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/$_Layout.cshtml file.$

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 $_Layout.cshtmlfileforourmainlayout, itallowsustomakechangesonceinthewholewebapp, savin$
The Views/$_ViewStart.cshtmlfilebringsintheViews/Shared/_Layout.cshtmlfiletoeachview.Wecanusethe$
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 $_context.[EntityName].ToListAsync()$)

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

In this section we will learn how to add validation to models, and ensure that the validation rules are enforced every time a user creates or edits data.

9.1 Keeping things DRY

One of the design tenets of MVC is DRY ("Don't repeat yourself"). ASP.NET MVC encourages developers to specify functionality or behaviour only once, and then have it reflected everywhere in an app. This reduces the amount of code we need to write and makes the written code less error prone, easier to test and easier to maintain.

The validation support provided by MVC and EF Core Code First is a good example of the DRY principle in action. We can declaratively specify validation rules in one place (in the model class) and the rules are enforced everywhere in the app.

9.2 Adding validation rules to a model

DataAnnotations provides a built-in set of validation attributes that we apply declaratively to any class or property. (It also contains formatting like DataType that help with formatting and don'r provide any validation.)

In the DataAnnotations, we have useful validation attributes such as Required, StringLength, RegularExpression and Range.

The validation attributes specify behaviour that we want to enforce on the model properties they are applied to. The Required and MinimumLength attributes indicate that a property must have a value; but nothing prevents a user from entering white spaces to satisfy this validation. The RegularExpression attribute is used to limit what characters can be input. The Range attribute constrains a value within a specified range. The StringLength attribute lets us set the maximum length of a string property, and optionally its minimum length.

Value types (such as decimal, int, float, DateTime) are inherently required and don't need the [Required] attribute.

Having validation rules automatically enforced by ASP.NET helps make our app more robust. It also ensures that we can't forget to validate something and inadvertently let bad data into the database.

9.3 Validation Error UI in MVC

If we try to fill out a form with some invalid values, as soon as jQuery client side validation detects the error, it displays an error message. The form will automatically render an appropriate validation error message in each field containing an invalid value. The errors are enforced both client-side (using JavaScript and jQuery) and server-side(in case a user has JavaScript) disabled.

A significant benefit is that we didn't need to change a single line of code in the rest of the code in order to enable this validation UI. The controller and views automatically pick up the validation rules that we specify by using validation attributes on the properties of the model.

The form data is not sent to the server until there are no client side validation errors.

9.4 How validation works

We might wonder how the validation UI was generated without any updates to the code in the controller or views. In the POST controller methods, the ModelState.IsValid is called to check whether there are any validation errors, that is sent done by checking that the data corresponds to what is stated in the Model, bringing the responsibility of checking the data back to it, and not requiring any changes in the Controller code.

In the view, when using the Input Tag Helpers we enforce validation with the asp-validation-for attribute. When rendering the page, the app fetches the validation information from the model, therefore making it unnecessary to change any code in the view.

What's really nice about this approach is that neither the controller nor the view know anything about the actual validation rules being enforced or about the specific error messages. The validation rules and the error strings are specified only in the model. These same validation rules are automatically applied to any views that we might create that somehow edit the model.

When we need to change validation logic, we can do so in exactly one place by adding validation attributes to the model. We won't have to worry about different parts of the application being inconsistent with how the rules are enforced - all validation logic will be defined in one place and used everywhere. This keeps the code very clean, and makes it easy to maintain and evolve. And it means that we'll be fully honoring the DRY principle.

