Building websites using ASP.NET Core

Websites are made up of multiple web pages loaded statically from the filesystem or generated dynamically by a server-side technology such as ASP.NET Core. A web browser makes GET requests using URLs that identify each page and can manipulate data stored on the server using the POST, PUT, and DELETE requests. With many websites, the web browser is treated as a presentation layer, with almost all of the processing performed on the server side. A small amount of JavaScript might be used on the client side to implement some presentation features, such as carousels.

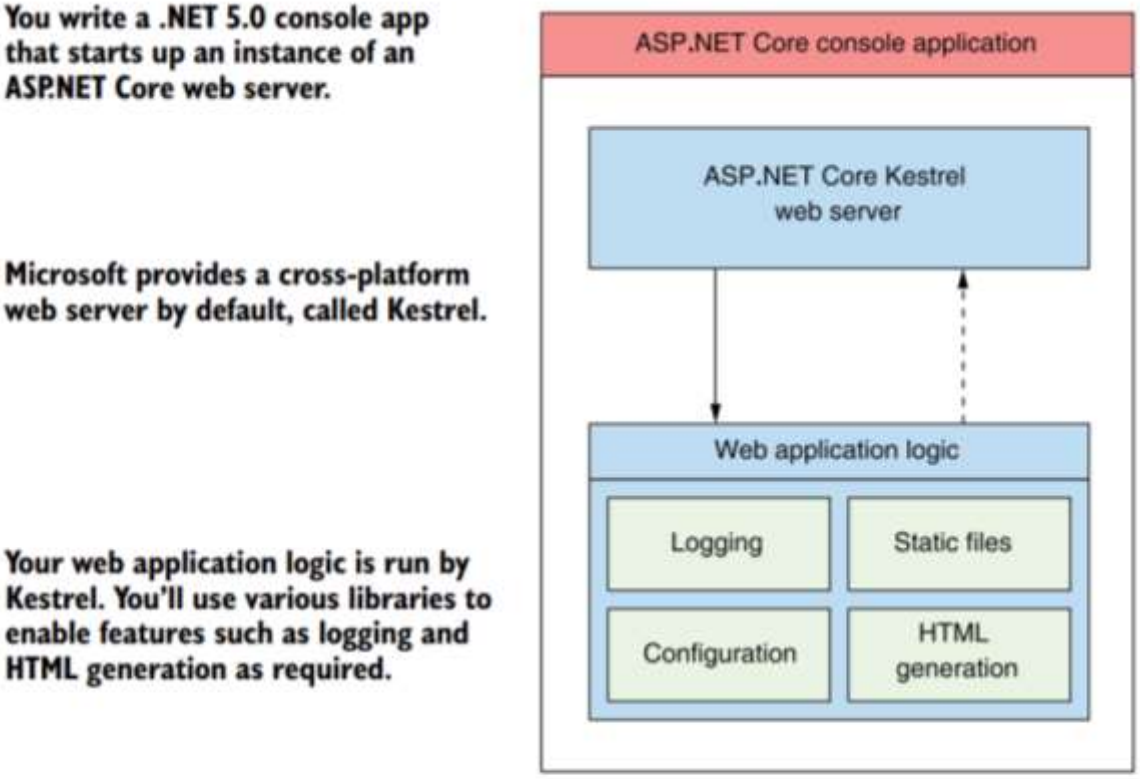


Figure 1.4 The ASP.NET Core application model. The .NET 5.0 platform provides a base console application model for running command-line apps. Adding a web server library converts this into an ASP.NET Core web app. Additional features, such as configuration and logging, are added by way of additional libraries.

