

ASP.NET Core: Model-View-Controller (MVC) Pattern

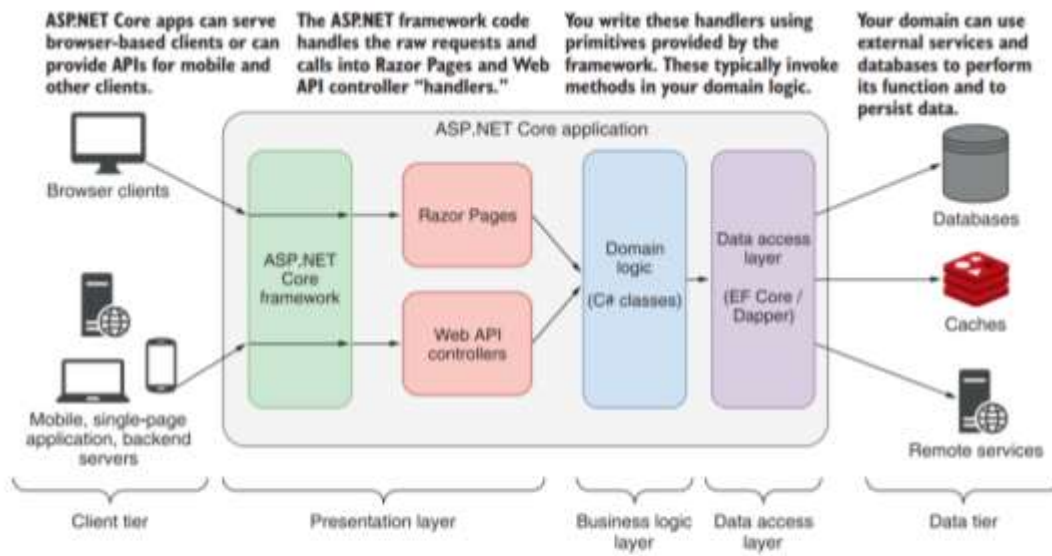


Figure 1.1 A typical ASP.NET Core application consists of several layers. The ASP.NET Core framework code handles requests from a client, dealing with the complex networking code. The framework then calls into handlers (Razor Pages and Web API controllers) that you write using primitives provided by the framework. Finally, these handlers call into your application's domain logic, which are typically C# classes and objects without any ASP.NET Core-specific dependencies.

Some of the key infrastructure improvements include:

- Middleware "pipeline" for defining your application's behavior
- Built-in support for dependency injection
- Combined UI (MVC) and API (Web API) infrastructure
- Highly extensible configuration system
- Scalable for cloud platforms by default using asynchronous programming

The largest obstacle likely to come across: **no longer support for Web Forms or WCF server.**

Getting an application up and running typically involves four basic steps:

1 Generate—Create the base application from a template to get started.

2 Restore—Restore all the packages and dependencies to the local project folder

using NuGet.

3 Build—Compile the application and generate all the necessary assets.

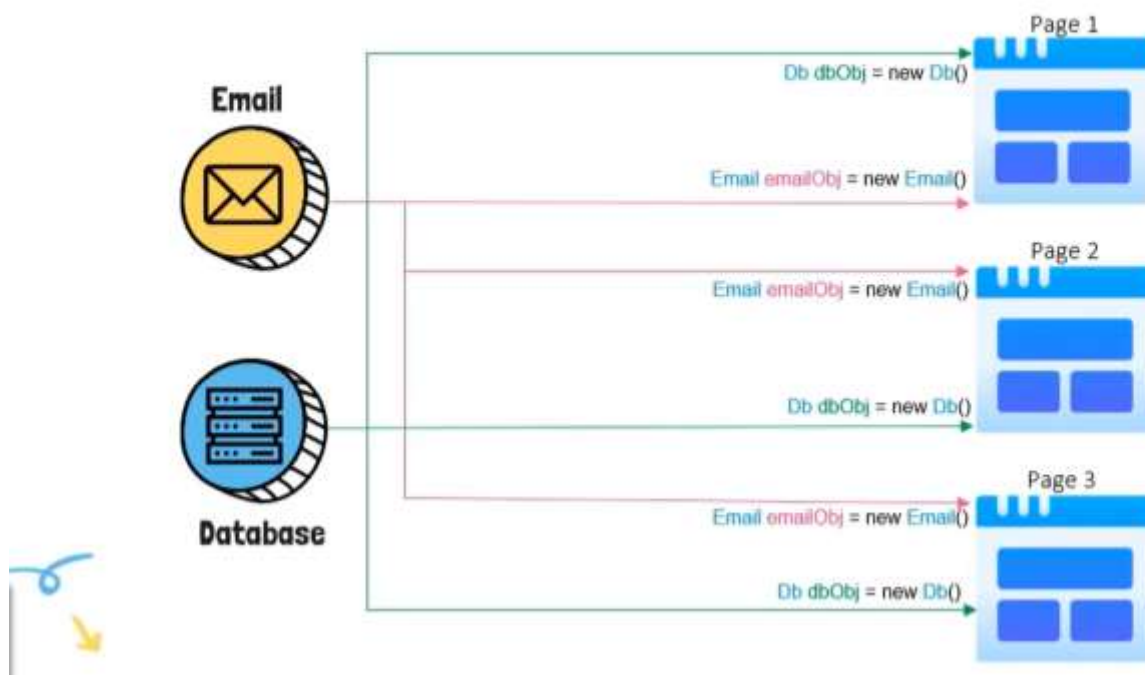
4 Run—Run the compiled application.