9.5 Using DataType Attributes

The System.ComponentModel.DataAnnotations namespace provides formatting attributes in addition to the built-in set of validation attributes. The DataType attributes only provide hints for the view engine to format data (and supply attributes such as <a> for URLs and mailto for emails). We can use the RegularExpression attribute to validate the format of the data. The DataType attribute is used to specify a data type that is more specific than the database intrinsic type, they are not validation attributes. In this case we only want to keep track of the data, not the time. The DateType enumeration provides for many data types, such as Date, Time, PhoneNumber, Currency, EmailAddress and more. The DateType attribute can also enable the application to automatically provide type-specific features. For example, a mailto: link can be created for DataType.EmailAddress, and a date selector can be provided for DataType.Date in browsers that support HTML5. The DataType attributes emits HTML 5 data- attributes that HTML 5 browsers can understand. The DataType attributes do not provide validation.

DataType.Date does not specify the format of the data that is displayed. By default, the data field is displayed according to the default formats based on the server's CultureInfo.

The DisplayFormat attribute is used to explicitly specify the date format. The ApplyFormatInEditMode setting specifies that the formatting should also be applied when the value is displayed in a text box for editing. We can use the DisplayFormat attribute by itself, but it's generally a good idea to use the DataType attribute. The DataType attribute conveys the semantics of the data as opposed to how to render it on a screen, and provides the following benefits that we don't get with DisplayFormat:

- The browser can enable HTML5 features (calendar control, the locale-appropriate currency symbol, email links, etc.)
- By default, the browser will render data using the correct format based on the locale
- The DataType attribute can enable MVC to choose the right field template to render the data (the DisplayFormat is used by itself uses the string template)

Note: jQuery validation does not work with the Range attribute and DateTime when used together.

We will need to disable jQuery date validation to use the Range attribute with DateTime. It's generally not a good practice to compile hard dates in our models, so using the Range attribute and DateTime is discouraged.

Chapter 10

Examining the Delete methods

There are two method related to deleting an entry from the database, the Delete and DeleteConfirmed methods that are scaffolded from the Models. Using again the movie database example, here is what they look like:

```
//GET: Movies/Delete/5
public async Task<IActionResult> Delete(int? id)
{
    if(id == null)
    {
        return NotFound();
    }

    var movie = await _context.Movie
        .SingleOrDefaultAsync(m => m.ID == id);
    if(movie == null)
    {
        return NotFound();
    }

    return View(movie);
}

//POST: Movies/Delete/5
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
    var movie = await _context.Movie
        SingleOrDefaultAsync(m => m.ID == id);
```

```
            _context.Movie.Remove(movie);
            await _context.SaveChangesAsync();
            return RedirectToAction("Index");
        }
```

Note that the HTTP GET Delete method doesn't delete the specified movie, it returns a view of the movie where we can submit (HttpPost) the deletion. Performing a delete operation in response to a GET request opens up a security hole.

The [HttpPost] method that deletes the data is named DeleteConfirmed to give the HTTP POST method a unique signature or name. The CLR requires overload methods to have a unique parameter signature. However, here we have two Delete methods – one for GET and one for POST – that both have the same parameter signature.

There are two approaches to this problem, one is to give the methods different names. That's what the scaffolding mechanism did in the preceding example. However, this introduces a small problem: ASP.NET maps segments of a URL to action methods by name, and if we rename a method, routing normally wouldn't be able to find that method. The solution is what we see in the example, which is to add the ActionName("Delete") attribute to the DeleteConfirmed method. That attribute performs mapping for the routing system so that a URL that includes /Delete/ for a POST request will find the DeleteConfirmed method.

Another common work around for methods that have identical names and signatures is to artificially change the signature of the POST method to include an extra (unused) parameter.

Chapter 11

Introduction to Identity on ASP.NET Core

ASP.NET Core Identity is a membership system which allows us to add login functionality to our application. Users can create an account and login with a username and password or they can use an external login provider such as Facebook, Google, Microsoft Account, Twitter or others. We can configure ASP.NET Core Identity to use a SQL Server database to store usernames, passwords and profile data. Alternatively, we can use our own persistent store.

11.1 Overview of Identity

In this topic we will learn how to use ASP.NET Core Identity to add functionality to register, log in, and log out a user.

