the little **ASP.NET Core** book

Nate Barbettini

Table of Contents

Introduction

Thanks for picking up the Little ASP.NET Core Book! I wrote this short book to help developers and people interested in web programming learn about ASP.NET Core 2.0, a new framework for building web applications and APIs.

The Little ASP.NET Core Book is structured as a tutorial. You'll build a to-do app from start to finish and learn:

- The basics of the MVC (Model-View-Controller) pattern
- How front-end code (HTML, CSS, JavaScript) works together with back-end code
- What dependency injection is and why it's useful
- How to read and write data to a database
- How to add log-in, registration, and security
- How to deploy the app to the web

Don't worry, you don't need to know anything about ASP.NET Core (or any of the above) to get started.

Before you begin

The code for the finished version of the application you'll build is available on GitHub (https://www.github.com/nbarbettini/little-aspnetcore-todo). Feel free to download it if you want to compare as you write your own code.

The book itself is updated frequently with bug fixes and new content. If you're reading a PDF, e-book, or print version, check the official website (littleasp.net/book) to see if there's an updated version available. The very last page of the book contains version information and a changelog.

Reading in your own language

Thanks to some fantastic multilingual folks, the Little ASP.NET Core Book has been translated into other languages:

Turkish - https://sahinyanlik.gitbooks.io/kisa-asp-net-core-kitabi/

Chinese - https://windsting.github.io/little-aspnetcore-book/book/

Who this book is for

If you're new to programming, this book will introduce you to the patterns and concepts used to build modern web applications. You'll learn how to build a web app (and how the big pieces fit together) by building something from scratch! While this little book won't be able to cover absolutely everything you need to know about programming, it'll give you a starting point so you can learn more advanced topics.

If you already code in a backend language like Node, Python, Ruby, Go, or Java, you'll notice a lot of familiar ideas like MVC, view templates, and dependency injection. The code will be in C#, but it won't look too different from what you already know.

If you're an ASP.NET MVC developer, you'll feel right at home! ASP.NET Core adds some new tools and reuses (and simplifies) the things you already know. I'll point out some of the differences below.

No matter what your previous experience with web programming, this book will teach you everything you need to create a simple and useful web application in ASP.NET Core. You'll learn how to build functionality using backend and frontend code, how to interact with a database, and how to test and deploy the app to the world.

What is ASP.NET Core?

ASP.NET Core is a web framework created by Microsoft for building web applications, APIs, and microservices. It uses common patterns like MVC (Model-View-Controller), dependency injection, and a request pipeline comprised of middleware. It's open-source under the Apache 2.0 license, which means the source code is freely available, and the community is encouraged to contribute bug fixes and new features.

ASP.NET Core runs on top of Microsoft's .NET runtime, similar to the Java Virtual Machine (JVM) or the Ruby interpreter. You can write ASP.NET Core applications in a number of languages (C#, Visual Basic, F#). C# is the most popular choice, and it's what I'll use in this book. You can build and run ASP.NET Core applications on Windows, Mac, and Linux.

Why do we need another web framework?

There are a lot of great web frameworks to choose from already: Node/Express, Spring, Ruby on Rails, Django, Laravel, and many more. What advantages does ASP.NET Core have?

- **Speed.** ASP.NET Core is fast. Because .NET code is compiled, it executes much faster than code in interpreted languages like JavaScript or Ruby. ASP.NET Core is also optimized for multithreading and asynchronous tasks. It's common to see a 5-10x speed improvement over code written in Node.js.
- **Ecosystem.** ASP.NET Core may be new, but .NET has been around for a long time. There are thousands of packages available on NuGet (the .NET package manager; think npm, Ruby gems, or Maven). There are already packages available for JSON deserialization, database connectors, PDF generation, or almost anything else you can think of.
- **Security.** The team at Microsoft takes security seriously, and ASP.NET Core is built to be secure from the ground up. It handles things like sanitizing input data and preventing cross-site request forgery (XSRF) automatically, so you don't have to. You also get the benefit of static typing with the .NET compiler, which is like having a very paranoid linter turned on at all times. This makes it harder to do something you didn't intend with a variable or chunk of data.

.NET Core and .NET Standard

Throughout this book, you'll be learning about ASP.NET Core (the web framework). I'll occasionally mention the .NET runtime (the supporting library that runs .NET code).

You may also hear about .NET Core and .NET Standard. The naming gets confusing, so here's a simple explanation:

.NET Standard is a platform-agnostic interface that defines what features and APIs are available in .NET. .NET Standard doesn't represent any actual code or functionality, just the API definition. There are different "versions" or levels of .NET Standard that reflect how many APIs are available (or how wide the API surface area is). For example, .NET Standard 2.0 has more APIs available than .NET Standard 1.5, which has more APIs than .NET Standard 1.0.

.NET Core is the .NET runtime that can be installed on Windows, Mac, or Linux. It implements the APIs defined in the .NET Standard interface with the appropriate platform-specific code on each operating system. This is what you'll install on your own machine to build and run ASP.NET Core applications.

And just for good measure, **.NET Framework** is a different implementation of .NET Standard that is Windows-only. This was the only .NET runtime until .NET

Core came along and opened .NET up to Mac and Linux. ASP.NET Core can also run on Windows-only .NET Framework, but I won't touch on this too much.

If you're confused by all this naming, no worries! We'll get to some real code in a bit.

A note to ASP.NET 4 developers

If you haven't used a previous version of ASP.NET, skip ahead to the next chapter!

ASP.NET Core is a complete ground-up rewrite of ASP.NET, with a focus on modernizing the framework and finally decoupling it from System.Web, IIS, and Windows. If you remember all the OWIN/Katana stuff from ASP.NET 4, you're already halfway there: the Katana project became ASP.NET 5 which was ultimately renamed to ASP.NET Core.

Because of the Katana legacy, the `Startup` class is front and center, and there's no more `Application_Start` or `Global.asax`. The entire pipeline is driven by middleware, and there's no longer a split between MVC and Web API: controllers can simply return views, status codes, or data. Dependency injection comes baked in, so you don't need

to install and configure a container like StructureMap or Ninject if you don't want to. And the entire framework has been optimized for speed and runtime efficiency.

Alright, enough introduction. Let's dive in to ASP.NET Core!

Your first application

Ready to build your first web app with ASP.NET Core? You'll need to gather a few things first:

Your favorite code editor. You can use Atom, Sublime, Notepad, or whatever editor you prefer writing code in. If you don't have a favorite, give Visual Studio Code a try. It's a free, cross-platform code editor that has rich support for writing C#, JavaScript, HTML, and more. Just search for "download visual studio code" and follow the instructions.

If you're on Windows, you can also use Visual Studio to build ASP.NET Core applications. You'll need Visual Studio 2017 version 15.3 or later (the free Community Edition is fine). Visual Studio has great code completion and other features specific to C#, although Visual Studio Code is close behind.

The .NET Core SDK. Regardless of the editor or platform you're using, you'll need to install the .NET Core SDK, which includes the runtime, base libraries, and command line tools you need for building ASP.NET Core apps. The SDK can be installed on Windows, Mac, or Linux.

Once you've decided on an editor, you'll need to get the SDK.

Get the SDK

Search for "download .net core" and follow the instructions on Microsoft's download page for your platform. After the SDK has finished installing, open up the Terminal (or PowerShell on Windows) and use the `dotnet` command line tool (also called a **CLI**) to make sure everything is working:

```
dotnet --version

2.0.0
```

You can get more information about your platform with the `--info` flag:

```
dotnet --info

.NET Command Line Tools (2.0.0)

Product Information:
 Version:            2.0.0
 Commit SHA-1 hash:  cdcd1928c9

Runtime Environment:
 OS Name:     Mac OS X
 OS Version:  10.12

(more details...)
```

If you see output like the above, you're ready to go!

Hello World in C#

Before you dive into ASP.NET Core, try creating and running a simple C# application.

You can do this all from the command line. First, open up the Terminal (or PowerShell on Windows). Navigate to the location you want to store your projects, such as your Documents directory:

```
cd Documents
```

Use the `dotnet` command to create a new project:

```
dotnet new console -o CsharpHelloWorld
cd CsharpHelloWorld
```

This creates a basic C# program that writes some text to the screen. The program is comprised of two files: a project file (with a `.csproj` extension) and a C# code file (with a `.cs` extension). If you open the former in a text or code editor, you'll see this:

`CsharpHelloWorld.csproj`

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>

</Project>
```

The project file is XML-based and defines some metadata about the project. Later, when you reference other packages, those will be listed here (similar to a `package.json` file for npm). You won't have to edit this file by hand often.

`Program.cs`

```
using System;

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

`static void Main` is the entry point method of a C# program, and by convention it's placed in a class (a type of code structure or module) called `Program`. The `using` statement at the top imports the built-in System classes from .NET and makes them available to the code in the class.

From inside the project directory, use `dotnet run` to run the program. You'll see the output written to the console after the code compiles:

```
dotnet run

Hello World!
```

That's all it takes to scaffold and run a .NET program! Next, you'll do the same thing for an ASP.NET Core application.

Create an ASP.NET Core project

If you're still in the directory you created for the Hello World sample, move back up to your Documents or home directory:

```
cd ..
```

Next, create a new project with `dotnet new`, this time with some extra options:

```
dotnet new mvc --auth Individual -o AspNetCoreTodo
cd AspNetCoreTodo
```

This creates a new project from the `mvc` template, and adds some additional authentication and security bits to the project. (I'll cover security in the *Security and identity* chapter.)

The `-o AspNetCoreTodo` flag tells `dotnet new` to create a new directory called `AspNetCoreTodo` for all the output files. You'll see quite a few files show up in this project directory. Once you `cd` into the new directory, all you have to do is run the project:

```
dotnet run

Now listening on: http://localhost:5000
Application started. Press Ctrl+C to shut down.
```

Instead of printing to the console and exiting, this program starts a web server and waits for requests on port 5000.

Open your web browser and navigate to `http://localhost:5000`. You'll see the default ASP.NET Core splash page, which means your project is working! When you're done, press Ctrl-C in the terminal window to stop the server.

The parts of an ASP.NET Core project

The `dotnet new mvc` template generates a number of files and directories for you. Here are the most important things you get out of the box:

- The **Program.cs** and **Startup.cs** files set up the web server and ASP.NET Core pipeline. The `Startup` class is where you can add middleware that handles and modifies incoming requests, and serves things like static content or error pages. It's also where you add your own services to the dependency injection container (more on this later).

- The **Models**, **Views**, and **Controllers** directories contain the components of the Model-View-Controller (MVC) architecture. You'll explore all three in the next chapter.
- The **wwwroot** directory contains static assets like CSS, JavaScript, and image files. By default, the bower tool is used to manage CSS and JavaScript packages, but you can use whatever package manager you prefer (npm and yarn are popular choices). Files in `wwwroot` will be served as static content, and can be bundled and minified automatically.
- The **appsettings.json** file contains configuration settings ASP.NET Core will load on startup. You can use this to store database connection strings or other things that you don't want to hard-code.

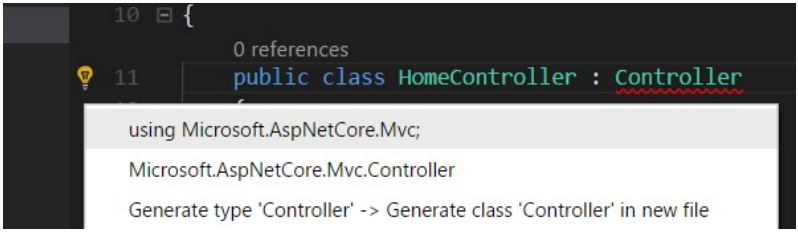
Tips for Visual Studio Code

If you're using Visual Studio Code (or Visual Studio) for the first time, here are a couple of helpful tips to get you started:

- **F5 to run (and debug breakpoints)**: With your project open, press F5 to run the project in debug mode. This is the same as `dotnet run` on the

command line, but you have the benefit of setting breakpoints in your code by clicking on the left margin:

- **Lightbulb to fix problems**: If your code contains red squiggles (compiler errors), put your cursor on the code that's red and look for the lightbulb icon on the left margin. The lightbulb menu will suggest common fixes, like adding a missing `using` statement to your code:



- **Compile quickly**: Use the shortcut `Command-Shift-B` or `Control-Shift-B` to run the Build task, which does the same thing as `dotnet build`.

A note about Git

If you use Git or GitHub to manage your source code, now is a good time to do `git init` and initialize a Git repository in the project directory. Make sure you add a `.gitignore` file that ignores the `bin` and `obj` directories. The Visual Studio template on GitHub's gitignore template repo (https://github.com/github/gitignore) works great.

There's plenty more to explore, so let's dive in and start building an application!

MVC basics

In this chapter, you'll explore the MVC system in ASP.NET Core. **MVC** (Model-View-Controller) is a pattern for building web applications that's used in almost every web framework (Ruby on Rails and Express are popular examples), as well as frontend JavaScript frameworks like Angular. Mobile apps on iOS and Android use a variation of MVC as well.

As the name suggests, MVC has three components: models, views, and controllers. **Controllers** handle incoming requests from a client or web browser and make decisions about what code to run. **Views** are templates (usually HTML plus some templating language like Handlebars, Pug, or Razor) that get data added to them and then are displayed to the user. **Models** hold the data that is added to views, or data that is entered by the user.

A common pattern for MVC code is:

- The controller receives a request and looks up some information in a database
- The controller creates a model with the information and attaches it to a view
- The view is rendered and displayed in the user's

browser
- The user clicks a button or submits a form, which sends a new request to the controller

If you've worked with MVC in other languages, you'll feel right at home in ASP.NET Core MVC. If you're new to MVC, this chapter will teach you the basics and will help get you started.

