

# Building Websites Using ASP.NET Core Razor Pages

This section is about building websites with a modern HTTP architecture on the server side using Microsoft ASP.NET Core. You will learn about building simple websites using the ASP.NET Core Razor Pages feature introduced with .NET Core 2.0 and the Razor class library feature introduced with .NET Core 2.1.

This section will cover the following topics:

- Understanding web development
- Understanding ASP.NET Core
- Exploring Razor Pages
- Using Entity Framework Core with ASP.NET Core
- Using Razor class libraries

# Understanding web development

Developing for the web is developing with **Hypertext Transfer Protocol (HTTP)**.

## Understanding HTTP

To communicate with a web server, the client, also known as the **user agent**, makes calls over the network using HTTP. As such, HTTP is the technical underpinning of the **web**. So, when we talk about web applications or web services, we mean that they use HTTP to communicate between a client (often a web browser) and a server.

A client makes an HTTP request for a resource, such as a page, uniquely identified by a **Uniform Resource Locator (URL)**, and the server sends back an HTTP response, as shown in the following diagram:

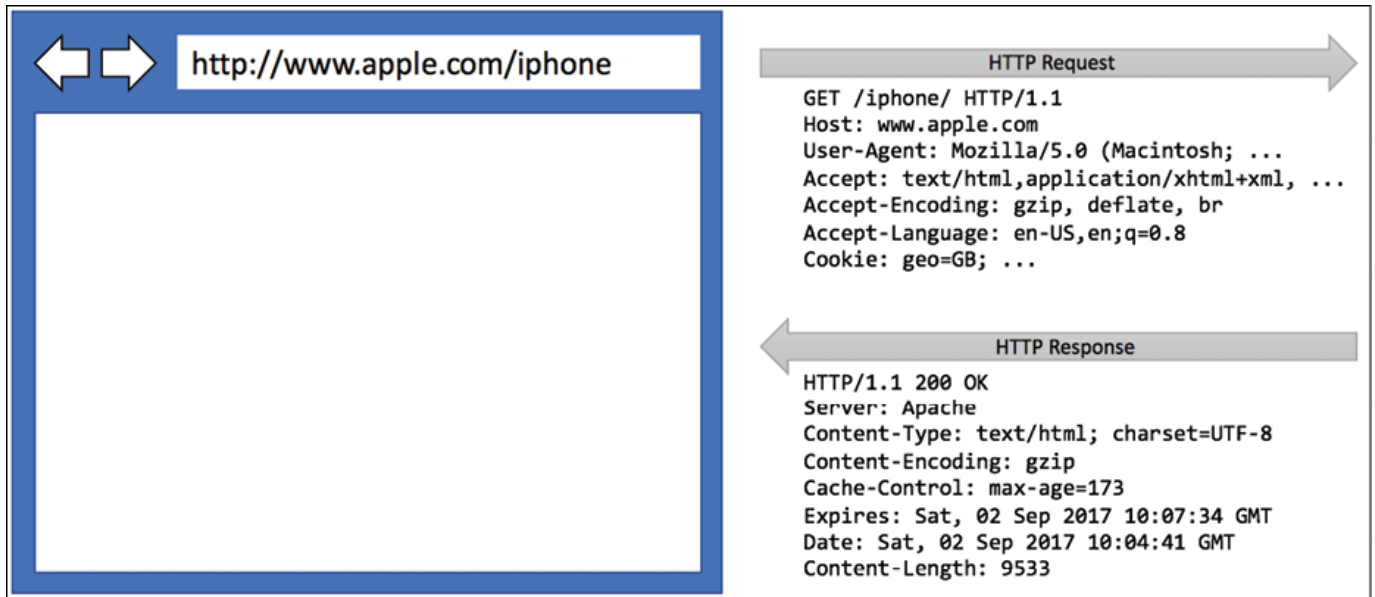


Figure 7.1: An HTTP request and response

You can use Google Chrome and other browsers to record requests and responses.

**Good Practice:** Google Chrome is available on more operating systems than any other browser, and it has powerful, built-in developer tools, so it is a good first choice of browser for testing your websites. Always test your web application with Chrome and at least two other browsers, for example, Firefox and Safari for macOS and iPhone. Microsoft Edge switched from using Microsoft's own rendering engine to using Chromium in 2019 so it is less important to test with it. If Microsoft's Internet Explorer is used at all, it tends to mostly be inside organizations for intranets.

Let's explore how to use Google Chrome to make HTTP requests:

1. Start **Google Chrome**.
2. To show developer tools in Chrome, do the following:
  - On macOS, press **Alt + Cmd + I**
  - On Windows, press **F12** or **Ctrl + Shift + I**
3. Click on the **Network** tab, and Chrome should immediately start recording the network traffic between your browser and any web servers, as shown in the following screenshot:

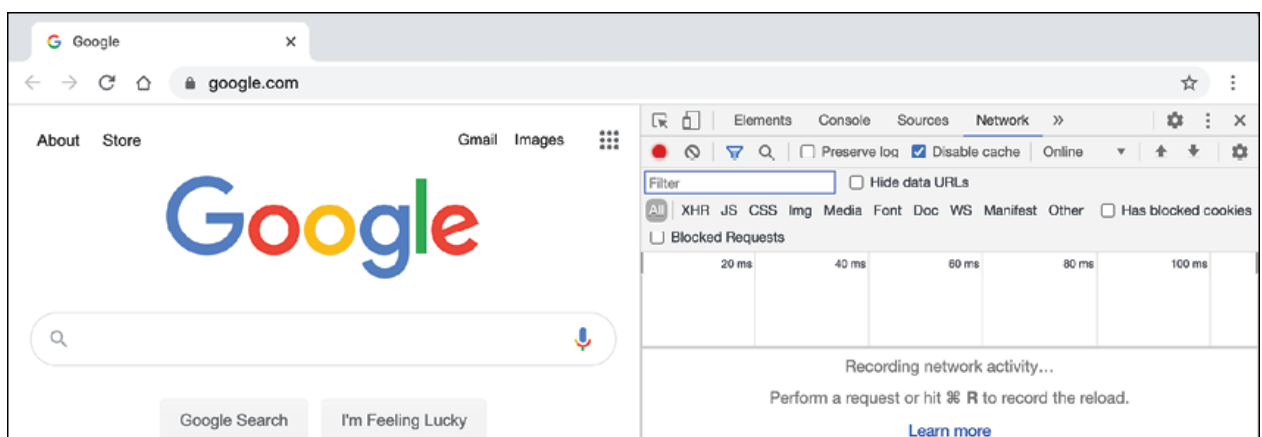


Figure 7.2: Chrome recording network traffic

- In Chrome's address box, enter the following URL: <https://dotnet.microsoft.com/learn/aspnet>.
- In the **Developer tools** window, in the list of recorded requests, scroll to the top and click on the first entry, the **document**, as shown in the following screenshot:

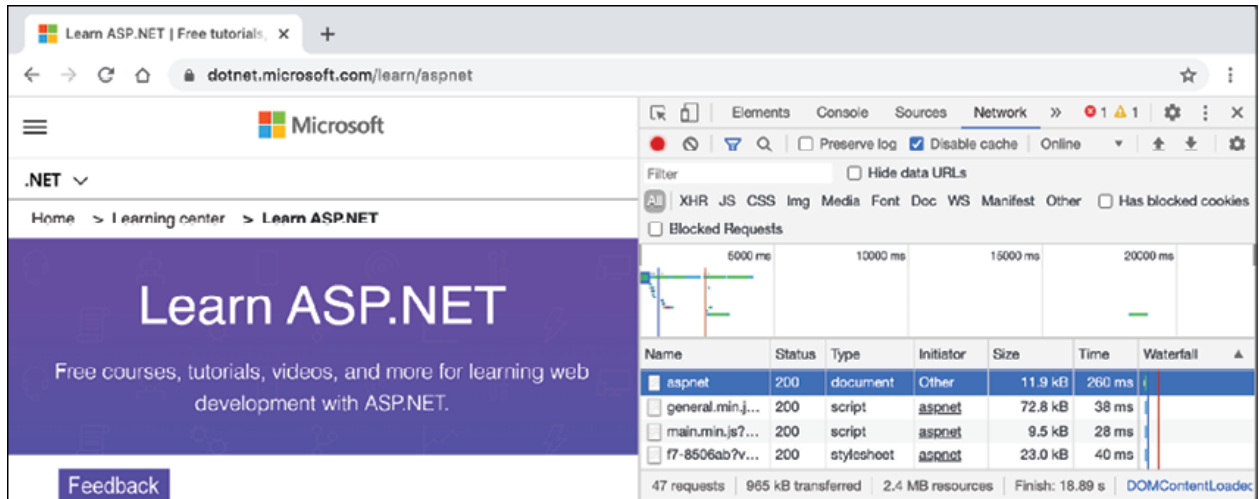


Figure 7.3: Recorded requests

- On the right-hand side, click on the **Headers** tab, and you will see details about the request and the response, as shown in the following screenshot:

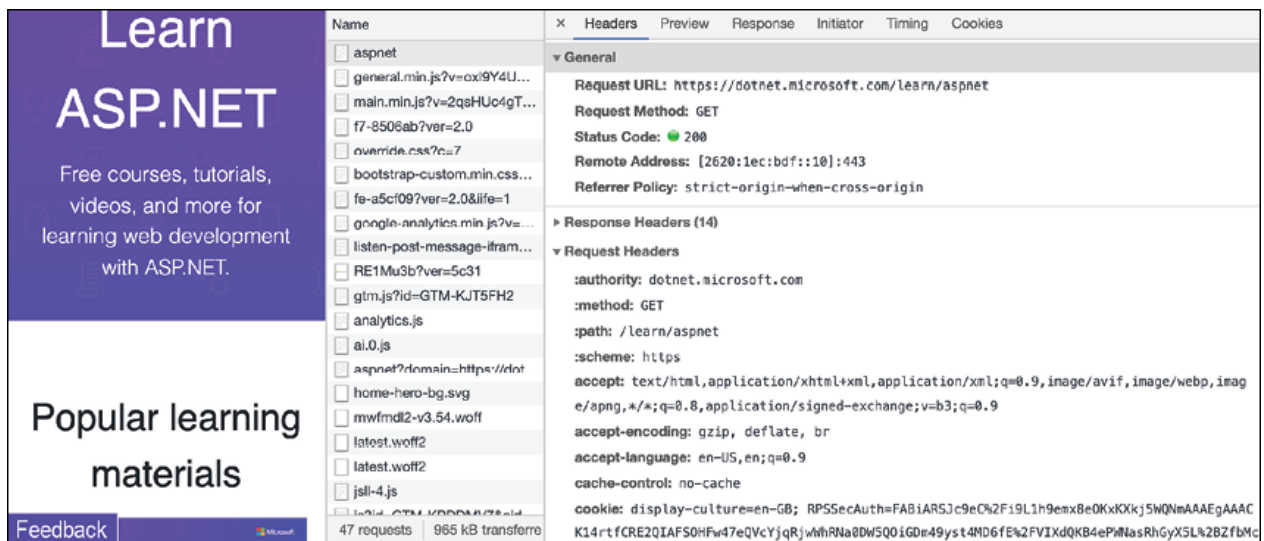


Figure 7.4: Request and response details

Note the following aspects:

- Request Method** is GET. Other methods that HTTP defines include POST, PUT, DELETE, HEAD, and PATCH.
- Status Code** is 200 OK. This means that the server found the resource that the browser requested and has returned it in the body of the response. Other status codes that you might see in response to a GET request include 301 Moved Permanently, 400 Bad Request, 401 Unauthorized, and 404 Not Found.
- Request Headers** sent by the browser to the web server include:
  - accept**, which lists what formats the browser accepts. In this case, the browser is saying it understands HTML, XHTML, XML, and some image formats, but it will accept all other files \*/\*. Default weightings, also known as quality values, are 1.0. XML is specified with a quality value of 0.9 so it is preferred less than HTML or XHTML. All other file types are given a quality value of 0.8 so are least preferred.
  - accept-encoding**, which lists what compression algorithms the browser understands. In this case, GZIP, DEFLATE, and Brotli.
  - accept-language**, which lists the human languages it would prefer the content to use. In this case, US English, which has a default quality value of 1.0, and then any dialect of English that has an explicitly specified quality value of 0.9.
- Response Headers**, **content-encoding** tells me the server has sent back the HTML web page response compressed using the GZIP algorithm because it knows that the client can decompress that format.

- Close Chrome.

## Client-side web development

When building websites, a developer needs to know more than just C# and .NET Core. On the client (that is, in the web browser), you will use a combination of the following technologies:

- **HTML5:** This is used for the content and structure of a web page.
- **CSS3:** This is used for the styles applied to elements on the web page.
- **JavaScript:** This is used to code any business logic needed on the web page, for example, validating form input or making calls to a web service to fetch more data needed by the web page.

Although HTML5, CSS3, and JavaScript are the fundamental components of frontend web development, there are many additional technologies that can make frontend web development more productive, including Bootstrap, the world's most popular frontend open source toolkit, and CSS preprocessors like SASS and LESS for styling, Microsoft's TypeScript language for writing more robust code, and JavaScript libraries like jQuery, Angular, React, and Vue. All these higher-level technologies ultimately translate or compile to the underlying three core technologies, so they work across all modern browsers.

As part of the build and deploy process, you will likely use technologies like Node.js; **Node Package Manager (NPM)** and Yarn, which are both client-side package managers; and Webpack, which is a popular module bundler, a tool for compiling, transforming, and bundling website source files.

# Understanding ASP.NET Core

Microsoft ASP.NET Core is part of a history of Microsoft technologies used to build websites and web services that have evolved over the years:

- **Active Server Pages (ASP)** was released in 1996 and was Microsoft's first attempt at a platform for dynamic server-side execution of website code. ASP files contain a mix of HTML and code that executes on the server written in the VBScript language.
- **ASP.NET Web Forms** was released in 2002 with the .NET Framework, and is designed to enable non-web developers, such as those familiar with Visual Basic, to quickly create websites by dragging and dropping visual components and writing event-driven code in Visual Basic or C#. Web Forms can only be hosted on Windows, but it is still used today in products such as Microsoft SharePoint. It should be avoided for new web projects in favor of ASP.NET Core.
- **Windows Communication Foundation (WCF)** was released in 2006 and enables developers to build SOAP and REST services. SOAP is powerful but complex, so it should be avoided unless you need advanced features, such as distributed transactions and complex messaging topologies.
- **ASP.NET MVC** was released in 2009 and is designed to cleanly separate the concerns of web developers between the *models*, which temporarily store the data; the *views* that present the data using various formats in the UI; and the *controllers*, which fetch the model and pass it to a view. This separation enables improved reuse and unit testing.
- **ASP.NET Web API** was released in 2012 and enables developers to create HTTP services, also known as REST services that are simpler and more scalable than SOAP services.
- **ASP.NET SignalR** was released in 2013 and enables real-time communication in websites by abstracting underlying technologies and techniques, such as WebSockets and Long Polling. This enables website features like live chat or updates to time-sensitive data like stock prices across a wide variety of web browsers even when they do not support an underlying technology like WebSockets.
- **ASP.NET Core** was released in 2016 and combines MVC, Web API, and SignalR, running on .NET Core. Therefore, it can execute cross-platform. ASP.NET Core has many project templates to get you started with its supported technologies.

**Good Practice:** Choose ASP.NET Core to develop websites and web services because it includes web-related technologies that are modern and cross-platform.

ASP.NET Core 2.0 to 2.2 can run on .NET Framework 4.6.1 or later (Windows only) as well as .NET Core 2.0 or later (cross-platform). ASP.NET Core 3.0 only supports .NET Core 3.0. ASP.NET Core 5 only supports .NET 5.

## Classic ASP.NET versus modern ASP.NET Core

Until now, ASP.NET has been built on top of a large assembly in the .NET Framework named `System.Web.dll` and it is tightly coupled to Microsoft's Windows-only web server named **Internet Information Services (IIS)**. Over the years, this assembly has accumulated a lot of features, many of which are not suitable for modern cross-platform development.

ASP.NET Core is a major redesign of ASP.NET. It removes the dependency on the `System.Web.dll` assembly and IIS and is composed of modular lightweight packages, just like the rest of .NET Core.

You can develop and run ASP.NET Core applications cross-platform on Windows, macOS, and Linux. Microsoft has even created a cross-platform, super-performant web server named **Kestrel**, and the entire stack is open source.

**More Information:** You can read more about Kestrel, including its HTTP/2 support, at the following link:  
<https://docs.microsoft.com/en-us/aspnet/core/fundamentals/servers/kestrel>

ASP.NET Core 2.2 or later projects default to the new in-process hosting model. This gives a 400% performance improvement when hosting in Microsoft IIS, but Microsoft still recommends using Kestrel for even better performance.

## Creating an ASP.NET Core project

We will create an ASP.NET Core project that will show a list of suppliers from the `Northwind` database.