- Create an ASP.NET Core Web Application project with Individual User Accounts
  - Create the new project using ”dotnet new mvc –auth Individual”
  - The created project contains the Microsoft.AspNetCore.Identity.EntityFrameworkCore package, which persists the Identity data and schema to SQL Server using EF Core
- Configure Identity services and add middleware in Startup.
  - The Identity services are added to the application in the ConfigureServices method in the Startup class:

```
public void ConfigureServices(IServiceCollection services
{
    services.AddDbContext<ApplicationDbContext>(options)
    //The line above is previously set up code so I cut
    //out of it
```

```
        services.AddIdentity<ApplicationUser, IdentityRole>(
            .AddEntityFrameworkStores<ApplicationDbContext>(
            .AddDefaultTokenProviders();

        services.Configure<IdentityOptions>(options =>
        {
            //Password settings
            options.Password.RequireDigit = true;
            options.Password.RequiredLength = 8;
            options.Password.RequireNonAlphanumeric = false;
            options.Password.RequireUppercase = true;
            options.Password.RequireLowercase = false;
            options.Password.RequiredUniqueChars = 6;

            //Lockout settings
            options.Lockout.DefaultLockoutTimeSpan = TimeSpa
                .FromMinutes(30);
            options.Lockout.MaxFailedAccessAttempts = 10;
            options.Lockout.AllowsForNewUsers = true;

            //User settings
            options.User.RequireUniqueEmail = true;
        });

        services.ConfigureApplicationCookie(options =>
        {
            //Cookie settings
            options.Cookie.HttpOnly = true;
            options.Cookie.Expiration = TimeSpan.FromDays(15
            options.LoginPath = "/Account/Login";
            options.LogoutPath = "/Account/Logout";
            options.AccessDeniedPath = "/Account/AccessDenie
            options.SlidingExpiration = true;
        });
    }
```

  - These services are made available to the application through dependency injection. Identity is enabled for the application by calling UseAuthentication in the Configure method. UseAuthentication adds authentication middleware to the requested pipeline.

- Create a user

  Launch the application and then click on the Register link. If this is the first time we're performing this action, we may be required to run migrations. The application prompts us to Apply Migrations. Alternatively, we can test using ASP.NET Core Identity with our app without a persistent database by using an in-memory database. To use an in-memory database, we need to add the Microsoft.EntityFrameworkCore.InMemory package to our app and modify our app's call to AddDbContext in ConfigureServices as follows:

```
            services.AddDbContext<ApplicationDbContext>(options =>
                options.UseInMemoryDatabase(Guid.NewGuid().ToString()));
```

When the user clicks the Register link, the Register action is invoked on AccountController. The Register action creates the user by calling CreateAsync on the $_userManagerobject(providedtoAccountControllerbydependencyinjection)$:

```
        var result = await _userManager.CreateAsync(user, model.Password);
```

If the user was created successfully, the user is logged in by the call to $_signInManager.SignInAsync$.

11.1.1 Log in

Users can sign in by clicking the Log in link at the top of the site, or they may be navigated to the Login page if they attempt to access a part of the site that requires authorization. When the user submits the form on the Login page, the AccountController Login action is called:

```
        [HttpPost]
        [AllowAnonymous]
        [ValidateAntiForgeryToken]
        public async Task<IActionResult> Login(LoginViewModel model, string retu
        {
            ViewData["ReturnUrl"] = returnURL;
            if(ModelState.IsValid)
            {
                //This doesn't count login failures towards account Lockout
                // To enable password failures to trigger account lockout,
                //set lockoutOnFailure: true
                var result = await _signInManager.PasswordSignInAsync(model.Emai
                    model.Password, model.RememberMe, lockoutOnFailure: false);
                if(result.Succeeded)
                {
                    _logger.LogInformation(1, "User logged in.");
                    return RedirectToLocal(returnUrl);
                }
                if(result.RequiresTwoFactor)
                {
                    return RedirectToAction(nameof(SendCode),
                        new{ReturnUrl = returnUrl,
                            RememberMe = model.RememberMe});
                }
                if(result.IsLockedOut)
                {
                    _logger.LogWarning(2, "User␣account␣is␣locked␣out");
                    return View("Lockout");
                }
                else
                {
                    ModelState.AddModelError(string.Empty, "Invalid␣login␣attemp
                    return View(model);
                }
            }

            //If we got this far, something failed, redisplay form
            return View(model);
        }
```

The Login action calls PasswordSignInAsync on the signInManager object (provided to AccountController by dependency injection).