Visual Studio and the .NET CLI include many ASP.NET Core templates for building different types of applications. For example,

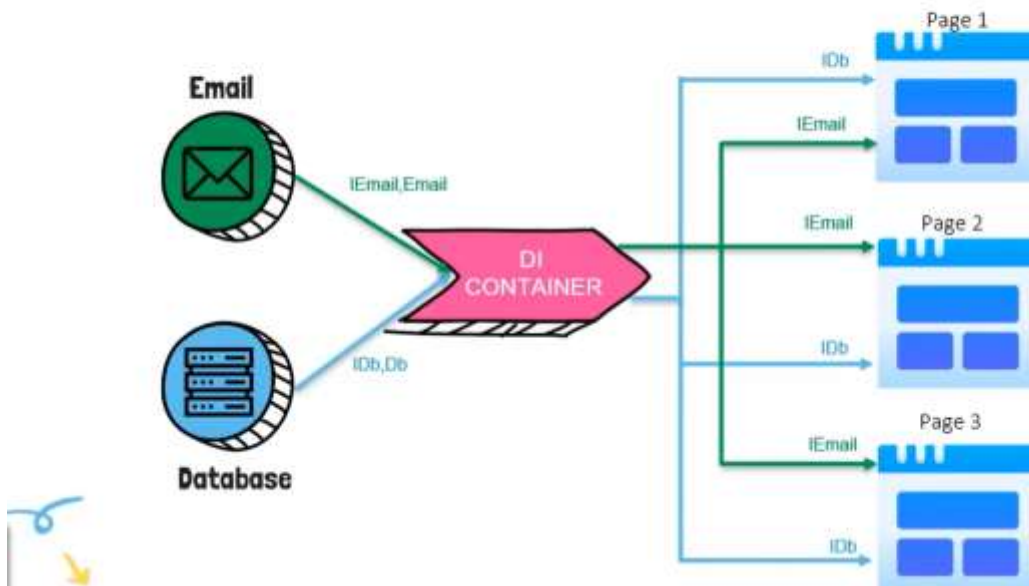
a. Razor Pages web application—Razor Pages applications generate HTML on the server and are designed to be viewed by users in a web browser directly.

b. MVC (Model-View-Controller) application—MVC applications are similar to Razor Pages apps in that they generate HTML on the server and are designed to be viewed by users directly in a web browser. They use traditional MVC controllers instead of Razor Pages.