The **dotnet** tool has many project templates that do a lot of work for you, but it can be difficult to discern which work best in a given situation, so we will start with the simplest web template and slowly add features step by step so that you can understand all the pieces:

1. In your existing `PracticalApps` folder, create a subfolder named `NorthwindWeb`, and add it to the `PracticalApps` workspace.
2. Navigate to **Terminal | New Terminal** and select `NorthwindWeb`.
3. In **TERMINAL**, enter the following command to create an **ASP.NET Core Empty** website: `dotnet new web`
4. In **TERMINAL**, enter the following command to restore packages and compile the website: `dotnet build`
5. Edit `NorthwindWeb.csproj`, and note the SDK is `Microsoft.NET.Sdk.Web`, as shown in the following markup:

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>
</Project>
```

In ASP.NET Core 1.0, you would need to include lots of package references. With ASP.NET Core 2.0, you would need a package reference named `Microsoft.AspNetCore.All`. With ASP.NET Core 3.0 and later, simply using this Web SDK is enough.

6. Open `Program.cs`, and note the following:
  - A website is like a console application, with a `Main` method as its entry point.
  - A website has a `CreateHostBuilder` method that creates a host for the website using defaults for a web host which is then built and run and specifies a `Startup` class that is used to further configure the website, as shown in the following code:

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }
}
```

```

}
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.UseStartup<Startup>();
        });
}

```

7. Open Startup.cs, and note its two methods:

- The `ConfigureServices` method is currently empty. We will use it later to add services like Razor Pages and a database context for working with the Northwind database.
- The `Configure` method sets up the HTTP request pipeline and currently does three things: first, it configures that when developing, any unhandled exceptions will be shown in the browser window for the developer to see its details; second, it uses routing; and third, it uses endpoints to wait for requests, and then for each HTTP GET request it asynchronously responds by returning the plain text "Hello World!", as shown in the following code:

```

public class Startup
{
    // This method gets called by the runtime.
    // Use this method to add services to the container.
    public void ConfigureServices(IServiceCollection services)
    {
    }
    // This method gets called by the runtime.
    // Use this method to configure the HTTP request pipeline.
    public void Configure(
        IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }
        app.UseRouting();
        app.UseEndpoints(endpoints =>
        {
            endpoints.MapGet("/", async context =>
            {
                await context.Response.WriteAsync("Hello World!");
            });
        });
    }
}

```

8. Close the Startup.cs class file.

How does ASP.NET Core know when we are running in development mode so that the `IsDevelopment` method returns `true`? Let's find out.

## Testing and securing the website

We will now test the functionality of the ASP.NET Core Empty website project. We will also enable encryption of all traffic between the browser and web server for privacy by switching from HTTP to HTTPS. HTTPS is the secure encrypted version of HTTP.

1. In **TERMINAL**, enter the `dotnet run` command, and note the web server has started listening on ports 5000 and 5001 and the hosting environment is Development, as shown in the following output:

```

info: Microsoft.Hosting.Lifetime[0]
      Now listening on: https://localhost:5001
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: http://localhost:5000
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: /Users/Code/PracticalApps/NorthwindWeb

```

2. Start Chrome.

3. Enter the address `http://localhost:5000/`, and note the response is Hello World! in plain text, from the cross-platform Kestrel web server, as shown in the following screenshot:

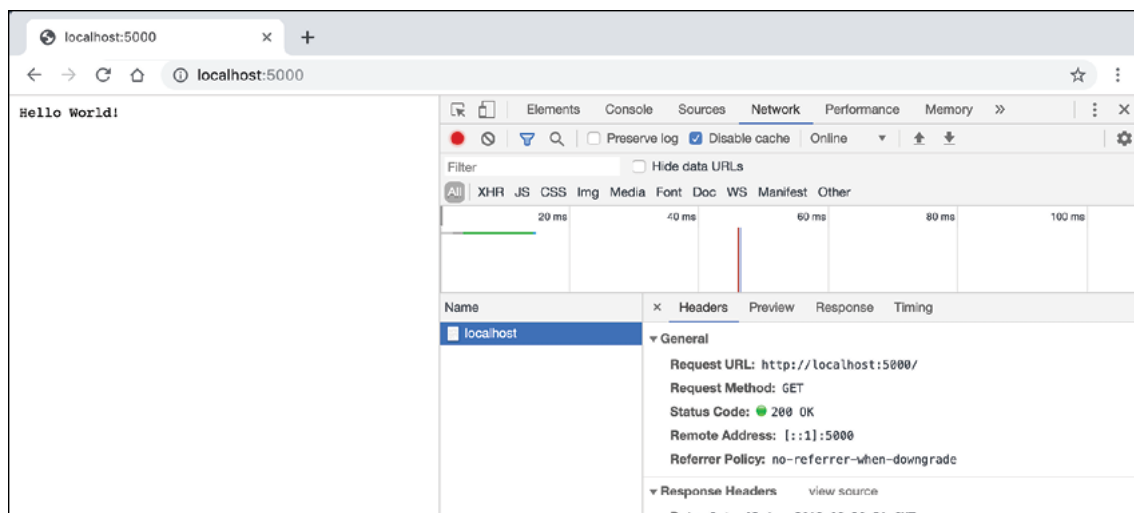


Figure 7.5: Plain text response from `http://localhost:5000/`

4. Enter the address `https://localhost:5001/`, and note the response is a privacy error, as shown in the following screenshot:

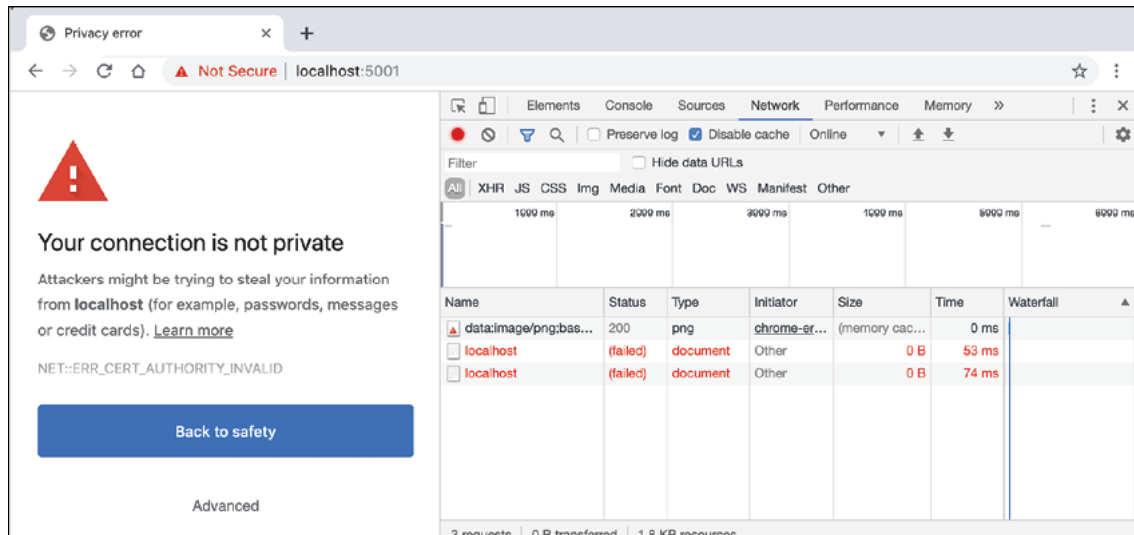


Figure 7.6: Privacy error showing SSL encryption has not been enabled with a certificate

This is because we have not configured a certificate that the browser can trust to encrypt and decrypt HTTPS traffic (so if you do not see this error, it is because you have already configured a certificate). In a production environment, you would want to pay a company like Verisign for one because they provide liability protection and technical support.

**More Information:** If you use a Linux variant that cannot create self-signed certificates or you do not mind reapplying for a new certificate every 90 days, then you can get a free certificate from the following link: <https://letsencrypt.org>

During development, you can tell your OS to trust a temporary development certificate provided by ASP.NET Core.

5. In **TERMINAL**, press `Ctrl + c` to stop the web server.
6. In **TERMINAL**, enter the `dotnet dev-certs https --trust` command, and note the message, **Trusting the HTTPS development certificate was requested**. You might be prompted to enter your password and a valid HTTPS certificate may already be present.
7. If Chrome is still running, close and restart it to ensure it has read the new certificate.
8. In `Startup.cs`, in the `Configure` method, add an `else` statement to enable HSTS when not in development, as shown highlighted in the following code:

```
if (env.IsDevelopment())
{
    app.UseDeveloperExceptionPage();
}
else
{
    app.UseHsts();
}
```

**HTTP Strict Transport Security (HSTS)** is an opt-in security enhancement. If a website specifies it and a browser supports it, then it forces all communication over HTTPS and prevents the visitor from using untrusted or invalid certificates.