The base Controller class exposes a User property that we can access from controller methods. For instance, we can enumerate User.Claims and make authorization decisions.

11.1.2 Log out

Clicking the Log out link on a web page calls the LogOut action:

```
    //POST: /Account/LogOut
    [HttpPost]
    [ValidateAntiForgeryToken]
    public async Task<IActionResult> LogOut()
    {
        await _signInManager.SignOutAsync();
        _logger.LogInformation(4, "User␣logged␣out.");
        return RedirectToAction(nameof(HomeController.Index). "Home");
    }
```

The preceding code above calls the signInManager.SignOutAsync method. The SignOutAsync method clears the user's claims stored in a cookie.

11.2 Identity Components

The primary reference assembly for the Identity system is Microsoft.AspNetCore.Identity. This package contains the core set of interfaces for ASP.NET Core Identity, and is included by Microsoft.AspNetCore.Identity.EntityFrameworkCore.

These dependencies are needed to use the Identity system in ASP.NET Core applications:

- Microsoft.AspNetCore.Identity.EntityFrameworkCore - Contains the required types to use Identity with Entity Framework Core.
- Microsoft.EntityFrameworkCore.SqlServer - Entity Framework is Microsoft's recommended data access technology for relation databases like SQL Server. For testing, we can use Microsoft.EntityFrameworkCore.InMemory.
- Microsoft.AspNetCore.Authentication.Cookies - Middleware that enables an app to use cookie-based authentication.

Chapter 12

Configure Identity

ASP.NET Core Identity has some default behaviours that we can override easily in our application's Startup class.

12.1 Passwords policy

By default, Identity requires that passwords contain an uppercase character, lowercase character, a digit, and a non-alphanumeric character. There are also some other restrictions. If we want to simplify password restrictions, we can do that in the Startup class of our application.

ASP.NET Core 2.0 added the RequiredUniqueChars property. Otherwise, the options are the same from ASP.NET COre 1.x.

```
        services.AddIdentity<ApplicationUser, IdentityRole>(options =>
        {
            //Password settings
            options.Password.RequireDigit = true;
            options.Password.RequiredLength = 8;
            options.Password.RequireNonAlphanumeric = true;
            options.Password.RequireUppercase = true;
            options.Password.RequireLowercase = true;
            options.Password.RequiredUniqueChars = 2;
        })
        .AddEntityFrameworkStores<ApplicationDbContext();
        .AddDefaultTokenProviders();
```

12.2 User's lockout

```
        services.AddIdentity<ApplicationUser, IdentityRole>(options =>
        {
            //Lockout settings
            options.Lockout.DefaultLockoutTimeSpan = TimeSpan.FromMinutes(5);
            options.Lockout.MaxFailedAccessAttempts = 5;
            options.Lockout.AllowedForNewUsers = true;
        })
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultTokenProviders();
```

IdentityOptions.Lockout has the following properties:

- DefaultLockoutTimeSpan: The amount of time a user is locked out when a lockout occurs. Defaults to 5 minutes.
- MaxFailedAccessAttempts: The number of failed access attempts until a user is locked out, if lockout is enabled. Defaults to 5.
- AllowedForNewUsers: Determines if a new user can be locked out. Defaults to true.

