using a **Content Delivery Network** (CDN).

4.23 Some CDNs for Bootstrap 5

Content Delivery Networks (CDNs) are a set of servers located around the World that can hold copies of some content and deliver this content to users much faster (since these globally distributed servers may be closer to your customers than your web application’s host server). There are multiple CDNs that host the Bootstrap 5 files, and you can use either one of them. We’ll give below a couple of options.

Note: We include a `<LINK>` element for the .css file and a `<SCRIPT>` element for the .js file!

JSDELIVR.NET.

- ```
  <LINK href="https://cdn.jsdelivr.net/npm/bootstrap@5.2.3/dist/css/bootstrap.min.css" rel="stylesheet">
  ```
- ```
  <SCRIPT src="https://cdn.jsdelivr.net/npm/bootstrap@5.2.3/dist/js/bootstrap.bundle.min.js"></SCRIPT>
  ```

Cloudflare

- ```
  <LINK rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/twitter-bootstrap/5.2.3/css/bootstrap-grid.min.css" integrity="sha512-JQksK36WdRekVrvdxNyV3B0Q1huqbTkIQNbz1dlcFVgNynEMR10F8OSqOGdVppLUDIvsOejhr/W5L3G/b3J+8w==" crossorigin="anonymous" referrerpolicy="no-referrer" />
  ```
- ```
  <SCRIPT src="https://cdnjs.cloudflare.com/ajax/libs/twitter-bootstrap/5.2.3/js/bootstrap.min.js" integrity="sha512-1/RvZTcCDEUjY/CypiMz+iqqtaoQfAITmNSJY17Myp4Ms5mdxPS5UV7iOfdZoxcGhzFbOm6sntTKJppjvuhg4g==" crossorigin="anonymous" referrerpolicy="no-referrer"></SCRIPT>
  ```

4.24 View Bootstrap 5 Source Files

We've seen already above that you can actually look into and read the CSS source files for Bootstrap 5 files. Similarly, you can read the JavaScript source files. In a browser, open the *minified* version (which you should link in your webpages):

```
https://cdn.jsdelivr.net/npm/bootstrap@5.2.3/dist/js/bootstrap.bundle.min.js
```

Then, go to the non-minified (developer-friendly) version which you can use for debugging purposes or for studying it. For this remove the *.min* from the URL:

```
https://cdn.jsdelivr.net/npm/bootstrap@5.2.3/dist/js/bootstrap.bundle.js
```

4.25 Bootstrap 5 navbar

The last example we'll see in here (we'll see other examples in the upcoming chapters) is the **navbar** (navigation bar). A great resource for this is the following: [12]. In that resource, you'll find many more options.

Here is what we would like to get: menu options (links) to show up at the top of the page (Fig. 4.26).

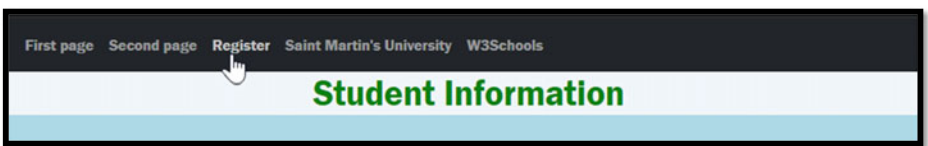


Fig. 4.26 The webpage displays a navigation menu at the top

As an added bonus, we will automatically get a responsive navbar. Namely, if the window is too narrow to display all menu options, these menu options will be stacked on top of each other, rather than horizontally aligned (Fig. 4.27).

Briefly: To create a **NAVBAR**, we use the <NAV> element. In it, we have an <UL> element, which will be the list of menu options. Each menu option is represented by a <LI> list item element whose content is an <A> element. To make use of Bootstrap 5, we also need to add various CSS classes (as seen below).

Let's create our NAVBAR; add the code below, right after the <BODY> tag (the beginning of the <BODY> content).

```
<NAV class="navbar navbar-expand-sm bg-dark navbar-dark">
    <DIV class="container ">
      <UL class="navbar-nav">
        <LI class="nav-item">
          <A class="nav-link" href="firstwebpage.html">First page</A>
        </LI>
        <LI class="nav-item">
            <A class="nav-link" href="secondwebpage.html">Second page</A>
        </LI>
        <LI class="nav-item">
          <A class="nav-link" href="register.html">Register</A>
        </LI>
        <LI class="nav-item">
          <A class="nav-link" href="https://www.stmartin.edu/">Saint Martin's University</A>
        </LI>
        <LI class="nav-item">
            <A class="nav-link" href="https://www.w3schools.com/">W3Schools</A>
          </LI>
      </UL>
    </DIV>
</NAV>
```

In here, we should see that we have a <NAV> element, which contains all other elements. This is our navbar. To get a responsive collapsible menu, we use the following CSS class: `navbar-expand-sm`. To choose a color for our navbar, we used the classes: `bg-dark navbar-dark`.

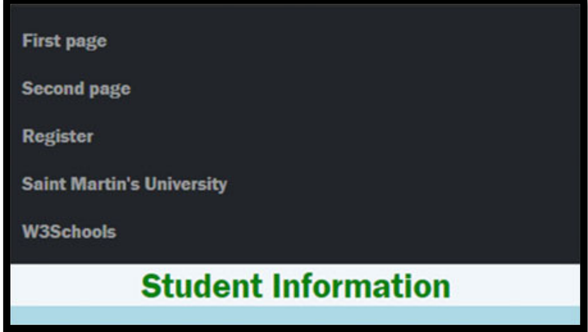


Fig. 4.27 The webpage displays the same navigation menu as seen in Fig. 4.26, but the menu options are now stacked vertically

Then, directly inside the <NAV>, we used a <DIV> element as a container for all other elements. In particular, we can choose between a CSS class of container (which provides a responsive *fixed width* container) and a CSS class of `container-fluid` (which provides a *full width* container, which spans the entire width of the viewport).

Then, each of the links (each an <A> element using the css class nav-link) are essentially list items (see the <LI> element) using the css class nav-item, in an unordered list (see the <UL> element) which is nested inside the <DIV> element mentioned above. For the <UL> element, we used the css class navbar-nav.

Note: Replacing the line

```
<DIV class="container">
```

with

```
<BUTTON class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-
target="#collapsibleNavbar">
    <SPAN class="navbar-toggler-icon"></SPAN>
</BUTTON>
<DIV class="collapse navbar-collapse" id="collapsibleNavbar">
```

you can get a collapsible menu (see Fig. 4.28) that looks like this (when the window width is narrow enough):

Then, clicking on the upper left button (with three horizontal lines on it), a menu opens (Fig. 4.29).

There is much more to say about responsive design, and about Bootstrap, but because of the focus of this book, we won't go into more detail.

To help you get a sense of what you can accomplish with Bootstrap, and give you some design ideas, we recommend you check out the following link for a set of themes build using this framework: [13].

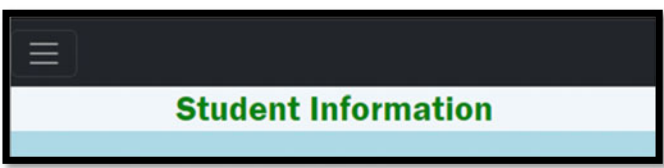


Fig. 4.28 Shows the same menu as seen in Fig. 4.27, but the menu is not collapsed (see the icon with three horizontal lines)

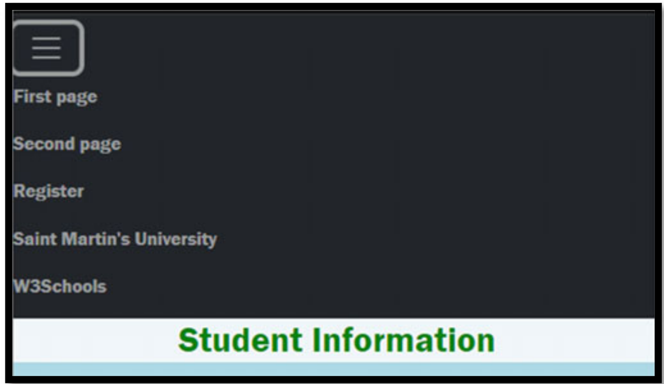


Fig. 4.29 Shows the same collapsed menu shown in Fig. 4.28, but it is now opened

5 Some C# Fundamentals

In this book, we will assume that you have some fundamental knowledge of an object-oriented programming language such as C#, Java, or C++. Below we'll introduce some concepts that we'll need throughout this book. We'll introduce other concepts later in the book, but this chapter should provide us with some C# knowledge baseline. Feel free to skip this chapter (or only read selected sections) if you are already proficient in C#.

5.1 Hello World in C# (Console Application)

Let's create the "Hello, World!" application using a *console application* in C#. This is the only console application we'll create in the entire book. Also, from here on we'll stop using Visual Studio Code and we'll exclusively use Visual Studio.

If you cannot install Visual Studio on your machine (for example, you are using a Linux distribution), then you should continue to use Visual Studio Code (but this edition of the book only focuses on Visual Studio).

For this, open the **Visual Studio** application. There, click on *Create a new project* button, then click on the *Console App* button. Then, in the next window, in the Configure your new project window, enter a project a name, and select a location, then click on the *Next* button.

At the next step, in the Additional Information window, make sure to choose .Net 6.0. For this demo, make sure to also check the *Do not use top-level statements.* We'll explain this later in this chapter, but for now make sure it is checked and then click *Next*.

You should get the following code in the *Program.cs* file (we'll explain the code below):

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024

R. A. Mezei, *Introduction to the Development of Web Applications Using ASP .Net (Core) MVC*, Synthesis Lectures on Computer Science,
https://doi.org/10.1007/978-3-031-30626-6_5

```
namespace HelloWorld
{
    internal class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello, World!");
        }
    }
}
```

To compile and run this code, you have two options:

- **Run without debugging** (press Ctrl + F5—or click on the dark green “play” button).
  - o Faster, but breakpoints will be omitted (we’ll explain breakpoints later in this chapter).
- **Run with debugging** (press F5—or click on the light green “play” button)
  - o May be a little slower, this option is better for debugging (breakpoints will not be omitted).

Let’s run this application *without debugging*. It should display “Hello, World!”.

```
Hello, World!
```

5.2 Top-Level Statements

If you create another project and follow the same steps as in the previous section, except that this time, in the Additional Information window, you leave *Do not use top-level statements* unchecked, you will get the following code in *Program.cs*:

```
// See https://aka.ms/new-console-template for more information
Console.WriteLine("Hello, World!");
```

If you run the application, you should get the same result as in the previous section:

```
Hello, World!
```

What is happening is as follows. If any one source file (in our case the file named *Program.cs*) contains statements outside of a namespace declaration, the compiler will wrap them inside a Main method, inside a class, and inside a namespace. See more about this in [1]. Our ASP .Net Core MVC applications that we'll create in the following chapters will make use of this.

"Starting in C# 9, you don't have to explicitly include a Main method in a console application project. Instead, you can use the top-level statements feature to minimize the code you have to write. In this case, the compiler generates a class and Main method entry point for the application" ([2]).

Important note: C# is case sensitive. That is `if` , `IF` , and `iF` are all considered different. So please be careful.

5.3 Namespaces, Using Directive, and Global Using Directive

5.3.1 Namespaces

The first building blocks for C# applications are *namespaces*. When we write code, we organize it into *namespaces*. **Namespaces** are containers of code. In them one can create other namespaces, classes, enumerations, and so on. To create a *namespace,* we use the **namespace** keyword, we give it a name, and then, within curly braces, we add its corresponding contents.

For example, to define our first namespace, named FirstNamespace we write.

```
namespace HelloWorld
{
      ... add code in here ...
}
```

Starting with C# version 10, the above code can also be written as follows:

```
namespace HelloWorld;

... add code in here ...
```

To learn more about this, check out [3].

5.3.2 Using Directives

Namespaces can be used as a great way to separate/isolate pieces of code, especially as a project becomes larger. In the example below, we created two namespaces. The class

created in the second namespace is unknown to the code in the first namespace (you will get a compiler error if you try to run this code):

```
namespace HelloWorld
{
    internal class Program
    {
        static void Main(string[] args)
        {
            MyClass obj = new MyClass();
            Console.WriteLine("Hello, World!");
        }
    }
}

namespace MySecondNamespace
{
    class MyClass
    {

    }
}
```

When you want to use code from different namespaces, you will have two options. Either use the **full class name** (that is <namespace name>.<class name>) as below:

```
namespace HelloWorld
{
    internal class Program
    {
        static void Main(string[] args)
        {
            MySecondNamespace.MyClass obj = new MySecondNamespace.MyClass();
            Console.WriteLine("Hello, World!");
        }
    }
}

namespace MySecondNamespace
{
    class MyClass
    {

    }
}
```

or use a **using** directive as shown below:

```
using MySecondNamespace;
namespace HelloWorld
{
    internal class Program
    {
        static void Main(string[] args)
        {
            MyClass obj = new MyClass();
            Console.WriteLine("Hello, World!");
        }
    }
}
namespace MySecondNamespace
{
    class MyClass
    {

    }
}
```

There is more to say about *namespaces* (for example, what happens if you have two classes with the same name declared in two *namespaces* and want to use both of them in a source file?) but since we won't use them in this book, we skipped them.

5.3.3 Implicit Using Directives

The Console class was created in a namespace called System, which is different than the current namespace. Yet, for the Console class, we did not need to use (but could have used) a directive such as

```
using System;
```

The reason for this is **implicit using directives**. Based on the type of project you're creating, a set of using directives is automatically added to your code by the C# compiler. In particular, for a *Console Application*, the compiler already added the following using directives:

```
using System;
using System.IO;
using System.Collections.Generic;
using System.Linq;
using System.Net.Http;
using System.Threading;
using System.Threading.Tasks;
```

Therefore, we did not need to add them. See more in [4].

5.3.4 Global Using Directives

The using directives only have effect in the file they are declared. If you want a using directive to import a namespace for your entire application, not only the file in which it was declared, use a **global using directive** instead. It is the same as before, but it uses the keyword global:

```
global using MySecondNamespace;
```

Now, you won't have to add a using MySecondNamespace; anywhere else in your project. See more in [4].

5.4 Comments

Comments are pieces of text embedded in our source code that will get ignored by the compiler. Similarly, to JavaScript, we can have single-line and multi-line comments.

- **Single-line comments** start with // and continue until the end of that line. Anything that we include in that line, following //, will be ignored by the compiler.
- **Multi-line comments** start with /* and continue, possibly on multiple lines, until the first */. Anything that we include in between /* and */ will be ignored by the compiler.

Example:

```
//this is an example of single-line comment

/* this is a
   multi-line
   comment
 */
Console.WriteLine("Hello, World!");
```

You should always document your code by writing meaningful comments.

On your own, you may want to also check out **documentation comments** in here [5].

5.5 Existing Data Types

For the remainder of this chapter, let's focus on the example that uses *top-level statements*, and build code in it. Here is the starting point (in the *Program.cs* file):

```
// See https://aka.ms/new-console-template for more information
Console.WriteLine("Hello, World!");
```

In C# all variables must have a declared type. C# is a **strongly typed** language, so the compiler will make sure (will enforce that) you only use variables in the context in which it makes sense for their declared type.

Some C# data types (see more in [6]) we'll make use of later in this book:

- **int**—used for whole numbers in the range −2,147,483,648 to 2,147,483,647;
- **double**—used for fractional numbers;
- **bool**—used to store Boolean (true/false) values;
- **string**—used to store strings (sequence of characters);
- **DateTime**—used to store dates and times.

For example, we let's create the following variables and assign them some initial values (delete all other codes from *Program.cs* and add the following):

```
int YearsOfExperience = 12;
double Salary = 100000; /*one can also use the Decimal type here */
bool IsVeteran = false;
string FullName = "Razvan A. Mezei";
DateTime HiringDate = DateTime.Parse("08/15/2018 8:00:00 AM");
DateTime CurrentDateTime = DateTime.Now; //displays the current date and time
```

Then, to display to the console the values of each variable, use the following Console.WriteLine statements:

```
Console.WriteLine(YearsOfExperience);
Console.WriteLine(Salary);
Console.WriteLine(IsVeteran);
Console.WriteLine(FullName);
Console.WriteLine(HiringDate);
Console.WriteLine(CurrentDateTime);
```

If you run the code, you should get something similar to

```
12
1000000
False
Razvan A. Mezei
8/15/2018 8:00:00 AM
12/10/2022 7:37:30 PM
```

Because C# is a strongly typed language, once you declare a variable, you can only assign it values of compatible types. For example, the following results in a compilation error because string is not a compatible type to be used for integers:

```
YearsOfExperience = "twelve";
```

A related topic here is to use the keyword **var** when you declare and initialize a variable, and let the compiler figure out what type to use for the declaration of that variable. This is called **type inference** or **implicit typing**. For example, one can replace.

```
int YearsOfExperience = 12;
```

with

```
var YearsOfExperience = 12;
```

Since the variable YearsOfExperience is initialized to an integer, the compiler can deduce that the type of it is an integer, so we can use var instead. This is probably not a huge win for this example, but as we deal with longer variable names or more complex types, the var keyword will quickly become convenient.

Important note: The following line of code will result in an error:

```
var YearsOfExperience;
```

even if you later add the (separate!) statement.

```
YearsOfExperience = 12;
```

This is because in the line var YearsOfExperience; the compiler does not have a value to use in order to decide what time should be assigned for this variable.

5.6 String Interpolation

We'll make use of the following several times throughout our book. To build a string that combines text and variable values, one can use **string interpolation**. For this, you need to add the $ right before the string, then inside the string use { } to embed expressions (for example, variables). Here is an example:

```
Console.WriteLine($"years of experience: {YearsOfExperience}");
Console.WriteLine($"full name: {FullName}, current salary: {Salary}");
Console.WriteLine($"hiring date: {HiringDate}, current date: {CurrentDateTime}");
```

This gave us the following:

```
years of experience: 12
full name: Razvan A. Mezei, current salary: 100000
hiring date: 8/15/208 8:00:00 AM, current date: 12/10/2022 7:43:15 PM
```

Inside { } one can also use format specifiers. For example, add :c to display the salary as currency, or :dd/MM/yyyy to specify a date format for the output:

```
Console.WriteLine($"years of experience: {YearsOfExperience}");
Console.WriteLine($"full name: {FullName}, current salary: {Salary:c}");
Console.WriteLine($"hiring date: {HiringDate}, current date: {CurrentDateTime:dd/MM/yyyy}");
```

We obtained:

```
years of experience: 12
full name: Razvan A. Mezei, current salary: $100,000.00
hiring date: 8/15/208 8:00:00 AM, current date: 12/10/2022 7:43:15 PM
```

5.7 Enumerations

Above we've seen some existing data types from C#. But we can also create custom types. Two ways to create custom types are *enumerations* and *classes*. In here we'll see *enumerations*.

To create an **enumerated type** (**enumeration**) we use the keyword enum, then use a name of our choice, and then in { } add the desired values. For example,

```
enum FacultyLevel {INSTRUCTOR, ASSISTANT, ASSOCIATE, FULLPROFESSOR};
```

Let's move this into its own source file. One way to do this is to select this code, right-click on it, and select the context menu option: *Quick Actions and Refactorings* .... Then click on *Move type to FacultyLevel.cs* and press the *enter* key:

In the Solution Explorer window, you should see a new file added to your project. If you double click on that new file (FacultyLevel.cs), you will see that your code has been moved in there.

Then, we can create variables of this newly created type/enumeration:

```
FacultyLevel CurrentLevel = FacultyLevel.ASSISTANT;
```

and use it, just like we used the other variables. For example,

```
Console.WriteLine($"Faculty level: {CurrentLevel}");
```

5.8 Classes

Another way to build custom types is by defining/creating *classes*. This is a very important topic, and we'll use it extensively in this book.

Let's consider the following "type" which currently does not exist in C#. We would like to work with a type named **Instructor**. Each instructor should have a name, a hiring date, and so on (we'll worry about this in the next section).

If you add the following line of code to your program: Instructor myself; you'll get a compilation error, similar to the one below:

```
CS0246: The type or namespace name 'Instructor' could not be found (are you
missing a using directive or an assembly reference?)
```

One way to create new types was by defining new *enumerations* (seen in the previous section). Another way is by defining new *classes*. To create a new class, we use the class keyword, followed by a chosen name, and then by {}.

If we add the following code, the above-mentioned error goes away (because now we have a type named Instructor):

```
class Instructor
{

}
```

We can add this code inside *Program.cs*, but let's be a little organized. Let's create a new file for it, in the same project. In the *Solution Explorer* window (if it is not already opened in Visual Studio, then go to View > Solution Explorer to open it), right-click on the project name and choose *Add* > *Class*… Then enter a name for the new class, (please enter Instructor.cs) and click on the *Add* button. You should see the newly created file in the *Solution Explorer*. It should contain the following lines of code:

```
namespace HelloWorld2
{
    internal class Instructor
    {
    }
}
```

Before we continue, make sure to delete the Instructor class definition from *Program.cs*. We do not need to have it defined in two files. Notice that now, the Instructor type is unknown in our *Program.cs* file. At the beginning of the *Program.cs* we need to

add the import directive for the namespace that contains the Instructor class definition. For the example in the screenshot above, that is,

```
using HelloWorld2;
```

we'll see more about classes in the next few sections below and use them extensively in this book.

5.9 References and Objects

Now that we defined a class called Instructor, we can create variables of that type. For example,

```
Instructor myself;
```

In this example, the variable myself is what we call a *reference* (more on it below).

To create a new **instance** of a *class* (also called **object**), we use the **new** keyword. For example,

```
new Instructor();
```

The line shown above will create a new instance of type Instructor. In order to be able to access it, we need some type of handle, a **reference**, which allows us to access it. In the example below:

```
Instructor myself = new Instructor();
```

myself is a reference that points to a newly created *object*.

Note: Above, we could have used the var keyword as follows:

```
var myself = new Instructor();
```

Note: Version 10 of C# (and beyond) also allows the code above to be written as.

```
Instructor myself = new();
```

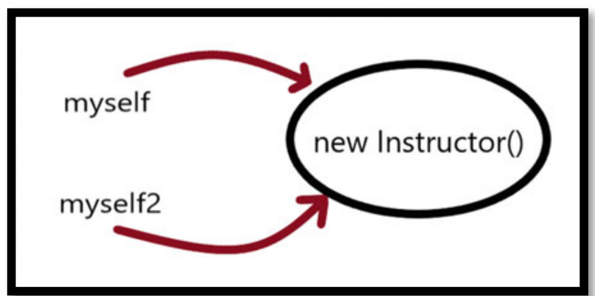


Fig. 5.1 Shows two references, myself and myself2, pointing to the same one object, created by new Instructor()

References are a little special. In the sense that they themselves do not hold the *object*, they just hold a *reference* to the *object*. In particular, in the code below:

```
Instructor myself = new Instructor();
Instructor myself2 = myself;
```

only one object is constructed, and we have two references (myself and myself2), both pointing/referring to the same object (see Fig. 5.1).

Important to remember:

- We use **classes** to define new types and new blueprints for **objects.**
- **Objects** are particular *instances* of **classes**.

For example, we can create a new *class* Car. Then, from it, we can build multiple **instances** (also called **objects**), for example,

```
Car myCar = new Car();
Car yourCar = new Car();
```

Above we created two instances of the Car class (so we have two **objects**) and we also have two **references** (myCar and yourCar) that we can use to reference/access those *objects*.

5.10 Instance Variables/Non-static Fields

So far, we created some new *classes*, but they don't do much. Let's add more to them. In here we'll add *fields*. We make use of **fields** to add **characteristics** to our *classes*. Fields (non-static fields) are also called **instance variables**.

Let's see some examples.

When you think of Instructors, what characteristics do they all have? For example, each instructor has a name, has a hiring date, and has a level, and some are tenured while others are not. When we define our Instructor class, we can declare these *instance variables/fields* in there. For example,

```
class Instructor
{
    //fields
    public string Name;
    public DateTime HiringDate;
    public FacultyLevel FacultyLevel; //confusing? let's discuss this
    public bool IsTenured;
}
```

Now that we declared these instance variables in the Instructor class, all instances of the Instructor class will have a HiringDate, a FacultyLevel, and whether or not they are tenured (IsTenured).

Note: In the line public FacultyLevel FacultyLevel; the second word is the type of the instance variable, while the third is the name of the instance variable. The compiler is able to know which one is which from the context it is used. We'll talk about the public *access modifier* below.

Let's see one more example. When you think of products that you buy online, what characteristics do they all have? For example, each product has a name, has a description, has a price, maybe a manufacturing date, a weight (needed for its shipping), and so on. Let's create a new class. This time, let's use the name Product (yes, the .cs part is optional, Visual Studio will automatically add this to the newly created file). Here is an example::

```
class Product
{
    public string ProductName;
    public string Description;
    public double Price;
    public DateTime ManufacturingDate;
    public double Weight;
}
```

Now we can use these *classes* to build more meaningful *objects*.

5.11 Dot Notation

To access the various characteristics of an object (to access its fields) we use the so-called **dot notation**. Namely, we use a dot after the reference name followed by the name of the instance variable we would like to access.

Note: We can use the dot notation with other members of a class, not just with fields (as seen below).

For example,

```
Instructor instructor1 = new Instructor();//a new instance is created
instructor1.Name = "Razvan A. Mezei"; //read it as the Name of instructor1 is ...
instructor1.HiringDate = DateTime.Parse("08/15/2018 8:00:00 AM");
instructor1.IsTenured = false;
instructor1.FacultyLevel = FacultyLevel.ASSISTANT;
```

It is important to understand that each instance has its own copy of instance variables. If we create a second instance, then the Name fields of instance1 and instance2 are not shared. This is why they are called **instance variables**.

```
Instructor instructor2 = new Instructor();//a new instance is created
instructor2.Name = "Suzanne Barton"; //read it as the Name of instructor2 is ...
instructor2.FacultyLevel = FacultyLevel.ASSOCIATE;
```

Let's make use of these instances:

```
Console.WriteLine($"Instructor {instructor1.Name} is a(n) {instructor1.FacultyLevel} Professor");
Console.WriteLine($"Instructor {instructor2.Name} is a(n) {instructor2.FacultyLevel} Professor");
```

We obtained (note how each instance (each Instructor in here) has their own instance fields (their own Name,…)):

```
Instructor Razvan A. Mezei is a(n) ASSISTANT Professor

Instructor Suzanne Barton is a(n) ASSOCIATE Professor
```

On your own, create one more object, this time an instance of the **Product** class created above. As soon as you type in the `.` after the instance name, a tool in Visual Studio, called *IntelliSense*, will help us choose one of the available fields (and several other options that will be explained later; hint: inheritance).

5.12 Methods

Above, we've seen how we can use *non-static fields* to store characteristics (or instance variables). To program behavior and actions that each instance can perform, we can make use of *methods*. Think of **methods** as a named set of statements that we can call when needed (in particular, we can easily reuse this set of statements).

Let's define a couple of methods in our Instructor class. We would like to be able to change the name of an instructor. To define such a *method*, we can use code similar to the one below (we give the entire class, to help you make sure your code is complete):

```
internal class Instructor //you can remove "internal" if you want
{
    //fields
    public string Name;
    public DateTime HiringDate;
    public FacultyLevel FacultyLevel; //confusing? let's discuss this
    public bool IsTenured;

    //methods
    public void ChangeName(string newName)
    {
        Name = newName;
    }

    public int YearsSinceHired()
    {
        return (int)(DateTime.Now.Subtract(HiringDate).Days / 365.25);
    }
}
```

Above, we defined two (*instance*) *methods*. One named ChangeName, and another named YearsSinceHired. You should note that these methods have access to the instance variables of the class, so these do not need to be passed as arguments/parameters for the method.

To make use of these *methods*, we can again use the *dot notation*. Since these are *methods*, we will not only use their names when we call them, but also must use parentheses, and if they have required *parameters*, we need to pass *arguments* for these *parameters*. Let's add this line in *Program.cs*.

```
instructor1.ChangeName("Alex Mezei");
Console.WriteLine($"Instructor {instructor1.Name} worked here for
{instructor1.YearsSinceHired()} years");
```

Then compile and run the application. You should get an output similar to

```
Instructor Razvan A. Mezei is a(n) ASSISTANT Professor
Instructor Suzanne Barton is a(n) ASSOCIATE Professor
Instructor Alex Mezei worked here for 4 years
```

5.13 The this Keyword

The **this** keyword inside a class represents the ***current instance***. In particular, the ChangeName method above could be rewritten as.

```
public void ChangeName(string newName)
{
    this.Name = newName;
}
```

In the code shown above, the keyword *this* is not necessary because from the given context (a method inside a class) it was clear that Name represents an instance variable (or a field/property).

The code shown below, however, contains a logical error. Can you spot it?

```
public void ChangeName(string Name)
{
    Name = Name;
}
```

Probably the intent is to use the value of the parameter Name (right side of =) to set the value of the instance variable Name (left side of =). But that's not what's happening. Inside the ChangeName method, the method's parameter Name will hide the instance field Name and hence the assignment just sets a value to itself without affecting the instance variable name. In order to disambiguate which one is which, you can use the this keyword to specify that the Name from the left side of the assignment operator (=) refers to the instance variable Name, not the parameter Name:

```
public void ChangeName(string Name)
{
    this.Name = Name;
}
```

5.14 Access Modifiers

Access modifiers allow us to control what parts of the application have access to our code. We can apply access modifiers to namespaces, classes, fields, methods, properties, and others. There are six *access modifiers* but the most used ones in our book are the following:

- **public**: the member that has this *access modifier* can be accessed by any other code.
- **private**: the member that has this *access modifier* can only be accessed by code in the same *class* (or *struct*).

To learn more about them (protected and internal in particular), we recommend the following [7, 8].

5.15 Properties

Properties are kind of a mix between *fields* and *methods*. They are very important, especially for upcoming chapters.

Let's first see the need for *properties*. Let's look at the following example:

```
class Product
{
    public double Price;
}
```

What parts of our code have access to the Price field? Since this field is public, any part of your program (inside or outside of the Product class) has access to it. Any part of your program can read it, or even modify it.

Challenge: What if you want to allow code outside the Product class to only read the Price field, but not change it? How can you create such a restriction? How can we provide such controlled access to our fields?

If you change the *access modifier* of the Price field to private, then your field is only accessible inside the Product class, and nowhere else. This only partially solves the challenge, but not completely.

One solution: One way to solve the above challenge (if you've used Java, you've probably seen this) is as follows:

- Make the field private:

```
private double Price;
```

- Then create public methods that allow us to read the field from the outside of the Product class (such a method is called a ***getter*** or an ***accessor***) and do not create any public method that allows us to change it:

```
public double getPrice()
{
    return Price;
}
```

This solves our challenge proposed above. Here is how the code put together looks like:

```
class Product
{
    private double Price;

    public double getPrice()
    {
        return Price;
    }
}
```

From outside on the Product class, we cannot change the price, but we can read it via the getPrice method (which is publicly accessible). Inside the Main method, code to read the price of a product would look something like.

```
Product laptop = new Product();
//Console.WriteLine(laptop.Price);//ERROR: Price is private
Console.WriteLine(laptop.getPrice());  //use the getPrice method instead
```

A similar discussion can be applied to the case where you want to allow users to modify a field, without being able to read the existing value. We could create a public method that allows us to change the field (such a method would be called a ***setter*** or ***mutator***). It would look something like.

```
public void setPrice(double newPrice)
{
    Price = newPrice;
}
```

Another/A better solution: C# has a more elegant way to solve this challenge (and many related ones), by making use of **Properties**. Here, we'll only give a simplified description of properties since we won't use their full potential, but we do encourage you to read more information on properties here [9]. We will replace the Price field declaration and the definition of the accessor with the following one line (that completely solves the above given challenge):

```
public double Price { get; private set; }
```

Let's test our code in Main. Notice how we use the dot notation on the property:

```
Product laptop = new Product();
//laptop.Price = 1099.99; ;//ERROR: Price setter is private
Console.WriteLine(laptop.Price);  //but the getter is public
```

From the code given above, we can see that we are able to read the value of Price, but not change it in Main.

If you want to allow Main to only change the value of Price but not read it, you will use the following property declaration instead:

```
public double Price { private get; set; }
```

Read the source mentioned above ([9]) to also learn about *computed properties*, *validation*, and many other capabilities that *properties* have. We will make extensive use of properties, but we will probably just use them in the following form:

```
public double Price { get; set; }
```

5.16 Constructors

Constructors are special methods that are used when creating new objects/instances. They look like regular *methods* but their name must match the name of the *class* in which they are defined, and they do not have a return type. If (and only if!) you define a class and do not include a constructor definition, one will automatically be created and added for you by the compiler. We typically use *constructors* to set the initial state/values for an *object*.

Constructors that have no parameters are called **default** constructors. In the line below, when we create a new *object* of type Product, we are calling the *default constructor* defined in the Product class:

```
Product laptop = new Product();
```

Since we did not define an explicit default constructor, one was added for us automatically. Check out the output of

```
Console.WriteLine(laptop.Price); //it displays 0 (default value for double)
```

Now, let's explicitly define a *default constructor* in the Product class:

```
public Product()
{
    Price = 10.99;
    ManufacturingDate = DateTime.Now;
}
```

Run again your code and see how the output changes, because we start with a different initial value for Price:

```
Console.WriteLine(laptop.Price); //it now displays 10.99 (default constructor set this value)
```

5.17 Method Overloading

We can have multiple *methods* with the same name (as long as their parameters have different types or a different number of parameters)—this is called **method overloading.** Similarly, we can have multiple *constructors*. Constructors that have parameters are called **non-default constructors.**

Let's see an example—add a second constructor to the Product class.

```
public Product(double initialPrice)
{
    Price = initialPrice;
}
```

Now, in Main, let's test these constructors:

```
Product desktop = new Product(899.67);//it calls the non-default constructor
Console.WriteLine(desktop.Price); //it displays 899.67

Product laptop = new Product();//it calls the default constructor
Console.WriteLine(laptop.Price); //it displays 10.99
```

5.18 Conditionals, Loops, and Lists

We will assume that you understand and can work with if statements and various forms of *loops* (for, while, foreach). Below we will give a few brief examples but if you need more resources, we recommend the following [10, 11].

Let's write an example where we create a *list* of Instructors. Then, we check if the list is empty. If it is empty, we display "There are no instructors in the list!". Otherwise, we display the name of each instructor.

At the end of the *Program.cs* file add the following code:

```
//let's create a list of instructors
List<Instructor> CSDepartment= new List<Instructor>();
CSDepartment.Add(new Instructor { Name = "Damian Rolfson", FacultyLevel =
FacultyLevel.ASSISTANT, IsTenured = false });
CSDepartment.Add(new Instructor { Name = "Estell Gottlieb", FacultyLevel =
FacultyLevel.ASSOCIATE, IsTenured = true });
CSDepartment.Add(new Instructor { Name = "Linnea Koelpin", FacultyLevel =
FacultyLevel.FULLPROFESSOR, IsTenured = true });
CSDepartment.Add(new Instructor { Name = "Nannie Little", FacultyLevel =
FacultyLevel.FULLPROFESSOR, IsTenured = false });
if(CSDepartment.Count == 0) //check if the list is empty
{   //for an empty list display a messages
    Console.WriteLine("There are no instructors in the list!");
}
else
{   //for a non-empty list display the name of each instructor
    for(int i=0; i< CSDepartment.Count; i++)
    {
        Console.WriteLine(CSDepartment[i].Name);
    }
}
```

Note: In class we typically ask students to help us provide some sample data. This is more fun this way and it provides another opportunity for students' engagement.

Running this code, we get the following displayed:

```
Damian Rolfson

Estell Gottlieb

Linnea Koelpin

Nannie Little
```

Let's now rewrite the above code so we make use of the var keyword and use foreach instead of the for loop. We'll get the same output.

```
//let's create a list of instructors
var CSDepartment= new List<Instructor>();
CSDepartment.Add(new Instructor { Name = "Damian Rolfson", FacultyLevel =
FacultyLevel.ASSISTANT, IsTenured = false });
CSDepartment.Add(new Instructor { Name = "Estell Gottlieb", FacultyLevel =
FacultyLevel.ASSOCIATE, IsTenured = true });
CSDepartment.Add(new Instructor { Name = "Linnea Koelpin", FacultyLevel =
FacultyLevel.FULLPROFESSOR, IsTenured = true });
CSDepartment.Add(new Instructor { Name = "Nannie Little", FacultyLevel =
FacultyLevel.FULLPROFESSOR, IsTenured = false });

if(CSDepartment.Count == 0) //check if the list is empty
{   //for an empty list display a message
    Console.WriteLine("There are no instructors in the list!");
}
else
{   //for a non-empty list display the name of each instructor
    foreach(var instructor in CSDepartment)
    {
        Console.WriteLine(instructor.Name);
    }
}
```

5.19 Collections and Generic Collections

This section is not meant to teach you *generic programming*. Instead, we want to make you aware of it. Generic programming, in particular *generic collections,* are used quite a bit in .Net applications. We will also make use of them in our book.

Collections are classes that allow you to group and manage multiple related objects. Some examples of collections are *array lists*, *linked lists*, *stacks*, *queues*, and so on.

There are *collection classes* developed in the System.Collections namespace that store everything as instances of class Object. Example of classes defined in here: ArrayList, (there are no *linked lists*!), Stack, Queue, and others (see more in [12]). You should not use them in new development, and consider them obsolete. Better choices are the following.

There are *collection classes* developed in the System.Collections.Generic namespace—these are generic classes, and users of these classes get to specify what types of objects they want to store in these collections. Examples of classes defined in here: List<T>, LinkedList<T>, Stack<T>, Queue<T>, and others (see more in [13]).

In the previous section, we made use of such generic class List.

```
//let's create a list of instructors
List<Instructor> CSDepartment= new List<Instructor>();
```

The class name is List, and in < > we specify the type of elements we want to store in our collection/list. In the example above, we used Instructor.

Let's see another example. If we want to create a generic *array list* of whole numbers, we can use code similar to the following:

```
List<int> grades = new List<int>(); //create a list of integers
```

To add elements to this list, we can use the `Add` method:

```
grades.Add(100);
grades.Add(90);
grades.Add(98);
grades.Add(65);
```

5.20 Inheritance

Inheritance is a very important topic; we'll make use of it throughout this book. We'll only cover here what we need to know for this book, but please consider learning more about it on your own. Here is a resource we recommend [14]:

Inheritance is the process of creating new classes by extending existing ones. The new class is called a **child class** or **derived class** and the existing/original class is called the **parent class** or **base class**. Below we give a quick example. For simplicity (and demonstration purposes), we'll overuse the public access modifier, but in a real context, you should question if that is the right access modifier to use.

For our example, let's create a very simple class called User. For it, let's create a new file, called *User.cs* and put the code in this file. What characteristics does each *User* have? What can each *User* do? The User class will be our *base class*.

A simple implementation for this class would be the following:

```
class User
{
    //properties
    public string UserName { get; set; }
    public string Password { get; set; }

    //actions
    public void Login()
    {
        throw new NotImplementedException();
    }

    public void Logout()
    {
        throw new NotImplementedException();
    }
}
```

Supposes now that your application needs to create more specialized *Users*. For example, assume that we need

- Professors (these are *Users*, in the sense they need to be able to login/logout, but also have characteristics such as whether or not they are tenured, maybe a hiring date) and
- Students (these are also *Users*, they too need to be able to login/logout, but also have characteristics such as major and admission date).

How do we create these Professor and Student classes?

Let's create a new file and in it a class named Professor. Since a Professor ***is a*** User, we will make use of inheritance. Note the use of : to specify *inheritance*. Here is a simplified example::

```
class Professor : User
{
    //characteristics + those inherited!
    public bool IsTenured { get; set; }
    public DateTime HiringDate { get; set; }
}
```

Note: All *fields*, *properties*, and *methods* from the *base* class will be *inherited* in the *derived* class. Whether or not the *derived* class has access to all of them depends on the *access modifiers* being used. In particular, a Professor has a UserName, a Password, and can Login and Logout:

```
Professor prof = new Professor();
prof.UserName = "rmezei";
prof.Password = "Password123!";
prof.Login();
prof.Logout();
prof.IsTenured= true;
```

As you type prof. see what options IntelliSense shows you.

Note: In the screenshot above, you may notice some methods that we did not include in our code above. In particular, note the methods GetHashCode, GetType, and ToString. They are the result of *inheritance* too. When you create a new class, if you do not specify any *base class* for it, the C# compiler will automatically make your new class a *derived class* of the Object class. Because of it, methods defined in the Object class will get inherited by your new class. In the example above, the methods GetHashCode, GetType, and ToString are inherited by the User class. Since Professor inherits from User class, Professor will inherit all properties, fields, and methods from User, including the methods GetHashCode, GetType, and ToString.

Next, we can once again use *inheritance* to create a Student class that has a UserName, a Password and can Login:

```
class Student : User
{
    public string Major { get; set; }
    public DateTime AddmissionDate { get; set; }
}
```

Then, in the *Program.cs* file we could use code such as the one below:

```
Student st = new Student();      //creates a new instance of Student
st.UserName = "razvan.mezei";    //property inherited from User
st.Password= "Password124!";     //property inherited from User
st.AddmissionDate = DateTime.Now;//property defined in the Student class
st.Login();
st.Logout();
```

5.21 The base Keyword and the Constructors

Something important happens with the constructors. To see this, let's add *default constructors* in both the User class and in the Professor class:

```
public Professor()//default constructor
{
    Console.WriteLine("Hello from the Professor default constructor");
}
```

And

```
public User()//default constructor
{
    Console.WriteLine("Hello from the User default constructor");
}
```

To double-check your code, your Professor class should now look similar to (Fig. 5.2).

Then, in *Program.cs*, remove all codes (except for the using *directive*) and add the following:

```
Professor prof = new Professor();
```

Then, run your program. The output should be similar to

```
Hello from the User default constructor
Hello from the Professor default constructor
```

What you should see is that the *constructor* for our Professor class, automatically called the *constructor* for the User class (which is the *base* class for our Professor).

```
class Professor : User
{
    //characteristics + those inherited!
    public bool IsTenured { get; set; }
    public DateTime HiringDate { get; set; }


    public Professor() //default constructor
    {
        Console.WriteLine("Hello from the Professor default constructor");
    }
}
```

Fig. 5.2 Shows the Professor class that extends the User class. It contains two properties, IsTenured and HiringDate, and a default constructor

IMPORTANT: In general, when we create *instances* of *derived classes*, the *constructors* for our *derived class* will always call the *default constructor* for the *base class* before it runs the code in the *constructor* (of our *derived class*).

Now, let's add non-default constructors to each class and see what happens in this case. Add the following non-default constructors into their respective classes:

```
public User(int num) //non-default constructor
{
    Console.WriteLine("Hello from the User non-default constructor");
}
```

and

```
public Professor(int num) //non-default constructor
{
    Console.WriteLine("Hello from the Professor non-default constructor");
}
```

Now, change the line in *Program.cs* so it looks similar to the line below (essentially calling the *non-default constructor* from the Professor class):

```
Professor prof = new Professor(2023);
```

Run the code and check out the results. You should obtain the following:

```
Hello from the User default constructor

Hello from the Professor non-default constructor
```

You should notice that the *non-default constructor* from the Professor class was called. This should not be a surprise since we passed a value to the constructor when we created the new instance of Professor class. But the important part to observe is that this constructor in turn called the *default constructor* from the base class (User class in here).

That means, all constructors, by default, will call the base constructor from the base class. We can change that behavior by explicitly calling the base constructor of our choice (default or non-default). This is where we'll make use of the base keyword.

Currently our *non-default* Professor constructor looks like

```
public Professor(int num) //non-default constructor
{
    Console.WriteLine("Hello from the Professor non-default constructor");
}
```

This is equivalent to

```
public Professor(int num) : base()  //non-default constructor
{
    Console.WriteLine("Hello from the Professor non-default constructor");
}
```

which is why the constructor calls the default constructor from the base class.

If we want to call the non-default constructor from the base class, we can pass parameters to the base() call, as follows:

```
public Professor(int num) : base(num*num)  //non-default constructor
{
    Console.WriteLine("Hello from the Professor non-default constructor");
}
```

Now, if we run again the program, we'll see that our non-default constructor from Professor class called the non-default constructor from its base class (the User class).

```
Hello from the User non-default constructor

Hello from the Professor non-default constructor
```

To recap, all constructors (default or non-default) from a class, by default, call the base class's default constructor. If we want a constructor to call a non-default constructor from the base class, we add the call to base() and pass parameters needed for that base non-default class constructor.

Methods do not have the same cascading effect as *constructors*, but we can build one using the base keyword again. We'll skip this part since we won't use it in our book.

5.22 Interfaces

Although we'll make extensive use of *interfaces*, in this book we will only define very few *interfaces*. As such we'll only briefly introduce them in here.

IMPORTANT: As a convention, interface names always start with a capital I (see examples below).

5.22.1 Some Motivation

Let's start with an example of *interface* that is included in the System library, namely IComparable.

If we want to sort *integers*, we can do so as long as we know how to compare each pair of numbers. Similarly, if we want to compare instances of let's say Professor class, again we can do so, as long as we have a way to compare every two professors. The essence of what we are trying to suggest here is that sometimes we want to expect similar functionalities from unrelated classes. If they are not related, *inheritance* is probably not the way to go (you should not use *inheritance* for unrelated classes). In this case, the *interfaces* will be the appropriate choice.

In particular, the *generic class* List, which can store elements of any given type, can also sort them, as long as the type used in the List implements the IComparable interface.

5.22.2 How to Define an Interface

To *define* an **interface**, we use the keyword interface, we choose a name (which by convention should start with I), and then, inside a block (use {}), we include zero or more declarations for methods, properties, (and starting with C# 8.0) fields, and other members.

Here is an example. In the *Solution Explorer* window, right-click on the project's name, select *Add* > *New Item …*, then click on the option *Interface* and enter a name, say IPrintable, then click on the *Add* button.

We can now *define* our *interface*. We'll define a simple one that contains only one method declaration:

```
interface IPrintable
{
    void PrintToConsole();
}
```

Important to notice: We did not include an *access modifier* (this is explained below).

5.22.3 How to Implement an Interface

Next, let us *implement* this interface. For a class to **implement** an *interface*, we need to.

- declare this by using the : (just like inheritance, which is why it's important to use the I in the interface name!);
- implement each member from the interface in our class.

Let's see this by example. Let's modify the Professor class so it now implements the IPrintable *interface*.

Since we already have an *inheritance* declaration (: is already in there) we only need to add a comma after User and add the name of the interface we intend to implement:

```
class Professor : User, IPrintable
{
    //...
}
```

Then, to implement the specified interface(s), we need to define all members from the interface declaration. In particular, we'll need to define a void PrintToConsole(); method.

Important to note: All members from the interface must be added as public members in the class (regardless of whether or not the interface uses the public access modifier when it declares those members). So, we'll need to define a public void PrintToConsole(); method.

What you put in this method it's up to you, the *interface* just cares that you have a definition for each member of the interface. Let's add the following method definition in the Professor class:

```
public void PrintToConsole()
{
    Console.WriteLine($"User: {UserName}, Hiring date: {HiringDate}");
}
```

Note: One class can only *inherit* from one *base class*, but it can *implement* multiple *interfaces*.

Important note: One can have multiple completely unrelated classes, all implementing the same interface. How can this be useful? That's what we'll see below.

5.23 How to Use an Interface

Let's start with the following example. If we define the method below:

```
void HappyMethod(User usr)
{
    //to do...
}
```

what can type of parameters can you pass to this method? The answer is you can pass in any instance of the User class, or any class derived from User (in particular, you can pass in an instance of a Professor).

Can you pass in an integer? Can you run HappyMethod(2024)? The answer is no, because 2024 is not an instance of User (or any derived class from User).

Interfaces provide us with more flexibility. Let's create a method, which uses an *interface* as a parameter type. For example, add the following to the end of the Program.cs file:

```
void HappyMethod(IPrintable obj)
{
    obj.PrintToConsole();
}
```

What can you pass to such a method? The answer is an instance of any class (or derived class) that implements the IPrintable interface. In particular, if you have two unrelated classes, say Professor and Classroom, both implementing the IPrintable *interface,* then you could pass an instance of either one to the HappyMethod. That's quite some flexibility.

See more about interfaces in here [15].

5.24 Lambda Expressions

This is a very brief introduction to lambda expressions. They are very convenient when we want to pass certain information to methods. See more in [16].

Lambda expressions are nice short ways to create so-called *anonymous functions. Lambda expressions* make use of the **lambda operator** =>(read it as: "goes to" or "becomes") and have two forms:

- **expression lambda**: (list of params) =>expression

  o example: `(a, b) => a + b;`

- **statement lambda**: (list of params) =>{ list of statements}

  o example: `(a, b) => {int c = a + b; return c;};`
  o example: `(a, b) => {return a+b;};`

If we have exactly one parameter, then the parentheses around it are not required. The following examples are equivalent:

- example: (a) => a * a ;
- example: a => a * a ;

Let's see some practical examples. Earlier, we created the following list:

```
var CSDepartment = new List<Instructor>();
CSDepartment.Add(new Instructor { Name = "Damian Rolfson", FacultyLevel = FacultyLevel.ASSISTANT, IsTenured = false });
CSDepartment.Add(new Instructor { Name = "Estell Gottlieb", FacultyLevel = FacultyLevel.ASSOCIATE, IsTenured = true });
CSDepartment.Add(new Instructor { Name = "Linnea Koelpin", FacultyLevel = FacultyLevel.FULLPROFESSOR, IsTenured = true });
CSDepartment.Add(new Instructor { Name = "Nannie Little", FacultyLevel = FacultyLevel.FULLPROFESSOR, IsTenured = false });
```

To display this list, we can use the following code:

```
foreach(var instr in CSDepartment)
{
    Console.WriteLine($"Name: {instr.Name}, Level: {instr.FacultyLevel}, Tenured: {instr.IsTenured} ");
}
```

And obtain

```
Name: Damian Rolfson, Level: ASSISTANT, Tenured: False
Name: Estell Gottlieb, Level: ASSOCIATE, Tenured: True
Name: Linnea Koelpin, Level: FULLPROFESSOR, Tenured: True
Name: Nannie Little, Level: FULLPROFESSOR, Tenured: False
```

which is the order in which we added each instructor into our list.

