Microsoft

Blazor for ASP.NET Web Forms Developers

Daniel Roth

Jeff Fritz

Taylor Southwick

Version 6.0

Microsoft

DOWNLOAD available at: https://aka.ms/blazor-ebook

EDITION v7.0 - Updated to .NET 7

Refer to changelog for the book updates and community contributions.

PUBLISHED BY

Microsoft Developer Division, .NET, and Visual Studio product teams

A division of Microsoft Corporation

One Microsoft Way

Redmond, Washington 98052-6399

Authors:

Daniel Roth, Principal Program Manager, Microsoft Corp.

Jeff Fritz, Senior Program Manager, Microsoft Corp.

Taylor Southwick, Senior Software Engineer, Microsoft Corp.

Scott Addie, Senior Content Developer, Microsoft Corp.

Steve "@ardalis" Smith, Software Architect and Trainer, NimblePros.com

Introduction

.NET has long supported web app development through ASP.NET, a comprehensive set of frameworks and tools for building any kind of web app. ASP.NET has its own lineage of web frameworks and technologies starting all the way back with classic Active Server Pages (ASP). Frameworks like ASP.NET Web Forms, ASP.NET MVC, ASP.NET Web Pages, and more recently ASP.NET Core, provide a productive and powerful way to build *server-rendered* web apps, where UI content is dynamically generated on the server in response to HTTP requests. Each ASP.NET framework caters to a different audience and app building philosophy. ASP.NET Web Forms shipped with the original release of the .NET Framework and enabled web development using many of the patterns familiar to desktop developers, like reusable UI controls with simple event handling. However, none of the ASP.NET offerings provide a way to run code that executed in the user's browser. To do that requires writing JavaScript and using any of the many JavaScript frameworks and tools that have phased in and out of popularity over the years: jQuery, Knockout, Angular, React, and so on.

Blazor is a new web framework that changes what is possible when building web apps with .NET. Blazor is a client-side web UI framework based on C# instead of JavaScript. With Blazor you can write your client-side logic and UI components in C#, compile them into normal .NET assemblies, and then run them directly in the browser using a new open web standard called WebAssembly. Or alternatively, Blazor can run your .NET UI components on the server and handle all UI interactions fluidly over a real-time connection with the browser. When paired with .NET running on the server, Blazor enables full-stack web development with .NET. While Blazor shares many commonalities with ASP.NET Web Forms, like having a reusable component model and a simple way to handle user events, it also builds on the foundations of .NET to provide a modern and high-performance web development experience.

This book introduces ASP.NET Web Forms developers to Blazor in a way that is familiar and convenient. It introduces Blazor concepts in parallel with analogous concepts in ASP.NET Web Forms while also explaining new concepts that may be less familiar. It covers a broad range of topics and concerns including component authoring, routing, layout, configuration, and security. And while the content of this book is primarily for enabling new development, it also covers guidelines and strategies for migrating existing ASP.NET Web Forms to Blazor for when you want to modernize an existing app.

Who should use the book

This book is for ASP.NET Web Forms developers looking for an introduction to Blazor that relates to their existing knowledge and skills. This book can help with quickly getting started on a new Blazor-based project or to help chart a roadmap for modernizing an existing ASP.NET Web Forms application.

How to use the book

The first part of this book covers what Blazor is and compares it to web app development with ASP.NET Web Forms. The book then covers a variety of Blazor topics, chapter by chapter, and relates

each Blazor concept to the corresponding concept in ASP.NET Web Forms, or explains fully any completely new concepts. The book also refers regularly to a complete sample app implemented in both ASP.NET Web Forms and Blazor to demonstrate Blazor features and to provide a case study for migrating from ASP.NET Web Forms to Blazor. You can find both implementations of the sample app (ASP.NET Web Forms and Blazor versions) on GitHub.

What this book doesn't cover

This book is an introduction to Blazor, not a comprehensive migration guide. While it does include guidance on how to approach migrating a project from ASP.NET Web Forms to Blazor, it does not attempt to cover every nuance and detail. For more general guidance on migrating from ASP.NET to ASP.NET Core, refer to the migration guidance in the ASP.NET Core documentation.