12.3 Sign in settings

```
        services.AddIdentity<ApplicationUser, IdentityRole>(options =>
        {
            //Signin settings
            options.SignIn.RequireConfirmedEmail = true;
            options.SignIn.RequireConfirmedPhoneNumber = false;
        })
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultTokenProviders();
```

Identity.Options.SignIn has the following properties:

- RequireConfirmedEmail: Requires a confirmed email to sign in. Defaults to false
- RequireConfirmedPhoneNumber: Requires a confirmed phone number to sign in. Defaults to false.

12.4 User validation settings

```
        services.AddIdentity<ApplicationUser, IdentityRole>(options =>
        {
            //User settings
            options.User.RequireUniqueEmail = true;
        })
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultTokenProviders();
```

Identity.User has the following properties:

- RequireUniqueEmail: Requires each User to have a unique email. Defaults to false.

- AllowedUserNameCharacters: Allowed characters in the username. Defaults to alphanumeric characters and -.@+

12.5 Application's cookie settings

Like the passwords policy, all the settings of the application's cookie can be changed in the Startup class.

```
services.ConfigureApplicationCookie(options =>
{
    options.Cookie.Name = "YourAppCookieName";
    options.Cookie.HttpOnly = true;
    options.ExpireTimeSpan = TimeSpan.FromMinutes(60);
    options.LoginPath = "/Account/Login";
    options.LogoutPath = "/Account/Logout";
    options.AccessDeniedPath = "/Account/AccessDenied";
    options.SlidingExpiration = true;
    //Requires using Microsoft.AspNetCore.Authentication.Cookies;
    options.ReturnUrlParameter = CookieAuthenticationDefaults.ReturnUrlP
});
```

CookieAuthenticationOptions has the following properties:

- Cookie.Name: The name of the cookie. Defaults to.AspNetCore.Cookies.

- Cookie.HttpOnly: When true, the cookie is not accessible from client-side scripts. Defaults to true.

- ExpireTimeSpan: Controls how much time the authentication ticket is stored in the cookie will remain valid from the point it is created. Defaults to 14 days.

- LoginPath: When a user is unauthorized, they will be redirected to this path to login. Defaults to /Account/Login.

- LogoutPath: When a user is logged out, they will be redirected to this path to login. Defaults to /Account/Logout.

- AccessDeniedPath: When a user fails an authorization check, they will be redirected to this path. Defaults to /Account/AccessDenied.

- SlidingExpiration: When true, a new cookie will be issued with a new expiration time when the current cookie is more than halfway through the expiration window. Defaults to true.

- ReturnUrlParameter: The ReturnUrlParameter determines the name of the query string parameter which is appended by the middleware when a 401 Unauthorized status code is changed to a 302 redirect onto the login path.

Chapter 13

Custome storage providers for ASP.NET Core Identity

ASP.NET Core Identity is an extensible system which enables us to create a custom storage provider and connect it to our app.

13.1 Introduction

By default, the ASP.NET Core Identity system stores user information in a SQL Server database using Entity Framework Core. For many apps, this approach works well. However, we may prefer to use a different persistence mechanism or data schema. We can write a customized provider for our storage mechanism and plug that provider into our app.

13.2 The ASP.NET Core Identity architecture

ASP.NET Core Identity consists of classes called managers and stores. Managers are high-level classes which an app developer uses to perform operations, such as creating an Identity user. Stores are lower-level classes that specify how entities, such as users and roles, are persisted. Stores follow the repository pattern and are closely coupled with the persistence mechanism. Managers are decoupled from stores, which means we can replace the persistence mechanism without changing our application code (except for configuration).

The web app interacts the following way: ASP.NET Core app - Identity Manager - Identity Store - Data Access Layer - Data Source.

To create a custom storage provider, create the data source, the data access layer and the store classes that interact with this data access layer. We don't need to customize the managers or our app code that interacts with them.

When creating a new instance of UserManager or RoleManager we can provide the type of the user class and pass an instance of the store class as an argument. This approach enables us to plug our customized classes into ASP.NET Core.