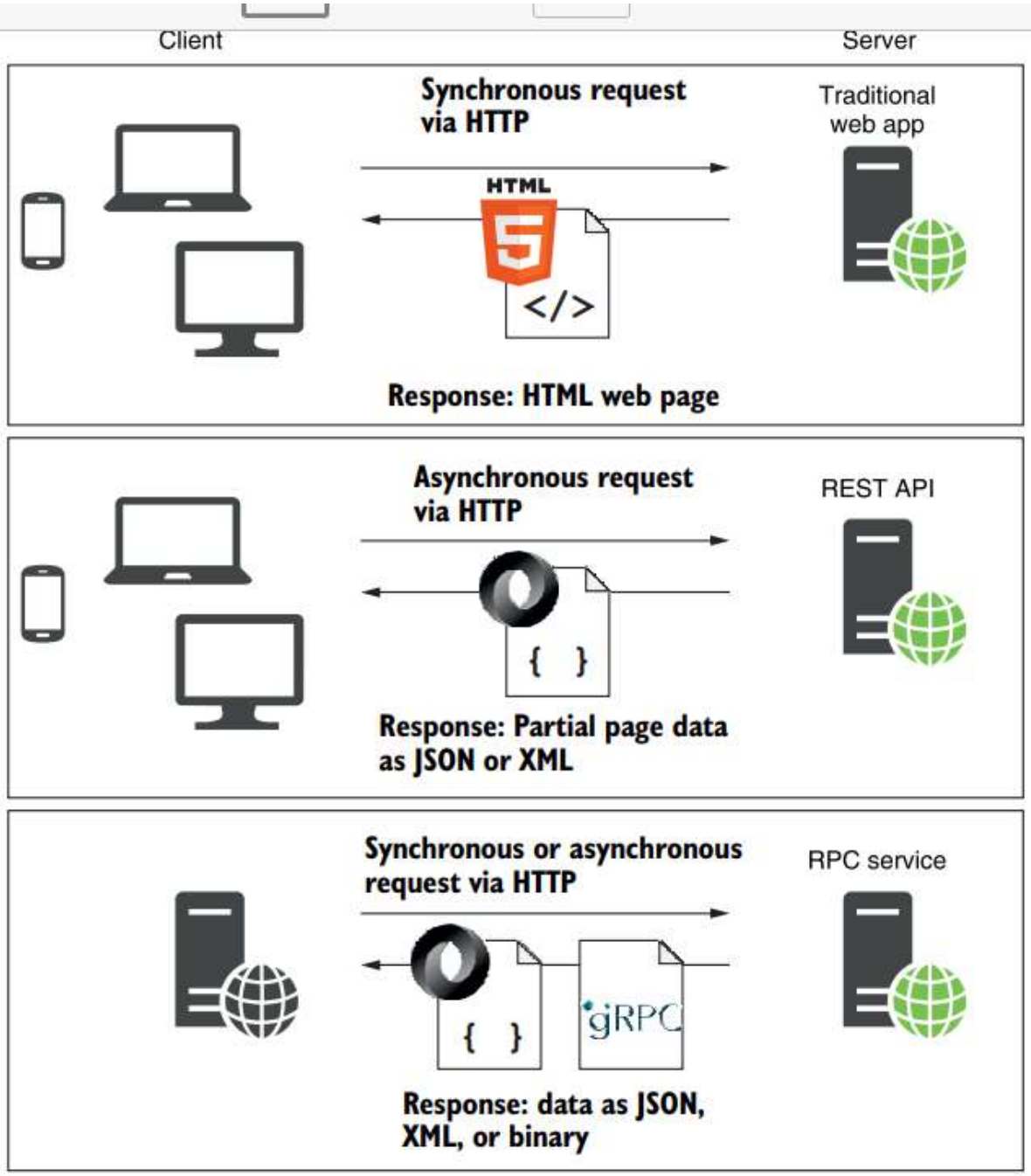


Figure 1.6 ASP.NET Core can act as the server-side application for a variety of clients: it can serve HTML pages for traditional web applications, it can act as a REST API for client-side SPA applications, or it can act as an ad hoc RPC service for client applications.

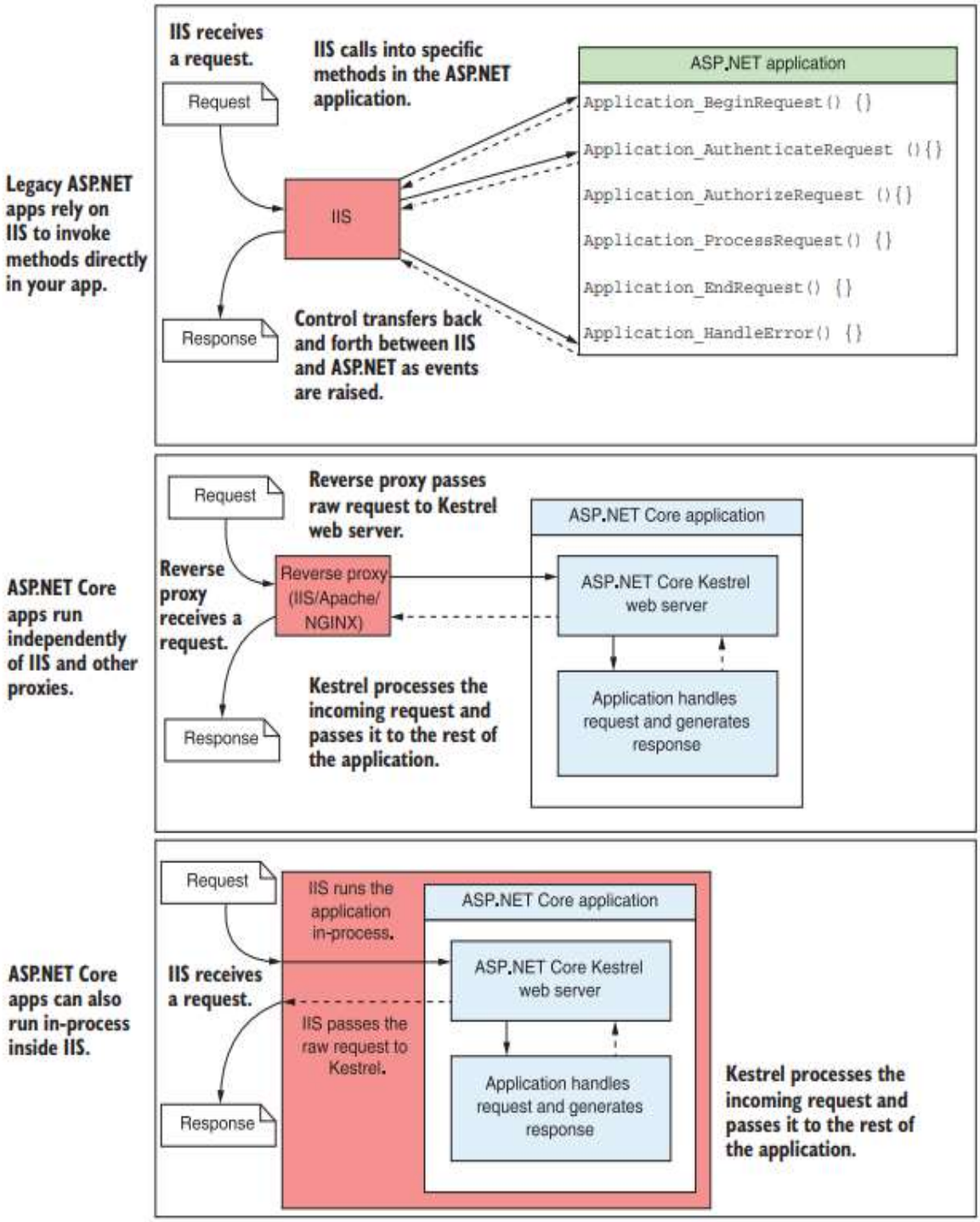


Figure 1.7 The difference between hosting models in ASP.NET (top) and ASP.NET Core (bottom). With the previous version of ASP.NET, IIS is tightly coupled with the application. The hosting model in ASP.NET Core is simpler; IIS hands off the request to a self-hosted web server in the ASP.NET Core application and receives the response, but has no deeper knowledge of the application.