What if we would like to sort this list? Before the foreach statement add the following statement:

```
CSDepartment.Sort();
```

Run again your code. Did it work? The output should contain a rather error message, including.

```
Unhandled exception. System.InvalidOperationException: Failed to compare two
elements in the array.
```

What this means is that the Sort method did not know how to compare two instances of Instructor.

We now have some solutions. One is to implement the IComparable interface inside the Instructor class.

Another, which we'll see below, is to provide a lambda expression to the Sort method, an expression that is essentially a comparison method that can be used by Sort when sorting our list.

Let's sort by name. Replace the CSDepartment.Sort(); with.

```
CSDepartment.Sort((instr1, instr2) => String.Compare(instr1.Name, instr2.Name) );
```

Above note that we provided the Sort method a lambda expression, an anonymous method that can be used to compare every two instructors. Running the code above, you should now get the list sorted by Name:

```
Name: Damian Rolfson, Level: ASSISTANT, Tenured: False
Name: Estell Gottlieb, Level: ASSOCIATE, Tenured: True
Name: Linnea Koelpin, Level: FULLPROFESSOR, Tenured: True
Name: Nannie Little, Level: FULLPROFESSOR, Tenured: False
```

Similarly, we can sort by Tenure. In this example, we will use the CompareTo method that is included for Boolean values. Replace the statement above with.

```
CSDepartment.Sort((instr1, instr2) => instr1.IsTenured.CompareTo(instr2.IsTenured) );
```

Running the code again, we now get.

```
Name: Damian Rolfson, Level: ASSISTANT, Tenured: False
Name: Nannie Little, Level: FULLPROFESSOR, Tenured: False
Name: Estell Gottlieb, Level: ASSOCIATE, Tenured: True
Name: Linnea Koelpin, Level: FULLPROFESSOR, Tenured: True
```

See how flexible the Sort method was? Allowing us to use lambda expressions, we were able to quickly specify how to sort our list. Above we've seen how to sort the list by name, or by tenure.

5.25 LINQ

This is another very brief introduction. To read more about LINQ, we recommend the following [17].

LINQ stands for **Language-Integrated Query** and it provides "integration of query capabilities directly into the C# language". LINQ works very nicely with *lambda expressions*.

Here is one example. On a List instance, we can use Where for filtering. Replace the foreach statement above with:

```
foreach(var instr in CSDepartment.Where(inst => inst.Name.Contains("nn")))
{
    Console.WriteLine($"Name: {instr.Name}, Level: {instr.FacultyLevel}, Tenured: {instr.IsTenured} ");
}
```

We were able to narrow down the list to only elements that match a specific condition (given as a lambda expression):

```
Name: Nannie Little, Level: FULLPROFESSOR, Tenured: False
Name: Linnea Koelpin, Level: FULLPROFESSOR, Tenured: True
```

We can then use another operation on the results of `Where`. One could, for example, use another `Where`.

```
foreach(var instr in CSDepartment.Where(inst => inst.Name.Contains("nn")).Where(inst => inst.Name.Contains("tt")))
{
    Console.WriteLine($"Name: {instr.Name}, Level: {instr.FacultyLevel}, Tenured: {instr.IsTenured} ");
}
```

And obtain:

```
Name: Nannie Little, Level: FULLPROFESSOR, Tenured: False
```

One last example:

```
foreach(var instr in CSDepartment.Where(inst => inst.Name.Contains("nn")).OrderBy(inst =>
inst.IsTenured).Reverse())
{
    Console.WriteLine($"Name: {instr.Name}, Level: {instr.FacultyLevel}, Tenured: {instr.IsTenured} ");
}
```

will yield:

```
Name: Linnea Koelpin, Level: FULLPROFESSOR, Tenured: True
Name: Nannie Little, Level: FULLPROFESSOR, Tenured: False
```

5.26 Working with null Values

We mentioned earlier that C# is a strongly typed language. That means that if a variable is, let's say, declared as an int, then you can only assign compatible values to it.

In particular, if you declare number as an int, then you cannot assign a string value or even a null value. You would get a compiling error:

```
int number;
number = 10; //OK
number = "two"; //error - wrong type
number = null; //error - wrong type
```

When dealing with web applications, it is possible the user may fill out all form fields. How would you treat that?

One is to use *nullable* types. "A **nullable value** type T? represents all values of its underlying value type T and an additional null value" (see more in [18]).

If you put a ? next to the int type, you obtain int? which is a *nullable* type, namely it represents all integers and the null value. In the previous example, we now get

```
int? number;
number = 10; //OK
number = "two"; //error - still wrong type
number = null; //OK now
```

Another example: bool represents the values true and false. Therefore, bool? represents the values: true, false, null.

Important: When you work with *nullable types* you should check for null values.

There is a lot more to learn about this topic, but we'll skip the rest. We recommend you to also check out the following [19]. Also, we won't use the following but we want to challenge you to find out what is the difference between each of the following:

```
int[] values;

int?[] values;

int[]? values;

int?[]? values;
```

5.27 Solution Files .sln

To reopen a Visual Studio project (in particular an ASP .Net Core MVC project), you should locate and double click on the *.sln* file. This is typically in the same directory as the project, or one level up.

We have seen (many times) learners trying to open a project by opening the .cs (the C#) file, which may (typically) also open in Visual Studio. But once the file opens, there is no compile button. Instead you should open the .sln file (a text-based file). See more about this file here [20].

5.28 Other Resources for Learning C#

This chapter is not meant to provide a comprehensive C# tutorial. For that, we recommend the following:

- W3School's *C# tutorial*: see [10].
- The Microsoft's *C# documentation*: see [11] (our main C# source).

Check for updates

6 Middleware, Services, Intro to Dependency Injection

We are finally ready to start working on ASP .Net web applications.

Important notes:

- This chapter applies to ASP .Net Core web applications, regardless of whether or not they use the MVC pattern.
- We'll start our project in this chapter. Then, until the end of the book, we'll add to it a little more in every chapter.

6.1 What Are ASP .Net (Core) MVC Web Applications?

First, let's cover some background knowledge. In the previous chapter, we saw **Console Applications**, where we interacted with our application using a **console window**:

```
Hello, World!
```

In here (and in all subsequent chapters) we'll see **Web Applications**—instead of a *console window*, we'll use a *web browser* to interact with our application (see Fig. 6.1).

When the user clicks on a button or link in a browser, a request is typically being sent to a server. That request follows a specific format/protocol (which we won't cover in this book) and it is called an **HTTP Request**. **HTTP** stands for **Hypertext Transfer Protocol**.

ASP .Net is a *web framework* created by Microsoft that we can use to create web applications and services. **ASP** stands for **Active Server Pages**.

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024

R. A. Mezei, *Introduction to the Development of Web Applications Using ASP .Net (Core) MVC*, Synthesis Lectures on Computer Science,
https://doi.org/10.1007/978-3-031-30626-6_6

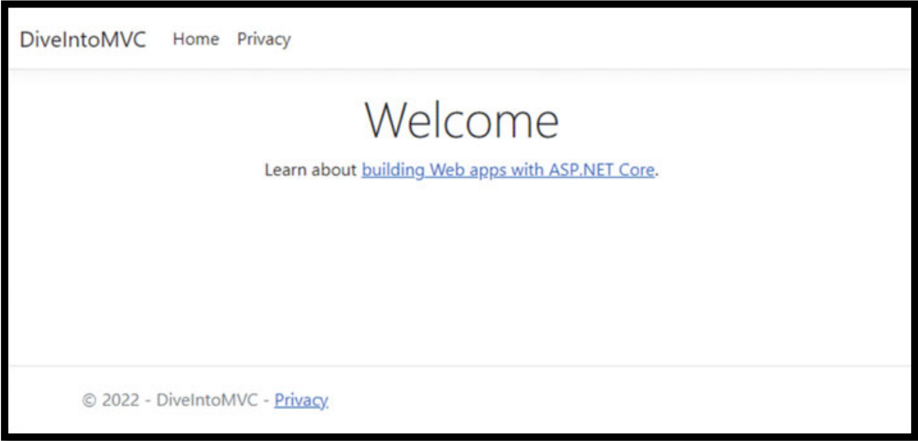


Fig. 6.1 Shows the welcome page of our web application in a web browser

There was a time when we had to choose between

- using ASP .Net/ASP .Net **Framework** (focused on Windows platforms) and
- using ASP .Net **Core** (intended to be cross-platform, but initially it wasn't as comprehensive as the other option).

The current version of .Net 6 is a *unified development platform* and now (for this version) **.Net** and **.Net Core** can be used interchangeably. With **.Net 6.0** (and beyond) you may sometimes see the word **Core** included (for example, ASP .Net Core) to emphasize this framework is cross-platform, but there is no need for it anymore (so for .Net 6.0 and beyond, ASP .Net and ASP .Net Core are equivalent names). You may want to read more about this in [46].

Now you know what ASP .Net (Core) means. What about the MVC? **Model-View-Controller** (**MVC**) is an architectural pattern, one that we'll introduce below and use for all subsequent chapters of this book.

In this book, we'll cover the ASP .Net (Core) MVC, but there are other frameworks that also use this pattern, for example, Spring MVC (which is a Java framework for building web applications).

6.2 An Introduction to the MVC Pattern

In here, we'll introduce the **Model-View-Controller** (**MVC**) as an architectural pattern. We hope that it makes some sense, but we'll see each one of the (Models, Views, and Controllers) in more depth in the upcoming chapters. So please make some sense of these, but do not panic if you do not have a complete grasp of everything just yet. For more information, check out [47].

One of the greatest values we get from using the MVC pattern is **separation of concerns**. The MVC pattern separates a web application into three components: the models, the views, and the controllers (see Fig. 6.2).

The **models** are the classes that represent the various types of objects managed by the web application. These objects represent the **state of the application**.

- For example (think about a web application such as Amazon, Canvas, Moodle, …), we could have the following model classes: Course, User, Student, Instructor, Administrator, Product, Seller, Buyer, and so on.

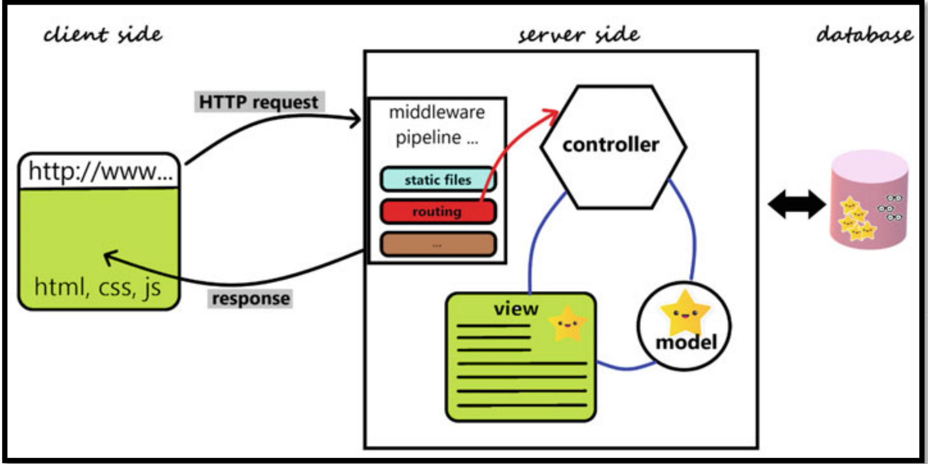


Fig. 6.2 Shows the main components of an MVC web application. In particular, the client side uses a browser (HTML, CSS, and JavaScript), then on the server side we have the middleware pipeline, controllers, models, and views, and lastly, the data may be stored in a database

- These are the typical C# classes we've seen in the previous chapter (plus other things we'll see later—these files will have extension .cs).

The **views** will make up the **user interface**. We'll present content to users via views (more accurately, we'll use views to build webpages that ultimately will get displayed in a user's browser). We'll see more about this later.

- For example (think about a web application such as Amazon, Canvas, Moodle, …), we can have a view that will be used to display a list of all courses taken by a student, or a list of all laptops available to purchase. We could use another view to build a page that allows our users to change their password, and yet another view to add a new laptop to sell it online.
- These are files that will combine HTML and CSS with C# (these files will have extension .cshtml).

The **controllers** will handle the **user interaction**. We'll give some examples below, but they will be better understood once we start using them in our project.

- For example (think about a web application such as Amazon, Canvas, Moodle, …). What happens when you click on a button? Or click on a link? Or load the first welcome page? In each of these, your requests will (eventually) be sent to a **Controller** (more specifically to an **Action** from that **Controller**). In many cases, the *Controller* will create an instance of a *Model*, then pass it to a *View* to build a page that will eventually show up in the user's browser.
- These are C# classes derived from the **Microsoft.AspNetCore.Mvc.Controller** class (these files will have extension .cs).

Confused? Do not worry! We'll cover these concepts in more depth in the next few chapters. This section is only meant to provide a quick introduction to what MVC pattern entails.

6.3 A Quick Dive into an MVC Example (Optional)

If you don't have time to review this section, you can skip it. In here we want to clarify a little more the concepts included above, and also give you a very quick tour into a simple MVC example.

Let's create an ASP .Net Core MVC web application. Open Visual Studio, then click on the *Create a new project* button.

Then, in the *Create a new project* window, enter the word MVC in the search bar, then select *ASP .NET Core Web App (Model-View-Controller),* and click the *Next* button.

Very important:

- Make sure to choose the option that uses C#, not some other programming language! If you look carefully, there are other languages available too (for example, F#).
- A common mistake we've seen in class is that students would choose *ASP .NET Core Web App*. If you look carefully at its description, you'll note that it uses **Razor Pages** instead. Please do NOT use that option for this book.

Next, choose a name and location for your project. We called ours *DiveIntoMVC*, then click on the *Next* button.

In the last step, in the *Additional information* window, you can choose a .Net framework to work with. In this book, we'll always go with *.Net 6.0 (Long Term Support)*. Also, uncheck the *Configure for HTTPS*. We won't use it in this example. Then click on the *Create* button.

Once the new project is ready to run, either press Ctrl + F5, or go to *Debug > Start Without Debugging*, or click on the light green play button located around the center of the top menu options.

This will start, in a browser, your newly created ASP .Net Core MVC web application (note the URL: `localhost:5096`).

You should note that the opened webpage has a **navbar** at the top of the page. As you click on those menu options or on any of the links from this webpage (Fig. 6.3), the (local) server responds (see Fig. 6.4) with another webpage (with URL: localhost:5096/Home/Privacy).

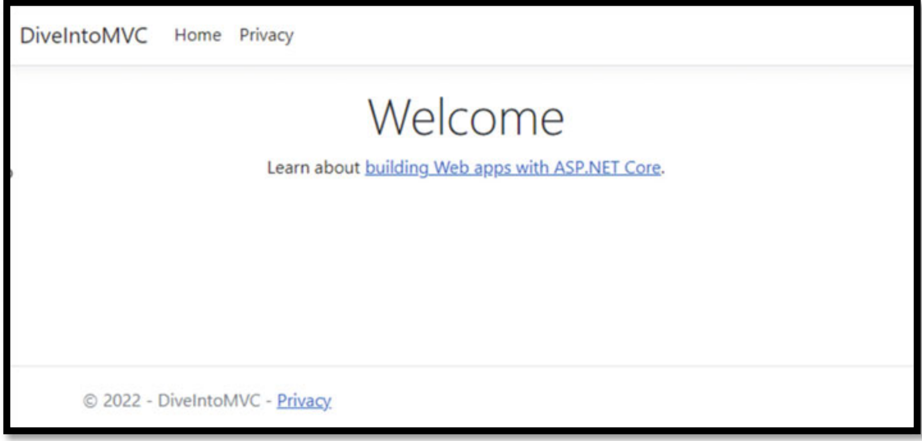


Fig. 6.3 This is the same as Fig. 6.1 described above

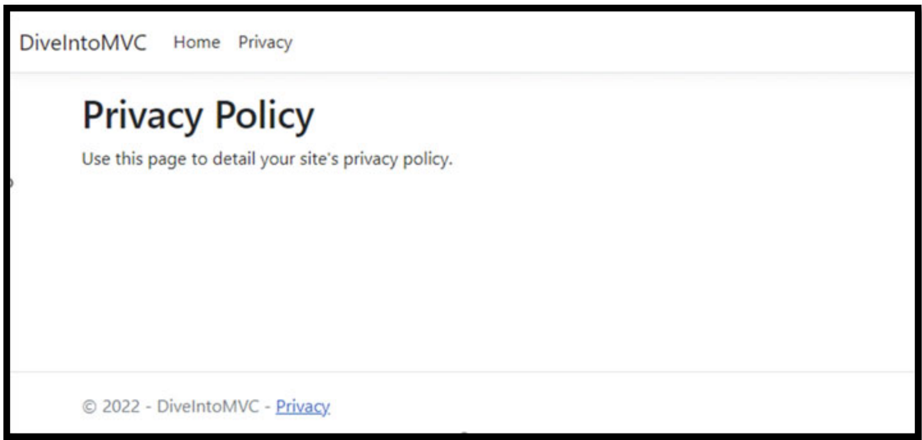


Fig. 6.4 Shows the Privacy page displayed in a browser

To know what goes into this project, check out the *Solution Explorer* window. There are several files and folders showing up in there. In particular, you should note the following:

- We have separate folders for **Models**, **Views**, and **Controllers**.
- There is a (special) folder, called **wwwroot**, and it seems to contain JavaScript, CSS, and other files.
- There is a **Program.cs** file, just like the one we've seen when we created Console applications.

Let's dive in a little deeper.

When you run the application, it opened in a browser window. In our example, it opened up with the URL: (localhost:5096).

If you have multiple browsers installed on your machine, you can open the same URL from multiple browsers. If you open the *launchSettings.json* file, you will see that the value (**port number**) 5097 was set in that file. You can change that value to another number, let's say **5096**. If you rerun your application, the new value will be used for your web application. Be careful, some port numbers are already used (by other applications) for other purposes.

```
"profiles": {
  "ASPBookProject": {
    "commandName": "Project",
    "dotnetRunMessages": true,
    "launchBrowser": true,
    "applicationUrl": "http://localhost:5096",
    "environmentVariables": {
      "ASPNETCORE_ENVIRONMENT": "Development"
    }
  },
```

If you open the site.css file found under *wwwroot* > *css*: you should find very familiar code (CSS styling).

```
html {
    font-size: 14px;
}


@media (min-width: 768px) {
  html {
    font-size: 16px;
  }
}


html {
  position: relative;
  min-height: 100%;
}

 body {
   margin-bottom: 60px;
 }
```

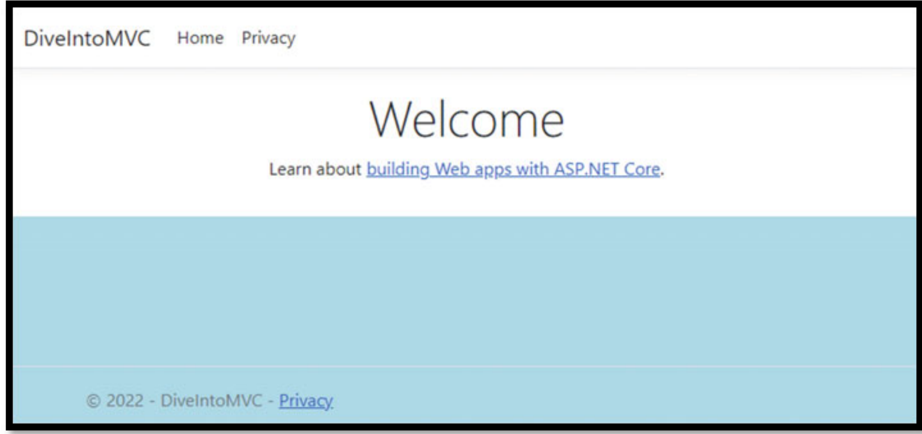


Fig. 6.5 Shows the Welcome page in a browser, after adding the CSS code, shown above, to the site.css file

After line 2, add the following line:

```
background-color:lightblue;
```

Then, either rerun the application. Did you see any changes?

Very important: Often, when you make changes to your CSS files, you may not see them show up in the browser. The reason is as follows: the browsers may cache the CSS file and if you reload the page, it will reuse an already downloaded CSS file. The solution/workaround: press **Ctrl + F5** in your browser to tell your browser to reload everything, including the CSS file. The outcome is shown in Fig. 6.5 (at the URL: localhost:5096).

The next stop is the *_Layout.cshtml*. In here you should note two things as follows:

- It defines the *navbar* we see on our pages (similar to the one we've seen in Chap. 4).
- It links to the *site.css* file we've seen above.

There are many things to show, but we'll learn about them in more details in the upcoming chapter. If you have time and a keen interest, look around on your own. In particular, please explore the Models, Views, and Controllers folders.

If you are using Visual Studio Mar, or Visual Studio Code, check out the links below to help you create your first MVC application:

- https://github.com/dotnet/AspNetCore.Docs/blob/main/aspnetcore/tutorials/first-mvc-app/start-mvc.md
- https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/start-mvc?view=aspnetcore-6.0&tabs=visual-studio.

6.4 Let's Start Our ASP .Net Core Application Project in Here

Open Visual Studio and *Create a new project*. Make sure to choose the ***ASP .NET Core Empty*** template!

Then, choose a name for your project. We called ours *ASPBookProject*. Then click on the *Next* button.

We'll then choose `.Net 6.0(Long Term Support)` for the framework, and make sure to uncheck the *Configure for HTTPS*. Then click the *Create* button.

Now run your application and make sure it opens in a browser (for us, the URL will use another random port: localhost:5125)—it should look similar to Fig. 6.6.

Before we move on to the next section, let us compare what we got here (where we created an Empty web application) against what we got in the previous section (where we created an MVC application). In this book, we will continue with our *empty application* and will build our way up to an *MVC application*.

6.4.1 The Empty Web Application Starting Point

The files in the project (seen from the *Solution Explorer* window) are far fewer than before (check it out!).

Also, the *Program.cs* file contains the following starting point/initial code:

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/", () => "Hello World!");

app.Run();
```

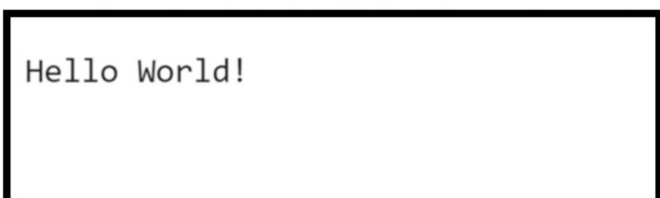


Fig. 6.6 Shows the "Hello World!" in a browser

6.4.2 The MVC Web Application Starting Point

The MVC web application has many more initial files (check out the *Solution Explorer* window!). In particular, you should note the wwwroot folder (containing other subfolders and CSS and JS files), the Models folder, the Controllers folders, the View folder, and many others. Also, note that the *Program.cs* file contains the following code:

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddControllersWithViews();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Home/Error");
}
app.UseStaticFiles();
app.UseRouting();

app.UseAuthorization();

app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");

app.Run();
```

6.5 Entry Point to Our Web Application: Program.cs

Just like we've seen in the previous chapter when we covered *Console Applications*, the **entry point** to a C# application is the **`Main`** method. Since we made use of **top-level statements** in our project, (in particular in the *Program.cs* file), the `Main` method is automatically created by the compiler for us in any one (and only one) file where we use top-level statements. For us, this is the *Program.cs* file.

As a consequence, we can look at *Program.cs* as the entry point to our web application (but really, `Main` is the real entry point, *Program.cs* can be renamed (for example, *EntryPoint.cs*) and our web application would still work).

In the *Program.cs* file we'll **configure services** and **create the middleware pipeline** for our web application. These two topics are introduced below, but they will make much more sense as we dive deeper into this book. *Services* in particular will make much more sense later when we get to see the need they solve and see how services work with the other parts of the application.

6.6 The Middleware Pipeline

Think of the **middleware pipeline** as the entry point into your web application. All **HTTP requests** pass through this point. In here, you can decide how to respond to your requests. In the *middleware pipeline,* one can add various **middleware components** that (see more details in [48])

- route certain requests to the appropriate controllers (we'll cover this in the next chapter);
- add authentication and authorization support;
- log all HTTP requests and responses;
- provide support for static files;
- provide support for caching responses;
- provide support for managing user sessions;
- and many others.

As we'll see below, each middleware component is responsible for **invoking the next component** in the pipeline, or not (in which case we say that the component is **short-circuiting** the pipeline, making this component a **terminal middleware**).

Before we see some examples, we should note the following. For every **HTTP request** received by our web application, the ASP .Net Core platform will create a **Request** object (that contains information regarding the request received from the client) and a **Response** object (that contains information about the response being sent back to the client)—see Fig. 6.7. Each middleware component can inspect the **Request** object and modify the **Response** as needed. We get access to these objects (and others) via a variable of type **HttpContext** (see the example below).

6.6.1 The Current Code in Our Project

Let's first go over the existing code in the *Program.cs* file, in our ASP .Net Core application example:

```
//set up the basic features of the ASP.NET Core platform
var builder = WebApplication.CreateBuilder(args);

//set up middleware components.
var app = builder.Build();

//set up one middleware component
app.MapGet("/", () => "Hello World!");

app.Run();
```

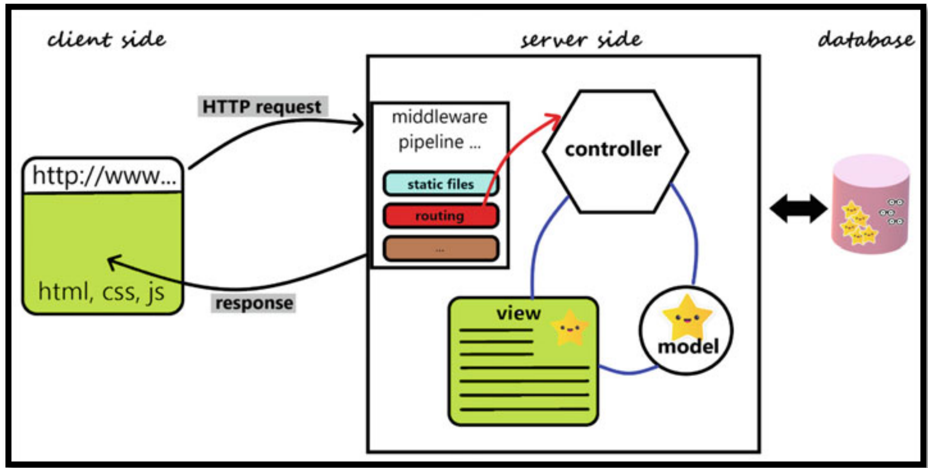


Fig. 6.7 Is similar to Fig. 6.2, it shows how HTTP requests and responses are being used for the interaction between a client and a server

If you run this application, you'll get the following response in a browser (URL: localhost:5125): `Hello World!`

The port number shown in your URL is probably different. If it is different, you should change it, so we all use the same port number. First, double click on the *launchSettings.json* file (inside *Solution Explorer* window look inside the `Properties` folder) to open it. Then, make sure to change the port number to match ours (`5125`):

```
"profiles": {
"DiveIntoMVC": {
  "commandName": "Project",
  "dotnetRunMessages": true,
  "launchBrowser": true,
  "applicationUrl": "http://localhost:5125",
  "environmentVariables": {
    "ASPNETCORE_ENVIRONMENT": "Development"
  }
},
```

Alternatively, you can leave the default value generated for your port number as is, but please make sure to use that number (instead of the one we use in this book: 5125) in all examples shown below.

6.6.2 Run, Use, and Map

Next, we'll introduce some simple *middleware components*, but please keep in mind that we'll see more of them (and more useful ones for this book) in future sections and chapters.

One way to build the *middleware pipeline* is by making use of **request delegates**. These can be configured using the *Run*, *Use*, and *Map* methods. Let's quickly introduce them, then see some examples. Note: we won't make much use of the *Use*, *Run*, and *Map* middleware after this chapter—we just use them to introduce other concepts.

The **Use** *middleware* allows a parameter, `next`, which is a *reference to the next middleware* in the pipeline. One can chain multiple request delegates using `next.Invoke()` (to proceed to the next middleware in the pipeline). If a `Use` middleware does not call `next.Invoke()` then it is said that this component is *short-circuiting the pipeline*.

The **Run** *middleware* is similar to the *Use middleware* but it does not have the `next` parameter, hence it cannot call a next middleware component. Because of this, `Run` is a *terminal middleware* and should be placed at the very bottom of the middleware pipeline.

The **Map** *middleware* branches the request pipeline based on matches of the given **request path**. We'll skip this in our book.

Important note: As we will see below,

- the order in which we declare our *middleware* components is important;
- the order in which we declare our *services* is not important.

6.6.3 First Example

Let's create an example of middleware pipeline based on the `Use` and `Run` defined above.

First, replace the line containing `app.MapGet` with the following:

```
app.Run(async context => { await context.Response.WriteAsync("Hello from Run middleware"); });
```

Your *Program.cs* file should look like this:

```
//set up the basic features of the ASP.NET Core platform
var builder = WebApplication.CreateBuilder(args);

//set up middleware components.
var app = builder.Build();

//set up one middleware component
app.MapGet("/", () => "Hello World!");

app.Run();
```

We used a simple middleware component. Whatever the *HTTP request* is, our web application responds with `Hello from Run middleware`.

Try each of the following requests:

http://localhost:5125/

http://localhost:5125/Ada

http://localhost:5125/Turing/Alan

You should get the same response in each case: **`Hello from Run middleware`**.

Let's change this a little bit, so we make use of the `Request` object. Replace the line.

```
app.Run(async context => { await context.Response.WriteAsync("Hello from Run middleware"); });
```

that was added earlier with the following and recompile your project (or use the **Hot reload** button and refresh your browser):

```
app.Run(async context => {
    await context.Response.WriteAsync($"You have requested {context.Request.Path}");
});
```

Now, if you use the links above, you should get different responses depending on the HTTP request used.

For example,

http://localhost:5125/ will give you the following response: **`You have requested/`**.

Similarly, http://localhost:5125/Turing/Alan will return: **`You have requested /Turing/Alan`**.

Using Microsoft *IntelliSense*, you should be able to play with the line of code we just added in Visual Studio and find more details about the `context`, `Request`, `Response`, etc.

If you hover your mouse over the text `context`, you'll find out it represents (references) an object of type: `HttpContext`.

If you put a dot right after `context`, you'll quickly find out what methods and properties it has. In particular, you should note the following properties: `Connection`, `Request`, `Response`, `Session`, and others.

Similarly, if you put a dot right after `Request`, you'll find out that you can get access to a lot of information regarding the *HTTP request*. In particular, you can find information

regarding the `Body` of the request the `Path` (see the example above), the `Method` (get vs. post), the `QueryString`, and so on.

You should note that in the example above, we are returning back to the user what we build inside the `context.Response` object.

6.6.4 Second Example

Now let's add a second middleware component. Add the following code, right before the very last line (before `app.Run();`)

```
app.Run(async context => {
    await context.Response.WriteAsync($"The second Run");
});
```

Rebuild your project and open the following in your browser (this will send an **HTTP request** to your web application):

http://localhost:5125/Turing/Alan

Here is the output: **`You have requested /Turing/Alan`**.

Why didn't our newly added code run? The answer is `Run` is a terminal middleware, so it won't invoke the next middleware component. To chain multiple middleware components, we'll make use of `Use`. Remember to always put this request delegate (`Run`) last in your middleware pipeline.

Note: Look at the last line in *Program.cs*. What is its role? Again, make use of IntelliSense, good documentation is very helpful in this case. Hover your mouse over the two app.Run calls shown below. What do you notice?

You should note that `Run` is an *overloaded* method. We can use it with a **request delegate** argument (the first of the two screenshots), but you can also use it with no arguments.

The first call is used to add a middleware delegate to the request pipeline. We use the second call, so the application doesn't shut down too early (since we're dealing with asynchronous calls). Make sure to not delete this last line by mistake.

6.6.5 Third Example

To chain multiple request delegates, we can utilize `Use`. Modify your *Program.cs* file so it looks like the code below:

```
Hello from Middleware component 1 start
Hello from Middleware component 2
Hello from Middleware component 1 end
```

Fig. 6.8 Shows the output (in a browser) from the middleware components described above

```
//sets up the basic features of the ASP.NET Core platform
var builder = WebApplication.CreateBuilder(args);

//set up middleware components.
var app = builder.Build();

app.Use(async (context, next) =>
{
    await context.Response.WriteAsync("Hello from Use start\n");
    await next.Invoke();
    await context.Response.WriteAsync("Hello from Use end\n");
});

app.Run(async (context) =>
{
    await context.Response.WriteAsync("Hello from Run\n");
});


app.Run();
```

What is the output? More importantly, in which order? Let's use the following: http://localhost:5125/Turing/Alan.

We obtained (see Fig. 6.8).

In particular, we should note that `next.Invoke();` was used to call the second component.

Very important: The order in which middleware components are declared is very important. In particular, what would happen if we switched the order of Run and Use, as shown below? Why?

```
app.Run(async (context) =>
{
    await context.Response.WriteAsync("Hello from Run\n");
});

app.Use(async (context, next) =>
{
    await context.Response.WriteAsync("Hello from Use start\n");
    await next.Invoke();
    await context.Response.WriteAsync("Hello from Use end\n");
});
```

If time, check out the following. What does it do? What URLs would unlock the secret?

```
app.Use(async (context, next) =>
{
    await context.Response.WriteAsync("Hello from Use start\n");

    if(context.Request.Path.ToString().Contains("SECRET"))
        await next.Invoke();

    await context.Response.WriteAsync("Hello from Use end\n");
});

app.Run(async (context) =>
{
    await context.Response.WriteAsync("SECRET UNLOCKED\n");
});
```

We won't be using the `Use` and `Run` in the upcoming chapters of this book. We only use them in here to give you some sense of what the *middleware pipeline* is and understand a little bit about what an *HTTP Request* is. In the next section, we'll introduce another *middleware component*, this one is very important, and we'll use this middleware component until the end of the book. Please make sure to understand it.

6.6.6 Other Middleware Components

ASP .NET Core comes with many *middleware components* ready for use. We will see some of them in this chapter, and others later in this book. To learn more about them, check out the table shown in [48]. Here is a list of some middleware components we'll see in this book:

- `UseAuthentication`—provides support for authentication.
- `UseAuthorization`—provides support for authorization.
- `UseDeveloperExceptionPage`—generates a detailed error page with information intended for use only in the *Development* environment.
- `UseExceptionHandler`—helps return a friendly error page that we'll want to use for the *Production* environment.
- `UseRouting`—helps process requests with MVC.
- `UseStaticFiles`—provides support for serving static files (seen next).

6.7 Static Files Middleware

6.7.1 What Are *Static Files*?

Files that do not change at *runtime* are called **static files**. These files are not *dynamically* generated or modified when the user interacts with our web application, so we call them **static**. The following are some examples of *static files*: CSS files, (some) HTML files, JavaScript files, and some images and videos (company logo, company intro). For now, focus on their functionality; we'll see them more in depth as we go through the next chapters. To learn more, check out [49].

6.7.2 Where Do We Store *Static Files*?

We typically store static files in the project's **web root directory** (this is just a folder named **wwwroot** created directly in the root of the project). We can store them elsewhere, but we won't do that in this book.

To create a *web root directory*, in the *Solution Explorer* window, right-click on the project's name (*ASPBookProject*) and select *Add > New Folder*. There, type in the name `wwwroot`. Once you press the enter key, you should notice that Visual Studio is using a special icon for this folder, which should suggest this is an important folder.

In this folder/directory, one can add *files* and *subdirectories*. Let's add a few images in there. First, in the *Solution Explorer* window, right-click on the wwwroot folder, and select Add > New Folder. Choose a name for this new folder (we chose *images*). Then, inside *images*, add a few images (one can use *drag and drop* to copy images in this subdirectory).

6.7.3 How Do We Allow Access to *Static Files*?

By default, files from *wwwroot* are NOT accessible from the client side. To give your clients access to these files, you can use the **UseStaticFiles** middleware component, as seen in the example below.

Let's change the *Program.cs* to match the following contents:

```
//sets up the basic features of the ASP.NET Core platform
var builder = WebApplication.CreateBuilder(args);

//set up middleware components.
var app = builder.Build();

app.UseStaticFiles(); //needed to give access to files in wwwroot

app.Run(async (context) =>
{
    await context.Response.WriteAsync("Welcome to our web application");
});


app.Run();
```

6.7.4 How Can We Access *Static Files*?

To access static files, you need to use paths relative to the web root. For example, if we wish to access the file: `wwwroot`**`/images/image01.JPG`**, we will use http://localhost:5125/images/image01.JPG (see Fig. 6.9).

You should note the following:

- The order of the middleware components is very important. Since we used the `UseStaticFiles` middleware component before the `Run`, we were able to obtain the static file (in our case `image01.JPG`).
- The `UseStaticFiles` is a *terminal middleware*, so the `Run` component was not run.

Fig. 6.9 Shows a static file (in here an image), displayed in a browser. Note the URL to this image contains the relative path of this image inside the web root

- But `UseStaticFiles` middleware will only run if the requested *path* matches a file inside the wwwroot (with the same path!). Otherwise, the next middleware will be called, in our case the `Run`
  - If you use the URL: localhost:/5125/images/image011.JPG, you should get the following text displayed inside the browser: `Welcome to our web application.`

Very important: We can (and later will) call `UseStaticFiles` before calling `UseAuthentication`. This allows access to static files even to users who are not yet logged in. Therefore, all files under *wwwroot* are publicly accessible, so be careful not to store any sensitive files in there.

If you have time and interest, check out and learn about the ***Directory browsing*** section in [49] which we skipped.

6.7.5 Default (Static) Page

To serve a default webpage from *wwwroot* without the need to provide the filename in the request URL, we can

- call the `UseDefaultFiles` middleware before (and in addition to) the `UseStaticFiles` middleware, and
- put the webpage directly inside the *wwwroot*, with either of the names: default.htm, default.html, index.htm, index.html.

We won't spend too much time on this default webpage because we'll use another default page when we work with the MVC pattern. So please don't spend too much time on this webpage.

As an example, we'll copy the *firstwebpage.html* into the *wwwroot* folder for our ASP .Net Core project. Once in there, rename it as *index.html*. Double-checked that you copied the file inside *wwwroot*, not inside the *images* folder!

Then, add the following line right before `app.UseStaticFiles();`

```
app.UseDefaultFiles(); //needed for the default page
```

That's it. Rebuild your web application and run it again (see Fig. 6.10). Here is what we got (URL: localhost:5125).

Let's now also copy the CSS file we created earlier. Let's put it inside the *css* subfolder (create it!) inside *wwwroot*. Then open the *index.html* page and update the link path from

```
<LINK rel="stylesheet" href="personal.css">
```

into

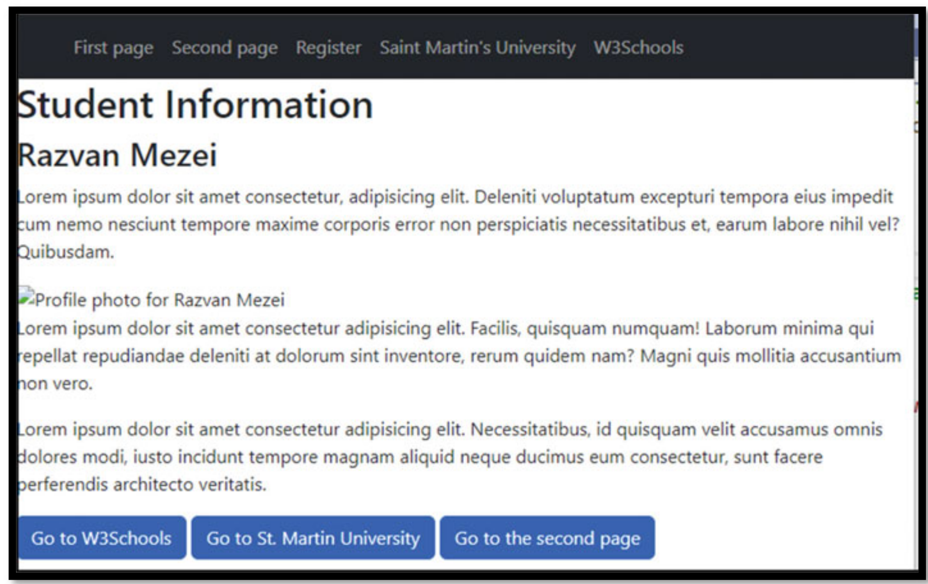


Fig. 6.10 Shows the default page (index.html) being set for our application. Note how some elements are not properly displayed (in particular the CSS and the embedded image need to be updated)

```
<link href="css/personal.css" rel="stylesheet" />
```

Note: You can drag and drop the .css file from the Solution Explorer into your *index.html* file: Visual Studio will automatically paste the corresponding link element.

Let's also update the image. Inside the *index.html*, we also need to update the path to the image used in it (currently, this is: `<IMG SRC ="image01.JPG"` …). One easy way to do this is as follows. Delete the text from the quotes shown in the image, then press Ctrl + Space to get IntelliSense support. We used: `<IMG SRC ="/images/image01.JPG"`.

With these changes, we are now able to make our web application display a default page that makes use of the following static resources: a CSS file and an image file (see Fig. 6.11).

Please comment out (or delete) the following CSS code from the *www* > *css* > *personal.css* file:

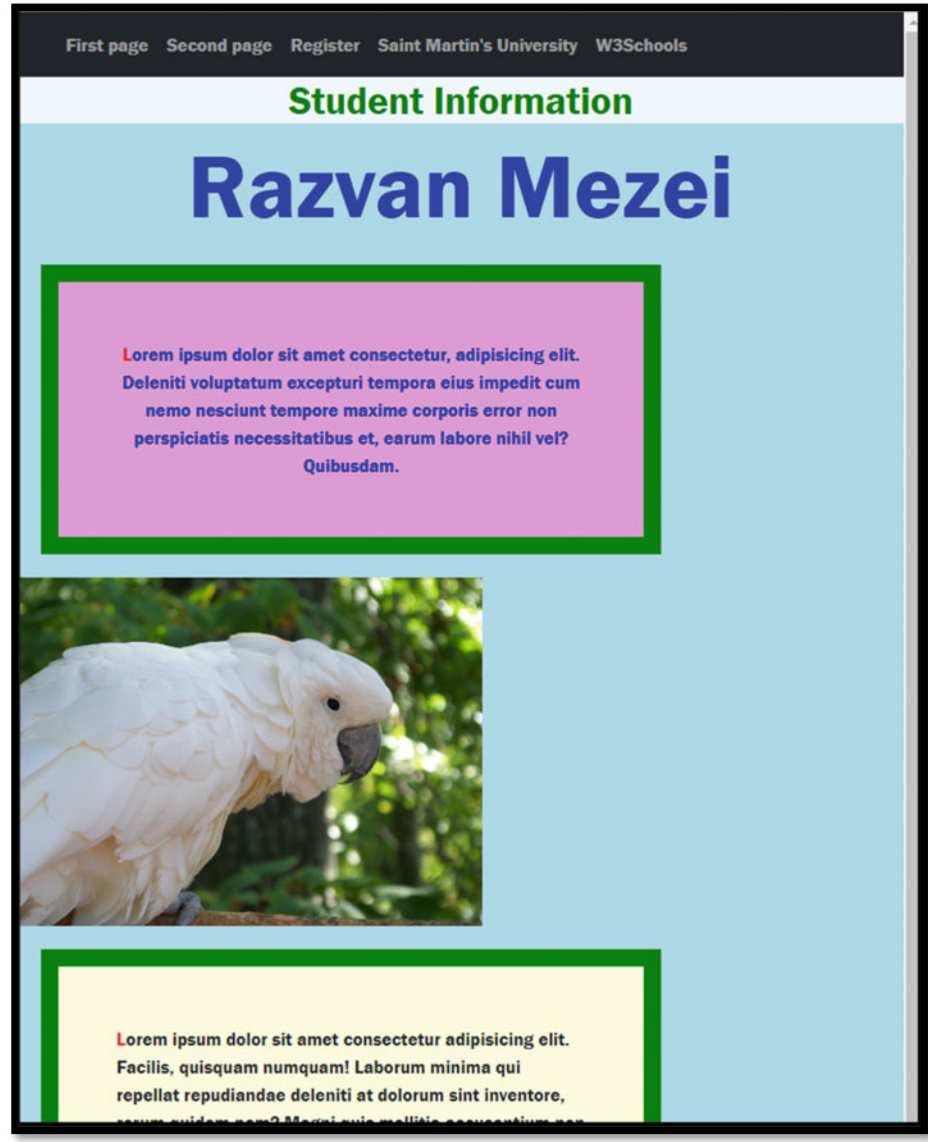


Fig. 6.11 This is the same as Fig. 6.10, but with the image and CSS contents properly set

```
p {
    background-color: lightyellow;
    width: 70%;
    border: 15px solid green;
    padding: 50px;
    margin: 20px;
}
```

With this change, the page (see Fig. 6.12) now looks a little better:

Fig. 6.12 This is the same as Fig. 6.11, after changing the personal.css file as discussed above

6.8 Introduction to Services (Optional)

Services are very important, and they will make more sense later when we get to solve real problems with them. In here we want to give a brief introduction to *services* since they are also using the *Program*.cs file. Feel free to skip this section if you wish and be sure to revisit this section again when you encounter *services*.

Services are essentially classes that can be reused easily in multiple locations without having to worry about instantiating them and their various dependencies.

One example of a *service* that we'll see later will be the class that will be responsible for connecting to a database. We won't use multiple instances of such an object (we only need one object connecting to the database), but we may need access to (we'll essentially reuse the same one instance of) this class from various parts of the web application. We'll get access *services* (to these instances) using a process called ***dependency injection***.

This is facilitated via a technique known as *Dependency Injection*. The **Dependency Injection** is (a factory) responsible for creating instances of the dependencies when they are needed and disposing of them when they are no longer needed. To register a *service* with the *Dependency Injection*, we'll make use of the `builder.Services` (we'll see an example below). Read more about *Dependency Injection* in here [50].

Let's see an example (inspired by the example in [51]). We'll see more useful *services* in the upcoming chapters, this is just to demonstrate the steps involved in creating and using a *service*.

6.8.1 Example—Step 1: Define a Class and An Interface

Any **class** that implements any **interface** can act as a *service*. Let's create a sample *interface* and a *class* that implements it (you can place them anywhere in your project, we created a *Services* folder and put them in there):

```
public interface IMyFirstService
{
    string MyMethod();
}


public class MyFirstService : IMyFirstService
{
    public string MyMethod()
    {
        return $"hash of current instace of my service: {this.GetHashCode()}";
    }
}
```

6.8.2 Example—Step 2: Register a Service

The class above is not yet a *service*. To make it a *service*, you'll need to **register** it as a *service* (in *Program.cs*). One way to register it as a service is as follows (more details below):

```
builder.Services.AddSingleton<IMyFirstService,MyFirstService>();
```

Make sure to add this line before the `Build` method is called, namely before the line:

```
var app = builder.Build();
```

Now we have a *service*. This code above will add the *service* to the *dependency injection* container. We won't have to worry about creating an instance of the *MyFirstService*. The *dependency injection* will manage its instance.

6.8.3 Example—Step 3: Use a Service

To use this newly created *service,* we need one last step. We'll need to *inject* the service where we want to use/access it. For completeness of this example, below we'll see how we can *inject* a service in a *middleware* component. But in later chapters, we'll see how easily (easier than this example) they can be injected into *controllers* and *views.*

Change the middleware component.

```
app.Run(async (context) =>
{
    await context.Response.WriteAsync("Welcome to our web application");
});
```

Into:

```
app.Run(async (context) =>
{
    var myService = app.Services.GetRequiredService<IMyFirstService>();
    await context.Response.WriteAsync(myService.MyMethod());
});
```

Then, open the following link in a browser: http://localhost:5125/anything.

While the application is still running, open the same link in another browser, or use an incognito session. We obtained the following results (one for each browser window opened):

```
hash of current instance of my service: 17556181

hash of current instance of my service: 17556181

hash of current instance of my service: 17556181
```

You should note that all display the same hash value, an indication that there is (probably) only one object used for both requests.

`AddSingleton` is used when you want to create one instance of the service for the web application's lifetime. So as long as you don't restart your web application, all requests will make use of the same one instance of the service.

An alternative to `AddSingleton` is `AddTransient`. Use `AddTransient` if you want the dependency injection to create a new instance of the service every single time the service is injected (in particular, every time you click on a link, a button, or refresh a page). To test this, replace `AddSingleton` with `AddTransient` in your code and run again your web application as seen above. Here is what we obtained:

```
hash of current instance of my service: 11429296

hash of current instance of my service: 42194754
```

7 Routing, Models, and Controllers

Our journey into the ASP .Net Core MVC development starts here. In this chapter, we'll focus on *routing*, but we'll also see a first example of *controllers* and other related topics. In particular, we'll see two types of routing, *conventional routing* and *attribute routing*. Routing gives us the developers complete control over the URLs used in our web application. In particular, this could be helpful for search engine optimization (SEO).

Important note: As we noted at the beginning of the book, the MVC design pattern emphasizes **separation of concerns**, by considering three major components: **models**, **views**, and **controllers**. We'll get a good understanding of these components as we go through this book, but please have patience until we finish this and the next two chapters. Some concepts (such as *routing and models*) should make complete sense in this chapter, while others (such as *actions, controllers, and views*) are only introduced in here, but will make more sense as we go through this and the next two chapters. We'll see several examples, and by the end of Chap. 9, you should have a good grasp of the MVC design pattern.

Because we're introducing several new concepts, some in more details in the following chapters, this chapter may be a little confusing at first. It will get better once we cover more details in the subsequent chapters.