What you'll build

The "Hello World" exercise of MVC is building a to-do list application. It's a great project since it's small and simple in scope, but it touches each part of MVC and covers many of the concepts you'd use in a larger application.

In this book, you'll build a to-do app that lets the user add items to their to-do list and check them off once complete. You'll build the server (the "backend") using ASP.NET Core, C#, and the MVC pattern. You'll use HTML, CSS, and JavaScript in the views (also called the "frontend").

If you haven't already created a new ASP.NET Core project using `dotnet new mvc`, follow the steps in the previous chapter. You should be able to build and run the project and see the default welcome screen.

Create a controller

There are already a few controllers in the project's Controllers folder, including the `HomeController` that renders the default welcome screen you see when you visit `http://localhost:5000`. You can ignore these controllers for now.

Create a new controller for the to-do list functionality, called `TodoController`, and add the following code:

`Controllers/TodoController.cs`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;

namespace AspNetCoreTodo.Controllers
{
    public class TodoController : Controller
    {
        // Actions go here
    }
}
```

Routes that are handled by controllers are called **actions**, and are represented by methods in the controller class. For example, the `HomeController` includes three action methods (`Index` , `About` , and `Contact`) which are mapped by ASP.NET Core to these route URLs:

```
localhost:5000/Home         -> Index()
localhost:5000/Home/About   -> About()
localhost:5000/Home/Contact -> Contact()
```

There are a number of conventions (common patterns) used by ASP.NET Core, such as the pattern that `FooController` becomes `/Foo` , and the `Index` action name can be left out of the URL. You can customize this behavior if you'd like, but for now, we'll stick to the default conventions.

Add a new action called `Index` to the `TodoController` , replacing the `// Actions go here` comment:

```
public class TodoController : Controller
{
    public IActionResult Index()
    {
        // Get to-do items from database

        // Put items into a model

        // Render view using the model
    }
}
```

Action methods can return views, JSON data, or HTTP status codes like `200 OK` or `404 Not Found`. The `IActionResult` return type gives you the flexibility to return any of these from the action.

It's a best practice to keep controllers as lightweight as possible. In this case, the controller should only be responsible for getting the to-do items from the database, putting those items into a model the view can understand, and sending the view back to the user's browser.

Before you can write the rest of the controller code, you need to create a model and a view.

Create models

There are two separate model classes that need to be created: a model that represents a to-do item stored in the database (sometimes called an **entity**), and the model that will be combined with a view (the **MV** in MVC) and sent back to the user's browser. Because both of them can be referred to as "models", I'll refer to the latter as a *view model*.

First, create a class called `TodoItem` in the Models directory:

`Models/TodoItem.cs`

```
using System;

namespace AspNetCoreTodo.Models
{
    public class TodoItem
    {
        public Guid Id { get; set; }

        public bool IsDone { get; set; }

        public string Title { get; set; }

        public DateTimeOffset? DueAt { get; set; }
    }
}
```

This class defines what the database will need to store for each to-do item: an ID, a title or name, whether the item is complete, and what the due date is. Each line defines a property of the class:

- The **Id** property is a guid, or a **g**lobally **u**nique **id**entifier. Guids (or GUIDs) are long strings of letters and numbers, like `43ec09f2-7f70-4f4b-9559-65011d5781bb`. Because guids are random and are extremely unlikely to be accidentally duplicated, they are commonly used as unique IDs. You could also use a number (integer) as a database entity ID, but you'd need to configure your database to always

increment the number when new rows are added to the database. Guids are generated randomly, so you don't have to worry about auto-incrementing.

- The **IsDone** property is a boolean (true/false value). By default, it will be `false` for all new items. Later you'll use write code to switch this property to `true` when the user clicks the item's checkbox in the view.
- The **Title** property is a string. This will hold the name or description of the to-do item.
- The **DueAt** property is a `DateTimeOffset`, which is a C# type that stores a date/time stamp along with a timezone offset from UTC. Storing the date, time, and timezone offset together makes it easy to render dates accurately on systems in different timezones.

Notice the `?` question mark after the `DateTimeOffset` type? That marks the DueAt property as *nullable*, or optional. If the `?` wasn't included, every to-do item would need to have a due date. The Id and IsDone properties aren't marked as nullable, so they are required and will always have a value (or a default value).

> Strings in C# are always nullable, so there's no need to mark the Title property as nullable. C# strings can be null, empty, or contain text.

Each property is followed by `get; set;`, which is a shorthand way of saying the property is read/write (or more technically, it has a getter and setter methods).

At this point, it doesn't matter what the underlying database technology is. It could be SQL Server, MySQL, MongoDB, Redis, or something more exotic. This model defines what the database row or entry will look like in C# so you don't have to worry about the low-level database stuff in your code. This simple style of model is sometimes called a "plain old C# object" or POCO.

The view model

Often, the model (entity) you store in the database is similar but not *exactly* the same as the model you want to use in MVC (the view model). In this case, the `TodoItem` model represents a single item in the database, but the view might need to display two, ten, or a hundred to-do items (depending on how badly the user is procrastinating).

Because of this, the view model should be a separate class that holds an array of `TodoItem`s:

`Models/TodoViewModel.cs`

```
using System.Collections.Generic;

namespace AspNetCoreTodo.Models
{
    public class TodoViewModel
    {
        public IEnumerable<TodoItem> Items { get; set
; }
    }
}
```

`IEnumerable<>` is a fancy C# way of saying that the `Items` property contains zero, one, or many `TodoItem`s. (In technical terms, it's not quite an array, but rather an array-like interface for any sequence that can be enumerated or iterated over.)

> The `IEnumerable<>` interface exists in the `System.Collections.Generic` namespace, so you need a `using System.Collections.Generic` statement at the top of the file.

Now that you have some models, it's time to create a view that will take a `TodoViewModel` and render the right HTML to show the user their to-do list.

Create a view

Views in ASP.NET Core are built using the Razor templating language, which combines HTML and C# code. (If you've written pages using Jade/Pug or Handlebars moustaches in JavaScript, ERB in Ruby on Rails, or Thymeleaf in Java, you've already got the basic idea.)

Most view code is just HTML, with the occasional C# statement added in to pull data out of the view model and turn it into text or HTML. The C# statements are prefixed with the `@` symbol.

The view rendered by the `Index` action of the `TodoController` needs to take the data in the view model (an array of to-do items) and display it as a nice table for the user. By convention, views are placed in the `Views` directory, in a subdirectory corresponding to the controller name. The file name of the view is the name of the action with a `.cshtml` extension.

`Views/Todo/Index.cshtml`

```
@model TodoViewModel

@{
```

```
    ViewData["Title"] = "Manage your todo list";
}

<div class="panel panel-default todo-panel">
  <div class="panel-heading">@ViewData["Title"]</div>


  <table class="table table-hover">
      <thead>
          <tr>
              <td>&#x2714;</td>
              <td>Item</td>
              <td>Due</td>
          </tr>
      </thead>

      @foreach (var item in Model.Items)
      {
          <tr>
              <td>
                <input type="checkbox" name="@item.I
d" value="true" class="done-checkbox">
              </td>
              <td>@item.Title</td>
              <td>@item.DueAt</td>
          </tr>
      }
  </table>

  <div class="panel-footer add-item-form">
    <form>
        <div id="add-item-error" class="text-danger">
</div>
        <label for="add-item-title">Add a new item:</
label>
```

```
        <input id="add-item-title">
        <button type="button" id="add-item-button">A
dd</button>
    </form>
  </div>
</div>
```

At the very top of the file, the `@model` directive tells Razor which model to expect this view to be bound to. The model is accessed through the `Model` property.

Assuming there are any to-do items in `Model.Items`, the `foreach` statement will loop over each to-do item and render a table row (`<tr>` element) containing the item's name and due date. A checkbox is also rendered that contains the item's ID, which you'll use later to mark the item as completed.

The layout file

You might be wondering where the rest of the HTML is: what about the `<body>` tag, or the header and footer of the page? ASP.NET Core uses a layout view that defines the base structure that the rest of the views are rendered inside of. It's stored in `Views/Shared/_Layout.cshtml`.

The default ASP.NET Core template includes Bootstrap and jQuery in this layout file, so you can quickly create a web application. Of course, you can use your own CSS and JavaScript libraries if you'd like.

Customizing the stylesheet

For now, just add these CSS style rules to the bottom of the `site.css` file:

`wwwroot/css/site.css`

```
div.todo-panel {
  margin-top: 15px;
}

table tr.done {
  text-decoration: line-through;
  color: #888;
}
```

You can use CSS rules like these to completely customize how your pages look and feel.

ASP.NET Core and Razor can do much more, such as partial views and server-rendered view components, but a simple layout and view is all you need for now. The official ASP.NET Core documentation (at `https://docs.asp.net`) contains a number of examples if you'd like to learn more.

Add a service class

You've created a model, a view, and a controller. Before you use the model and view in the controller, you also need to write code that will get the user's to-do items from a database.

You could write this database code directly in the controller, but it's a better practice to keep all the database code in a separate class called a **service**. This helps keep the controller as simple as possible, and makes it easier to test and change the database code later.

> Separating your application logic into one layer that handles database access and another layer that handles presenting a view is sometimes called a layered, 3-tier, or n-tier architecture.

.NET and C# include the concept of **interfaces**, where the definition of an object's methods and properties is separate from the class that actually contains the code for those methods and properties. Interfaces make it easy to keep your classes decoupled and easy to test, as you'll see here (and later in the *Automated testing* chapter).

First, create an interface that will represent the service that can interact with to-do items in the database. By convention, interfaces are prefixed with "I". Create a new file in the Services directory:

Services/ITodoItemService.cs

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
using AspNetCoreTodo.Models;

namespace AspNetCoreTodo.Services
{
    public interface ITodoItemService
    {
        Task<IEnumerable<TodoItem>> GetIncompleteIte
msAsync();
    }
}
```

Note that the namespace of this file is `AspNetCoreTodo.Services`. Namespaces are a way to organize .NET code files, and it's customary for the namespace to follow the directory the file is stored in (`AspNetCoreTodo.Services` for files in the `Services` directory, and so on).

Because this file (in the `AspNetCoreTodo.Services` namespace) references the `TodoItem` class (in the `AspNetCoreTodo.Models` namespace), it needs to include a `using` statement at the top of the file to import that namespace. Without the `using` statement, you'll see an error like:

```
The type or namespace name 'TodoItem' could not be f
ound (are you missing a using directive or an assemb
ly reference?)
```

Since this is an interface, there isn't any actual code here, just the definition (or **method signature**) of the `GetIncompleteItemsAsync` method. This method requires no parameters and returns a `Task<IEnumerable<TodoItem>>`.

> If this syntax looks confusing, think: "a Task that contains a list of TodoItems".

The `Task` type is similar to a future or a promise, and it's used here because this method will be **asynchronous**. In other words, the method may not be able to return the list of to-do items right away because it needs to go talk to the database first. (More on this later.)

Now that the interface is defined, you're ready to create the actual service class. I'll cover database code in depth in the *Use a database* chapter, but for now you'll just fake it and return hard-coded values:

Services/FakeTodoItemService.cs

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
using AspNetCoreTodo.Models;

namespace AspNetCoreTodo.Services
{
    public class FakeTodoItemService : ITodoItemServ
ice
    {
        public Task<IEnumerable<TodoItem>> GetIncomp
leteItemsAsync()
        {
            // Return an array of TodoItems
            IEnumerable<TodoItem> items = new[]
            {
                new TodoItem
                {
                    Title = "Learn ASP.NET Core",
                    DueAt = DateTimeOffset.Now.AddDa
ys(1)
                },
                new TodoItem
                {
                    Title = "Build awesome apps",
                    DueAt = DateTimeOffset.Now.AddDa
ys(2)
```

```
                }
            };

            return Task.FromResult(items);
        }
    }
}
```

This `FakeTodoItemService` implements the `ITodoItemService` interface but always returns the same array of two `TodoItem`s. You'll use this to test the controller and view, and then add real database code in *Use a database*.

Use dependency injection

Back in the `TodoController`, add some code to work with the `ITodoItemService`:

```
public class TodoController : Controller
{
    private readonly ITodoItemService _todoItemServi
ce;

    public TodoController(ITodoItemService todoItemS
ervice)
    {
        _todoItemService = todoItemService;
    }

    public IActionResult Index()
    {
        // Get to-do items from database

        // Put items into a model

        // Pass the view to a model and render
    }
}
```

Since `ITodoItemService` is in the `Services` namespace, you'll also need to add a `using` statement at the top:

```
using AspNetCoreTodo.Services;
```

The first line of the class declares a private variable to hold a reference to the `ITodoItemService`. This variable lets you use the service from the `Index` action method later (you'll see how in a minute).

The `public TodoController(ITodoItemService todoItemService)` line defines a **constructor** for the class. The constructor is a special method that is called when you want to create a new instance of a class (the `TodoController` class, in this case). By adding an `ITodoItemService` parameter to the constructor, you've declared that in order to create the `TodoController`, you'll need to provide an object that matches the `ITodoItemService` interface.

> Interfaces are awesome because they help decouple (separate) the logic of your application. Since the controller depends on the `ITodoItemService` interface, and not on any *specific* service class, it doesn't know or care which class it's actually given. It could be the `FakeTodoItemService`, a different one that talks to a live database, or something else! As long as it matches the interface, the controller doesn't care. This makes it really easy to test parts of your application separately. (I'll cover testing more in the *Automated testing* chapter.)

Now you can finally use the `ITodoItemService` (via the private variable you declared) in your action method to get to-do items from the service layer:

```
public IActionResult Index()
{
    var todoItems = await _todoItemService.GetIncomp
leteItemsAsync();

    // ...
}
```

Remember that the `GetIncompleteItemsAsync` method returned a `Task<IEnumerable<TodoItem>>`? Returning a `Task` means that the method won't necessarily have a

result right away, but you can use the `await` keyword to make sure your code waits until the result is ready before continuing on.

The `Task` pattern is common when your code calls out to a database or an API service, because it won't be able to return a real result until the database (or network) responds. If you've used promises or callbacks in JavaScript or other languages, `Task` is the same idea: the promise that there will be a result - sometime in the future.

> If you've had to deal with "callback hell" in older JavaScript code, you're in luck. Dealing with asynchronous code in .NET is much easier thanks to the magic of the `await` keyword! `await` lets your code pause on an async operation, and then pick up where it left off when the underlying database or network request finishes. In the meantime, your application isn't blocked, because it can process other requests as needed. This pattern is simple but takes a little getting used to, so don't worry if this doesn't make sense right away. Just keep following along!

The only catch is that you need to update the `Index` method signature to return a `Task<IActionResult>` instead of just `IActionResult`, and mark it as `async`:

```
public async Task<IActionResult> Index()
{
    var todoItems = await _todoItemService.GetIncomp
leteItemsAsync();

    // Put items into a model

    // Pass the view to a model and render
}
```

You're almost there! You've made the `TodoController` depend on the `ITodoItemService` interface, but you haven't yet told ASP.NET Core that you want the `FakeTodoItemService` to be the actual service that's used under the hood. It might seem obvious right now since you only have one class that implements `ITodoItemService`, but later you'll have multiple classes that implement the same interface, so being explicit is necessary.

Declaring (or "wiring up") which concrete class to use for each interface is done in the `ConfigureServices` method of the `Startup` class. Right now, it looks something like this:

Startup.cs

```
public void ConfigureServices(IServiceCollection ser
vices)
{
    // (... some code)

    services.AddMvc();
}
```

The job of the `ConfigureServices` method is adding things to the **service container**, or the collection of services that ASP.NET Core knows about. The `services.AddMvc` line adds the services that the internal ASP.NET Core systems need (as an experiment, try commenting out this line). Any other services you want to use in your application must be added to the service container here in `ConfigureServices`.

Add the following line anywhere inside the `ConfigureServices` method:

```
services.AddSingleton<ITodoItemService, FakeTodoItem
Service>();
```

This line tells ASP.NET Core to use the `FakeTodoItemService` whenever the `ITodoItemService` interface is requested in a constructor (or anywhere else).

`AddSingleton` adds your service to the service container as a **singleton**. This means that only one copy of the `FakeTodoItemService` is created, and it's reused whenever the service is requested. Later, when you write a different service class that talks to a database, you'll use a different approach (called **scoped**) instead. I'll explain why in the *Use a database* chapter.

That's it! When a request comes in and is routed to the `TodoController`, ASP.NET Core will look at the available services and automatically supply the `FakeTodoItemService` when the controller asks for an `ITodoItemService`. Because the services the controller depends on are "injected" from the service container, this pattern is called **dependency injection**.

Finish the controller

The last step is to finish the controller code. The controller now has a list of to-do items from the service layer, and it needs to put those items into a `TodoViewModel` and bind that model to the view you created earlier:

`Controllers/TodoController.cs`

```
public async Task<IActionResult> Index()
{
    var todoItems = await _todoItemService.GetIncomp
leteItemsAsync();

    var model = new TodoViewModel()
    {
        Items = todoItems
    };

    return View(model);
}
```

If you haven't already, make sure these `using` statements are at the top of the file:

```
using AspNetCoreTodo.Services;
using AspNetCoreTodo.Models;
```

If you're using Visual Studio or Visual Studio Code, the editor will suggest these `using` statements when you put your cursor on a red squiggly line.

Test it out

To start the application, press F5 (if you're using Visual Studio or Visual Studio Code), or just run `dotnet run` in the terminal. If the code compiles without errors, the server will spin up on port 5000 by default.

If your web browser didn't open automatically, open it and navigate to http://localhost:5000/todo. You'll see the view you created, with the data pulled from your fake database layer (for now).

Congratulations! You've built a working ASP.NET Core application. Next, you'll take it further with third-party packages and real database code.

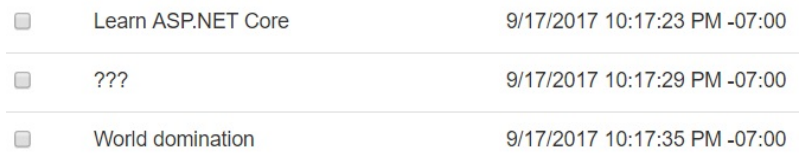
Add external packages

One of the big advantages of using a mature stack like .NET is that the ecosystem of third-party packages and plugins is huge. Just like other package systems (npm, Maven, RubyGems), you can download and install .NET packages that help with almost any task or problem you can imagine.

NuGet is both the package manager tool and the official package repository (at https://www.nuget.org). You can search for NuGet packages on the web, and install them from your local machine through the terminal (or the GUI, if you're using Visual Studio).

Install the Humanizer package

At the end of the last chapter, the to-do application displayed to-do items like this:



The due date column is displaying dates in a format that's good for machines (called ISO 8601), but clunky for humans. Wouldn't it be nicer if it simply read "X days from now"? You could write code that converted a date into a human-friendly string, but fortunately, there's a faster way.

The Humanizer package on NuGet (https://www.nuget.org/packages/Humanizer) solves this problem by providing methods that can "humanize" or rewrite almost anything: dates, times, durations, numbers, and so on. It's a fantastic and useful open-source project that's published under the permissive MIT license.

To add it to your project, run this command in the terminal:

```
dotnet add package Humanizer
```

If you peek at the `AspNetCoreTodo.csproj` project file, you'll see a new `PackageReference` line that references `Humanizer`.

Use Humanizer in the view

To use a package in your code, you usually need to add a `using` statement that imports the package at the top of the file.

Since Humanizer will be used to rewrite dates rendered in the view, you can use it directly in the view itself. First, add a `@using` statement at the top of the view:

Views/Todo/Index.cshtml

```
@model TodoViewModel
@using Humanizer

// ...
```

Then, update the line that writes the `DueAt` property to use Humanizer's `Humanize` method:

```
<td>@item.DueAt.Humanize()</td>
```

Now the dates are much more readable:

☐	Learn ASP.NET Core	2 days from now
☐	???	2 days from now
☐	World domination	2 days from now

There are packages available on NuGet for everything from parsing XML to machine learning to posting to Twitter. ASP.NET Core itself, under the hood, is nothing more than a collection of NuGet packages that are added to your project.

> The project file created by `dotnet new mvc` includes a single reference to the `Microsoft.AspNetCore.All` package, which is a convenient "metapackage" that references all of the other ASP.NET Core packages you need for a typical project. That way, you don't need to have hundreds of package references in your project file.

In the next chapter, you'll use another set of NuGet packages (a system called Entity Framework Core) to write code that interacts with a database.

Use a database

Writing database code can be tricky. Unless you really know what you're doing, it's a bad idea to paste raw SQL query strings into your application code. An **object-relational mapper** (ORM) makes it easier to write code that interacts with a database by adding a layer of abstraction between your code and the database itself. Hibernate in Java and ActiveRecord in Ruby are two well-known ORMs.

There are a number of ORMs for .NET, including one built by Microsoft and included in ASP.NET Core by default: Entity Framework Core. Entity Framework Core makes it easy to connect to a number of different database types, and lets you use C# code to create database queries that are mapped back into C# models (POCOs).

> Remember how creating a service interface decoupled the controller code from the actual service class? Entity Framework Core is like a big interface over your database, and you can swap out different providers depending on the underlying database technology.

Entity Framework Core can connect to SQL database like SQL Server and MySQL, and also works with NoSQL (document) databases like Mongo. You'll use a SQLite database for this project, but you can plug in a different database provider if you'd like.

Connect to a database

There are a few things you need to use Entity Framework Core to connect to a database. Since you used `dotnet new` and the MVC + Individual Auth template to set your project, you've already got them:

- **The Entity Framework Core packages**. These are included by default in all ASP.NET Core projects.
- **A database** (naturally). The `app.db` file in the project root directory is a small SQLite database created for you by `dotnet new`. SQLite is a lightweight database engine that can run without requiring you to install any extra tools on your machine, so it's easy and quick to use in development.
- **A database context class**. The database context is a C# class that provides an entry point into the database. It's how your code will interact with the database to read and save items. A basic context class already exists in the `Data/ApplicationDbContext.cs` file.
- **A connection string**. Whether you are connecting to a local file database (like SQLite) or a database hosted elsewhere, you'll define a string that contains

the name or address of the database to connect to. This is already set up for you in the `appsettings.json` file: the connection string for the SQLite database is `DataSource=app.db`.

Entity Framework Core uses the database context, together with the connection string, to establish a connection to the database. You need to tell Entity Framework Core which context, connection string, and database provider to use in the `ConfigureServices` method of the `Startup` class. Here's what's defined for you, thanks to the template:

```
services.AddDbContext<ApplicationDbContext>(options
=>
    options.UseSqlite(Configuration.GetConnectionStr
ing("DefaultConnection")));
```

This code adds the `ApplicationDbContext` to the service container, and tells Entity Framework Core to use the SQLite database provider, with the connection string from configuration (`appsettings.json`).

As you can see, `dotnet new` creates a lot of stuff for you! The database is set up and ready to be used. However, it doesn't have any tables for storing to-do items. In order to store your `TodoItem` entities, you'll need to update the context and migrate the database.

Update the context

There's not a whole lot going on in the database context yet:

Data/ApplicationDbContext.cs

```
public class ApplicationDbContext : IdentityDbContex
t<ApplicationUser>
{
    public ApplicationDbContext(DbContextOptions<App
licationDbContext> options)
        : base(options)
    {
    }

    protected override void OnModelCreating(ModelBui
lder builder)
    {
        base.OnModelCreating(builder);
        // Customize the ASP.NET Identity model and
override the defaults if needed.
        // For example, you can rename the ASP.NET I
dentity table names and more.
        // Add your customizations after calling bas
e.OnModelCreating(builder);
    }
}
```

Add a `DbSet` property to the `ApplicationDbContext`, right below the constructor:

```
public ApplicationDbContext(DbContextOptions<Applica
tionDbContext> options)
    : base(options)
{
}

public DbSet<TodoItem> Items { get; set; }

// ...
```

A `DbSet` represents a table or collection in the database. By creating a `DbSet<TodoItem>` property called `Items`, you're telling Entity Framework Core that you want to store `TodoItem` entities in a table called `Items`.

You've updated the context class, but now there's one small problem: the context and database are now out of sync, because there isn't actually an `Items` table in the database. (Just updating the code of the context class doesn't change the database itself.)

In order to update the database to reflect the change you just made to the context, you need to create a **migration**.

> If you already have an existing database, search the web for "scaffold-dbcontext existing database" and read Microsoft's documentation on using the `Scaffold-DbContext` tool to reverse-engineer your database structure into the proper `DbContext` and model classes automatically.

Create a migration

Migrations keep track of changes to the database structure over time. They make it possible to undo (roll back) a set of changes, or create a second database with the same structure as the first. With migrations, you have a full history of modifications like adding or removing columns (and entire tables).

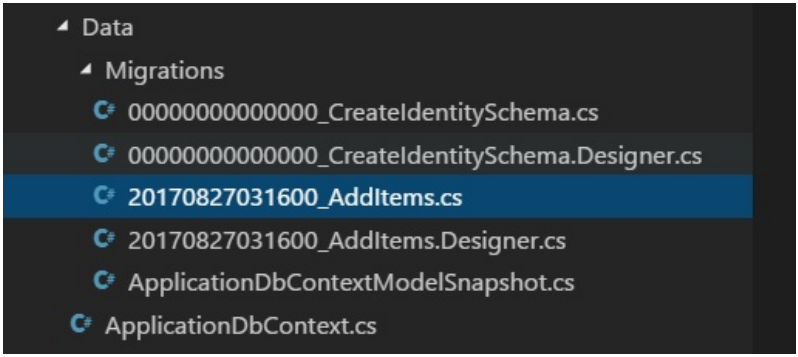
In the previous chapter, you added an `Items` set to the context. Since the context now includes a set (or table) that doesn't exist in the database, you need to create a migration to update the database:

```
dotnet ef migrations add AddItems
```

This creates a new migration called `AddItems` by examining any changes you've made to the context.

> If you get an error like `No executable found matching command "dotnet-ef"`, make sure you're in the right directory. These commands must be run from the project root directory (where the `Program.cs` file is).

If you open up the `Data/Migrations` directory, you'll see a few files:



The first migration file (with a name like `00_CreateIdentitySchema.cs`) was created and applied for you way back when you ran `dotnet new`. Your new `AddItem` migration is prefixed with a timestamp when you create it.

> You can see a list of migrations with `dotnet ef migrations list`.

If you open your migration file, you'll see two methods called `Up` and `Down`:

`Data/Migrations/<date>_AddItems.cs`

```
protected override void Up(MigrationBuilder migratio
nBuilder)
{
    // (... some code)

    migrationBuilder.CreateTable(
        name: "Items",
        columns: table => new
```