The first port of call after the reverse proxy forwards a request is the ASP.NET

Core web server, which is the default cross-platform Kestrel server. Kestrel takes the raw incoming network request and uses it to generate an HttpContext object that the rest of the application can use.

NOTE Kestrel isn't the only HTTP server available in ASP.NET Core, but it's the most performant and is cross-platform. The main alternative, HTTP.sys, only runs on Windows and can't be used with IIS.1 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.

Razor Pages are responsible for generating the HTML that makes up the pages of

a typical ASP.NET Core web app. They're also typically where you find most of the business logic of your app, calling out to various services in response to the data contained in the original request. Not every app needs an MVC or Razor Pages block, but it's typically how you'll build most apps that display HTML to a user.

The HttpContext object

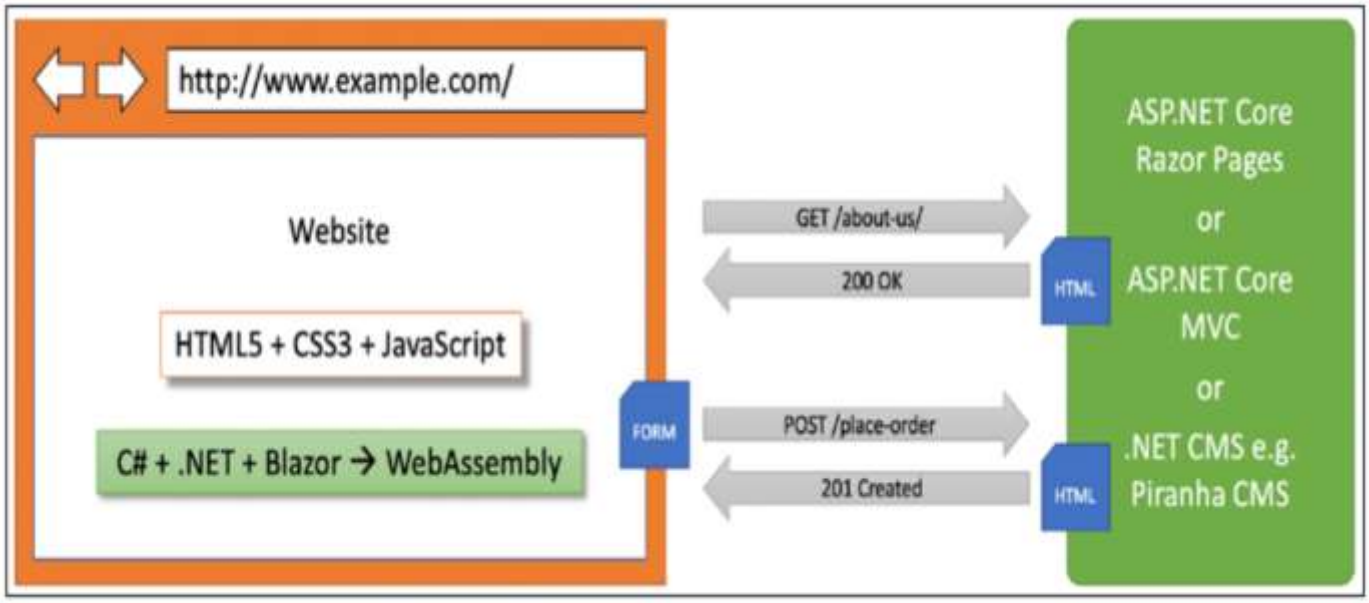
The HttpContext constructed by the ASP.NET Core web server is used by the application as a sort of storage box for a single request. Anything that's specific to this particular request and the subsequent response can be associated with it and stored in it. This could include properties of the request, request-specific services, data that's been loaded, or errors that have occurred. The web server fills the initial HttpContext with details of the original HTTP request and other configuration details and passes it on to the rest of the application.

ASP.NET Core 3.0 provides three technologies for building websites:

• **ASP.NET Core Razor** Pages and Razor class libraries are ways to dynamically generate HTML for simple websites.

• **ASP.NET Core MVC** is an implementation of the Model-View-Controller design pattern that is popular for developing complex websites.

• **Blazor** lets you build server-side or client-side components and user interfaces using C# instead of JavaScript.



Understanding web applications

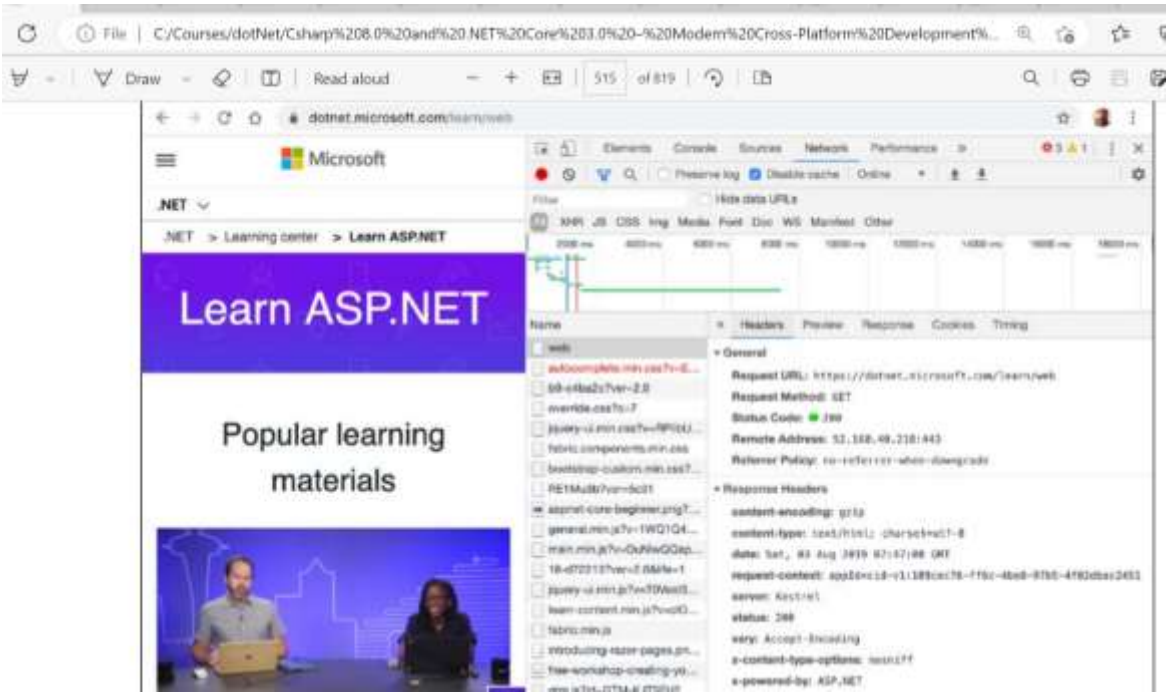
Web applications, also known as Single-Page Applications (SPAs), are made up of a single web page built with a frontend technology such as Angular, React, Vue, or a proprietary JavaScript library that can make requests to a backend web service for getting more data when needed and posting updated data, using common serialization formats, such as XML and JSON. The canonical examples are Google web apps like Gmail, Maps, and Docs.