7.1 A Little Cleanup Before We Continue

Before we proceed, let's clean up our project a little bit. In *Program.cs* delete all lines of code except for the following:

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024

R. A. Mezei, *Introduction to the Development of Web Applications Using ASP .Net (Core) MVC*, Synthesis Lectures on Computer Science,
https://doi.org/10.1007/978-3-031-30626-6_7

```
var builder = WebApplication.CreateBuilder(args);

//set up middleware components.
var app = builder.Build();

app.UseStaticFiles(); //needed to give access to files in wwwroot

app.Run();
```

Optionally, you can also remove the following file and folder which we won't use anymore in this book:

- The file: *wwwroot* > *index.html*.
- The folder: *Services* (if you created it above) and its contents.

If you recompile and run your code, you should get the `HTTP ERROR 404`. This should be expected, we do not have anything in the middleware pipeline to answer the HTTP requests from clients.

7.2 Some Essential MVC Concepts and the HTTP Request Lifecycle

Let's have a very brief review of some essential MVC concepts. We'll go into more details in this and the next two chapters.

The **models** are the classes that represent the various types of objects managed by the web application. These objects represent the ***state of the application***.

- For example (think about a web application such as Amazon, Canvas, Moodle, …), we could have the following model classes: Course, User, Student, Instructor, Administrator, Product, Seller, Buyer, and so on.

The **views** will make up the ***user interface***. We'll present content to users via views (more accurately, we'll use views to build webpages that ultimately will get displayed in a user's browser). We'll see more about this later.

- For example (think about a web application such as Amazon, Canvas, Moodle, …), we can have a view that will be used to display a list of all courses taken by a student, or a list of all laptops available to buy. We could use another view to build a page that allows our users to change their passwords.

The **controllers** will handle the ***user interaction***. We'll see them below.

- For example (think about a web application such as Amazon, Canvas, Moodle, …). What happens when you click on a button? Or click on a link? Or load the first welcome page? In each of these, your requests will (eventually) be sent to a **Controller** (more specifically to an **Action** from that **Controller**). In many cases, the *Controller* will create an instance of a *Model*, then pass it to a *View* to build a page that will eventually show up in the user's browser.

Now, let's see the **request lifecycle** (see Fig. 7.1). This is just an introduction, so you get some sense of what we're dealing with. We'll get into more details below. What happens when a user requests a page (either types in a URL, or clicks on a link or button):

Step 1: The user sends an **HTTP request**.

- For example, the user enters the following URL: http://www.mysite.com/instructor/show/1.
- For our book, we will run the server from a local host. So instead of http://www.mysite.com we'll use http://localhost:5125.

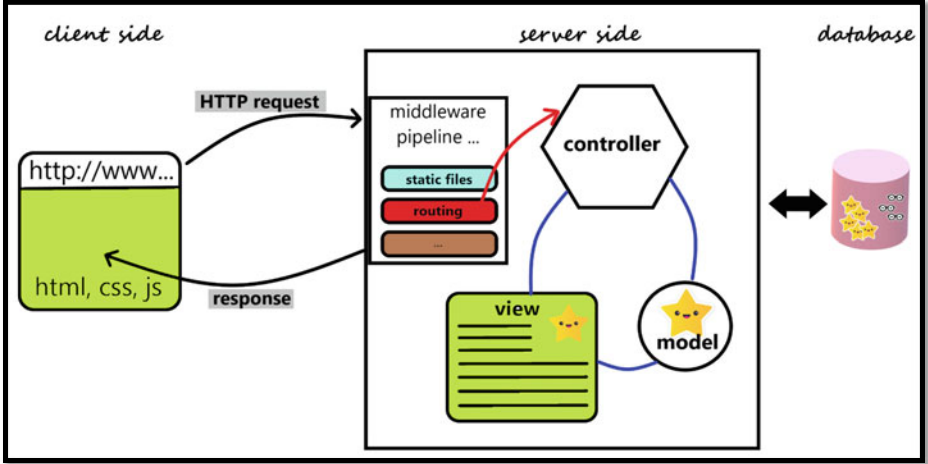


Fig. 7.1 Shows the main components of an MVC web application. In particular, the client side uses a browser (HTML, CSS, and JavaScript), then on the server side we have the middleware pipeline, controllers, models, and views, and lastly, the data may be stored in a database

Step 2: A **controller** object is instantiated to respond to this request.

- The URL **routing** determines which **controller** & **action** will handle the HTTP request.
- If we assume the **default routing**, presented below, the InstructorController will be instantiated.

Step 3: An **action** method is then called by the **controller**.

- If we assume the default routing, a Show **action** will be called by the controller.
- A **model binder** helps us determine the values passed to the **action** as parameters (e.g., 1).
- If needed, the **action** may create a new instance of a **model** class. In our example an **Instructor** object.
- Typically, the **action** will pass the model object to a **view** to display the requested results.

Step 4:

- We'll make use of views to produce the output that is sent back to the client's browser.

By the end of this and the next two chapters, you should completely understand the steps described above. For now, they are just meant to provide a map of what we're dealing with.

7.3 Introduction to Routing

As we will see below, **routing** is one *middleware component* that will send requests (route them) to *actions* in *controllers*. Since we didn't yet see what *actions* and *controllers* are, think of *routing* as the *middleware component* that will send incoming *HTTP requests* (only those that follow a specific format) to the MVC part of our web application.

Routing gives us, the developers, full control over the format of the URLs in our web application. It lets us describe what URL paths will be matched to what *actions*. As we will see later (when we cover *tag helpers* and *HTML helpers*), routing is also used when generating URLs for various links and buttons—if we change the routing, the (tag and html) helpers will generate different links that conform to our prescribed routing.

7.3.1 Adding MVC to Our ASP .Net Core Application

To add the **MVC framework** to our ASP .Net Core web application, we'll need the following two lines of code:

```
builder.Services.AddControllersWithViews(); //adds services needed for controllers
```

```
app.UseRouting();  //adds route matching to the middleware pipeline
```

Add these lines so *Program.cs* looks as follows:

```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddControllersWithViews();//adds services needed for controllers

var app = builder.Build();//set up middlewate components
app.UseStaticFiles(); //needed to give access to files in wwwroot
app.UseRouting(); //adds route matching to the middleware pipeline
app.Run();
```

7.3.2 Default Routing, the Home Controller, and Actions

7.3.2.1 Default Routing

To understand what *routing* is, let's first start with the **default routing**. After `app.UseRouting();` add the following:

```
app.MapDefaultControllerRoute(); //adds default routing
```

Or, equivalently, one can add:

```
app.MapControllerRoute( //adds default routing
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
```

Let's make sense of this **default routing**. In here, we have the following:

- A `name` that must be present (and distinct from other route names), but it is not used in the routing itself.
- A `pattern` that describes what requests it needs to match.
  - We can have multiple routes. Above is just one, we'll see more below.
  - HTTP requests that do not match this pattern will be ignored by this route.
  - In our program so far, if a request does not match the route, we'll get the HTTP 404 error.

The pattern above has three **segments** (each segment is described in a pair of `{}`) separated by `/`.

- The first segment `{controller=Home}`
  - It specifies that the first value will represent a **controller**. If we don't provide one, the `HomeController` will be used.

- The second segment `{action=Index}`
  - It specifies that the second value will represent an **action**. If we don't provide one, the `Index` action will be used.
- The third segment `{id?}`
  - It specifies that the third value will represent a value for a variable called `id` (we'll use it for **action** parameters with the name `id`). The `?` means that a value for `id` is optional (so the URL may not include a value for this third segment).

This will make more sense once we add a *controller* and see it in action. Please be patient. It will make complete sense soon.

With the *default routing* described above:

- The link: http://localhost:5125/ will send our HTTP request to the controller **HomeController**, action **Index**, and there is no specified **Id**.
- The link: http://localhost:5125/Home will send our HTTP request to the controller **HomeController**, action **Index**, and there is no specified **Id**.
- The link: http://localhost:5125/Instructor will send our HTTP request to the controller **InstructorController**, action **Index**, and there is no specified **Id**.
- The link: http://localhost:5125/Home/SecondAction will send our HTTP request to the controller **HomeController**, action **SecondAction**, and there is no specified **Id**.
- The link: http://localhost:5125/Instructor/ListAll will send our HTTP request to the controller **InstructorController**, action **ListAll**, and there is no specified **Id**.
- The link: http://localhost:5125/Instructor/Display/10 will send our HTTP request to the controller **InstructorController**, action **Display**, and the **Id** = 10 (we should use an Id parameter for the Display action).
- The link: http://localhost:5125/Instructor/Display/10/20 **does not match the route above**. The default routing uses up to three segments, but we sent four.

7.3.2.2 Add Our First Controller (The HomeController)

IMPORTANT: Before we create our first **controller**, we want to caution you to be very careful: for some parts (for controllers in particular), the naming is very important. In particular, we say that **MVC relies on convention over configuration** which means that if you use the appropriate name patterns (if you follow the convention), little to no setup is needed. If, however, you choose not to follow the naming convention, you may need to do extra work (either pass extra parameters, or do some setup, and so on).

Let's create our first **controller**. To be nicely organized, we'll create a folder, called **Controllers** directly in the root of the project (make sure to double-check the spelling and location of this folder!). So, in the *Solution Explorer* window, right-click on the project, then select **Add > New Folder**.

For the name, type **Controllers**. To create a new controller, right-click on the *Controllers* folder then select *Add* > *Controller …*

In the *Add New Scaffolded Item* window that opens, select *MVC Controller—Empty* (we'll talk about the other options later) then click on the *Add* button. In the new window that opens, for the name field, enter *HomeController.cs* (make sure to have the correct spelling!).

Congratulations. You just created your first *controller*!

To test this code, change the `Index` method (it's called an *action*) to match the following:

```
public IActionResult Index()
{
    return Content("Hello World from Index action, HomeController!");
}
```

Now we can finally run our web application again. Run your application. You should get (URL: `localhost:5125`).

```
Hello World from Index action, HomeController!
```

How does this relate to the *default routing* we set up above?

7.3.2.3 Introduction to Actions

A **controller** (see more in [54]) is a class derived from the `Microsoft.AspNetCore.Mvc.Controller` class and is used to define a set of related *methods* called *actions*. An **action** is any public method defined inside a controller class (as long as it does not have the `[NoAction]` attribute).

To get some ideas of what actions do, just look at the table below (we'll see them in more depth later in this book). Notice how the related *actions* are included in a *controller* class.

Model class	Controller	Actions
Instructor	InstructorController	Add
		Delete
		Edit
		ShowDetails
		Index or ListAll
User	AccountController	Register
		Login
		Logout
Student	StudentController	Add
		Delete
		Edit

Model class	Controller	Actions
		ShowDetails
		Index or ListAll

For example, when working with `Instructor` data (model), what can we do? We can add a new instructor, delete an existing instructor, edit an existing instructor, show details/display an instructor, and maybe list all instructors. These are (related) *actions* that we can put together in one *controller* class, in here called `InstructorController`.

Similarly, for `User` data (model), we have an `AccountController` that allows us (by means of *actions*) to register a new user account, login a user, or logout a user.

IMPORTANT: All *controller* classes must reside in the project's root-level *Controllers* folder.

We'll see much more on *actions* soon; this chapter merely introduces actions and controllers.

Inside the `HomeController` class, let's add a second action. This time we'll add a very simple public method:

```
public string SecondAction(int id)
{
    return $"({id})^2 = {id * id}";
}
```

Using the *default routing* discussed above, how can you call this *action*?

The following URLs:

http://localhost:5125/

http://localhost:5125/Home/

http://localhost:5125/Home/Index

They all call the `Index` action from the `HomeController`.

To call our `SecondAction` method, we need to use.

http://localhost:5125/Home/SecondAction

We obtained the following in the browsing window: **(0)^2 = 0**.

Or better yet we can also pass a value for the `id`:

http://localhost:5125/Home/SecondAction/4

We obtained the following in the browsing window: **(4)^2 = 16**.

Note: If you don't pass an `Id`, and one is needed by the action, the *model binder* will use the default value (for integers that is 0).

Another note: One can also use query strings to pass values. The *model binder* is clever enough to use them (we'll see *model binder* later):

http://localhost:5125/Home/SecondAction?id=10.

We obtained the following in the browsing window: **(10)^2 = 100**.

What happens when you use any of the following URLs? Why?
http://localhost:5125/Home/SecondAction/4/5
http://localhost:5125/SecondAction/
http://localhost:5125/Home/SecondAction?number=10.

7.4 Add a Model, a Controller, and Views

7.4.1 Add a Model Class

In here we'll add a more meaningful example of *controller* and *actions*. We'll also add a *model* as well as some *views*.

Let's start with creating a model class, let's say a `Student` class.

First, create a new folder named `Models` in the root of the project (use the *Solution Explorer* window). Then, inside this folder, add a new one called `Student`.

When you think of a student, what characteristics would each student have? Below is just a set of characteristics we'll use for this demonstration, but feel free to add more. We will add the following (properties) in our model class:

```
public enum Major { CS, IT, MATH, OTHER }

public class Student
{
    public int StudentId { get; set; }
    public string? FirstName { get; set; }
    public string? LastName { get; set; }
    public bool IsVeteran { get; set; }
    public DateTime AdmissionDate { get; set; }
    public double GPA { get; set; }
    public Major Major { get; set; }
}
```

Above, we tried to include multiple types for our properties, such as integers, strings, Booleans, and an enumeration.

That's pretty much it for a model. Easy, right? We'll add more to this soon, but for now, you should feel quite comfortable defining model classes. They are just POCO (plain old CLR object) classes that you have seen also in Chap. 5, where we reviewed some fundamental concepts in C#.

7.4.2 Add a (Second) Controller Class

We've seen above how we can add a *controller* class. Let's add a more meaningful one in here. This one will work (in conjunction) with the *model* class created above.

Let's first decide what actions we would like to be able to perform on this model. Let's say we would like to.

- get a list of Students: the `Index` action;
- get more details for any one particular Student: the `ShowDetails` action;
- later, we'll also want to be able to add a new Student: the `Add` action;
- later, we'll also want to be able to edit the values for any one particular Student: the `Edit` action;
- later, we'll also want to be able to delete an existing Student: the `Delete` action.

We put all these *actions* into one class, a *controller* class, and we'll name it `StudentController`. It is typical for controller classes that work with a model, say called `XYZ`, to be named `XYZController`. To create our controller, right-click on the *Controllers* folder (inside the **Solution Explorer** window) and select *Add* > *Controller* ....

And just like we've seen of the first controller, select *MVC Controller - Empty*, then name it StudentController.cs. Please double-check your spelling for the name of this class.

In this class, delete the default `Index` action, and add the following rather simple *actions*:

```
public IActionResult Index()
{
    return Content("Student Controller\nTO DO: display a list of student in here");
}

public IActionResult ShowDetails(int id)
{
    //create an instance of the Student model
    // ... this data would normally come from a database ...
    Student st = new Student();
    if (id == 10) //creating one sample student
    {
        st.FirstName = " Aylin";
        st.LastName = "Hopper";
        st.Major = Major.CS;
        st.IsVeteran = true;
        st.GPA = 4.0;
        st.AdmissionDate = DateTime.Parse("2022-08-15");
    }
    else //creating another sample student
    {
        st.FirstName = "Rahsaan";
        st.LastName = "Lubowitz";
        st.Major = Major.IT;
        st.IsVeteran = false;
        st.GPA = 3.95;
        st.AdmissionDate = DateTime.Parse("2021-01-07");
    }
    ViewBag.student = st; //pass the student to the view
    return View();
    }
```

Above, make sure to add the appropriate `using` directive for the `Student` class.

```
using ASPBookProject.Models;
```

One way to pass data from an *action* to a *view* is by making use of the dynamic object `ViewBag`. `ViewBag` is a dynamic wrapper of the `ViewData` dictionary, so we could use either one, but in this book, we'll only use the `ViewBag`.

One way to add data to the `ViewBag` is to use the *dot notation*. Since it is a dynamic object, you'll get no IntelliSense support so just be careful on the names you're using. In the example above, we used `ViewBag.student`. Instead of `student`, you could use any identifier of your choice.

Later we'll see better alternatives to `ViewBag`, (we'll use *strongly typed views*) but this is simpler to use for now.

7.4.3 Add a First View

The simplest way to add a *view* for an *action* is to right-click anywhere inside the action (for us, this is inside the `ShowDetails` method) and select *Add View …*:

Make sure to add a view for the `ShowDetails` action, not the `Index` action. Then, in the *Add New Scaffolded Item* window that opens, select *Razor View* (not *Razor View—Empty*!), and click on the *Add* button.

Then, just confirm that the view's name matches the action name, `ShowDetails`, and uncheck all the options (we'll learn about them later). Then click on the *Add* button.

You should note that Visual Studio created a new folder, called *Views*. Inside it, it created a subfolder named *Student* (from the name `StudentController`), and inside it, you have a new file, for the newly created view (check this out using the *Solution Explorer* window).

Once the *view* file is created, you should see the following starting code in it:

```
@{
    Layout = null;
}

<!DOCTYPE html>

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

Does this look familiar? Except for the first three lines (which you can ignore for now), what you have there is pretty much the basic HTML template we used in the first few chapters of this book. You should feel in a familiar territory.

Let's add some code to this *view*, so it displays the student that was passed via the ViewBag object. Change the contents of the *ShowDetails.cshtml* file to match the following (we'll see views in more depth in the next chapter!):

```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Showing Details of a Students</title>
</head>
<body>
    <h1>Showing details of student: @ViewBag.student.LastName, @ViewBag.student.FirstName</h1>
    <p>Major: @ViewBag.student.Major</p>
    <p>Is veteran:  @ViewBag.student.IsVeteran</p>
    <p>GPA: @ViewBag.student.GPA </p>
    <p>Admission date:  @ViewBag.student.AdmissionDate</p>
</body>
</html>
```

7.4.4 Test Our Code so Far

Compile and run the project. The landing page (based on the default constructor) should be the Index action of the HomeController. The browsing window should display:

Hello World from Index action, HomeController!

Either of the following links should give you the same result as above:
http://localhost:5125
http://localhost:5125/Home
http://localhost:5125/Home/Index

Then, use the following URL: http://localhost:5125/Student

Note: The Student in the URL refers to the StudentController, not the Student model.

Now let's call the Index action of the StudentController class. What URL would you use?

Any of the following should work (with the default routing we have set up so far):
http://localhost:5125/Student
http://localhost:5125/Student/Index

We obtained:

Showing details of student: Lubowitz, Rahsaan

Major: IT

Is veteran: False

GPA: 3.95

Admission date: 1/7/2021 12:00:00 AM

Fig. 7.2 Shows the result of calling the ShowDetails action (we see the view displayed in a browser) when id is set to 20, or no value is provided in the URL request

Showing details of student: Hopper, Aylin

Major: CS

Is veteran: True

GPA: 4

Admission date: 8/15/2022 12:00:00 AM

Fig. 7.3 Shows the page ShowDetails page when id is set to 10

```
Student Controller
TO DO: display a list of students in here
```

Now let's call the `ShowDetails` action. Run both of the following URLs. Can you explain the results in each case (see Fig. 7.2)?

http://localhost:5125/Student/ShowDetails

http://localhost:5125/Student/ShowDetails/20

And for http://localhost:5125/Student/ShowDetails/10 (see Fig. 7.3).

Our web application is very simple, but this example helped us get a quick view of *models*, *views*, and *controllers*. We'll see them in more depth in the upcoming chapters. And starting with Chap. 11, we'll grab/load our data from a database.

7.5 Various Action Result Types

In `StudentController` we currently have two *actions*, one that returns `Content(…)`, and one that returns `View()`. Also, if you look at these two *actions*,

their return type, in both cases, was declared as `IActionResult`. Let's explain this in more depth.

Our actions can have various *return types*. Here is a list of common ones (although we will mostly focus on `ViewResult` for the remainder of this book).

- `ContentResult`: Use this to send responses containing plain text or XML.
  - Note: We used `return Content(…)` and this has the return type `ContentResult`.
- `ViewResult`: Use this to render a `View` as a response (we'll build entire HTML pages).
  - Note: We used `return View(…)` and this has the return type `ViewResult`.
- `RedirectResult`: Use this to redirect a request to a different URL.
  - Note: We used `return Redirect(…)` and this has the return type `RedirectResult`.
- On your own, you may want to check out the following:
  - `JsonResult`: used to send a JSON object as a result.
  - `RedirectToActionResult`: used when we want to redirect to another *action* (from the same or another Controller).
  - `StatusCodeResult`: used when we want to send an HTTP status code as a result.

All these types enumerated here are implementing the `IActionResult` interface. If you hold the *Ctrl* key and while doing so, also click on `IActionResult`, Visual Studio will open the *IActionResult.cs* that contains the following definition of this interface:

```
// Licensed to the .NET Foundation under one or more agreements.
// The .NET Foundation licenses this file to you under the MIT license.

using System.Threading.Tasks;

namespace Microsoft.AspNetCore.Mvc
{
    /// <summary>
    /// Defines a contract that represents the result of an action method.
    /// </summary>
    public interface IActionResult
    {
        /// <summary>
        /// Executes the result operation of the action method asynchronously. This method is
called by MVC to process
        /// the result of an action method.
        /// </summary>
        /// <param name="context">The context in which the result is executed. The context
information includes
        /// information about the action that was executed and request information.</param>
        /// <returns>A task that represents the asynchronous execute operation.</returns>
        Task ExecuteResultAsync(ActionContext context);
    }
}
```

In our project, all *actions* are declaring this return type instead of the class type they are returning. That is, we use.

```
public IActionResult Index()

{
    return Content("Student Controller\nTO DO: display a list of student in here");
}

public IActionResult ShowDetails(int id)
{
    //...
    return View();
}

public IActionResult GoToGoole()
{
    return Redirect("https://www.google.com/");
}
```

which is simpler than using different return types for each *action*:

```
public ContentResult Index()
{
    return Content("Student Controller\nTO DO: display a list of student in here");
}

public ViewResult ShowDetails(int id)
{
    //...
    return View();
}

public RedirectResult GoToGoole()
{
    return Redirect("https://www.google.com/");
}
```

On your own, you may want to add the following actions to the StudentController class and test them:

```
public IActionResult GoToGoole()
{
    return Redirect("https://www.google.com/");
}

public IActionResult AnotherIndex()
{
    return RedirectToAction("Index");
}
```

To learn more about these Action Results, we recommend [53].

7.6 Conventional Versus Attribute Routing

For the remaining chapters of this book, we will mostly use the *default routing;* for this reason, we'll keep this section short. But we thought it would be good for you to know a little more about *routing*, to better understand it. In here we'll compare the *conventional routing* against the *attribute routing*. Check out the main source for this chapter [54].

7.6.1 Conventional Routing

As we've seen above, the *default routing* is equivalent to.

```
app.UseRouting();  //adds route matching to the middleware pipeline

app.MapControllerRoute( //adds default routing
    name: "default",
pattern: "{controller=Home}/{action=Index}/{id?}");
```

In this code, we used the `MapControllerRoute` to add one single route. One can add more than one route (see the example below). Defining routes this way, in one central location, the *Program.cs* file is called **conventional routing**.

We already explained the route given above, but now that we have seen a few examples, this should make more sense. Let's review it once more. The route uses three **path segments**, separated by `/`, and described in the `pattern` parameter. It has the following:

- `{controller=Home}` specifies that the first value will represent a **controller**.
  - A default value of `HomeController` is given.
- `{action=Index}` specifies that the second value will represent an **action**.
  - A default value of `Index` is given.
- `{id?}` specifies that the third value will represent a value for a variable called `id`.
  - We used `id` for **actions** that have a parameter with the name `id`.
  - The `?` means that a value for `id` is optional.

When can add more than one route. In this case, each route has a higher priority for matching than the subsequent ones. Therefore, the order in which routes are declared is very important. If we have URLs that match multiple routes, the first route matching our URL will be used.

Note: There are many ways to declare routes, we'll just give here one more route.

Immediately after the line `UseRouting` already in the *Program.cs* (so right before the default route) let's add the following:

```
app.MapControllerRoute( //adds a second route
    name: "secondroute",
    pattern: "Display/{id?}",
    defaults: new { controller = "Student", action= "ShowDetails" });
```

The `name` parameter is required that must be distinct from the names of the other routes, but it has no impact on URL matching. It is only used internally when URLs are generated.

Next, let's explain the `pattern` parameter given above, then test it. This `pattern` has two parts:

Showing details of student: Hopper, Aylin

Major: CS

Is veteran: True

GPA: 4

Admission date: 8/15/2022 12:00:00 AM

Fig. 7.4 Shows the result of the HTTP request for the ShowDetails action using “secondroute” route

- `Display`—since this is not inside `{}`, URLs must contain the exact word (case insensitive) to match the route.
- `{id?}`—since this contains `{}`, the value will represent an `id`. This segment is optional because we have `?`.

To match this route, your HTTP requests must look like this `/Display` or `/Display/somevalue`. For example,

http://localhost:5125/Display
http://localhost:5125/Display/10
http://localhost:5125/Display/20

Which code (*controller* and *action*) will handle these requests? The `defaults` parameter sets the request to be handled by the `ShowDetails` action from `StudentController`.

In particular, note that the following HTTP request, which is handled by different routes, will yield the same results (see Figs. 7.4 and 7.5):

http://localhost:5125/Display/10
http://localhost:5125/Student/ShowDetails/10.

One can use a “catch-all” route, but we will later use a friendly error page instead. Therefore, for the purposes of this book, you may not want to add it to your routes. We put it here just for completeness (it should be added last in the list of routes):

```
app.MapControllerRoute( //adds a catch-all route
    name: "catch-all",
    pattern: "{*anything}",
    defaults: new { controller = "Home", action = "Index" });
```

Test it with http://localhost:5125/Test/Anything/you/wish.

Showing details of student: Hopper, Aylin

Major: CS

Is veteran: True

GPA: 4

Admission date: 8/15/2022 12:00:00 AM

Fig. 7.5 Shows the same result as Fig. 7.4, but using a different (the default) route

7.6.2 Attribute Routing

After this chapter, we will not be using *attribute routing*. We only include them here for completeness, and because we found them very easy to use, you may use them if you go beyond the concepts covered in this book and learn about Web API. The reason why we don't recommend them for large MVC applications is that such routes are distributed across multiple (controller) files, and hence they can quickly get out of control—especially if you have multiple teams doing development of various controller classes.

Here is a good comparison between conventional routing and attribute routing: "The conventional default route handles routes more succinctly. However, attribute routing allows and requires precise control of which route templates apply to each action" [54].

Let's see some examples (see more in [54]). We'll add attribute routes to the `StudentController` class.

If we run our web application, right now the HomeController, Index action is the default page (see Fig. 7.6):

Let's add the following attribute to our `Index` action: `[Route("")]`. Our action now looks as follows:

```
[Route("")]
public IActionResult Index()
{
    return Content("Student Controller\nTO DO: display a list of student in here");
}
```

Hello World from Index action, HomeController!

Fig. 7.6 Shows the default page for our application is currently coming from the Index action in HomeController

```
Student Controller
TO DO: display a list of student in here
```

Fig. 7.7 Shows the default page for our application is now coming from the Index action in StudentController

Rebuild and run your web application. This made the `Index` action of the `StudentController` the default page look similar to Fig. 7.7.

Let's add a second and a third route to the same *action*. These are essentially alternative routes (one can use either *route* to call our *action*):

```
[Route("")]
[Route("second")]
[Route("third/fourth")]
public IActionResult Index()
{
    return Content("Student Controller\nTO DO: display a list of student in here");
}
```

Now, we can access this page using any of the following URLs:

http://localhost:5125/

http://localhost:5125/second

http://localhost:5125/third/fourth.

IMPORTANT: Following HTTP requests will not get routed to our action from `Student` controller by the default routing:

http://localhost:5125/third/

http://localhost:5125/Student/Index (we'll explain this one in the next subsection).

One can use **token replacement** for *action* and *controller* names. For example, we can add the following attribute routes to our actions inside `StudentController`:

```
public class StudentController : Controller
{
    [Route("TestMe/[controller]/[action]")]
    public IActionResult Index()
    {
        return Content("Student Controller\nTO DO: display a list of student in here");
    }

    [Route("TestMe/[controller]/[action]/{id?}")]
    public IActionResult ShowDetails(int id)
    {
        // ...
```

In these attribute routes we added above, the class name will be used for `[controller]`, and the action name for `[action]`. In particular, to access these actions, we'll need to use URLs as the ones below (see Figs. 7.8 and 7.9):

http://localhost:5125/TestMe/Student/Index

http://localhost:5125/TestMe/Student/ShowDetails/10.

```
Student Controller
TO DO: display a list of student in here
```

Fig. 7.8 Shows how the attribute routing can be used to call on the Index action from StudentController

```
Showing details of student: Hopper, Aylin

Major: CS

Is veteran: True

GPA: 4

Admission date: 8/15/2022 12:00:00 AM
```

Fig. 7.9 Shows how the attribute routing can be used to call on the ShowDetails action from StudentController

Note that the routes have a lot of repeated code. We can improve our code by applying the repetitive part of the route at the class level, so we don't have to copy and paste it for each *action*. Here is how that code would look like (the links above would work the same):

```
[Route("TestMe/[controller]/[action]")]
public class StudentController : Controller
{
    [Route("")]
    public IActionResult Index()
    {
        return Content("Student Controller\nTO DO: display a list of student in here");
    }

    [Route("{id?}")]
    public IActionResult ShowDetails(int id)
    {
        // ...
```

There is much more to say about *attribute routing*, but we won't be using it in our book, so we'll skip the other details. Check out [54] for more examples.

7.6.3 Mixing Routings

We've seen above that we can use both *conventional* and *attribute routing* in the same project.

```
Hello World from Index action, HomeController!
```

Fig. 7.10 Shows you that requesting the Index action from StudentController will not give you the expected results. This is because, in this example, we mixed attribute routing with conventional routing for the same Controller class

When should we use either one? "It's typical to use conventional routes for controllers serving HTML pages for browsers, and attribute routing for controllers serving REST APIs" [54].

IMPORTANT: In the same web application one can use both *attribute* and *conventional routing*. But they should not both apply to the same controller class. "Actions that define attribute routes cannot be reached through the conventional routes" [54].

In particular (see Fig. 7.10), http://localhost:5125/Student/Index will not give you the Index action from the `StudentController`.

For the remaining part of this book, we'll only use conventional routing.

Check for updates

8 More on Controllers and Views, Introduction to Razor Syntax

In this chapter, we'll create a new *model*, and a new *controller*, and learn more about *views*. In particular, we'll introduce in here the *Razor syntax* and some *tag helpers*. By the end of this chapter, you should have a good understanding of *controllers* and be fairly comfortable with *views*. We'll see *views* in more depth in the next chapter.

8.1 A Little Cleanup Before We Continue

Before we continue with this chapter, let's simplify our code. Let's remove all *attribute routes* from our `StudentController` class. If you wish, you may also remove all conventional routes except for the default route.

We make these changes so it becomes easier to debug our project in case we get any errors along the way. You may choose to disregard this, which is fine. Here is how our *Program.cs* file looks like after these changes:

```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddControllersWithViews(); //adds services needed for controllers

var app = builder.Build();//set up middleware components.
app.UseStaticFiles(); //needed to give access to files in wwwroot
app.UseRouting();  //adds route matching to the middleware pipeline
app.MapDefaultControllerRoute(); //adds default routing

app.Run();
```

Next, let's quickly review the main MVC concepts (you should become more familiar with these as we go through this and the next chapter).

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024

R. A. Mezei, *Introduction to the Development of Web Applications Using ASP .Net (Core) MVC*, Synthesis Lectures on Computer Science,
https://doi.org/10.1007/978-3-031-30626-6_8

8.2 Some Essential MVC Concepts and the HTTP Request Lifecycle (Revisited)

The **models** are the classes that represent the various types of objects managed by the web application.

- These objects represent the ***state of the application***.
- For example (think about a web application such as Amazon, Canvas, Moodle, …), we could have the following model classes: Course, User, Student, Instructor, Administrator, Product, Seller, Buyer, and so on.
- Often, model object will use data retrieved from a database (we'll add this capability in Chap. 11).

The **views** will make up the ***user interface***.

- We'll use *views* to build webpages that ultimately will get displayed in a user's browser.
- Typically, *views* will display our *model* data.
- For example (think about a web application such as Amazon, Canvas, Moodle, …), we can have a view that will be used to display a list of all courses taken by a student, or a list of all laptops available to buy. We could use another view to build a page that allows our users to change their passwords.

The **controllers** will handle the user interaction.

- For example (think about a web application such as Amazon, Canvas, Moodle, …). What happens when you click on a button? Or click on a link? Or load the first welcome page? In each of these, your requests will (eventually) be sent to a **Controller** (more specifically to an **Action** from that **Controller**).
- In many cases, the *Controller* will create an instance of a *Model*, then pass it to a *View* to build a page that will eventually show up in the user's browser.

Now let's see again the **HTTP request lifecycle** (see Fig. 8.1). You should have a much better grasp of it since we covered routing in the last chapter. What happens when a user requests a page (either types in a URL, or clicks on a link or button)? Below, let's assume the *default routing* has been set up.

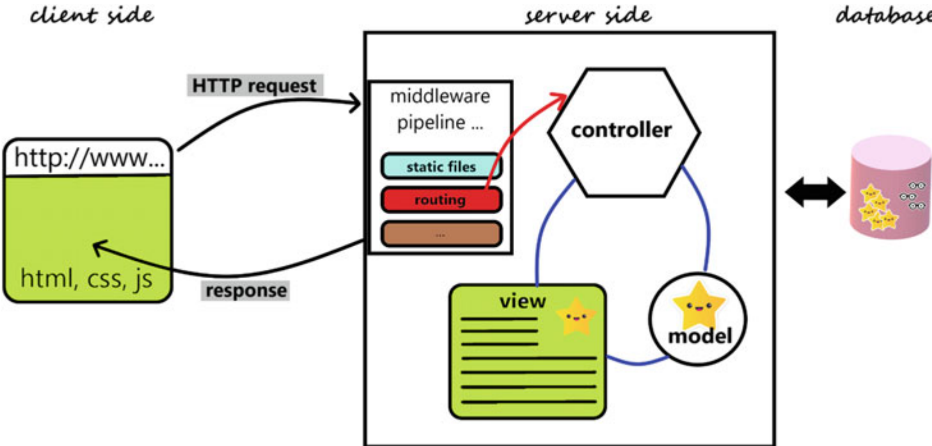


Fig. 8.1 Shows various components of an MVC web application. In particular, note the middleware pipeline, routing, static files, controllers, models, and views

Step 1: The user sends an **HTTP request**.

- For example, the user enters the following URL: http://www.mysite.com/student/show/1.
- For our book, we will run the server from a local host. So instead of http://www.mysite.com we'll use http://localhost:5125.

Step 2: A **controller** object is instantiated to respond to this request.

- The URL **routing** determines which **controller** & **action** will handle the HTTP request.
- In our case, the `StudentController` will be instantiated.

Step 3: An **action** method is then called by the **controller**.

- A Show **action** will be called by the controller.
- A **model binder** helps us determine the values passed to the **action** as parameters (e.g., 1).
- If needed, the **action** may create a new instance of a **model** class. In our example a **`Student`** object.
- Typically, the **action** will pass the model object to a **view** to display the requested results.

Step 4:

- We'll make use of views to produce the output that is sent back to the client's browser.

Notes:

- Typically, we have one controller class for each model class.
  - For example, for the `Student` model, we have `StudentController` that will allow us to `add/edit/delete/display` student data.
  - Another example: for the `User` model, we will have `AccountController` that will allow us to `register/login/logout` users.
  - Yet, for the `HomeController` we will have no model class.
- Each *controller* can have multiple *views*
  - We will see this below, but typically, we'll create one *view* for each *action*.
  - And as seen above, a *controller* may have multiple *actions*.

8.3 Another Example of Model, Controller, and Views

In here we'll add another *model*, *controller*, and *corresponding* views. But this time we'll go in more depth with the MVC. In particular, we'll focus more on *views*, and we'll introduce the *Razor syntax* (which essentially allows us to embed C# code inside *views*).

8.3.1 The Instructor Model

Let's start by adding a new model. For this example, we'll create a new class called `Instructor`. Make sure to add this new class inside the *Models* folder (in the *Solution Explorer* window, right-click on the *Models* folder then select Add > Class…).

Next, we need to choose what characteristics to add to this class. In our example below, we added the following:

- `InstructorId`: An integer that uniquely identifies an instructor (a primary key, if you think from a database perspective).
- `FirstName`: A string that will store an instructor's first name.
- `LastName`: A string that will store an instructor's last name.
- `IsTenured`: A Boolean value that will be set to true if an instructor is tenured.
- `Rank`: A value of enumerated type that will store an instructor's rank.
- `HiringDate`: A DateTime value that will store an instructor's hiring date.

Feel free to add other properties too. The ones we added above should be sufficient to demonstrate the topics we want to cover in this book.

Here is the code we added to *Instructor.cs*:

```
public enum Ranks { Adjunct, Instructor, AssistantProfessor, AssociateProfessor,
};                 FullProfessor

public class Instructor
{
    public int InstructorId { get; set; }
    public string? FirstName { get; set; }
    public string? LastName { get; set; }
    public bool IsTenured { get; set; }
    public Ranks Rank { get; set; }
    public DateTime HiringDate { get; set; }
}
```

We hope that by now you feel very comfortable with creating *model* classes. We'll see more exciting things about them later, but what we have so far should be sufficient for now.

8.3.2 The `InstructorController` Class

8.3.2.1 Adding a New Controller

Let's add a *controller* for the `Instructor` model seen above. As seen earlier in this book, to create a controller one can right-click on *Controllers* folder (in the *Solution Explorer* window), then select *Add* > *Controller*…. Make sure to choose the MVC Controller—Empty option (the other options that show up are MVC Controller with read/write actions and MVC Controller with views, using Entity Framework).

Then click on the *Add* button and choose a name of *InstructorController*. Make sure to double-check your spelling for this class name!

Your newly created file should contain the following contents:

```
using Microsoft.AspNetCore.Mvc;

namespace ASPBookProject.Controllers
{
    public class InstructorController : Controller
    {
        public IActionResult Index()
        {
            return View();
        }
    }
}
```

8.3.2.2 Adding Some Sample Data to Our Controller

Let's add some sample data for our controller, and some hard-coded values. A later chapter will use data from a database instead of these values, but we'll do one step at a time.

Inside the `InstructorController` class (we'll add this right before the action method), let's create a List of Instructors with some hard-coded data. Make sure to add the necessary `using` statement. Here is an example of what we added (feel free to add more data):

```
List<Instructor> InstructorsList = new List<Instructor>()
        {
    new Instructor() {InstructorId = 100,
        FirstName = "Maegan", LastName = "Borer",
        IsTenured=false, HiringDate=DateTime.Parse("2018-08-15"),
        Rank = Ranks.AssistantProfessor},

    new Instructor() {InstructorId = 200,
        FirstName = "Antonietta ", LastName = "Emmerich",
        IsTenured=true, HiringDate=DateTime.Parse("2022-08-15"),
        Rank = Ranks.AssociateProfessor},

    new Instructor() {InstructorId = 300,
        FirstName = "Antonietta", LastName = "Lesch",
        IsTenured=false, HiringDate=DateTime.Parse("2015-01-09"),
        Rank = Ranks.FullProfessor},

    new Instructor() {InstructorId = 400,
        FirstName = "Anjali", LastName = "Jakubowski",
        IsTenured=true, HiringDate=DateTime.Parse("2016-01-10"),
        Rank = Ranks.Adjunct}
};
```

Note: In class we typically ask students to help us provide some sample data. This is more fun this way and it provides another opportunity for students' engagement. Here we used an online sample name generator (see, for example [55]).

8.3.2.3 Adding Actions to Our Controller

When you think of `Instructor` data, what operations/actions would you like to be able to do?

Here are the actions we'll add to our controller class:

- `Index`: Used to display a list of instructors.
  - `ShowAll`: Same as the one above, we'll use this to demonstrate redirect to action
  - `DisplayAll`: Same as above, we'll use this to demonstrate a view with a name
- `Show(int id)`: Used to display all details for one instructor.
- `Add`: Used to create/add a new instructor. As we will see, this is a two-step process (so we'll have two actions).
- `Edit(int id)`: Used to edit an existing instructor. This is also a two-step process (so we'll have two actions).
- `Delete (int id)`: Used to delete an existing instructor. Optionally, this could also be a two-step process.

Let's add the following *actions* to our `InstructorController` (we'll implement each of them below).

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

public IActionResult ShowAll()
{
    return View();
}

public IActionResult DisplayAll()
{
    return View();
}

public IActionResult ShowDetails(int id)
{
    return View();
}

public IActionResult Add()
{
    return View();
}

public IActionResult Edit(int id)
{
    return View();
}

public IActionResult Delete(int id)
{
    return View();
}
```

IMPORTANT: Above that several actions have the same `return View();` statement. As we will see below, they will actually return different views/results. In here we make use of the **convention over configuration**. In particular,

- the `return View();` statement from `Edit` action will return the `Edit` view;
- the `return View();` statement from `Add` action will return the `Add` view.

The view files used are **Razor view** files (also called **Razor-based view templates**). These files have extension *.cshtml* and will contain both C# and HTML code. The **Razor engine** will use the C# and HTML code from a *view* to render the corresponding HTML content (HTML response) that will be sent back to the client who made the request (and will ultimately be displayed in a browser). To read more about views, see also [56].

On a side note, the above actions should remind you of the **CRUD operations**, often seen in a Database course. CRUD stands for the following (see more here: [57]):

- **C**reate operations: Used to create/add new data.
- **R**ead operations: Used to retrieve/search/view existing data.
- **U**pdate operations: Used to update/edit existing data.
- **D**elete operations: Used to delete existing data.

A *controller* often implements the CRUD operations.

If you add a new controller, let's call it `TestController` and select *MVC Controller with read/write access.*

You get the following actions as part of the template (we'll see many of these lines throughout this book, as we add more to our controller classes):

```
public class TestController : Controller
{
    // GET: TestController
    public ActionResult Index()
    {
        return View();
    }

    // GET: TestController/Details/5
    public ActionResult Details(int id)
    {
        return View();
    }

    // GET: TestController/Create
    public ActionResult Create()
    {
        return View();
    }

    // POST: TestController/Create
    [HttpPost]
    [ValidateAntiForgeryToken]
    public ActionResult Create(IFormCollection collection)
    {
        try
        {
            return RedirectToAction(nameof(Index));
        }
        catch
        {
            return View();
        }
    }

    // GET: TestController/Edit/5
    public ActionResult Edit(int id)
    {
        return View();
```

```
        }

        // POST: TestController/Edit/5
        [HttpPost]
        [ValidateAntiForgeryToken]
        public ActionResult Edit(int id, IFormCollection collection)
        {
            try
            {
                return RedirectToAction(nameof(Index));
            }
            catch
            {
                return View();
            }
        }

        // GET: TestController/Delete/5
        public ActionResult Delete(int id)
        {
            return View();
        }

        // POST: TestController/Delete/5
        [HttpPost]
        [ValidateAntiForgeryToken]
        public ActionResult Delete(int id, IFormCollection collection)
        {
            try
            {
                return RedirectToAction(nameof(Index));
            }
            catch
            {
                return View();
            }
        }
    }
```

8.4 The Index Action and View

This section will introduce many new and important concepts. We'll revisit them, in subsequent sections and chapters.

8.4.1 Add a View for Our `Index` Action

We would like to define this *action* to be used for requests that will ultimately display (in the *view*) a list of instructors. For teaching/demonstration purposes, we'll actually use a table instead of a list.

The `Index` action so far looks as follows:

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

As seen in the previous chapter, the easiest way to add a corresponding view is to right-click anywhere in the action and select *Add View* …. Then select the *Razor View* option (for this example do not select *Razor View—Empty* option):

Make sure the name matches the action name (`Index` for us) and make sure all *Options* are unchecked. Then click on the *Add* button.

IMPORTANT: You should note where this newly created *view* was added—it was added inside the *Instructor* folder, which is inside the *Views* folder (see the *Solution Explorer* windows).

All views are under the *Views* folder, inside a subfolder that matches the controller's name.

- All views for the `InstructorController` will be created under the *Views* folder, *Instructor* subfolder.
- All views for the `StudentController` will be created under the *Views* folder, *Student* subfolder.

Here are the contents of the Index view that were added automatically by the View template:

```
@{
    Layout = null;
}

<!DOCTYPE html>

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

Inside the `<body>` element, add the following code:

```
<h1>TO DO: add a list/table of instructors</h1>
```

To see this view, run the application and use the following URL in a browser (see Fig. 8.2):

http://localhost:5125/Instructor/Index

TO DO: add a list/table of instructors

Fig. 8.2 Shows the Index view from InstructorController displayed in a browser

When you use the URL above and press enter to send an *HTTP request* to the server, based on the *default routing* (which we set up in our project), your request will be sent to the `InstructorController`, `Index` action. Based on the code above, the action returns the `Index` view, which contains `<h1> TO DO: add a list/table of instructors </h1>`.

Before we add more code to this view, let's introduce two important topics.

8.4.2 Strongly Typed and Weakly Typed Views

In the previous chapter, we saw how to use the dynamic object `ViewBag` to pass information from an *action* to its *view*. In here we'll see a better (when appropriate) approach, namely *strongly typed* views.

A *view* can be

- **strongly typed**—if it has a `@model` declaration at the top of the view page.
  - The `@model` will declare the type of object this view works with.
  - A view can work with one instance of a model: `@model ASPBookProject.Models.Instructor`
  - A view can work with a collection of instances of a model: `@model IEnumerable <ASPBookProject.Models.Instructor>`
    In this case, we'll use `@foreach` to iterate through the collection
  - Important: You cannot have more than one `@model` directive in a view!
- **dynamically typed**—if it does not have a `@model` declaration at the top of the view's page.
  - We use this type if the view does not work with any model or
  - We use this type if the view needs to work with more than one model.
  - Important: Before using the model inside the view, you'll need to check it is not `null`!

Going back to our example, we would like to pass the `InstructorsList` to the view to display it. For this, we do the following:

- Inside the `Index` action, make sure to pass this as a parameter to the `View` method.
  - Replace `return View();` with `return View(InstructorsList);`

- Inside the `Index` view, at the top of the page, we need to add the `@model` directive (this makes the view strongly typed):
  - @model IEnumerable <ASPBookProject.Models.Instructor>

Here is how the `Index` action (from `InstructorController`) looks after the change:

```
public IActionResult Index()
{
    return View(InstructorsList);
}
```

In the `Index` view, you can delete (if you wish—it will make no difference, we'll explain this later).

```
@{
    Layout = null;
}
```

Here is how our `Index` view looks after the change specified above:

```
@model IEnumerable<ASPBookProject.Models.Instructor>

<!DOCTYPE html>

<html>
<head>

     <meta name="viewport" content="width=device-width" />
     <title>Index</title>
 </head>
 <body>
     <h1>TO DO: add a list/table of instructors</h1>
 </body>
 </html>
```

8.4.3 Introduction to Razor Engine and Razor Syntax

Before we continue, let's have the `Index` action of `InstructorController` be the default action called when we first run the application (essentially the default page). To accomplish this, go to the *Program.cs* and replace the line:

```
app.MapDefaultControllerRoute(); //adds default routing
```

with:

```
app.MapControllerRoute( //modified default routing
    name: "default",
    pattern: "{controller=Instructor}/{action=Index}/{id?}");
```

Now we would like to add code to our `Index` view, so it displays information about all our instructors. As a starting point, let's add the following table inside the body of the *Index* view, right after the `<H1>` element:

```
<TABLE>
    <THEAD>
        <TR>
            <TH>Course ID</TH>
            <TH>Course name</TH>
            <TH>Course link</TH>
        </TR>
    </THEAD>
    <TBODY>
        <TR>
            <TD>CSC200</TD>
            <TD>Object Oriented Programming</TD>
            <TD><a href="https://www.stmartin.edu/">Course link</a></TD>
        </TR>
        <TR>
            <TD>CSC340</TD>
            <TD>Data Structures and Algorithms</TD>
            <TD><a href="https://www.stmartin.edu/">Course link</a></TD>
        </TR>
        <TR>
            <TD>CSC495</TD>
            <TD>ST ASP .Net Core MVC</TD>
            <TD><a href="https://www.stmartin.edu/">Course link</a></TD>
        </TR>
    </TBODY>
</TABLE>
```

Then rebuild your web application and run it. It should display the following default page, similar to Fig. 8.3 (URL: localhost:5125/Instructor/Index):

Next, let's modify this table to use the following table headers: First name, Last name, and Rank. For this, replace the lines:

TO DO: add a list/table of instructors

Course ID	Course name	Course link
CSC200	Object Oriented Programming	Course link
CSC340	Data Structures and Algorithms	Course link
CSC495	ST ASP .Net Core MVC	Course link

Fig. 8.3 Shows the Index view from InstructorController displayed in a browser, now containing a table

```
<TH>Course ID</TH>
<TH>Course name</TH>
<TH>Course link</TH>
```

with

```
<TH>First name</TH>
<TH>Last name</TH>
<TH>Rank</TH>
```

To display all values from the list of instructors, we'll need to use C#. This is where *Razor* comes in handy.

"**Razor** is a markup syntax for embedding .NET based code into webpages" (see more in [58]). Therefore, using *Razor syntax*, we can embed C# code in our (*Razor*) *views*. Then, a mechanism called **Razor Engine** will go through the *view* and run the C# and HTML code, giving us only HTML content which is what we send as a response to the client's request.

Razor uses the `@` symbol to transition from HTML to C#. For the transition from C# back to HTML there is no symbol, Razor will (typically correctly) infer where this needs to be done. These are called **implicit Razor expressions** and an example of it is the following:

```
@DateTime.Now
```

When the implicit expressions aren't correctly interpreting a Razor expression, we can use **explicit Razor expressions** by making use of parentheses `@()`, for example, `@(DateTime.Now - TimeSpan.FromDays(7))`.

For **Razor code block** we use curly braces, `@{}`. We'll see examples below. Just like C#, Razor supports.

- Conditionals: `@if`, `else if`, `else`, and `@switch`. For example (can you guess what it does?),

```
@if (User.Identity.IsAuthenticated) //if the user is logged in
{
    <li class="nav-item">
        <a class="nav-link" asp-action="Logout" asp-controller="Account">Logout</a>
    </li>
}
else
{
    <li class="nav-item">
        <a class="nav-link" asp-action="Login" asp-controller="Account">Login</a>
    </li>
}
```

- Looping: `@for`, `@foreach`, `@while`, and `@do` while. We'll see an example of this below.
- Comments: `@* … *@`.
- C# comments (`//` and `/*…*/`) are also supported.

We'll see more examples as we go through this book. Now let's use this information to build our table of instructors for the `Index` view. Go to the *Index.cshtml* file. Inside the `<TBODY>` element we have three `<TR>` elements (one for each row). We would like to replace those three rows with code that generates one row for each instructor from the `InstructorsList`.

Some general notes:

- The *Index* view is strongly typed: it has the following @model directive:
- `@model IEnumerable<ASPBookProject.Models.Instructor>.`
- The `InstructorsList` was passed from the *action* to this *view*, and inside the *view*, we refer to this list as `Model`.
- IMPORTANT: We use uppercase M in our *view*, except for the `@model` directive where we use lowercase m.
- The `Model` represents the list, and we can use the dot notation to access its values.

Now, replace the `<TBODY>` element (and all its contents) with the following.