WITHOUT DEPENDENCY INJECTION



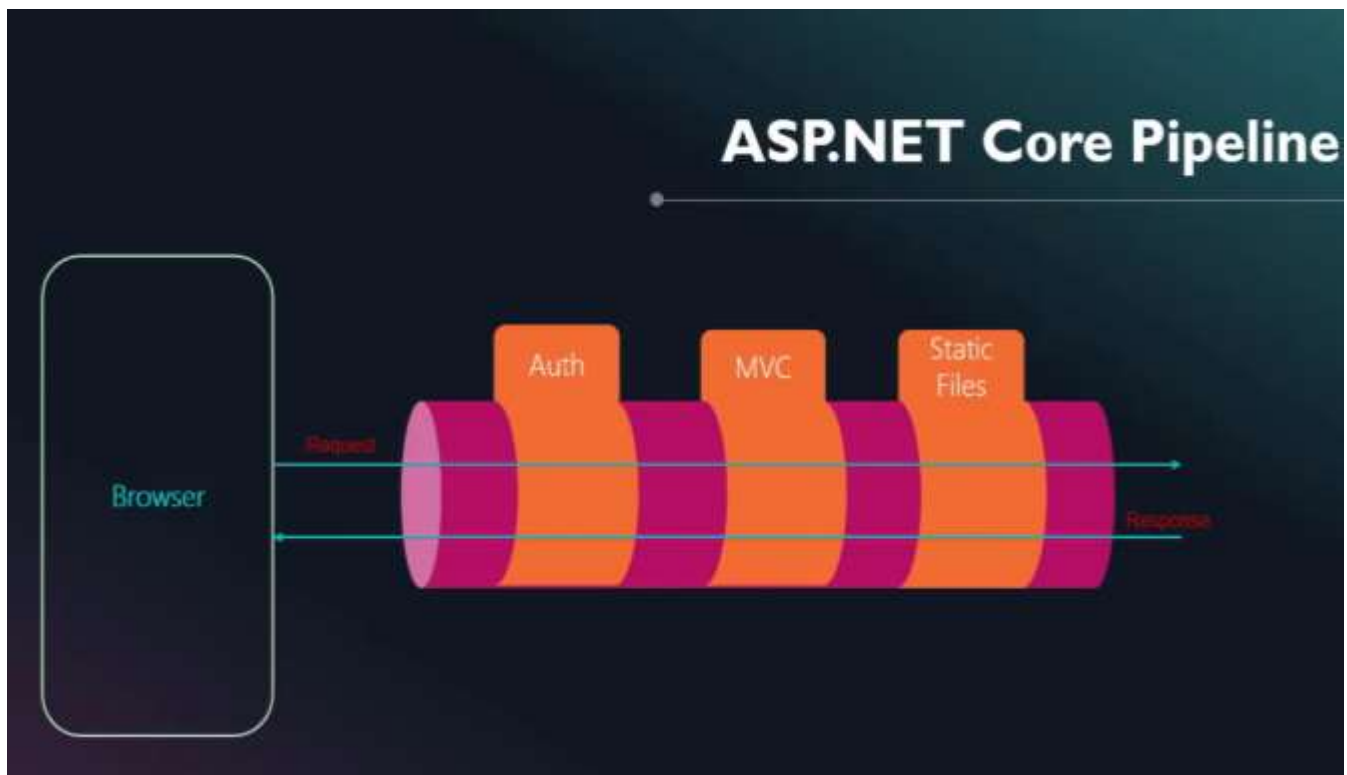
WITH DEPENDENCY INJECTION



WHAT IS DEPENDENCY INJECTION ?

- Dependency Injection (DI) is a software pattern.
- Dependency Injection is basically providing the objects that an object needs, instead of having it construct the objects themselves.
- DI is a technique whereby one object supplies the dependencies of another object.
- With the help of DI, we can write loosely coupled code.

Kestrel is responsible for receiving the request data and constructing a C# representation of the request, but it doesn't attempt to generate a response directly. For that, Kestrel hands the HttpContext to the middleware pipeline found in every ASP.NET Core application. This is a series of components that process the incoming request to perform common operations such as logging, handling exceptions, or serving static files.



At the end of the middleware pipeline is the endpoint middleware. This middleware is responsible for calling the code that generates the final response. In most applications that will be an MVC or Razor Pages block.

Understanding Middleware and the Request Pipeline

The purpose of the ASP.NET Core platform is to receive HTTP requests and send responses to them, which ASP.NET Core delegates to middleware components. Middleware components are arranged in a chain, known as the request pipeline.

When a new HTTP request arrives it goes through the middleware components in the chain which inspect and modify it in some way if needed. Once the request has made its way through the pipeline, the ASP.NET Core platform sends the response.

Some components focus on generating responses for requests, but others are there to provide supporting features, such as formatting specific data types or reading and writing cookies.

If no response is generated by the middleware components, then ASP.NET Core will return a response with the HTTP 404 Not Found status code.


```

public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddRazorPages();
    }
    public void Configure(IApplicationBuilder app)
    {
        app.UseWelcomePage("/");
        app.UseExceptionHandler("/Error");
        app.UseStaticFiles();
        app.UseRouting();
        app.UseEndpoints(endpoints =>
        {
            endpoints.MapRazorPages();
        });
    }
}

```

← WelcomePageMiddleware handles all requests to the "/" path and returns a sample HTML response.

← Requests to "/" will never reach the endpoint middleware.

Even though you know the endpoint middleware can also handle the "/" path, `WelcomePageMiddleware` is earlier in the pipeline, so it returns a response when it receives the request to "/", short-circuiting the pipeline, as shown in figure 3.13. None of the other middleware in the pipeline runs for the request, so none has an opportunity to generate a response.