Additional resources

You can find the official Blazor home page and documentation at https://blazor.net.

Contents

CHAPTER 1

An introduction to Blazor for ASP.NET Web Forms developers

The ASP.NET Web Forms framework has been a staple of .NET web development since the .NET Framework first shipped in 2002. Back when the Web was still largely in its infancy, ASP.NET Web Forms made building web apps simple and productive by adopting many of the patterns that were used for desktop development. In ASP.NET Web Forms, web pages can be quickly composed from reusable UI controls. User interactions are handled naturally as events. There's a rich ecosystem of Web Forms UI controls provided by Microsoft and control vendors. The controls ease the efforts of connecting to data sources and displaying rich data visualizations. For the visually inclined, the Web Forms designer provides a simple drag-and-drop interface for managing controls.

Over the years, Microsoft has introduced new ASP.NET-based web frameworks to address web development trends. Some such web frameworks include ASP.NET MVC, ASP.NET Web Pages, and more recently ASP.NET Core. With each new framework, some have predicted the imminent decline of ASP.NET Web Forms and criticized it as an outdated, outmoded web framework. Despite these predictions, many .NET web developers continue to find ASP.NET Web Forms a simple, stable, and productive way to get their work done.

At the time of writing, almost half a million web developers use ASP.NET Web Forms every month. The ASP.NET Web Forms framework is stable to the point that docs, samples, books, and blog posts from a decade ago remain useful and relevant. For many .NET web developers, "ASP.NET" is still synonymous with "ASP.NET Web Forms" as it was when .NET was first conceived. Arguments on the pros and cons of ASP.NET Web Forms compared to the other new .NET web frameworks may rage on. ASP.NET Web Forms remains a popular framework for creating web apps.

Even so, innovations in software development aren't slowing. All software developers need to stay abreast of new technologies and trends. Two trends in particular are worth considering:

1. The shift to open-source and cross-platform
2. The shift of app logic to the client

An open-source and cross-platform .NET

When .NET and ASP.NET Web Forms first shipped, the platform ecosystem looked much different than it does today. The desktop and server markets were dominated by Windows. Alternative platforms like macOS and Linux were still struggling to gain traction. ASP.NET Web Forms ships with the .NET Framework as a Windows-only component, which means ASP.NET Web Forms apps can only run on Windows Server machines. Many modern environments now use different kinds of platforms for servers and development machines such that cross-platform support for many users is an absolute requirement.

Most modern web frameworks are now also open-source, which has a number of benefits. Users aren't beheld to a single project owner to fix bugs and add features. Open-source projects provide improved transparency on development progress and upcoming changes. Open-source projects enjoy contributions from an entire community, and they foster a supportive open-source ecosystem. Despite the risks of open-source, many consumers and contributors have found suitable mitigations that enable them to enjoy the benefits of an open-source ecosystem in a safe and reasonable way. Examples of such mitigations include contributor license agreements, friendly licenses, pedigree scans, and supporting foundations.

The .NET community has embraced both cross-platform support and open-source. .NET Core is an open-source and cross-platform implementation of .NET that runs on a plethora of platforms, including Windows, macOS, and various Linux distributions. Xamarin provides Mono, an open-source version of .NET. Mono runs on Android, iOS, and a variety of other form factors, including watches and smart TVs. In 2020, Microsoft released .NET 5 that reconciled .NET Core and Mono into "a single .NET runtime and framework that can be used everywhere and that has uniform runtime behaviors and developer experiences."

Will ASP.NET Web Forms benefit from the move to open-source and cross-platform support? The answer, unfortunately, is no, or at least not to the same extent as the rest of the platform. The .NET team made it clear that ASP.NET Web Forms won't be ported to .NET Core or .NET 7. Why is that?