```
<TBODY>
    @foreach (var instructor in Model)
    {
        <TR>
            <TD>@instructor.FirstName </TD>
            <TD>@instructor.LastName</TD>
            <TD>@instructor.Rank</TD>
        </TR>
    }
</TBODY>
```

Also replace the `<H1>` element with:

```
<h1>All instructors</h1>
```

Then rebuild your application and run it, check out the results (URL: localhost:5125)—they should look similar to Fig. 8.4.

We'll see several more examples (involving the *Razor syntax*) below, so please be patient.

8.4.4 Action Using a View with a Different Name

We would like to conclude this part with the following two brief examples.

If you want an *action* to use/return a *view* with the same name (for `Index` action, to use the `Index` view), we just used `return View();` and later we used `return View(InstructorsList);`

Fig. 8.4 Shows the updated Index view from InstructorController displayed in a browser

All instructors

First name	Last name	Rank
Maegan	Borer	AssistantProfessor
Antonietta	Emmerich	AssociateProfessor
Antonietta	Lesch	FullProfessor
Anjali	Jakubowski	Adjunct

In both cases, we did not have to specify to use the "Index" view, the compiler just knew to use that view. This is an example of *convention over configuration.*

If instead we want an *action* (say `DisplayAll`) to use a view with a different name (say `Index`), then we must pass the name as the first argument to the `View()` method call. Here is an example (see both actions side by side):

```
public IActionResult Index()
{
    return View(InstructorsList); //will use the Index.cshtml view
}

public IActionResult DisplayAll()
{
    return View("Index", InstructorsList);//will use the Index.cshtml view
}
```

To test this, run http://localhost:5125/Instructor/DisplayAll and this should show a page (Fig. 8.5) similar to what we've seen in Fig. 8.4.

All instructors

First name	Last name	Rank
Maegan	Borer	AssistantProfessor
Antonietta	Emmerich	AssociateProfessor
Antonietta	Lesch	FullProfessor
Anjali	Jakubowski	Adjunct

Fig. 8.5 Same as the image in Fig. 8.4, but we now used a different action to request it (a different URL)—the view is the same. The URL after the request stayed the same as the URL used in the request

All instructors

First name	Last name	Rank
Maegan	Borer	AssistantProfessor
Antonietta	Emmerich	AssociateProfessor
Antonietta	Lesch	FullProfessor
Anjali	Jakubowski	Adjunct

Fig. 8.6 Same as the image in Fig. 8.4, but we now used a different action to request it (a different URL)—the view is the same. The URL after the request has changed (was redirected) to a URL different from the URL used in the request

IMPORTANT: Please note the URL, it shows the *action* being called is `DisplayAll`, although the *view* used is `Index`.

The second example is the following. Modify the `ShowAll` action so its body is as shown below:

```
public IActionResult ShowAll()
{
    return RedirectToAction("Index", InstructorsList);
}
```

To test this, run http://localhost:5125/Instructor/ShowAll

You will get the same result as above (see Fig. 8.6), but now the request was redirected to the `Index` action (note the URL: localhost:5125/?Capacity=4&Count=4):

8.5 The ShowDetails Action and View

Below we'll get to revisit some of the concepts we covered above (namely the *Razor syntax* and *strongly typed views*), then we'll introduce a few new ones. For the `ShowDetails` action, we would like to create a *view* that nicely displays information related to one instance of our model (one instance of `Instructor`).

8.5.1 The ShowDetails Action

What should we expect from this *action*? What should this *action* do? It should allow us to provide an instructor `id`, search (normally in a database) for the instructor that matches this `id`, and as a response, display the instructor.

The ShowDetails *action* needs a parameter that uniquely identifies one Instructor. Inside the Instructor class, that would be the InstructorId property. However, since the *default routing* uses id as one of its segments, it is important that we use id instead of InstructorId as a parameter for the ShowDetails *action*. The reason for this has to do with model binding which we'll cover a little later in this book.

Alternatively, one can also add another route, one that uses a segment containing InstructorId (or whatever name you would like to use for the parameter of the ShowDetails action).

When the ShowDetails action is being called, a value for id is/should be provided (this is the parameter of the action). Then, the action will search for an instance of Instructor from our InstructorsList that has the value of property InstructorId equal to the given Id. For this, we could use the code below (explained next):

```
public IActionResult ShowDetails(int id)
{
    //search for the instructor whose InstructorId matches the given id
    //  here we are using InstructorsList, later we'll use a database!
    Instructor? instr = InstructorsList.FirstOrDefault(ins => ins.InstructorId == id);

    if (instr != null) //was an instructor found?
        return View(instr);
    //if no instructor was found ...
    return NotFound();
}
```

Above, we first used the FirstOrDefault method to search for an instructor whose InstructorId equals id. Note the lambda expression we used to essentially tell the FirstOrDefault method how to search for our instructor. It's a short and elegant statement, but as we'll see later, we'll pretty much use the same code to search in a database (via the Entity Framework Core) instead of a List, so please spend some extra time if needed to understand this statement. The returned value will either be a reference to an instance of Instructor (if one was found for the provided id) or the null reference (if none was found).

If we found an/the Instructor, we'll pass that to a *view* to prepare the client-side code to be displayed in a browser. Otherwise, we'll return the NotFound view that is part of the ASP .Net (we did not implement this ourselves). We'll see it below when we test our ShowDetails view.

Next, we'll implement the ShowDetails *view* for the ShowDetails *action*.

8.5.2 The ShowDetails View

We need to create a view for the ShowDetails action. Right-click anywhere in this action as select *Add View …*, then follow the same steps as seen above to create a *view*.

Since the action is passing an instance of `Instructor` to this *view*, we should make our *view* a *strongly typed view* by using the `model` directive:

```
@model ASPBookProject.Models.Instructor
```

Note: When using the **@model** directive, we needed to use the full class name that has the form: namespace.classname.

8.5.2.1 The @using Directive

Instead of the one line `@model` directive:

```
@model ASPBookProject.Models.Instructor
```

we could use the following two directives:

```
@using ASPBookProject.Models
@model Instructor
```

Similarly, in the `Index` view, we can replace.

```
@model IEnumerable<ASPBookProject.Models.Instructor>
```

with

```
@using ASPBookProject.Models
@model IEnumerable<Instructor>
```

which is easier to read.

8.5.2.2 The *_ViewImports* File

Instead of using the `@using ASPBookProject.Models` directive in every *view* in our project, we have a more elegant solution. We can create a file named *_ViewImports.cshtml* (directly under ***Views*** folder!) and copy this directive in there. Then, we won't need to add that directive to any of our views anymore (and we can remove them from our views).

Let's add the *_ViewImports* file. For this, in the *Solution Explorer* window, right-click on the *Views* folder, then select *Add* > *New Item* ….

Then, select *Razor View Imports* and click on the *Add* button. Make sure the name field contains *_ViewImports.cshtml*.

In this file (*_ViewImports.cshtml*), add the line:

```
@using ASPBookProject.Models
```

Then, you can remove this directive from the *Index* and *ShowDetails* view files. You won't need to add this directive to any views you create from now on in this project.

More specifically, the *Index* view should have no `@using` directive, only the following directive:

```
@model IEnumerable<Instructor>
```

and similarly, for the *ShowDetails* view, it should only have the directive:

```
@model Instructor
```

If you wonder why we have an underscore (_) in the *_ViewImports.cshtml* filename, here is an explanation: “Files in the Views folder whose names begin with an underscore (the _ character) are not returned to the user” (see [5]).

8.5.2.3 Let’s Finish the Implementation for the *ShowDetails* View

In strongly typed views, we can use `@Model` (and the *dot notation*) to access the fields of our model. Here is how one could write a very simple implementation of our *ShowDetails* view (we’ll improve it later):

```
@model Instructor

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Instructor @Model.LastName Details</title>
</head>
<body>
    <h1>Instructor @Model.LastName details</h1>
    <p>First name: @Model.FirstName</p>
    <p>Last name: @Model.LastName</p>
    <p>Is tenured: @Model.IsTenured</p>
    <p>Academic rank: @Model.Rank</p>
    <p>Hiring date: @Model.HiringDate</p>

</body>
</html>
```

Note the use of Razor syntax (we used `@`) to embed our C# code inside HTML. To test it (see Fig. 8.7), use the following two URLs:

http://localhost:5125/Instructor/ShowDetails/200

http://localhost:5125/Instructor/ShowDetails/20

You should get the HTTP ERROR 404—*This localhost page can’t be found.*

Note: We could have checked for a `null` reference inside the *view* (instead of checking for it in the *action*) and provided a more friendly output via the *ShowDetails* view, but we won’t need this. Later we’ll provide links for existing instructors and a friendly error page for HTTP error codes.

Instructor Emmerich details

First name: Antonietta

Last name: Emmerich

Is tenured: True

Academic rank: AssociateProfessor

Hiring date: 8/15/2022 12:00:00 AM

Fig. 8.7 Same as the ShowDetails, for InstructorController, displayed in a browser

8.6 A First Look at Tag Helpers and HTML Helpers

In this section, we would like to add links to our *views*, so we provide an easier navigation for our users. One way to do this is by making use of *HTML helpers* and/or *tag helpers*. They both use the project's routing to generate links; on your own, we challenge you to test this by modifying the routes and see how the links created by these helpers are changed accordingly!

8.6.1 A First Example of an HTML Helper

HTML helpers are essentially C# functions that return (or build) HTML code. Here is an example:

```
@Html.ActionLink("click for details", "ShowDetails", new{id=100})
```

Above, the `Html.ActionLink` function provided three arguments:

- one value for the hyperlink text ("click for details");
- one value for the name of the action to call ("ShowDetails");
- and one (optional) parameter—id—is provided a value (id = 100).

The result of the HTML helper above: it will show up in a browser as click for details.

The HTML code generated by the above *HTML helper*, using default routing, is:

```
<a href="/Instructor/ShowDetails/100">click for details</a>
```

8.6.2 A First Example of a Tag Helper

Instead of *HTML helpers* one can use *tag helpers*. **Tag helpers** "enable server-side code to participate in creating and rendering HTML elements in Razor files" [59]. Tag helpers look very similar to HTML code, and in the author's opinion, they are easier to read as developers. In this book, we will mostly focus on *tag helpers,* but we will occasionally also make use of *HTML helpers.*

We give below the *tag helper* equivalent to the *HTML helper* given above:

```
<a asp-action="ShowDetails" asp-route-id="100">click for details</a>
```

The output and HTML code generated is identical to what the HTML helper gave us. Note how natural this tag helper is. Since we need to create a link, we used the `<A>` element (just like we did in Chap. 3, when we covered HTML). The content of this element ("**`click for details`**") is the text that appears as a link (pretty much what we've seen in Chap. 3). But, thanks to *tag helpers*, the `<A>` element has two "attributes" (which are not HTML, these are part of the tag helpers) we used:

- `asp-action="ShowDetails"`—to denote the action name (`ShowDetails`) we want to use.
- `asp-route-id="100"`—to specify that the `id` part of the route should use the value 100.
  - IMPORTANT: If your route uses a different name instead of `id`, say `instructorId`, then you should use `asp-route-instructorId="100"`

Very important: To use *tag helpers*, you will need to add the following directive:

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

either

- in every view that uses tag helpers, or
- only in one file, namely in *_ViewImports.cshtml* (we used this for our book).

Before you proceed, please add this directive into your *_ViewImports.cshtml*. Here are its contents now:

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

We will see a lot more about tag helpers in the next sections and chapters. But if you're eager to learn more now, we recommend the following Ref. [59].

8.6.3 Add Links to the Index View Using Tag Helpers and HTML Helpers

Let's make use of the helpers introduced above to add links to the table displayed by `Index` view of `Student` controller. To the table defined in *Index.cshtml*, add two more `<TH>` elements:

```
<TH>Details (HTML helper)</TH>
<TH>Details (tag helper)</TH>
```

and two more `<TD>` elements inside the `<TR>` element:

```
<TD>@Html.ActionLink("details", "ShowDetails", new{id=@instructor.InstructorId})</TD>
<TD><a asp-action="ShowDetails" asp-route-id="@instructor.InstructorId">click for details</a> </TD>
```

Here is how the `<TABLE>` element looks like with the changes above:

```
<TABLE>
    <THEAD>
        <TR>
            <TH>First name</TH>
            <TH>Last name</TH>
            <TH>Rank</TH>
            <TH>Details (HTML helper)</TH>
            <TH>Details (tag helper)</TH>
        </TR>
    </THEAD>
    <TBODY>
        @foreach (var instructor in Model)
        {
            <TR>
                <TD>@instructor.FirstName </TD>
                <TD>@instructor.LastName</TD>
                <TD>@instructor.Rank</TD>
                <TD>@Html.ActionLink("details", "ShowDetails", new{id=@instructor.InstructorId})</TD>
                <TD><a asp-action="ShowDetails" asp-route-id="@instructor.InstructorId"> details</a> </TD>
            </TR>
        }
    </TBODY>
</TABLE>
```

Above we used the helpers as seen in the previous subsection, but instead of hardcoding the value 100, we used (via the dot notation) the `InstructorId` property for each instructor included in the table. If you rebuild and run your project, you should get a table (URL: localhost:5125), similar to the one in Fig. 8.8:

If you click on any of those links, you will get to the *ShowDetails* page (see Fig. 8.9) of the corresponding instructor. For example (URL: localhost:5125/Instructor/ShowDetails/200),

8.6.4 Add Bootstrap to the Index View

We will make our web application prettier in a future chapter. For now, focus on the functional part of the application.

All instructors

First name	Last name	Rank	Details (HTML helper)	Details (tag helper)
Maegan	Borer	AssistantProfessor	details	details
Antonietta	Emmerich	AssociateProfessor	details	details
Antonietta	Lesch	FullProfessor	details	det[illegible]
Anjali	Jakubowski	Adjunct	details	details

localhost:5125/Instructor/ShowDetails/200

Fig. 8.8 Shows the Index view for InstructorController displays a table of instructors. Each row in this table contains links to the ShowDetails actions. Note how each link contains the ID of the highlighted Instructor (in our example: 200)

Instructor Emmerich details

First name: Antonietta

Last name: Emmerich

Is tenured: True

Academic rank: AssociateProfessor

Hiring date: 8/15/2022 12:00:00 AM

Fig. 8.9 Shows the ShowDetails view, displayed in a browser, that we obtained by clicking on a link from the Index page

To keep you motivated, we will do one small detour and make our table prettier using Bootstrap 5 tables. If you review the section where we covered Bootstrap 5 tables, then the following should be very familiar to you.

To use Bootstrap5 in our *Index.cshtml* view, we add the following links inside the <HEAD> element, let's say before the <TITLE> element:

```
<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.2.3/dist/css/bootstrap.min.css" rel="stylesheet">
<script src="https://cdn.jsdelivr.net/npm/bootstrap@5.2.3/dist/js/bootstrap.bundle.min.js"></script>
```

All instructors

First name	Last name	Rank	Details (HTML helper)	Details (tag helper)
Maegan	Borer	AssistantProfessor	details	details
Antonietta	Emmerich	AssociateProfessor	details	details
Antonietta	Lesch	FullProfessor	details	details
Anjali	Jakubowski	Adjunct	details	details

localhost:5125/Instructor/ShowDetails/200

Fig. 8.10 This is the same page (Index) as the one shown in Fig. 8.9, but after adding the Bootstrap library and corresponding classes (as described above)

Then, add the following CSS classes to the <TABLE> element: `class="table table-dark table-hover"`. It should look like

```
<TABLE class="table table-dark table-hover">
```

Now, run again your web application. The table should look like the one in Fig. 8.10:

For now, please have patience, we'll make our web application look prettier when we get to introduce layouts (in Chap. 12). This way we'll be more efficient because we will minimize redundant work (otherwise we would duplicate code in multiple places).

8.6.5 Add Links to the *ShowDetails* View

We finish this section with one more example of a *tag helper* and an *html helper* used to generate links. Add the following lines right before the end tag of the <BODY> element inside the *ShowDetails* view:

```
<a asp-action="Index">go to Index</a>
@Html.ActionLink("go to Index", "Index")
```

Now, running your web application and clicking on a link to show the details of any instructor, you should see two identical links, one created using a tag helper and one using an HTML helper. Clicking on any of them will take you to *Index*.

For example, for the link with URL: localhost:5125/Instructor/ShowDetails/200 we obtained Fig. 8.11:

In the next chapter, we'll develop the *actions* and *views* for the following operations: `Add`, `Edit`, and `Delete`.

Instructor Emmerich details

First name: Antonietta

Last name: Emmerich

Is tenured: True

Academic rank: AssociateProfessor

Hiring date: 8/15/2022 12:00:00 AM

go to Index go to Index

Fig. 8.11 Shows the ShowDetails view, as seen in Fig. 8.9, but with two hyperlinks added (as described above)

More on Views, Data Annotations

9

In this chapter, we'll develop the *actions* and *views* for the following operations: `Add`, `Edit`, and `Delete`. Along the way, we'll also introduce *Data Annotations*, the *Model Binder*, *Model Validation*, and some *HTTP Verb Attributes*.

IMPORTANT: Before we proceed, make sure your *model* classes use *properties*, not *fields*! Otherwise, you may get unexpected errors or behaviors.

9.1 Introduction to Data Annotations

We'll start with some simple examples of *data annotations* that probably won't seem to be very useful, but as we introduce more *data annotations*, you'll find them very useful and very powerful. They are well worth your time and we'll make extensive use of them, so please invest your time in understanding them.

Data annotations "are used to define metadata for ASP.NET MVC and ASP.NET data controls" (see [60]). In particular, HTML helpers and tag helpers work very nicely with Data Annotations (as we will see below).

9.1.1 Update the ShowDetails View

First, let's modify the *ShowDetails* view. In the `<BODY>` element, replace the following lines:

```
<p>First name: @Model.FirstName</p>
<p>Last name: @Model.LastName</p>
<p>Is tenured: @Model.IsTenured</p>
<p>Academic rank: @Model.Rank</p>
<p>Hiring date: @Model.HiringDate</p>
```

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024

R. A. Mezei, *Introduction to the Development of Web Applications Using ASP .Net (Core) MVC*, Synthesis Lectures on Computer Science,
https://doi.org/10.1007/978-3-031-30626-6_9

with

```
<p><label asp-for="@Model.FirstName"></label>: @Model.FirstName</p>
<p><label asp-for="@Model.LastName"></label>: @Model.LastName</p>
<p><label asp-for="@Model.IsTenured"></label>: @Model.IsTenured</p>
<p><label asp-for="@Model.Rank"></label>: @Model.Rank</p>
<p><label asp-for="@Model.HiringDate"></label>: @Model.HiringDate</p>
```

If we rebuild and run our project, and go to the *ShowDetails* page for any of our instructors, we'll see something similar to Fig. 9.1 (URL: localhost:5125/Instructor/ShowDetails/200).

This doesn't look like an improvement (yet!), especially since the labels displayed do not contain spaces between words. For example, it now shows "FirstName" instead of "First Name". How can we fix this? We cannot have spaces in variable names, for example, the following property name is not valid (see Fig. 9.2).

To fix the spacing in the displayed property name, we will use *data annotations*. They are very powerful, and we'll soon see why. But for now, let's use an easy data annotation, namely the **display data annotation**. If you go to the `Instructor` class definition, we

Instructor Emmerich details

FirstName: Antonietta

LastName: Emmerich

IsTenured: True

Rank: AssociateProfessor

HiringDate: 8/15/2022 12:00:00 AM

go to Index go to Index

Fig. 9.1 This looks similar to Fig. 8.11, for now. To build it, we now used tag helpers

```
public string? First Name { get; set; }
```

Fig. 9.2 Shows how creating a property with name containing spaces ("First Name") will lead to compilation errors

can add display data annotations to each of our properties, so instead of the property names, we can display any text we want. For this, change the code in *Instructor.cs* from.

```
public int InstructorId { get; set; }
public string? FirstName { get; set; }
public string? LastName { get; set; }
public bool IsTenured { get; set; }
public Ranks Rank { get; set; }
public DateTime HiringDate { get; set; }
```

to

```
public int InstructorId { get; set; }

[Display(Name = "First name")]
public string? FirstName { get; set; }

[Display(Name = "Last name")]
public string? LastName { get; set; }

[Display(Name = "Is tenured")]
public bool IsTenured { get; set; }

[Display(Name = "Academic rank")]
public Ranks Rank { get; set; }

[Display(Name = "Hiring date")]
public DateTime HiringDate { get; set; }
```

To use data annotations, we also need to include the following `using` directive:

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

Notes:

- One can add multiple data annotations for the same property, and
- some properties may have no data annotations.

A data annotation applies to the very next property that follows the data annotation.

With these changes, now the *ShowDetails* looks a little better (see Fig. 9.3).

What did we get out of this section? The most important part for now is that we briefly introduced some display data annotations. Also, we have seen how we can use tag helpers to display the name of the properties in our views, and how using **display data annotations**, we can instead display any text we wish in place of property names. This is very powerful in case we later want to change how a property is being displayed—we only need to change the value in the display data annotation, without having to change the name of the property, which means the code will still compile and work, but the displayed value will be different. Just imagine changing a property from a class that is being used in many files—you would need to change all those files; using data annotations, we did not change any property name, we just changed how they can be displayed.

Instructor Emmerich details

First name: Antonietta

Last name: Emmerich

Is tenured: True

Academic rank: AssociateProfessor

Hiring date: 8/15/2022 12:00:00 AM

go to Index go to Index

Fig. 9.3 Shows the same view as in Fig. 9.1, and it also uses tag helpers. Using data annotations, however, we were able to override what is being displayed for each property (for example, we displayed "Academic Rank" for the property named Rank)

In the above example, you may notice that the `enum` value of "AssociateProfessor" isn't very nice looking. We can use similar display data annotations for our enumerated type too. In *Instuctor.cs* replace.

```
public enum Ranks { Adjunct, Instructor, AssistantProfessor, AssociateProfessor, FullProfessor
};
```

with

```
public enum Ranks { Adjunct,
    Instructor,
    [Display(Name = "Assistant Professor")] AssistantProfessor,
    [Display(Name = "Associate Professor")] AssociateProfessor,
    [Display(Name = "Full Professor")] FullProfessor
};
```

IMPORTANT: To see annotations being displayed we cannot just use the property name (as in `@Model.FirstName`), we need to use *tag helpers/html helpers* to make use of these display data annotations. Above we used a *tag helper* with the `<label>` element to display property names. Below we'll make changes (we can add *HTML helpers* or *tag helpers*) to display property values too.

For our example, please replace the lines in *ShowDetails.cshtml*:

```
<p><label asp-for="@Model.FirstName"></label>: @Model.FirstName</p>
<p><label asp-for="@Model.LastName"></label>: @Model.LastName</p>
<p><label asp-for="@Model.IsTenured"></label>: @Model.IsTenured</p>
<p><label asp-for="@Model.Rank"></label>: @Model.Rank</p>
<p><label asp-for="@Model.HiringDate"></label>: @Model.HiringDate</p>
```

with

```
<p><label asp-for="@Model.FirstName"></label>:  @Html.DisplayFor(m => m.FirstName)</p>
<p><label asp-for="@Model.LastName"></label>:  @Html.DisplayFor(m => m.LastName)</p>
<p><label asp-for="@Model.IsTenured"></label>: @Html.DisplayFor(m => m.IsTenured)</p>
<p><label asp-for="@Model.Rank"></label>: @Html.DisplayFor(m => m.Rank)</p>
<p><label asp-for="@Model.HiringDate"></label>: @Html.DisplayFor(m => m.HiringDate)</p>
```

Using these helpers, we ensured that the values displayed will make use of the Data Annotations in our code. Above we used the `DisplayFor` *Html helper.* In order to use this HTML helper, we provided it with a lambda expression specifying which property we wanted to display. Rebuild and rerun your application, and see that the displayed enum value has been fixed (see Fig. 9.4).

You should also note that the *html helper* (and similarly for *tag helpers*) made use of the fact that `IsTenured` was declared as a Boolean property. Because of it, it displayed its value as a checkbox (*checked* for `true`, *unchecked* for `false`). We'll see more of this in the next sections.

One last fix to do in here. In terms of hiring date, no one really records the time, just the date. So we should only display the date portion and omit the time portion of our DateTime property: `HiringDate`. To accomplish this, go to the *Instructor.cs* file, and add the following *data annotation* right before the `HiringDate` property:

```
[DataType(DataType.Date)]
```

So, this property now looks like

Fig. 9.4 Shows the same view as in Fig. 9.3 but now it is making use of the HTML helper DisplayFor. In particular, note how a Boolean property such as IsTenured was displayed as a checkbox

Fig. 9.5 Shows the same view as in Fig. 9.4 but now the DateTime properly, named HiringDate, only shows a Date (no Time)

```
[Display(Name = "Hiring date")]
[DataType(DataType.Date)]
public DateTime HiringDate { get; set; }
```

Rebuild and run your application. Here (see Fig. 9.5) is what we obtained now (much better!).

9.1.2 Update the Index View (Optional)

Let's update the `Index` view so that it uses *tag helpers* and the *display data annotations* as seen above. Also, we should use an `if` statement to check if the list is empty (this is different than `null`!). When the list is empty, we should display some specific text, such as "No instructors found!". To do this, add the following code to *Index.cshtml* and move the `<TABLE>` element inside the if block (Note: to use C#, we used Razor syntax):

```
@if(Model.Count()>0)
{
    //put the table element in here
}
else
{
    <h2>No instructors found! </h2>
}
```

Next, we would like to use *tag helpers* similar to what we have seen in the previous subsection, to use the data annotations for the names of the table's columns.

Challenge: Now the `model` reference is not pointing to one instance of `Instructor`, but a list of Instructors, so our syntax will be a little different. We give below the entire *Index.cshtml* file so you can also check your work:

```
@model IEnumerable<Instructor>
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.2.3/dist/css/bootstrap.min.css" rel="stylesheet">
    <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.2.3/dist/js/bootstrap.bundle.min.js"></script>
    <title>Index</title>
</head>
<body>
    <h1>All instructors</h1>

    @if(Model.Count()>0)
    {
        <TABLE class="table table-dark table-hover">
            <THEAD>
                <TR>
                    <TH><label asp-for="First().FirstName"></label></TH>
                    <TH><label asp-for="First().LastName"></label></TH>
                    <TH><label asp-for="First().Rank"></label></TH>
                    <TH>Details (HTML helper)</TH>
                    <TH>Details (tag helper)</TH>
                </TR>
            </THEAD>
            <TBODY>
                @foreach (var instructor in Model)
                {
                    <TR>
                        <TD>@instructor.FirstName </TD>
                        <TD>@instructor.LastName</TD>
                        <TD>@instructor.Rank</TD>
                        <TD>@Html.ActionLink("details", "ShowDetails", new{id=@instructor.InstructorId})</TD>
                        <TD><a asp-action="ShowDetails" asp-route-id="@instructor.InstructorId">details</a></TD>
                    </TR>
                }
            </TBODY>
        </TABLE>
    }
    else
    {
        <h2>No instructors found!</h2>
    }
</body>
</html>
```

As an exercise, we want to let you figure out how to use the HTML helpers seen in the previous subsection (namely DisplayFor) so the Academic rank will make use of the data annotations set in *Instructor.cs*. Your table should look similar to Fig. 9.6.

Here is one solution:

All instructors

First name	Last name	Academic rank	Details (HTML helper)	Details (tag helper)
Maegan	Borer	Assistant Professor	details	details
Antonietta	Emmerich	Associate Professor	details	details
Antonietta	Lesch	Full Professor	details	details
Anjali	Jakubowski	Adjunct	details	details

Fig. 9.6 Shows the same view as in Fig. 8.6 but it should now be using HTLM helpers

```
<TD>@Html.DisplayFor(m => instructor.FirstName) </TD>
<TD>@Html.DisplayFor(m => instructor.LastName)</TD>
<TD>@Html.DisplayFor(m => instructor.Rank)</TD>
```

9.2 The Add Action and View

This section is very important as it introduces several new important concepts, and it may look a little more challenging than it actually is. Most concepts will be reviewed again in the next section and you will probably have a much better understanding then. Please have patience.

As we will see below, the `Add` operation is actually a two-step process and we'll need to create two *Add* actions:

- one for the GET operation that will be used to send a request for a form to fill out, and
- one for the POST operation that we will use to send all data from the form to the server.

9.2.1 The Add Action—GET

Let's start easy. First, in the `InstructorController`, let's add an `Add` action (if you already have this, do not add it again!), as shown below.

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

Note: To access this action easily, we next add a link to this *action* in the `Index` view (this is what we first see when we run our application). In the *Index.cshtml*, right before the `</BODY>` tag, add the following link to our `Add` action (it's a tag helper!):

```
<a asp-action="Add">Add a new instructor </a>
```

Now, let's run the application. You should get a new link at the end of the Index page (see Fig. 9.7).

If you click on that link, you will get an error message ("InvalidOperationException: The view 'Add' was not found".), which should make sense, since we did not yet create a *view* for the `Add` action (try it!).

Now let's add a view. Then ask yourself, what do you expect to get when you click on that link? That's what we'll put in the *Add* view. Just like we've seen earlier, make sure the view name matched the action name, and no options are selected.

All instructors

First name	Last name	Academic rank	Details (HTML helper)	Details (tag helper)
Maegan	Borer	Assistant Professor	details	details
Antonietta	Emmerich	Associate Professor	details	details
Antonietta	Lesch	Full Professor	details	details
Anjali	Jakubowski	Adjunct	details	details

Add a new instructor

Fig. 9.7 Shows the same view as in Fig. 9.6 but it should include a new link, Add a new instructor, that would point to the Add action of InstructorController

Now, if we try again to click on the "Add a new instructor" link, we should get this newly added view, which is empty.

Let's add contents to this *view*.

9.2.2 The Add View

9.2.2.1 Add a Form Inside the View

Inside the *Add.cshtml* file, we should create a form that allows the user to enter the data needed to create a new instance of `Instructor`. We've seen how to create forms at the beginning of this book (in Chap. 3), and now we'll make use of that knowledge and make use of *tag helpers*.

Inside the `Add` view, inside the `<BODY>` element add the following code (on your own also change the `<TITLE>` element):

```
<h1>Create a new Instructor</h1>
<form asp-action="Add" asp-controller="Instructor" method="post">
    @* add form contents in here ...  *@
    <input type="submit" value="Create Instructor"/>
</form>
```

You should note that in the above we made use of the *tag helpers* to specify where should the data/request be sent to (namely the `Add` action from `InstructorController`) when the user clicks on the submit button (the button will display the text: `Create Instructor`). Here, in Fig. 9.8 is what we get so far if you click on the "Add a new instructor" link (URL: localhost:5125/Instructor/Add).

We'll explain the reason why we set the `method="post"` below.

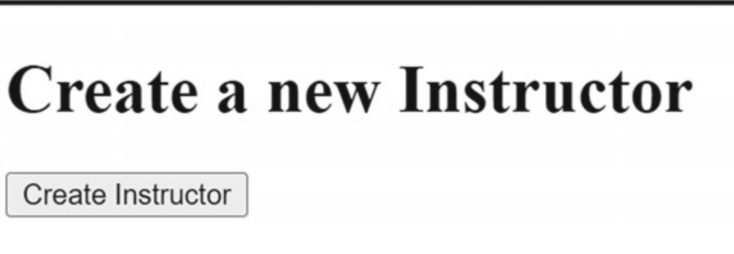


Fig. 9.8 Shows the Add view. It contains an H1 element and a form that only contains the submit button

Before we continue, we need to make our view strongly typed. This is because the form will work with an instance of Instructor, and also to get Model Binder support (see below). Make sure to add the following at the beginning of the *Add.cshtml* file:

```
@model Instructor
```

9.2.2.2 Populate the Form with Input Elements

Now let's add input elements so the user can enter data needed for our new `Instructor` instance. Replace the code `@* add form contents in here... *@` with the following lines (this should look similar to what we've seen in the previous sections):

```
<label asp-for="InstructorId"></label>   <input asp-for="InstructorId" />   <br>
<label asp-for="FirstName"></label>   <input asp-for="FirstName" />   <br>
<label asp-for="LastName"></label>   <input asp-for="LastName" />   <br>
<label asp-for="IsTenured"></label>   <input asp-for="IsTenured" />   <br>
<label asp-for="HiringDate"></label>   <input asp-for="HiringDate" />   <br>
<label asp-for="Rank"></label>   <input asp-for="Rank" />   <br>
<br />
```

In particular, you should note that we used *tag helpers* for both:

- To display the names of the properties (via data annotations), we used the following tag helper of the form:
  - `<label asp-for="PropertyName"></label>`
- To create an input field for the properties (again via data annotations), we used
  - `<input asp-for="PropertyName" />`

Running your application again, the form now (see Fig. 9.9) looks much better (do not bother fixing `InstructorId`, we'll remove it in Chap. 11).

Because we used *tag helpers*, Razor engine was able to figure out that.

- `IsTenured` is a Boolean value, hence the corresponding "input" element can be displayed as a **checkbox**.
  - `<input asp-for = "IsTenured" />`

Fig. 9.9 This is the same as Fig. 9.8, but the form now includes several labels and input elements (as described above)

- `HiringDate` is `DateTime` (and the display data annotation declared this as a `Date`), hence the "input" element can be displayed as **date** selector.
  - `<input asp-for = "HiringDate" />`

Can you see how powerful tag helpers and data annotations are?

9.2.2.3 Create a Dropdown List for Properties of an Enum Type

IMPORTANT: The *Academic rank* (`Rank` property) wasn't properly displayed. We would like to get a dropdown list instead of input box (which is currently being displayed). To accomplish this, replace.

```
<input asp-for="Rank" />
```

with

```
<select asp-for="Rank" asp-items="@Html.GetEnumSelectList(typeof(Ranks))"></select>
```

Run again your application and check that you obtained a result similar to Fig. 9.10. For completeness, we provide you below all the contents of the `Add` view:

Fig. 9.10 Is similar to Fig. 9.9, but now a dropdown menu was added for the Academic rank option. Note that the first option is automatically selected

```
@model Instructor

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Create a new Instructor</title>
</head>
<body>
    <h1>Create a new Instructor</h1>
    <form asp-action="Add" asp-controller="Instructor" method="post">
        <label asp-for="InstructorId"></label>
        <input asp-for="InstructorId" />
        <br>
        <label asp-for="FirstName"></label>
        <input asp-for="FirstName" />
        <br>
        <label asp-for="LastName"></label>
        <input asp-for="LastName" />
        <br>
        <label asp-for="IsTenured"></label>
        <input asp-for="IsTenured" />
        <br>
        <label asp-for="HiringDate"></label>
        <input asp-for="HiringDate" />
        <br>
        <label asp-for="Rank"></label>
        <select asp-for="Rank" asp-items="@Html.GetEnumSelectList(typeof(Ranks))"></select>
<br>
        <br />
        <input type="submit" value="Create Instructor" />
    </form>

</body>
</html>
```

One more fix for our dropdown list: We would like to not have `Adjunct` selected by default. To fix this, we'll add the following content for the `<SELECT>` element above:

```
<option value="">Select</option>
```

That is, if you replace:

```
<select asp-for="Rank" asp-items="@Html.GetEnumSelectList(typeof(Ranks))"> </select>
```

with:

```
<select asp-for="Rank" asp-items="@Html.GetEnumSelectList(typeof(Ranks))">
   <option value="">Select</option>
</select>
```

then your `Rank` selector will not have a default value selected (see Fig. 9.11).

Rebuild and run again the application and fill in data into the form included in the `Add` view. What happens when you click on the *Create Instructor* (the submit) button? Per our code above, the submit button will send your data, as a POST request, to the `Add` action (as of right now that is the same action as we created above). We would like it to go to another action. For reason we'll see later (when we do validation) we would like our second action to also be called `Add`. Let's create a second action, this one with a parameter of type `Instructor` (our model class).

Fig. 9.11 Is similar to Fig. 9.10, but note that a default value is not preselected for the Academic rank

9.2.3 The Add Action—POST

9.2.3.1 The Need for HttpGet and HttpPost Attributes

Add the following new *action* in the *InstructorController.cs* file (we added it right after the previously added `Add` action):

```
public IActionResult Add(Instructor newInstructor)
{
    return View();
}
```

This code compiles, because method overloading is allowed, but it will introduce a challenge for the routing. Run your application again and click on the "*Add a new instructor …*" link. You will get the following error: `AmbiguousMatchException: The request matched multiple endpoints.`

To fix this, we can add the following **HTTP VERB Attributes**:

`[HttpGet]`—for the first Add action and

`[HttpPost]`—for the second Add action.

You should now have

```
[HttpGet]
public IActionResult Add()
{
    return View();
}

[HttpPost]
public IActionResult Add(Instructor newInstructor)
{
    return View();
}
```

These attributes essentially restrict what type of requests their respective *actions* will respond to.

Using [HttpPost] for an *action*, we are restricting that *action* to only respond to POST requests.

- Typically, when you click on a link or type in a URL in the browser, you will make a GET request.
- Typically, when you click on submit button for a form, or upload a file to send to the server, you will make a POST request.

Using these attributes will take care of the ambiguity that the routing complained about. The code should not compile without any errors.

9.2.3.2 Add Action (POST) Implementation, Introduction to Model Binding

Let's finish implementing the second `Add` *action*. We'll explain why it works the way it works below. Modify this action so it contains the code shown below:

```
[HttpPost]
public IActionResult Add(Instructor newInstructor)
{
    InstructorsList.Add(newInstructor); //add the new instructor to our list
    return View("Index", InstructorsList); //temporary fix - do not refresh the page!!!
}
```

Very important: The *Add* method above has one parameter of type **`Instructor`**. We can choose any parameter type we need for an action, but in this case, we chose `Instructor` because we have a very helpful mechanism, called the **model binding**, which is helping us behind the scenes. Namely, when this second *Add* action is called, the **model binding** will.

- look to find the information needed for our *action*—in our case, we need an instance of `Instructor`;
- look at the information sent to the server—in our case, it will look into the data sent by the form (and other sources too); and finally
- with the values entered in the form create an instance of the `Instructor` and pass it along to our *Add* action.

We'll see *Model Binding* again below, and it will make more sense in there. This is merely an introduction.

Let's test our code and see the steps involved to add a new instructor. Let's explain this step by step:

- First, we click on the *Add a new instructor* link from the Index page.
- This will send an HTTP GET request to the first *Add* action which in turn returns a view that contains a form for the user to interact with (see Fig. 9.12).
- The user can fill in the form as desired (see Fig. 9.13).

And when they click on the submit button (Create Instructor), a second HTTP request (a POST request because the form included the following in the `<FROM>` tag: `method="post")` is being sent, and this one will go to the second `Add` action (because the second one is the one that responds to POST request).

That second action will add the new Instructor object created by the model binding and send this action to the `InstructorsList` and then returns the `Index` view, which displays (see Fig. 9.14) the instructors (note the URL: localhost:5125/Instructor/Add).

We used `return View` instead of `return RedirectToAction` because this did not involve a new request being sent to the server. The bad part is that the URL (look in the screenshot above) will be localhost:5125/Instructor/Add. We did this "temporary fix" because our data is not persistent. As soon as you go to localhost:5125 (or navigate via any links on the page), you will lose this newly added instructor (see Fig. 9.15).

We'll deal with data persistency later.

Create a new Instructor

InstructorId
First name
Last name
Is tenured
Hiring date mm/dd/yyyy
Academic rank Select

Create Instructor

Fig. 9.12 The Add view for adding a new Instructor. It contains an H1 element, a form with multiple labels, input elements, and a submit button

Create a new Instructor

InstructorId 900
First name Ines
Last name Lindgren
Is tenured
Hiring date 12/28/2022
Academic rank Associate Professor

Create Instructor

Fig. 9.13 This is the same view as the one seen in Fig. 9.13, but it contains some user entered values

All instructors

First name	Last name	Academic rank	Details (HTML helper)	Details (tag helper)
Maegan	Borer	Assistant Professor	details	details
Antonietta	Emmerich	Associate Professor	details	details
Antonietta	Lesch	Full Professor	details	details
Anjali	Jakubowski	Adjunct	details	details
Ines	Lindgren	Associate Professor	details	details

Add a new instructor

Fig. 9.14 Shows the Index view for the InstructorController. It displays a table containing a list of instructors' details

All instructors

First name	Last name	Academic rank	Details (HTML helper)	Details (tag helper)
Maegan	Borer	Assistant Professor	details	details
Antonietta	Emmerich	Associate Professor	details	details
Antonietta	Lesch	Full Professor	details	details
Anjali	Jakubowski	Adjunct	details	details

Add a new instructor

Fig. 9.15 Shows the same information as shown in Fig. 9.14, but the newly added row is not there anymore (it wasn't saved)

9.2.4 A Few More Details About the Model Binding

In the example seen in the previous section, the *model binding system* was able to collect various data from our form and turn them into an instance of the `Instructor` needed for the `Add` action.

In general, the **model binding system** can retrieve data from various sources, in this order:

- Form fields.
- Route data.
- Query string parameters.
- Uploaded files.

The **model binding system** can provide this data to controllers and even **convert** the various string data to.Net types.

To learn more about model binding in ASP .Net Core, check out the following source [61].

We finish this subsection with one more example. Suppose you have an action that looks like.