```
        {
            Id = table.Column<Guid>(type: "BLOB", nu
llable: false),
            DueAt = table.Column<DateTimeOffset>(typ
e: "TEXT", nullable: true),
            IsDone = table.Column<bool>(type: "INTEG
ER", nullable: false),
            Title = table.Column<string>(type: "TEXT"
, nullable: true)
        },
        constraints: table =>
        {
            table.PrimaryKey("PK_Items", x => x.Id);
        });

    // (some code...)
}

protected override void Down(MigrationBuilder migrat
ionBuilder)
{
    // (... some code)

    migrationBuilder.DropTable(
        name: "Items");

    // (some code...)
}
```

The `Up` method runs when you apply the migration to the database. Since you added a `DbSet<TodoItem>` to the database context, Entity Framework Core will create an

`Items` table (with columns that match a `TodoItem`) when you apply the migration.

The `Down` method does the opposite: if you need to undo (roll back) the migration, the `Items` table will be dropped.

Workaround for SQLite limitations

There are some limitations of SQLite that get in the way if you try to run the migration as-is. Until this problem is fixed, use this workaround:

- Comment out the `migrationBuilder.AddForeignKey` lines in the `Up` method.
- Comment out any `migrationBuilder.DropForeignKey` lines in the `Down` method.

If you use a full-fledged SQL database, like SQL Server or MySQL, this won't be an issue and you won't need to do this (admittedly hackish) workaround.

Apply the migration

The final step after creating one (or more) migrations is to actually apply them to the database:

```
dotnet ef database update
```

This command will cause Entity Framework Core to create the `Items` table in the database.

> If you want to roll back the database, you can provide the name of the *previous* migration: `dotnet ef database update CreateIdentitySchema` This will run the `Down` methods of any migrations newer than the migration you specify.
>
> If you need to completely erase the database and start over, run `dotnet ef database drop` followed by `dotnet ef database update` to re-scaffold the database and bring it up to the current migration.

That's it! Both the database and the context are ready to go. Next, you'll use the context in your service layer.

Create a new service class

Back in the *MVC basics* chapter, you created a `FakeTodoItemService` that contained hard-coded to-do items. Now that you have a database context, you can create a new service class that will use Entity Framework Core to get the real items from the database.

Delete the `FakeTodoItemService.cs` file, and create a new file:

`Services/TodoItemService.cs`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using AspNetCoreTodo.Data;
using AspNetCoreTodo.Models;
using Microsoft.EntityFrameworkCore;

namespace AspNetCoreTodo.Services
{
    public class TodoItemService : ITodoItemService
    {
        private readonly ApplicationDbContext _conte
xt;

        public TodoItemService(ApplicationDbContext
context)
        {
            _context = context;
        }

        public async Task<IEnumerable<TodoItem>> Get
IncompleteItemsAsync()
        {
            var items = await _context.Items
                .Where(x => x.IsDone == false)
                .ToArrayAsync();

            return items;
        }
    }
}
```

You'll notice the same dependency injection pattern here that you saw in the MVC basics chapter, except this time it's the `ApplicationDbContext` that gets injected into the service. The `ApplicationDbContext` is already being added to the service container in the `ConfigureServices` method, so it's available for injection here.

Let's take a closer look at the code of the `GetIncompleteItemsAsync` method. First, it uses the `Items` property of the context to access all the to-do items in the `DbSet`:

```
var items = await _context.Items
```

Then, the `Where` method is used to filter only the items that are not complete:

```
.Where(x => x.IsDone == false)
```

The `Where` method is a feature of C# called LINQ (**l**anguage **in**tegrated **q**uery), which takes cues from functional programming and makes it easy to express database queries in code. Under the hood, Entity Framework Core translates the method into a statement like `SELECT * FROM Items WHERE IsDone = 0`, or an equivalent query document in a NoSQL database.

Finally, the `ToArrayAsync` method tells Entity Framework Core to get all the entities that matched the filter and return them as an array. The `ToArrayAsync` method is asynchronous (it returns a `Task`), so it must be `await`ed to get its value.

To make the method a little shorter, you can remove the intermediate `items` variable and just return the result of the query directly (which does the same thing):

```
public async Task<IEnumerable<TodoItem>> GetIncomple
teItemsAsync()
{
    return await _context.Items
        .Where(x => x.IsDone == false)
        .ToArrayAsync();
}
```

Update the service container

Because you deleted the `FakeTodoItemService` class, you'll need to update the line in `ConfigureServices` that is wiring up the `ITodoItemService` interface:

```
services.AddScoped<ITodoItemService, TodoItemService
>();
```

`AddScoped` adds your service to the service container using the **scoped** lifecycle. This means that a new instance of the `TodoItemService` class will be created during each web request. This is required for service classes that interact with a database.

> Adding a service class that interacts with Entity Framework Core (and your database) with the singleton lifecycle (or other lifecycles) can cause problems, because of how Entity Framework Core manages database connections per request under the hood. To avoid that, always use the scoped lifecycle for services that interact with Entity Framework Core.

The `TodoController` that depends on `ITodoItemService` will be blissfully unaware of the change, but under the hood you'll be using Entity Framework Core and talking to a real database!

Test it out

Start up the application and navigate to `http://localhost:5000/todo`. The fake items are gone, and your application is making real queries to the database. There just doesn't happen to be any saved to-do items!

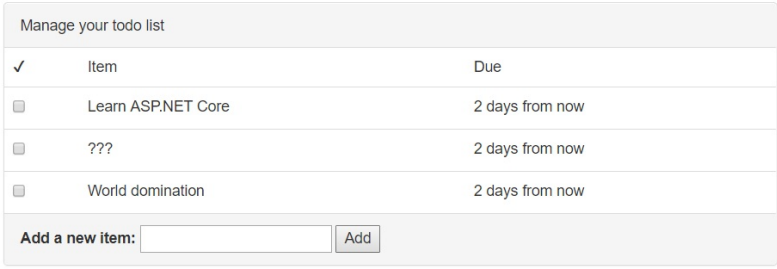
In the next chapter, you'll add more features to the application, starting with the ability to create new to-do items.

Add more features

Now that you've connected to a database using Entity Framework Core, you're ready to add some more features to the application. First, you'll make it possible to mark a to-do item as complete by checking its checkbox.

Add new to-do items

The user will add new to-do items with a simple form below the list:



Adding this feature requires a few steps:

- Adding JavaScript that will send the data to the server
- Creating a new action on the controller to handle this request
- Adding code to the service layer to update the database

Add JavaScript code

The `Todo/Index.cshtml` view already includes an HTML form that has a textbox and a button for adding a new item. You'll use jQuery to send a POST request to the

server when the Add button is clicked.

Open the `wwwroot/js/site.js` file and add this code:

```
$(document).ready(function() {

    // Wire up the Add button to send the new item t
o the server
    $('#add-item-button').on('click', addItem);

});
```

Then, write the `addItem` function at the bottom of the file:

```
function addItem() {
    $('#add-item-error').hide();
    var newTitle = $('#add-item-title').val();

    $.post('/Todo/AddItem', { title: newTitle }, fun
ction() {
        window.location = '/Todo';
    })
    .fail(function(data) {
        if (data && data.responseJSON) {
            var firstError = data.responseJSON[Object
.keys(data.responseJSON)[0]];
            $('#add-item-error').text(firstError);
            $('#add-item-error').show();
        }
    });
}
```

This function will send a POST request to `http://localhost:5000/Todo/AddItem` with the name the user typed. The third parameter passed to the `$.post` method (the function) is a success handler that will run if the server responds with `200 OK`. The success handler function uses `window.location` to refresh the page (by setting the location to `/Todo`, the same page the browser is currently on). If the server responds with `400 Bad Request`, the `fail` handler attached to the `$.post` method will try to pull out an error message and display it in a the `<div>` with id `add-item-error`.

Add an action

The above JavaScript code won't work yet, because there isn't any action that can handle the `/Todo/AddItem` route. If you try it now, ASP.NET Core will return a `404 Not Found` error.

You'll need to create a new action called `AddItem` on the `TodoController`:

```
public async Task<IActionResult> AddItem(NewTodoItem
 newItem)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }

    var successful = await _todoItemService.AddItemA
sync(newItem);
    if (!successful)
    {
        return BadRequest(new { error = "Could not a
dd item" });
    }

    return Ok();
}
```

The method signature defines a `NewTodoItem` parameter, which is a new model that doesn't exist yet. You'll need to create it:

`Models/NewTodoItem.cs`

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

namespace AspNetCoreTodo.Models
{
    public class NewTodoItem
    {
        [Required]
        public string Title { get; set; }
    }
}
```

This model definition (one property called `Title`) matches the data you're sending to the action with jQuery:

```
$.post('/Todo/AddItem', { title: newTitle }  // ...

// A JSON object with one property:
// {
//   title: (whatever the user typed)
// }
```

ASP.NET Core uses a process called **model binding** to match up the parameters submitted in the POST request to the model definition you created. If the parameter names match (ignoring things like case), the request data will be placed into the model.

After binding the request data to the model, ASP.NET Core also performs **model validation**. The `[Required]` attribute on the `Title` property informs the validator that the `Title` property should not be missing (blank). The validator won't throw an error if the model fails validation, but the validation status will be saved so you can check it in the controller.

> Sidebar: It would have been possible to reuse the `TodoItem` model instead of creating the `NewTodoItem` model, but `TodoItem` contains properties that will never be submitted by the user (ID and done). It's cleaner to declare a new model that represents the exact set of properties that are relevant when adding a new item.

Back to the `AddItem` action method on the `TodoController`: the first block checks whether the model passed the model validation process. It's customary to do this right at the beginning of the method:

```
if (!ModelState.IsValid)
{
    return BadRequest(ModelState);
}
```

If the `ModelState` is invalid (because the required property is empty), the action will return 400 Bad Request along with the model state, which is automatically converted into an error message that tells the user what is wrong.

Next, the controller calls into the service layer to do the actual database operation:

```
var successful = await _todoItemService.AddItemAsync
(newItem);
if (!successful)
{
    return BadRequest(new { error = "Could not add i
tem." });
}
```

The `AddItemAsync` method will return `true` or `false` depending on whether the item was successfully added to the database. If it fails for some reason, the action will return `400 Bad Request` along with an object that contains an `error` property.

Finally, if everything completed without errors, the action returns `200 OK`.

Add a service method

If you're using a code editor that understands C#, you'll see red squiggely lines under `AddItemAsync` because the method doesn't exist yet.

As a last step, you need to add a method to the service layer. First, add it to the interface definition in `ITodoItemService`:

```
public interface ITodoItemService
{
    Task<IEnumerable<TodoItem>> GetIncompleteItemsAs
ync();

    Task<bool> AddItemAsync(NewTodoItem newItem);
}
```

Then, the actual implementation in `TodoItemService`:

```
public async Task<bool> AddItemAsync(NewTodoItem new
Item)
{
    var entity = new TodoItem
    {
        Id = Guid.NewGuid(),
        IsDone = false,
        Title = newItem.Title,
        DueAt = DateTimeOffset.Now.AddDays(3)
    };

    _context.Items.Add(entity);

    var saveResult = await _context.SaveChangesAsync
();
    return saveResult == 1;
}
```

This method creates a new `TodoItem` (the model that represents the database entity) and copies the `Title` from the `NewTodoItem` model. Then, it adds it to the context and uses `SaveChangesAsync` to persist the entity in the database.

> Sidebar: The above is just one way to build this functionality. If you want to display a separate page for adding a new item (for a complicated entity that contains a lot of properties, for example), you could create a new view that's bound to the model you need the user to provide values for. ASP.NET Core can render a form automatically for the properties of the model using a feature called **tag helpers**. You can find examples in the ASP.NET Core documentation at https://docs.asp.net.

Try it out

Run the application and add some items to your to-do list with the form. Since the items are being stored in the database, they'll still be there even after you stop and start the application again.

> As a further challenge, try adding a date picker using HTML and JavaScript, and let the user choose an (optional) date for the `DueAt` property. Then, use that date instead of always making new tasks that are due in 3 days.

Complete items with a checkbox

Adding items to your to-do list is great, but eventually you'll need to get things done, too. In the `Views/Todo/Index.cshtml` view, a checkbox is rendered for each to-do item:

```
<input type="checkbox" name="@item.Id" value="true"
class="done-checkbox">
```

The item's ID (a guid) is saved in the `name` attribute of the element. You can use this ID to tell your ASP.NET Core code to update that entity in the database when the checkbox is checked.

This is what the whole flow will look like:

- The user checks the box, which triggers a JavaScript function
- JavaScript is used to make an API call to an action on the controller
- The action calls into the service layer to update the item in the database
- A response is sent back to the JavaScript function to

indicate the update was successful
- The HTML on the page is updated

Add JavaScript code

First, open `site.js` and add this code to the `$(document).ready` block:

wwwroot/js/site.js

```
$(document).ready(function() {

    // ...

