Chapter 1

Start a project

- Create a folder for the project
- Open CMD on the folder (or terminal)
- Make sure that .NET Core SDK is installed with the dotnet –version command
  - If it is installed, move to the next step
  - If it isn't installed google .net core sdk and follow the installation procedures
- To create a new MVC project on that file, run: dotnet new mvc

Chapter 2

Write the code for the database (Code-First Approach)

2.1 Add the Entity Framework Core packages to the project

- Open Visual Studio Code on the folder of the project you just started
- Open the terminal inside VS Code
- Run dotnet add package Microsoft.VisualStudio.Web.CodeGeneration.Design
- Run dotnet add package Microsoft.EntityFrameworkCore.Tools.DotNet

2.2 Add the classes representing the tables

In the Code First approach, each of the classes that will be created in the Models folder will represent an entity in the database.

2.2.1 Add the class file

- In the file explorer, right-click on the ”Models” folder and select add file
- Add the name of the Entity to the file and end with the .cs extension

2.2.2 Add code to the class

After creating the class, it needs to be added to the Models namespace and have its attributes describing the attributes of the entity. The general structure for the class is the following:

```
using System;
using System.Collections.Generic;

namespace [ProjectName].Models
{
    public class [ClassName]
    {
        //Add your entity attributes here. Integers would be ints
        //Strings would be string
        //Date Time types would be DateTime
        public int ID {get; set;}

        //This class doesn't have any constructor

        //In order to establish a one-to-many relationship where the
        //current entity has many entries in another table, do:
        public ICollection<[OtherEntityName]> [CollectionName] {get; set
    }
}
```

2.3 Create the Database Context

The database context is the main class that coordinates the Entity Framework functionality for a given data model. We create this class by deriving it from the Microsoft.EntityFrameworkCore.DbContext class. In the code for this class we must specify which entities are included in the data model. We can also customize certain Entity Framework behaviour.

- In the project folder, create a folder named Data
- In the Data folder create a new file named [ProjectName]Context.cs and insert the following code:

```
using [ProjectName].Models;
using Microsoft.EntityFrameworkCode;

namespace [ProjectName].Data
{
    public class [ProjectName]Context : DbContext
    {
        public [ProjectName]Context(DbContextOptions<[ProjectName]Contex
        {
        }

        public DbSet<[EntityName]> [EntityName] {get; set;}
```

```
        public DbSet<[EntityName]> [EntityName] {get; set;}
        //... Continue with the DbSets for as many entities as
        //necessary for the project
    }
}
```

This code creates a DbSet property for each entity set. In Entity Framework terminology, an entity set typically corresponds to a database table, and an entity corresponds to a row in the table.

When the database is created, EF creates tables that have names the same as the DbSet property names. Property names for collections are typically plural, however, developers disagree about whether table names should be pluralized or not. If you want the name of your tables to be in the singular, you have to add the following code in DbContext to override the default behaviour of EF:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<EntityName>().ToTable("EntityName");
    //Repeat for the other entities
}
```

2.4 Register the context with dependency injection

ASP.NET Core implements dependency injection by default. Services such as the EF database context are registered with dependency injection during application startup. Components that require these services (such as MVC controllers) are provided these services via constructor parameters. To register the database context as a service, we need to:

- Open Startup.cs
- Add the following code in the ConfigureServices method:

```
services.AddDbContext<[DbContextName]>(options =>
options.UseSqlite("Data Source=[ProjectName].db"));
```

In order for the DbContext to be recognized, as well as the UseSqlServer options, we need to add the following using statements at the beginning of Startup.cs:

```
using [ProjectName].Data;
using Microsoft.EntityFrameworkCore;
```

Chapter 3

Add code to initialize the database with test data

After the previous steps, Entity Framework will create an empty database for us. In this chapter we will write a method that is called after the database is created in order to populate it with test data.

Here we will use the EnsureCreated method to automatically create the database. In order to create and populate the database we have to:

- Create a file DbInitializer.cs in the Data folder
- Insert the following code:

```
using [ProjectName].Models;
using System;
using System.Linq;

namespace [ProjectName].Data
{
    public static class DbInitializer
    {
        public static void Initialize([DbContextName] context)
        {
            context.Database.EnsureCreated();