With a web application, the client-side uses JavaScript libraries to implement

sophisticated user interactions, but most of the important processing and data access still happens on the server-side, because the web browser has limited access to local system resources.

Understanding HTTP and HTTPS

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:



Just as with the generic web page example, the request process starts when a user's browser sends an HTTP request to the server, as shown in figure 1.9.

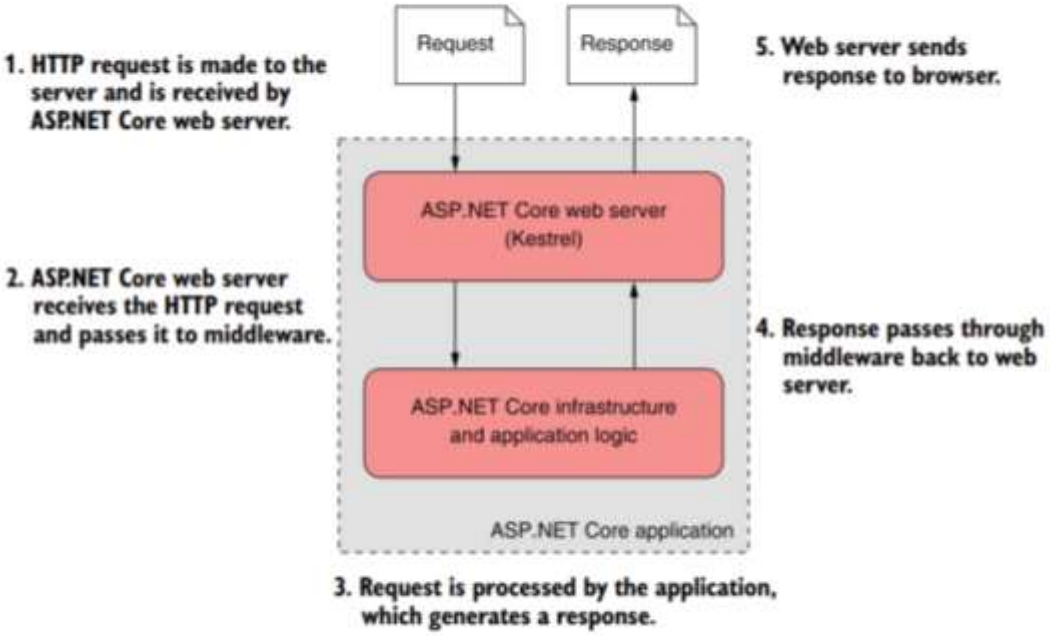


Figure 1.9 How an ASP.NET Core application processes a request. A request is received by the ASP.NET Core application, which runs a self-hosted web server. The web server processes the request and passes it to the body of the application, which generates a response and returns it to the web server. The web server sends this response to the browser.

Major Points to keep in mind about ASP.NET core

- ASP.NET Core is a new web framework built with modern software architecture practices and modularization as its focus.
- Fetching a web page involves sending an HTTP request and receiving an HTTP response.
- ASP.NET Core allows you to dynamically build responses to a given request.
- An ASP.NET Core application contains a web server (named Kestrel), which serves as the entry point for a request.
- ASP.NET Core apps are typically protected from the internet by a reverse-proxy server (i.e. IIS server for Windows), which forwards requests to the application.

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 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.

Creating an ASP.NET Core project

1. Create a new project by selecting ASP.NET Core app options. Place your solution to webapp folder (or you can choose your own folder name).
2. Go to Terminal and run the following command: dotnet build.

Note: <Project Sdk="Microsoft.NET.Sdk.Web">, different SDK in webapp.csporj file.