    // Wire up all of the checkboxes to run markComp
leted()
    $('.done-checkbox').on('click', function(e) {
        markCompleted(e.target);
    });

});
```

Then, add the `markCompleted` function at the bottom of the file:

```
function markCompleted(checkbox) {
    checkbox.disabled = true;

    $.post('/Todo/MarkDone', { id: checkbox.name },
function() {
        var row = checkbox.parentElement.parentEleme
nt;
        $(row).addClass('done');
    });
}
```

This code uses jQuery to send an HTTP POST request to `http://localhost:5000/Todo/MarkDone`. Included in the request will be one parameter, `id`, containing the item's ID (pulled from the `name` attribute).

If you open the Network Tools in your web browser and click on a checkbox, you'll see a request like:

```
POST http://localhost:5000/Todo/MarkDone
Content-Type: application/x-www-form-urlencoded

id=<some guid>
```

The success handler function passed to `$.post` uses jQuery to add a class to the table row that the checkbox sits in. With the row marked with the `done` class, a CSS rule in the page stylesheet will change the way the row looks.

Add an action to the controller

As you've probably guessed, you need to add a `MarkDone` action on the `TodoController`:

```
public async Task<IActionResult> MarkDone(Guid id)
{
    if (id == Guid.Empty) return BadRequest();

    var successful = await _todoItemService.MarkDone
Async(id);

    if (!successful) return BadRequest();

    return Ok();
}
```

Let's step through each piece of this action method. First, the method accepts a `Guid` parameter called `id` in the method signature. Unlike the `AddItem` action, which used a model (the `NewTodoItem` model) and model binding/validation, the `id` parameter is very simple. If the incoming request includes a parameter called `id`, ASP.NET Core will try to parse it as a guid.

There's no `ModelState` to check for validity, but you can still check to make sure the guid was valid. If for some reason the `id` parameter in the request was missing

couldn't be parsed as a guid, it will have a value of `Guid.Empty`. If that's the case, the action can return early:

```
if (id == Guid.Empty) return BadRequest();
```

The `BadRequest()` method is a helper method that simply returns the HTTP status code `400 Bad Request`.

Next, the controller needs to call down into the service to update the database. This will be handled by a new method called `MarkDoneAsync` on the `ITodoItemService`, which will return true or false depending on if the update succeeded:

```
var successful = await _todoItemService.MarkDoneAsyn
c(id);
if (!successful) return BadRequest();
```

Finally, if everything looks good, the `Ok()` method is used to return status code `200 OK`. More complex APIs might return JSON or other data as well, but for now returning a status code is all you need.

Add a service method

First, add `MarkDoneAsync` to the interface definition:

`Services/ITodoItemService.cs`

```
Task<bool> MarkDoneAsync(Guid id);
```

Then, add the concrete implementation to the `TodoItemService`:

`Services/TodoItemService.cs`

```
public async Task<bool> MarkDoneAsync(Guid id)
{
    var item = await _context.Items
        .Where(x => x.Id == id)
        .SingleOrDefaultAsync();

    if (item == null) return false;

    item.IsDone = true;

    var saveResult = await _context.SaveChangesAsync
();
    return saveResult == 1; // One entity should hav
e been updated
}
```

This method uses Entity Framework Core and `Where` to find an entity by ID in the database. The `SingleOrDefaultAsync` method will return either the item (if it exists) or `null` if the ID was bogus. If it didn't exist, the code can return early.

Once you're sure that `item` isn't null, it's a simple matter of setting the `IsDone` property:

```
item.IsDone = true;
```

Changing the property only affects the local copy of the item until `SaveChangesAsync` is called to persist your changes back to the database. `SaveChangesAsync` returns an integer that reflects how many entities were updated during the save operation. In this case, it'll either be 1 (the item was updated) or 0 (something went wrong).

Try it out

Run the application and try checking some items off the list. Refresh the page and they'll disappear completely, because of the `Where` filter in the `GetIncompleteItemsAsync` method.

Right now, the application contains a single, shared to-do list. It'd be even more useful if it kept track of individual to-do lists for each user. In the next chapter, you'll use ASP.NET Core Identity to add security and authentication features to the project.

Security and identity

Security is a major concern of any modern web application or API. It's important to keep your user or customer data safe and out of the hands of attackers. This encompasses things like

- Sanitizing data input to prevent SQL injection attacks
- Preventing cross-domain (XSRF) attacks in forms
- Using HTTPS (TLS) so data can't be intercepted as it travels over the Internet
- Giving users a way to securely sign in with a password or social login credentials
- Designing password reset or multi-factor authentication flows with security in mind

ASP.NET Core can help make all of this easier to implement. The first two (protection against SQL injection and cross-domain attacks) are already built-in, and you can add a few lines of code to enable HTTPS support. This chapter will mainly focus on the **identity** aspects of security: handling user accounts (registration, login), authenticating (logging in) your users securely, and making authorization decisions once they are authenticated.

> Authentication and authorization are distinct ideas that are often confused. **Authentication** deals with whether a user is logged in, while **authorization** deals with what they are allowed to do *after* they log in. You can think of authentication as asking the question, "Do I know who this user is?" While authorization asks, "Does this user have permission to do X?"

The MVC + Individual Authentication template you used to scaffold the project includes a number of classes built on top of ASP.NET Core Identity, an authentication and identity system that's part of ASP.NET Core.

What is ASP.NET Core Identity?

ASP.NET Core Identity is the identity system that ships with ASP.NET Core. Like everything else in the ASP.NET Core ecosystem, it's a set of NuGet packages that can be installed in any project (and are already included if you use the default template).

ASP.NET Core Identity takes care of storing user accounts, hashing and storing passwords, and managing roles for users. It supports email/password login, multi-

factor authentication, social login with providers like Google and Facebook, as well as connecting to other services using protocols like OAuth 2.0 and OpenID Connect.

The Register and Login views that ship with the MVC + Individual Auth template already take advantage of ASP.NET Core Identity, and they already work! Try registering for an account and logging in.

Add Facebook login

Out of the box, the Individual Auth template includes functionality for registering using an email and password. You can extend this by plugging in additional identity providers like Google and Facebook.

For any external provider, you typically need to do two things:

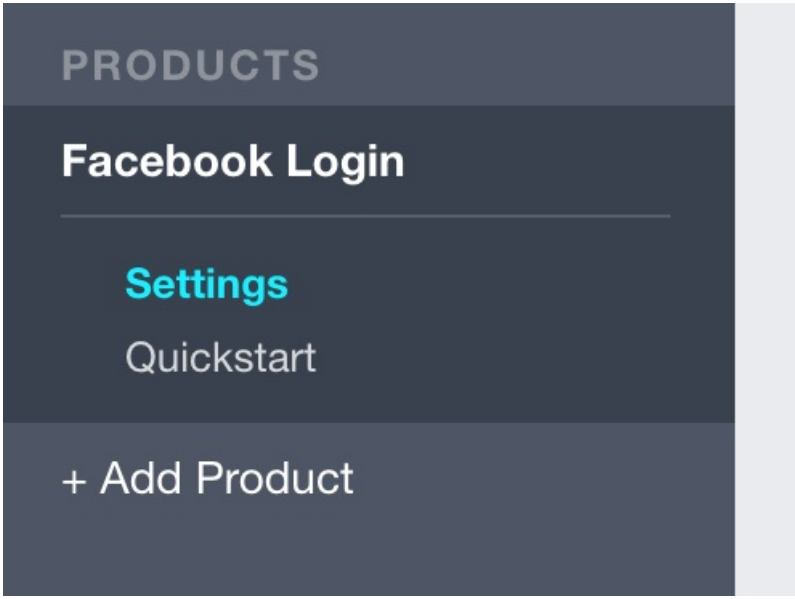
1. Create an app (sometimes called a *client*) on the external provider that represents your application
2. Copy the ID and secret generated by the provider and put them in your code

Create an app in Facebook

You can create new Facebook apps using the Facebook Developer console at https://developers.facebook.com/apps. Click **Add a New App** and follow the instructions to create an app ID.

> If you don't have a Facebook account, you can set up Google or Twitter login instead. The steps on the provider's site will be different, but the code is almost identical.

Next, set up Facebook Login and then click Settings on the left side, under Facebook Login:



Add the following URL to the **Valid OAuth redirect URIs** box:

```
http://localhost:5000/signin-facebook
```

> The port that your application runs on may differ. It's typically port 5000 if you use `dotnet start`, but if you're on Windows, it could be a random port like 54574. Either way, you can always see the port your application is running on in the address bar of your web browser.

Click **Save Changes** and then head over to the Dashboard page. Here you can see the app ID and secret generated by Facebook, which you'll need in a moment (keep this tab open).

To enable Facebook login in ASP.NET Core Identity, add this code anywhere in the `ConfigureServices` method in the `Startup` class:

```
services
    .AddAuthentication()
    .AddFacebook(options =>
    {
        options.AppId = Configuration["Facebook:AppI
d"];
        options.AppSecret = Configuration["Facebook:
AppSecret"];
    });
```

Instead of hardcoding the Facebook app ID and secret in your code, the values are pulled from the configuration system. The `appsettings.json` file is normally the place to store configuration data for your project. However, since it's checked into source control, it's not good for sensitive data like an app secret. (If your app secret was pushed to GitHub, for example, anyone could steal it and do bad things on your behalf.)

Store secrets safely with the Secrets Manager

You can use the Secrets Manager tool for sensitive data like an app secret. Run this line in the terminal to make sure it's installed (make sure you're currently in the project directory):

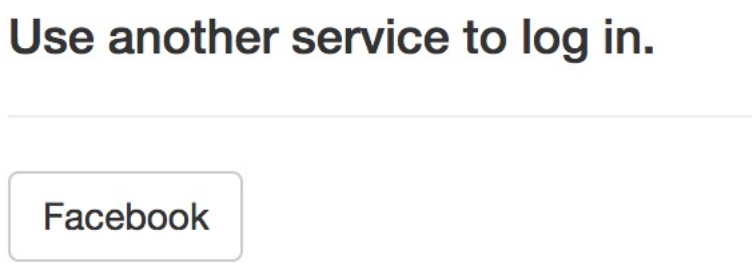
```
dotnet user-secrets --help
```

Copy the app ID and secret from the Facebook app dashboard and use the `set` command to save the values in the Secrets Manager:

```
dotnet user-secrets set Facebook:AppId <paste app id
>
dotnet user-secrets set Facebook:AppSecret <paste ap
p secret>
```

The values from the Secrets Manager are loaded into the `Configuration` property when your application starts up, so they're available to the code in `ConfigureServices` you added before.

Run your application and click the Login link in the navbar. You'll see a new button for logging in with Facebook:



Try logging in with Facebook. You'll be redirected and prompted to give your app permission in Facebook, then redirected back and logged in.

Require authentication

Often you'll want to require the user to log in before they can access certain parts of your application. For example, it makes sense to show the home page to everyone (whether you're logged in or not), but only show your to-do list after you've logged in.

You can use the `[Authorize]` attribute in ASP.NET Core to require a logged-in user for a particular action, or an entire controller. To require authentication for all actions of the `TodoController`, add the attribute above the first line of the controller:

```
[Authorize]
public class TodoController : Controller
{
    // ...
}
```

Add this `using` statement at the top of the file:

```
using Microsoft.AspNetCore.Authorization;
```

Try running the application and accessing `/todo` without being logged in. You'll be redirected to the login page automatically.

> Despite the name of the attribute, we are really doing an authentication check here, not an authorization check. Sorry to be confusing.

Using identity in the application

The to-do list items themselves are still shared between all users, because the to-do entities aren't tied to a particular user. Now that the `[Authorize]` attribute ensures that you must be logged in to see the to-do view, you can filter the database query based on who is logged in.

First, inject a `UserManager<ApplicationUser>` into the `TodoController`:

`Controllers/TodoController.cs`

```
[Authorize]
public class TodoController : Controller
{
    private readonly ITodoItemService _todoItemServi
ce;
    private readonly UserManager<ApplicationUser> _u
serManager;

    public TodoController(ITodoItemService todoItemS
ervice,
        UserManager<ApplicationUser> userManager)
    {
        _todoItemService = todoItemService;
        _userManager = userManager;
    }

    // ...
}
```

You'll need to add a new `using` statement at the top:

```
using Microsoft.AspNetCore.Identity;
```

The `UserManager` class is part of ASP.NET Core Identity. You can use it to look up the current user in the `Index` action:

```
public async Task<IActionResult> Index()
{
    var currentUser = await _userManager.GetUserAsyn
c(User);
    if (currentUser == null) return Challenge();

    var todoItems = await _todoItemService.GetIncomp
leteItemsAsync(currentUser);

    var model = new TodoViewModel()
    {
        Items = todoItems
    };

    return View(model);
}
```

The new code at the top of the action method uses the `UserManager` to get the current user from the `User` property available in the action:

```
var currentUser = await _userManager.GetUserAsync(Us
er);
```

If there is a logged-in user, the `User` property contains a lightweight object with some (but not all) of the user's information. The `UserManager` uses this to look up the full user details in the database via the `GetUserAsync`.

The value of `currentUser` should never be null, because the `[Authorize]` attribute is present on the controller. However, it's a good idea to do a sanity check, just in case. You can use the `Challenge()` method to force the user to log in again if their information is missing:

```
if (currentUser == null) return Challenge();
```

Since you're now passing an `ApplicationUser` parameter to `GetIncompleteItemsAsync`, you'll need to update the `ITodoItemService` interface:

Services/ITodoItemService.cs

```
public interface ITodoItemService
{
    Task<IEnumerable<TodoItem>> GetIncompleteItemsAs
ync(ApplicationUser user);

    // ...
}
```

The next step is to update the database query and show only items owned by the current user.

Update the database

You'll need to add a new property to the `TodoItem` entity model so each item can reference the user that owns it:

```
public string OwnerId { get; set; }
```

Since you updated the entity model used by the database context, you also need to migrate the database. Create a new migration using `dotnet ef` in the terminal:

```
dotnet ef migrations add AddItemOwnerId
```

This creates a new migration called `AddItemOwner` which will add a new column to the `Items` table, mirroring the change you made to the `TodoItem` entity model.

> Note: You'll need to manually tweak the migration file if you're using SQLite as your database. See the *Create a migration* section in the *Use a database* chapter for more details.

Use `dotnet ef` again to apply it to the database:

```
dotnet ef database update
```

Update the service class

With the database and the database context updated, you can now update the `GetIncompleteItemsAsync` method in the `TodoItemService` and add another clause to the `Where` statement:

Services/TodoItemService.cs