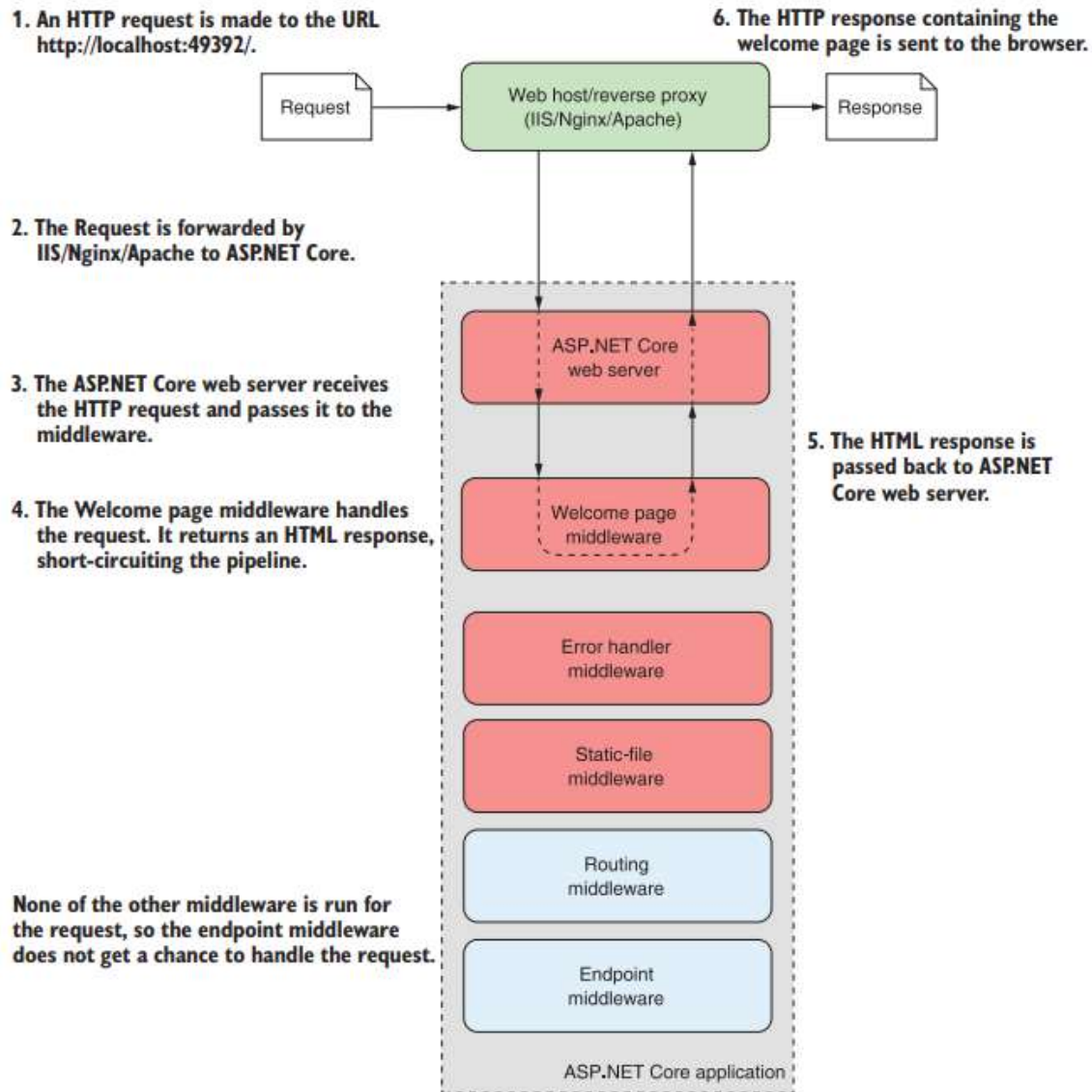


Figure 3.13 Overview of the application handling a request to the `/` path. The welcome page middleware is first in the middleware pipeline, so it receives the request before any other middleware. It generates an HTML response, short-circuiting the pipeline. No other middleware runs for the request.

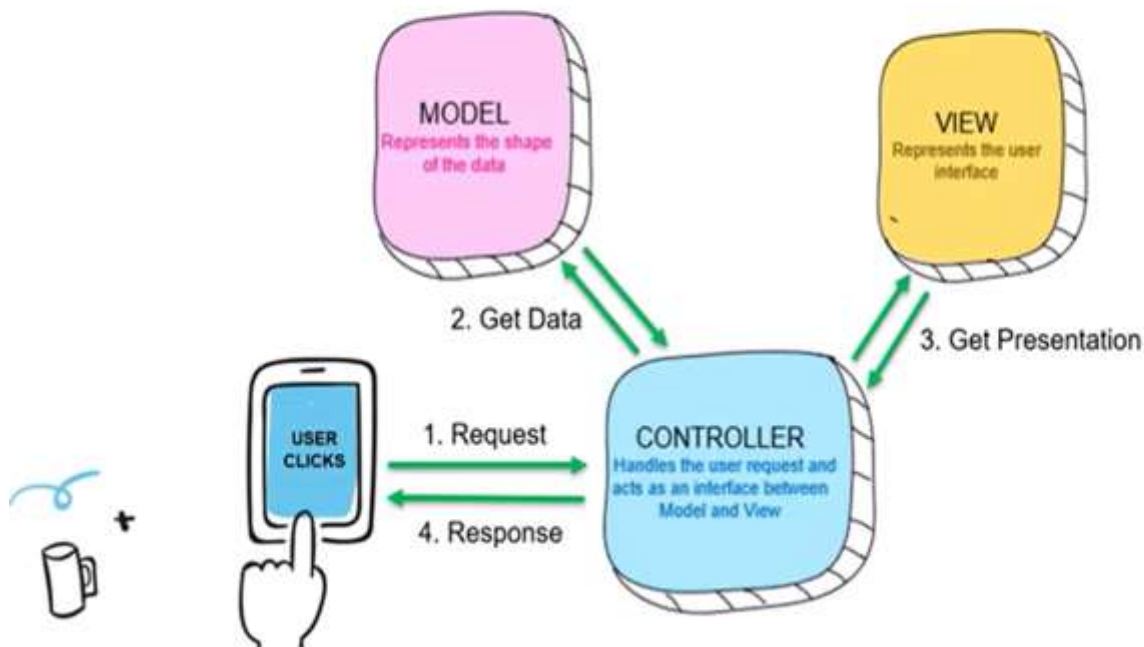
In ASP.NET Core web applications, your middleware pipeline will normally include the `EndpointMiddleware`. This is typically where you write the bulk of your application logic, calling various other classes in your app. It also serves as the main entry point for users to interact with your app. It typically takes one of three forms:

1. **An HTML web application** designed for direct use by users—If the application is consumed directly by users, as in a traditional web application, then Razor Pages is responsible for generating the web pages that the user interacts with. It handles requests for URLs, it receives data posted using forms, and it generates the HTML that users use to view and navigate your app.
2. **An API** designed for consumption by another machine or in code—The other main possibility for a web application is to serve as an API to backend server processes, to a mobile app, or to a client framework for building single-page applications (SPAs). In this case, your application serves data in machine-readable formats such as JSON or XML instead of the human-focused HTML output.
3. **Both an HTML web application and an API**—It is also possible to have applications that serve both needs. This can let you cater to a wider range of clients while sharing logic in your application.

For Razor Pages, the entry point is a page handler that resides in a Razor Page's PageModel. A page handler is a method that runs in response to a request.

By default, the path of a Razor Page on disk controls the URL path that the Razor Page responds to. For example, a request to the URL `/products/list` corresponds to the Razor Page at the path `pages/Products/List.cshtml`. Razor Pages can contain any number of page handlers, but only one runs in response to a given request.

MVC ARCHITECTURE



ROUTING IN MVC

The URL pattern for routing is considered after the domain name.

- <https://localhost:5555/Category/Index/3>
- <https://localhost:5555/{controller}/{action}/{id}>

URL	Controller	Action	Id
https://localhost:5555/Category/Index	Category	Index	Null
https://localhost:5555/Category	Category	Index	Null
https://localhost:5555/Category/Edit/3	Category	Edit	3
https://localhost:5555/Product/Details/3	Product	Details	3



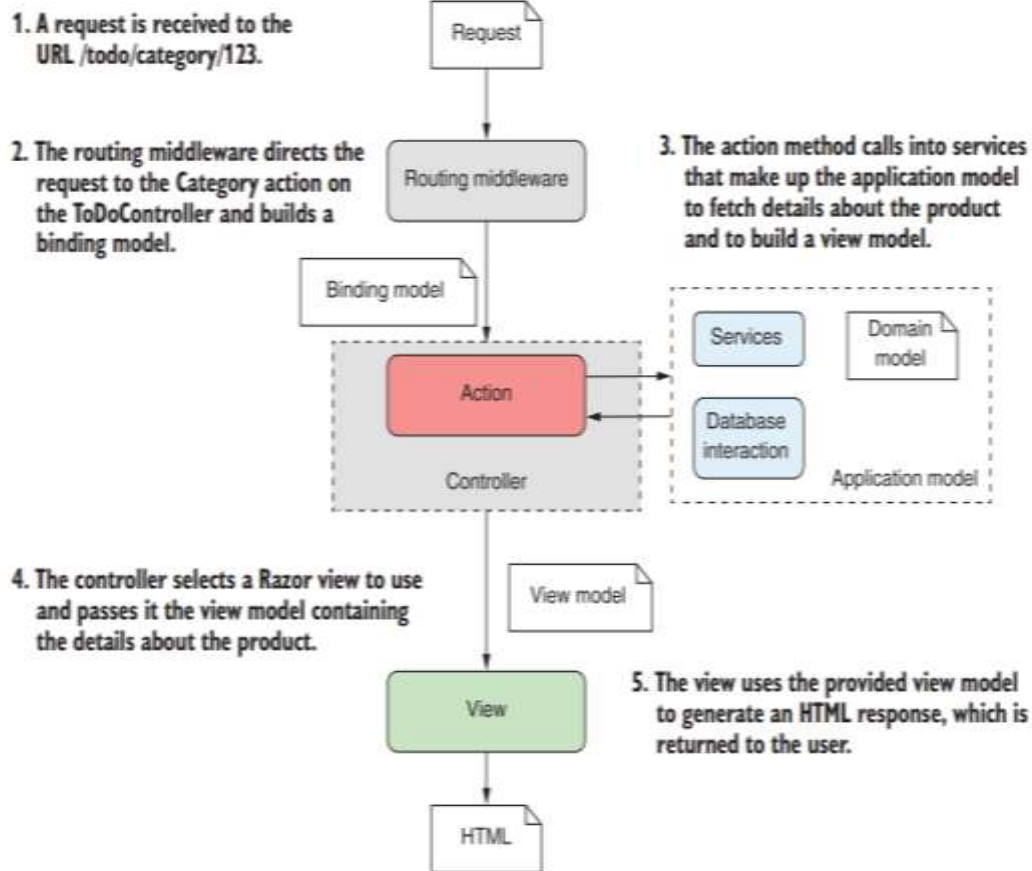


Figure 4.10 A complete MVC controller request for a category. The MVC controller pattern is almost identical to that of Razor Pages, shown in figure 4.6. The controller is equivalent to a Razor Page, and the action is equivalent to a page handler.