3. 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 specifies a Startup class that is used to configure the website, which is then built and run, 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>();
 });
}
```

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

° The ConfigureServices method is currently empty. We will use it later to add services like MVC.

° The Configure method 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: See the code in VS.

To secure the data transfers between client and server, enter the command:

dotnet dev-certs https --trust, in Terminal. 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.

Enabling static and default files

1. In the webapp folder, add a new folder named wwwroot.
2. Add a new file to the wwwroot folder named index.html or default.html.
3. Write the html content as shown in VS.

If we 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.

4. In order to configure this, we need to add two lines of code in Startup.cs file after app.useRouting() method (see the code in VS).
   app.UseDefaultFiles(); // index.html, default.html, and so on
   app.UseStaticFiles();

Now the new content of index.html will be rendered on the browser when we run dotnet run command in the terminal.

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.

Enabling 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.

1. Add a new folder called, Pages, in the webapp project.
2. Move the index.html file into the Pages folder.
3. Rename the file extension from .html to .cshtml.
4. In Startup.cs, in the ConfigureServices method, add statements to add Razor Pages and its related services like model binding, authorization, antiforgery, views, and tag helpers, as shown highlighted in the following code (see code in VS).

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

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

NuGet packages and the .NET command-line interface

One of the foundational components of .NET 5.0 cross-platform development is the .NET command-line interface (CLI). This provides several basic commands for creating, building, and running .NET 5.0 applications. Visual Studio effectively calls these automatically, but you can also invoke them directly from the command line if you're using a different editor. The most common commands used during development are

- `dotnet restore`
- `dotnet build`
- `dotnet run`

Each of these commands should be run inside your project folder and will act on that project alone.

Most ASP.NET Core applications have dependencies on various external libraries, which are managed through the NuGet package manager. These dependencies are listed in the project, but the files of the libraries themselves aren't included. Before you can build and run your application, you need to ensure there are local copies of each dependency in your project folder. The first command, `dotnet restore`, ensures your application's NuGet dependencies are copied to your project folder.

ASP.NET Core projects list their dependencies in the project's .csproj file. This is an XML file that lists each dependency as a `PackageReference` node. When you run `dotnet restore`, it uses this file to establish which NuGet packages to download and copy to your project folder. Any dependencies listed are available for use in your application.

The restore process typically happens implicitly when you build or run your application, but it can sometimes be useful to run it explicitly, in continuous-integration build pipelines, for example.

You can compile your application using `dotnet build`. This will check for any errors in your application and, if there are no issues, will produce output that can be run using `dotnet run`.

Each command contains a number of switches that can modify its behavior. To see the full list of available commands, run

```
dotnet --help
```

or to see the options available for a particular command, `new` for example, run

```
dotnet new --help
```

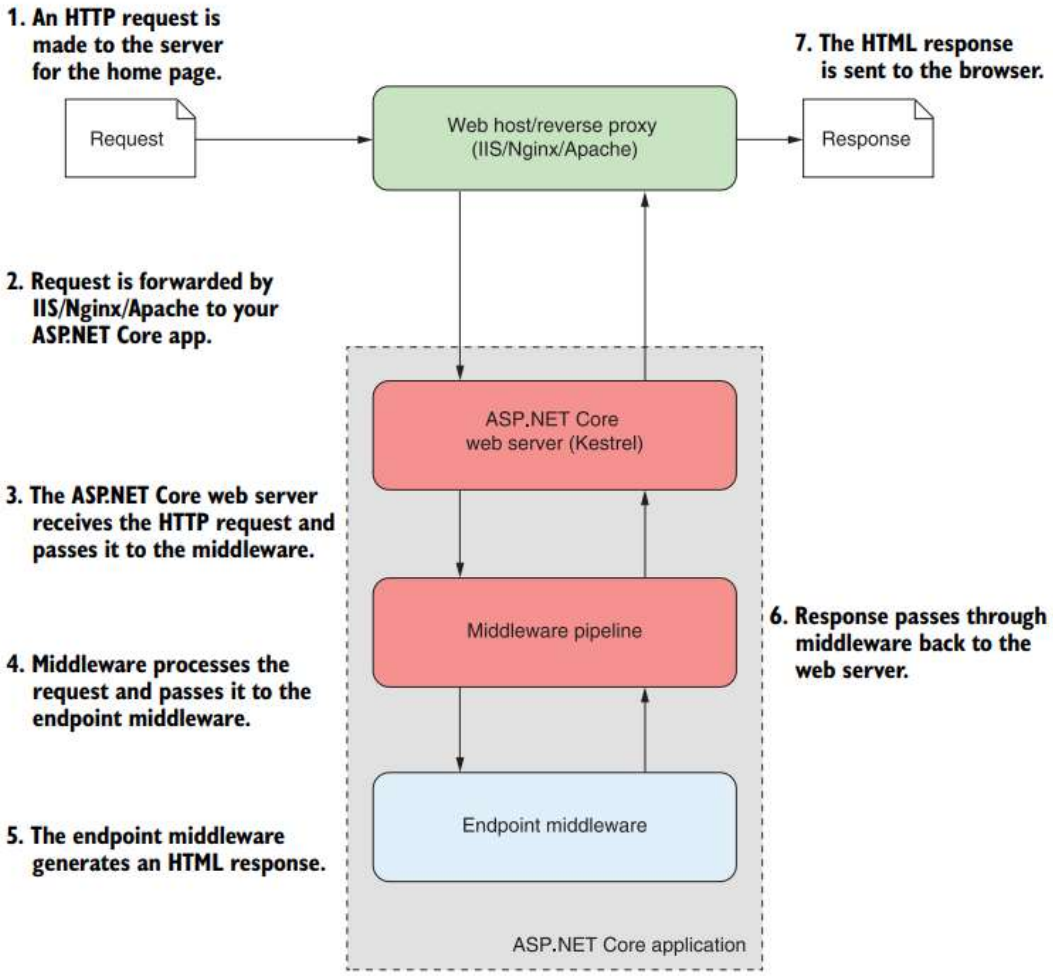


Figure 2.1 An overview of an ASP.NET Core application. The ASP.NET Core application receives an incoming HTTP request from the browser. Every request passes to the middleware pipeline, which potentially modifies it and then passes it to the endpoint middleware at the end of the pipeline to generate a response. The response passes back through the middleware to the server, and finally out to the browser.

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 convert html page to razor page:

1. Add the @page statement to the top of the file.

2. After the @page statement, add an @functions statement block.

3. Define a property to store the name of the current day as a string value.

4. 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");
 }
}
```

