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 refactoring support for 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 applications. 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 to get the .NET Core SDK. 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

2.1.104
```

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

```
dotnet --info

.NET Command Line Tools (2.1.104)

Product Information:
 Version:            2.1.104
 Commit SHA-1 hash:  48ec687460

Runtime Environment:
 OS Name:     Mac OS X
 OS Version:  10.13

(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
```

The `dotnet new` command creates a new .NET project in C# by default. The `console` parameter selects a template for a console application (a program that outputs text to the screen). The `-o CsharpHelloWorld` parameter tells `dotnet new` to create a new directory called `CsharpHelloWorld` for all the project files. Move into this new directory:

```
cd CsharpHelloWorld
```

`dotnet new console` creates a basic C# program that writes the text `Hello World!` 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 very 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 your 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 directory to store your entire project, and move into it:

```
mkdir AspNetCoreTodo
cd AspNetCoreTodo
```

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.)

> You might be wondering why you have a directory called `AspNetCoreTodo` inside another directory called `AspNetCoreTodo`. The top-level or "root" directory can contain one or more project directories. The root directory is sometimes called a **solution directory**. Later, you'll add more project directories side-by-side with the `AspNetCoreTodo` project directory, all within a single root solution directory.

You'll see quite a few files show up in the new 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).

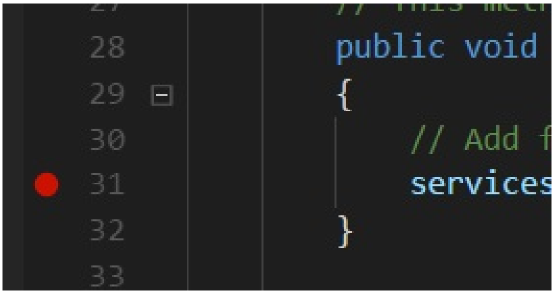
- The **Models**, **Views**, and **Controllers** directories contain the components of the Model-View-Controller (MVC) architecture. You'll explore all three in the next chapter.

- The **wwwroot** directory contains static assets like CSS, JavaScript, and image files. 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

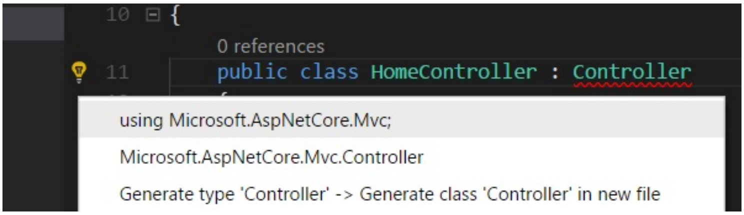
If you're using Visual Studio Code for the first time, here are a couple of helpful tips to get you started:

- **Open the project root folder**: In Visual Studio Code, choose File - Open or File - Open Folder. Open the `AspNetCoreTodo` folder (the root directory), not the inner project directory. If Visual Studio Code prompts you to install missing files, click Yes to add them.
- **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:



```
public void
{
    // Add f
    services
}
```

- **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:



```
{
    0 references
    public class HomeController : Controller
```

using Microsoft.AspNetCore.Mvc;
Microsoft.AspNetCore.Mvc.Controller
Generate type 'Controller' -> Generate class 'Controller' in new file

- **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`.

> These tips apply to Visual Studio (not Code) on Windows too. If you're using Visual Studio, you'll need to open the `.csproj` project file directly. Visual Studio will later prompt you to save the Solution file, which you should save in the root directory (the first `AspNetCoreTodo` folder). You can also create an ASP.NET Core project directly within Visual Studio using the templates in File - New Project.

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 root directory:

```
cd ..
git init
```

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), plus 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 a 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, and the cycle repeats

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. More specifically, you'll be creating:

- A web application server (sometimes called the "backend") using ASP.NET Core, C#, and the MVC pattern
- A database to store the user's to-do items using the SQLite database engine and a system called Entity Framework Core
- Web pages and an interface that the user will interact with via their browser, using HTML, CSS, and JavaScript (called the "frontend")
- A login form and security checks so each user's to-do list is kept private

Sound good? Let's built it! 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 directory, 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` and `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 will 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;
using System.ComponentModel.DataAnnotations;

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

        public bool IsDone { get; set; }

        [Required]
        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 an item's checkbox in the view.

- The **Title** property is a string (text value). This will hold the name or description of the to-do item. The `[Required]` attribute tells ASP.NET Core that this string can't be null or empty.

- 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

```
namespace AspNetCoreTodo.Models
{
    public class TodoViewModel
    {
        public TodoItem[] Items { get; set; }
    }
}
```

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 Handlebars moustaches, 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 (a sequence of to-do items) and display it in 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.

Create a `Todo` directory inside the `Views` directory, and add this file:

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" class="done-checkbox">
                </td>
                <td>@item.Title</td>
                <td>@item.DueAt</td>
            </tr>
        }
    </table>

    <div class="panel-footer add-item-form">
      <!-- TODO: Add item 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 will let the user mark the item as complete.

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 every other view is 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

The default template also includes a stylesheet with some basic CSS rules. The stylesheet is stored in the `wwwroot/css` directory. Add a few new 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.