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.

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 your code separate. Why? In a big, real-world application, you'll have to juggle many concerns:

- **Rendering views** and handling incoming data: this is what your controller already does.
- **Performing business logic**, or code and logic that's related to the purpose and "business" of your application. In a to-do list application, business logic means decisions like setting a default due date on new tasks, or only displaying tasks that are incomplete. Other examples of business logic include 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.
- **Saving and retrieving** items from a database.

Again, it's possible to do all of these things in a single, massive controller, but that quickly becomes too hard to manage and test. Instead, it's common to see applications split up into two, three, or more "layers" or tiers that each handle one (and only one) concern. This helps keep the controllers as simple as possible, and makes it easier to test and change the business logic and database code later.

Separating your application this way is sometimes called a **multi-tier** or **n-tier architecture**. In some cases, the tiers (layers) are isolated in completely separate projects, but other times it just refers to how the

classes are organized and used. The important thing is thinking about how to split your application into manageable pieces, and avoid having controllers or bloated classes that try to do everything.

For this project, you'll use two application layers: a **presentation layer** made up of the controllers and views that interact with the user, and a **service layer** that contains business logic and database code. The presentation layer already exists, so the next step is to build a service that handles to-do business logic and saves to-do items to a database.

> Most larger projects use a 3-tier architecture: a presentation layer, a service logic layer, and a data repository layer. A **repository** is a class that's only focused on database code (no business logic). In this application, you'll combine these into a single service layer for simplicity, but feel free to experiment with different ways of architecting the code.

Create an interface

The C# language includes 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). You'll use an interface to 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<TodoItem[]> GetIncompleteItemsAsync();
    }
}
```

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 found (are you
missing a using directive or an assembly 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<TodoItem[]>`.

> If this syntax looks confusing, think: "a Task that contains an array 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.)

Create the service class

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, so for now you'll just fake it and always return two hard-coded items:

Services/FakeTodoItemService.cs

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

namespace AspNetCoreTodo.Services
{
    public class FakeTodoItemService : ITodoItemService
    {
        public Task<TodoItem[]> GetIncompleteItemsAsync()
        {
            var item1 = new TodoItem
            {
                Title = "Learn ASP.NET Core",
                DueAt = DateTimeOffset.Now.AddDays(1)
            };

            var item2 = new TodoItem
            {
                Title = "Build awesome apps",
                DueAt = DateTimeOffset.Now.AddDays(2)
            };

            return Task.FromResult(new[] { item1, item2 });
        }
    }
}
```

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 _todoItemService;

    public TodoController(ITodoItemService todoItemService)
    {
        _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* 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 can use it. This makes it really easy to test parts of your application separately. I'll cover testing in detail 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 items = await _todoItemService.GetIncompleteItemsAsync();

    // ...
}
```

Remember that the `GetIncompleteItemsAsync` method returned a `Task<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 items = await _todoItemService.GetIncompleteItemsAsync();

    // 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 services)
{
    // (... 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, FakeTodoItemService>();
```

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 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 items = await _todoItemService.GetIncompleteItemsAsync();

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

    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 type `dotnet run` in the terminal. If the code compiles without errors, the server will start 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 (for now).

Although it's possible to go directly to `http://localhost:5000/todo`, it would be nicer to add an item called **My to-dos** to the navbar. To do this, you can edit the shared layout file.

Update the layout

The layout file at `Views/Shared/_Layout.cshtml` contains the "base" HTML for each view. This includes the navbar, which is rendered at the top of each page.

To add a new item to the navbar, find the HTML code for the existing navbar items:

Views/Shared/_Layout.cshtml

```
<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>
```

Add your own item that points to the `Todo` controller instead of `Home`:

```
<li>
    <a asp-controller="Todo" asp-action="Index">My to-dos</a>
</li>
```

The `asp-controller` and `asp-action` attributes on the `<a>` element are called **tag helpers**. Before the view is rendered, ASP.NET Core replaces these tag helpers with real HTML attributes. In this case, a URL to the `/Todo/Index` route is generated and added to the `<a>` element

as an `href` attribute. This means you don't have to hard-code the route to the `TodoController`. Instead, ASP.NET Core generates it for you automatically.

> If you've used Razor in ASP.NET 4.x, you'll notice some syntax changes. Instead of using `@Html.ActionLink()` to generate a link to an action, tag helpers are now the recommended way to create links in your views. Tag helpers are useful for forms, too (you'll see why in a later chapter). You can learn about other tag helpers in the documentation at https://docs.asp.net.