9. Add a statement after the call to `app.UseRouting` to redirect HTTP requests to HTTPS, as shown in the following code:

```
app.UseHttpsRedirection();
```

10. In **TERMINAL**, enter the `dotnet run` command to start the web server.
11. In Chrome, request the address `http://localhost:5000/`, and note how the server responds with a 307 Temporary Redirect to port 5001, and that the certificate is now valid and trusted, as shown in the following screenshot:

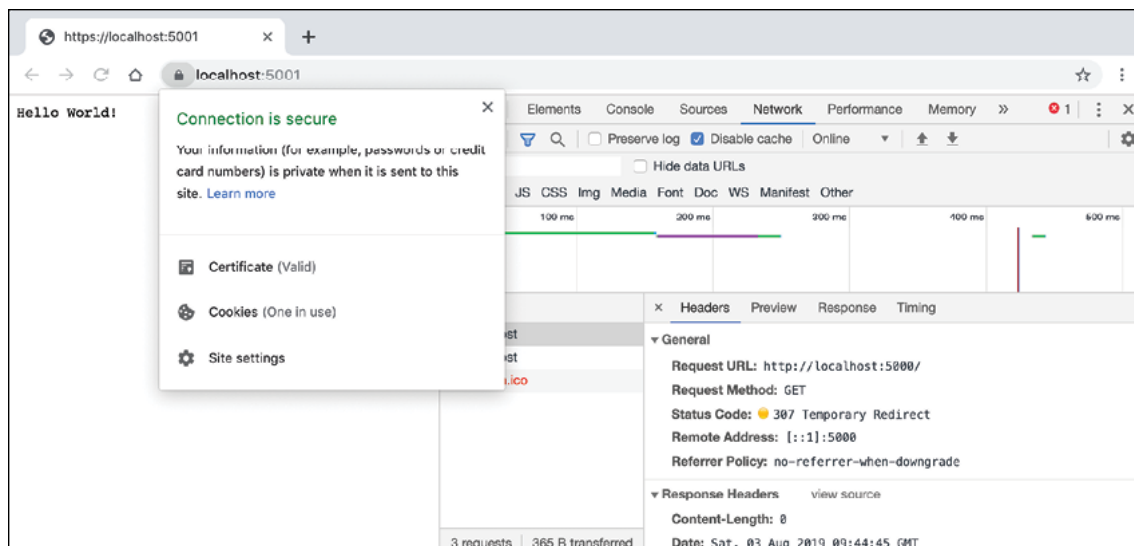


Figure 7.7: The connection is now secured using a valid certificate



12. Close Chrome.
13. In **TERMINAL**, press `ctrl + c` to stop the web server.

Remember to stop the Kestrel web server whenever you have finished testing a website.

## Controlling the hosting environment

ASP.NET Core can read from environment variables to determine what hosting environment to use, for example, `DOTNET_ENVIRONMENT` or `ASPNETCORE_ENVIRONMENT` when the `ConfigureWebHostDefaults` method is called, as it is in `Program.cs` in this project.

You can override these settings during local development:

1. In the `NorthwindWeb` folder, expand the folder named `Properties`, open the file named `launchSettings.json`, and note the profile named `NorthwindWeb` that sets the hosting environment to `Development`, as shown highlighted in the following configuration:

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:56111",
      "sslPort": 44329
    }
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "NorthwindWeb": {
      "commandName": "Project",
      "launchBrowser": true,
      "applicationUrl": "https://localhost:5001;http://localhost:5000",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

2. Change the environment to `Production`.
3. In **TERMINAL**, start the website using the `dotnet run` command and note the hosting environment is `Production`, as shown in the following output:

```
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Production
```

4. In **TERMINAL**, press `ctrl + c` to stop the website.
5. In `launchSettings.json`, change the environment back to `Development`.

**More Information:** You can learn more about working with ASP.NET Core hosting environments at the following link: <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/environments>

## Enabling static and default files

A website that only ever returns a single plain text message isn't very useful!

At a minimum, it ought to return static HTML pages, CSS that the web pages will use for styling, and any other static resources such as images and videos.

You will now create a folder for your static website resources and a basic index page that uses Bootstrap for styling:

**More Information:** Web technologies like Bootstrap commonly use a **Content Delivery Network (CDN)** to efficiently deliver their source files globally. You can read more about CDNs at the following link: [https://en.wikipedia.org/wiki/Content\\_delivery\\_network](https://en.wikipedia.org/wiki/Content_delivery_network)

1. In the `NorthwindWeb` folder, create a folder named `wwwroot`.
2. Add a new file to the `wwwroot` folder named `index.html`.
3. Modify its content to link to CDN-hosted Bootstrap for styling, and use modern good practices such as setting the viewport, as shown in the following markup:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <!-- Required meta tags -->
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no" />
  <!-- Bootstrap CSS -->
  <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.css" integrity="sha384-JcKb8q3iqJ61gNV9KGb8t" />
  <title>Welcome ASP.NET Core!</title>
</head>
<body>
  <div class="container">
    <div class="jumbotron">
      <h1 class="display-3">Welcome to Northwind!</h1>
      <p class="lead">We supply products to our customers.</p>
      <hr />
      <h2>This is a static HTML page.</h2>
      <p>Our customers include restaurants, hotels, and cruise lines.</p>
      <p>
        <a class="btn btn-primary" href="#">Get products</a>
      </p>
    </div>
  </div>
</body>
</html>
```



```
        href="https://www.asp.net/">Learn more</a>
    </p>
</div>
</div>
</body>
</html>
```

**More Information:** To get the latest `<link>` element for Bootstrap, copy and paste it from the *Getting Started - Introduction* page in the documentation at the following link: <https://getbootstrap.com/>

If you were to start the website now, and enter `http://localhost:5000/index.html` in the address box, the website would return a 404 Not Found error saying no web page was found. To enable the website to return static files such as `index.html`, we must explicitly configure that feature.

Even if we enable static files, if you were to start the website and enter `http://localhost:5000/` in the address box, the website would return a 404 Not Found error because the web server doesn't know what to return by default if no named file is requested.

You will now enable static files, explicitly configure default files, and change the URL path registered that returns Hello World:

1. In `Startup.cs`, in the `Configure` method, modify the statement that maps a GET request to returning the Hello World! plain text response to only respond to the URL path `/hello`, and add statements to enable static files and default files, as shown highlighted in the following code:

```
public void Configure(
    IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseHsts();
    }
    app.UseRouting();
    app.UseHttpsRedirection();
    app.UseDefaultFiles(); // index.html, default.html, and so on
    app.UseStaticFiles();
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapGet("/hello", async context =>
        {
            await context.Response.WriteAsync("Hello World!");
        });
    });
}
```

The call to `UseDefaultFiles` must be before the call to `UseStaticFiles`, or it won't work!

2. Start the website by entering `dotnet run` in **TERMINAL**.
3. In Chrome, enter `http://localhost:5000/`, and note that you are redirected to the HTTPS address on port 5001, and the `index.html` file is now returned because it is one of the possible default files for this website.
4. In Chrome, enter `http://localhost:5000/hello`, and note that it returns the plain text Hello World! as before.

If all web pages are static, that is, they only get changed manually by a web editor, then our website programming work is complete. But almost all websites need dynamic content, which means a web page that is generated at runtime by executing code.

The easiest way to do that is to use a feature of ASP.NET Core named **Razor Pages**.

# Exploring Razor Pages

Razor Pages allow a developer to easily mix HTML markup with C# code statements. That is why they use the `.cshtml` file extension.

By default, ASP.NET Core looks for Razor Pages in a folder named `Pages`.

## Enabling Razor Pages

You will now change the static HTML page into a dynamic Razor Page, and then add and enable the Razor Pages service:

1. In the `NorthwindWeb` project, create a folder named `Pages`.
2. Copy the `index.html` file into the `Pages` folder.
3. Rename the file extension from `.html` to `.cshtml`.
4. Remove the `<h2>` element that says that this is a static HTML page.
5. In `Startup.cs`, in the `ConfigureServices` method, add statements to add Razor Pages and its related services like model binding, authorization, anti-forgery, views, and tag helpers, as shown highlighted in the following code:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages();
}
```

6. In `Startup.cs`, in the `Configure` method, in the configuration to use endpoints, add a statement to use `MapRazorPages`, as shown highlighted in the following code:

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapRazorPages();
    endpoints.MapGet("/hello", async context =>
    {
        await context.Response.WriteAsync("Hello World!");
    });
});
```

## Defining a Razor Page

In the HTML markup of a web page, Razor syntax is indicated by the `@` symbol.

Razor Pages can be described as follows:

- They require the `@page` directive at the top of the file.
- They can have an `@functions` section that defines any of the following:
  - Properties for storing data values, like in a class definition. An instance of that class is automatically instantiated named `Model` that can have its properties set in special methods and you can get the property values in the markup.
  - Methods named `OnGet`, `OnPost`, `OnDelete`, and so on, that execute when HTTP requests are made, such as `GET`, `POST`, and `DELETE`.

Let's now convert the static HTML page into a Razor page:

1. In Visual Studio Code, open `index.cshtml`.
2. Add the `@page` statement to the top of the file.
3. After the `@page` statement, add an `@functions` statement block.
4. Define a property to store the name of the current day as a `string` value.
5. Define a method to set `DayName` that executes when an HTTP `GET` request is made for the page, as shown in the following code:

```
@page
@functions
{
    public string DayName { get; set; }
    public void OnGet()
    {
        Model.DayName = DateTime.Now.ToString("dddd");
    }
}
```

6. Output the day name inside one of the paragraphs, as shown in the following markup:

```
<p>It's @Model.DayName! Our customers include restaurants, hotels, and cruise lines.</p>
```

7. Start the website, visit it with Chrome as you did before by navigating to the URL `http://localhost:5000/`, and note the current day name is output on the page, as shown in the following screenshot:

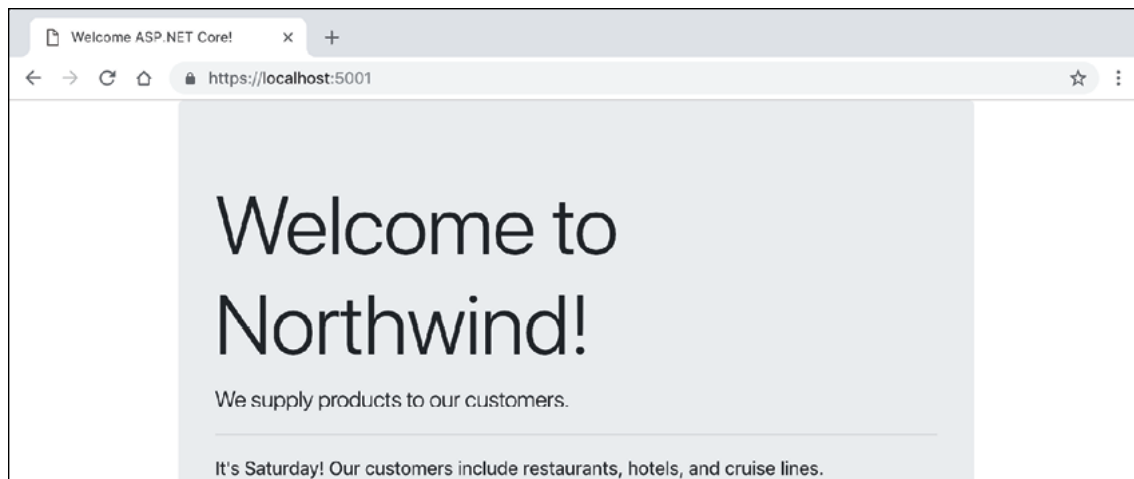


Figure 7.8: Welcome to Northwind!

8. In Chrome, enter `http://localhost:5000/index.html`, which exactly matches the static filename, and note that it returns the static HTML page as before.
9. Close Chrome and stop the web server by pressing `Ctrl + C` in **TERMINAL**.

## Using shared layouts with Razor Pages

Most websites have more than one page. If every page had to contain all of the boilerplate markup that is currently in `index.cshtml`, that would become a pain to manage. So, ASP.NET Core has **layouts**.

To use layouts, we must create a Razor file to define the default layout for all Razor Pages (and all MVC views) and store it in a `Shared` folder so that it can be easily found by convention. The name of this file can be anything, but `_Layout.cshtml` is good practice. We must also create a specially named file to set the default layout for all Razor Pages (and all MVC views). This file must be named `_ViewStart.cshtml`:

1. In the `Pages` folder, create a file named `_ViewStart.cshtml`.
2. Modify its content, as shown in the following markup:

```
@{
    Layout = "_Layout";
}
```

3. In the `Pages` folder, create a folder named `Shared`.
4. In the `Shared` folder, create a file named `_Layout.cshtml`.
5. Modify the content of `_Layout.cshtml` (it is similar to `index.cshtml` so you can copy and paste the HTML markup from there), as shown in the following markup:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <!-- Required meta tags -->
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no" />
  <!-- Bootstrap CSS -->
  <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.css" integrity="sha384-JcKb8q3iqJ61gNV9KGb8t" />
  <title>@ViewData["Title"]</title>
</head>
<body>
  <div class="container">
    @RenderBody()
    <hr />
    <footer>
      <p>Copyright © 2020 - @ViewData["Title"]</p>
    </footer>
  </div>
  <!-- JavaScript to enable features like carousel -->
  <!-- jQuery first, then Popper.js, then Bootstrap JS -->
  <script src="https://code.jquery.com/jquery-3.5.1.slim.min.js" integrity="sha384-DfXdz2htPH0lsSSs5nCTpuj/zy4C+OGpamoFVy38MVBnE+IbbVYUew+OrCXaRkfj" crossorigin="anonymous"></script>
  <script src="https://cdn.jsdelivr.net/npm/popper.js@1.14.7/umd/popper.min.js" integrity="sha384-U02eT0CpQdSjQ6hJty5KVphtPhzWj9W0ic1" />
  <script src="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/js/bootstrap.min.js" integrity="sha384-JjSmVgyd0p3pXB1rRibZUAYoIIy60Rq6VrjIEaF" />
  @RenderSection("Scripts", required: false)
</body>
</html>
```

While reviewing the preceding markup, note the following:

- `<title>` is set dynamically using server-side code from a dictionary named `viewData`. This is a simple way to pass data between different parts of an ASP.NET Core website. In this case, the data will be set in a Razor Page class file and then output in the shared layout.
  - `@RenderBody()` marks the insertion point for the page being requested.
  - A horizontal rule and footer will appear at the bottom of each page.
  - At the bottom of the layout are some scripts to implement some cool features of Bootstrap that we can use later like a carousel of images.
  - After the `<script>` elements for Bootstrap, we have defined a section named `Scripts` so that a Razor Page can optionally inject additional scripts that it needs.
6. Modify `index.cshtml` to remove all HTML markup except `<div class="jumbotron">` and its contents, and leave the C# code in the `@functions` block that you added earlier.

7. Add a statement to the `OnGet` method to store a page title in the `ViewData` dictionary, and modify the button to navigate to a suppliers page (which we will create in the next section), as shown highlighted in the following markup:

```
@page
@functions
{
    public string DayName { get; set; }
    public void OnGet()
    {
        ViewData["Title"] = "Northwind Website";
        Model.DayName = DateTime.Now.ToString("dddd");
    }
}
<div class="jumbotron">
<h1 class="display-3">Welcome to Northwind!</h1>
<p class="lead">We supply products to our customers.</p>
<hr />
<p>It's @Model.DayName! Our customers include restaurants, hotels, and cruise lines.</p>
<p>
<a class="btn btn-primary" href="suppliers">
    Learn more about our suppliers</a>
</p>
</div>
```

8. Start the website, visit it with Chrome, and note that it has similar behavior as before, although clicking the button for suppliers will give a 404 Not Found error because we have not created that page yet.

## Using code-behind files with Razor Pages

Sometimes, it is better to separate the HTML markup from the data and executable code, so Razor Pages allows **code-behind** class files.

You will now create a page that shows a list of suppliers. In this example, we are focusing on learning about code-behind files. In the next topic, we will load the list of suppliers from a database, but for now, we will simulate that with a hardcoded array of `string` values:

1. In the `Pages` folder, add two new files named `suppliers.cshtml` and `suppliers.cshtml.cs`.
2. Add statements to `suppliers.cshtml.cs`, as shown in the following code:

```
using Microsoft.AspNetCore.Mvc.RazorPages;
using System.Collections.Generic;
namespace NorthwindWeb.Pages
{
    public class SuppliersModel : PageModel
    {
        public IEnumerable<string> Suppliers { get; set; }
        public void OnGet()
        {
            ViewData["Title"] = "Northwind Web Site - Suppliers";
            Suppliers = new[] {
                "Alpha Co", "Beta Limited", "Gamma Corp"
            };
        }
    }
}
```

While reviewing the preceding markup, note the following:

- `SuppliersModel` inherits from `PageModel`, so it has members such as the `ViewData` dictionary for sharing data. You can click on `PageModel` and press `F12` to see that it has lots more useful features, like the entire `HttpContext` of the current request.
- `SuppliersModel` defines a property for storing a collection of `string` values named `Suppliers`.
- When an HTTP `GET` request is made for this Razor Page, the `Suppliers` property is populated with some example supplier names.

3. Modify the contents of `suppliers.cshtml`, as shown in the following markup:

```
@page
@model NorthwindWeb.Pages.SuppliersModel
<div class="row">
<h1 class="display-2">Suppliers</h1>
<table class="table">
<thead class="thead-inverse">
<tr><th>Company Name</th></tr>
</thead>
<tbody>
@foreach(string name in Model.Suppliers)
{
    <tr><td>@name</td></tr>
}
</tbody>
</table>
</div>
```

While reviewing the preceding markup, note the following:

- The model type for this Razor Page is set to `SuppliersModel`.
  - The page outputs an HTML table with Bootstrap styles.
  - The data rows in the table are generated by looping through the `Suppliers` property of `Model`.
4. Start the website, visit it using Chrome, click on the button to learn more about suppliers, and note the table of suppliers, as shown in the following screenshot:

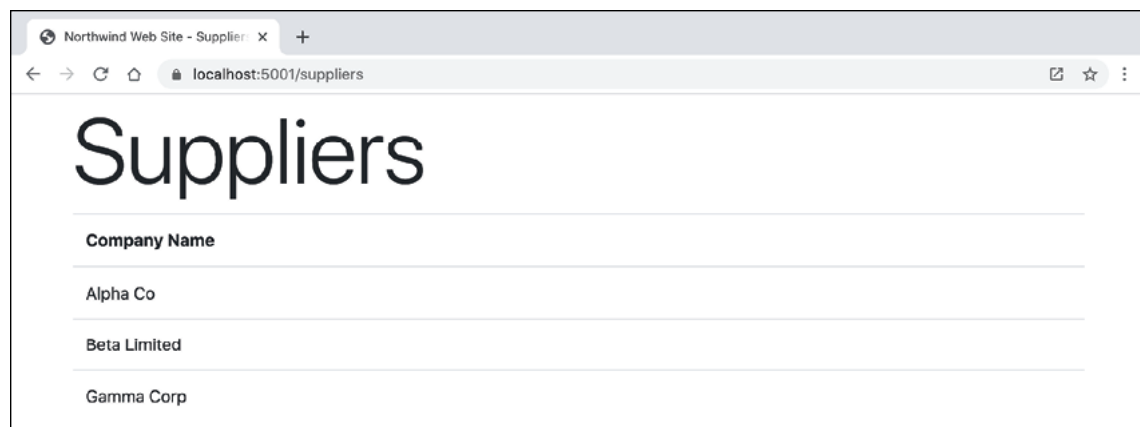


Figure 7.9: The temporary list of suppliers

# Using Entity Framework Core with ASP.NET Core

Entity Framework Core is a natural way to get real data into a website. In *Section 6, Introducing Practical Applications of C# and .NET*, you created two class libraries: one for the entity models and one for the Northwind database context.