There were efforts in the early days of .NET Core to port ASP.NET Web Forms. The number of breaking changes required were found to be too drastic. There's also an admission here that even for Microsoft, there's a limit to the number of web frameworks that it can support simultaneously. Perhaps someone in the community will take up the cause of creating an open-source and cross-platform version of ASP.NET Web Forms. The source code for ASP.NET Web Forms has been made available publicly in reference form. But for the time being, it seems ASP.NET Web Forms will remain Windows-only and without an open-source contribution model. If cross-platform support or open-source become important for your scenarios, then you'll need to look for something new.

Does this mean ASP.NET Web Forms is *dead* and should no longer be used? Of course not! As long as the .NET Framework ships as part of Windows, ASP.NET Web Forms will be a supported framework. For many Web Forms developers, the lack of cross-platform and open-source support is a non-issue. If you don't have a requirement for cross-platform support, open-source, or any of the other new features in .NET Core or .NET 7, then sticking with ASP.NET Web Forms on Windows is fine. ASP.NET Web Forms will continue to be a productive way to write web apps for many years to come.

But there's another trend worth considering, and that's the shift to the client.

Client-side web development

All of the .NET-based web frameworks, including ASP.NET Web Forms, have historically had one thing in common: they're *server-rendered*. In server-rendered web apps, the browser makes a request to the server, which executes some code (.NET code in ASP.NET apps) to produce a response. That response is sent back to the browser to handle. In this model, the browser is used as a thin rendering engine. The hard work of producing the UI, running the business logic, and managing state occurs on the server.

However, browsers have become versatile platforms. They implement an ever-increasing number of open web standards that grant access to the capabilities of the user's machine. Why not take advantage of the compute power, storage, memory, and other resources of the client device? UI interactions in particular can benefit from a richer and more interactive feel when handled at least partially or completely client-side. Logic and data that should be handled on the server can still be handled server-side. Web API calls or even real-time protocols, like WebSockets, can be used. These benefits are available to web developers for free if they're willing to write JavaScript. Client-side UI frameworks, such as Angular, React, and Vue, simplify client-side web development and have grown in popularity. ASP.NET Web Forms developers can also benefit from leveraging the client, and even have some out-of-the-box support with integrated JavaScript frameworks like ASP.NET AJAX.

But bridging two different platforms and ecosystems (.NET and JavaScript) comes with a cost. Expertise is required in two parallel worlds with different languages, frameworks, and tools. Code and logic can't be easily shared between client and server, resulting in duplication and engineering overhead. It can also be difficult to keep up with the JavaScript ecosystem, which has a history of evolving at breakneck speed. Front-end framework and build tool preferences change quickly. The industry has observed the progression from Grunt to Gulp to Webpack, and so on. The same restless churn has occurred with front-end frameworks such as jQuery, Knockout, Angular, React, and Vue. But given JavaScript's browser monopoly, there was little choice in the matter. That is, until the web community got together and caused a *miracle* to happen!

WebAssembly fulfills a need

In 2015, the major browser vendors joined forces in a W3C Community Group to create a new open web standard called WebAssembly. WebAssembly is a byte code for the Web. If you can compile your code to WebAssembly, it can then run on any browser on any platform at near native speed. Initial efforts focused on C/C++. The result was a dramatic demonstration of running native 3D graphics engines directly in the browser without plugins. WebAssembly has since been standardized and implemented by all major browsers.

Work on running .NET on WebAssembly was announced in late 2017 and released in 2020, including support in .NET 5 and beyond. The ability to run .NET code directly in the browser enables full-stack web development with .NET.

Blazor: full-stack web development with .NET

On its own, the ability to run .NET code in a browser doesn't provide an end-to-end experience for creating client-side web apps. That's where Blazor comes in. Blazor is a client-side web UI framework based on C# instead of JavaScript. Blazor can run directly in the browser via WebAssembly. No browser plugins are required. Alternatively, Blazor apps can run server-side on .NET and handle all user interactions over a real-time connection with the browser.

