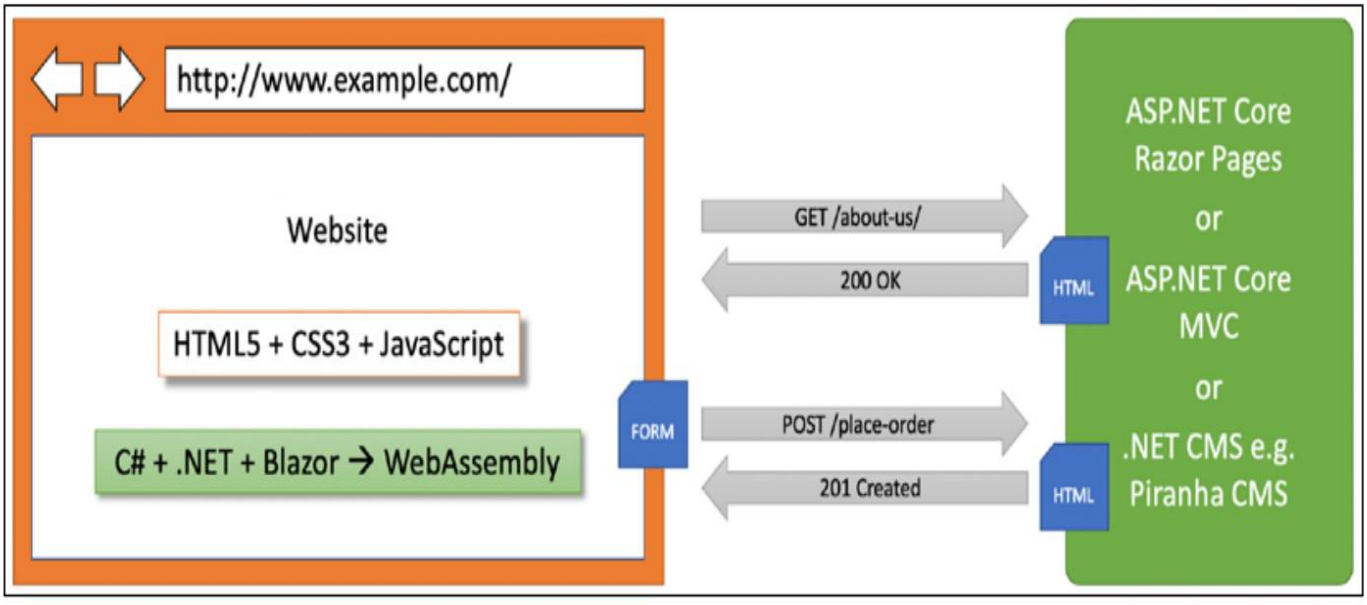
Building websites using ASP.NET Core (Chs.14-15)

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.

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

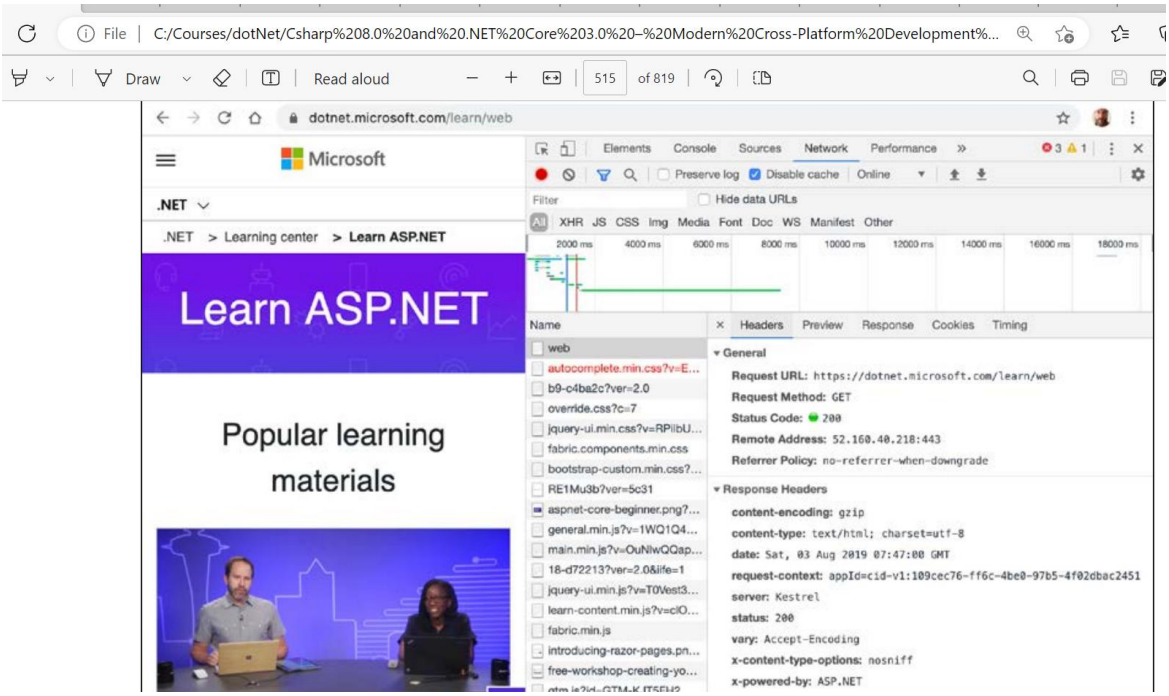
of a single web page built with a frontend technology such as Angular, React, Vue, or a proprietary JavaScript library that can make requests to a backend web service for getting more data when needed and posting updated data, using common serialization formats, such as XML and JSON. The canonical examples 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:



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

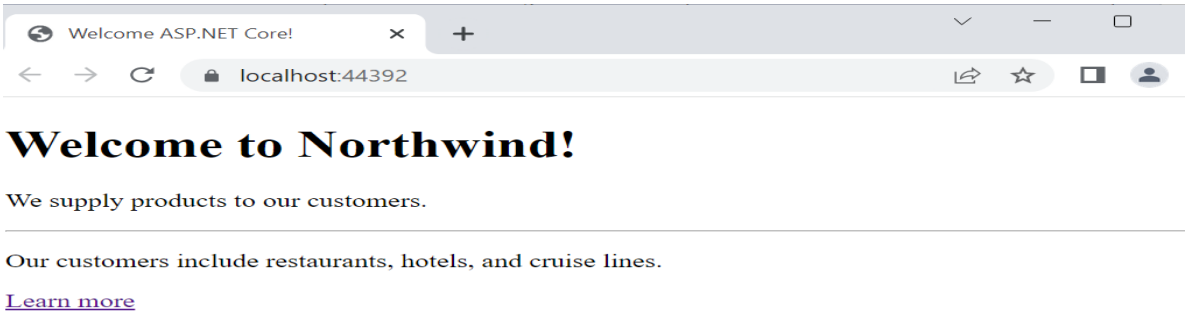
1. In the webapp folder, add a new folder named wwwroot.
2. Add a new file to the wwwroot folder named index.html or default.html.
3. Write the html content as shown in VS.

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();
});
```

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>
```

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.