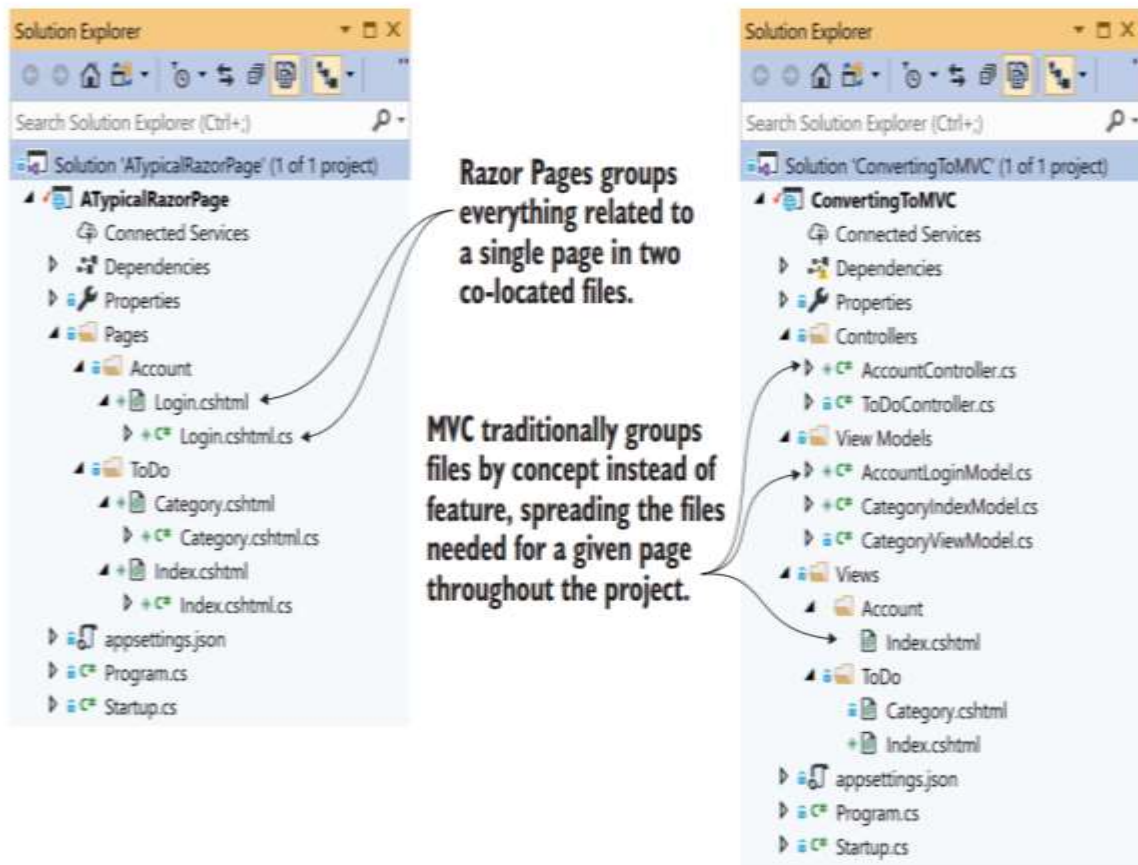


Figure 4.11 Comparing the folder structure for an MVC project to the folder structure for a Razor Pages project

Model or View Model

This defines the shape of the entity or data and via EF and Linq query the controller fetches the data from the database tables as per the need, which is later passed onto the views by action method of the controller.

What is a Controller?

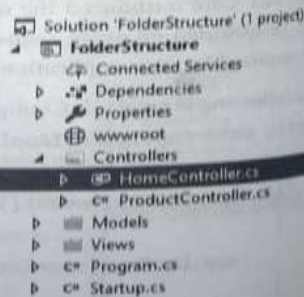
The Controller acts as the middleman - it will combine your **Model** with a **View** and serve the result to the end-user. However, neither a Model nor a View is required - the Controller can act on its own for the most basic operations, e.g. delivering a simple text message or redirecting the user to somewhere else.

In ASP.NET MVC, a Controller is just like any other class, so it has a `.cs` file extension (or `.vb` if you use Visual Basic) and looks like any other .NET class. However, there are a few things that will allow you (and the .NET framework) to recognize it as an MVC Controller:

- It's usually placed in a folder called "Controllers" in the root of your project
- It inherits from `Microsoft.AspNetCore.Mvc.Controller` (or from one of your own classes which then inherits the `Microsoft.AspNetCore.Mvc.Controller` class)
- The name of the class will usually end with the word Controller, e.g. "HomeController" or "ProductsController"

If you don't follow these conventions, the .NET framework will not be able to recognize your class as a Controller, so it makes sense to follow them. However, if you insist on e.g. naming your Controller classes differently, you can decorate it with the `[Controller]` attribute, placed right before the class declaration.