13.3 ASP.NET Core Identity stores data types

ASP.NET Core Identity data types are detailed in the following sections:

13.3.1 Users

Registered users of our web site. The IdentityUser type may be extended or used as an example for our own custom type. We do not need to inherit from a particular type to implement our own custom identity storage solution.

13.3.2 User Claims

A set of statements (or Claims) about the user that represent the user'safe identity. Can enable greater expression of the user's identity than can be achieved through roles.

13.3.3 User Logins

Information about the external authentication provider (like Facebook or a Microsoft account) to use when logging a user.

13.3.4 Roles

Authorization groups for our site. Includes the role ID and role name (like ”Admin” or ”Employee”).

13.4 The data access layer

This topic assumes we are familiar with the persistence mechanism that we are going to use and how to create entities for that mechanism. This topic does not provide details about how to create the repositories or data access classes; it provides some suggestions about design decisions when working with ASP.NET Core Identity.

We have a lot of freedom when designing the data access layer for a customized store provider. We only need to create persistence mechanisms for features that we intend to use in our app. For example, if we are not using roles in our app, we do not need to create storage for roles or user role associations. Our technology and existing infrastructure may require a structure that is very different from the default implementation of ASP.NET Core Identity. In our data access layer, we provide the logic to work with the structure of our storage implementation.

The data access layer provides the logic to save the data from ASP.NET Core Identity to a data source. The data access layer for our customized storage provider might include the following classes to store user and role information.

13.4.1 Context class

Encapsulates the information to connect to our persistence mechanism and execute queries. Several data classes require an instance of this class, typically provided through dependency injection.

13.4.2 User Storage

Stores and retrieves user information (such as user name and password hash).

13.4.3 Role Storage

Stores and retrieves role information (such as the role name).

13.4.4 UserClaims Storage

Stores and retrieves user claim information (such as the claim type and value).

13.4.5 UserLogins Storage

Stores and retrieves user login information (such as an external authentication provider).

13.4.6 UserRole Storage

Stores and retrieves which roles are assigned to which user. Tip: Only implement the classes we intend to use in our app.

In the data access classes, provide code to perform data operations for our persistence mechanism. For example, within a custom provider, we might have the following code to create a new user in the store class:

```
public async Task<IdentityResult> CreateAsync(ApplicationUser user,
    Cancellation Token cancellationToken = default(CancellationToken))
{
    cancellationToken.ThrowIfCancellationRequested();
    if(user == null) throw new ArgumentNullException(nameof(User));

    return await _usersTable.CreateAsync(user);
}
```

The implementation logic for creating the user is in the usersTable.CreateAsync method.

13.5 Customize the user class

When implementing a storage provider, create a user class which is equivalent to the IdentityUser class.

At a minimum, our user class must include an Id and a UserName property. The IdentityUser class defines properties that the UserManager calls when performing requested operations. The default type of the Id property is a a string, but we can inherit from IdentityUser<TKey, TUserClaim, TUserRole, TUserRole, TUserLogin, TUserToken> and specify a different type. The framework expects the storage implementation to handle data type conversions.

13.6 Customize the user store

Create a UserStore class that provides methods for all data operations on the user. This class is equivalent to the UserStore class. In our UserStore class, implement IUserStore<TUser> and the optional interfaces required. We select which optional interfaces to implement based on the functionality provided in our app.

$OptionalInterfaces$

- IUserRoleStore
- IUserClaimStore
- IUserPasswordStore
- IUserSecurityStampStore
- IUserEmailStore
- IPhoneNumberStore
- IQueryableUserStore
- IUserLoginStore
- IUserTwoFactorStore
- IUserLockoutStore