Blazor has great tooling support in Visual Studio and Visual Studio Code. The framework also includes a full UI component model and has built-in facilities for:

- Forms and validation
- Dependency injection
- Client-side routing
- Layouts
- In-browser debugging
- JavaScript interop

Blazor has a lot in common with ASP.NET Web Forms. Both frameworks offer component-based, event-driven, stateful UI programming models. The main architectural difference is that ASP.NET Web Forms runs only on the server. Blazor can run on the client in the browser. But if you're coming from an ASP.NET Web Forms background, there's a lot in Blazor that will feel familiar. Blazor is a natural solution for ASP.NET Web Forms developers looking for a way to take advantage of client-side development and the open-source, cross-platform future of .NET.

This book provides an introduction to Blazor that is catered specifically to ASP.NET Web Forms developers. Each Blazor concept is presented in the context of analogous ASP.NET Web Forms features and practices. By the end of this book, you'll have an understanding of:

- How to build Blazor apps.
- How Blazor works.
- How Blazor relates to .NET.
- Reasonable strategies for migrating existing ASP.NET Web Forms apps to Blazor where appropriate.

Get started with Blazor

Getting started with Blazor is easy. Go to https://blazor.net and follow the links to install the appropriate .NET SDK and Blazor project templates. You'll also find instructions for setting up the Blazor tooling in Visual Studio or Visual Studio Code.

CHAPTER 2

Architecture comparison of ASP.NET Web Forms and Blazor

While ASP.NET Web Forms and Blazor have many similar concepts, there are differences in how they work. This chapter examines the inner workings and architectures of ASP.NET Web Forms and Blazor.

ASP.NET Web Forms

The ASP.NET Web Forms framework is based on a page-centric architecture. Each HTTP request for a location in the app is a separate page with which ASP.NET responds. As pages are requested, the contents of the browser are replaced with the results of the page requested.

Pages consist of the following components:

- HTML markup
- C# or Visual Basic code
- A code-behind class containing logic and event-handling capabilities
- Controls

Controls are reusable units of web UI that can be programmatically placed and interacted with on a page. Pages are composed of files that end with *.aspx* containing markup, controls, and some code. The code-behind classes are in files with the same base name and an *.aspx.cs* or *.aspx.vb* extension, depending on the programming language used. Interestingly, the web server interprets contents of the *.aspx* files and compiles them whenever they change. This recompilation occurs even if the web server is already running.

Controls can be built with markup and delivered as user controls. A user control derives from the `UserControl` class and has a similar structure to the Page. Markup for user controls is stored in an *.ascx* file. An accompanying code-behind class resides in an *.ascx.cs* or *.ascx.vb* file. Controls can also be built completely with code, by inheriting from either the `WebControl` or `CompositeControl` base class.

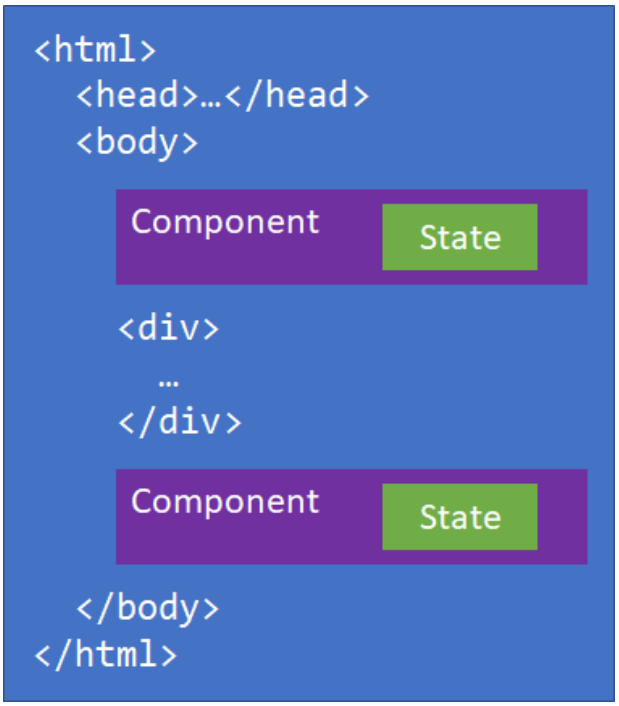
Pages also have an extensive event lifecycle. Each page raises events for the initialization, load, prerender, and unload events that occur as the ASP.NET runtime executes the page's code for each request.