```
public IActionResult SomeAction(int id, bool isPriority)
```

If your client is making an HTTP request that has the form:
http://mysite.com/Home/SomeAction/70?ISPRIORITY=true

Then *model binding system* will do the following for us:

- Convert the string 70 which is given as part of the HTTP request into the integer 70 needed for the action.
- Convert the string true which is given as part of the HTTP request into the Boolean true needed for the action.
- It will even match the ISPRIORITY given in the HTTP request to isPriority needed for the action.

9.2.5 A Few More Details About the GET Versus POST

We've briefly seen `GET` and `POST` (called **HTTP verbs**) earlier. They are both used to send client information to a web server and are the most common. But there are some important differences between them.

In class, we like to demonstrate the examples shown in:

- https://www.w3schools.com/html/tryit.asp?filename=tryhtml_form_get
- https://www.w3schools.com/html/tryit.asp?filename=tryhtml_form_post

GET requests

- can be cached;
- can be seen in the browser history;
- can be bookmarked;
- should not be used when dealing with sensitive data (such as username and password);
- have length restrictions;
- data is visible in the URL!;
- only ASCII characters are allowed for the data;
  - we typically use them to send request for forms;
  - when clicking on web links, we send GET requests.

POST requests

- do not remain in the browser history;
- cannot be bookmarked;
- have no restrictions on data length;
- data is not visible in the URL!;
  - we typically use them to send data from forms (but we can sometimes also use GET requests);
  - we must use POST requests to send files to a server.

There are other HTTP verbs, such as PUT, HEAD, DELETE, and PATH, and you can even define your own. But we won't make use of them in this book. To read more about them, check out the following resource [13].

Above, we've seen that `Add` was a two-step process. We had the following:

- Step 1: Add—the GET request: This was used to send a request to the server and in return get a view that contains a form so the user can type in their data.
  - This is when the user clicks a link (and later a button).
- Step 2: Add—the POST request: This was used to send the data entered into the form to the server.
  - This is when the user clicks on the submit button.

Let's briefly discuss other examples.

The login process (we'll implement this in a later chapter).

- Step 1: Login—the GET request: This sends a request to the server to get a form so the user can type in their username and password.
- Step 2: Login—the POST request: When the user clicks on the submit button, the data is sent to the server for processing (verify the correct login credentials and send back some authentication cookies).

The Edit action (we'll implement this below).

- Step 1: Edit—the GET request: This sends a request to the server to get a form loaded with existing data.
- Step 2: Edit—the POST request: When the user clicks on the submit button, the data is sent to the server for processing (save changes to the server).

9.3 The Edit Action and View

In this section, we'll implement the Edit action. As discussed above, we'll have two Edit actions:

- One for the GET request that will return a form loaded with existing data, and
- one for the POST request that will allow the user to save changes to the database.

For simplicity, we'll assume that only some fields are editable. In particular, we'll assume the InstructorId is not editable.

9.3.1 The Edit Action—GET

We start with the following. We need an action that allows us to specify an Id. With this Id in hand, we'll search in the InstructorsList (later we'll search in a database) to find an instance of Instructor whose InstructorId matches the given Id.

If such an Instructor was found, send it to the view to prepopulate a form (to allow the user to change its values).

If such an Instructor was not found, use the NotFound method.

Since we will have two actions with the same name, we will again use the [HttpGet] attribute which will limit this action to only respond to GET requests. Replace the following in *InstructorController.cs* file:

```
public IActionResult Edit(int id)
{
    return View();
}
```

with

```
[HttpGet]
public IActionResult Edit(int id)
{
    //we should look for the instance that has the given Id
    //   ... later we'll search in the database
    Instructor? instr = InstructorsList.FirstOrDefault(inst => inst.InstructorId == id);

    if (instr != null) //if found, send it to the view
        return View(instr);
    //if no matching instructor was found ...
    return NotFound();
}
```

9.3.2 Add Edit Links in the Index View

To make it easier for us to test the Edit functionality, let's add an Edit link to each entry in our table from the Index view.

All instructors

First name	Last name	Academic rank	Details (HTML helper)	Details (tag helper)	Edit
Maegan	Borer	Assistant Professor	details	details	edit this
Antonietta	Emmerich	Associate Professor	details	details	edit this
Antonietta	Lesch	Full Professor	details	details	edit this
Anjali	Jakubowski	Adjunct	details	details	edit this

Add a new instructor

localhost:5125/Instructor/Edit/300

Fig. 9.16 Is similar to Fig. 9.15, but a new column (the Edit) is being added. This column contains links that can be used to call the Edit action for each row/Instructor. In particular, note that these links contain the corresponding ID for each instructor (in our example, we can see the ID 300 being used)

Add a new <TH> entry:

```
<TH>Edit </TH>
```

And a new <TD> entry:

```
<TD><a asp-action="Edit" asp-route-id="@instructor.InstructorId">edit this</a> </TD>
```

When you run your application, you should now have new links, edit links, one for each row in the table from the Index view (see Fig. 9.16).

IMPORTANT: As you hover your mouse over an "edit this" link, make sure the link shown in the lower left part of the webpage has a link that includes the `InstructorId`. In our example, we see that 300 is included in localhost:5125/Instructor/Edit/300 .

To pass this `Id` to our *edit this* link, we include the following in the tag helper used above (where `id` is the third segment used in our modified default routing set in *Program.cs*):

```
asp-route-id="@instructor.InstructorId"
```

If you click on any of the edit links above, you will see an error. This is because we did not yet implement the `Edit` *view* for the `Edit` *action*. Let's implement it next.

9.3.3 The Edit View

Anywhere in the `Edit` (the GET) action, right-click and select *Add View* .... Follow the steps similar as above to create a view with the name `Edit`. Since this view will work with an instance of the `Instructor` class, we'll make this view *strongly typed* by adding the following at the very beginning of the *Edit.cshtml* file:

```
@model Instructor
```

Next, inside the <BODY> element we'll add an <H1> element and a <FORM> element containing just some select fields from `Instructor` class. Namely, we omitted the `FirstName` field, assuming we don't want to allow this to be editable:

```
<h1>Edit an instructor profile</h1>
<form asp-action="Edit" method="post">
    <input asp-for="@Model.InstructorId" type="hidden" />
    @*needed so the InstructorId is sent to the Edit(POST)*@

    <label asp-for="LastName"></label>
    <input asp-for="LastName" />
    <br>
    <label asp-for="IsTenured"></label>
    <input asp-for="IsTenured" />
    <br>
    <label asp-for="HiringDate"></label>
    <input asp-for="HiringDate" />
    <br>
    <label asp-for="Rank"></label>
    <select asp-for="Rank" asp-items="@Html.GetEnumSelectList(typeof(Ranks))"></select>
    <br>
    <br />
    <input type="submit" value="Save changes" />
    <input type="button" onclick="history.back()" value="Cancel">
</form>
```

Above, we included a **"go back"** button that will act as a `Cancel` button:

```
<input type="button" onclick="history.back()" value="Cancel">
```

IMPORTANT: Above, we had to include a field containing the `InstructorId`. This value is needed by the `Edit` action (the POST) to decide which instructor to update. The model binding will find it if we include it in the request sent to the server when the user clicks on the submit button (Save changes button). Since the user does not need to see it, we'll have it hidden from the user's view.

Let's run the web application to see what we got so far (see Fig. 9.17). When you click on the `edit this` link, you send a request to the server. Namely, you will send a GET request to the server (to the `Edit` action—the GET) along with the `Id` of the instructor you want to modify.

All instructors

First name	Last name	Academic rank	Details (HTML helper)	Details (tag helper)	Edit
Maegan	Borer	Assistant Professor	details	details	edit this
Antonietta	Emmerich	Associate Professor	details	details	edit this
Antonietta	Lesch	Full Professor	details	details	edit this
Anjali	Jakubowski	Adjunct	details	details	edit this

Add a new instructor

localhost:5125/Instructor/Edit/200

Fig. 9.17 Is similar to Fig. 9.18

Fig. 9.18 Displays the Edit view for InstructorController. It contains an H1 element and a form that is prepopulated with the current Instructor data

The way we wrote our code, the `Edit` action will search for the `Instructor` from the `InstructorsList` that has `InstructorId` equal to the provided `Id`. Once found, it will pass that to the `Edit` view to display it in a form. And this form (loaded with the values of the `Instructor` found above) is what we're getting next, in Fig. 9.18 (note the URL: localhost:5125/Instructor/Edit/200).

The `Cancel` will take you back to the previous page.

The `Save` changes button will send a `POST` request to the `Edit` action (which we'll implement next).

9.3.4 The Edit Action—POST

Next, we will implement the `Edit` action that will respond to POST requests from the form above. For this we will create a method/action in *InstructorController.cs* that looks like:

```
[HttpPost]
public IActionResult Edit(Instructor instructorChanges)
{
    //to do ...
}
```

You should note that we again will make use of the *model binding system*, by using a parameter of type `Instructor`. The *model binding system* will search for values needed to create an instance on `Instructor`, in particular it will collect the values from the form (seen in the `Edit` view) and create an instance of `Instructor` and pass it to the `instructorChanges` parameter.

The effect is that the `instructorChanges` parameter contains the values in the form at the time the user clicks on the submit button.

Next, we need to use these values to change the `Instructor` in the `InstructorsList` to match these new values. For this, use the code below to replace the `//to do` … comment inside the `Edit` (the POST!) action:

```
//find the instructor from InstructorList
//   who has the same InstructorId as the changes.InstructorId
Instructor? instr = InstructorsList.FirstOrDefault(instr => instr.InstructorId ==
instructorChanges.InstructorId);

if (instr != null) //if found, change the values in InstructorsList to match the changes
{
    instr.LastName = instructorChanges.LastName;
    instr.IsTenured = instructorChanges.IsTenured;
    instr.HiringDate = instructorChanges.HiringDate;
    instr.Rank = instructorChanges.Rank;
}
return View("Index", InstructorsList); //temporary fix - do not refresh the page!!!
```

If we change some or all values from the form (for example, to look like the ones below in Fig. 9.19).

And click on the Save changes (the form's submit button), this will send a POST request to the `Edit` action to save these changes. In our example, we got the following (Fig. 9.20).

It looks like we successfully changed the values from the `InstructorsList`.

The actions are working as expected, but there is a problem. Every time you click on a link (or a button), you are sending a new HTTP request to the server. This in turn means that we are creating a new instance of the controller, so all our changes (add or edit, for example) are lost. Our data is not persistent! Test it, click on http://localhost:5125/—all changes are lost!

Fig. 9.19 Is similar to Fig. 9.18, but it has some values changed

All instructors

First name	Last name	Academic rank	Details (HTML helper)	Details (tag helper)	Edit
Maegan	Borer	Assistant Professor	details	details	edit this
Antonietta	Willms III	Full Professor	details	details	edit this
Antonietta	Lesch	Full Professor	details	details	edit this
Anjali	Jakubowski	Adjunct	details	details	edit this

Add a new instructor

Fig. 9.20 Shows the Index view that contains a table with Instructors information. In particular, it appears that the Edit operation was successful

One fix for this is to use a *database*, which will see in a future chapter. Until then, a "temporary fix" would be to use a service. This "temporary fix" is not very useful in practice, but we want to use it as a reason for us to show you how to create a *service*—for teaching purposes, we think this "temporary fix" is very useful even if it may not make complete sense.

9.3.5 An Example of a Service

As mentioned above, this "temporary fix" isn't very useful in practice, but it will give us a chance to talk about *services* and *dependency injection*. We'll see these topics again in future chapters, so we wanted to use this section as an introduction to what *services* are and how to use *dependency injection*.

As of right now, our hard-coded data for instructors is essentially put in the constructor of the controller. That's a problem because each time a new HTTP request is received, a new instance of the controller is created, meaning that all changes done to the `InstructorsList` are lost. One "temporary fix" is to create a *service* and make that service only be created once for the lifetime of the web application. That will ensure that our data in `InstructorsList` will "survive" multiple HTTP requests. As long as we do not restart the web application, the data in `InstructorsList` will appear to be persistent. In Chap. 11, we'll create real persistency by making use of a *database*.

There are three steps involved when creating new *services*.

9.3.5.1 Create a Class and an Interface

The first step is to create an *interface* (in here you should include the methods and properties that you want others to access), and a *class* that implements that interface.

To be organized, we'll create a folder called *Services* (if you already have this folder, then just use it). We'll include our class and interface source files inside this folder. If you right-click on this folder, you can choose to *Add* > *New Item …*.

There you have many options, including a *class* and an *interface* (choose both, one by one). We'll call our interface IMyFakeDataService, and our class MyFakeDataService.

For the **interface**, for this example, we only need one property, let's call it InstructorsList. Here is a sample code:

```
using ASPBookProject.Models;

namespace ASPBookProject.Services
{
    public interface IMyFakeDataService
    {
        List<Instructor> InstructorsList { get; }
    }
}
```

For the class, we will make it implement the interface defined above, and in the constructor for this class we'll add our hard-coded data for instructors (copy and paste code from the InstructorController class):

```
using ASPBookProject.Models;

namespace ASPBookProject.Services
{
    public class MyFakeDataService : IMyFakeDataService
    {
        public List<Instructor> InstructorsList { get; }

        public MyFakeDataService()//constructor
        {
            InstructorsList = new List<Instructor>()
            {
                new Instructor() {InstructorId = 100,
                    FirstName = "Maegan", LastName = "Borer",
                    IsTenured=false, HiringDate=DateTime.Parse("2018-08-15"),
                    Rank = Ranks.AssistantProfessor},

                new Instructor() {InstructorId = 200,
                    FirstName = "Antonietta ", LastName = "Emmerich",
                    IsTenured=true, HiringDate=DateTime.Parse("2022-08-15"),
                    Rank = Ranks.AssociateProfessor},

                new Instructor() {InstructorId = 300,
                    FirstName = "Antonietta", LastName = "Lesch",
                    IsTenured=false, HiringDate=DateTime.Parse("2015-01-09"),
                    Rank = Ranks.FullProfessor},

                new Instructor() {InstructorId = 400,
                    FirstName = "Anjali", LastName = "Jakubowski",
                    IsTenured=true, HiringDate=DateTime.Parse("2016-01-10"),
                    Rank = Ranks.Adjunct}
            };
        }
    }
}
```

9.3.5.2 Register Your Service

The code above is just defining an *interface*, and a *class* that implements that interface. Nothing special. To make the code above a ***service***, one needs to **register it as a service**. To do this, in *Program.cs* we add the following line:

```
builder.Services.AddSingleton<IMyFakeDataService,
MyFakeDataService>();
```

We must add this line before we call the `builder.Build();` method (so the first part of the *Program.cs* file will look similar to):

```
using ASPBookProject.Services;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddSingleton<IMyFakeDataService, MyFakeDataService>(); //our data service
builder.Services.AddControllersWithViews(); //adds services needed for controllers

var app = builder.Build();//set up middleware components.
app.UseStaticFiles(); //needed to give access to files in wwwroot
```

That's it. Now we have a **service**. The *dependency injection* will take care of creating a new instance of this class when needed. And since we used the `AddSingleton` method, only one instance will be created as long as the web application is not restarted.

9.3.5.3 Inject Your Service and Make Use of It

Now, finally, the using part. To use the service in various parts of our web application, we'll make use of a mechanism called **service injection**. Please get familiar with this because we'll use this again in future chapters.

In particular, we would like to use this service (it's data actually) inside a controller (`InstructorController.cs` for our example).

Let's first do some cleanup. Inside InstructorController.cs, delete the following code:

```
List<Instructor> InstructorsList = new List<Instructor>()
        {
            new Instructor() {InstructorId = 100,
                FirstName = "Maegan", LastName = "Borer",
                IsTenured=false, HiringDate=DateTime.Parse("2018-08-15"),
                Rank = Ranks.AssistantProfessor},

            new Instructor() {InstructorId = 200,
                FirstName = "Antonietta ", LastName = "Emmerich",
                IsTenured=true, HiringDate=DateTime.Parse("2022-08-15"),
                Rank = Ranks.AssociateProfessor},

            new Instructor() {InstructorId = 300,
                FirstName = "Antonietta", LastName = "Lesch",
                IsTenured=false, HiringDate=DateTime.Parse("2015-01-09"),
                Rank = Ranks.FullProfessor},

            new Instructor() {InstructorId = 400,
                FirstName = "Anjali", LastName = "Jakubowski",
                IsTenured=true, HiringDate=DateTime.Parse("2016-01-10"),
                Rank = Ranks.Adjunct}
        };
```

How to Inject a Service?

Now, let's **inject** the *service* into our *controller*. To do this, we need to.

Part 1: Define a (private) field of type `IMyFakeDataService`. We'll access our fake data through this field:

```
private readonly IMyFakeDataService _fakeData;
```

Part 2: Add a parameter to the constructor that will be used to set the field defined above. The "magic" of dependency injection is that the Dependency Injection will automatically set the value for your parameter when the constructor is being called:

```
public InstructorController(IMyFakeDataService fakeData)
{
    _fakeData = fakeData;
}
```

How to Use the Service?

Using the service is easy, we'll just make use of the private field defined in step 1.

In all the actions defined in *InstructorController.cs*, replace all occurrences of `InstructorsList` with `_fakeData.InstructorsList`.

For your reference, here is how the first part of the `InstructorController` class looks after making the changes mentioned above:

```
using ASPBookProject.Models;
using ASPBookProject.Services;
using Microsoft.AspNetCore.Mvc;

namespace ASPBookProject.Controllers
{
    public class InstructorController : Controller
    {
        private readonly IMyFakeDataService _fakeData;

        public InstructorController(IMyFakeDataService fakeData)
        {
            _fakeData = fakeData;
        }


        public IActionResult Index()
        {
            return View(_fakeData.InstructorsList); //will use the Index.cshtml view
        }

        public IActionResult DisplayAll()
        {
            return View("Index", _fakeData.InstructorsList);//will use the Index.cshtml view
        }


        public IActionResult ShowAll()
        {
            return RedirectToAction("Index", _fakeData.InstructorsList);
        }


        public IActionResult ShowDetails(int id)
        {
            //search for the instructor whose InstructorId matches the given id
            //  here we are using InstructorsList, later we'll use a database!
            Instructor? instr = _fakeData.InstructorsList.FirstOrDefault(ins => ins.InstructorId == id);

            if (instr != null) //was an instructor found?
                return View(instr);
             //if no instructor was found ...
             return NotFound();
         }


         [HttpGet]
         public IActionResult Add()

        ...
```

Some "Cosmetics"

Now that we have some "temporary fix" for our persistency problem, replace the two lines that have the code (from the `Add` and `Edit`—POST actions defined in *InstructorController.cs*):

```
return View("Index", _fakeData.InstructorsList); //temporary fix - do not refresh the page!!!
```

With

```
return RedirectToAction("Index");
```

This will make the URL look nicer after we finish adding/editing an instructor. The URL will be the one for the `Index` action. Also, the two actions have a little cleaner code.

Test your code. Now you should be able to add multiple new instructors, perform multiple changes, and see them all in the `Index` view. As long as you don't restart the application, these changes seem to be persistent.

9.4 The Delete Action and View

Finally, we got to the last CRUD operation, the *delete*. This operation can be done in one step or two. Below we'll follow the previously mentioned two-step process.

9.4.1 The Delete Action—GET

We need an *action* that allows us to specify an `Id`. With this `Id` in hand, we'll search in our `InstructorsList` (later we'll search in our database) to find an instance of `Instructor` whose `InstructorId` matches the given `Id`.

- If such an `Instructor` was found, send it to the view to prepare the information to be displayed in a browser.
- If such an `Instructor` was not found, use the `NotFound` method.

We will again use the `[HttpGet]` attribute which will limit this action to only respond to `GET` requests. Make sure your *InstructorController.cs* file contains the following action:

```
[HttpGet]
public IActionResult Delete(int id)
{
    //we should look for the instance that has the given Id
    //   ... later we'll search in the database
    Instructor? instr = _fakeData.InstructorsList.FirstOrDefault(inst => inst.InstructorId == id);

    if (instr != null) //if found, send it to the view
        return View(instr);
    //if no instructor was found ...
    return NotFound();

}
```

9.4.2 Add Delete Links in the Index View

To make it easier for us to test the Delete functionality, let's add a `Delete` link to each entry in our table from the `Index` view.

Add a new <TH> entry:

```
<TH>Delete </TH>
```

And a new <TD> entry:

```
<TD><a asp-action="Delete" asp-route-id="@instructor.InstructorId">delete this</a> </TD>
```

9.4.3 The Delete View

Let's add a *view* for our `Delete` (the GET) *action*. We'll make this view *strongly typed* by adding the following at the very beginning of the *Delete.cshtml* file:

```
@model Instructor
```

The view should ask for a confirmation ("Are you sure …?") and contain two buttons, one for *yes, delete*, and one for *no, cancel*. In order to send a POST request from the button, we'll put it inside a form. As above, we must include a hidden field containing the ID so this value can be sent to the server. Here is a possible <BODY> element for our view.

Next, inside the <BODY> element we'll add an <H1> element and a <FORM> element containing just some select fields from `Instructor` class. Namely, we omitted the `FirstName` field, assuming we don't want to allow this to be editable:

```
<body>
    <h1>Are you sure you want to delete this instructor?</h1>
    <form asp-action="DeleteConfirmed" method="post">
        <input asp-for="InstructorId" type="hidden" />
        <p><label asp-for="@Model.FirstName"></label>:  @Html.DisplayFor(m => m.FirstName)</p>
        <p><label asp-for="@Model.LastName"></label>:  @Html.DisplayFor(m => m.LastName)</p>
        <p><label asp-for="@Model.Rank"></label>: @Html.DisplayFor(m => m.Rank)</p>
        <br />
        <input type="submit" value="YES, delete" />
        <input type="button" onclick="history.back()" value="NO, cancel">
    </form>
</body>
```

Notice again how we included the hidden field containing the `InstructorId`. This is needed for the DeleteConfirmed action (the POST). The model binding will find it and use it if the called action (`DeleteConfirmed`) needs it.

9.4.4 The DeleteConfirmed Action—POST

Next, we will implement the `DeleteConfirmed` action that will respond to POST requests from the form above.

First, let us explain the name. The second `Delete` action does not need an entire `Instructor` parameter, and it only needs an int parameter (the `Id`). The problem is that in C# we cannot have two `Delete` methods both having one `int` parameter. It doesn't matter we have one marked as `HttpGet` and the other `HttpPost`.

There are two solutions:

- either use a different parameter type (for example, we could use Instructor type—just like we did for Edit and Add)
- or use different action names (we chose in here to do this—named our second action `DeleteConfirmed`).

For this example, we will create a method that looks like

```
[HttpPost]
public IActionResult DeleteConfirmed(int instructorId)
{
    //we should look for the instance that has the given Id
    //   ... later we'll search in the database
    Instructor? instr = _fakeData.InstructorsList.FirstOrDefault(inst => inst.InstructorId ==
instructorId);

    if (instr != null) //if found, delete it from list
    {
        _fakeData.InstructorsList.Remove(instr);
        return RedirectToAction("Index");
    }
    //if no instructor was found ...
    return NotFound();
}
```

Let's test this. When you run the web application, you should get the following table (see Fig. 9.21).

All instructors

First name	Last name	Academic rank	Details (HTML helper)	Details (tag helper)	Edit	Delete
Maegan	Borer	Assistant Professor	details	details	edit this	delete this
Antonietta	Emmerich	Associate Professor	details	details	edit this	delete this
Antonietta	Lesch	Full Professor	details	details	edit this	delete this
Anjali	Jakubowski	Adjunct	details	details	edit this	delete this

Add a new instructor

localhost:5125/Instructor/Delete/300

Fig. 9.21 Is similar to the table shown in Fig. 9.20, but a new column (the Delete column) was added. This column contains links that can be used to send Delete requests for each of the Instructors in the list. Note that each link contains the ID of the currently selected instructor (in the screenshot above, the ID = 300 is being shown)

If you click on any `delete this` link, you will send a GET request to the `Delete` action. This action returns a view (see Fig. 9.22) that displays the following (URL: localhost:5125/Instructor/Delete/300).

If you click on the `NO, cancel` button, you'll be taken back to the previous page.

If you click on the `YES, delete` button, you'll send a POST request to the `DeleteConfirmed` action, because this is what we used in the tag helper above:

```
<form asp-action="DeleteConfirmed" method="post">
```

Are you sure you want to delete this instructor?

First name: Antonietta

Last name: Lesch

Academic rank: Full Professor

YES, delete | NO, cancel

Fig. 9.22 Shows the Delete view displayed in a browser. In particular, this displays information about the current instructor to be deleted, and it includes two buttons

All instructors

First name	Last name	Academic rank	Details (HTML helper)	Details (tag helper)	Edit	Delete
Maegan	Borer	Assistant Professor	details	details	edit this	delete this
Antonietta	Emmerich	Associate Professor	details	details	edit this	delete this
Anjali	Jakubowski	Adjunct	details	details	edit this	delete this

Add a new instructor

Fig. 9.23 Shows the Index view, displayed in a browser, with one entry/row removed from the table

Fig. 9.24 Shows the Index view, displayed in a browser, when all rows are deleted from the table. In particular, note that the table is not displayed (since it's empty). Instead, the "No instructors found!" is being displayed.

This action will delete the selected Instructor. Now the table looks like (see Fig. 9.23).

On your own, go ahead and delete all instructors. You should get the following (Fig. 9.24).

Which part of the code is responsible for creating this outcome?

10 Model Validation

Before we continue, let's add a few more properties to the `Instructor` class. This will allow us to better demonstrate some concepts. Let's add the following *properties* and *data annotations* to *Instructor.cs* file:

```
[Display(Name = "Office phone number")]
public String? PhoneNumber{ get; set; }

[Display(Name = "Email address")]
public String? EmailAddress { get; set; }

[Display(Name = "Personal webpage")]
public String? PersonalURL { get; set; }

[Display(Name = "Password (we won't use this!)")]
[DataType(DataType.Password)]
public string? UnusedPassword { get; set; }
```

Then, add the corresponding code to the `Add` view (inside *Add.cshtml* file, right before the "submit" button):

```
<label asp-for="PhoneNumber"></label>
<input asp-for="PhoneNumber" />
<br>
<label asp-for="EmailAddress"></label>
<input asp-for="EmailAddress" />
<br>
<label asp-for="PersonalURL"></label>
<input asp-for="PersonalURL" />
<br>
<label asp-for="UnusedPassword"></label>
<input asp-for="UnusedPassword" />
<br>
<br>
<br>
```

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024

R. A. Mezei, *Introduction to the Development of Web Applications Using ASP .Net (Core) MVC*, Synthesis Lectures on Computer Science,
https://doi.org/10.1007/978-3-031-30626-6_10

Create a new Instructor

InstructorId 900
First name Wilhelmine
Last name Watsica
Is tenured
Hiring date mm/dd/yyyy
Academic rank Select
Office phone number two three five
Email address my email
Personal webpage personal page
Password (we won't use this!) ••••••
Create Instructor

Fig. 10.1 Shows the Add view for InstructorController. It contains a form, and several input fields have values in it

If you run your application and go to `add` a new `Instructor`, you should note the following (see Fig. 10.1):

- The password is nicely hidden from plain view;
- If you press the `Create Instructor` button, it will take/save your answers, even though they are not valid.
  - The value entered for "my email" is not a valid email address
  - The value "two three five" is not a valid phone number format.

You can even submit an empty form (multiple times too!), as shown in Fig. 10.2.

Here is the result (see Fig. 10.3).

We should not accept any random data; we should at least enforce some validation. This is what we'll see next. On the server side, we will add some **model validation**. The

Create a new Instructor

InstructorId

First name

Last name

Is tenured

Hiring date mm/dd/yyyy

Academic rank Select

Office phone number

Email address

Personal webpage

Password (we won't use this!)

Create Instructor

Fig. 10.2 Is the same as Fig. 10.1, but no data has been entered in the form

All instructors

First name	Last name	Academic rank	Details (HTML helper)	Details (tag helper)	Edit	Delete
Maegan	Borer	Assistant Professor	details	details	edit this	delete this
Antonietta	Emmerich	Associate Professor	details	details	edit this	delete this
Antonietta	Lesch	Full Professor	details	details	edit this	delete this
Anjali	Jakubowski	Adjunct	details	details	edit this	delete this
Wilhelmine	Watsica	Adjunct	details	details	edit this	delete this
		Adjunct	details	details	edit this	delete this
		Adjunct	details	details	edit this	delete this

Add a new instructor

Fig. 10.3 Shows the Index view, for InstructorController. In it, note that the last two rows are the result of submitting empty forms to the server:

main reference for this page is [62] and we encourage you to look over it. There are three main steps to follow:

- Make use of **built-in validation attributes** (we'll also see how to build our own **custom** validation attributes).
- **Enforce validation** by making use of the `ModelState`.
- Add **validation helpers** to display error messages.

10.1 Step 1: Add (Built-in or Custom) Validation Attributes

There are several built-in validation attributes that we can use. Here is a short list (see more in [62]).

- `[EmailAddress]` checks that the property has a valid email format.
- `[Phone]` checks that the property has a valid telephone number format.
- `[Range]` checks that the property value is within a given range.
- `[RegularExpression]` checks that the property value matches a given regular expression.
- `[Required]` checks that the field is not null.
- `[StringLength]` checks that the field does not exceed a given max length. It also accepts a `MinimumLength` value.
- `[Url]` checks that the property has a proper URL format.

Let's use some of these in our web application.

- We will add `[Required]` to fields that must be given a value by the user. We will occasionally make use of the `ErrorMessage` property to specify a custom error message (otherwise a default error message will be used).
- We will also make use of the `[Url]` and `[EmailAddress]` attributes to check for us that the values entered have a valid format. Note: since the properties using these attributes are not `[Required]`, empty values will pass the validation check.
- For the phone number, we could use `[Phone]`, but we opted to use the `[RegularExpression]` instead just for practice purposes.

Here is how the `Instructor` class looks like once we added the *built-in validation attributes*:

```
public class Instructor
{
    [Required]
    public int InstructorId { get; set; }

    [Display(Name = "First name")]
    public string? FirstName { get; set; }

    [Required(ErrorMessage = "last name is required")]
    [Display(Name = "Last name")]
    public string? LastName { get; set; }

    [Display(Name = "Is tenured")]
    public bool IsTenured { get; set; }

    [Required]
    [Display(Name = "Academic rank")]
    public Ranks Rank { get; set; }


    [Display(Name = "Hiring date")]
    [DataType(DataType.Date)]
    public DateTime HiringDate { get; set; }

    [RegularExpression("[0-9]{3}-[0-9]{3}-[0-9]{4}",
                       ErrorMessage = "you must follow the format 000-000-0000!")]
    [Display(Name = "Office phone number")]
    public String? PhoneNumber { get; set; }

    [EmailAddress]
    [Display(Name = "Email address")]
    public String? EmailAddress { get; set; }

    [Url]
    [Display(Name = "Personal webpage")]
    public String? PersonalURL { get; set; }

    [Required]
    [StringLength(10, MinimumLength = 5)]
    [Display(Name = "Password (we won't use this!)")]
    [DataType(DataType.Password)]
    public string? UnusedPassword { get; set; }
}
```

10.2 Step 2: Enforce Validation by Making Use of the ModelState

We'll show you next how to use `ModelState` to enforce the *validation attributes* we added above. "For web apps, it's the app's responsibility to inspect `ModelState.IsValid` and react appropriately" [62].

Where do we want to validate user data? We want to do this when we add or edit an instructor. Therefore, we'll make changes in the `Add` action and `Edit` action (POST).

Change `Add` action (inside *InstructorController.cs*):

```
[HttpPost]
public IActionResult Add(Instructor newInstructor)
{
    _fakeData.InstructorsList.Add(newInstructor); //add the new instructor to our list
    return RedirectToAction("Index");
}
```

into

```
[HttpPost]
public IActionResult Add(Instructor newInstructor)
{
    if (!ModelState.IsValid) //if the data is invalid
        return View(); //go back to the view

    _fakeData.InstructorsList.Add(newInstructor); //add the new instructor to our list
    return RedirectToAction("Index");
}
```

Some explanation. Before we add any new data into our `InstructorsList`, we want to make sure it is valid. So, what we did above is as follows: if the data is not valid (we check the status of the data by checking the value of the `ModelState.IsValid`) then return `View`. What view will it be?

Remember "convention over configuration": in the line above, `View()` means the view with the same name as the action, so in this case it will be the `Add.cshtml`. This is where it was useful to use the same name for both the POST and the GET `Add` actions: they both have the same name, so the `View` in both cases will be the same one containing the `Add` form.

Similarly, change the `Edit` action to include the same check as above. After change, your `Edit` (POST) should look as follows:

```
public IActionResult Edit(Instructor instructorChanges)
{
    if (!ModelState.IsValid) //if the data is invalid
        return View(); //go back to the view

    //find the instructor from InstructorList
    //   who has the same InstructorId as the changes.InstructorId
    Instructor? instr = _fakeData.InstructorsList.FirstOrDefault(instr => instr.InstructorId
== instructorChanges.InstructorId);

    if (instr != null) //if found, change the values in InstructorsList to match the changes
    {
        instr.LastName = instructorChanges.LastName;
        instr.IsTenured = instructorChanges.IsTenured;
        instr.HiringDate = instructorChanges.HiringDate;
        instr.Rank = instructorChanges.Rank;
    }
    return RedirectToAction("Index");
}
```

10.3 Step 3: Display Error Messages via Validation Tag Helpers

There are two types of error messages we can display. We can display

- **in-line error messages**—display an error message right next to the input element where the error occurs;
- **summary of all error messages**—display all error messages combined in one place.

10.3.1 To Display a Summary of All Error Messages

Use the following tag helper to display all error messages in one place. You can add this wherever you want, for example, at the beginning of the page (or alternatively at the end of the page):

```
<div asp-validation-summary="All"></div>
```

We will add this in both the *Add.cshtml* and in the *Edit.cshtml*. We'll add this line right before the <FORM> element.

10.3.2 To Display In-line Error Messages

One can also add display validation error messages for each property individually. For this, we will add tag helpers that will look like

```
<span asp-validation-for="InstructorId"></span>
```

Add this for all fields (and change **InstructorId** with the property name in each case).

Here is how the *Edit.cshtml* looks like after the above-mentioned changes:

```
@model Instructor

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Edit</title>
</head>
<body>
    <h1>Edit an instructor profile</h1>
    <div asp-validation-summary="All"></div>
    <form asp-action="Edit" method="post">
        <input asp-for="@Model.InstructorId" type="hidden" /> @*needed so the InstructorId is
sent to the Edit(POST)*@

        <label asp-for="LastName"></label>
        <input asp-for="LastName" />
        <span asp-validation-for="LastName"></span>
        <br>
        <label asp-for="IsTenured"></label>
        <input asp-for="IsTenured" />
        <span asp-validation-for="IsTenured"></span>
        <br>
        <label asp-for="HiringDate"></label>
        <input asp-for="HiringDate" />
        <span asp-validation-for="HiringDate"></span>
        <br>
        <label asp-for="Rank"></label>
        <select asp-for="Rank" asp-items="@Html.GetEnumSelectList(typeof(Ranks))"></select>
        <br>
        <br />
        <input type="submit" value="Save changes" />
        <input type="button" onclick="history.back()" value="Cancel">
    </form>
</body>
</html>
```

Similarly, the *Add.cshtml* should be as follows:

```
@model Instructor

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Create a new Instructor</title>
</head>
<body>
    <h1>Create a new Instructor</h1>
    <div asp-validation-summary="All"></div>

    <form asp-action="Add" asp-controller="Instructor" method="post">
        <label asp-for="InstructorId"></label>
        <input asp-for="InstructorId" />
        <span asp-validation-for="InstructorId"></span>
        <br>
        <label asp-for="FirstName"></label>
        <input asp-for="FirstName" />
        <span asp-validation-for="FirstName"></span>
        <br>
        <label asp-for="LastName"></label>
        <input asp-for="LastName" />
        <span asp-validation-for="LastName"></span>
        <br>
        <label asp-for="IsTenured"></label>
        <input asp-for="IsTenured" />
        <span asp-validation-for="IsTenured"></span>
        <br>
        <label asp-for="HiringDate"></label>
        <input asp-for="HiringDate" />
        <span asp-validation-for="HiringDate"></span>
        <br>
        <label asp-for="Rank"></label>
        <select asp-for="Rank" asp-items="@Html.GetEnumSelectList(typeof(Ranks))">
            <option value="">Select</option>
         </select>
         <span asp-validation-for="Rank"></span>
        <br>
        <br />
        <label asp-for="PhoneNumber"></label>
        <input asp-for="PhoneNumber" />
        <span asp-validation-for="PhoneNumber"></span>
        <br>
        <label asp-for="EmailAddress"></label>
        <input asp-for="EmailAddress" />
        <span asp-validation-for="EmailAddress"></span>
        <br>
        <label asp-for="PersonalURL"></label>
        <input asp-for="PersonalURL" />
        <span asp-validation-for="PersonalURL"></span>
        <br>
        <label asp-for="UnusedPassword"></label>
        <input asp-for="UnusedPassword" />
        <span asp-validation-for="UnusedPassword"></span>
        <br>
        <br>
        <br>
        <input type="submit" value="Create Instructor" />
    </form>
</body>
</html>
```

10.4 Let's Test Our Model Validation

Go to Add a new instructor and attempt to submit an empty form (see Fig. 10.4).

You should see the summary validation errors on top. Then, in-line, you should see one by one each of those errors.

Here is another example (see Fig. 10.5).

Once you fix the email address error, then the next error shows up (see Fig. 10.6).

This is because some of these errors are checked in the client's browser and hence no HTTP request has been sent to the server yet. For this reason, the *Summary list of errors* did not show up. Once you fix these errors checked on the client side (but not all errors) you will then be able to see the Summary list of errors (for the errors caught on the server side), as seen in Fig. 10.7.

Fixing the errors (by entering valid values), we get one last to fix (see Fig. 10.8). Can you figure out what we missed here?

Go back to the *Instructor.cs* and

- either make `HiringDate` required or
- make `HiringDate` to use a nullable type (change `DateTime` into `DateTime?`):

Create a new Instructor

- The value '' is invalid.
- last name is required
- The value '' is invalid.
- The value '' is invalid.
- The Password (we won't use this!) field is required.

InstructorId The value '' is invalid.
First name
Last name last name is required
Is tenured
Hiring date mm/dd/yyyy
Academic rank Select The value '' is invalid.

Office phone number
Email address
Personal webpage
Password (we won't use this!) The Password (we won't use this!) field is required.

Create Instructor

Fig. 10.4 If a user tries to submit an empty form, note that some validation messages (such as "last name is required") are being displayed in the browser:

Fig. 10.5 Is the same as Fig. 10.4, but some values are entered in the form so it passes some of the input validation. Note that the Email address still does not pass the input validation:

```
[Display(Name = "Hiring date")]
[DataType(DataType.Date)]
public DateTime? HiringDate { get; set; }
```

Now rebuild and run your application again and try entering the same values again. It should work now.

On your own, make sure that the validation also works for the `Edit` operation. Also, you may want to remove the [Required] validation attribute used for Password field (in *Instructor.cs*) since we did not include it in Edit and we won't really make use of it in this book.

10.5 Custom Validation Attributes (Optional)

Let's introduce the custom validation attributes next. We'll start with an example. Suppose we would like to only allow the `IsTenured` checkbox to be checked (i.e., set to `true`) if the `HiringDate` contains a date that is, let's say either on or after August 15th, 2007. In this particular example, we have two properties that need to be used for validation. How would you enforce this?

Fig. 10.6 Is similar to Fig. 10.5. In here, a valid email address was entered. Now the Personal webpage field produces an input validation error:

You should quickly realize that the built-in validation attributes are great for many common validation scenarios, but they are not sufficient for all cases. Luckily, it's quite easy (see below) to create our own **custom validation attributes**.

10.5.1 Create a Custom Validation Attribute

For this part, let's create a new folder, let's call it *CustomValidations* (feel free to choose a better name). In the *Solution Explorer* window, right-click on the project name, then select Add > New Folder.

In this new folder create a new *class*, make sure it is derived from the `ValidationAttribute` class, and the newly added class's name ends with `ValidationAttribute`. We will call it: `TenuredOnlyAfter2007ValidationAttribute`.

Create a new Instructor

- The value '' is invalid.
- The value '' is invalid.
- you must follow the format 000-000-0000!

InstructorId 900
First name Geovanni
Last name Ferry
Is tenured
Hiring date mm/dd/yyyy
Academic rank Select The value '' is invalid.

Office phone number two three four you must follow the format 000-000-0000!
Email address geovanni@stmartin.edu
Personal webpage https://www.stmartin.edu/dir
Password (we won't use this!)

Create Instructor

Fig. 10.7 Is another validation error, namely an invalid office phone number

In this class, we need to overload the `IsValid` which is used to check if the property value is valid. There are two overloaded `IsValid` methods (make use of IntelliSense to find out more about these methods):

```
IsValid(object? value)
```

```
IsValid(object? value, ValidationContext validationContext)
```

The first one is great if you only need a custom validation attribute that only needs to look at one (the current) property.

Since in our example we need to access two properties (the `IsTenured` and the `HiringDate`), we'll need to override the second method, which gives us access to the entire model object. Here is the code we'll use to implement our custom validation attribute:

Create a new Instructor

- The value '' is invalid.

InstructorId 900
First name Geovanni
Last name Ferry
Is tenured
Hiring date mm/dd/yyyy
Academic rank Assistant Professor
Office phone number 360-688-2748
Email address geovanni@stmartin.edu
Personal webpage https://www.stmartin.edu/dir
Password (we won't use this!)

Create Instructor

Fig. 10.8 Shows the same form as seen in Fig. 10.7, but now the validation error isn't very clear. It only specifies "The value 'is invalid'"

```
using ASPBookProject.Models;
using System.ComponentModel.DataAnnotations;

namespace ASPBookProject.CustomValidations
{
    public class TenuredOnlyAfter2007ValidationAttribute : ValidationAttribute
    {
        protected override ValidationResult? IsValid(object? value, ValidationContext
validationContext)
        {
            Instructor currentInstructor = (Instructor)validationContext.ObjectInstance;

            //only allow the IsTenured checkbox to be checked (set to true)
            //  if the HiringDate contains a date, let's say on or after August 15th, 2007.
            if (currentInstructor.IsTenured == true
                && currentInstructor.HiringDate < DateTime.Parse("2007-08-15"))
                return new ValidationResult(ErrorMessage);//not valid!!!

            return ValidationResult.Success; // all other cases are valid
        }
    }
}
```

In this method, we make use of the `validationContext.ObjectInstance` which represents the instance of the model where the attribute is being applied. Then, we use typical C# code to make sure the values of the two properties (the `IsTenured` and the `HiringDate`) are valid. If they are not valid, we return `ValidationResult` and display the value for `ErrorMessage` (we could alternatively choose our own `ErrorMessage` right in here and use our own string instead).

10.5.2 Use a Custom Validation Attribute

Using the *custom validation attribute* created above is very easy. It's just like using the built-in ones. We could use the entire class name used above, or omit the "Attribute" part:

```
[TenuredOnlyAfter2007Validation]
[Display(Name = "Is tenured")]
public bool IsTenured { get; set; }
```

To make our code friendlier, we can add an `ErrorMessage`:

```
[TenuredOnlyAfter2007Validation(ErrorMessage ="Tenured only offered after Aug. 15, 2007")]
[Display(Name = "Is tenured")]
public bool IsTenured { get; set; }
```

Make sure your *Instructor.cs* file contains the proper `using` directives. In our case, we have.

```
using ASPBookProject.CustomValidations;
using System.ComponentModel.DataAnnotations;
```

10.5.3 Let's Test the Newly Added Custom Validation

Let's test our newly added custom validation attribute. Add a *hiring date* before Aug. 15th, 2007, and make sure the *Is tenured* checkbox is checked. This should result in a model validation error as shown in Fig. 10.9.

Leave *Is tenured* checked and choose a *hiring date* after Aug. 15th, 2007 (see Fig. 10.10). If you attempt to submit this data, the error message from our custom validation attribute should go away, meaning the model validation error for this part was resolved.

Create a new Instructor

- The value '' is invalid.
- last name is required
- Tenured only offered after Aug. 15, 2007
- The value '' is invalid.

InstructorId The value '' is invalid.
First name
Last name last name is required
Is tenured Tenured only offered after Aug. 15, 2007
Hiring date 08/12/2007
Academic rank Select The value '' is invalid.

Office phone number
Email address
Personal webpage
Password (we won't use this!)

Create Instructor

Fig. 10.9 displays the same view as seen in Fig. 10.8, but now a custom validation error is been displayed for the Is tenured field. The error message displayed is "Tenured only offered after Aug. 15, 2007"

10.6 Validation Text Styling

For this part, we would like to add some styling for the validation errors displayed by

```
<div asp-validation-summary="All"></div>
```

Inside this element, we can add in-line CSS to specify the colors to be used for the error messages displayed. In particular, you can replace the line above with

```
<div asp-validation-summary="All" style="color:red"></div>
```

Create a new Instructor

- The value '' is invalid.
- last name is required
- The value '' is invalid.

InstructorId [ ] The value '' is invalid.
First name [ ]
Last name [ ] last name is required
Is tenured [x]
Hiring date 08/16/2007
Academic rank Select The value '' is invalid.

Office phone number [ ]
Email address [ ]
Personal webpage [ ]
Password (we won't use this!) [ ]

Create Instructor

Fig. 10.10 Is similar to Fig. 10.9. In here, the custom validation error message is not displayed since the Hiring Date is set to 08/16/2007, which now passes the custom validation logic we defined above

Do this for both `Add` and `Edit` views.

Now, if you are trying to submit an empty form, you would get all errors in the summary part displayed in red (see Fig. 10.11).

Also, we would like to use a better error message for academic rank. Users need to know they need to actually select an academic rank, not that `'' is invalid`. Let's fix that. Go to *Instructor.cs* and add an error message to the validation attribute used for the Rank property. Namely, change

```
[Required]
[Display(Name = "Academic rank")]
public Ranks Rank { get; set; }
```

Fig. 10.11 Note how the validation summary is displayed with red text (because of the CSS styling we added above)

into

```
[Required(ErrorMessage = "You must choose an academic rank!")]
[Display(Name = "Academic rank")]
public Ranks? Rank { get; set; }
```

Note: You must also make the type for Rank a nullable type, otherwise a default value will be sent to Validation Attribute.

Now, trying to submit an empty form will display a friendlier error message when the user does not select an academic rank (see Fig. 10.12).

Create a new Instructor

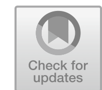
- The value '' is invalid.
- last name is required
- You must choose an academic rank!

InstructorId The value '' is invalid.
First name
Last name last name is required
Is tenured
Hiring date mm/dd/yyyy
Academic rank Select You must choose an academic rank!

Office phone number
Email address
Personal webpage
Password (we won't use this!)

Create Instructor

Fig. 10.12 Shows an error message (of "You must choose an academic rank!") when the user omits selecting an Academic rank



11 Persistent Data: Entity Framework Core

11.1 Introduction

In this chapter, we'll learn how to use the *Entity Framework Core* to work with data from a database. In particular, we'll see how to work with an SQLite Database and, alternatively, with a Microsoft SQL Server database. For more information, you may want to check out the following sources [63–65].

An **object relational mapper** is a mechanism that maps *objects* (whose classes are called **entities**) to *tables*. This allows you to work with data from databases using an object-oriented approach. Some examples of *object relational mappers*: Entity Framework Core, Django, and Hibernate.

Entity Framework Core is an *object relational mapping* (*ORM*) framework. It allows a .Net application (web application, console application, …) to work with the data stored inside a *database*. *Entity Framework Core* provides a level of abstraction between an *application* and a *database* and, as you will see below, it simplifies your data access (the CRUD operations) from the database.

You won't even need to know any SQL syntax in order to work with SQL databases (via Entity Framework Core), although it certainly helps having some foundational knowledge of databases (for example, understanding what primary keys and foreign keys are).

In this chapter, we'll mostly use an SQLite database. Then, we'll show you how easy it is to switch to a Microsoft SQL Server with very little change to your existing code (which is one of the benefits of using a layer of abstraction between your (web) application and your database).

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024

R. A. Mezei, *Introduction to the Development of Web Applications Using ASP .Net (Core) MVC*, Synthesis Lectures on Computer Science,
https://doi.org/10.1007/978-3-031-30626-6_11

11.2 Classes Involved: Providers, DbContext, and DbSet

Here is a quick introduction to what classes are involved when using Entity Framework Core. We'll see them again in the next section, where we'll go over the steps needed to set up the *Entity Framework Core* to work with our application.

To work with various types of databases, the Entity Framework uses different types of so-called **provider** classes:

- Microsoft SQL Provider—used to connect to SQL Databases (SQL Server, or Azure SQL Database).
- SQLite Provider—used to connect to an SQLite database.
- Memory Provider—used to mimic a database in memory, great for testing.
- Other providers—provided by other vendors.

There are two main classes we'll make of when working with Entity Framework Core:

- **DbContext** class has many important responsibilities, including
  - database connections (open, close, and manage connections to a database);
  - data operations (adding data, modifying data, deleting data, and data querying);
  - change tracking (it keeps track of changes in your application—so you can save them to the database);
  - data mapping (it maps properties from entities to columns in tables);
  - transaction management (when `SaveChanges` is called, a transaction is created for all pending changes. If an error/exception occurs when the changes are applied to the database, they are all rolled back).
- **DbSet <TEntity>** class
  - `DbSet<TEntity>` classes are added as properties to our class derived from `DbContext`;
  - each DbSet property represents the data (as a collection) from one table in the database;
  - we'll use this to perform database operations for one table;
  - by default, entity/model properties are mapped to database columns with the same name.

We call **entity classes** the (*model*) classes that we map to *tables* in the database.

11.3 Add Entity Framework Core to Our Web Application

Next, we'll go over the steps needed to add Entity Framework Core to our web application. These steps may look challenging the first time you see them, but you will only need to go over them once. Once you set up Entity Framework Core in your application,

you'll see how great the benefits are: it will be quite easy to perform operations on data stored in databases.

11.3.1 Step 1: Create/Choose Your Entity Classes

We call **entity classes** the (*model*) classes that we want to map to *tables* in the database. In our example, we will choose the `Student` and the `Instructor` classes as our *entities*.

Make sure each *entity* class contains one property that

- has the name `Id`, or
- has the name `[classname]Id`, or
- has the `[Key]` attribute.

Such a property will be mapped as the *primary key* of the corresponding table, to uniquely identify each row in the table.

11.3.2 Step 2: Install NuGet Packages

Very important! Before you proceed with this step, make sure your current code has no compilation errors. Otherwise, you may not be able to install NuGet packages!

Next, we need to get access to Entity Framework Core classes. For this, we will install the following package:

- For SQLite: **`Microsoft.EntityFrameworkCore.Sqlite`**
- Later, for MS SQL Server databases, we'll use **`Microsoft.EntityFramework Core. SqlServer`**.

We will also install the following NuGet package that provides support for Migrations:

- **Microsoft.EntityFrameworkCore.Design**.

To install NuGet packages, inside Visual Studio, go to *Tools* > *Manage NuGet Packages for Solution …* (note: one can also use the *Package Manager Console*). Then, in the window that opens up, make sure to select the *Browse* tab!

In there, type in the **name** of the package you want to install, choose the **project** where to install it, and the **version** of the package you want to get installed. For this book, we searched for `Microsoft.EntityFrameworkCore.Sqlite` and we selected the version 6.0.12. Then, on the right side of this window, make sure to click on the checkbox next to your project name (ASPBookProject for us), then click on the **Install** button.

A **Preview Changes** window will appear that will indicate what changes Visual Studio will make to your project. Click OK.

Then the **License Acceptance** window. We clicked on **I Accept** button. Wait for a few seconds so the installation finishes, then confirm that you got no errors (check out the Error List window in Visual Studio).

Now repeat the same steps for the other package: `Microsoft.EntityFrameworkCore.Design`.

11.3.3 Step 3: Create a Class Derived from DbContext

In this step, we'll create a class derived from the class `DbContext`. As mentioned above, this class will be responsible for connecting to the database and performing various operations, among other things.

First, let's create a new folder in our project and give it a name. We chose *Data*, but feel free to give it a better name if you prefer.

In this newly added folder, *Data*, create a new class (we called it `OurDbContext`) derived from `DbContext`. In it, make sure to add the `using` directive:

```
using Microsoft.EntityFrameworkCore;
```

Here is what we have so far in *OurDbContext.cs* file:

```
using Microsoft.EntityFrameworkCore;

namespace ASPBookProject.Data
{
    public class OurDbContext: DbContext
    {
    }
}
```

In this class, we will need to create a `DbSet` *property* for each *table* in the database. The name that we choose for each `DbSet` property will be used when creating/using the table in the database (of course we can use attributes to override this if needed). The `DbSet` is a generic class, and it needs to know what entity class to work with—essentially each row in the table will be mapped to an object/instance of class: the entity class.

In our book project, we will initially work with two entities, `Student` and `Instructor`. Let's choose a name for these tables. Let's say `Roster` (the table of students) and `Instructors` (the table of instructors). For this, we will need to define the two `DbSet` properties shown below:

```
using ASPBookProject.Models;
using Microsoft.EntityFrameworkCore;

namespace ASPBookProject.Data
{
    public class OurDbContext : DbContext
    {
        public DbSet<Student> Roster { get; set; }
        public DbSet<Instructor> Instructors { get; set; }
    }
}
```

11.3.4 Step 4: Data Seeding

Next, would like to put some sample data in our database when we create the tables specified above. To do this, we can override the `OnModelCreating` method (see more in [66]) inside `OurDbContext` class, as shown below. For completion, we included the entire contents of *OurDbContext.cs*:

```
using ASPBookProject.Models;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Hosting;


namespace ASPBookProject.Data
{
    public class OurDbContext : DbContext
    {
        public DbSet<Student> Roster { get; set; }
        public DbSet<Instructor> Instructors { get; set; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            base.OnModelCreating(modelBuilder); //do not remove this!

            modelBuilder.Entity<Instructor>().HasData( //populate the table containing Instructors
                 new Instructor()
                 {
                     InstructorId = 100,
                     FirstName = "Maegan",
                     LastName = "Borer",
                     IsTenured = false,
                     HiringDate = DateTime.Parse("2018-08-15"),
                     Rank = Ranks.AssistantProfessor
                 },

                new Instructor()
                {
                    InstructorId = 200,
                    FirstName = "Antonietta ",
                    LastName = "Emmerich",
                    IsTenured = true,
                    HiringDate = DateTime.Parse("2022-08-15"),
                    Rank = Ranks.AssociateProfessor
                },

                new Instructor()
                {
                    InstructorId = 300,
                    FirstName = "Antonietta",
                    LastName = "Lesch",
                    IsTenured = false,
                    HiringDate = DateTime.Parse("2015-01-09"),
                    Rank = Ranks.FullProfessor
                },

                new Instructor()
                {
                    InstructorId = 400,
                    FirstName = "Anjali",
                    LastName = "Jakubowski",
                    IsTenured = true,
                    HiringDate = DateTime.Parse("2016-01-10"),
                    Rank = Ranks.Adjunct
                }
                );

            modelBuilder.Entity<Student>().HasData( //populate the table containing Students
                    new Student()
                    {
                        StudentId = 10,
                        FirstName = "Elisa",
```

```
                        LastName = "Wiza",
                        Major = Major.CS,
                        IsVeteran = true,
                        GPA = 4.0,
                        AdmissionDate = DateTime.Parse("2022-08-15")
                    },

                    new Student()
                    {
                        StudentId = 20,
                        FirstName = "Simeon",
                        LastName = "Senger",
                        Major = Major.IT,
                        IsVeteran = true,
                        GPA = 3.75,
                        AdmissionDate = DateTime.Now
                    },

                    new Student()
                    {
                        StudentId = 30,
                        FirstName = "Blaise",
                        LastName = "Nicholas",
                        Major = Major.MATH,
                        IsVeteran = true,
                        GPA = 3.98,
                        AdmissionDate = DateTime.Parse("2020-08-15")
                    }
                );
            }
        }
    }
```

11.3.5 Step 5: Register Our DbContext as a Service, and Use a Connection String

In order for us to be able to pass parameters to the constructor of `OurDbContext` class, we need to define a non-default constructor in this class.

The parameters we'll pass to this constructor will actually need to be passed along to the constructor of the base class (`DbContext`); hence, we need to make sure that the non-default constructor of `OurDbContext` calls the non-default constructor of `DbContext` and passes these parameters along.

Here is how we accomplish this. Add the following constructor to the `OurDbContext` class.