            //Before populating, make sure there is no data already
            if(context.[EntityName].Any())
            {
                return; //DB has been seeded
            }

            var [entityName] = new [EntityName][]
```

```
                {
                    new [EntityName]{//add code here for the different attri
                    //Add more entries to the table here
                };
                foreach([EntityName] n in [varPreviouslyCreated])
                {
                    context.[EntityName]s.Add(n);
                }
                //Do the previous 2 structures (var and foreach)
                //for the remaining entities of the table

                context.SaveChanges();
            }
        }
    }
```

The previous code checks if there are any members of one of the entities in the database, and if not, it assumes that the database is new and needs to be seeded with test data. It loads test data into arrays rather than List<t> collections to optimize performance.

After doing this, we need to head to Program.cs and modify the main method to do the following on application startup:

- Get a database context instance from the dependency injection container
- Call the seed method, passing to it the context
- Dispose the context when the seed method is done

The end result should look something like this:

```
    using Microsoft.Extensions.DependencyInjection;
    using [ProjectName].Data;

    //Lines not shown for brevity

    public static void Main(string[] args)
    {
        var host = BuildWebHost(args);

        using(var scope = host.Services.CreateScope())
        {
            var services = scope.ServiceProvider;
            try
            {
                var context = services.GetRequiredService<[DbContextName]>()
                DbInitializer.Initialize(context);
            }
```

```
                catch(Exception ex)
                {
                    var logger = services.GetRequiredService<ILogger<Program>>()
                    logger.LogError(ex, "An error occurred while seeding the dat
                }
            }

            host.Run();
        }
```

Now, the first time we will run the application, the database will be created and seeded with test data. Whenever we change our data model, we can delete the database, update the seed method, and start afresh with a new database the same way.

Chapter 4

Create a controller and views

Unfortunately, as of now, ASP.NET Core does not allow for automatic scaffolding of views and controller for our database model using Visual Studio Code. Therefore we will need to hand code them, which will allow us to have a better understanding of how controllers and views are built and give us better customization options, but will, unfortunately, cost us more time.

4.1 Adding a Controller

Controllers are classes that handle browser requests. They retrieve model data and call view templates that return a response. In an MVC app, the controller handles and responds to user input and interaction. The controller handles route data and query-string values, and passes them to the model. To add a controller in Visual Studio Code, we have to:

- Right-click in the Controllers folder and select New File
- Write the name of the new controller (usually corresponds to the name of an entity) and end with the .cs extension

In a controller class, every public method in a controller is callable as an HTTP endpoint. An HTTP endpoint is a targetable URL in the web application and combines the protocol used: HTTP, the network location of the webserver and the target URI.

The first comment stated before a public method usually states the pathway to the method.

MVC invokes controller classes (and the action method within them) depending on the incoming URL. The default URL routing logic used by MVC uses a format like this to determine what code to invoke: /[Controller]/[ActionName]/[Parameters].

We can modify the code to pass parameter information from the URL to the controller. By adding parameters to the action method, we specify the names of the variables and types that are being expected to be in the parameters section of the URL. In order to pass those parameters, we need to add the right name and type of value in the URL.

4.2 Adding a view

We use Razor view template files to cleanly encapsulate the process of generating HTML responses to a client We create a view template file using Razor. Razor-based template files have a .cshtml file extension. They provide an elegant way to create HTML output using C.

If we want a controller to return a view to the browser, we have to add the following code:

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

The preceding code returns a View object. It uses a view template to generate an HTML response to the browser. Controller methods (also known as action methods) such as the Index method above, generally return an IActionResult (or a class derived from ActionResult), not a type like string.

After asking the controller to return the view, we now need to create the view template that will generate the HTML for our view. We add a view to a controller by doing:

- Add a new folder in Views with the name of the controller class (without the Controller part)
- Add a new file to the previously created folder with the name of the method and ending in .cshtml

We can then write the html mixed with UI code to generate the view.

4.3 Changing views and layout pages

If we tap the menu links, we can easily notice that each page shows the same menu layout. The menu layout is implemented in the Views/Shared/$_{L}ayout.cshtmlfile$.

Layout templates allow us to specify the HTML container layout of our side in one place and then apply it across multiple pages in our site. The RenderBody call in the Layout.cshtml file is a placeholder where all the view-specific pages we create show up, wrapped in the layout page. For example, if we select the About link, it is the Views/Home/About.cshtml view that is rendered inside the RenderBody method.

4.4 Change the title and menu link in the layout file

We can change the contents of the title element. We can change the anchor text in the layout template to the name of our app and the controller from Home to whatever name we wish.

If we want to change the title of the app, we have to go to the line of code that looks like this:

```
<title>@ViewData["Title"] - [AppName]</title>
```

Then, inside the navbar-header div, we change the main title that appears in the web page on the top left corner:

```
<a asp-area="" asp-controller="[MainControllerName]" asp-action="Index"
```

By using the $_Layout.cshtml file for our main layout, it allows us to make changes once in the whole web app, savi$
The Views/$_ViewStart.cshtml file brings in the Views/Shared/_Layout.cshtml file to each view. We can use the$

We can change the title of the Index view as well. When we open the Views/Home/Index.cshtml file, there are two places to make a title change:

- The text that appears in the title of the browser
- The secondary header (<h2> element)

To see what does what, we can make them slightly different to see which bit of code changes which part of the app.

4.5 Passing Data from the Controller to the View

Controller actions are invoked in response to an incoming URL request. A controller class is where we write code that handles the incoming browser requests. The controller retrieves data from a data source and decides what type of response to send back to the browser. View templates can be used from a controller to generate and format an HTML response to the browser.

Controllers are responsible for providing data required in order for a view template to render a response. A best practice: View templates should not perform business logic or interact with a database directly. Rather, a view template should work only with the data that's provided to it by the controller. Maintaining this "separation of concerns" helps keep our code clean, testable and maintainable.

If we have a controller receiving values in the URL as parameters, instead of rendering them as a string back to the browser, we can change the controller to use a view template instead. The view template will generate a dynamic response, which means that we need to pass the appropriate bits of data from the controller to the view in order to generate the response. We can do this by having the controller put the dynamic data (parameters) that the view template needs in a ViewData dictionary that the view template can then access.

Let's imagine a HelloWorldController file with a welcome method that receives parameters that need to be sent to the view. That method receives a name and ID parameters and then outputs the values directly to the browser. We can now change the Welcome method to add a Message and NumTimes value to the ViewData dictionary. The ViewData dictionary is a dynamic object, which means we can put whatever we want in to it; the ViewData object has no defined properties until we put something inside it. The MVC model binding system automatically maps the named parameters (name and numTimes) from the query string in the address bar to parameters in our method.

The finished HelloWorldController would look like this:

```
using Microsoft.AspNetCore.Mvc;
using System.Text.Encodings.Web;

namespace MvcMovie.Controllers
{
    public class HelloWorldController : Controller
    {
        public IActionResult Index()
        {
            return View();
        }

        public IActionResult Welcome(string name, int numTimes = 1)
        {
            ViewData["Message"] = "Hello " + name;
            ViewData["NumTimes"] = numTimes;

            return View();
        }
    }
}
```

The ViewData dictionary object contains data that will be passed to the view.

We can then create a Welcome view template named Views/HelloWorld/Welcome.cshtml We'll create a loop in the Welcome.cshtml view template that displays the Message for NumTimes. We replace the contents of Views/HelloWorld/Welcome.cshtml with the following:

```
@{
    ViewData["Title"] = "Welcome";
}

<h2>Welcome</h2>

<ul>
```

```
        @for (int i = 0; i < (int)ViewData["NumTimes"]; i++)
        {
            <li>@ViewData["Message"]</li>
        }
    </ul>
```

The data is taken from the URL and passed to the controller using the MVC model binder. The controller packages the data into a ViewData dictionary and passes that object to the view. The view renders the data as HTML to the browser.

In the sample above, we used the ViewData dictionary to pass data from the controller to a view. Later we will also use a view model to pass data from a data from a controller to a view. The view model approach of passing data is generally much preferred over the ViewData dictionary approach.

Chapter 5

Scaffolding Controllers for your data model

In order to generate controllers for your data model, you have to open the terminal and run the following command for each of your DbContexts:

- dotnet restore
- dotnet aspnet-codegenerator controller -name [ControllerName] -m [ModelName] -dc [DbContext] –relativeFolderPath Controllers –useDefaultLayout –referenceScriptLibraries

With these commands, the scaffolding engine creates the following:

- A controller for the model
- CRUD (Create, Delete, Details, Edit and Index) Razor view pages for the model

The automatic creation of CRUD action methods and views is known as scaffolding. If you ever run into problems during the scaffolding process, try deleting the obj folder from the project, and running both the commands again.

The controller takes a DbContext object as a constructor parameter. ASP.NET dependency injection will take care of passing an instance of the DbContext into the controller. We configured that in the Startup.cs file earlier.

The controller contains an Index action method, which displays all the data for that entity in the database. The method gets a list of all the data in the entity set by reading the appropriate property of the database context, by the method:

- return View(await $_{c}ontext.[EntityName].ToListAsync()$)

The view associated with that model displays the list in a table, using @foreach and @Html.DisplayFor Razor statements.

Chapter 6

Working with SQLite

The DbContext object handles the task of connecting to the database and mapping the objects to database records. The database context is registered with the Dependency Injection container in the ConfigureServices method in the Startup.cs.

You can use DB Browser or other apps to manage SQLite databases.

Chapter 7

Controller methods and views

In order to change the name and format of data when it gets displayed in the views, we have to go to the model for which we want to optimize the display of data, and do the following changes:

- Add using System.ComponentModel.DataAnnotations; to the used dependencies
- Add the data annotation that you wish to apply to the data.

In data annotations, the display attribute specifies what to display for the name of a field. The DataType attribute specifies the type of the data, so the time information store in the field is not displayed.

The Edit, Details and Delete links are generated by the Core MVC Anchor Tag Helper in the Index view of an entity. Tag Helpers enable server-side code to participate in creating and rendering HTML elements in Razor files. The AnchorTagHelper dynamically generated the HTML href attribute value from the controller action method and route id. We use View Source from our favorite browser or use the developer tools to examine the generated markup.

Here is a sample of a GET edit method in a controller that fetches the movie and populates the edit form generated by the Edit.cshtml Razor file:

```
public async Task<IActionResult> Edit(int? id).
{
    if(id == null)
    {
        return NotFound();
    }

    var data = await _context.[EntityName].SingleOrDefaultAsync(m => m.I
    if(data == null)
```

```
        {
            return NotFound();
        }

        return View(data);
    }
```

The next sample method shows a HTTP POST Edit method, which processes the posted values to change Movies details:

```
    //Post: Movies/Edit/5
    [HttpPost]
    [ValidateAntiForgeryToken]
    public async Task<IActionResult> Edit(int id, [Bind("ID,Title,ReleaseDat
    {
        if(id != movie.ID)
        {
            return NotFound();
        }
        if(ModelState.IsValid)
        {
            try
            {
                _context.Update(movie);
                await _context.SaveChangesAsync();
            }
            catch(DbUpdateConcurrencyException)
            {
                if(!MovieExists(movie.ID))
                {
                    return NotFound();
                }
                else
                {
                    throw;
                }
            }
            return RedirectToAction("Index");
        }
        return View(movie);
    }
```

The [Bind] attribute is one way to protect against over-posting. We should only include properties in the [Bind] attribute that we want to change. This Edit action is also preceded by the [HttpPost] attribute.

The HttpPost attribute specifies that this Edit method can be invoked only for POST requests. We could apply the [HttpGet] attribute to the first edit method, but that's not necessary because [HttpGet] is the default.

The ValidateAntiForgeryToken attribute is used to prevent forgery of a request and is paired up with an anti-forgery token generated in the edit view file. The edit view file generates the anti-forgery token with the Form Tag Helper: form asp-action="Edit".

The Form Tag Helper generates a hidden anti-forgery token that must match the [ValidateAntiForgeryToken] generated anti-forgery token in the Edit method of the Movies controller.

The HttpGet Edit method takes the movie ID parameter, looks up the movie using the Entity Framework SingleOrDefaultAsync method, and returns the selected movie to the Edit view. If a movie cannot be found, NotFound (HTTP 404) is returned.

When the scaffolding system creates the Edit view, it examined the Movie class and created code to render <label> and <input> elements for each property of the class.

When looking at a scaffolded view, we notice that a reference to the model is one of the first statements at the top of the file. The reference specifies that the view expects the model for the view template to be of a certain type.

The scaffolded code uses several Tag Helper method to streamline the HTML markup. The Label tag Helper displays the name of the field. The Input Tag Helper renders an HTML input element. The Validation Tag Helper displays any validation messages associated with that property.

Back to the controller, the [ValidateAntiForgeryToken] attribute validates the hidden XSRF token generated by the anti-forgery token generator in the Form Tag Helper.

The model binding system takes the posted form values and creates a Movie object that's passed as the movie parameter. The ModelState.IsValid method verifies that the data submitted in the form can be used to modify (edit or update) a Movie object. If the data is valid it's saved. The updated (edited) movie data is saved to the database by calling the SaveChangesAsync method of database context. After saving the data, the code redirects the user to the Index action method of the MoviesController class, which displays the movie collection, including the changes that have just been made.

Before the form is posted to the server, client side validation checks any validation rules on the fields. If there are any validation errors, an error message is displayed and the form is not posted. If JavaScript is disabled, we won't have client side validation but the server will detect the posted values that are not valid, and the form values will be redisplayed with error messages.

All the HttpGet methods in the movie controller follow a similar pattern. They get a movie object (or list of objects), and pass the object (model) to the view. The Create method passes an empty movie object to the Create view. All the methods that create, edit, delete or otherwise modify data do so in the [HttpPost] overload of the method. Modifying data in an HTTP GET method is a security risk. Modifying data in an HTTP GET method also violates HTTP best practices and the architectural REST pattern, which specifies that GET requests should not change the state of the application. In other words, performing a GET operation should be a safe operation that has no side effects

and doesn't modify our persisted data.

Chapter 8

Adding Search

In this section, we will work with examples of a movie renting web app and add search capability to the Index action method that lets us search movies by title. In order to look for movies only by title, we can modify the Index view in the following manner:

```
public async Task<IActionResult> Index(string searchString)
{
    var movies = from m in _context.Movie select m;

    if(!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title.Contains(searchString));
    }

    return View(await movies.ToListAsync());
}
```

The first line of the Index action method creates a LINQ query to select the movies. The query is only defined at this point, it has not been run against the database. If the searchString parameter contains a string, the movies query is modified to filter on the value of the search string. The s => s.Title.Contains() code above is a Lambda Expression. Lambdas are used in method-based LINQ queries as arguments to standard query operator methods such as the Where method or Contains (used in the code above). LINQ queries are not executed when they are defined or when they are modified by calling a method such as Where, Contains or OrderBy. Rather, query execution is deferred. That means that the evaluation of an expression is delayed until its realized value is actually iterated over or the ToListAsync method is called.

Note: The Contains method is run on the database, not in the c code shown above. The case sensitivity on the query depends on the database and the collation. On SQL Server, Contains maps to SQL LIKE, which is case insensitive. In SQLite, with the default collation, it's case sensitive.

If we change the signature of the Index method to have a parameter named id, the id parameter will match the option id placeholder for the default routes set in Startup.cs. That means that we can change the name of the parameter in the Index method and instead of having to type the name of the parameter in the URL segment, we can straight away write the string of the movie(s) we are looking for.

However, we can't expect users to modify the URL every time they want to search for a movie. So now we'll add UI to help them filter movies.

We add the UI by modifying the Index.cshtml file and add some <form> markup:

```
<form asp-controller="Movies" asp-action="Index">
    <p>
        Title: <input type="text" name="SearchString">
        <input type="submit" value="Filter" />
    </p>
</form>
```

The HTML form tag uses the Form Tag Helper, so when we submit the form, the filter string is posted to the Index action of the controller. There is no [HttpPost] overload of the Index method as we might expect. We don't need it, because the method isn't changing the state of the app, just filtering data. We could add a HTTP Post overload of the Index method, but it wouldn't do much benefit in this case, especially if you would like to share the search results with other people.

We can enforce the HTTP GET method, by adding the following parameter in the form Tag Helper: method="get".

Chapter 9

Adding validation