```
public async Task<IEnumerable<TodoItem>> GetIncomple
teItemsAsync(ApplicationUser user)
{
    return await _context.Items
        .Where(x => x.IsDone == false && x.OwnerId =
= user.Id)
        .ToArrayAsync();
}
```

If you run the application and register or log in, you'll see an empty to-do list once again. Unfortunately, any items you try to add disappear into the ether, because you haven't updated the Add Item operation to save the current user to new items.

Update the Add Item and Mark Done operations

You'll need to use the `UserManager` to get the current user in the `AddItem` and `MarkDone` action methods, just like you did in `Index`. The only difference is that these

methods will return a `401 Unauthorized` response to the frontend code, instead of challenging and redirecting the user to the login page.

Here are both updated methods in the `TodoController`:

```
public async Task<IActionResult> AddItem(NewTodoItem
 newItem)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }

    var currentUser = await _userManager.GetUserAsyn
c(User);
    if (currentUser == null) return Unauthorized();

    var successful = await _todoItemService.AddItemA
sync(newItem, currentUser);
    if (!successful)
    {
        return BadRequest(new { error = "Could not a
dd item." });
    }

    return Ok();
}

public async Task<IActionResult> MarkDone(Guid id)
{
    if (id == Guid.Empty) return BadRequest();

    var currentUser = await _userManager.GetUserAsyn
```

```
c(User);
    if (currentUser == null) return Unauthorized();

    var successful = await _todoItemService.MarkDone
Async(id, currentUser);
    if (!successful) return BadRequest();

    return Ok();
}
```

Both service methods must now accept an `ApplicationUser` parameter. Update the interface definition in `ITodoItemService`:

```
Task<bool> AddItemAsync(NewTodoItem newItem, Applica
tionUser user);

Task<bool> MarkDoneAsync(Guid id, ApplicationUser us
er);
```

And finally, update the service method implementations in the `TodoItemService`.

For the `AddItemAsync` method, set the `Owner` property when you construct a `new TodoItem`:

```
public async Task<bool> AddItemAsync(NewTodoItem new
Item, ApplicationUser user)
{
    var entity = new TodoItem
    {
        Id = Guid.NewGuid(),
        OwnerId = user.Id,
        IsDone = false,
        Title = newItem.Title,
        DueAt = DateTimeOffset.Now.AddDays(3)
    };

    // ...
}
```

The `Where` clause in the `MarkDoneAsync` method also needs to check for the user's ID, so a rogue user can't complete someone else's items by guessing their IDs:

```
public async Task<bool> MarkDoneAsync(Guid id, Appli
cationUser user)
{
    var item = await _context.Items
        .Where(x => x.Id == id && x.OwnerId == user.
Id)
        .SingleOrDefaultAsync();

    // ...
}
```

All done! Try using the application with two different user accounts. The to-do items stay private for each account.

Authorization with roles

Roles are a common approach to handling authorization and permissions in a web application. For example, you might have an Administrator role that allows admins to see and manage all the users registered for your app, while normal users can only see their own information.

Add a Manage Users page

First, create a new controller:

Controllers/ManageUsersController.cs

```
using System;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Identity;
using AspNetCoreTodo.Models;
using Microsoft.EntityFrameworkCore;

namespace AspNetCoreTodo.Controllers
{
    [Authorize(Roles = "Administrator")]
    public class ManageUsersController : Controller
    {
        private readonly UserManager<ApplicationUser
```

```
> _userManager;

        public ManageUsersController(UserManager<App
licationUser> userManager)
        {
            _userManager = userManager;
        }

        public async Task<IActionResult> Index()
        {
            var admins = await _userManager
                .GetUsersInRoleAsync("Administrator"
);

            var everyone = await _userManager.Users
                .ToArrayAsync();

            var model = new ManageUsersViewModel
            {
                Administrators = admins,
                Everyone = everyone
            };

            return View(model);
        }
    }
}
```

Setting the `Roles` property on the `[Authorize]` attribute will ensure that the user must be logged in **and** assigned the `Administrator` role in order to view the page.

Next, create a view model:

Models/ManageUsersViewModel.cs

```
using System.Collections.Generic;
using AspNetCoreTodo.Models;

namespace AspNetCoreTodo
{
    public class ManageUsersViewModel
    {
        public IEnumerable<ApplicationUser> Administ
rators { get; set; }

        public IEnumerable<ApplicationUser> Everyone
 { get; set; }
    }
}
```

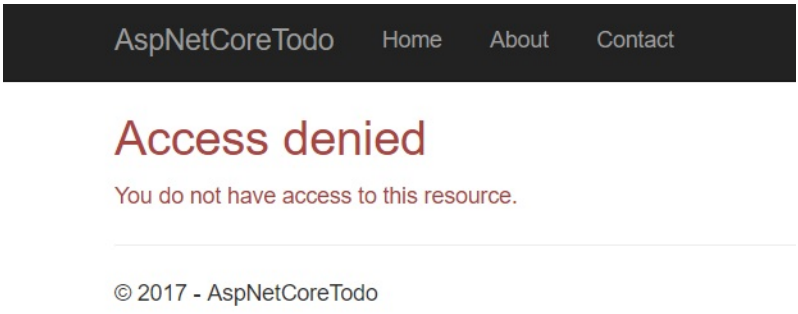
Finally, create a view for the Index action:

Views/ManageUsers/Index.cshtml

```
@model ManageUsersViewModel

@{
    ViewData["Title"] = "Manage users";
}

<h2>@ViewData["Title"]</h2>

<h3>Administrators</h3>

<table class="table">
    <thead>
```

```
            <tr>
                <td>Id</td>
                <td>Email</td>
            </tr>
        </thead>

        @foreach (var user in Model.Administrators)
        {
            <tr>
                <td>@user.Id</td>
                <td>@user.Email</td>
            </tr>
        }
    </table>

    <h3>Everyone</h3>

    <table class="table">
        <thead>
            <tr>
                <td>Id</td>
                <td>Email</td>
            </tr>
        </thead>

        @foreach (var user in Model.Everyone)
        {
            <tr>
                <td>@user.Id</td>
                <td>@user.Email</td>
            </tr>
        }
    </table>
```

Start up the application and try to access the `/ManageUsers` route while logged in as a normal user. You'll see this access denied page:



That's because users aren't assigned the `Administrator` role automatically.

Create a test administrator account

For obvious security reasons, there isn't a checkbox on the registration page that makes it easy for anyone to create an administrator account. Instead, you can write some code in the `Startup` class that will create a test admin account the first time the application starts up.

Add this code to the `if (env.IsDevelopment())` branch of the `Configure` method:

`Startup.cs`

```
if (env.IsDevelopment())
{
    // (... some code)

    // Make sure there's a test admin account
    EnsureRolesAsync(roleManager).Wait();
    EnsureTestAdminAsync(userManager).Wait();
}
```

The `EnsureRolesAsync` and `EnsureTestAdminAsync` methods will need access to the `RoleManager` and `UserManager` services. You can inject them into the `Configure` method, just like you inject any service into your controllers:

```
public void Configure(IApplicationBuilder app,
    IHostingEnvironment env,
    UserManager<ApplicationUser> userManager,
    RoleManager<IdentityRole> roleManager)
{
    // ...
}
```

Add the two new methods below the `Configure` method. First, the `EnsureRolesAsync` method:

```
private static async Task EnsureRolesAsync(RoleManag
er<IdentityRole> roleManager)
{
    var alreadyExists = await roleManager.RoleExists
Async(Constants.AdministratorRole);

    if (alreadyExists) return;

    await roleManager.CreateAsync(new IdentityRole(C
onstants.AdministratorRole));
}
```

This method checks to see if an `Administrator` role exists in the database. If not, it creates one. Instead of repeatedly typing the string `"Administrator"`, create a small class called `Constants` to hold the value:

`Constants.cs`

```
namespace AspNetCoreTodo
{
    public static class Constants
    {
        public const string AdministratorRole = "Adm
inistrator";
    }
}
```

> Feel free to update the `ManageUsersController` you created before to use this constant value as well.

Next, write the `EnsureTestAdminAsync` method:

`Startup.cs`

```
private static async Task EnsureTestAdminAsync(UserM
anager<ApplicationUser> userManager)
{
    var testAdmin = await userManager.Users
        .Where(x => x.UserName == "admin@todo.local"
)
        .SingleOrDefaultAsync();

    if (testAdmin != null) return;

    testAdmin = new ApplicationUser { UserName = "ad
min@todo.local", Email = "admin@todo.local" };
    await userManager.CreateAsync(testAdmin, "NotSec
ure123!!");
    await userManager.AddToRoleAsync(testAdmin, Cons
tants.AdministratorRole);
}
```

If there isn't already a user with the username `admin@todo.local` in the database, this method will create one and assign a temporary password. After you log in for the first time, you should change the account's password to something secure.

> Because these two methods are asynchronous and return a `Task`, the `Wait` method must be used in `Configure` to make sure they finish before `Configure` moves on. You'd normally use `await` for this, but for technical reasons you can't use `await` in `Configure`. This is a rare exception - you should use `await` everywhere else!

When you start the application next, the `admin@todo.local` account will be created and assigned the `Administrator` role. Try logging in with this account, and navigating to `http://localhost:5000/ManageUsers`. You'll see a list of all users registered for the application.

> As an extra challenge, try adding more administration features to this page. For example, you could add a button that gives an administrator the ability to delete a user account.

Check for authorization in a view

The `[Authorize]` attribute makes it easy to perform an authorization check in a controller or action method, but what if you need to check authorization in a view? For example, it would be nice to display a "Manage users" link in the navigation bar if the logged-in user is an administrator.

You can inject the `UserManager` directly into a view to do these types of authorization checks. To keep your views clean and organized, create a new partial view that will add an item to the navbar in the layout:

`Views/Shared/_AdminActionsPartial.cshtml`

```
@using Microsoft.AspNetCore.Identity
@using AspNetCoreTodo.Models

@inject SignInManager<ApplicationUser> SignInManager
@inject UserManager<ApplicationUser> UserManager

@if (SignInManager.IsSignedIn(User))
{
    var currentUser = await UserManager.GetUserAsync
(User);

    var isAdmin = currentUser != null
        && await UserManager.IsInRoleAsync(currentUs
er, Constants.AdministratorRole);

    if (isAdmin) {
        <ul class="nav navbar-nav navbar-right">
            <li><a asp-controller="ManageUsers" asp-
action="Index">Manage Users</a></li>
        </ul>
    }
}
```

A **partial view** is a small piece of a view that gets embedded into another view. It's common to name partial views starting with an `_` underscore, but it's not necessary.

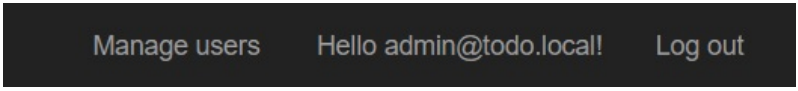
This partial view first uses the `SignInManager` to quickly determine whether the user is logged in. If they aren't, the rest of the view code can be skipped. If there *is* a logged-in user, the `UserManager` is used to look up their details and perform an authorization check with `IsInRoleAsync`. If all checks succeed, a navbar item is rendered.

To include this partial in the main layout, edit `_Layout.cshtml` and add it in the navbar section:

`Views/Shared/_Layout.cshtml`

```
<div class="navbar-collapse collapse">
    <ul class="nav navbar-nav">
        <li><a asp-area="" asp-controller="Home" asp
-action="Index">Home</a></li>
        <li><a asp-area="" asp-controller="Home" asp
-action="About">About</a></li>
        <li><a asp-area="" asp-controller="Home" asp
-action="Contact">Contact</a></li>
    </ul>
    @await Html.PartialAsync("_LoginPartial")
    @await Html.PartialAsync("_AdminActionsPartial")
</div>
```

When you log in with an administrator account, you'll now see a new item on the top right:



Wrap up

ASP.NET Core Identity is a powerful security and identity system that helps you add authentication and authorization checks, and makes it easy to integrate with external identity providers. The `dotnet new` templates give you pre-built views and controllers that handle common scenarios like login and registration so you can get up and running quickly.

There's much more that ASP.NET Core Identity can do. You can learn more in the documentation and examples available at https://docs.asp.net.

Automated testing

Writing tests is an important part of building any application. Testing your code helps you spot and avoid bugs, and makes it easier to refactor your code later without breaking functionality or introducing new problems.

In this chapter you'll learn how to write both **unit tests** and **integration tests** that exercise your ASP.NET Core application. Unit tests are small tests that make sure a single method or a few lines of code are working properly. Integration tests (sometimes called **functional** tests) are larger tests that simulate real-world scenarios and exercise multiple layers or parts of your application.

Unit testing

Unit tests are small, quick tests that check the behavior of a single method or chunk of logic. Instead of testing a whole group of classes, or the entire system (as integration tests do), unit tests rely on **mocking** or replacing the objects the method-under-test depends on.

For example, the `TodoController` has two dependencies: an `ITodoItemService` and the `UserManager`. The `TodoItemService`, in turn, depends on the `ApplicationDbContext`. (The idea that you can draw a line from `TodoController` -> `TodoItemService` -> `ApplicationDbContext` is called a *dependency graph*).

When the application runs normally, the ASP.NET Core dependency injection system injects each of those objects into the dependency graph when the `TodoController` or the `TodoItemService` is created.

When you write a unit test, on the other hand, you'll manually inject mock or test-only versions of those dependencies. This means you can isolate just the logic in the class or method you are testing. (If you're testing a service, you don't want to also be accidentally writing to your database!)

Create a test project

It's a common practice to create a separate project for your tests, to keep things clean and organized. The new test project should live in a directory that's next to (not inside) your main project's directory.

If you're currently in your project directory, `cd` up one level. (This directory will also be called `AspNetCoreTodo`). Then use these commands to scaffold a new test project:

```
mkdir AspNetCoreTodo.UnitTests
cd AspNetCoreTodo.UnitTests
dotnet new xunit
```

xUnit.NET is a popular test framework for .NET code that can be used to write both unit and integration tests. Like everything else, it's a set of NuGet packages that can be installed in any project. The `dotnet new xunit` template already includes everything you need.

Your directory structure should now look like this:

```
AspNetCoreTodo/
    AspNetCoreTodo/
        AspNetCoreTodo.csproj
        Controllers/
        (etc...)