```
public OurDbContext(DbContextOptions<OurDbContext> options) : base(options)
{
}
```

Note: The type of the parameter used for options is a generic class, `DbContextOptions`, for which we used our own `OutDbContext` class as a type, hence the use of `DbContextOptions<OurDbContext>`.

Here is how the OurDbContext class looks like once we added the above code and added a few comments:

```
namespace ASPBookProject.Data
{
    public class OurDbContext : DbContext
    {
        //used to map to tables
        public DbSet<Student> Roster { get; set; }
        public DbSet<Instructor> Instructors { get; set; }

        //constructor
        public OurDbContext(DbContextOptions<OurDbContext> options) : base(options)
        {
        }

        //used to seed the databases with some data
        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            // ... skipped …

        }
    }
}
```

Here is the reason for the paragraphs given above. We'll need to make use of this non-default constructor in the code below, to register `OurDbContext` class as a service. We will use the code as shown here (do not add this code yet!):

```
builder.Services.AddDbContext<OurDbContext>(
    options => options.UseSqlite("connection string to your database")
    );
```

In the code shown above, the string `"connection string to your database"` can be replaced by any valid *connection string*. The **connection string** is essentially a string (text) containing information on where and how to connect to our database (see more in [67]). Here are some examples of connections strings:

For an SQLite database (for a database file located in c:\myowndb.db, or for a database located in memory only):

- `"Data Source=c:\\myowndb.db;Version=3;"`
- `"Data Source=:memory:;Version=3;New=True;"`

For a Microsoft SQL Server database (on a local server, or a remote one):

- `"Data Source=(localdb)\\MSSQLLocalDB;Initial Catalog=master;Integrated Security=True;Connect Timeout=30;Encrypt=False;ApplicationIntent=ReadWrite;MultiSubnetFailover=False;"`
- `"Data Source=142.251.33.110,1433;Network Library=DBMSSOCN;Initial Catalog=myDataBase;User ID=aUsername;Password=aPassword;"`

So, for our application, we could use the following (don't add this code yet!):

```
builder.Services.AddDbContext<OurDbContext>(
    options => options.UseSqlite("Data source=aspbook.db")
    );
```

But instead of the code above, we'll go one step further and make use of the *appsettings.json* file which is a file where we can put various application configuration settings. We'll see this file again later in this book.

Open the *appsettings.json* file (you should be able to see it in Solution Explorer window). In it, add the following entry to set a connection string:

```
,
"ConnectionStrings": {
  "BookConnectionString":"Data source=aspbook.db"
}
```

Our appsettings.json file now looks as follows:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "BookConnectionString": "Data source=aspbook.db"
  }
}
```

Instead of the key `BookConnectionString`, feel free to choose a better, more meaningful, name.

Then, back into the *Program.cs* file, we will use code that will read the *connection string* defined above (now copy this!):

```
builder.Services.AddDbContext<OurDbContext>(
    options => options.UseSqlite(builder.Configuration.GetConnectionString("BookConnectionString"))
    );
```

This will register *Entity Framework Core* (`OurDbContext`) as a *service*. We are now able to use it throughout our application (via *dependency injection*).

Our *Program.cs* file looks as follows:

```
using ASPBookProject.Data;
using ASPBookProject.Services;
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddSingleton<IMyFakeDataService, MyFakeDataService>(); //our data service
builder.Services.AddControllersWithViews(); //adds services needed for controllers
builder.Services.AddDbContext<OurDbContext>(
    options =>
options.UseSqlite(builder.Configuration.GetConnectionString("BookConnectionString"))
    );

var app = builder.Build();//set up middleware components.
app.UseStaticFiles(); //needed to give access to files in wwwroot
app.UseRouting();  //adds route matching to the middleware pipeline
app.MapControllerRoute( //modified default routing
    name: "default",
    pattern: "{controller=Instructor}/{action=Index}/{id?}");


app.Run();
```

One last step and we are done with this a little long configuration process. But as you'll see below, it is so worth it!

The last step we want to do is the following. Once all services are set up we would like to call the method called `EnsureCreated` that will make sure our database is created:

- If the database exists, nothing will be done, but
- if the database doesn't exist, the `EnsureCreated` method will make sure to create it (based on the specifications of the `OurDbContext` class).

In *Program.cs*, right before `app.UseStaticFiles()`, add the following code:

```
var context = app.Services.CreateScope().ServiceProvider.GetRequiredService<OurDbContext>();
context.Database.EnsureCreated(); //if our database does not exist, then create it!
```

11.3.6 Test Our Database

We are finally finished with setting up Entity Framework Core for our application. If you rebuild your application, you should see that an SQLite database file (names *aspbook.db*) was created. See it in the *Solution Explorer* window (above the Program.cs file).

Inside Visual Studio (Solution Explorer window), if you right-click on the *aspbook.db* file, you'll see the option of opening it with a program (**Open With …**). Click on this option.

Then click on the *Add…* button and select the *DbBrowser* application we installed at the beginning of the book. On my computer, that location is *C:\Program Files\DB Browser for SQLite\DB Browser for SQLite.exe.* Click the OK button. Then another OK button.

Once the database file opens in DbBrowser, you should be able to see the tables in our database (make sure you are in the *Database Structure* tab). For us, we see that our database currently has three tables:

- Instructors.
- Roster.
- sqlite_sequence.

You should note that for the tables named *Instructors*, and *Roster*, these are the names we gave to our two `DbSet` properties of `OurDbContext` class.

Now click on the *Browse Data* tab and you'll see the data in our tables. By default, the *Instructors* table opens up.

You can choose to see the data from Roster table (see the dropdown menu at the top of the table, on the very left side).

Very important: Make sure to click the *Close Database* button before you run your web application, otherwise you may get read access errors. This button is on the top right of the DbBrowser window.

From now on, if you ever wish to reopen a database, the easiest way to do so is as follows: run DbBrowser, then under the File menu, at the bottom of the menu window that opens, click on the database name shown in that list.

11.4 Use Entity Framework Core in Our Web Application, Dependency Injection Revisited

Now let's make use of our database via Entity Framework core. We have everything set up, so using the database will be quite easy.

To use Entity Framework Core in our Controller classes, we will need to "inject" it in there and then make use of it. This is similar to injecting services seen in the previous chapter:

- Create a private field and use the constructor to populate it.
- Let's update all `InstructorController` actions so it makes use of the database.

11.4.1 Inject Entity Framework Core in InstructorController

Let's inject the entity framework core in `InstructorController`. Since we won't need the `MyFakeService` anymore, we'll replace that with `OurDbContext` class. Inside *InstructorController.cs* file, replace the code:

```
//injecting the IMyFakeDataService
private readonly IMyFakeDataService _fakeData;
public InstructorController(IMyFakeDataService fakeData)
{
    _fakeData = fakeData;
}
```

with

```
//injecting the EntityFramwework Core - OurDbContext
private readonly OurDbContext _dbContext;
public InstructorController(OurDbContext dbContext)
{
    _dbContext = dbContext;
}
```

Optionally (only if you wish to clean up the code), you can delete all code related to `MyFakeService`, since we won't use this anymore.

11.4.2 Update the Actions to Use Entity Framework Core

Next, we will update each *action*, to make use of the Entity Framework Core. All we need is to replace `_fakeData.InstructorsList` with `_dbContext.Instructors`.

- `_dbContext` represents the instance of `OurDbContext` managed by the dependency injection service.
- `Instructors` is the `DbSet <Instructor>` property we defined in `OurDbContext` class.

IMPORTANT: Until you call the `SaveChanges` method, all changes to `_dbContext.Instructors` will not be saved into the database, and they are only changed in the web application's memory! Therefore, please make sure to call `_dbContext.SaveChanges();` after every Add/Edit/Delete.

That's it! We give below our complete code for *InstructorController.cs* so you can check your work.

```
using ASPBookProject.Data;
using ASPBookProject.Models;
using ASPBookProject.Services;
using Microsoft.AspNetCore.Mvc;

namespace ASPBookProject.Controllers
{
    public class InstructorController : Controller
    {
        //injecting the EntityFramwework Core - OurDbContext
        private readonly OurDbContext _dbContext;
        public InstructorController(OurDbContext dbContext)
        {
            _dbContext = dbContext;
        }
public IActionResult Index()
{
    return View(_dbContext.Instructors); //will use the Index.cshtml view
}

public IActionResult DisplayAll()
{
    return View("Index", _dbContext.Instructors);//will use the Index.cshtml view
}


public IActionResult ShowAll()
{
    return RedirectToAction("Index", _dbContext.Instructors);
}


public IActionResult ShowDetails(int id)
{
    //search for the instructor whose InstructorId matches the given id
    //  here we are using InstructorsList, later we'll use a database!
    Instructor? instr = _dbContext.Instructors.FirstOrDefault(ins => ins.InstructorId == id);

    if (instr != null) //was an instructor found?
        return View(instr);
    //if no instructor was found ...
    return NotFound();
}


[HttpGet]
public IActionResult Add()
{
    return View();
}

[HttpPost]
public IActionResult Add(Instructor newInstructor)
{
    if (!ModelState.IsValid) //if the data is invalid
        return View(); //go back to the view

    _dbContext.Instructors.Add(newInstructor); //add the new instructor to our list
    _dbContext.SaveChanges();
    return RedirectToAction("Index");
}
```

```
        [HttpGet]
        public IActionResult Edit(int id)
        {
            //we should look for the instance that has the given Id
            //   ... later we'll search in the database
            Instructor? instr = _dbContext.Instructors.FirstOrDefault(inst => inst.InstructorId == id);

            if (instr != null) //if found, send it to the view
                return View(instr);
            //if no matching instructor was found ...
            return NotFound();
        }

        [HttpPost]
        public IActionResult Edit(Instructor instructorChanges)
        {
            if (!ModelState.IsValid) //if the data is invalid
                return View(); //go back to the view

            //find the instructor from InstructorList
            //   who has the same InstructorId as the changes.InstructorId
            Instructor? instr = _dbContext.Instructors.FirstOrDefault(instr => instr.InstructorId ==
instructorChanges.InstructorId);

            if (instr != null) //if found, change the values in InstructorsList to match the changes
            {
                instr.LastName = instructorChanges.LastName;
                instr.IsTenured = instructorChanges.IsTenured;
                instr.HiringDate = instructorChanges.HiringDate;
                instr.Rank = instructorChanges.Rank;
                _dbContext.SaveChanges();
            }
            return RedirectToAction("Index");
        }


        [HttpGet]
        public IActionResult Delete(int id)
        {
            //we should look for the instance that has the given Id
            //   ... later we'll search in the database
            Instructor? instr = _dbContext.Instructors.FirstOrDefault(inst => inst.InstructorId == id);

            if (instr != null) //if found, send it to the view
                return View(instr);
            //if no instructor was found ...
            return NotFound();

        }

        [HttpPost]
        public IActionResult DeleteConfirmed(int instructorId)
        {
            //we should look for the instance that has the given Id
            //   ... later we'll search in the database
            Instructor? instr = _dbContext.Instructors.FirstOrDefault(inst => inst.InstructorId ==
instructorId);

            if (instr != null) //if found, delete it from list
            {
                _dbContext.Instructors.Remove(instr);
                _dbContext.SaveChanges();
                return RedirectToAction("Index");
            }
            //if no instructor was found ...
            return NotFound();
        }
    }
}
```

11.4.3 Important: Automated Id Generation

Currently, if we go to *Add a new instructor* (see Fig. 11.1), we are asked to provide an `InstructorId`.

This is.

- Inconvenient—why should the users come up with an `ID`?
- Problematic—if you enter an `Id` that is already in use, you'll get an error (a `SqliteException` error).

Fig. 11.1 Shows the Add view displayed in a browser. In particular, note that we have an input field used for the InstructorId

The good news is that we can get the `Id` automatically generated. Go to the *Add* view and remove the field that asks the user to enter an id. Remove the following lines from *Add.cshtml*:

```
<label asp-for="InstructorId"></label>
<input asp-for="InstructorId" />
<span asp-validation-for="InstructorId"></span>
<br>
```

That's it. Now, a unique `InstructorId` will automatically be generated for us by the database.

11.4.4 Let's Test That We Have Persistent Data

Go to *add a new Instructor* (see Fig. 11.2).

A new row should be seen in the table. Try adding the same information again. You'll see a new instructor (with a different `InstructorId`) will be created (see Fig. 11.3).

If you are curious to see what InstructorId values were created, you can open the database file in DbBrowser and check out the contents of the *Instructors* table. Also, if you want to know how our database keeps track of what Id to use, check out the contents of the *sqlite_sequence* table (you'll find in there a column names `seq`, and an entry for each of our two tables: `Instructors`, `Roster`).

Before you continue, make sure to close the database in Db Browser! Click on the ***Close Database*** icon.

Create a new Instructor

First name: Aliyah
Last name: Wiza
Is tenured: ☐
Hiring date: 12/29/2022
Academic rank: Full Professor

Office phone number: 360-688-2748
Email address: Wiza@stmartin.edu
Personal webpage:
Password (we won't use this!):

Create Instructor

Fig. 11.2 Is similar to the one seen in Fig. 11.1, but it no longer contains any input field used for InstructorId

All instructors

First name	Last name	Academic rank	Details (HTML helper)	Details (tag helper)	Edit	Delete
Maegan	Borer	Assistant Professor	details	details	edit this	delete this
Antonietta	Emmerich	Associate Professor	details	details	edit this	delete this
Antonietta	Lesch	Full Professor	details	details	edit this	delete this
Anjali	Jakubowski	Adjunct	details	details	edit this	delete this
Aliyah	Wiza	Full Professor	details	details	edit this	delete this

Add a new instructor

Fig. 11.3 Shows the Index view displayed in a browser. It contains a table with multiple rows, one for each Instructor

Now, on your own, please modify some of the existing instructors, and delete one. Then, rebuild your application and check that the changes are persistent. Those changes are saved in a database, so changes should survive web application restarts.

11.4.5 EnsureDeleted

As we make modifications to our tables, you may want to add the following line right before `EnsureCreated` method calls (inside *Program.cs*):

```
context.Database.EnsureDeleted(); //if our database exists, then erase it!
```

This way our database will be recreated each time we rebuild our application. Comment this line out and only use it when you need to recreate the database (for example, when we are modifying a property type, or other scenarios that will lead to compilation errors related to entity framework).

11.5 Practice: Update the StudentController Class

To have some more functionality to work with later, let's update the `StudentController` class.

11.5.1 Inject Entity Framework in StudentController

To inject entity framework (and similarly to any other service), we need the following:

- A private field: `OurDbContext _ourDbContext`
- A constructor that will initialize the field declared above with the instance managed by the Dependency Injection.

Here is the code we added to StudentController.cs class:

```
private readonly OurDbContext _dbContext;
public StudentController(OurDbContext ourDbContext)
{
    _dbContext = ourDbContext;
}
```

Now we have access to our database via Entity Framework Core. Next, let's create/update actions and make use of our database.

11.5.2 Use Entity Framework Core in StudentController Actions

We give below a possible solution to StudentController.cs. On your own, create/update the necessary views for each action.

```
using ASPBookProject.Data;
using ASPBookProject.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;

namespace ASPBookProject.Controllers
{
    public class StudentController : Controller
    {
        private readonly OurDbContext _dbContext; //inject OurDbContext in this class
        public StudentController(OurDbContext ourDbContext)
        {
            _dbContext = ourDbContext;
        }

        public IActionResult Index()
        {
            return View(_dbContext.Roster);
        }

        public IActionResult ShowDetails(int id)
        {
            //search for the student whose StudentId matches the given id
            Student? st = _dbContext.Roster.FirstOrDefault(student => student.StudentId == id);

            if (st != null) //was a student found?
                return View(st);
            //if no student was found ...
            return NotFound();
        }

        [HttpGet]
        public IActionResult Add()
        {
            return View();
        }

        [HttpPost]
        public IActionResult Add(Student newStudent)
        {
            if (!ModelState.IsValid) //if the data is invalid
                return View(); //go back to the view

            _dbContext.Roster.Add(newStudent); //add the new student to our list
            _dbContext.SaveChanges();
            return RedirectToAction("Index");
        }

        [HttpGet]
        public IActionResult Edit(int id)
        {
           //we should look for the instance that has the given Id
           Student? st = _dbContext.Roster.FirstOrDefault(st => st.StudentId == id);

           if (st != null) //if found, send it to the view
               return View(st);
           //if no student was found ...
           return NotFound();
       }


       [HttpPost]
       public IActionResult Edit(Student studentChanges)
       {
           if (!ModelState.IsValid) //if the data is invalid
               return View(); //go back to the view

           //find the student who has the same StudentId as the studentChanges.StudentId
           Student? st = _dbContext.Roster.FirstOrDefault(student => student.StudentId ==
studentChanges.StudentId);

           if (st != null) //if found, change the values in the database to match our changes
           {
               st.LastName = studentChanges.LastName;
               // add other properties, as needed
               _dbContext.SaveChanges();
           }
           return RedirectToAction("Index");
       }
```

```
        [HttpGet]
        public IActionResult Delete(int id)
        {
            //we should look for the instance that has the given Id
            Student? st = _dbContext.Roster.FirstOrDefault(student => student.StudentId == id);

            if (st != null) //if found, send it to the view
                return View(st);
            //if no student was found ...
            return NotFound();

        }

        [HttpPost]
        public IActionResult DeleteConfirmed(int studentId)
        {
            //we should look for the instance that has the given Id
            Student? st = _dbContext.Roster.FirstOrDefault(student => student.StudentId == studentId);

            if (st != null) //if found, delete it from list
            {
                _dbContext.Roster.Remove(st);
                _dbContext.SaveChanges();
                return RedirectToAction("Index");
            }
            //if no student was found ...
            return NotFound();
        }


        public IActionResult GoToGoole()
        {
            return Redirect("https://www.google.com/");
        }


        public IActionResult AnotherIndex()
        {
            return RedirectToAction("Index");
        }
    }
}
```

11.6 How to Use Microsoft SQL Server Instead of SQLite (Optional)

In this section, we would like to show you how easy it is to switch your web application such that instead of using a SQLite database it uses an Microsoft SQL Server database.

IMPORTANT: Before you continue, you may want to make a copy of your project. The remaining chapters of this book will continue with SQLite.

11.6.1 Install SQL Server Express LocalDB Database on Your Machines

For this exercise, you will need to have LocalDB installed on your machine unless you choose to use another existing installation of Microsoft SQL server.

One way to install the *SQL Server Express 2019 LocalDB* database is by opening **Visual Studio Installer**. Then click on the *Modify* button. Then, go to *Individual Components* tab, and make sure the option *SQL Server Express 2019 LocalDB* is checked.

11.6.2 Make Changes so Entity Framework Core Now Works with a Microsoft SQL Server Database

We will need to install the following NuGet package: `Microsoft.EntityFrameworkCore.SqlServer` (follow the steps seen earlier to install this package).

Feel free to remove the NuGet package: `Microsoft.EntityFrameworkCore.SqlServer`. For this, go to the *Installed* tab, select the packaged, then on the right side click on the **Uninstall** button.

We are almost done. We next need to use an SQL Server provider instead of the SQLite provider (in *Program.cs*):

```
//builder.Services.AddDbContext<OurDbContext>(
//    options => options.UseSqlite(builder.Configuration.GetConnectionString("BookConnectionString"))
//    );
builder.Services.AddDbContext<OurDbContext>(
    options => options.UseSqlServer(builder.Configuration.GetConnectionString("BookConnectionString"))
    );
```

And the last step, update the connection string set in *appsettings.json* so it points to our SQL Server database instead:

```
"BookConnectionString": "Data Source=(localdb)\\MSSQLLocalDB;Initial Catalog=master;Integrated
Security=True;Connect
Timeout=30;Encrypt=False;ApplicationIntent=ReadWrite;MultiSubnetFailover=False;"
```

The *appsettings.json* file should now contain the following:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    //"BookConnectionString": "Data source=aspbook.db"
    "BookConnectionString": "Data Source=(localdb)\\MSSQLLocalDB;Initial Catalog=master;Integrated
Security=True;Connect Timeout=30;Encrypt=False;ApplicationIntent=ReadWrite;MultiSubnetFailover=False;"
  }

}
```

That's it. Your controllers should work the same with this new database. Do you see the advantage of using Entity Framework and how it provides a layer of abstraction between our code and the database?

Run your application. You should see how the database was recreated with our seed data (see Fig. 11.4).

All instructors

First name	Last name	Academic rank	Details (HTML helper)	Details (tag helper)	Edit	Delete
Maegan	Borer	Assistant Professor	details	details	edit this	delete this
Antonietta	Emmerich	Associate Professor	details	details	edit this	delete this
Antonietta	Lesch	Full Professor	details	details	edit this	delete this
Anjali	Jakubowski	Adjunct	details	details	edit this	delete this

Add a new instructor

Fig. 11.4 Shows the Index view that displays information from our seed data

Fig. 11.5 Shows the Add view containing information entered by a user

To check the data that exists in our database, from inside Visual Studio, go to *View > SQL Server Object Explorer*. From the *SQL Server Object Explorer* window, check out the tables created in the *master* database (under SQL Server > (localdb)\MSSQLLocalDB … > Databases > System Databases > master > Tables, you should see *dbo.Instructors* and *dbo.Roster*).

Right-click on any of the two tables and select *View Data*. This should allow you to see the data that is currently stored in that table.

Create a new instructor, for example (Fig. 11.5).

Then check the database again. You should see that a new row is added into the *Instructors* table (if needed, make sure to press the refresh button). You should also note that the password was saved as plain text. We'll address this issue in Chap. 14.

Check for updates

12 Consistent Look: Layouts, Friendly Error Pages, and Environments

Before we continue, make sure to return back to the version of the project that uses the SQLite database (feel free to continue with the Microsoft SQL Server database if you prefer, in our book we will use our SQLite database).

This chapter contains several smaller concepts put together in one chapter. In this chapter, among other things, we'll make our web application look pretty. We'll add a filter button, we'll create a consistent look and feel for our application, and we'll create friendly error pages.

12.1 Filter Results

In this section, we will add a filter functionality that would allow a user to narrow down the results shown in the `Instructors` table displayed in `Index`. In our example, we will allow users to filter results based on the `LastName` field. For more information on this, read the following source [68].

The first step is adding an input field (search field) and a button to narrow the results based on the input field's value. Where would you add these fields? In which file?

12.1.1 Update the Index View

We can start by adding an input field and two buttons (one for filter, one for reset filter/cancel) in the *Index.cshtml*, right after the `<H1>` element:

```
<input type="text" placeholder=" last name includes ..." />
<button>Filter results</button>
<button>Clear the filter</button>
```

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024

R. A. Mezei, *Introduction to the Development of Web Applications Using ASP .Net (Core) MVC*, Synthesis Lectures on Computer Science,
https://doi.org/10.1007/978-3-031-30626-6_12

Fig. 12.1 Shows a part of the Index view in a browser. It contains an input box and two buttons

The result is shown below (see Fig. 12.1).

Next, we would like to add functionality to these buttons. To group those elements together and easily program them, we'll put them inside a <FORM> element.

```
<form>
    <input type="text" placeholder="last name includes ..." />
    <button>Filter results</button>
    <button>Clear the filter</button>
</form>
```

The output looks the same as above, but now we can easily add functionality to those buttons.

The first button can be turned into a *submit* button. Also, when we click on that submit button, we want the request to be sent to the Index action. Let's make the requests a GET requests, so we can see the searched values in the URL. Lastly, the input text value will not be sent to the server unless we give it a name, so let's give it a name, say "SearchByLastName":

```
<form asp-action="Index" method="get">
    <input type="text" placeholder="last name includes ..." name="SearchByLastName" />
    <button type="submit">Filter results</button>
    <button>Clear the filter</button>
</form>
```

Let's test what we have got so far. Type in some text in the textbox (see Fig. 12.2), then click on the submit button.

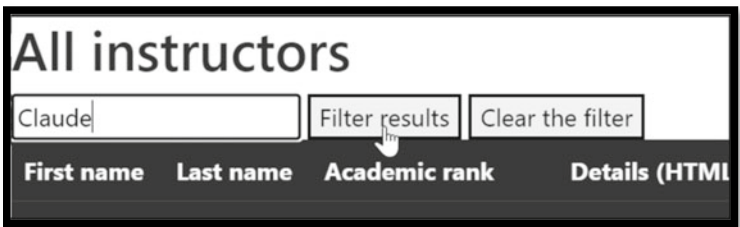


Fig. 12.2 Is similar to Fig. 12.1, but now the input element contains some text ("Claude")

You should see that the text we enter in the input box above will appear in the URL (once we click on the *Filter Results* button), along with the name we defined above: `SearchByLastName`:

```
localhost:5125/?SearchByLastName=Claude
```

As of right now, the `Index` action sends all instructors, the entire list, to the view to be displayed. It does not yet use our value from the textbox (represented by the variable `SearchByLastName`). We'll fix this next.

12.1.2 Update the Index Action

Let's add code to make use of this value. Change the `Index` action (from `InstructorController`) to match the code below.

Namely, we will add one parameter, which matches the name of the text field variable: `SearchByLastName`, and then will make use of it. How is the ***model binding*** helping us in here? The model binding grabs the value of `SearchByLastName` from the **HTTP request** (the URL in this case) and passed it to the `Index` action. Then we narrow down the results based on the parameter: `SearchByLastName`. And lastly, we pass the results to the View.

```
public IActionResult Index(string SearchByLastName)
{
    //var instructors = from instr in _dbContext.Instructors
    //                  select instr; //we start with all instructors - LINQ syntax

    var instructors = _dbContext.Instructors.AsEnumerable(); //an alternative

    if (SearchByLastName != null) //narrow down our results
    {
        instructors = instructors.Where(instr => instr.LastName.Contains(SearchByLastName));
    }
    return View(instructors);
}
```

This should narrow down the results. Try, for example, the following (see Fig. 12.3).

Then click on the *Filter results* button (you'll be taken to a page with the URL: *localhost:5125/?SearchByLastName=er*), as seen in Fig. 12.4.

Fig. 12.3 Is similar to Fig. 12.1, but now the input element contains some text ("er")

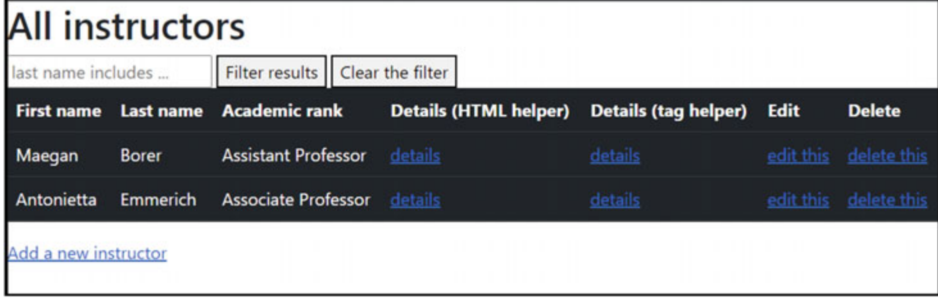


Fig. 12.4 Shows the result of narrowing the table listed in Index view. In this example, only rows containing "er" in the Last name column are displayed

You should get the correct results. Also, if you look at the URL, it contains the filtering value (because we chose to use the GET method!).

There is only one problem though. We would like to have the filtering value (chosen by the user) stay in the textbox. That would be more user-friendly. One solution is to make use of the `ViewBag` object (set a value in the `Index` *action*) and pass this value to the `Index` *view*, then use this value in the *view* as the default value for the textbox.

In the `Index` *action*, before the return statement, add the following line:

```
ViewBag.SearchByLastName = SearchByLastName; //pass this value to the view
```

Then, in the `Index` *view*, change the line:

```
<input type="text" placeholder="last name includes ..."/>
```

into

```
<input type="text" placeholder="last name includes ..." name="SearchByLastName"
value="@ViewBag.SearchByLastName" />
```

That's it. If you test your work, the web application should now keep the value entered in the text field (see Fig. 12.5).

12.1.3 Implement the *Clear the Filter* Button

In here, we would like to be able to click on the *Clear the filter* button to clear the textbox and display the entire list. One way to do this is to give our text field an id (we chose: `LastNameFilter`), so we can easily refer to it by *id*, using JavaScript. Below we added `id="LastNameFilter"`:

```
<input type="text" placeholder="last name includes ..." id="LastNameFilter"
name="SearchByLastName" value="@ViewBag.SearchByLastName" />
```

All instructors

er | Filter results | Clear the filter

First name	Last name	Academic rank	Details (HTML helper)	Details (tag helper)	Edit	Delete
Maegan	Borer	Assistant Professor	details	details	edit this	delete this
Antonietta	Emmerich	Associate Professor	details	details	edit this	delete this

Add a new instructor

Fig. 12.5 After the user clicks on the Filter results button, the contents of the filtering input box ("er" in this example) are preserved

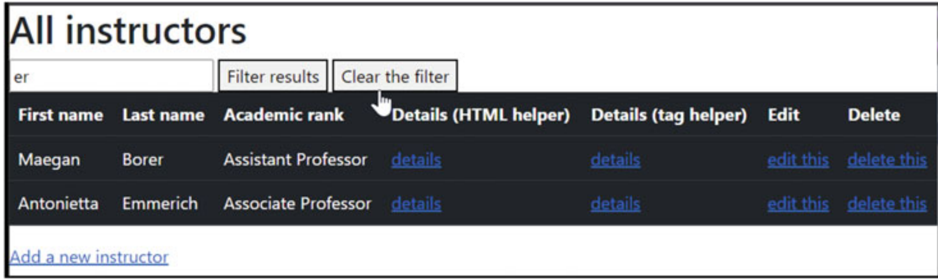


Fig. 12.6 Shows the mouse hovering above the Clear the filter button

Then, for the clear button, we add the JavaScript code that set the text field above to `null`. In particular, we add the JavaScript code to respond to the click event (when the user clicks on this button). Change the Clear the filter <BUTTON> element to match the code below:

```
<button onclick="document.getElementById('LastNameFilter').value = null">Clear the
filter</button>
```

In here (see Fig. 12.6).

If you click on the *Clear the filter* button, you should get back the entire list/table, and the text field should be cleared.

12.2 Filter Results Using a Dropdown List (Optional)

In this part, we would like to add a **dropdown** list so the user can select to only view specific ranks (for example, *Assistant Professors*). To make it a little more complex, we would like to only allow in the dropdown list options that are available. For example, if there are no *Adjunct* instructors in our table, this option should not be available in the dropdown list.

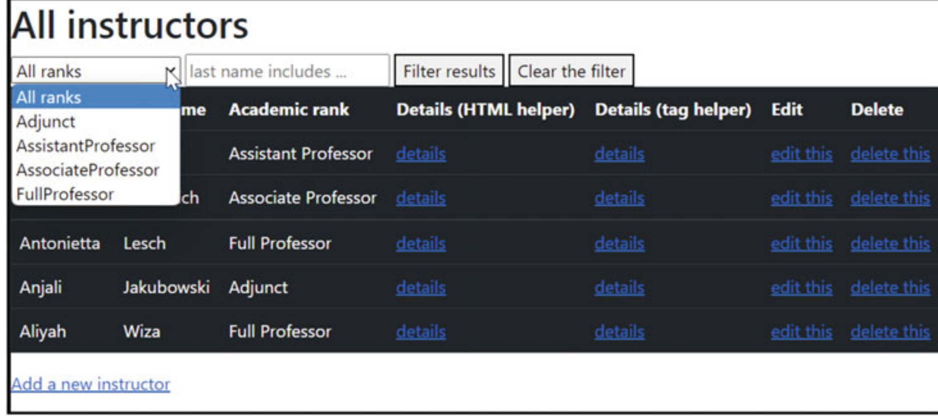


Fig. 12.7 Shows a figure similar to the one in Fig. 12.5. But the input field was changed to a dropdown list selector

12.2.1 Create the Dropdown List Items in the Index Action

In the `Index` *action* (of `InstructorController` class), we first need to create the list of available ranks and send it to the view (via the dynamic object `ViewBag`). Add this code right before the `return View` statement.

```
var AvailableRanks = from instr in instructors //get a list of available ranks
                      orderby instr.Rank
                     select instr.Rank;
//pass the available ranks to the view - distinct only!

ViewBag.AvailableRanks = AvailableRanks.Distinct();
```

12.2.2 Display the Dropdown List Items in the Index View

Add the following line to the form in the Index view, right before the text field used for filtering:

```
<select asp-items="@(new SelectList(ViewBag.AvailableRanks, "All"))">
        <option value="">All ranks</option>
</select>
```

You should obtain the following dropdown list (Fig. 12.7).

12.2.3 Use of the Dropdown List to Filter Our Results

In order to be able to filter our results based on the dropdown list, we will need to pass the dropdown selected value from the `Index` *view* back to the `Index` *action* and narrow

down the results based on that value. For this, add a variable name to the `<SELECT>` element (the dropdown list) we used `name="SelectedRank"`:

```
<select asp-items="@(new SelectList(ViewBag.AvailableRanks, "All"))" name="SelectedRank">
   <option value="">All ranks</option>
</select>
```

Then add a parameter to the `Index` action (make sure it matches the name above so the model binding system will help passing it to the action), and make use of it to filter down the results:

```
//narrow down instructors based on SelectedRank:
if(!string.IsNullOrEmpty(SelectedRank) )
{
    instructors = instructors.Where(instr => instr.Rank==Enum.Parse<Ranks>(SelectedRank));
}
```

You should now be able to filter by `LastName` (see Fig. 12.8).

And also, by `Rank` (make sure to click on Filter result in order to do the filtering)—see Fig. 12.9.

Check out the URL obtained for the example above. What is the *query string* of that URL?

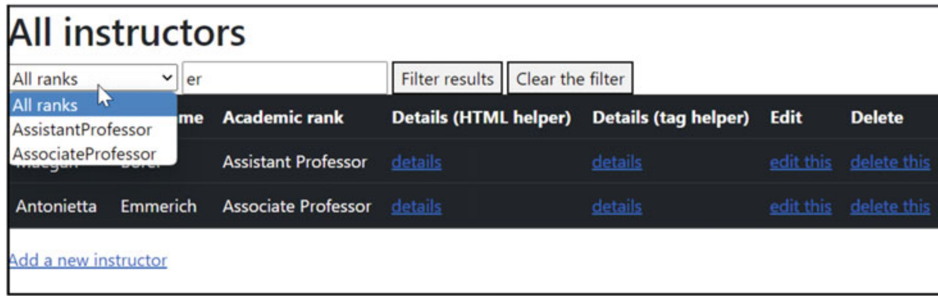


Fig. 12.8 Is similar to Fig. 12.6, but the dropdown list now only contains rank values that are contained in at least one row from the table

All instructors

All ranks | er | Filter results | Clear the filter

First name	Last name	Academic rank	Details (HTML helper)	Details (tag helper)	Edit	Delete
Antonietta	Emmerich	Associate Professor	details	details	edit this	delete this

Add a new instructor

Fig. 12.9 Shows the result of narrowing the table by rank (using the dropdown list) and last name (using the input field)

12.2.4 The Code

Here is the entire code for the `Index` *action* and the newly added `<FORM>` in the `Index` *view* so you can check your work:

```
public IActionResult Index(string SearchByLastName, string SelectedRank)
{
    //var instructors = from instr in _dbContext.Instructors
    //                  select instr; //we start with all instructors - LINQ syntax

    var instructors = _dbContext.Instructors.AsEnumerable(); //an alternative

    if (SearchByLastName != null) //narrow down our results
    {
        instructors = instructors.Where(instr => instr.LastName.Contains(SearchByLastName));
    }

    //narrow down instructors based on SelectedRank:

    if (!string.IsNullOrEmpty(SelectedRank))
    {
        instructors = instructors.Where(instr => instr.Rank == Enum.Parse<Ranks>(SelectedRank));
    }

    ViewBag.SearchByLastName = SearchByLastName; //pass this value to the view

    var AvailableRanks = from instr in instructors //get a list of available ranks
                         orderby instr.Rank
                         select instr.Rank;
    //pass the available ranks to the view - distinct only!
    ViewBag.AvailableRanks = AvailableRanks.Distinct();

    return View(instructors);
}
<form asp-action="Index" method="get">
    <select asp-items="@(new SelectList(ViewBag.AvailableRanks, "All"))" name="SelectedRank">
        <option value="">All ranks</option>
    </select>
    <input type="text" placeholder="last name includes ..." id="LastNameFilter"
                    name="SearchByLastName" value="@ViewBag.SearchByLastName" />
    <button type="submit">Filter results</button>
    <button onclick="document.getElementById('LastNameFilter').value = null">Clear the filter</button>
</form>
```

12.3 Consistent Webpages—Using Razor Layouts

A professional web application should be easy to navigate, intuitive, and with a consistent look and feel from one page to the next one. Our web application is (as of right now) far from providing a consistent user experience. In this section, we'll introduce **layouts** and see how they can be used to help us create a nice consistent style for all our webpages.

Take a look, for example, at any of the following websites:

- https://www.amazon.com/
- https://www.microsoft.com/
- https://moodle.stmartin.edu/.

As you click on various links on those sites, you should notice that all pages belonging to one website have the same look and feel. How can we achieve this?

One (bad!) option is to copy and paste the same template into every view. That should do the job, but what happens if you want to add a new link or remove an obsolete one? You would have to make those changes in every view page.

A better solution is to use **Layout** pages (see below). Let's see the steps involved in using a layout. To learn more about them, we recommend you the following source [69].

12.3.1 Create a Layout

To create *layouts*, first you need to know the location. We put them inside the folder: *Views* > *Shared*. Let's create the *Shared* folder. In the *Solution Explorer* window, right-click on *Views* folder, then select *Add* > *New Folder*.

Then, right-click on the Shared folder and select *Add* > *New Item* then select *Razor Layout*. We will use the default name (*_Layout.cshtml*), but feel free to choose another name if you prefer. Let's see what code was included in the template:

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

You should note that a layout looks pretty much like any other HTML page (and like the *views* we've seen so far, but we'll soon simplify our *views*). Two important things to note in this code:

- The title comes from `ViewBag.Title`.
  - Later, we'll **set** the title in the views, and the layout will **use** it.
- Notice the method call: `RenderBody()`.
  - This is where the contents of our *views* will be displayed.
  - If we put any links above this line, those links will be displayed on all views that make use of this layout.

That's pretty much for now. We'll come back to this once we change our views to make use of this layout.

12.3.2 Use the Layout in Our Views

12.3.2.1 The Layout Directive

To use a layout, we simply need to use the following directive in every view that uses the layout file (*_Layout.cshtml*):

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

This will make the contents of the views to be included inside the layout, at the point where the `RenderBody` function is being called. You will need to copy this directive in every view that makes use of our layout.

Side note: In this book, we only use one layout, but it is possible to define more than one layout files (one at a time)!

12.3.2.2 The Razor View Start File

Instead of copying the code above in every view, we can add that code in the *_ViewStart.cshtml* file. This file needs to be directly inside *Views* folder. In the *Solution Explorer* window, right-click on *Views* folder and select *Add > New Item ...*.

Then select the *Razor View Start* file (double-check that the name of this new file is *_ViewStart.cshtml*) and click the *Add* button. In this newly added file, make sure to have the following layout directive:

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

If you chose a different name for your layout, make sure to use that name inside the string above.

12.3.2.3 Modify View Files to Use Our Layout

Next, we'll modify all our *views* to make use of the layout created above. The strategy is as follows:

- Keep only the HTML code that is inside the `<BODY>` element (not including the `<BODY>` tags).
  - Also, keep the `@model` directive—if any.
- For each view, set a title (using the `ViewBag.Title`).
- If there are any CSS or JavaScript links, we can move them into the *_Layout* file (they will now be available to all the *views* that make use of our *layout*).

The Layout File

From the *Index.cshtml* view, we move the following links into the *_Layout.cshtml* file:

```
<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.2.3/dist/css/bootstrap.min.css" rel="stylesheet">
<script src="https://cdn.jsdelivr.net/npm/bootstrap@5.2.3/dist/js/bootstrap.bundle.min.js"></script>
```

Here is how the *layout* file looks now:

```
<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.2.3/dist/css/bootstrap.min.css" rel="stylesheet">
    <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.2.3/dist/js/bootstrap.bundle.min.js"></script>
    <title>@ViewBag.Title</title>
</head>
<body>
    <div>
        @RenderBody()
    </div>
</body>
</html>
```

The Index View

After following the steps mentioned above, the *Index* view will look as follows:

```
@model IEnumerable<Instructor>
@{
    ViewBag.Title = "Index";
}

<h1>All instructors</h1>
<form asp-action="Index" method="get">
    <select asp-items="@(new SelectList(ViewBag.AvailableRanks, "All"))" name="SelectedRank">
        <option value="">All ranks</option>
    </select>
    <input type="text" placeholder="last name includes ..."
             id="LastNameFilter" name="SearchByLastName" value="@ViewBag.SearchByLastName" />
    <button type="submit">Filter results</button>
    <button onclick="document.getElementById('LastNameFilter').value = null">Clear the filter</button>
</form>
@if (Model.Count() > 0)
{

    <TABLE class="table table-dark table-hover">
        <THEAD>
            <TR>
                <TH><label asp-for="First().FirstName"></label></TH>
                <TH><label asp-for="First().LastName"></label></TH>
                <TH><label asp-for="First().Rank"></label></TH>
                <TH>Details (HTML helper)</TH>
                <TH>Details (tag helper)</TH>
                <TH>Edit </TH>
                <TH>Delete </TH>
            </TR>
        </THEAD>
        <TBODY>

            @foreach (var instructor in Model)
            {
                <TR>
                    <TD>@Html.DisplayFor(m => instructor.FirstName) </TD>
                    <TD>@Html.DisplayFor(m => instructor.LastName)</TD>
                    <TD>@Html.DisplayFor(m => instructor.Rank)</TD>
                    <TD>@Html.ActionLink("details", "ShowDetails", new{id=@instructor.InstructorId})</TD>
                    <TD><a asp-action="ShowDetails" asp-route-id="@instructor.InstructorId">details</a> </TD>
                    <TD><a asp-action="Edit" asp-route-id="@instructor.InstructorId">edit this</a> </TD>
                    <TD><a asp-action="Delete" asp-route-id="@instructor.InstructorId">delete this</a> </TD>
                </TR>
            }
        </TBODY>

    </TABLE>
}
else
{
    <h2>No instructors found! </h2>
}
<a asp-action="Add">Add a new instructor </a>
```

The ShowDetails View

After following the steps mentioned above, the *ShowDetails* view will look as follows (isn't this much easier to read?):

```
@model Instructor
@{
    ViewBag.Title = "Instructor Details";
}

<h1>Instructor @Model.LastName details</h1>
<p><label asp-for="@Model.FirstName"></label>:  @Html.DisplayFor(m => m.FirstName)</p>
<p><label asp-for="@Model.LastName"></label>:  @Html.DisplayFor(m => m.LastName)</p>
<p><label asp-for="@Model.IsTenured"></label>: @Html.DisplayFor(m => m.IsTenured)</p>
<p><label asp-for="@Model.Rank"></label>: @Html.DisplayFor(m => m.Rank)</p>
<p><label asp-for="@Model.HiringDate"></label>: @Html.DisplayFor(m => m.HiringDate)</p>

<a asp-action="Index">go to Index</a>
@Html.ActionLink("go to Index", "Index")
```

In each view, you only need to include the data specific to that view. What needs to be repeated for all views should be put inside the layout file.

The Add View

After following the steps mentioned above, the *Add* view will look as follows:

```
@model Instructor
@{
    ViewBag.Title = "Add a new instructor";
}

<h1>Create a new Instructor</h1>
<div asp-validation-summary="All" style="color:red"></div>

<form asp-action="Add" asp-controller="Instructor" method="post">
    <label asp-for="FirstName"></label>
    <input asp-for="FirstName" />
    <span asp-validation-for="FirstName"></span>
    <br>
    <label asp-for="LastName"></label>
    <input asp-for="LastName" />
    <span asp-validation-for="LastName"></span>
    <br>
    <label asp-for="IsTenured"></label>
    <input asp-for="IsTenured" />
    <span asp-validation-for="IsTenured"></span>
    <br>
    <label asp-for="HiringDate"></label>
    <input asp-for="HiringDate" />
    <span asp-validation-for="HiringDate"></span>
    <br>
    <label asp-for="Rank"></label>
    <select asp-for="Rank" asp-items="@Html.GetEnumSelectList(typeof(Ranks))">
        <option value="">Select</option>
    </select>
    <span asp-validation-for="Rank"></span>
    <br>
    <br />
    <label asp-for="PhoneNumber"></label>
    <input asp-for="PhoneNumber" />
    <span asp-validation-for="PhoneNumber"></span>
    <br>
    <label asp-for="EmailAddress"></label>
    <input asp-for="EmailAddress" />
    <span asp-validation-for="EmailAddress"></span>
    <br>
    <label asp-for="PersonalURL"></label>
    <input asp-for="PersonalURL" />
    <span asp-validation-for="PersonalURL"></span>
    <br>
    <label asp-for="UnusedPassword"></label>
    <input asp-for="UnusedPassword" />
    <span asp-validation-for="UnusedPassword"></span>
    <br>
    <br>
    <br>
    <input type="submit" value="Create Instructor" />
</form>
```

The Edit View

After following the steps mentioned above, the *Edit* view will look as follows:

```
@model Instructor
@{
    ViewBag.Title = "Edit an instructor";
}

<h1>Edit an instructor profile</h1>
<div asp-validation-summary="All"></div>

<form asp-action="Edit" method="post">
    <input asp-for="@Model.InstructorId" type="hidden" />
    @*needed so the InstructorId is sent to the Edit(POST)*@

    <label asp-for="LastName"></label>
    <input asp-for="LastName" />
    <span asp-validation-for="LastName"></span>
    <br>
    <label asp-for="IsTenured"></label>
    <input asp-for="IsTenured" />
    <span asp-validation-for="IsTenured"></span>
    <br>
    <label asp-for="HiringDate"></label>
    <input asp-for="HiringDate" />
    <span asp-validation-for="HiringDate"></span>
    <br>
    <label asp-for="Rank"></label>
    <select asp-for="Rank" asp-items="@Html.GetEnumSelectList(typeof(Ranks))"></select>
    <br>
    <br>
    <input type="submit" value="Save changes" />
    <input type="button" onclick="history.back()" value="Cancel">
</form>
```

The Delete View

After following the steps mentioned above, the *Delete* view will look as follows:

```
@model Instructor
@{
    ViewBag.Title = "Delete an instructor";
}

<h1>Are you sure you want to delete this instructor?</h1>
<form asp-action="DeleteConfirmed" method="post">
    <input asp-for="InstructorId" type="hidden" />
    <p><label asp-for="@Model.FirstName"></label>:  @Html.DisplayFor(m => m.FirstName)</p>
    <p><label asp-for="@Model.LastName"></label>:  @Html.DisplayFor(m => m.LastName)</p>
    <p><label asp-for="@Model.Rank"></label>: @Html.DisplayFor(m => m.Rank)</p>
    <br />
    <input type="submit" value="YES, delete" />
    <input type="button" onclick="history.back()" value="NO, cancel">
</form>
```

Test Your *Work*—Then Link Our *.css* File into the Layout File

Let's test our work so far. Run your application. It should run as before. Some pages may look slightly different because now we included the links to Bootstrap 5 in our layout, hence for all our views (that use our layout).

Inside the *_Layout.cshtml*, right after the <SCRIPT> element, let's add the following link to our own CSS file (you can also drag and drop the CSS file into the layout file, Visual Studio will "drop" the link below for you):

```
<link href="~/css/personal.css" rel="stylesheet" />
```

Now, when you run your application, you should see all files using the same lightblue background color (and other elements included in the .css file):

The Index view (see Fig. 12.10).

The Add view (see Fig. 12.11).

Fig. 12.10 Shows the result of adding styling to the Index view

Fig. 12.11 Shows the result of adding styling to the Add view

And so on. On your own, you should play with the *personal.css* file and modify it so your web application looks better.

IMPORTANT: If you make modifications to the CSS file, and they are not reflected in your webpages, make sure to press Ctrl + F5 inside your browser, to make your browser download the latest version of the .css files. Otherwise, your browser may try to use a cached version of the .css files (the ones before you made modifications).

12.3.3 Add a Bootstrap 5 Navbar to Our Layout

Next, to see how great layout files are, we'll add a navigation bar to our *layout* file. This will in turn be used on all *views* that make use of our *layout* file.

Note: Instead of having to add a layout in every view individually, we only need to do it once, in the *layout*. Isn't this great?

Let's start by adding the navbar we saw in Chap. 4 into our layout (to learn more about navbars, use [25]). Copy and paste that navbar into the *_Layout.cshtml* file. The layout file should now look as follows:

```
<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.2.3/dist/css/bootstrap.min.css" rel="stylesheet">
    <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.2.3/dist/js/bootstrap.bundle.min.js"></script>
    <link href="~/css/personal.css" rel="stylesheet" />
    <title>@ViewBag.Title</title>
</head>
<body>
    <NAV class="navbar navbar-expand-sm bg-dark navbar-dark">
        <DIV class="container-fluid">
            <UL class="navbar-nav">
                <LI class="nav-item">
                    <A class="nav-link" href="firstwebpage.html">First page</A>
                </LI>
                <LI class="nav-item">
                    <A class="nav-link" href="secondwebpage.html">Second page</A>
                </LI>
                <LI class="nav-item">
                    <A class="nav-link" href="register.html">Register</A>
                </LI>
                <LI class="nav-item">
                    <A class="nav-link" href="https://www.stmartin.edu/">Saint Martin's University</A>
                </LI>
                <LI class="nav-item">
                    <A class="nav-link" href="https://www.w3schools.com/">W3Schools</A>
                </LI>
            </UL>
        </DIV>
    </NAV>

    <div>
        @RenderBody()
    </div>
</body>
</html>
```

Run your application. Your navbar should be on every page that uses the layout. Your pages have a more consistent look and feel (Fig. 12.12).

12.3.4 Add Navigation Links to Various Actions and Controllers

Next, we'll fix the links in our navbar. For this, we will make use of *tag helpers*.

IMPORTANT: The tag helpers used in the navbar should include the name of the *controller* for each *action*; otherwise, going from one controller's action to another controller's action won't work (as you would expect).

Here is how our navbar looks after adding links to various actions:

Fig. 12.12 Shows the effects of adding a navbar to the layout. In here we see again the Index view