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

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

Workflow example of rendering Privacy html page

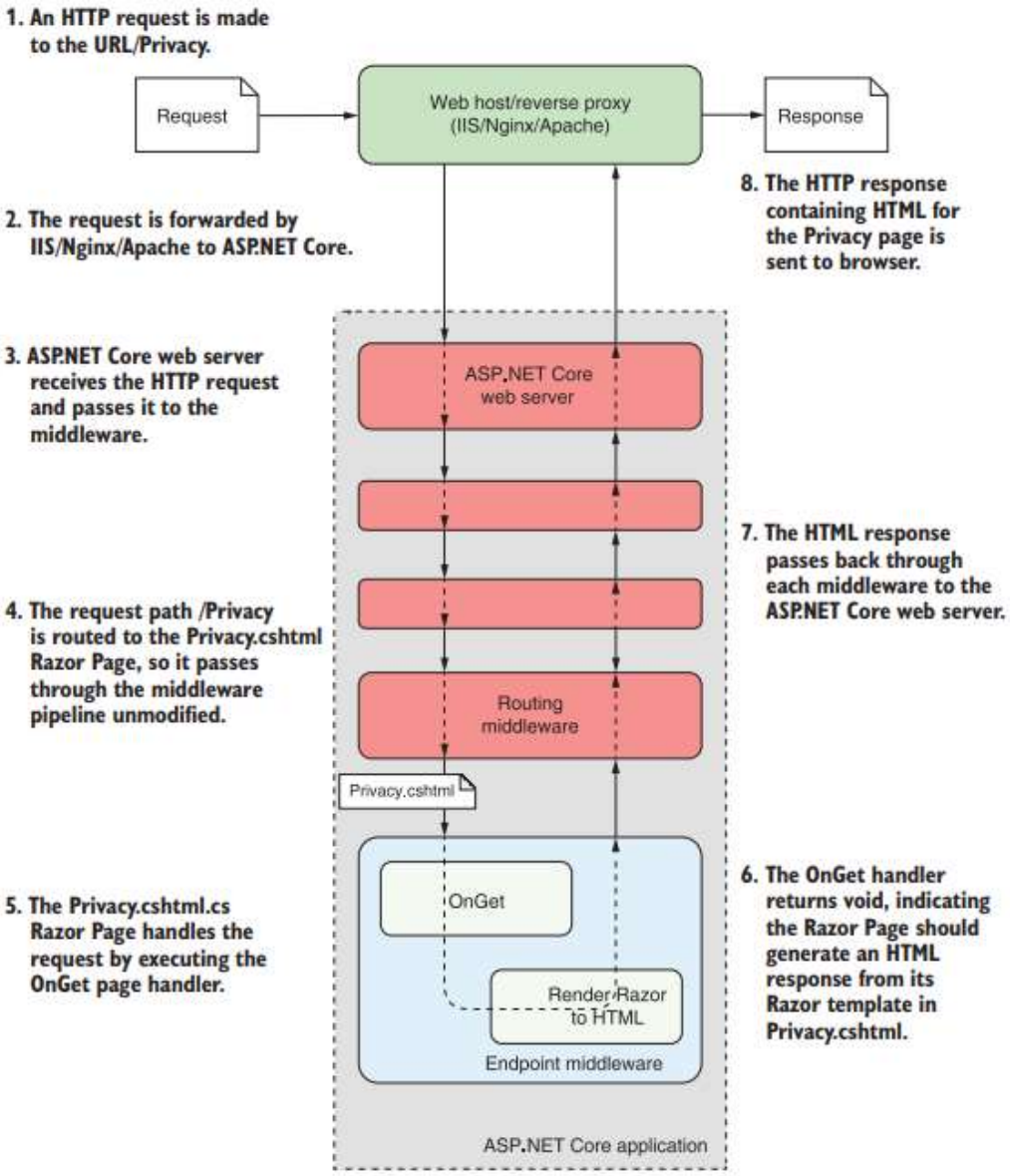


Figure 2.16 An overview of a request to the `/Privacy` URL for the sample ASP.NET Razor Pages application. The routing middleware routes the request to the `OnGet` handler of the Privacy.cshtml.cs Razor Page. The Razor Page generates an HTML response by executing the Razor template in Privacy.cshtml and passes the response back through the middleware pipeline to the browser.

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 can copy and paste from there), as shown in the following markup:

```
<!DOCTYPE html>
<html lang="en">
<head>
```

See the code in VS.

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 will 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");
  }
}
```

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.

We 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.

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"
      };
    }
  }
}
```

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.

Run the app and compare the results.

Configure Entity Framework Core as a service for database use

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

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>netcoreapp3.0</TargetFramework>
</PropertyGroup>
```

<ItemGroup>

<ProjectReference Include=

"..\NorthwindContextLib\NorthwindContextLib.csproj" />

</ItemGroup>

</Project>

3. In Terminal, restore packages and compile the project by entering the following command: dotnet build

4. Open Startup.cs and import the following name spaces (packages)

using System.IO;

using Microsoft.EntityFrameworkCore;

using NorthwindEntitiesLib;

5. Add a statement 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 = "c:\\database\\Database\\Northwind.db"
services.AddDbContext<Northwind>(options =>
 options.UseSqlite($"Data Source={databasePath}"));
```

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

```
using System.Linq;
using NorwindEntitiesLib;
```