    AspNetCoreTodo.UnitTests/
        AspNetCoreTodo.UnitTests.csproj
```

Since the test project will use the classes defined in your main project, you'll need to add a reference to the main project:

```
dotnet add reference ../AspNetCoreTodo/AspNetCoreTod
o.csproj
```

Delete the `UnitTest1.cs` file that's automatically created. You're ready to write your first test.

Write a service test

Take a look at the logic in the `AddItemAsync` method of the `TodoItemService`:

```
public async Task<bool> AddItemAsync(NewTodoItem new
Item, ApplicationUser user)
{
    var entity = new TodoItem
    {
        Id = Guid.NewGuid(),
        OwnerId = user.Id,
        IsDone = false,
        Title = newItem.Title,
        DueAt = DateTimeOffset.Now.AddDays(3)
    };

    _context.Items.Add(entity);

    var saveResult = await _context.SaveChangesAsync
();
    return saveResult == 1;
}
```

This method makes a number of decisions or assumptions about the new item before it actually saves it to the database:

- The `OwnerId` property should be set to the user's ID
- New items should always be incomplete (`IsDone = false`)
- The title of the new item should be copied from `newItem.Title`
- New items should always be due 3 days from now

> These types of decisions made by your code are called *business logic*, because it's logic that relates to the purpose or "business" of your application. Other examples of business logic include things like calculating a total cost based on product prices and tax rates, or checking whether a player has enough points to level up in a game.

These decisions make sense, and it also makes sense to have a test that ensures that this logic doesn't change down the road. (Imagine if you or someone else refactored the `AddItemAsync` method and forgot about one of these assumptions. It might be unlikely when your services are simple, but it becomes important to have automated checks as your application becomes more complicated.)

To write a unit test that will verify the logic in the `TodoItemService`, create a new class in your test project:

`AspNetCoreTodo.UnitTests/TodoItemServiceShould.cs`

```
using System;
using System.Threading.Tasks;
using AspNetCoreTodo.Data;
using AspNetCoreTodo.Models;
using AspNetCoreTodo.Services;
using Microsoft.EntityFrameworkCore;
using Xunit;

namespace AspNetCoreTodo.UnitTests
{
    public class TodoItemServiceShould
    {
        [Fact]
        public async Task AddNewItem()
        {
            // ...
        }
    }
}
```

The `[Fact]` attribute comes from the xUnit.NET package, and it marks this method as a test method.

> There are many different ways of naming and organizing tests, all with different pros and cons. I like postfixing my test classes with `Should` to create a readable sentence with the test method name, but feel free to use your own style!

The `TodoItemService` requires an `ApplicationDbContext`, which is normally connected to your development or live database. You won't want to use that for tests. Instead, you can use Entity Framework Core's in-memory database provider in your test code. Since the entire database exists in memory, it's wiped out every time the test is restarted. And, since it's a proper Entity Framework Core provider, the `TodoItemService` won't know the difference!

Use a `DbContextOptionsBuilder` to configure the in-memory database provider, and then make a call to `AddItem`:

```
var options = new DbContextOptionsBuilder<Applicatio
nDbContext>()
    .UseInMemoryDatabase(databaseName: "Test_AddNewI
tem").Options;

// Set up a context (connection to the DB) for writi
ng
using (var inMemoryContext = new ApplicationDbContex
t(options))
{
    var service = new TodoItemService(inMemoryContex
t);

    var fakeUser = new ApplicationUser
    {
        Id = "fake-000",
        UserName = "fake@fake"
    };

    await service.AddItemAsync(new NewTodoItem { Tit
le = "Testing?" }, fakeUser);
}
```

The last line creates a new to-do item called `Testing?`, and tells the service to save it to the (in-memory) database. To verify that the business logic ran correctly, retrieve the item:

```
// Use a separate context to read the data back from
 the DB
using (var inMemoryContext = new ApplicationDbContex
t(options))
{
    Assert.Equal(1, await inMemoryContext.Items.Coun
tAsync());

    var item = await inMemoryContext.Items.FirstAsyn
c();
    Assert.Equal("Testing?", item.Title);
    Assert.Equal(false, item.IsDone);
    Assert.True(DateTimeOffset.Now.AddDays(3) - item
.DueAt < TimeSpan.FromSeconds(1));
}
```

The first verification step is a sanity check: there should never be more than one item saved to the in-memory database. Assuming that's true, the test retrieves the saved item with `FirstAsync` and then asserts that the properties are set to the expected values.

Asserting a datetime value is a little tricky, since comparing two dates for equality will fail if even the millisecond components are different. Instead, the test checks that the `DueAt` value is less than a second away from the expected value.

> Both unit and integration tests typically follow the AAA (Arrange-Act-Assert) pattern: objects and data are set up first, then some action is performed, and finally the test checks (asserts) that the expected behavior occurred.

Here's the final version of the `AddNewItem` test:

AspNetCoreTodo.UnitTests/TodoItemServiceShould.cs

```
public class TodoItemServiceShould
{
    [Fact]
    public async Task AddNewItem()
    {
        var options = new DbContextOptionsBuilder<Ap
plicationDbContext>()
            .UseInMemoryDatabase(databaseName: "Test
_AddNewItem")
            .Options;

        // Set up a context (connection to the DB) f
or writing
        using (var inMemoryContext = new Application
DbContext(options))
        {
            var service = new TodoItemService(inMemo
ryContext);
            await service.AddItemAsync(new NewTodoIt
em { Title = "Testing?" }, null);
        }

        // Use a separate context to read the data b
```

```
ack from the DB
        using (var inMemoryContext = new Application
DbContext(options))
        {
            Assert.Equal(1, await inMemoryContext.It
ems.CountAsync());

            var item = await inMemoryContext.Items.F
irstAsync();
            Assert.Equal("Testing?", item.Title);
            Assert.Equal(false, item.IsDone);
            Assert.True(DateTimeOffset.Now.AddDays(3
) - item.DueAt < TimeSpan.FromSeconds(1));
        }
    }
}
```

Run the test

On the terminal, run this command (make sure you're still in the `AspNetCoreTodo.UnitTests` directory):

```
dotnet test
```

The `test` command scans the current project for tests (marked with `[Fact]` attributes in this case), and runs all the tests it finds. You'll see an output similar to:

```
Starting test execution, please wait...
[xUnit.net 00:00:00.7595476]   Discovering: AspNetCo
reTodo.UnitTests
[xUnit.net 00:00:00.8511683]   Discovered:  AspNetCo
reTodo.UnitTests
[xUnit.net 00:00:00.9222450]   Starting:    AspNetCo
reTodo.UnitTests
[xUnit.net 00:00:01.3862430]   Finished:    AspNetCo
reTodo.UnitTests

Total tests: 1. Passed: 1. Failed: 0. Skipped: 0.
Test Run Successful.
Test execution time: 1.9074 Seconds
```

You now have one test providing test coverage of the `TodoItemService`. As an extra-credit challenge, try writing unit tests that ensure:

- `MarkDoneAsync` returns false if it's passed an ID that doesn't exist
- `MarkDoneAsync` returns true when it makes a valid item as complete
- `GetIncompleteItemsAsync` returns only the items owned by a particular user

Integration testing

Compared to unit tests, integration tests exercise the whole application stack (routing, controllers, services, database). Instead of isolating one class or component, integration tests ensure that all of the components of your application are working together properly.

Integration tests are slower and more involved than unit tests, so it's common for a project to have lots of unit tests but only a handful of integration tests.

In order to test the whole stack (including controller routing), integration tests typically make HTTP calls to your application just like a web browser would.

To write integration tests that make HTTP requests, you could manually start your application run tests that make requests to `http://localhost:5000` (and hope the app is still running). ASP.NET Core provides a nicer way to host your application for testing, however: using the `TestServer` class. `TestServer` can host your application for the duration of the test, and then stop it automatically when the test is complete.

Create a test project

You could keep your unit tests and integration tests in the same project (feel free to do so), but for the sake of completeness, I'll show you how to create a separate project for your integration tests.

If you're currently in your project directory, `cd` up one level to the base `AspNetCoreTodo` directory. Use these commands to scaffold a new test project:

```
mkdir AspNetCoreTodo.IntegrationTests
cd AspNetCoreTodo.IntegrationTests
dotnet new xunit
```

Your directory structure should now look like this:

```
AspNetCoreTodo/
    AspNetCoreTodo/
        AspNetCoreTodo.csproj
        Controllers/
        (etc...)

    AspNetCoreTodo.UnitTests/
        AspNetCoreTodo.UnitTests.csproj

    AspNetCoreTodo.IntegrationTests/
        AspNetCoreTodo.IntegrationTests.csproj
```

Since the test project will use the classes defined in your main project, you'll need to add a reference to the main project:

```
dotnet add reference ../AspNetCoreTodo/AspNetCoreTod
o.csproj
```

You'll also need to add the `Microsoft.AspNetCore.TestHost` NuGet package:

```
dotnet add package Microsoft.AspNetCore.TestHost
```

Delete the `UnitTest1.cs` file that's created by `dotnet new`. You're ready to write an integration test.

Write an integration test

There are a few things that need to be configured on the test server before each test. Instead of cluttering the test with this setup code, you can factor out this setup to a separate class. Create a new class called `TestFixture`:

`AspNetCoreTodo.IntegrationTests/TestFixture.cs`

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Net.Http;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.TestHost;
using Microsoft.Extensions.Configuration;

namespace AspNetCoreTodo.IntegrationTests
```

```
{
    public class TestFixture : IDisposable
    {
        private readonly TestServer _server;

        public TestFixture()
        {
            var builder = new WebHostBuilder()
                .UseStartup<AspNetCoreTodo.Startup>(
)
                .ConfigureAppConfiguration((context,
 configBuilder) =>
                {
                    configBuilder.SetBasePath(Path.C
ombine(
                        Directory.GetCurrentDirector
y(), "..\\..\\..\\..\\AspNetCoreTodo"));

                    configBuilder.AddJsonFile("appse
ttings.json");

                    // Add fake configuration for Fa
cebook middleware (to avoid startup errors)
                    configBuilder.AddInMemoryCollect
ion(new Dictionary<string, string>()
                    {
                        ["Facebook:AppId"] = "fake-a
pp-id",
                        ["Facebook:AppSecret"] = "fa
ke-app-secret"
                    });
                });
            _server = new TestServer(builder);

            Client = _server.CreateClient();
```

```
            Client.BaseAddress = new Uri("http://loc
alhost:5000");
        }

        public HttpClient Client { get; }

        public void Dispose()
        {
            Client.Dispose();
            _server.Dispose();
        }
    }
}
```

This class takes care of setting up a `TestServer`, and will help keep the tests themselves clean and tidy.

> If you configured Facebook login in the *Security and identity* chapter., it's necessary to add fake values for the Facebook app ID and secret (in the `ConfigureAppConfiguration` block above). This is because the test server doesn't have access to the values in the Secrets Manager. Adding some fake values in this fixture class will prevent an error when the test server starts up.

Now you're (really) ready to write an integration test. Create a new class called `TodoRouteShould`:

`AspNetCoreTodo.IntegrationTests/TodoRouteShould.cs`

```
using System.Net;
using System.Net.Http;
using System.Threading.Tasks;
using Xunit;

namespace AspNetCoreTodo.IntegrationTests
{
    public class TodoRouteShould : IClassFixture<Tes
tFixture>
    {
        private readonly HttpClient _client;

        public TodoRouteShould(TestFixture fixture)
        {
            _client = fixture.Client;
        }

        [Fact]
        public async Task ChallengeAnonymousUser()
        {
            // Arrange
            var request = new HttpRequestMessage(Htt
pMethod.Get, "/todo");

            // Act: request the /todo route
            var response = await _client.SendAsync(r
equest);

            // Assert: anonymous user is redirected
to the login page
            Assert.Equal(HttpStatusCode.Redirect, re
sponse.StatusCode);
            Assert.Equal("http://localhost:5000/Acco
unt/Login?ReturnUrl=%2Ftodo",
```

```
                            response.Headers.Location.To
String());
        }
    }
}
```

This test makes an anonymous (not-logged-in) request to the `/todo` route and verifies that the browser is redirected to the login page.

This scenario is a good candidate for an integration test, because it involves multiple components of the application: the routing system, the controller, the fact that the controller is marked with `[Authorize]`, and so on. It's also a good test because it ensures you won't ever accidentally remove the `[Authorize]` attribute and make the to-do view accessible to everyone.