```
<NAV class="navbar navbar-expand-sm bg-dark navbar-dark">
    <DIV class="container-fluid">
        <UL class="navbar-nav">
            <LI class="nav-item">
                <A class="nav-link" href="https://www.stmartin.edu/">Saint Martin's
University</A>
            </LI>
            <LI class="nav-item">
                <A class="nav-link" href="https://www.w3schools.com/">W3Schools</A>
            </LI>
            <LI class="nav-item">
                <A class="nav-link" href="https://learn.microsoft.com/en-
us/aspnet/core/mvc/overview?view=aspnetcore-6.0">Learn ASP.NET Core MVC</A>
            </LI>
            <LI class="nav-item dropdown">
                <a class="nav-link dropdown-toggle" role="button" data-bs-
toggle="dropdown">Instructors</a>
                <UL class="dropdown-menu">
                    <LI><a class="dropdown-item" asp-controller="Instructor" asp-
action="Index">List all instructors</a></LI>
                    <LI><a class="dropdown-item" asp-controller="Instructor" asp-
action="Add">Add a new instructor</a></LI>
                </UL>
            </LI>
            <LI class="nav-item dropdown">
                <a class="nav-link dropdown-toggle" role="button" data-bs-
toggle="dropdown">Students</a>
                <UL class="dropdown-menu">
                    <LI><a class="dropdown-item" asp-controller="Student" asp-
action="Index">List all students</a></LI>
                    <LI><a class="dropdown-item" asp-controller="Student" asp-action="Add">Add
a new student</a></LI>
                </UL>
            </LI>
        </UL>
    </DIV>
</NAV>
```

Now you should have a functional navigation bar that can be used for all views that use our layout file (see Fig. 12.13).

You should also note that one of the reasons why you should use the Bootstrap framework is that it includes responsive design. To see this, narrow down the webpage above. See what happens to the navbar if it does not have sufficient width to display all options side by side (Fig. 12.14).

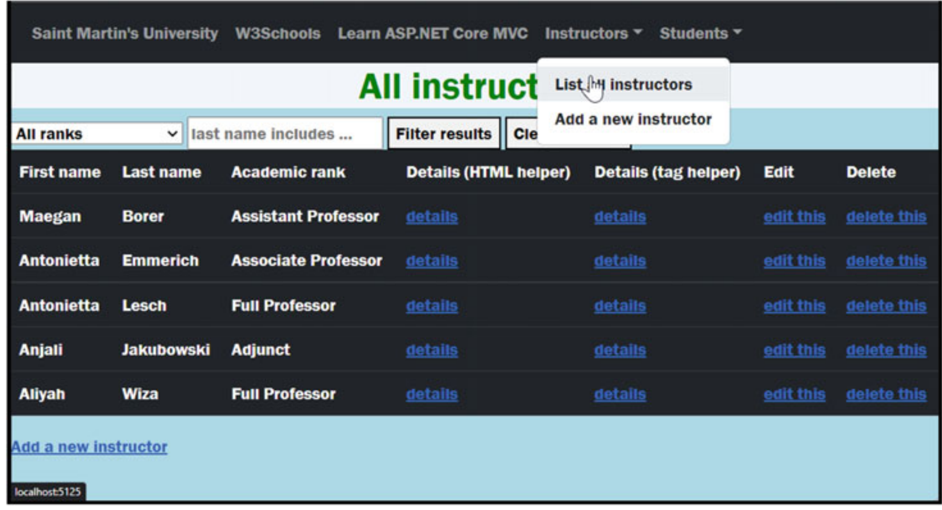


Fig. 12.13 Shows the same view as seen in Fig. 12.12, but the navbar contains two more dropdown menu options (Instructors and Students)

12.4 Layout Sections (Optional)

Next, we would like to introduce layout *sections*. Think of **sections** as pieces of code that can be defined by each view individually, but it's up to the layout to decide where (and how) to display these sections.

We'll demonstrate sections by solving the following: we want to let each view define certain links that will appear at the bottom of the page (later we could move them if needed) and we will place them centered.

12.4.1 Define a Section

To define a section, a view will use the format: `@section SectionName{contents …}`.

Note: It does not matter where inside the section is defined inside a view. It's up to the layout where it gets displayed.

12.4.1.1 The Index View (For the InstructorController Class)

Will not define such a section. One may be added later.

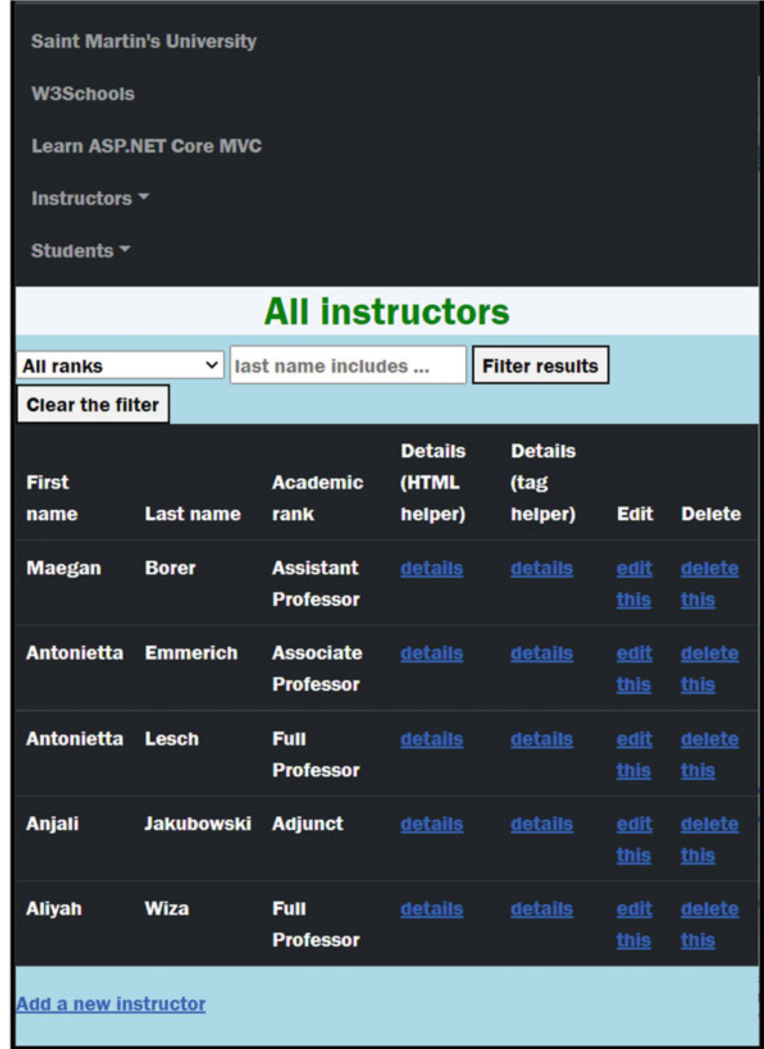


Fig. 12.14 Shows the same Index view as seen in Fig. 12.13, but the nav bar is responsive to the actual width of the window. If you narrow the window width sufficiently, the menu options will show up stacked (in the upper left corner)

12.4.1.2 The Add View (For the InstructorController Class)

Let's add the following (just for practice)—you may remove unnecessary links if you want:

```
@section OurFooterLinks{
    <input type="button" value="GO BACK" onclick="history.back()" />
}
```

12.4.1.3 The ShowDetails View (For the InstructorController Class)

Let's add the following (just for practice)—you may remove unnecessary links if you want:

```
@section OurFooterLinks{
    <input type="button" value="GO BACK" onclick="history.back()" />
    <a asp-action="Edit" asp-controller="Instructor" asp-route-id="@Model.InstructorId">EDIT</a>
    <a asp-action="Delete" asp-controller="Instructor" asp-route-id="@Model.InstructorId">DELETE</a>
}
```

12.4.1.4 The Edit, Delete Views (For the InstructorController Class)

Let's add the following (just for practice):

```
@section OurFooterLinks{
    <input type="button" value="GO BACK" onclick="history.back()" />
    <a asp-action="ShowDetails" asp-controller="Instructor" asp-route-id="@Model.InstructorId">SEE DETAILS</a>
}
```

12.4.2 Make Use of a Section

To display *section* in the *layout*, use `@RenderSection` in the layout (at the location where you want it displayed—we'll put them at the end of the _Layout.cshtml, right before the `</BODY>` end tag).

```
@RenderSection("OurFooterLinks", false)
```

The second parameter is needed, it tells the compiler that not all views have a section named "`OurFooterLinks`" defined. Setting that value to true would force your application to define that section in every view that uses our layout, or get a compiling error.

Lastly, we would like the buttons defined in the `OurFooterLinks` section to be centered and displayed one line below their current position. One way to do this is as follows. Add a `<DIV>` element, and nest the `RenderSection` directive inside a `<DIV>` element, then apply the *class* selector (which we already defined in our *personal.css* file):

```
<BR>
<DIV id="CenterFooterLinks">
    @RenderSection("OurFooterLinks", false)
</DIV>
```

To remind you, our *personal.css* file already contains

```
.CenterFooterLinks {
    text-align: center;
}
```

Now run your application and check the changes. Make sure to press Ctrl + F5 if the latest version of the personal.css is not yet loaded in the browser.

12.5 Make Use of Bootstrap 5 Buttons

We have already seen in Chap. 4 how to make use of Bootstrap 5 buttons, and how we can also use Bootstrap to make links look like buttons. Let's make use of these to improve the look of our web application.

12.5.1 The Index View

In particular (see Fig. 12.15), we would like to change the appearance of this view (we'll provide the solution at the end of this section):

Add the following to all our <a> elements: `class="btn btn-primary"` defined inside *Index.cshtml*. Now the page looks a little better (see Fig. 12.16).

Let's remove the Details (HTML helper) column. Remove the following lines:

```
<TH>Details (HTML helper)</TH>
<TD>@Html.ActionLink("details", "ShowDetails", new{id=@instructor.InstructorId})</TD>
```

And rename the columns *Details (tag helper)* to *Details*. See the result in Fig. 12.17.

Lastly, let's use the various colors available to Bootstrap 5 buttons.

- For the *details* button, add the class: `btn-success`
- For the *edit* button, add the class: `btn-warning`
- For the *delete* button, add the class: `btn-danger`

Here is how the view looks now (see Fig. 12.18).

Here is the entire code for this view:

Saint Martin's University W3Schools Learn ASP.NET Core MVC Instructors Students

All instructors

All ranks | last name includes ... | Filter results | Clear the filter

First name	Last name	Academic rank	Details (HTML helper)	Details (tag helper)	Edit	Delete
Maegan	Borer	Assistant Professor	details	details	edit this	delete this
Antonietta	Emmerich	Associate Professor	details	details	edit this	delete this
Antonietta	Lesch	Full Professor	details	details	edit this	delete this
Anjali	Jakubowski	Adjunct	details	details	edit this	delete this
Aliyah	Wiza	Full Professor	details	details	edit this	delete this

Add a new instructor

Fig. 12.15 Shows the current state of the Index view

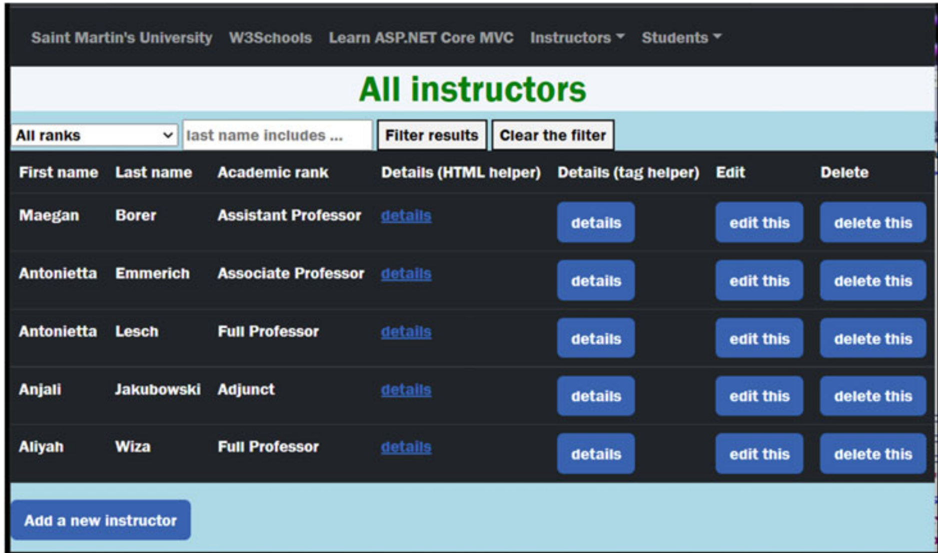


Fig. 12.16 The Index view looks similar to the one in Fig. 12.15, but some of the links look like buttons

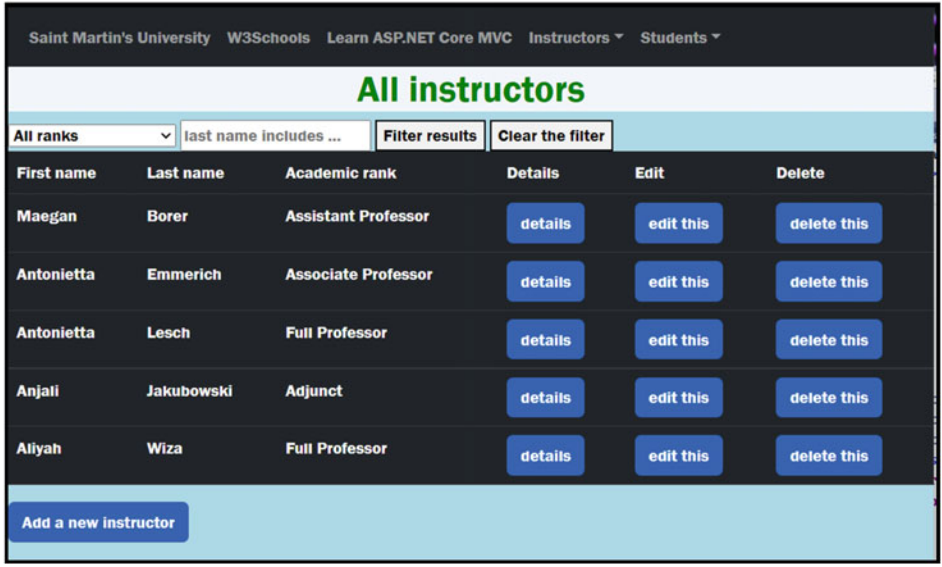


Fig. 12.17 Shows the result of removing the Details (HTML helper) column

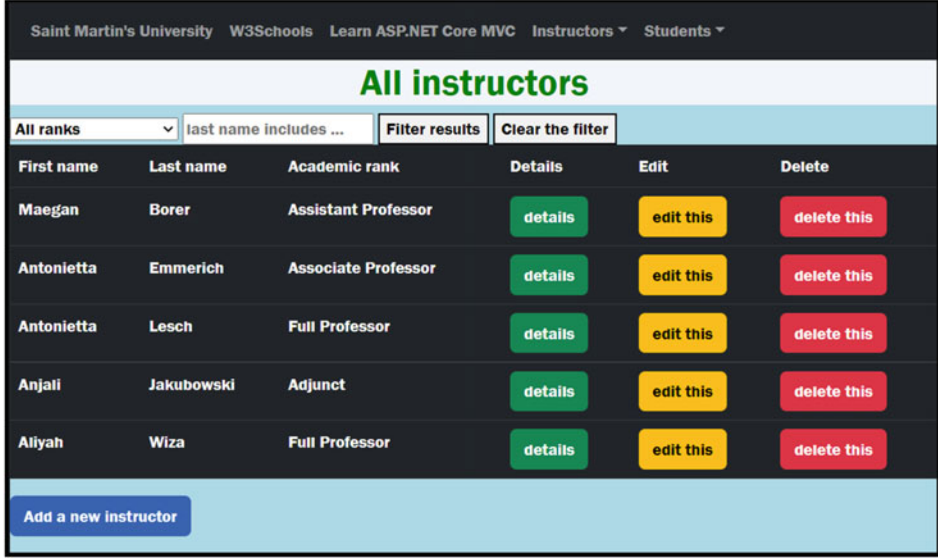


Fig. 12.18 Shows the same contents as seen in Fig. 12.17, to which we added various Bootstrap 5 colors (we used green for the details buttons, yellow for the edit buttons, and red for the delete buttons)

```
@model IEnumerable<Instructor>
@{
    ViewBag.Title = "Index";
}

<h1>All instructors</h1>
<form asp-action="Index" method="get">
    <select asp-items="@(new SelectList(ViewBag.AvailableRanks, "All"))" name="SelectedRank">
        <option value="">All ranks</option>
    </select>
    <input type="text" placeholder="last name includes ..." id="LastNameFilter" name="SearchByLastName"
value="@ViewBag.SearchByLastName" />
    <button type="submit">Filter results</button>
    <button onclick="document.getElementById('LastNameFilter').value = null">Clear the filter</button>
</form>
@if (Model.Count() > 0)
{

    <TABLE class="table table-dark table-hover">
        <THEAD>
            <TR>
                <TH><label asp-for="First().FirstName"></label></TH>
                <TH><label asp-for="First().LastName"></label></TH>
                <TH><label asp-for="First().Rank"></label></TH>
                <TH>Details</TH>
                <TH>Edit </TH>
                <TH>Delete </TH>
            </TR>
        </THEAD>
        <TBODY>

            @foreach (var instructor in Model)
            {   <TR>
                    <TD>@Html.DisplayFor(m => instructor.FirstName) </TD>
                    <TD>@Html.DisplayFor(m => instructor.LastName)</TD>
                    <TD>@Html.DisplayFor(m => instructor.Rank)</TD>
                    <TD><a asp-action="ShowDetails" asp-route-id="@instructor.InstructorId" class="btn btn-success">details</a>
</TD>
                    <TD><a asp-action="Edit" asp-route-id="@instructor.InstructorId" class="btn btn-warning">edit this</a> </TD>
                    <TD><a asp-action="Delete" asp-route-id="@instructor.InstructorId" class="btn btn-danger">delete this</a>
</TD>
                </TR>
            }
        </TBODY>

    </TABLE>
}
else
{
    <h2>No instructors found! </h2>
}
<a asp-action="Add" class="btn btn-primary">Add a new instructor </a>
```

12.5.2 The ShowDetails View

Here is the current state of this view (see Fig. 12.19).

Let's remove the go to index links (we already have links in the navbar for the *Index* action). Add CSS classes to the GO BACK, EDIT, and DELETE (as seen in the previous section).

Here is the solution so far:

```
<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.2.3/dist/css/bootstrap.min.css" rel="stylesheet">
    <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.2.3/dist/js/bootstrap.bundle.min.js"></script>
    <link href="~/css/personal.css" rel="stylesheet" />
    <title>@ViewBag.Title</title>
</head>
<body>
    <NAV class="navbar navbar-expand-sm bg-dark navbar-dark">
        <DIV class="container-fluid">
            <UL class="navbar-nav">
                <LI class="nav-item">
                    <A class="nav-link" href="https://www.stmartin.edu/">Saint Martin's University</A>
                </LI>
                <LI class="nav-item">
                    <A class="nav-link" href="https://www.w3schools.com/">W3Schools</A>
                </LI>
                <LI class="nav-item">
                    <A class="nav-link" href="https://learn.microsoft.com/en-us/aspnet/core/mvc/overview?view=aspnetcore-
6.0">Learn ASP.NET Core MVC</A>
                </LI>
                <LI class="nav-item dropdown">
                    <a class="nav-link dropdown-toggle" role="button" data-bs-toggle="dropdown">Instructors</a>
                    <UL class="dropdown-menu">
                        <LI><a class="dropdown-item" asp-controller="Instructor" asp-action="Index">List all
instructors</a></LI>
                        <LI><a class="dropdown-item" asp-controller="Instructor" asp-action="Add">Add a new instructor</a></LI>
                    </UL>
                </LI>

                <LI class="nav-item dropdown">
                    <a class="nav-link dropdown-toggle" role="button" data-bs-toggle="dropdown">Students</a>
                    <UL class="dropdown-menu">
                        <LI><a class="dropdown-item" asp-controller="Student" asp-action="Index">List all students</a></LI>
                        <LI><a class="dropdown-item" asp-controller="Student" asp-action="Add">Add a new student</a></LI>
                    </UL>
                </LI>
            </UL>
        </DIV>
    </NAV>


    <div>
        @RenderBody()
    </div>

    <BR>
    <DIV class="CenterFooterLinks">
        @RenderSection("OurFooterLinks", false)
    </DIV>

</body>
</html>
```

Challenge: how would you center these contents? Here is the end result (Fig. 12.20).

We'll let you change the other views on your own, but here are some suggested outcomes.

For the *Add* view (see Fig. 12.21).

For the *Edit* view (see Fig. 12.22).

And the *Delete* view (see Fig. 12.23).

12.5.3 Use Bootstrap for Styling Validation Errors

Lastly, in this part, we would like to make all our Validation errors show in red (see Fig. 12.24).

Fig. 12.19 Shows the current state of the ShowDetails view for InstructorController

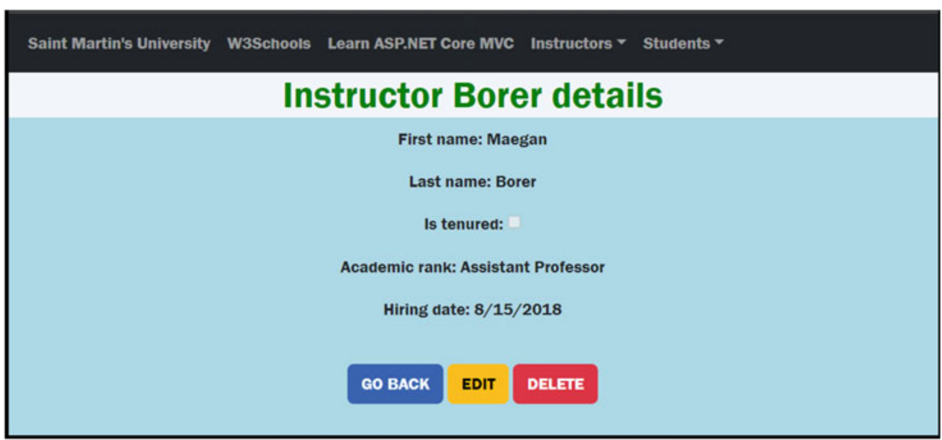


Fig. 12.20 Shows how we would like to make the ShowDetails view look after adding (Bootstrap 5) styling

Using the developer tool (press F12 then go to the *Elements* tab), you will find that the errors are displayed in elements having the following CSS classes:

- For the summary: `validation-summary-errors`
- For the individual fields: `field-validation-error`

Fig. 12.21 Shows how we would like to make the Add view to look after adding (Bootstrap 5) styling

Fig. 12.22 Shows how we would like to make the Edit view to look after adding (Bootstrap 5) styling

Therefore, we just need to add the following to our *personal.css* file:

```
.field-validation-error , .validation-summary-errors {
    color: red;
}
```

And now the errors show up with red text (see Fig. 12.25).

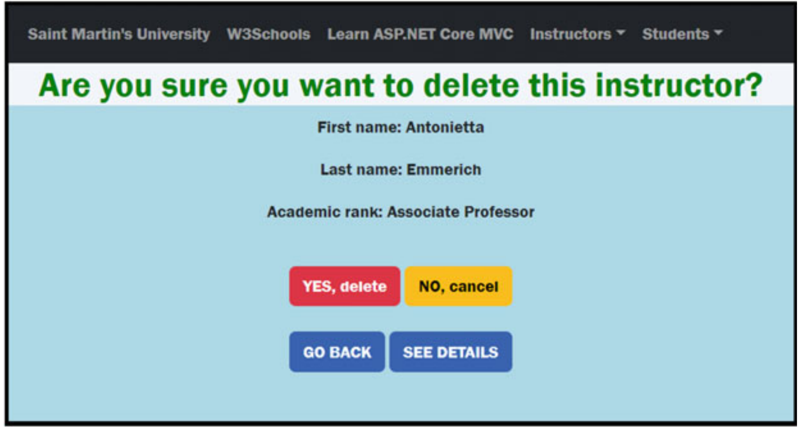


Fig. 12.23 Shows how we would like to make the Delete view to look after adding (Bootstrap 5) styling

Fig. 12.24 Shows how we would like to make the Add view to look after adding (Bootstrap 5) styling

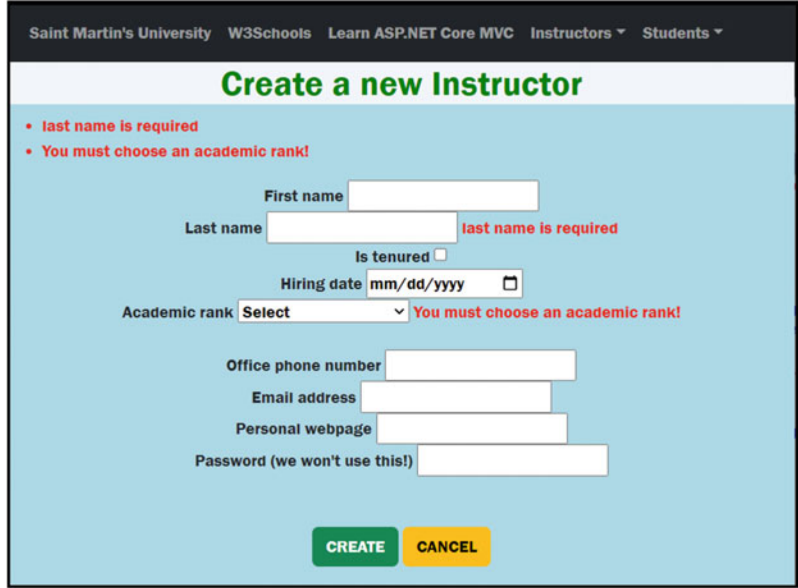


Fig. 12.25 Shows the same image as in Fig. 12.24, but now that we added the styling shown above, the errors show up using a red text

12.6 Configure a Friendly Error Page

12.6.1 Introduction

What happens if you send an unexpected HTTP request to a website such as Amazon.com, Microsoft.com, or stmartin.edu? On your own, try out the following:

- https://www.amazon.com/MEZEI
- https://www.microsoft.com/MEZEI
- http://www.stmartin.edu/MEZEI

In each of these cases, you should note that a nice error page is created, most importantly one that has a similar layout as the other pages on the same website. Some websites even include random images displayed in their error page.

What happens if our application encounters an error? Or receives an unexpected HTTP request? Try, for example,

- http://localhost:5125/MEZEI

We got the HTTP ERROR 404 page.

This isn't as nice as the friendly error page seen above. Next, we will create a friendly error page that uses our layout file, but first, let's talk about using multiple environments in an ASP.Net Core application.

12.6.2 Work with Multiple Environments

"ASP.NET Core configures app behavior based on the runtime environment using an environment variable" (read more in [70, 71]). In particular, we can use the environment variable `ASPNETCORE_ENVIRONMENT` to specify if our application is in production, in development, or in some (any) other stage. Then, we can use code to check which environment we are in and act appropriately.

There are multiple ways to set the value of the `ASPNETCORE_ENVIRONMENT` environmental variable. We will change it as follows: from *Solution Explorer* window open the file *Properties* > *launchSettings.json*. In there, one can set the value of the `ASPNETCORE_ENVIRONMENT` environmental variable to the desired stage. Right now, it is set to "`Development`". We will later switch it to "`Production`", but please note that you can use any other values too, for example, "`Staging`", etc.

```
"profiles": {
  "ASPBookProject": {
    "commandName": "Project",
    "dotnetRunMessages": true,
    "launchBrowser": true,
    "applicationUrl": "http://localhost:5125",
    "environmentVariables": {
      "ASPNETCORE_ENVIRONMENT": "Development"
    }
  },
```

We will make use of this environment variable, so our application behaves differently in various environments:

- During ***development***, if an error occurs, we would like to get as many details as possible about the error.
- During ***production***, we want to hide all details about the error and instead use a nice friendly error page (see Fig. 12.26).

Here is how we'll use the `ASPNETCORE_ENVIRONMENT` environmental variable, in the middleware pipeline. In *Program.cs*, right after

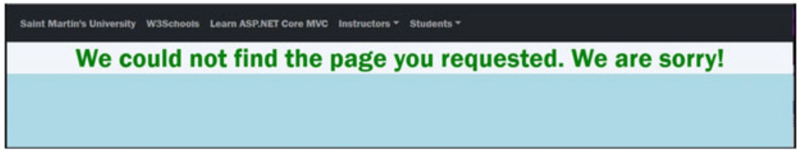


Fig. 12.26 Shows a simple error page that only contains some generic text, without including all details of the error

```
var app = builder.Build();//set up middleware components.
```

add the following code (explained in the next two subsections).

```
if(app.Environment.IsDevelopment())
{
    app.UseDeveloperExceptionPage();               //show all details for errors
}
else
{
    app.UseExceptionHandler("/Error/Index");  //show a friendly page, hide all details
    app.UseStatusCodePagesWithRedirects("/Error/Index"); //include HTTP Error codes too
}
```

Note: The `app.Environment` can be used to check against for environment, not just the `Development`. After `app.Environment` type a dot, and IntelliSense (from Visual Studio) will show you several options including *IsDevelopment*, *IsEnvironment*, *IsProduction*, and *IsStaging*.

To check if the current environment is "`TEST`", you could use code similar to:

```
if(app.Environment.IsEnvironment("TEST"))
```

12.6.3 The Developer Exception Page

During the *development* stage, we would like to get as many details as possible about errors. For this, we will use the following middleware component: `app.UseDeveloperExceptionPage();`

You should only be using this for the *Development environment* (or related stages, such as Testing, Code Review, …) because (as you will see below) the text generated by this middleware pipeline can include portions of your source code, which is something you do not want to disclose to your clients, especially to potential attackers.

For example, we could use the following in *Program.cs*:

```
var app = builder.Build();//set up middleware components.
if (app.Environment.IsDevelopment())
{
    app.UseDeveloperExceptionPage();               //show all details for errors
}
else
{
...
```

To test this, make sure to set the environment variable: "ASPNETCORE_ENVIRONMENT": "Development". Then, go to the Index action of the InstructorController class, and add the following line of code (to generate an error) as the very first statement of this method:

```
throw new Exception("testing the error page");
```

Run your application. You should get a response page containing many details about the error page. In particular, you can get the *Stack* information, the *Query* information, *Cookies*, *Headers*, and *Routing* information. You can even get the line number responsible for the Exception that was created.

Before you continue, please remove the line we just added (throw new Exception("testing the...");).

Another way to test this is by renaming, let's say the *Index.cshtml* file to *Index2.cshtml*. You should get a similar error page full of details (as seen above).

Rename back the Index view before you continue!

You should note that some errors (such as HTTP 404—not found) do not have such detailed HTTP responses. For example, if you use the following HTTP request: http://localhost:5125/MEZEI you will get the HTTP ERROR 404 page.

12.6.4 The Friendly Error Page

Next, we'll configure the friendly error page, so that when an error occurs, the user gets a nice friendly response. Then, we'll go over the same errors as seen above and check the outcome.

During the *production* stage, we would like to get no details displayed to the user, just that an error occurred, and use our layout file, so the user still has access to the other links for an easier navigation. For this, we will use the following middleware components:

```
app.UseExceptionHandler("/Error/Index");  //show a friendly page, hide all details
app.UseStatusCodePagesWithRedirects("/Error/Index"); //for responses such as 404 Not found
```

If you hover your mouse over UseExceptionHandler method (in *Program.cs*), you will find that this method is a middleware component that will catch exceptions and reset the request path. You can do the same for the other method.

Make sure in the *Program.cs* you are using an if statement to check which environment is being used.

```
if (app.Environment.IsDevelopment())
{
    app.UseDeveloperExceptionPage();                //show all details for errors
}
else
{
    app.UseExceptionHandler("/Error/Index");  //show a friendly page, hide all details
    app.UseStatusCodePagesWithRedirects("/Error/Index"); //for responses such as 404 Not found
}
```

Now, let's explain the string "/Error/Index". One way to prepare a friendly error page is to create an ErrorController class, with an *action* (we used Index above) and a corresponding *view*. Then, use the path "/Error/Index" to access it. Note: for default routing, we don't need to specify the /Index portion. We could just use /Error.

To our project, let's add a new controller class, called ErrorController. Here are the contents of *ErrorController.cs*:

```
using Microsoft.AspNetCore.Mvc;

namespace ASPBookProject.Controllers
{
    public class ErrorController : Controller
    {
        public IActionResult Index()
        {
            return View();
        }
    }
}
```

Next, add a view for the *Index* action. This time, we can select *Razor View—Empty* (you can also select the other option, but it will generate more code than we need, so rather than delete extra code, we chose the Empty option this time).

In the *Add New Item* window that opens, make sure to use the name that matches the action's name, and click on the *Add* button.

Since we are using a layout and we included the following in *_ViewStart.cshtml*:

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

our view will make use of the layout. So, in our view, we will only include the following line:

```
<h1>We could not find the page you requested. We are sorry!</h1>
```

Feel free to also add some friendly images if you wish.

To test this, make sure to set the environment variable: "ASPNETCORE_ENVIRONMENT": "Production". Then, go to the Index action of the InstructorController class, and add the following line of code (to generate an error):

```
throw new Exception("testing the error page");
```

Run your application. You should get a response similar to Fig. 12.26.

Isn't this better? Before you continue, please remove the line we just added (throw new Exception("testing the...");).

Another way to test this is by renaming, let's say the *Index.cshtml* file to *Index2.cshtml*. You should get the same friendly page as above. Please rename back the *Index* view before you continue!

Now let's try the following HTTP request: http://localhost:5125/MEZEI you will again get something similar to Fig. 12.26.

If you wish, you can also display a random image in the error page generated above. One solution is to use the images we already have in our wwwroot > images (in our example, we have the following files: *image01.JPG*, *image02.JPG*, *image03.JPG*, *image04.JPG*, *image05.JPG*, *image06.JPG*, *image07.JPG*).

Each time a View is generated, we should randomly generate a value between 1 and 7 (since we have 7 images) and display one of them.

```
<h1>We could not find the page you requested. We are sorry!</h1>

<div style="text-align:center">
    @{
        Random randGenerator = new Random();
        int x = randGenerator.Next(1, 8);
        string imagePath = $"/images/image0{x}.JPG";
    }
    <br />

    <img src="@imagePath" width="50%" alt="error image" class="img-thumbnail" />

</div>
```

Now, each time you get the friendly error page displayed, a randomly chosen image will show up (see Figs. 12.27 and 12.28). Note the URL: localhost:5125/Error/Index.

That's it for us. ASP .Net Core has many more related topics that we did not cover, please read more in [70, 71].

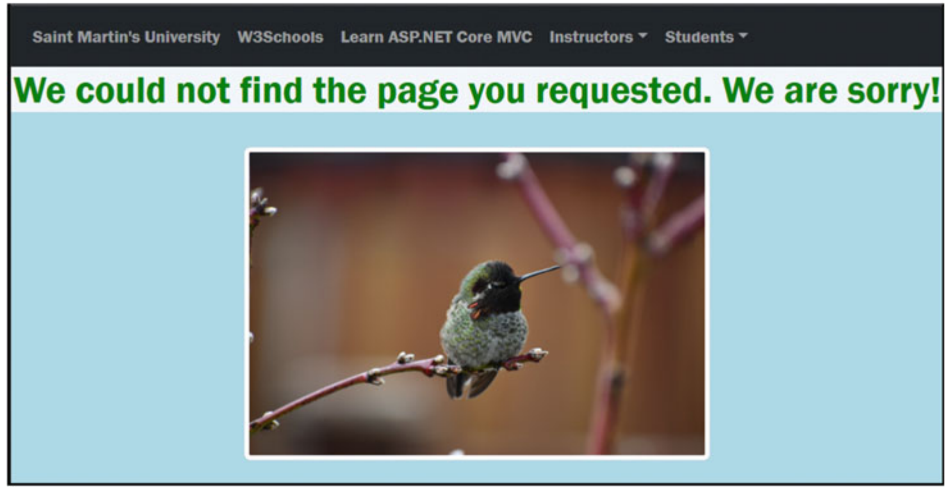


Fig. 12.27 Shows the error page seen similar to Fig. 12.26, but it now includes a randomly selected image from web root

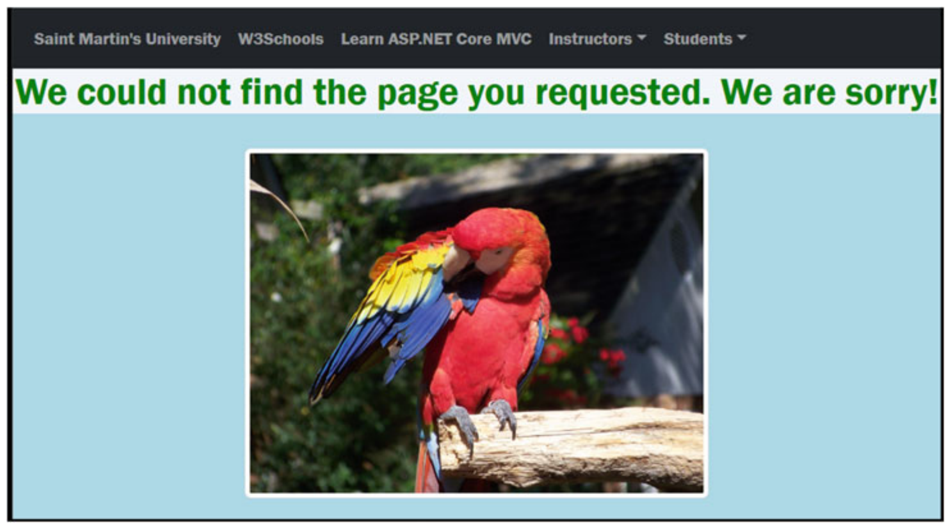


Fig. 12.28 Is similar to Fig. 12.27, but it now includes another randomly selected image

Before we continue, please make sure to set your environment back to `Development`. Otherwise, it will be more challenging to debug potential errors.

Work with Images (Optional) 13

In here, we would like to improve our web application so we can optionally store (in our database) a profile photo for every instructor and also display it. We recommend reading the following source for this chapter [72].

Important note: This is a very simplistic example, meant to show the steps involved in saving images into a database and retrieving them. We did not include any security considerations in this code. You are strongly encouraged to consider the security implications and make use of mitigating techniques. Here is one resource which we did not use but you could use as a starting point [73].

13.1 Add a New Property for the Image to the Model/Entity Class

We'll start by adding one new property in the `Instructor` class that will be used to store the image. We'll store the image as an array of bytes since SQLite does not have a type that can be mapped to images.

The long story short (see below for more details) is that we will store images as an array of bytes, then we will convert them back to images when we want to display them.

Go to the *Instructor.cs* class and add the following property:

```
[Display(Name ="Profile photo")]
public byte[]? InstructorProfilePhoto { get; set; }
```

Important: Since we are adding a new field to an existing table, we will make sure to enable the `EnsureDeleted` method in *Program.cs*, so our database is recreated (we added the following lines right before: the line containing **`app.UseStaticFiles();`**).

```
var context = app.Services.CreateScope().ServiceProvider.GetRequiredService<OurDbContext>();
context.Database.EnsureDeleted();//if our database exists, then erase it!
context.Database.EnsureCreated();//if our database does not exist, then create it!
```

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024

R. A. Mezei, *Introduction to the Development of Web Applications Using ASP .Net (Core) MVC*, Synthesis Lectures on Computer Science,
https://doi.org/10.1007/978-3-031-30626-6_13

Run again the application. A new column will be created in the `Instructors` table (use DbBrowser to see the columns from *Instructors* table): the new column name is **`InstructorProfilePhoto`**.

You can now comment out the line containing `EnsureDeleted`, so data modifications remain saved (so the database tables don't get recreated each time we run the application).

For the remaining part of this chapter, we'll follow each step of an *HTTP request* and modify the code so it allows the user to upload an image, transform it into a byte array, and then save it into the database. Then, we'll follow the request from getting the byte array from the database, change it back into an image, and display it in a browser.

13.2 Modify the Add View, so It Allows a User to Upload an Image

In here, we'll add the code needed so we allow the user to upload an image (any file actually, not just images).

To allow a form to upload files to the server, you need to add the following inside the `<FORM>` tag:

```
enctype="multipart/form-data"
```

After adding the code above, the `<FORM>` tag, inside the *Add.cshtml* file, looks as follows:

```
<form asp-action="Add" asp-controller="Instructor" method="post" enctype="multipart/form-data">
```

Then, we need to add input field that allows the user to select an image from their computer and upload it with the form. Before the `submit` button, add the following:

```
<input type="file" name="InstructorImageFile" />
<br>
<br>
```

Remember, in order for data from input fields to be sent to the server, we need to use a `name` attribute.

To test what we accomplished so far, go to the Add action: http://localhost:5125/Instructor/Add. You should see a Choose File button that allows the user to select a file from their own computer (see Fig. 13.1).

If you click on this button, a new window opens that allows you to select a file. Once selected, you'll see its name next to the button (see Fig. 13.2):

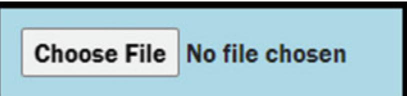


Fig. 13.1 Shows how the Choose File button should show up in your browser

Fig. 13.2 Shows similarly to Fig. 13.1, but now a file has been selected

Side note: in _Layout.cshtml, we enclosed the following lines:

```
<DIV>
    @RenderBody()
</DIV>


<BR>
<DIV id="CenterFooterLinks">
    @RenderSection("OurFooterLinks", false)
</DIV>
```

Inside (we just added these lines):

```
<DIV class="container-fluid p-5 ">

…

</DIV>
```

This way we added some padding around our view contents. See more in here [16].

13.3 Modify the Add Action so the File Uploaded Gets Saved into the Database

In the `Add` action (the POST) is where we receive user data and save it into the database. We now need to grab the file sent by the user, convert it into a byte array, and save it into the database, along with other data.

Add the following code after validating the user data:

```
//if a file/image was uploaded, convert it to byte[] and save it
if(Request.Form.Files.Count >0) //did the user upload a file?
{
    var file = Request.Form.Files[0];//our view only allows one file

    MemoryStream ms = new MemoryStream();
    file.CopyTo(ms); //copy the file into a memory stream object
    newInstructor.InstructorProfilePhoto = ms.ToArray();//save the bytes into newInstructor

    ms.Close();
    ms.Dispose();
}
```

In this code, we first check if the user uploaded a file. Since we only allow one file to be uploaded at a time, we read the file at index 0 (otherwise we could use a for each to read each file). Then, using a `MemoryStream` object we convert the file into a byte array and save it into the `InstructorProfilePhoto` property. This will then be saved into the database, along with other values.

The `Add` action should now look as follows:

```
[HttpPost]
public IActionResult Add(Instructor newInstructor)
{
    if (!ModelState.IsValid) //if the data is invalid
        return View(); //go back to the view

    //if a file/image was uploaded, convert it to byte[] and save it
    if(Request.Form.Files.Count >0) //did the user upload a(ny) file?
    {
        var file = Request.Form.Files[0];//our view only allows one file

        MemoryStream ms = new MemoryStream();
        file.CopyTo(ms); //copy the file into a memory stream object
        newInstructor.InstructorProfilePhoto = ms.ToArray();//save the bytes into newInstructor

        ms.Close();
        ms.Dispose();
    }

    _dbContext.Instructors.Add(newInstructor); //add the new instructor to our list
    _dbContext.SaveChanges();
    return RedirectToAction("Index");
}
```

To test this work, go ahead and add/create a new Instructor. Make sure to select an image and enter some data (see Fig. 13.3).

After you click on the *Create Instructor* button, go ahead and check that it was saved into the database. In particular, using DbBrowser, open your database and check that the `InstructorProfilePhoto` column shows BLOB for the newly added instructor (the last row).

With the code we have so far, we are now able to save data into the database. Next, we'll retrieve this data and display it in a view.

Fig. 13.3 Shows the Add view, where a user can enter information for a new instructor, and also select an image/file to be uploaded to the server

13.4 Modify the ShowDetails Action to Transform the Byte Array Back into an Image

Inside the `ShowDetails` action, change the following two lines:

```
if(instr!=null) //was an instructor found?
    return View(instr);
```

into

```
if(instr!=null) //was an instructor found?
{
    if(instr.InstructorProfilePhoto!= null) //is there an image on file?
    {
        string imageBase64Data = Convert.ToBase64String(instr.InstructorProfilePhoto);
        string imageDataURL = string.Format("data:image/jpg;base64,{0}", imageBase64Data);
        ViewBag.InstructorProfilePhoto = imageDataURL;
    }
    return View(instr);
}
```

More specifically, in here we first check if we have an image (a byte array really) in the `InstructorProfilePhoto` property. If we do, we convert the byte array into an image and we provide the location of this new image to the view via the `ViewBag` object.

13.5 Modify the ShowDetails View so It Displays the Profile Image

In the view, we first check if we have an image URL saved in `ViewBag`. If we do, we display it. Add the following to the `ShowDetails` view (wherever you would like to display the image):

```
@if (ViewBag.InstructorProfilePhoto != null)
{
 <img src="@ViewBag.InstructorProfilePhoto" width="20%"
      alt="profile picture for @Model.LastName"
      class="img-thumbnail" />
}
```

Optionally, we can display a default photo for Instructors who do not have an image saved. For this, add

```
else
{
<img src="~/images/noprofilephoto.png" width="20%"
      alt="generic profile photo"
      class="img-thumbnail" />
}
```

where *noProfilePhoto.png* is an image that we created and added inside *wwwroot>images* folder.

That's it! Let's test our work. If we go to `ShowDetails` for an Instructor for which we did not provide a profile image, here (see Fig. 13.4) is what we see (note: we "cleaned" our *personal.css* file for a better appearance):

Now, let's check our instructor for which we did provide a profile image (see Fig. 13.5):

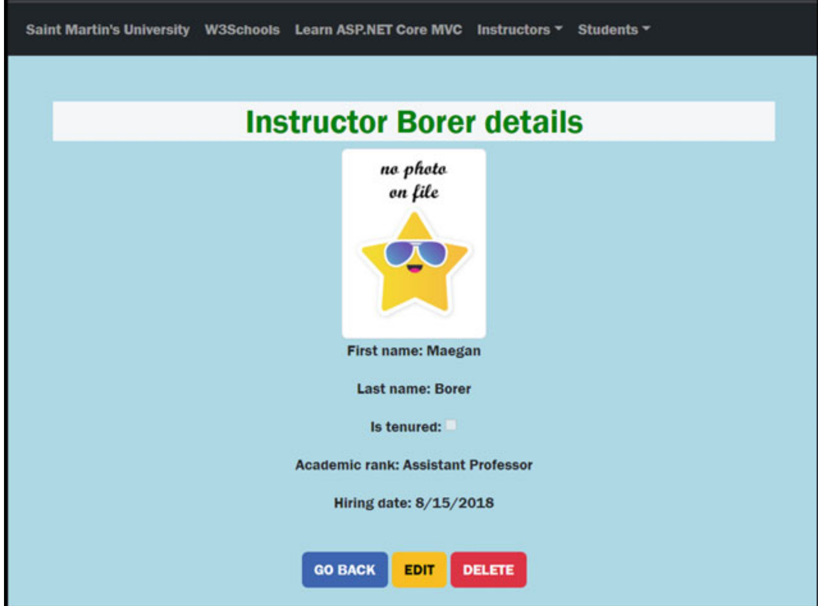


Fig. 13.4 Shows how the ShowDetails view shows up in a browser. In particular, this Instructor did not contain a profile picture, so we displayed a generic image

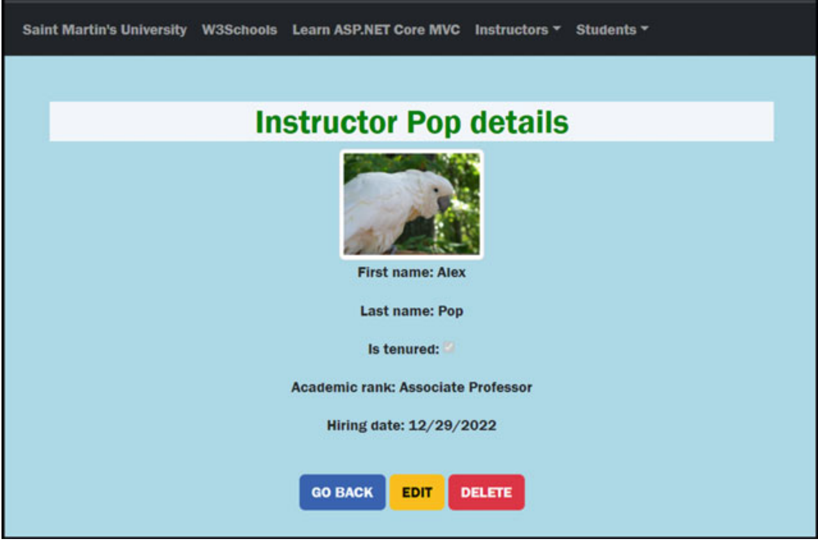


Fig. 13.5 Shows how the ShowDetails view shows up in a browser. In particular, this Instructor did contain a profile picture, so we displayed that picture

Using a combination of the sections presented in this chapter, you should be able to allow the user to edit an existing Instructor and change/keep any existing profile picture.

13.6 Bootstrap Card Deck for the Index Action and View (Optional)

This section is just meant to stimulate your interest in researching more on your own—Bootstrap is really awesome, you just need to look it up. In particular, you may want to consider displaying a **card deck** instead of a **table**, or in addition to a table. For this, check out the following resource [74].

In our `Index` action, let's add the following code that will create a *dictionary* of images and send them to the `Index` *view*. Add the code below right before the `return` statement of `Index` action (from the `InstructorController` class).