## Configure Entity Framework Core as a service

Functionality like Entity Framework Core database contexts that are needed by ASP.NET Core must be registered as a service during website startup:

1. In the NorthwindWeb project, modify NorthwindWeb.csproj to add a reference to the NorthwindContextLib project, as shown highlighted in the following markup:

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <ProjectReference Include=
      "..\NorthwindContextLib\NorthwindContextLib.csproj" />
  </ItemGroup>
</Project>
```

2. In **TERMINAL**, restore packages and compile the project by entering the following command: `dotnet build`
3. Open Startup.cs and import the System.IO, Microsoft.EntityFrameworkCore, and Packt.Shared namespaces, as shown in the following code:

```
using System.IO;
using Microsoft.EntityFrameworkCore;
using Packt.Shared;
```

4. Add statements to the ConfigureServices method to register the Northwind database context class to use SQLite as its database provider and specify its database connection string, as shown in the following code:

```
string databasePath = Path.Combine("...", "Northwind.db");
services.AddDbContext<Northwind>(options =>
    options.UseSqlite($"Data Source={databasePath}"));
```

5. In the NorthwindWeb project, in the Pages folder, open suppliers.cshtml.cs, and import the Packt.Shared and System.Linq namespaces, as shown in the following code:

```
using System.Linq;
using Packt.Shared;
```

6. In the SuppliersModel class, add a private field and a constructor to get the Northwind database context, as shown in the following code:

```
private Northwind db;
public SuppliersModel(Northwind injectedContext)
{
    db = injectedContext;
}
```

7. In the OnGet method, modify the statements to get the names of suppliers by selecting the company names from the Suppliers property of the database context, as shown highlighted in the following code:

```
public void OnGet()
{
    ViewData["Title"] = "Northwind Web Site - Suppliers";
    Suppliers = db.Suppliers.Select(s => s.CompanyName);
}
```

8. In **TERMINAL**, enter the command `dotnet run` to start the website, in Chrome, enter `http://localhost:5000/`, click the button to go to the Suppliers page, and note that the supplier table now loads from the database, as shown in the following screenshot:

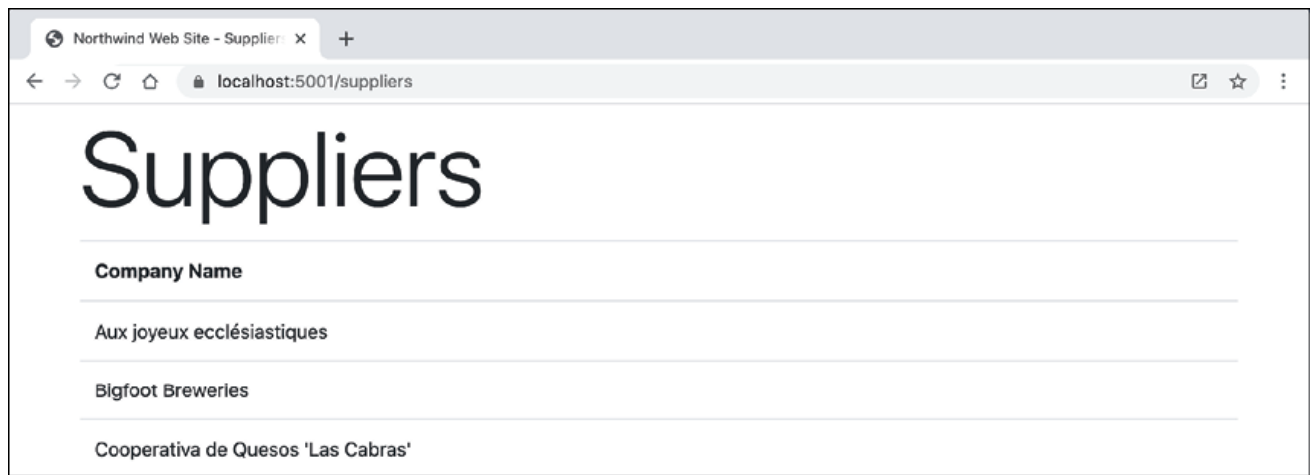


Figure 7.10: The suppliers table loading from the database

## Manipulating data using Razor Pages

You will now add functionality to insert a new supplier.

### Enabling a model to insert entities

First, you will modify the supplier model so that it responds to HTTP `POST` requests when a visitor submits a form to insert a new supplier:

1. In the NorthwindWeb project, in the Pages folder, open `suppliers.cshtml.cs` and import the following namespace:

```
using Microsoft.AspNetCore.Mvc;
```

2. In the `SuppliersModel` class, add a property to store a supplier, and a method named `OnPost` that adds the supplier if its model is valid, as shown in the following code:

```
[BindProperty]
public Supplier Supplier { get; set; }
public IActionResult OnPost()
{
    if (ModelState.IsValid)
    {
        db.Suppliers.Add(Supplier);
        db.SaveChanges();
        return RedirectToPage("/suppliers");
    }
    return Page();
}
```

While reviewing the preceding code, note the following:

- We added a property named `Supplier` that is decorated with the `[BindProperty]` attribute so that we can easily connect HTML elements on the web page to properties in the `Supplier` class.
- We added a method that responds to HTTP `POST` requests. It checks that all property values conform to validation rules and then adds the supplier to the existing table and saves changes to the database context. This will generate a SQL statement to perform the insert into the database. Then it redirects to the **Suppliers** page so that the visitor sees the newly added supplier.

### Defining a form to insert new suppliers

Second, you will modify the Razor page to define a form that a visitor can fill in and submit to insert a new supplier:

1. Open `suppliers.cshtml`, and add tag helpers after the `@model` declaration so that we can use tag helpers like `asp-for` on this Razor page, as shown in the following markup:

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

2. At the bottom of the file, add a form to insert a new supplier, and use the `asp-for` tag helper to connect the `CompanyName` property of the `Supplier` class to the input box, as shown in the following markup:

```
<div class="row">
    <p>Enter a name for a new supplier:</p>
    <form method="POST">
        <div><input asp-for="Supplier.CompanyName" /></div>
        <input type="submit" />
    </form>
</div>
```

While reviewing the preceding markup, note the following:



- The `<form>` element with a `POST` method is normal HTML, so an `<input type="submit" />` element inside it will make an HTTP `POST` request back to the current page with values of any other elements inside that form.
  - An `<input>` element with a tag helper named `asp-for` enables data binding to the model behind the Razor page.
3. Start the website, click **Learn more about our suppliers**, scroll down the table of suppliers to the bottom of the form to add a new supplier, enter Bob's Burgers, and click on **Submit**, as shown in the following screenshot:

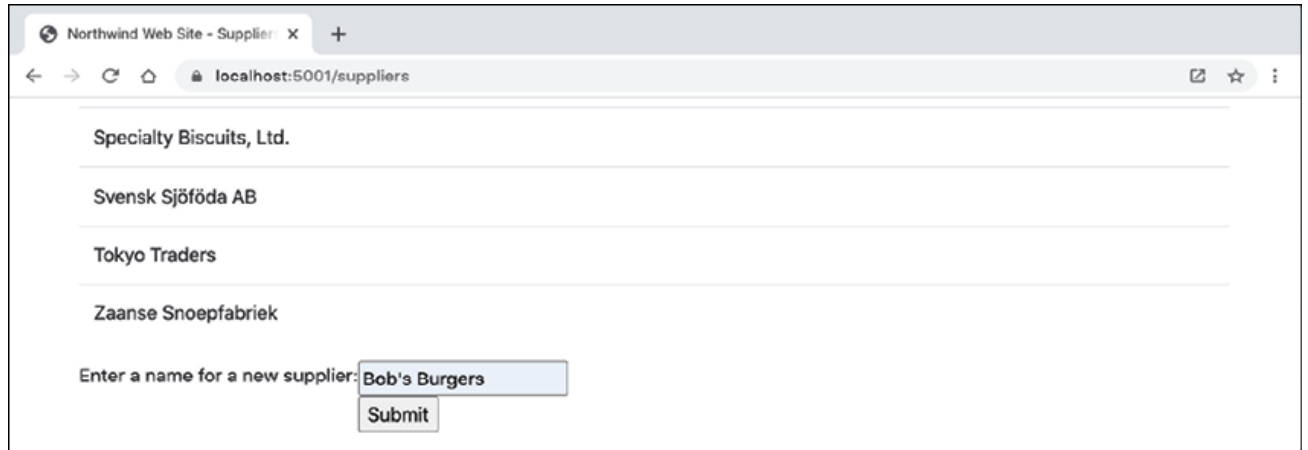


Figure 7.11: Entering a new supplier

4. Note that you see a refreshed *Suppliers* list with the new supplier added.
5. Close the browser.

# Using Razor class libraries

Everything related to a Razor page can be compiled into a class library for easier reuse. With .NET Core 3.0 and later, this can include static files. A website can either use the Razor page's view as defined in the class library or override it.

## Creating a Razor class library

Let us create a new Razor class library:

1. Create a subfolder in PracticalApps named NorthwindEmployees.
2. In Visual Studio Code, add the NorthwindEmployees folder to the PracticalApps workspace.
3. Navigate to **Terminal** | **New Terminal** and select NorthwindEmployees.
4. In **TERMINAL**, enter the following command to create a **Razor Class Library** project:

```
dotnet new razorclasslib -s
```

**More Information:** The `-s` option is short for `--support-pages-and-views` that enables the class library to use Razor Pages and `.cshtml` file views.

## Disabling compact folders

Before we implement our Razor class library, I want to explain a recent Visual Studio Code feature that confused some readers of the previous edition because the feature was added after publishing.

The **compact folders** feature means that nested folders like `/Areas/MyFeature/Pages/` are shown in a compact form if the intermediate folders in the hierarchy do not contain files, as shown in the following screenshot:

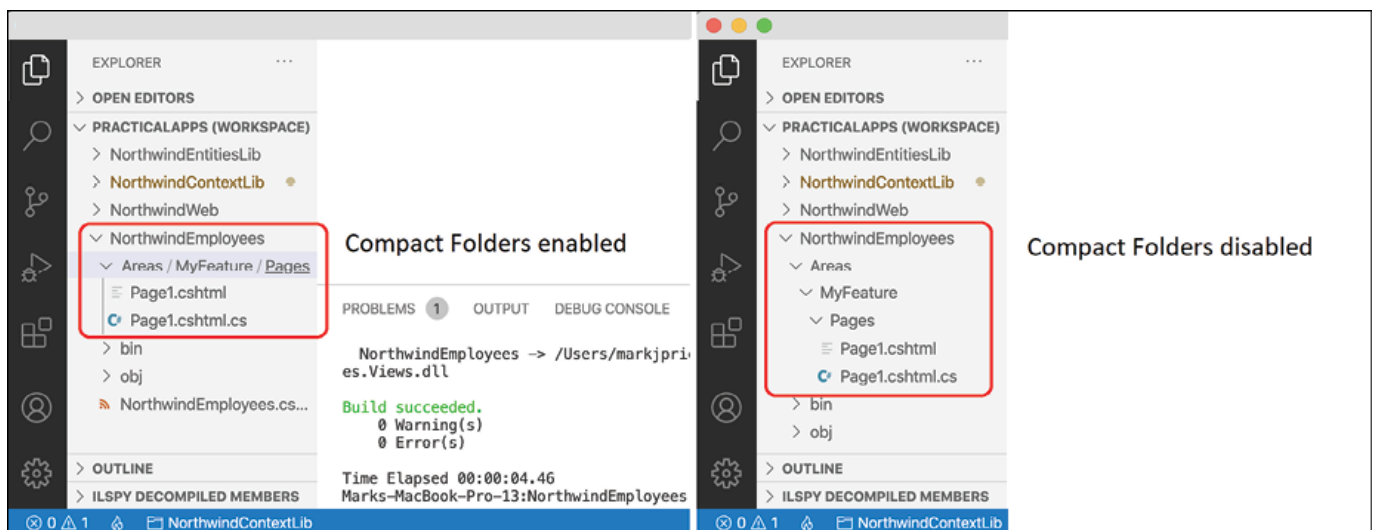


Figure 7.12: Compact folders enabled or disabled

If you would like to disable the Visual Studio Code compact folders feature, complete the following steps:

1. On macOS, navigate to **Code** | **Preferences** | **Settings**, or press `Cmd + ,`. On Windows, navigate to **File** | **Preferences** | **Settings**, or press `Ctrl + ,`.
2. In the **Search** settings box, enter `compact`.
3. Clear the **Explorer: Compact Folders** checkbox, as shown in the following screenshot:

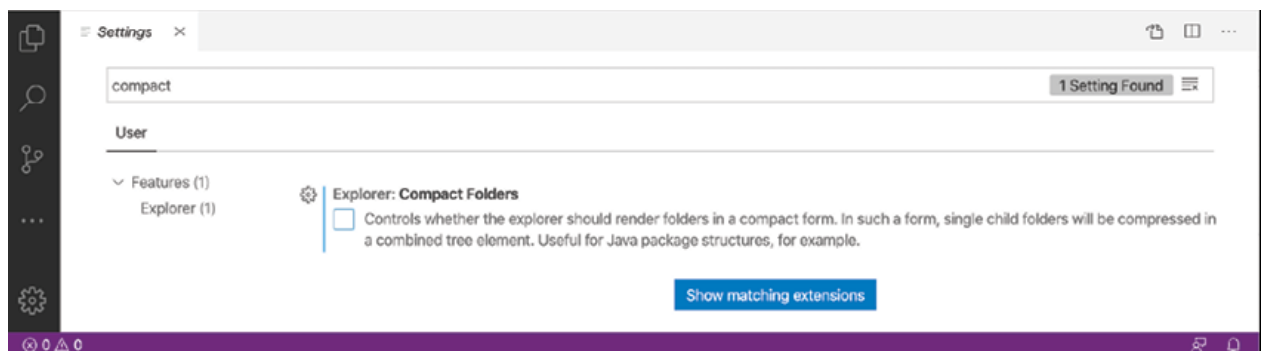


Figure 7.13: Disabling compact folders

4. Close the **Settings** tab.

**More Information:** You can read about the Compact Folders feature introduced with Visual Studio Code 1.41 in November 2019 at the following link: [https://github.com/microsoft/vscode-docs/blob/vnext/release-notes/v1\\_41.md#compact-folders-in-explorer](https://github.com/microsoft/vscode-docs/blob/vnext/release-notes/v1_41.md#compact-folders-in-explorer)

## Implementing the employees feature using EF Core

Now we can add a reference to our entity models to get the employees to show in the Razor class library:

1. Edit `NorthwindEmployees.csproj`, note the SDK is `Microsoft.NET.Sdk.Razor`, and add a reference to the `NorthwindContextLib` project, as shown highlighted in the following markup:

```
<Project Sdk="Microsoft.NET.Sdk.Razor">
  <PropertyGroup>
    <TargetFramework>net5.0</TargetFramework>
    <AddRazorSupportForMvc>true</AddRazorSupportForMvc>
  </PropertyGroup>
  <ItemGroup>
    <FrameworkReference Include="Microsoft.AspNetCore.App" />
  </ItemGroup>
  <ItemGroup>
    <ProjectReference Include=
      "..\NorthwindContextLib\NorthwindContextLib.csproj" />
  </ItemGroup>
</Project>
```

2. In **TERMINAL**, enter the following command to restore packages and compile the project: `dotnet build`
3. In **EXPLORER**, under the `Areas` folder, right-click the `MyFeature` folder, select **Rename**, enter the new name `PacktFeatures`, and press **Enter**.
4. In **EXPLORER**, under the `PacktFeatures` folder, and in the `Pages` subfolder, add a new file named `_ViewStart.cshtml`.
5. Modify its content, as shown in the following markup:

```
@{
  Layout = "_Layout";
}
```

6. In the `Pages` subfolder, rename `Page1.cshtml` to `employees.cshtml`, and rename `Page1.cshtml.cs` to `employees.cshtml.cs`.
7. Modify `employees.cshtml.cs`, to define a page model with an array of `Employee` entity instances loaded from the `Northwind` database, as shown in the following code:

```
using Microsoft.AspNetCore.Mvc.RazorPages; // PageModel
using Packt.Shared;                       // Employee
using System.Linq;                        // ToArray()
using System.Collections.Generic;          // IEnumerable<T>
namespace PacktFeatures.Pages
{
    public class EmployeesPageModel : PageModel
    {
        private Northwind db;
        public EmployeesPageModel(Northwind injectedContext)
        {
            db = injectedContext;
        }
        public IEnumerable<Employee> Employees { get; set; }
        public void OnGet()
        {
            Employees = db.Employees.ToArray();
        }
    }
}
```

8. Modify `employees.cshtml`, as shown in the following markup:

```
@page
@using Packt.Shared
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@model PacktFeatures.Pages.EmployeesPageModel
<div class="row">
  <h1 class="display-2">Employees</h2>
</div>
<div class="row">
  @foreach(Employee employee in Model.Employees)
  {
    <div class="col-sm-3">
      <partial name="_Employee" model="employee" />
    </div>
  }
</div>
```

While reviewing the preceding markup, note the following:

- We import the `Packt.Shared` namespace so that we can use classes in it like `Employee`.
- We add support for tag helpers so that we can use the `<partial>` element.
- We declare the model type for this Razor page to use the class that you just defined.
- We enumerate through the `Employees` in the model, outputting each one using a partial view. Partial views are like small pieces of a Razor page and you will create one in the next few steps.

**More Information:** The `<partial>` tag helper was introduced in ASP.NET Core 2.1. You can read more about it at the following link: <https://docs.microsoft.com/en-us/aspnet/core/mvc/views/tag-helpers/built-in/partial-tag-helper>

## Implementing a partial view to show a single employee

Now we will define a partial view to render a single employee:

1. In the Pages folder, create a Shared folder.
2. In the Shared folder, create a file named `_Employee.cshtml`.
3. Modify `_Employee.cshtml`, as shown in the following markup:

```
@model Packt.Shared.Employee
<div class="card border-dark mb-3" style="max-width: 18rem;">
  <div class="card-header">@Model.FirstName
    @Model.LastName</div>
  <div class="card-body text-dark">
    <h5 class="card-title">@Model.Country</h5>
    <p class="card-text">@Model.Notes</p>
  </div>
</div>
```

While reviewing the preceding markup, note the following:

- By convention, the names of partial views start with an underscore.
- If you put a partial view in the Shared folder, then it can be found automatically.
- The model type for this partial view is an `Employee` entity.
- We use Bootstrap card styles to output information about each employee.

## Using and testing a Razor class library

You will now reference and use the Razor class library in the website project:

1. Modify the `NorthwindWeb.csproj` file to add a reference to the `NorthwindEmployees` project, as shown highlighted in the following markup:

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <ProjectReference Include=
      "..\NorthwindContextLib\NorthwindContextLib.csproj" />
    <ProjectReference Include=
      "..\NorthwindEmployees\NorthwindEmployees.csproj" />
  </ItemGroup>
</Project>
```

2. Modify `Pages\index.cshtml` to add a link to the Packt feature `employees` page after the link to the suppliers page, as shown in the following markup:

```
<p>
  <a class="btn btn-primary" href="packtfeatures/employees">
    Contact our employees
  </a>
</p>
```

3. Start the website, visit the website using Chrome, and click the button to see the cards of employees, as shown in the following screenshot:

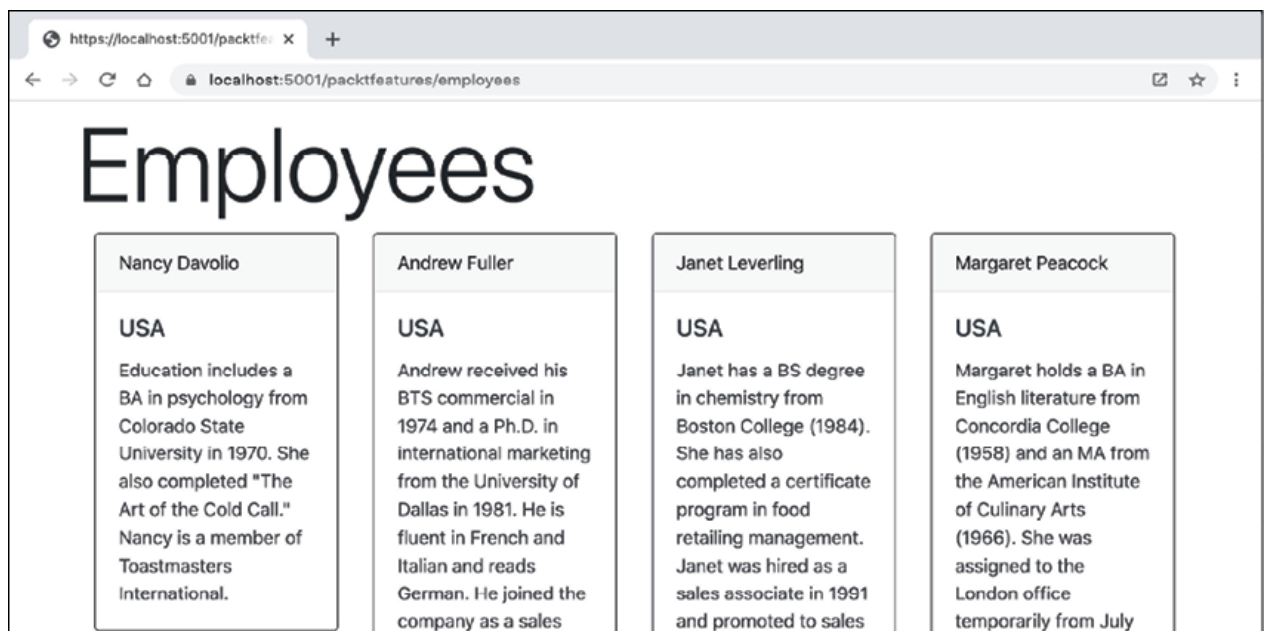


Figure 7.14: A list of employees

# Explore topics

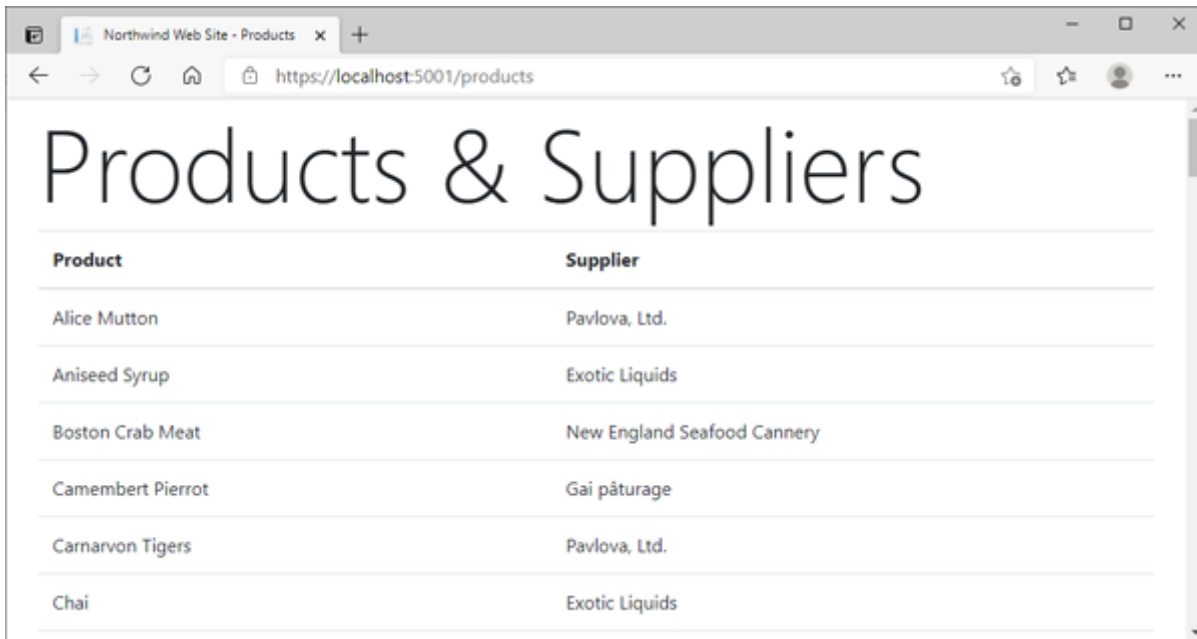
Use the following links to read more details about this section's topics:

- **ASP.NET Core fundamentals:** <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/>
- **Static files in ASP.NET Core:** <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/static-files>
- **Introduction to Razor Pages in ASP.NET Core:** <https://docs.microsoft.com/en-us/aspnet/core/razor-pages/>
- **Razor syntax reference for ASP.NET Core:** <https://docs.microsoft.com/en-us/aspnet/core/mvc/views/razor>
- **Layout in ASP.NET Core:** <https://docs.microsoft.com/en-us/aspnet/core/mvc/views/layout>
- **Tag Helpers in ASP.NET Core:** <https://docs.microsoft.com/en-us/aspnet/core/mvc/views/tag-helpers/intro>
- **ASP.NET Core Razor Pages with EF Core:** <https://docs.microsoft.com/en-us/aspnet/core/data/ef-rp/intro>
- **DEEP DIVE: HOW IS THE ASP.NET CORE MIDDLEWARE PIPELINE BUILT?**  
<https://www.stevejgordon.co.uk/how-is-the-asp-net-core-middleware-pipeline-built>

# Exercises

## Exercise 7.1.

- add a razor page to your NorthwindWeb application that displays all the products Northwind Traders have for sale
- also display the supplier for each product



Product	Supplier
Alice Mutton	Pavlova, Ltd.
Aniseed Syrup	Exotic Liquids
Boston Crab Meat	New England Seafood Cannery
Camembert Pierrot	Gai pâturage
Carnarvon Tigers	Pavlova, Ltd.
Chai	Exotic Liquids