Run the test in the terminal with `dotnet test`. If everything's working right, you'll see a success message:

```
Starting test execution, please wait...
[xUnit.net 00:00:00.7237031]   Discovering: AspNetCo
reTodo.IntegrationTests
[xUnit.net 00:00:00.8118035]   Discovered:  AspNetCo
reTodo.IntegrationTests
[xUnit.net 00:00:00.8779059]   Starting:    AspNetCo
reTodo.IntegrationTests
[xUnit.net 00:00:01.5828576]   Finished:    AspNetCo
reTodo.IntegrationTests

Total tests: 1. Passed: 1. Failed: 0. Skipped: 0.
Test Run Successful.
Test execution time: 2.0588 Seconds
```

Wrap up

Testing is a broad topic, and there's much more to learn. This chapter doesn't touch on UI testing or testing frontend (JavaScript) code, which probably deserve entire books of their own. You should, however, have the skills and base knowledge you need to practice and learn more about writing tests for your own applications.

As always, the ASP.NET Core documentation (https://docs.asp.net) and StackOverflow are good resources for learning more and finding answers when you get stuck.

Deploy the application

You've come a long way, but you're not quite done yet. Once you've created a great application, you need to share it with the world!

Because ASP.NET Core applications can run on Windows, Mac, or Linux, there are a number of different ways you can deploy your application. In this chapter, I'll show you the most common (and easiest) ways to go live.

Deployment options

ASP.NET Core applications are typically deployed to one of these environments:

- **Any Docker host**. Any machine capable of hosting Docker containers can be used to host an ASP.NET Core application. Creating a Docker image is a very quick way to get your application deployed, especially if you're familiar with Docker. (If you're not, don't worry! I'll cover the steps later.)

- **Azure**. Microsoft Azure has native support for ASP.NET Core applications. If you have an Azure subscription, you just need to create a Web App and upload your project files. I'll cover how to do this with the Azure CLI in the next section.
- **Linux (with Nginx)**. If you don't want to go the Docker route, you can still host your application on any Linux server (this includes Amazon EC2 and DigitalOcean virtual machines). It's typical to pair ASP.NET Core with the Nginx reverse proxy. (More about Nginx below.)
- **Windows**. You can use the IIS web server on Windows to host ASP.NET Core applications. It's usually easier (and cheaper) to just deploy to Azure, but if you prefer managing Windows servers yourself, it'll work just fine.

Kestrel and reverse proxies

> If you don't care about the guts of hosting ASP.NET Core applications and just want the step-by-step instructions, feel free to skip to one of the next two sections!

ASP.NET Core includes a fast, lightweight development web server called Kestrel. It's the server you've been using every time you ran the app locally and browsed to `http://localhost:5000`. When you deploy your application to a production environment, it'll still use Kestrel behind the scenes. However, it's recommended that you put a reverse proxy in front of Kestrel, because Kestrel doesn't yet have load balancing and other features that bigger web servers have.

On Linux (and in Docker containers), you can use Nginx or the Apache web server to receive incoming requests from the internet and route them to your application hosted with Kestrel. If you're on Windows, IIS does the same thing.

If you're using Azure to host your application, this is all taken care of for you automatically. I'll cover setting up Nginx as a reverse proxy in the Docker section.

Deploy to Azure

Deploying your ASP.NET Core application to Azure only takes a few steps. You can do it through the Azure web portal, or on the command line using the Azure CLI. I'll cover the latter.

What you'll need

- Git (use `git --version` to make sure it's installed)
- The Azure CLI (follow the install instructions at https://github.com/Azure/azure-cli)
- An Azure subscription (the free subscription is fine)
- A deployment configuration file in your project root

Create a deployment configuration file

Since there are multiple projects in your directory structure (the web application, and two test projects), Azure won't know which one to show to the world. To fix this, create a file called `.deployment` at the very top of your directory structure:

```
.deployment
```

```
[config]
project = AspNetCoreTodo/AspNetCoreTodo.csproj
```

Make sure you save the file as `.deployment` with no other parts to the name. (On Windows, you may need to put quotes around the filename, like `".deployment"`, to prevent a `.txt` extension from being added.)

If you `ls` or `dir` in your top-level directory, you should see these items:

```
.deployment
AspNetCoreTodo
AspNetCoreTodo.IntegrationTests
AspNetCoreTodo.UnitTests
```

Set up the Azure resources

If you just installed the Azure CLI for the first time, run

```
az login
```

and follow the prompts to log in on your machine. Then, create a new Resource Group for this application:

```
az group create -l westus -n AspNetCoreTodoGroup
```

This creates a Resource Group in the West US region. If you're located far away from the western US, use `az account list-locations` to get a list of locations and find one closer to you.

Next, create an App Service plan in the group you just created:

```
az appservice plan create -g AspNetCoreTodoGroup -n
AspNetCoreTodoPlan --sku F1
```

> Sidebar: `F1` is the free app plan. If you want to use a custom domain name with your app, use the D1 ($10/month) plan or higher.

Now create a Web App in the App Service plan:

```
az webapp create -g AspNetCoreTodoGroup -p AspNetCor
eTodoPlan -n MyTodoApp
```

The name of the app (`MyTodoApp` above) must be globally unique in Azure. Once the app is created, it will have a default URL in the format: http://mytodoapp.azurewebsites.net

Update the application settings

> Sidebar: This is only necessary if you configured Facebook login in the *Security and identity* chapter.

Your application won't start up properly if it's missing the `Facebook:AppId` and `Facebook:AppSecret` configuration values. You'll need to add these using the Azure web portal:

1. Log in to your Azure account via https://portal.azure.com
2. Open your Web App (called `MyTodoApp` above)
3. Click on the **Application settings** tab
4. Under the **App settings** section, add `Facebook:AppId` and `Facebook:AppSecret` with their respective values
5. Click **Save** at the top

Deploy your project files to Azure

You can use Git to push your application files up to the Azure Web App. If your local directory isn't already tracked as a Git repo, run these commands to set it up:

```
git init
git add .
git commit -m "First commit!"
```

Next, create an Azure username and password for deployment:

```
az webapp deployment user set --user-name nate
```

Follow the instructions to create a password. Then use `config-local-git` to spit out a Git URL:

```
az webapp deployment source config-local-git -g AspN
etCoreTodoGroup -n MyTodoApp --out tsv

https://nate@mytodoapp.scm.azurewebsites.net/MyTodoA
pp.git
```

Copy the URL to the clipboard, and use it to add a Git remote to your local repository:

```
git remote add azure <paste>
```

You only need to do these steps once. Now, whenever you want to push your application files to Azure, check them in with Git and run

```
git push azure master
```

You'll see a stream of log messages as the application is deployed to Azure. When it's complete, browse to http://yourappname.azurewebsites.net to check it out!

Deploy with Docker

Containerization technologies like Docker can make it much easier to deploy web applications. Instead of spending time configuring a server with the dependencies it needs to run your app, copying files, and restarting processes, you can simply create a Docker image that contains everything your app needs to run, and spin it up as a container on any Docker host.

Docker can make scaling your app across multiple servers easier, too. Once you have an image, using it to create 1 container is the same process as creating 100 containers.

Before you start, you need the Docker CLI installed on your development machine. Search for "get docker for (mac/windows/linux)" and follow the instructions on the official Docker website. You can verify that it's installed correctly with

```
docker --version
```

> If you set up Facebook login in the *Security and identity* chapter, you'll need to use Docker secrets to securely set the Facebook app secret inside your container. Working with Docker secrets is outside the scope of this book. If you want, you can comment out the `AddFacebook` line in the `ConfigureServices` method to disable Facebook log in.

Add a Dockerfile

The first thing you'll need is a Dockerfile, which is like a recipe that tells Docker what your application needs.

Create a file called `Dockerfile` (no extension) in the web application root, next to `Program.cs`. Open it in your favorite editor. Write the following line:

```
FROM microsoft/dotnet:latest
```

This tells Docker to start your image from an existing image that Microsoft publishes. This will make sure the container has everything it needs to run an ASP.NET Core app.

```
COPY . /app
```

The `COPY` command copies the contents of your local directory (the source code of your application) into a directory called `/app` in the Docker image.

```
WORKDIR /app
```

`WORKDIR` is the Docker equivalent of `cd`. The remainder of the commands in the Dockerfile will run from inside the `/app` folder.

```
RUN ["dotnet", "restore"]
RUN ["dotnet", "build"]
```

These commands will execute `dotnet restore` (which downloads the NuGet packages your application needs) and `dotnet build` (which compiles the application).

```
EXPOSE 5000/tcp
```

By default, Docker containers don't expose any network ports to the outside world. You have to explicitly let Docker know that your app will be communicating on port 5000 (the default Kestrel port).

```
ENV ASPNETCORE_URLS http://*:5000
```

The `ENV` command sets environment variables in the container. The `ASPNETCORE_URLS` variable tells ASP.NET Core which network interface and port it should bind to.

```
ENTRYPOINT ["dotnet", "run"]
```

The last line of the Dockerfile starts up your application with the `dotnet run` command. Kestrel will start listening on port 5000, just like it does when you use `dotnet run` on your local machine.

The full Dockerfile looks like this:

`Dockerfile`

```
FROM microsoft/dotnet:latest
COPY . /app
WORKDIR /app
RUN ["dotnet", "restore"]
RUN ["dotnet", "build"]
EXPOSE 5000/tcp
ENV ASPNETCORE_URLS http://*:5000
ENTRYPOINT ["dotnet", "run"]
```

Create an image

Make sure the Dockerfile is saved, and then use `docker build` to create an image:

```
docker build -t aspnetcoretodo .
```

Don't miss the trailing period! That tells Docker to look for a Dockerfile in the current directory.

Once the image is created, you can run `docker images` to to list all the images available on your local machine. To test it out in a container, run

```
docker run -it -p 5000:5000 aspnetcoretodo
```

The `-it` flag tells Docker to run the container in interactive mode. When you want to stop the container, press `Control-C`.

Set up Nginx

At the beginning of this chapter, I mentioned that you should use a reverse proxy like Nginx to proxy requests to Kestrel. You can use Docker for this, too.

The overall architecture will consist of two containers: an Nginx container listening on port 80, forwarding requests to a separate container running Kestrel and listening on port 5000.

The Nginx container needs its own Dockerfile. To keep it from colliding with the Dockerfile you just created, make a new directory in the web application root:

```
mkdir nginx
```

Create a new Dockerfile and add these lines:

nginx/Dockerfile

```
FROM nginx
COPY nginx.conf /etc/nginx/nginx.conf
```

Next, create an `nginx.conf` file:

nginx/nginx.conf

```
events { worker_connections 1024; }

http {

        server {
              listen 80;

              location / {
                proxy_pass http://kestrel:5000;
                proxy_http_version 1.1;
                proxy_set_header Upgrade $http_upgra
de;
                proxy_set_header Connection 'keep-al
ive';
                proxy_set_header Host $host;
                proxy_cache_bypass $http_upgrade;
              }
        }
}
```

This configuration file tells Nginx to proxy incoming requests to `http://kestrel:5000`. (You'll see why `kestrel:5000` works in a moment.)

Set up Docker Compose

There's one more file to create. Up in the web application root directory, create `docker-compose.yml`:

`docker-compose.yml`

```
nginx:
    build: ./nginx
    links:
        - kestrel:kestrel
    ports:
        - "80:80"
kestrel:
    build: .
    ports:
        - "5000"
```

Docker Compose is a tool that helps you create and run multi-container applications. This configuration file defines two containers: `nginx` from the `./nginx/Dockerfile` recipe, and `kestrel` from the `./Dockerfile` recipe. The containers are explicitly linked together so they can communicate.

You can try spinning up the entire multi-container application by running:

```
docker-compose up
```

Try opening a browser and navigating to `http://localhost` (not 5000!). Nginx is listening on port 80 (the default HTTP port) and proxying requests to your ASP.NET Core application hosted by Kestrel.

Set up a Docker server

Specific setup instructions are outside the scope of this Little book, but any modern Linux distro (like Ubuntu) can be set up as a Docker host. For example, you could create a virtual machine with Amazon EC2, and install the Docker service. You can search for "amazon ec2 set up docker" (for example) for instructions.

I prefer DigitalOcean because they've made it really easy to get started. DigitalOcean has both a pre-built Docker virtual machine, and in-depth tutorials for getting Docker up and running (search for "digitalocean docker").

Conclusion

Thanks for making it to the end of the Little ASP.NET Core Book! There's a lot more to ASP.NET Core can do that couldn't fit in this short book, including

- building RESTful APIs and microservices
- using ASP.NET Core with single-page apps like Angular and React
- Razor Pages
- bundling and minifying static assets
- WebSockets

I write about many of these topics on my blog: https://www.recaffeinate.co

The official ASP.NET Core documentation also covers these topics, and more: https://docs.asp.net

Happy coding!

About the author

Hey, I'm Nate! I wrote the Little ASP.NET Core Book in a caffeine-fueled weekend because I love the .NET community and wanted to give back in my own little way. I

hope it helped you learn something new!

You can stay in touch with me on Twitter (@nbarbettini) or on my blog (https://www.recaffeinate.co).

Special thanks

To Jennifer, who always supports my crazy ideas.

To the following contributors who improved the Little ASP.NET Core Book:

- 0xNF

To these amazing polyglot programmers who translated the Little ASP.NET Core Book:

- sahinyanlik (Turkish)
- windsting, yuyi (Simplified Chinese)

Changelog

1.0.4 (2018-01-15): Added explanation of service container lifecycles, clarified server ports and the -o flag, and removed semicolons after Razor directives. Corrected Chinese translation author credit. Fixed other small typos and issues noticed by readers.

1.0.3 (2017-11-13): Typo fixes and small improvements suggested by readers.

1.0.2 (2017-10-20): More bug fixes and small improvements. Added link to translations.

1.0.1 (2017-09-23): Bug fixes and small improvements.

1.0.0 (2017-09-18): Initial release.