```
Dictionary<int, string> allPhotos = new();
foreach (var ins in instructors)
{


    if (ins != null && ins.InstructorProfilePhoto != null)
    {
        string imageBase64Data = Convert.ToBase64String(ins.InstructorProfilePhoto);
        string imageDataURL = string.Format($"data:image/jpg;base64,{imageBase64Data}");
        allPhotos.Add(ins.InstructorId, imageDataURL);
    }
}
ViewBag.AllImages = allPhotos;
```

Then, inside the view, we used that dictionary object inside a card deck and displayed them (you can add this right below the `<TABLE>` element):

```
@*card deck *@
<div class="container">
    <div class="row">
        @foreach (var ins in Model)
        {
            <div class="col" style="text-align:center">
                <div class="card" style="width:300px">
                    @{
                        string? str = null;
                        bool result = @ViewBag.AllImages.TryGetValue(ins.InstructorId, out str);
                    }

                    @if (@str != null)  @*if we have an image on file*@
                    {
                        <img class="card-img-top" src="@str" alt="Card image" style="width:100%">
                    }
                    else
                    {                    @*if no image in the database, display some default image*@
                        <img class="card-img-top" src="~/images/noProfilePhoto.png" alt="Card image" style="width:100%">
                    }
                    <div class="card-body">
                        <h4 class="card-title">@ins.FirstName @ins.LastName</h4>
                        <br class="card-text">
                        <label asp-for="@Model.First().HiringDate"></label>:  @Html.DisplayFor(m => ins.Rank)<br>
                        <label asp-for="@Model.First().IsTenured"></label>: @Html.DisplayFor(m => ins.IsTenured)<br>
                        <label asp-for="@Model.First().Rank"></label>: @Html.DisplayFor(m => ins.Rank)<br>

                        <a asp-action="Edit" asp-route-id="@ins.InstructorId" class="btn btn-warning">Edit Profile</a>
                        <a asp-action="Delete" asp-route-id="@ins.InstructorId" class="btn btn-danger">Delete profile</a>
                    </div>
                </div>
            </div>
        }
    </div>
</div>
```

We obtained (see Fig. 13.6).

You should consider making the button at the bottom of the page be centered.

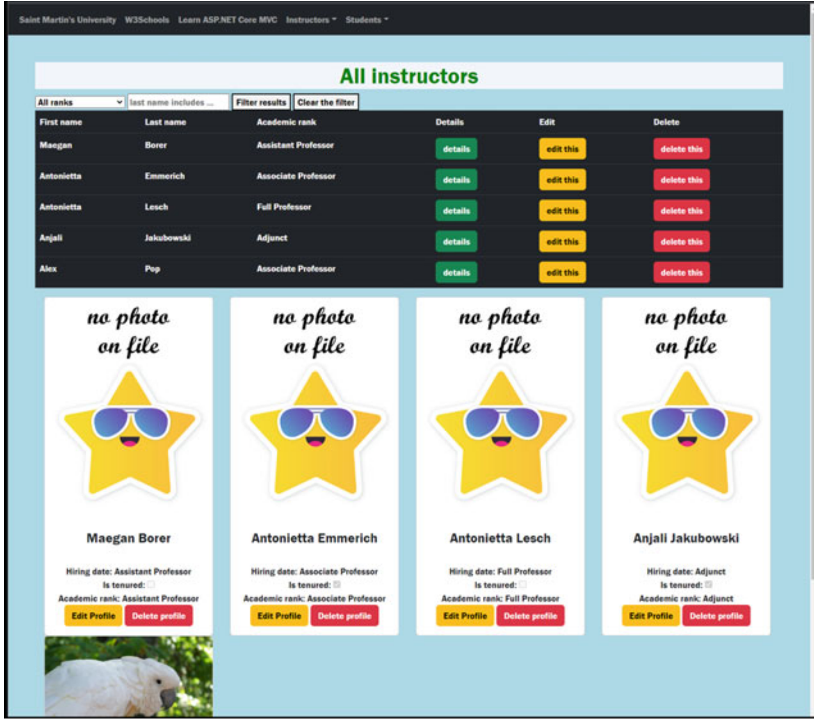


Fig. 13.6 Shows how the Index view shows up in a browser. In particular, below the table of instructors, we also included a card deck, one card for each instructor from the table. Some of those cards contained the instructor's profile pictures (if one was saved into an instructor's profile) while other cards contain the generic profile picture (if we did not have a picture saved for an instructor's profile)

Check for updates

14 Introduction to Authentication. User Login, Logout, and Registration

We finally got to our final chapter. In here we will only introduce *authentication* and *simple authorization*. In particular, we'll implement functionality such as user *account registration*, *login*, and *logout*. To learn more about ASP .Net Core security-related topics, we recommend [75]. In a future edition of this book, we plan to also include topics such as Roles, Policies, and how to allow users to login using credentials from other web applications, such as Facebook and LinkedIn.

14.1 Introduction to Some Security Concepts

Of the following four terms *Identification, Authentication*, *Authorization,* and *Accountability* (IAAA), we will only deal with the first three in here:

- **Identification**: Who are you? How do you identify yourself?
  - One can use usernames, ID numbers, email addresses, employee numbers, social security numbers, etc.

    In this chapter, we'll use usernames.

- **Authentication**: How do you prove you are who you say you are?

  There are multiple types of authentication techniques. In this chapter, we'll use passwords.
  - Type 1 authentication: Something you know (e.g., password, passphrase, pin number).
  - Type 2 authentication: Something you have (e.g., passport, badge, smart card, cookie on a system).

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024

R. A. Mezei, *Introduction to the Development of Web Applications Using ASP .Net (Core) MVC*, Synthesis Lectures on Computer Science,
https://doi.org/10.1007/978-3-031-30626-6_14

  - Type 3 authentication: Something you are (biometrics) (e.g., fingerprint, facial recognition).
  - Type 4 authentication: Somewhere you are (e.g., IP address).
  - Type 5 authentication: Something you do (e.g., signature, pattern unlock).

- **Authorization**: once you're authenticated, what can you do? What are you allowed to access?

  In this chapter, we'll only see *simple authorization.*

- **Accountability** (or **Auditing**)
  - This allows us to trace an action to a user's identity.

Read more on the IAAA concepts in [76]. In this chapter, we will only make use of the first three.

14.2 Introduction to ASP .Net (Core) Identity

In this chapter, we will make use of the **ASP.NET Core Identity** to handle login functionality. In particular, we'll use **Identity** to provide functionality such as *register* new user accounts, *login,* and *logout*. Let's go over the steps needed to configure and use **Identity**, namely the following:

- Install the NuGet Package(s) needed for Identity.
- Update the DbContext class so it works with Identity.
- Register the Identity service.
- Configure the Authentification and Authorization middleware.
- Implement actions for the following:
  - Register new users
  - Login
  - Logout.
- Then, we'll see how we can require authentication for certain actions.

Before we continue, you should uncomment the following line from Program.cs, since our changes will modify our database (new tables will be created):

```
context.Database.EnsureDeleted();
```

At the end of the chapter, please make sure to comment out this line so changes remain in our database even after we restart our web application. For more information on how to implement **Identity** on existing ASP .Net (Core) applications, check out [77].

14.2.1 Step 1: Install NuGet Packages

For this part, we will install the following two NuGet packages:

- `Microsoft.AspNetCore.Identity.EntityFrameworkCore`
- `Microsoft.AspNetCore.Identity.UI`.

14.2.2 Step 2: Define Our User Class (Derived from IdentityUser)

For this step, we need a class that can be used to manage the users of our application. For this, ASP .Net Core has a class called `IdentityUser`. If its properties (which include an `Id`, `UserName`, `Email`, and `PasswordHash`) are sufficient for your application's needs, then you are all set and can skip this step. Here are some of its properties (as seen from Visual Studio):

- public virtual TKey Id { get; set;}
- public virtual string UserName { get; set;}
- public virtual string NormalizedUserName { get; set;}
- public virtual string Email { get; set;}
- public virtual string NormalizedEmail { get; set;}
- public virtual bool EmailConfirmed { get; set;}
- public virtual string PasswordHash { get; set;}
- public virtual string SecurityStamp { get; set;}
- public virtual string ConcurrencyStamp { get; set;} = Guid.NewGuid().ToString();
- public virtual string PhoneNumber { get; set;}
- public virtual bool PhoneNumberConfirmed { get; set;}
- public virtual bool TwoFactorEnabled { get; set;}
- public virtual DateTimeOffset? LockoutEnd { get; set;}
- public virtual bool LockoutEnabled { get; set;}
- public virtual int AccessFailedCount { get; set;}

If you need to add more properties, you can create your own class derived from IdentityUser. For us, let's say we do need to also store a FirstName and a LastName (you can easily add more). Here is what we'll use in our project. First, inside the *Data* folder, let's create a new class, derived from IdentityUser, and give it a name (we called ours User).

Your code should look like

```
using Microsoft.AspNetCore.Identity;

namespace ASPBookProject.Data
{
    public class User : IdentityUser
    {
        //the base class already contains
        // id, username, email.
        public string? FirstName { get; set; }
        public string? LastName { get; set; }
    }
}
```

You should note that in our database we do not store plaintext passwords (we did not include any Password property). We instead store hash values of the passwords (in PasswordHash property). Why is that?

14.2.3 Step 3: Update Our DbContext Derived Class to Use Identity

We need to change our DbContext derived class to inherit from IdentityDbContext. This last class has built-in logic to manage users. Notice that IdentityDbContext is a generic class, and it needs to know the type of users it will work with. You could use IdentityUser if you skipped Step 2 or use the class you defined in Step 2 (above).

For us, we will replace

```
public class OurDbContext : DbContext
```

with

```
public class OurDbContext : IdentityDbContext<User>
```

IMPORTANT: Notice that we did not need a DbSet property for our User/IdentityUser class. That's because the IdentityDbContext class will manage user tables for us.

14.2.4 Step 4: Register the Identity Services

In *Program.cs* after the code:

```
builder.Services.AddDbContext<OurDbContext>(
    options => options.UseSqlite(builder.Configuration.GetConnectionString("BookConnectionString"))
    );
```

Add

```
builder.Services.AddDefaultIdentity<User>(//user your IdentityUser class in <>
    options =>
    {
        options.SignIn.RequireConfirmedAccount = false;

        options.Password.RequireDigit = true;
        options.Password.RequireNonAlphanumeric = true;
        options.Password.RequireUppercase = true;
        options.Password.RequireLowercase = true;
        options.Password.RequiredLength = 8;

        options.User.RequireUniqueEmail = true;
    }
    ).AddEntityFrameworkStores<OurDbContext>(); //use your DbContextClass
```

Above, note that we were able to set various options, so that passwords must include digits, lowercase, and uppercase letters, and also ensure that user emails are district (unique). Check out the following source to learn about other options (including lockout options and options related to characters allowed for usernames) available: [78].

14.2.5 Step 5: Add Authentification and Authorization Middleware Components

In *Program.cs*, right after `app.UseStaticFiles()`, add

```
app.UseAuthentication();
```

We add this after the `UseStaticFiles` middleware so that we continue to allow public access to `wwwroot` folder even to users who are not logged in.

Then optionally (this is only needed for Step 7: authorization!), after `app.UseRouting()` and after `app.UseAuthentication()`, also add

```
app.UseAuthentication();
```

Here is how we put them inside the *Program.cs*:

```
var context = app.Services.CreateScope().ServiceProvider.GetRequiredService<OurDbContext>();
context.Database.EnsureDeleted(); //if our database exists, then erase it!
context.Database.EnsureCreated(); //if our database does not exist, then create it!

app.UseStaticFiles(); //needed to give access to files in wwwroot
app.UseAuthentication();
app.UseRouting();  //adds route matching to the middleware pipeline
app.UseAuthorization();
app.MapControllerRoute( //modified default routing
    name: "default",
    pattern: "{controller=Instructor}/{action=Index}/{id?}");
```

Later in this chapter, we'll enforce access to certain actions (such as edit and delete) only to users who are logged in.

14.2.6 Test Your Work

Let's rebuild our application to see what we have accomplished so far. If you reopen your database file in DB Browser, you should note new tables being added to it by the ASP .Net Core Identity. In particular, note the *AspNetUsers* table.

In the `AspNetUsers` table, you should observe the various columns (they include columns for the properties of the `User` class). This table is now empty, but we'll fix that in the next section where we'll add functionality to register new users, login, and logout.

14.2.7 Step 6: Register, Login, and Logout

In this subsection, we'll create a new *controller*, the `AccountController` class, and see how to use instances of the `UserManager` class to *register* new users, and the `SignInManager` class to *login/logout* an existing user. We will also introduce *View Models*.

14.2.7.1 Add the AccountController Class

Before we continue, let's add a new controller to our project, named `AccountController`. When creating it, choose the *MVC Controller—Empty* option. Below we'll add actions to this class. We'll add the `Register` (GET and POST), `Login`, and `Logout` actions.

14.2.7.2 Add the Login and Logout Actions

In here we would like to add the *login* and *logout* functionality for existing users of our web application.

Inject the SignInManager Instance

Let's start by injecting an instance of the `SignInManager` class into our controller and adding a new action, `Login`. Our *AccountController.cs* file contains the following:

```
using ASPBookProject.Data;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;

namespace ASPBookProject.Controllers
{
    public class AccountController : Controller
    {
        private readonly SignInManager<User> _signInManager; //needed to login/logout accounts
        public AccountController(SignInManager<User> signInManager)
        {
            _signInManager = signInManager;
        }

        public IActionResult Login()
        {
            return View(); //presents the Login form to the user
        }
    }
}
```

Note: The `SignInManager` is a generic class, and you need to pass to it your `User` class (derived from `IdentityUser`), or the `IdentityUser` (if you did not use a derived class). We'll use this to login/logout our users.

Create the Login View Model

Next, we will need a form to collect information from our clients, information that will only be used for login purposes. In particular, we will need to ask for a username, a password, and optionally, if they want to make use of the remember me functionality (a cookie will be stored in their browser, so they won't need to login again if they reopen the page in a certain period of time).

Since the login information is only used for user authentication during login, essentially only used with the login view (we will not store passwords into our database, we will hash them and store their hash into the database), a *model/entity* isn't quite appropriate in here. This is where we'll use a *view model* instead. Think of a **view model** as a **model**, but we only use it with views, and we won't save its data (not directly) into our database.

In the root of our project, let's create a folder, called *ViewModels*.

Then, inside the *ViewModels* folder, let's create a class called `LoginViewModel`. In this class, we need three properties (we'll ask the user for three pieces of information): a username, a password, and if they want to be remembered. We'll also add Data Annotations since we need to make usernames and passwords required information:

```
using System.ComponentModel.DataAnnotations;

namespace ASPBookProject.ViewModels
{
    public class LoginViewModel
    {
        [Display(Name = "User Name")]
        [Required(ErrorMessage = "a username is required")]
        public string? UserName { get; set; }
        [DataType(DataType.Password)]
        [Required(ErrorMessage = "a password is required")]
        public string? Password { get; set; }

        [Display(Name = "Remember me?")]
        public bool RememberMe{ get; set; }
    }
}
```

Make sure to add the following using directive in the *_ViewImports.cshtml* file so we can use our View Model classes in views without having to specify their class names with the namespace prepended:

```
@using ASPBookProject.ViewModels
```

Create the Login View/Form

Now we are ready to create the form for user login. First, add a view (use *Razor View - Empty*) to the `Login` action. Then, add a form to this *Login.cshtml* view. We'll make the view strongly typed.

```
@model LoginViewModel

@{
    ViewBag.Title = "Login";
}

<h1>Please login</h1>
<form asp-action="Login" asp-controller="Account">
    <div asp-validation-summary="All"></div>
    <label asp-for="UserName"></label> <input asp-for="UserName"/>
    <br>
    <label asp-for="Password"></label> <input asp-for="Password" />
    <br>
    <label asp-for="RememberMe"></label> <input asp-for="RememberMe" />
    <br>
    <br>
    <input type="submit" value="LOGIN"/>
</form>
```

This form will allow users to enter their credentials. When they click on the `LOGIN` button, a POST request will be sent to the `Login` action. Let's create that action.

Create the Login (POST) Action

In this action, we'll get the information provided by the user and attempt to log them in (using the `SignInManager` instance). If we succeed, we'll redirect the user to the `Index` action of the `Instructor` controller. Otherwise, we will stay in the `Login` view and display error information. Add the following action to the *AccountController.cs*:

```
[HttpPost]
public async Task<IActionResult> Login(LoginViewModel loginInfo)
{
    if (ModelState.IsValid)
    {
        //try to log in the user
        var result = await _signInManager.PasswordSignInAsync(loginInfo.UserName,
            loginInfo.Password, loginInfo.RememberMe, false);

        if (result.Succeeded)
        {
            return RedirectToAction("Index", "Instructor");
        }
        else
        {
            ModelState.AddModelError("", "Failed to login");
        }
    }
    return View(loginInfo);//go back to login form ...
}
```

Note: Since the `PasswordSignInAsync` is an asynchronous method, we had to make the entire action asynchronous.

To find more information about the parameters used in `PasswordSignInAsync` method, inside Visual Studio, hover your mouse over the method and you'll find more information provided by Microsoft *IntelliSense*.

Create the Logout Action

This is a rather short action. We once again make use of the reference to `SignInManager`. To perform a logout, we call the `SignOutAsync` method, then redirect the user back to the `Index` action of `Instructor` controller.

```
public async Task<IActionResult> Logout()
{
    await _signInManager.SignOutAsync();
    return RedirectToAction("Index", "Instructor");
}
```

Add Login/Logout Links to the Navbar in the Layout File

In the navbar from our *_Layout.cshtml* file we would like to add a `Login` menu option if the user is not logged in, and a `Logout` menu option if the user is logged in. To accomplish this, add the following code, right before the following lines:

```
        </UL>
    </DIV>
</NAV>
```

Add

```
@if(User.Identity.IsAuthenticated) //if the user is logged in
{
    <LI class="nav-item">
        <a class="nav-link" asp-action="Logout" asp-controller="Account">Logout</a>
    </LI>
}
else //if the user is not logged in
{
    <LI class="nav-item">
        <a class="nav-link" asp-action="Login" asp-controller="Account">Login</a>
    </LI>
}
```

We'll test this later, but for now make sure your code compiles without any errors.

14.2.7.3 Add the Register Action

Next, we would like to add functionality for registering new users for our web application.

Inject the `UserManager` Instance

Let's start by injecting an instance of the `UserManager` class into our controller and adding a new action, `Register`. Our *AccountController.cs* file now includes the following code:

```
public class AccountController : Controller
{
    private readonly SignInManager<User> _signInManager; //needed to login/logout accounts
    private readonly UserManager<User> _userManager; //needed to create new user accounts
    public AccountController(SignInManager<User> signInManager, UserManager<User> userManager)
    {
        _userManager = userManager;
        _signInManager = signInManager;
    }

    public IActionResult Register()
    {
        return View(); //presents the Register form to the user
    }
//...
```

Note: The `UserManager` is a generic class, and you need to pass to it your `User` class (derived from `IdentityUser`), or the `IdentityUser` (if you did not use a derived class).

Note: Above we have two services injected. Their order is not important. They can be injected in any order.

Create the Register View Model

Next, we will need a form to collect information from our clients, information that will be used to create new users. In particular, we will need to ask for a username, email address, a password, and so on. Since we will not store passwords into our database (we will hash them and store their hash into the database), we will once again use a *view model*. Inside the *ViewModels* folder, create a new class, let's call it `RegisterViewModel`.

In this class include properties for all data you want to ask from your users in order to create a new account. In particular, we need them to provide a first name, last name, email address, a username, a password, a phone number, and for validation purposes, let's ask them to provide the password twice, to make sure they know what they entered.

Here is an example of this class:

```
using System.ComponentModel.DataAnnotations;

namespace ASPBookProject.ViewModels
{
    public class RegisterViewModel
    {
        [Display(Name = "User Name")]
        [Required(ErrorMessage = "a username is required")]
        public string? UserName { get; set; }

        [Required(ErrorMessage = "a password is required")]
        [DataType(DataType.Password)]
        public string? Password { get; set; }

        [Display(Name = "Confirm Password")]
        [Required(ErrorMessage = "you must confirm your password")]
        [DataType(DataType.Password)]
        public string? ConfirmPassword { get; set; }
        [Display(Name = "First name")]
        public string? FirstName { get; set; }

        [Display(Name = "Last name")]
        public string? LastName { get; set; }


        [Display(Name = "Email address")]
        [DataType(DataType.EmailAddress)]
        [Required(ErrorMessage = "email address required!")]
        public string? Email { get; set; }

        [RegularExpression("[0-9]{3}-[0-9]{3}-[0-9]{4}", ErrorMessage = "you must follow the
format 000-000-0000!")]
        [Display(Name = "Phone number")]
        public string? Phone { get; set; }
    }
}
```

Create the Register View

Now we are ready to create the form for `Register` view. First, add a view (use *Razor View - Empty*) to the `Register` action. Then, add a form to this *Register.cshtml* view. We'll make the view strongly typed.

```
@model RegisterViewModel

@{
    ViewBag.Title = "Register a new account";
}

<h1>Register a new account</h1>
<form asp-action="Register">
    <div asp-validation-summary="All"></div>
    <label asp-for="UserName"></label>: <input asp-for="UserName" /><br />
    <label asp-for="Password"></label>: <input asp-for="Password" /><br />
    <label asp-for="ConfirmPassword"></label>: <input asp-for="ConfirmPassword" /><br />
    <label asp-for ="FirstName"></label>: <input asp-for="FirstName" /><br />
    <label asp-for="LastName"></label>: <input asp-for="LastName" /><br />
    <label asp-for="Email"></label>: <input asp-for="Email" /><br />
    <label asp-for="Phone"></label>: <input asp-for="Phone" /><br />

    <input type="submit" value="REGISTER ACCOUNT"/>
</form>
```

This form will allow users to enter the data needed to create/register a new account. When they click on the `REGISTER ACCOUNT` button, a POST request will be sent to the `Register` action. Let's create that action.

Create the Register (POST) Action

In this action, we'll get the information provided by the user and attempt to register a new account (using the `UserManager` instance). If we succeed, we'll redirect the user to the `Index` action of the `Instructor` controller. Otherwise, we will stay in the `Register` view and display error information.

In particular, note that we are creating an instance of our `IdentityUser` derived class, the `User` class, and pass that instance to the `CreateAsync` method. As a separate argument, we're passing the password (which will be hashed before it makes its way to the database).

Add the following action to the *AccountController.cs*:

```
[HttpPost]
public async Task<IActionResult> Register(RegisterViewModel userEnteredData)
{
    if (ModelState.IsValid)
    {
        //we do not pass the password or the passwordconfirmed!
        User newUser = new User();
        newUser.UserName = userEnteredData.UserName;
        newUser.FirstName = userEnteredData.FirstName;
        newUser.LastName = userEnteredData.LastName;
        newUser.Email = userEnteredData.Email;
        newUser.PhoneNumber = userEnteredData.Phone;

        //attempt to register the new account
        var result = await _userManager.CreateAsync(newUser, userEnteredData.Password);

        if (result.Succeeded)
            return RedirectToAction("Index", "Instructor");
        else
        {
            foreach (var error in result.Errors)
                ModelState.AddModelError("", error.Description);
        }

    }
    //go back to the view
    return View(userEnteredData);
}
```

Note: Since the `CreateAsync` is an asynchronous method, we had to make the entire action asynchronous.

To find more information about the parameters used in `CreateAsync` method, in Visual Studio, hover your mouse over this method and you'll find more information provided by *IntelliSense*.

Add Register Link to the Navbar in the Layout File

In the navbar from our *_Layout.cshtml* file we would like to add a `Login` menu option if the user is not logged in, and a `Logout` menu option if the user is logged in. To accomplish this, add the following code, right before the last `</UL>` tag:

```
<LI class="nav-item">
    <a class="nav-link" asp-action="Register" asp-controller="Account">Register new account</a>
</LI>
```

Add Custom Validation to Check the Confirm Password (Optional)

Let's add a custom validation attribute to make sure the two passwords are identical. For this, add a new class to the *CustomValidations* folder, let's name it `ConfirmPasswordValidationAttribute`. In this file, include the code below (feel free to improve this code!) that will check if the properties `Password` and `ConfirmPassword` from `RegisterViewModel` have the same values:

```
using ASPBookProject.ViewModels;
using System.ComponentModel.DataAnnotations;

namespace ASPBookProject.CustomValidations
{
    public class ConfirmPasswordValidationAttribute : ValidationAttribute
    {
        protected override ValidationResult? IsValid(object? value, ValidationContext validationContext)
        {
            RegisterViewModel instanceRegisterViewModel = (RegisterViewModel)validationContext.ObjectInstance;

            if (instanceRegisterViewModel.Password != instanceRegisterViewModel.ConfirmPassword)
                return new ValidationResult("Password and Confirm Password fields must match!");

            return ValidationResult.Success; // all other cases are valid
        }
    }
}
```

Now, let's apply this to the `ConfirmPassword` property in *RegisterViewModel.cs*:

```
[ConfirmPasswordValidation]
[Display(Name = "Confirm Password")]
[Required(ErrorMessage = "you must confirm your password")]
[DataType(DataType.Password)]
public string? ConfirmPassword { get; set; }
```

14.2.7.4 Test Our Work

You can now comment out the line (in *Program.cs*) so data changes remain saved even after the web application restarts:

```
context.Database.EnsureDeleted(); //if our database exists, then erase it!
```

Let's test our work. When we run our application, we see the newly added *Login* and *Register new account* menu options (Fig. 14.1):

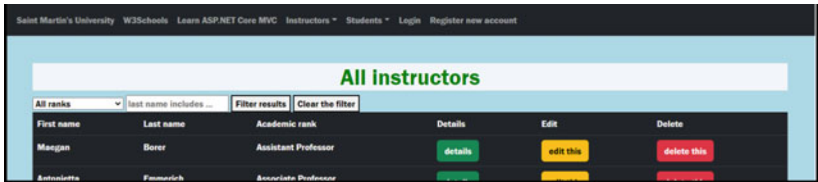


Fig. 14.1 Shows the Index view. Note that the navbar displayed at the top of the page contains the newly added menu options Login and Register new account

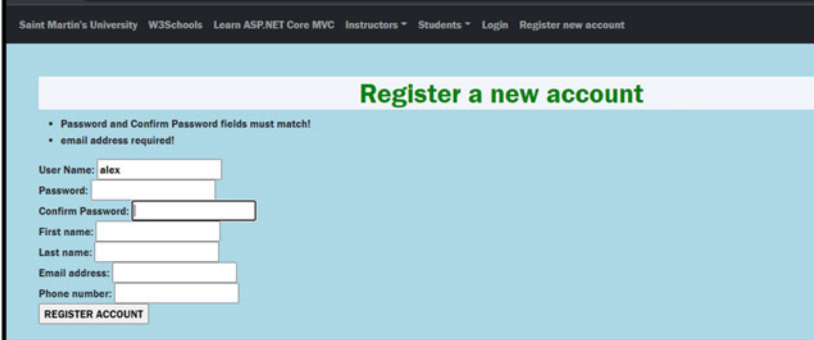


Fig. 14.2 Shows the Register view. Note that the page displays validation errors for the email address missing (this is a required field) and for non-matching passwords

Click on the *Register new account*:

Try to create an account but put two different passwords in the Password and Confirm Password fields. You should see the errors displayed at the top of the form (see Fig. 14.2):

Change your input so there are no errors and create an account. Here is what we entered (see Fig. 14.3):

Then, using DbBrower, reopen your database and check that this data has been added to the database—see table `AspNetUsers`.

Now, try to create another user but use the same email address as the one used above. You should get an error (see Fig. 14.4):

Now try entering a password of “123”, you should get the following errors (see Fig. 14.5):

Do you remember where we set these constraints? Hint: Check out the *Program.cs* file.

Lastly, let’s try to login. First, enter a wrong password (see Fig. 14.6):

Try again, this time enter the correct password. After you successfully login, you should see (Fig. 14.7) the *Logout* menu option instead of the *Login*:

Fig. 14.3 Shows the Register view with data entered in the fields so that no validation errors are produced

User Name: sebastian
Password: ••••••••••••
Confirm Password: ••••••••••••
First name: Sebastian
Last name: Pop
Email address: sebastian@stmartin.edu
Phone number: 360-688-2748
REGISTER ACCOUNT

Fig. 14.4 Shows the validation error displayed in the Register view when the user enters an email address that already exists in the database

- Email 'sebastian@stmartin.edu' is already taken.

User Name: alice
Password:
Confirm Password:
First name: Alice
Last name: Wonderland
Email address: sebastian@stmartin.edu
Phone number:
REGISTER ACCOUNT

14.2.8 Step 7: Add Simple Authorization to Our Web Application (Optional)

We got to the last section of this book. In here we'll make some changes to our code so only users who are logged in can access certain actions (add, edit, and delete). All users should still be able to view all data.

For this step, make sure both of the following middleware components were added to *Program.cs*:

```
app.UseAuthentication();
//…
app.UseAuthorization();
```

- Passwords must be at least 8 characters.
- Passwords must have at least one non alphanumeric character.
- Passwords must have at least one lowercase ('a'-'z').
- Passwords must have at least one uppercase ('A'-'Z').

User Name: alice
Password:
Confirm Password:
First name: Alice
Last name: Wonderland
Email address: sebastian@stmartin.edu
Phone number:
REGISTER ACCOUNT

Fig. 14.5 Shows the validation error displayed in the Register view when the user enters a password that does not follow the server side specifications (for example, it does not include at least 8 characters, or it does not have at least one lowercase and at least one upper case letter)

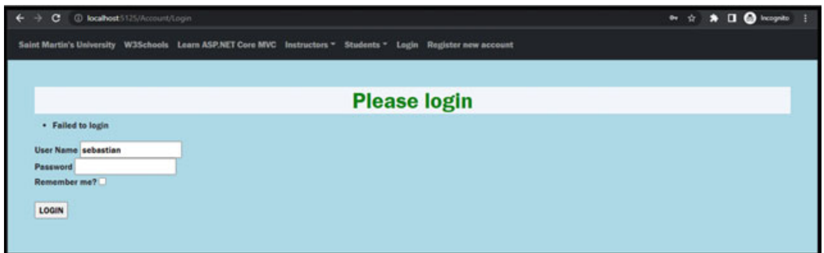


Fig. 14.6 Shows the validation error displayed in the Login view when the user has invalid credentials

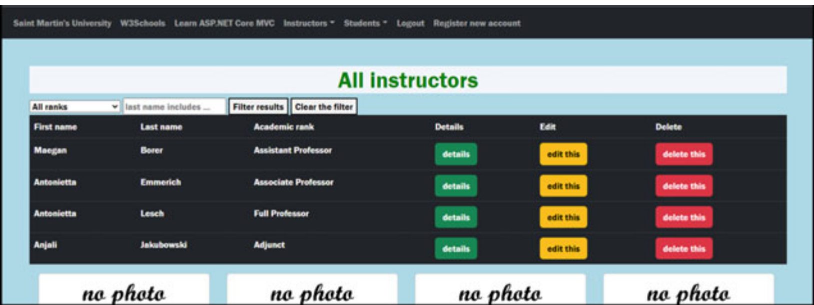


Fig. 14.7 Shows the Index view displayed after the user logs into their account. In particular, note the Logout menu option in the top navbar (which indicates that the user has been successfully authenticated)

Then, in *Program.cs*, change the line:

```
builder.Services.AddDefaultIdentity<User>(//user your IdentityUser class in <>
```

with

```
builder.Services.AddIdentity<User, IdentityRole>(//user your IdentityUser class in <>
```

(You'll also need the following `using` directive: `using Microsoft.AspNetCore.Identity;`
Then, make use of the following attributes:

- `[Authorize]`—can be applied to actions or controller classes; it prevents unauthorized users from accessing this resource while not logged in.
- `[AllowAnonymous]`—makes the resource publicly available.

In the `InstructorController` class, add the `[Authorize]` attribute right before the `Add`, `Edit`, and `Delete` actions. For example,

```
[Authorize]
[HttpPost]
public IActionResult Add(Instructor newInstructor)
{
    if (!ModelState.IsValid) //if the data is invalid
        return View(); //go back to the view

    //if a file/image was uploaded, convert it to byte[] and save it
    if (Request.Form.Files.Count > 0) //did the user upload a file?
    {
        var file = Request.Form.Files[0];//our view only allows one file

        MemoryStream ms = new MemoryStream();
        file.CopyTo(ms); //copy the file into a memory stream object
        newInstructor.InstructorProfilePhoto = ms.ToArray();//save the bytes into newInstructor

        ms.Close();
        ms.Dispose();
    }

    _dbContext.Instructors.Add(newInstructor); //add the new instructor to our list
    _dbContext.SaveChanges();
    return RedirectToAction("Index");
}
```

Now rebuild your application, and if a user is already logged in, please logout. You should be able to see the `Index` action of `InstructorController`, even if you are not logged in.

Now click on *Add a new instructor* button. You should be prompted to login (see Fig. 14.8). Before you log in, please note the URL.

Login, then try again to add a new instructor. While logged in, you should now be able to do it.

In the URL above you should note the encoded query string: `ReturnUrl=/Instructor/Add`. To make use of it, we can modify our `Login` (the GET) action to capture that value:

```
public IActionResult Login(string? returnUrl)
{
    TempData["CapturedReturnUrl"] = returnUrl;
    return View(); //presents the Login form to the user
}
```

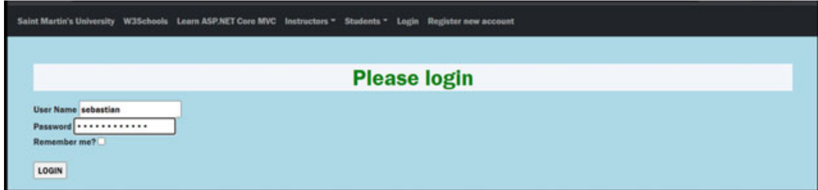


Fig. 14.8 Shows the view that prompts the user to log into their account

Then, in the `Login` (POST) action, we can make use of it:

```
[HttpPost]
public async Task<IActionResult> Login(LoginViewModel loginInfo)
{
    if (ModelState.IsValid)
    {
        //try to log in the user
        var result = await _signInManager.PasswordSignInAsync(loginInfo.UserName,
            loginInfo.Password, loginInfo.RememberMe, false);

        if (result.Succeeded)
        {
            if (!string.IsNullOrEmpty(TempData["CapturedReturnUrl"] as string))
                return Redirect(TempData["CapturedReturnUrl"] as string);
            else
                return RedirectToAction("Index", "Instructor");
        }
        else
        {
            ModelState.AddModelError("", "Failed to login");
        }
    }
    return View(loginInfo);//go back to login form ...
}
```

Now, after you are prompted to login, you will be redirected to the page that sent you to login (that needed authorization). See this for more details [79].

Note: Above we made use of temp data, which allows a controller to preserve data between requests. This is particularly useful when performing requests. Temp data marks values for deletion once they are read and then removed when the request has been processed. See more in [5].

Important note: This chapter is an overly simplistic section meant to stimulate your interest in learning more. In particular, we did not go over roles, policies, and how to allow users to log into our web application using their credentials from other web applications, such as Facebook and LinkedIn. Please look into the mentioned references to learn more about these.

References

1. W3Schools, "W3Schools Online Web Tutorials," [Online]. Available: https://www.w3schools.com/. [Accessed 4th December 2022].
2. Microsoft, "Get started with ASP.NET Core MVC," [Online]. Available: https://learn.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/start-mvc. [Accessed 4th December 2022].
3. M. M. Girgis and L. Petry, "An Effective Quiz Strategy for Enhancing Student Engagement while Discouraging Academic Dishonesty. Girgis," [Online]. Available: http://people.se.cmich.edu/yelam1k/asee/proceedings/2018/1/52.pdf. [Accessed 4th December 2022].
4. Microsoft, "Get started with ASP.NET Core MVC [Visual Studio Code]," [Online]. Available: https://learn.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/start-mvc?view=aspnetcore-6.0&tabs=visual-studio-code. [Accessed 4th December 2022].
5. A. Freeman, "Pro ASP.NET Core 6: Develop Cloud-Ready Web Applications Using MVC, Blazor, and Razor Pages, 9th ed. Edition," Apress.
6. W3Schools, "HTML Tutorial," [Online]. Available: https://www.w3schools.com/html/. [Accessed 4th December 2022].
7. W3Schools, "HTML <title> Tag," [Online]. Available: https://www.w3schools.com/tags/tag_title.asp. [Accessed 4th December 2022].
8. Mozilla, "How whitespace is handled by HTML, CSS, and in the DOM," [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Whitespace. [Accessed 4th December 2022].
9. W3Schools, "HTML <label> Label," [Online]. Available: https://www.w3schools.com/tags/tag_label.asp. [Accessed 4 December 2022].
10. W3Schools, "HTML <select> Tag," 4th December 2022. [Online]. Available: https://www.w3schools.com/tags/tag_select.asp.
11. W3Schools, "HTML Forms," [Online]. Available: https://www.w3schools.com/html/html_forms.asp. [Accessed 4 December 2022].
12. W3Schools, "HTML Input Types," [Online]. Available: https://www.w3schools.com/html/html_form_input_types.asp. [Accessed 4 December 2022].
13. W3Schools, "HTTP Request Methods," [Online]. Available: https://www.w3schools.com/tags/ref_httpmethods.asp. [Accessed 4 December 2022].
14. W3Schools, "CSS Tutorial," [Online]. Available: https://www.w3schools.com/css/default.asp. [Accessed 4th December 2022].

© The Editor(s) (if applicable) and The Author(s), under exclusive license to Springer Nature Switzerland AG 2024

R. A. Mezei, *Introduction to the Development of Web Applications Using ASP .Net (Core) MVC*, Synthesis Lectures on Computer Science,
https://doi.org/10.1007/978-3-031-30626-6

15. W3Schools, "CSS Fonts," [Online]. Available: https://www.w3schools.com/css/css_font.asp. [Accessed 4th December 2022].
16. W3Schools, "Bootstrap 5 Tutorial," [Online]. Available: https://www.w3schools.com/bootstrap5/. [Accessed 4 December 2022].
17. W3Schools, "Bootstrap 5 Buttons," [Online]. Available: https://www.w3schools.com/bootstrap5/bootstrap_buttons.php. [Accessed 4 December 2022].
18. W3Schools, "JavaScript Tutorial," [Online]. Available: https://www.w3schools.com/js/default.asp. [Accessed 4th December 2022].
19. W3Schools, "JavaScript HTML DOM," [Online]. Available: https://www.w3schools.com/js/js_htmldom.asp. [Accessed 4th December 2022].
20. W3Schools, "JavaScript HTML DOM Elements," [Online]. Available: https://www.w3schools.com/js/js_htmldom_elements.asp. [Accessed 4th December 2022].
21. W3Schools, "How TO - Toggle Dark Mode," [Online]. Available: https://www.w3schools.com/howto/howto_js_toggle_dark_mode.asp. [Accessed 24 December 2022].
22. Mozilla, "History.back()," [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/History/back. [Accessed 4 December 2022].
23. W3Schools, "Bootstrap 5 Tutorial," [Online]. Available: https://www.w3schools.com/bootstrap5/. [Accessed 4 December 2022].
24. Bootstrap, "Introduction - Bootstrap v5.0," [Online]. Available: https://getbootstrap.com/docs/5.0/getting-started/introduction/. [Accessed 4 December 2022].
25. W3Schools, "Bootstrap 5 Navbars," [Online]. Available: https://www.w3schools.com/bootstrap5/bootstrap_navbar.php. [Accessed 4 December 2022].
26. Start Bootstrap, "Bootstrap Templates & Themes," [Online]. Available: https://startbootstrap.com/themes?showVue=false&showAngular=false&showPro=false. [Accessed 24 December 2022].
27. Microsoft, "Top-level statements - programs without Main methods," [Online]. Available: https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/program-structure/top-level-statements. [Accessed 4 December 2022].
28. Microsoft, "Declare namespaces to organize types," [Online]. Available: https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/types/namespaces. [Accessed 4 December 2022].
29. Microsoft, "C# console app template generates top-level statements," [Online]. Available: https://learn.microsoft.com/en-us/dotnet/core/tutorials/top-level-templates. [Accessed 4 December 2022].
30. Microsoft, "Recommended XML tags for C# documentation comments," [Online]. Available: https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/xmldoc/recommended-tags. [Accessed 4 December 2022].
31. W3Schools, "C# Data Types," [Online]. Available: https://www.w3schools.com/cs/cs_data_types.php. [Accessed 4 December 2022].
32. Microsoft, "Access Modifiers (C# Programming Guide)," [Online]. Available: https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/access-modifiers. [Accessed 4 December 2022].
33. W3Schools, "C# Access Modifiers," [Online]. Available: https://www.w3schools.com/cs/cs_access_modifiers.php. [Accessed 4 December 2022].
34. Microsoft, "Properties," [Online]. Available: https://learn.microsoft.com/en-us/dotnet/csharp/properties. [Accessed 4 December 2022].
35. W3Schools, "C# Tutorial," [Online]. Available: https://www.w3schools.com/cs/index.php. [Accessed 4 December 2022].
36. Microsoft, "C# documentation," [Online]. Available: https://learn.microsoft.com/en-us/dotnet/csharp/. [Accessed 4 December 2022].

37. Microsoft, "System.Collections Namespace," [Online]. Available: https://learn.microsoft.com/en-us/dotnet/api/system.collections?view=net-6.0. [Accessed 4 December 2022].
38. Microsoft, "System.Collections.Generic Namespace," [Online]. Available: https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic?view=net-6.0. [Accessed 4 December 2022].
39. Microsoft, "Inheritance in C# and .NET," [Online]. Available: https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/tutorials/inheritance. [Accessed 4 December 2022].
40. Microsoft, "Interfaces - define behavior for multiple types," [Online]. Available: https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/types/interfaces. [Accessed 4 December 2022].
41. Microsoft, "Lambda expressions and anonymous functions," [Online]. Available: https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/lambda-expressions. [Accessed 24 December 2022].
42. S. Giesel, "LINQ Explained with sketches," [Online]. Available: https://linkdotnetblogstorage.azureedge.net/blog/20220811_LinqWithSketchesEBook/LINQ.pdf. [Accessed 24 December 2022].
43. Microsoft, "Nullable value types (C# reference)," [Online]. Available: https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/nullable-value-types. [Accessed 4 December 2022].
44. Microsoft, "?? and ??= operators - the null-coalescing operators," [Online]. Available: https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/null-coalescing-operator. [Accessed 24 December 2022].
45. Microsoft, "Solution (.sln) file," [Online]. Available: https://learn.microsoft.com/en-us/visualstudio/extensibility/internals/solution-dot-sln-file?view=vs-2022. [Accessed 24 December 2022].
46. A. Chiarelli, ".NET 6 Highlights," [Online]. Available: https://auth0.com/blog/dotnet6-whats-new/. [Accessed 4 December 2022].
47. Microsoft, "Overview of ASP.NET Core MVC," [Online]. Available: https://learn.microsoft.com/en-us/aspnet/core/mvc/overview?view=aspnetcore-6.0. [Accessed 4 December 2022].
48. Microsoft, "ASP.NET Core Middleware," [Online]. Available: https://learn.microsoft.com/en-us/aspnet/core/fundamentals/middleware/?view=aspnetcore-6.0. [Accessed 4 December 2022].
49. Microsoft, "Static files in ASP.NET Core," [Online]. Available: https://learn.microsoft.com/en-us/aspnet/core/fundamentals/static-files?view=aspnetcore-6.0. [Accessed 4 December 2022].
50. Microsoft, "Dependency injection in ASP.NET Core," [Online]. Available: https://learn.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-6.0. [Accessed 4 December 2022].
51. Microsoft, "Dependency injection in .NET," [Online]. Available: https://learn.microsoft.com/en-us/dotnet/core/extensions/dependency-injection. [Accessed 4 December 2022].
52. Microsoft, "Handle requests with controllers in ASP.NET Core MVC," [Online]. Available: https://learn.microsoft.com/en-us/aspnet/core/mvc/controllers/actions?view=aspnetcore-6.0. [Accessed 4 December 2022].
53. S. Bageri, "Action Results In ASP.NET MVC Core," Tutexchange, 31 December 2019. [Online]. Available: https://tutexchange.com/action-results-in-asp-net-mvc-core/.
54. Microsoft, "Routing to controller actions in ASP.NET Core," [Online]. Available: https://learn.microsoft.com/en-us/aspnet/core/mvc/controllers/routing?view=aspnetcore-6.0. [Accessed 4 December 2022].
55. Comment Picker, "Fake Name Generator," [Online]. Available: https://commentpicker.com/fake-name-generator.php. [Accessed 24 December 2022].
56. Microsoft, "Views in ASP.NET Core MVC," [Online]. Available: https://learn.microsoft.com/en-us/aspnet/core/mvc/views/overview?view=aspnetcore-6.0. [Accessed 4 December 2022].

57. java T point, "CRUD Operations in SQL," [Online]. Available: https://www.javatpoint.com/crud-operations-in-sql. [Accessed 4 December 2022].
58. Microsoft, "Razor syntax reference for ASP.NET Core," [Online]. Available: https://learn.microsoft.com/en-us/aspnet/core/mvc/views/razor?view=aspnetcore-6.0. [Accessed 4 December 2022].
59. Microsoft, "Tag Helpers in ASP.NET Core," [Online]. Available: https://learn.microsoft.com/en-us/aspnet/core/mvc/views/tag-helpers/intro?view=aspnetcore-6.0. [Accessed 4 December 2022].
60. Microsoft, "System.ComponentModel.DataAnnotations.Schema Namespace," [Online]. Available: https://learn.microsoft.com/en-us/dotnet/api/system.componentmodel.dataannotations.schema?view=net-6.0. [Accessed 24 December 2022].
61. Microsoft, "Model Binding in ASP.NET Core," [Online]. Available: https://learn.microsoft.com/en-us/aspnet/core/mvc/models/model-binding?view=aspnetcore-6.0. [Accessed 4 December 2022].
62. Microsoft, "Model validation in ASP.NET Core MVC and Razor Pages," [Online]. Available: https://learn.microsoft.com/en-us/aspnet/core/mvc/models/validation?view=aspnetcore-6.0. [Accessed 4 December 2022].
63. Microsoft, "Part 4, add a model to an ASP.NET Core MVC app," [Online]. Available: https://learn.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/adding-model?view=aspnetcore-6.0. [Accessed 4 December 2022].
64. Microsoft, "Part 5, work with a database in an ASP.NET Core MVC app," [Online]. Available: https://learn.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/working-with-sql?view=aspnetcore-6.0. [Accessed 4 December 2022].
65. "Learn Entity Framework Core," [Online]. Available: https://www.learnentityframeworkcore.com/. [Accessed 4 December 2022].
66. Microsoft, "Data Seeding," [Online]. Available: https://learn.microsoft.com/en-us/ef/core/modeling/data-seeding. [Accessed 4 December 2022].
67. "The Connection Strings Reference," [Online]. Available: https://www.connectionstrings.com/. [Accessed 4 December 2022].
68. Microsoft, "Part 7, add search to an ASP.NET Core MVC app," [Online]. Available: https://learn.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/search?view=aspnetcore-6.0. [Accessed 4 December 2022].
69. Microsoft, "Layout in ASP.NET Core," [Online]. Available: https://learn.microsoft.com/en-us/aspnet/core/mvc/views/layout?view=aspnetcore-6.0. [Accessed 4 December 2022].
70. Microsoft, "Use multiple environments in ASP.NET Core," [Online]. Available: https://learn.microsoft.com/en-us/aspnet/core/fundamentals/environments?view=aspnetcore-6.0. [Accessed 4 December 2022].
71. Microsoft, "Handle errors in ASP.NET Core," [Online]. Available: https://learn.microsoft.com/en-us/aspnet/core/fundamentals/error-handling?view=aspnetcore-6.0. [Accessed 4 December 2022].
72. binaryintellect.net, "Store Images In SQL Server Using EF Core And ASP.NET Core," 09 December 2019. [Online]. Available: http://www.binaryintellect.net/articles/2f55345c-1fcb-4262-89f4-c4319f95c5bd.aspx.
73. Microsoft, "Upload files in ASP.NET Core," [Online]. Available: https://learn.microsoft.com/en-us/aspnet/core/mvc/models/file-uploads?view=aspnetcore-6.0. [Accessed 4 December 2022].
74. W3Schools, "Bootstrap 5 Cards," [Online]. Available: https://www.w3schools.com/bootstrap5/bootstrap_cards.php. [Accessed 4 December 2022].

75. Microsoft, “ASP.NET Core security topics,” [Online]. Available: https://learn.microsoft.com/en-us/aspnet/core/security/?view=aspnetcore-6.0. [Accessed 4 December 2022].
76. T. Pedersen, “CISSP – IAAA (Identification and Authentication, Authorization and Accountability),” 17 August 2017. [Online]. Available: https://thorteaches.com/cissp-iaaa/.
77. Programming in CSharp, “Implement Identity On Existing ASP.NET Project,” 25 January 2022. [Online]. Available: https://programmingcsharp.com/implement-identity-on-existing-asp-project.
78. Microsoft, “Introduction to Identity on ASP.NET Core,” [Online]. Available: https://learn.microsoft.com/en-us/aspnet/core/security/authentication/identity?view=aspnetcore-6.0&tabs=visual-studio. [Accessed 4 December 2022].
79. Stack overflow, “ReturnUrl is null in ASP.NET Core login,” [Online]. Available: https://stackoverflow.com/questions/44478657/returnurl-is-null-in-asp-net-core-login. [Accessed 4 December 2022].
80. "What is .NET?,” Microsoft, [Online]. Available: https://dotnet.microsoft.com/en-us/learn/dotnet/what-is-dotnet. [Accessed 4th December 2022].
81. "careers.wa.gov - Find a job working for Washington state,” [Online]. Available: https://www.careers.wa.gov/. [Accessed 4th December 2022].
82. Microsoft, “What is ASP.NET?,” [Online]. Available: https://dotnet.microsoft.com/en-us/learn/aspnet/what-is-aspnet. [Accessed 4th December 2022].
83. Microsoft, “What’s new in .NET 5,” [Online]. Available: https://learn.microsoft.com/en-us/dotnet/core/whats-new/dotnet-5. [Accessed 4 December 2022].
84. N. Barbettini, “The Little ASP.NET Core Book,” 2018. [Online]. Available: https://s3.amazonaws.com/recaffeinate-files/LittleAspNetCoreBook.pdf.