By inheriting the `Microsoft.AspNetCore.Mvc.Controller` class, you get some added functionality that you can use for MVC purposes, e.g. the ability to return Views/Partial Views. It also allows your Controller class to access HTTP related information like the query string. In other words, it turns a regular .NET class into a web-aware class, allowing you to do stuff you would normally be able to do in your PHP or ASP Classic file or any of the many other web technologies out there.



The Controller Responsibilities

The Controller has three major responsibilities

- Handle the Request
- Build a Model
- Send the Response

Handle the Request

The Controller is responsible to Handle Request from the User.

Builds the Module

The Controller Action method executes the application logic and builds a model

Sends the Result

Finally, it should return the Result in HTML/File/Json/XML or whatever the format requested by the user.

Actions

Since a Controller is just a regular .NET class, it can have fields, properties and methods. Especially the methods of a Controller class are interesting, because they are the connection between the browser (and therefore the user) and your application. For that reason, the **methods of a Controller class is referred to as actions** - a method usually corresponds to an action in your application, which then returns something to the browser/user.

Since a browser works by calling a URL, you need something that translates URL's to a corresponding Controller and action (method). For instance, the browser might request a URL like */products/1/* and then you want your *ProductsController* to handle this request with a method/action called *Details*. This is done with the concept of Routing, just know that **Routing is what connects URL's to actions on your Controllers**.

Now the Edit action can only be accessed with a GET request. This has the added benefit of allowing you to have multiple methods with the same name, as long as they don't accept the same request method. So for instance, you could have two methods called Edit: The first one would be available for GET requests and generate the FORM for editing an item, while the second one would only be available for POST requests and be used to update the item when the FORM was posted back to the server. It could look like this:

```
[HttpGet]
public IActionResult Edit()
{
    return View();
}

[HttpPost]
public IActionResult Edit(Product product)
{
    product.Save();
    return Content("Product updated!");
}
```

Now whenever a request is made to the Edit() method/action, the actual method responding to the request will be based on whether it's a GET or a POST request.

Action Result Types

Actions are the methods defined on a Controller. Actions are invoked when a requested URL is matched to an Action on a Controller (this is done by the Routing system). When the Action (method) finishes its work, it will usually return something to the client and that something will usually be implementing the **IActionResult** interface (or **Task<IActionResult>** if the method is asynchronous).

As soon as your Controller inherits the built-in base Controller class (*Microsoft.AspNetCore.Mvc.Controller*), we are supplied with a wide range of helper methods for generating the response (the action result). The **View()** method. It will take the matching View file and turn it into an instance of the **ViewResult** class, which implements the **IActionResult** interface. After that, ASP.NET MVC will automatically transform it into a response to the browser - in this case in the form of an "OK" status code and then a body of HTML.

A view is far from the only possible result of a Controller action, though. Since the result will eventually be returned to a browser, the available methods should cover all possible outcomes of a HTTP request, so a Controller action can of course also result in a redirect, a 404 (Page not Found) or any of the other HTTP status codes. Here's an incomplete list of the most interesting and useful methods for generating an Action result:

- **Content()** - returns the specified string as plain text to the client (usually a browser)
- **View()** - returns a View to the client
- **PartialView()** - returns a Partial View to the client

- **File()** - returns the content of a specified file to the client
- **Json()** - returns a JSON response to the client
- **Redirect()** and **RedirectPermanent()** - returns a redirect response to the browser (temporary or permanent), redirecting the user to another URL
- **StatusCode()** - returns a custom status code to the client

For instance, the **Content()** method simply generates an instance of the **ContentResult** class and fills the **Content** property with the value you pass into the method.

Since there are many types of responses which implements the **IActionResult** interface, your actions can return any of them, based on the logic of your method. A common use case for this is to return either a **View** or a piece of content if the requested content is found, or a 404 (Page not Found) error if its not found. It could look like this:

```
public IActionResult Details(int id)
{
    Product product = GetProduct(id);
    if (product != null)
        return View(product);
    return NotFound();
}
```

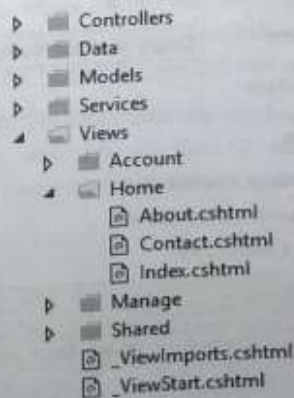
In the first line, we try to load the requested product, using a to-be-implemented **GetProduct()** method. If the product is found, we return it inside of a **View** using the **View()** method - if not, we return a 404 error using the **NotFound()** helper method, which basically just creates an instance of the **NotFoundResult** class. So in this case, there are two possible outcomes of the action, but there could be even more, if needed - you are free to return anything that implements the **IActionResult** interface.