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

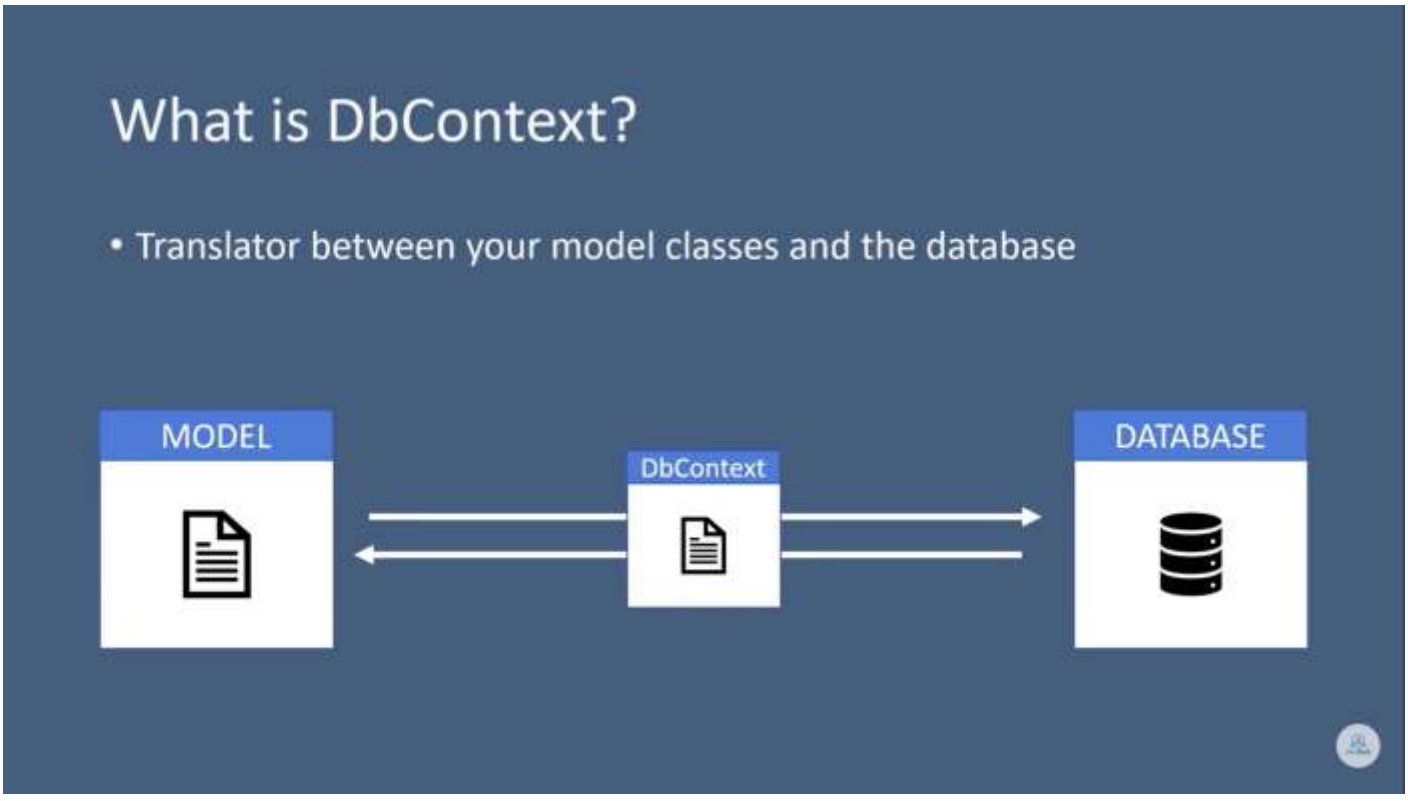
```
private Northwind db;
```

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

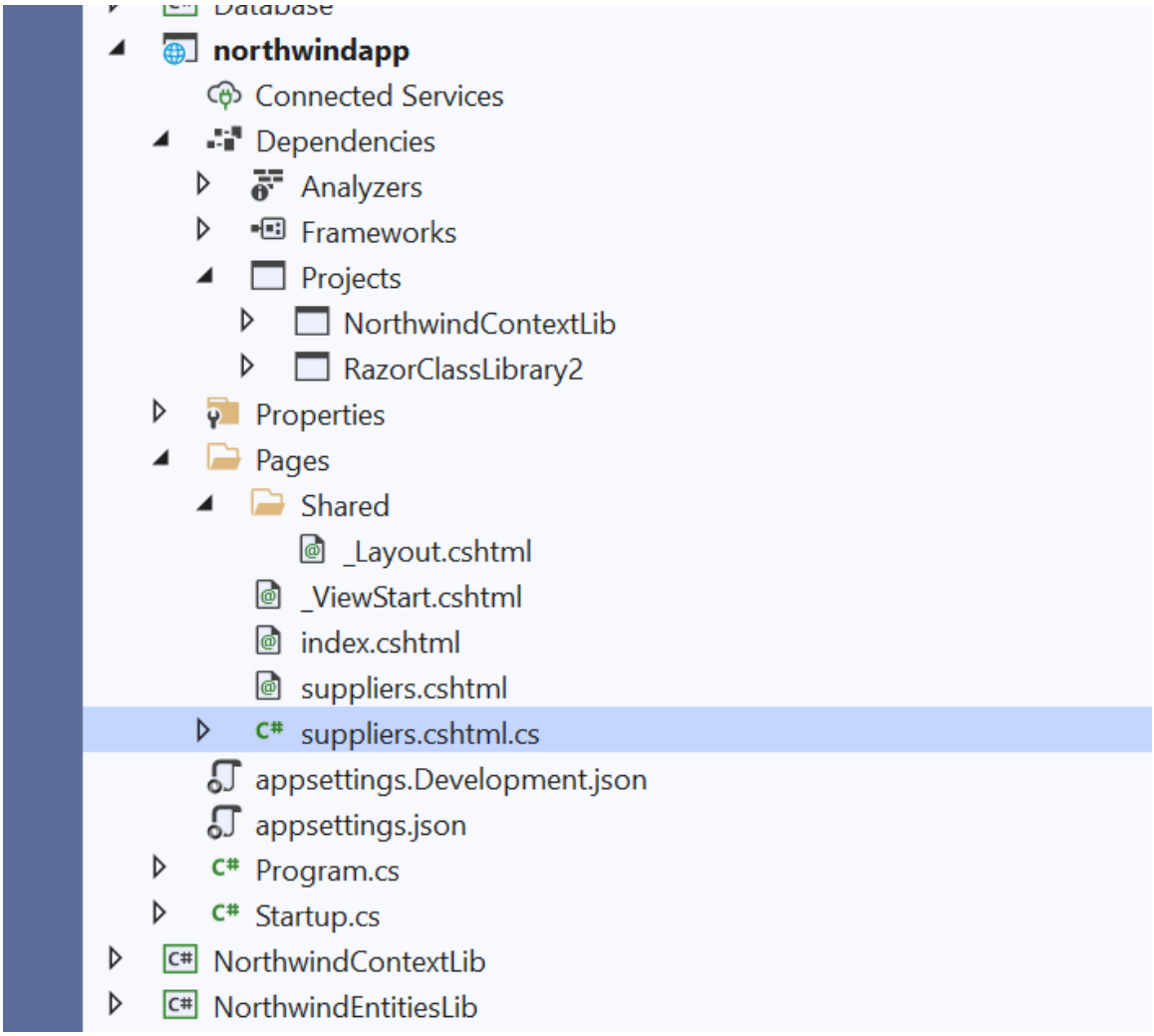
8. In the OnGet method, modify the statements to get the names of suppliers by selecting the company names from the Suppliers property of the database context, as shown highlighted in the following code:

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

Run the app and you will see the list of suppliers in the body part of your webapp.



The folder structure of your projects should look like this:



Enabling a model to insert entities (inserting rows in the supplier table)

First, 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 changed to the database context. This will generate an 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 table in the DB

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.

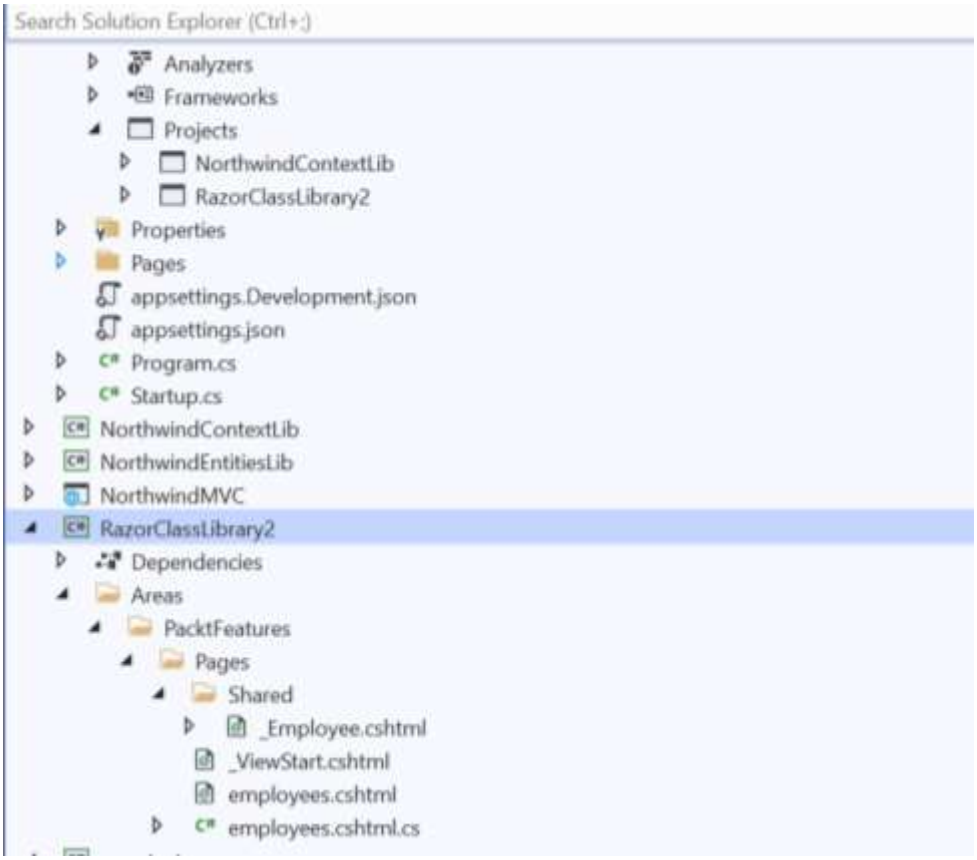
° An <input> element with a tag helper named asp-for enables data binding to the model behind the Razor page.

Run the app and you will see an input box where you can add a new supplier name and submit. This will be added to the table and displayed on the body part of the SPA.

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 now include static files. A website can either use the Razor page's view as defined in the class library or override it.

1. Create a new razorclass library project by right-clicking the solution name, in the following structure, I have used RazorClassLibrary2 as the project name. Check on the checkbox Support Pages and Views so that you will get the proper folder structure for this library project.



2. Add the following project in the RazorClassLibrary2.csproj file as below:
   <ItemGroup>
    <ProjectReference Include=

```
    "..\NorthwindContextLib\NorthwindContextLib.csproj" />
  </ItemGroup>
```

3. In Terminal, enter the following command to restore packages and compile the project: dotnet build
4. In Explorer, expand the Areas folder, and rename the MyFeature folder to PacktFeatures (you can give any name you want)
5. In Explorer, expand the PacktFeatures folder, and in the Pages subfolder, add a new file named _ViewStart.cshtml.
6. Modify its content, as shown in the following markup:

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

7. In the Pages subfolder, rename Page1.cshtml to employees.cshtml, and rename Page1.cshtml.cs to employees.cshtml.cs (see the code in Gitgub).

While reviewing the markup in the .cshtml file , note the following:

° We import the NorwindEntitiesLib 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.

8. In the Pages folder, create a Shared folder.
9. In the Shared folder, create a file named _Employee.cshtml (partial view file).

10. Modify _Employee.cshtml, as shown in the following markup:

```
@model NorthwindEntitiesLib.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 a Razor class library

We will now reference and use the Razor class library in the website project, add this project reference in the northwindapp.csproj file.

Modify Pages\index.cshtml of northwindapp project to add a link to the 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>
```

Run the app and click on the employees link, this will call the packtfeatures/employees view from razor class library.