The optional interfaces inherit from IUserStore. We can see a partially implemented sample user store below:

```
using Microsoft.AspNetCore.Identity;
using System;
using System.Threading.Tasks;
using System.Threading;

namespace CustomIdentityProviderSample.CustomProvider
{
```

```
///<summary>
///This store is only partially implemented. It supports
///user creation and find methods.
///</summary>
public class CustomUserStore: IUserStore<ApplicationUser>,
    IUserPasswordStore<ApplicationUser>
{
    private readonly DapperUsersTable _usersTable;

    public CustomUserStore (DapperUsersTable usersTable)
    {
        _usersTable = usersTable;
    }

    #region createuser
    public async Task<IdentityResult> CreateAsync(ApplicationUser us
        CancellationToken cancellationToken = default(CancellationTo
    {
        cancellationToken.ThrowIfCancellationRequested();
        if(user == null) throw new ArgumentNullException(nameof(user

        return await _usersTable.CreateAsync(user);
    }
    #endregion

    public async Task<IdentityResult> DeleteAsync(ApplicationUser us
        CancellationToken cancellationToken = default(CancellationTo
    {
        cancellationToken.ThrowIfCancellationRequested();
        if(user == null) throw new ArgumentNullException(nameof(user

        return await _usersTable.DeleteAsync(user);
    }

    public void Dispose()
    {
    }

    public async Task<ApplicationUser> FindByIdAsync(string userId,
        CancellationToken cancellationToken = default(CancellationTo
    {
        cancellationToken.ThrowIfCancellationRequested();
        if (userId == null) throw new ArgumentNullException(nameof(u
        Guid idGuid;
        if(!Guid.TryParse(userId, out idGuid))
        {
```

```
        throw new ArgumentException("Not␣a␣valid␣Guid␣id", namec
    }

    return await _usersTable.FindByIdAsync(idGuid);

}

public async Task<ApplicationUser> FindByNameAsync(string userNar
    CancellationToken cancellationToken = default(CancellationTo
{
    cancellationToken.ThrowIfCancellationRequested();
    if (userName == null) throw new ArgumentNullException(nameof

    return await _usersTable.FindByNameAsync(userName);
}

public Task<string> GetNormalizedUserNameAsync(ApplicationUser u
{
    throw new NotImplementedException();
}

public Task<string> GetPasswordHashAsync(ApplicationUser user, C
{
    cancellationToken.ThrowIfCancellationRequested();
    if (user == null) throw new ArgumentNullException(nameof(use

    return Task.FromResult(user.PasswordHash);
}

public Task<string> GetUserIdAsync(ApplicationUser user, Cancella
{
    cancellationToken.ThrowIfCancellationRequested();
    if (user == null) throw new ArgumentNullException(nameof(use

    return Task.FromResult(user.Id.ToString());
}

public Task<string> GetUserNameAsync(ApplicationUser user, Cance
{
    cancellationToken.ThrowIfCancellationRequested();
    if (user == null) throw new ArgumentNullException(nameof(use

    return Task.FromResult(user.UserName);
}

public Task<bool> HasPasswordAsync(ApplicationUser user, Cancella
                {
                    throw new NotImplementedException();
                }

                public Task SetNormalizedUserNameAsync(ApplicationUser user, str
                {
                    cancellationToken.ThrowIfCancellationRequested();
                    if (user == null) throw new ArgumentNullException(nameof(use
                    if (normalizedName == null) throw new ArgumentNullException(

                    user.NormalizedUserName = normalizedName;
                    return Task.FromResult<object>(null);
                }

                public Task SetPasswordHashAsync(ApplicationUser user, string pa
                {
                    cancellationToken.ThrowIfCancellationRequested();
                    if (user == null) throw new ArgumentNullException(nameof(use
                    if (passwordHash == null) throw new ArgumentNullException(na

                    user.PasswordHash = passwordHash;
                    return Task.FromResult<object>(null);

                }

                public Task SetUserNameAsync(ApplicationUser user, string userNam
                {
                    throw new NotImplementedException();
                }

                public Task<IdentityResult> UpdateAsync(ApplicationUser user, Ca
                {
                    throw new NotImplementedException();
                }
            }
        }
```