RENDERING HTML WITH VIEWS

In the Model-View-Controller (MVC) pattern, the *view* handles the app's data presentation and user interaction. A view is an HTML template with embedded Razor markup. Razor markup is code that interacts with HTML markup to produce a webpage that's sent to the client.

In ASP.NET Core MVC, views are *.cshtml* files that use the C# programming language in Razor markup. Usually, view files are grouped into folders named for each of the app's controllers. The folders are stored in a *Views* folder at the root of the app:

The *Home* controller is represented by a *Home* folder inside the *Views* folder. The *Home* folder contains the



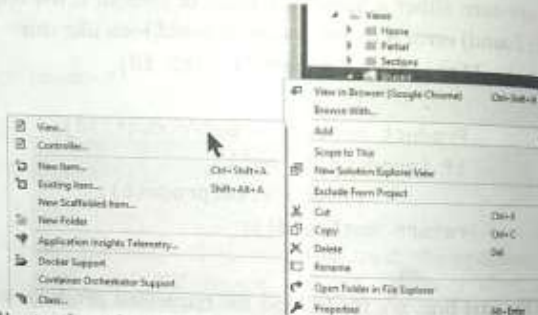
views for the *About*, *Contact*, and *Index* (homepage) webpages. When a user requests one of these three webpages, controller actions in the *Home* controller determine which of the three views is used to build and return a webpage to the user.

Use layouts to provide consistent webpage sections and reduce code repetition. Layouts often contain the header, navigation and menu elements, and the footer. The header and footer usually contain boilerplate markup for many metadata elements and links to script and style assets. Layouts help you avoid this boilerplate markup in your views.

Partial views reduce code duplication by managing reusable parts of views.

Partial Views in ASP.NET MVC usually follow these conventions:

- They are located in the same folder as the View using it, or if multiple views are using it (the most common situation) they should be placed in a "Shared" folder inside of your Views folder
- Its filename usually starts with an underscore, simply to indicate that this is not a regular/full View. With that in place, let's add the Partial View. We'll add it to our *Shared* folder, to ensure that all of our Views can access it easily:



Benefits of using Views

Views help to establish separation of concerns (SoC) within an MVC app by separating the user interface markup from other parts of the app. Following SoC design makes your app modular, which provides several benefits:

- The app is easier to maintain because it's better organized. Views are generally grouped by app feature. This makes it easier to find related views when working on a feature.
- The parts of the app are loosely coupled. You can build and update the app's views separately from the business logic and data access components. You can modify the views of the app without necessarily having to update other parts of the app.
- It's easier to test the user interface parts of the app because the views are separate units.
- Due to better organization, it's less likely that you'll accidentally repeat sections of the user interface.

When to choose MVC controllers over Razor Pages

1. When you don't want to render views—Razor Pages are best for page-based applications, where you're rendering a view for the user. If you're building a Web API, you should use MVC controllers instead.

2. When you're converting an existing MVC application to ASP.NET Core—If you already have an ASP.NET application that uses MVC, it's probably not worth converting your existing MVC controllers to Razor Pages. It makes more sense to keep your existing code, and perhaps to look at doing new development in the application with Razor Pages.
3. When you're doing a lot of partial page updates—It's possible to use JavaScript in an application to avoid doing full page navigations by only updating part of the page at a time. This approach, halfway between fully server-side rendered and a client-side application may be easier to achieve with MVC controllers than Razor Pages.

Summary

1. The MVC design pattern allows for a separation of concerns between the business logic of your application, the data that's passed around, and the display of data in a response.
2. Razor Pages are built on the ASP.NET Core MVC framework, and they use many of the same primitives. They use conventions and a different project layout to optimize for page-based scenarios.
3. MVC controllers contain multiple action methods, typically grouped around a high-level entity. Razor Pages groups all the page handlers for a single page in one place, grouping around a page/feature instead of an entity.
4. Each Razor Page is equivalent to a mini controller focused on a single page, and each Razor Page handler corresponds to a separate action method.
5. Razor Pages should inherit from the PageModel base class.
6. A single Razor Page handler is selected based on the incoming request's URL, the HTTP verb, and the request's query string, in a process called routing.
7. Page handlers should generally delegate to services to handle the business logic required by a request, instead of performing the

changes themselves. This ensures a clean separation of concerns that aids testing and improves application structure.

8. Page handlers can have parameters whose values are taken from properties of the incoming request in a process called model binding. Properties decorated with `[BindProperty]` can also be bound to the request.
9. By default, properties decorated with `[BindProperty]` are not bound for GET requests. To enable binding, use `[BindProperty(SupportsGet = true)]`.
10. Page handlers can return a `PageResult` or `void` to generate an HTML response.
11. You can send users to a new Razor Page using a `RedirectToPageResult`.
12. The `PageModel` base class exposes many helper methods for creating an `ActionResult`.