Within the UserStore class, we use the data access classes that we created to perform operations. These are passed in using dependency injection. For example, in the SQL Server with Dapper implementation, the UserStore class has the CreateAsync method which uses an instance of DapperUsersTable to insert a new record:

```
        public async Task<IdentityResult> CreateAsync(ApplicationUser user)
        {
            string sql = "INSERT INTO dbo.CustomUser " +
                "VALUES (@id, @Email, @EmailConfirmed, @PasswordHash, @UserName)
            int rows = await _connection.ExecuteAsync(sql, new{user.Id, user.Em
            if(rows > 0)
            {
                return IdentityResult.Success;
            }
            return IdentityResult.Failed(new IdentityError{ Description = $"Coul
        }
```

13.6.1 Interfaces to implement when customoizing user store

- IUserStore
  The IUserStore<TUser> interface is the only interface we must implement in the user store. It defines methods for creating, updating, deleting and retrieving users.
- IUserClaimStore
  The IUserClaimStore<TUser> interface defines the methods we implement to enable user claims. It contains methods for adding, removing and retrieving users.
- IUserLoginStore
  The IUserLoginStore<TUser> defines the methods we implement to enable external authentication providers. It contains methods for adding, removing and retrieving user logins, and a method for retrieving a user based on the login information.
- IUserRoleStore
  The IUserRoleStore<TUser> interface defines the methods we implement to map a user to a role. It contains methods to add, remove, and retrieve a user's roles, and a method to check if a user is assigned to a role.
- IUserPasswordStore
  The IUserPasswordStore<TUser> interface defines the methods we implement to persist hashed passwords. It contains methods for getting and setting the hashed password, and a method that indicates whether the user has set a password.
- IUserSecurityStampStore
  The IUserSecurityStampStore<TUser> interface defines the methods we implement to use a security stamp for indicating whether the user's account information has changed. This stamp is updated when a user changes the password, or adds or removes logins. It contains methods for getting and setting the security stamp.

- IUserTwoFactorStore
  The IUserTwoFactorStore<TUser> interface defines the methods we implement to support two factor authentication. It contains methods for getting and setting whether two factor authentication is enabled for a user.

- IUserPhoneNumberStore
  The IUserPhoneNumberStore<TUser> interface defines the methods we implement to store user phone numbers. It contains methods for getting and setting the phone number and whether the phone number is confirmed.

- IUserEmailStore
  The IUserEmailStore<TUser> interface defines the methods we implement to store user email addresses. It contains methods for getting and setting the email address and whether the email address is confirmed.

- IUserLockoutStore
  The IUserLockoutStoreM<TUser> interface defines the methods we implement to store information about locking an account. It contains methods for tracking failed access attempts and lockouts.

- IQueryableUserStore
  The IQueryableUserStore<TUser> interface defines the members implement to provide a queryable user store.

We implement only the interfaces that are needed in our app. For example:

```
public class UserStore: IUserStore<IdentityUser>,
    IUserClaimStore<IdentityUser>,
    IUserLoginStore<IdentityUser>,
    IUserRoleStore<IdentityUser>,
    IUserPasswordStore<IdentityUser>,
    IUserSecurityStampStore<IdentityUser>
{
    //interface implementations not shown
}
```

13.6.2 IdentityUserClaim, IdentityUserLogin, and IdentityUserRole

The Microsoft.AspNet.Identity.EntityFramework namespace contains implementations of the IdentityUserClaim, IdentityUserLogin and IdentityUserRole classes. If we are using these features, we may want to create our own versions of these classes and define properties for our app. However, sometimes it is more efficient to not load these entities into memory when performing basic operations (such as adding or removing a user's claim). Instead, the backend store classes can execute these operations directly on the data source. For example, the UserStore.GetClaimsAsync method can call the userClaimTable.FindByUserId(user.Id) method to execute a query on that table directly and return a list of claims.

13.7 Customize the role class

When implementing a role storage provider, we can create a custom role type. It need not implement a particular interface, but it must have an Id and typically it will have a Name property.