Controls on a Page typically post-back to the same page that presented the control, and carry along with them a payload from a hidden form field called `ViewState`. The `ViewState` field contains information about the state of the controls at the time they were rendered and presented on the page, allowing the ASP.NET runtime to compare and identify changes in the content submitted to the server.

Blazor

Blazor is a client-side web UI framework similar in nature to JavaScript front-end frameworks like Angular or React. Blazor handles user interactions and renders the necessary UI updates. Blazor *isn't* based on a request-reply model. User interactions are handled as events that aren't in the context of any particular HTTP request.

Blazor apps consist of one or more root components that are rendered on an HTML page.

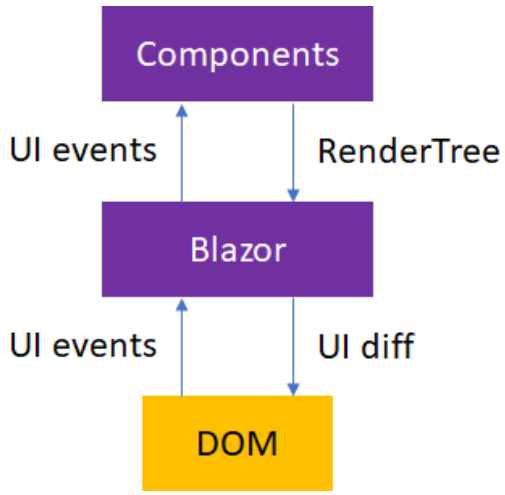


How the user specifies where components should render and how the components are then wired up for user interactions is hosting model specific.

Blazor components are .NET classes that represent a reusable piece of UI. Each component maintains its own state and specifies its own rendering logic, which can include rendering other components. Components specify event handlers for specific user interactions to update the component's state.

After a component handles an event, Blazor renders the component and keeps track of what changed in the rendered output. Components don't render directly to the Document Object Model (DOM).

They instead render to an in-memory representation of the DOM called a `RenderTree` so that Blazor can track the changes. Blazor compares the newly rendered output with the previous output to calculate a UI diff that it then applies efficiently to the DOM.



Components can also manually indicate that they should be rendered if their state changes outside of a normal UI event. Blazor uses a `SynchronizationContext` to enforce a single logical thread of execution. A component's lifecycle methods and any event callbacks that are raised by Blazor are executed on this `SynchronizationContext`.

Blazor app hosting models

Blazor apps can be hosted in one of the following ways:

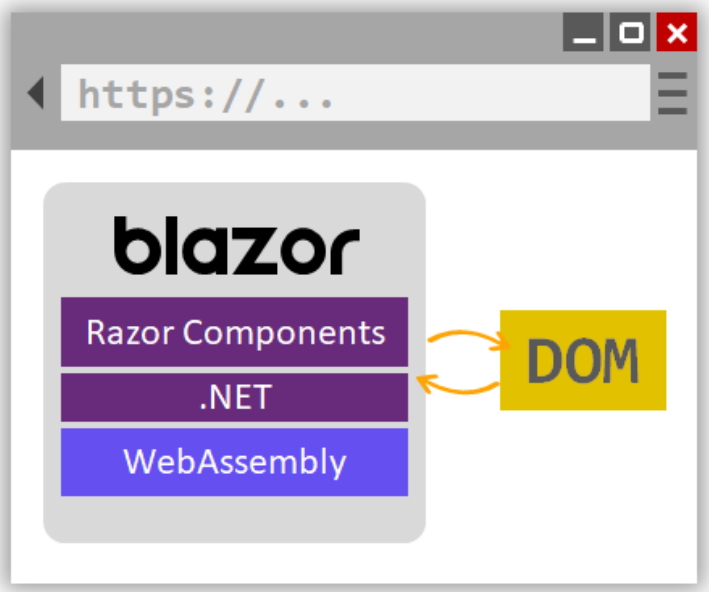
- Client-side in the browser on WebAssembly.
- Server-side in an ASP.NET Core app.

Blazor WebAssembly apps

Blazor WebAssembly apps execute directly in the browser on a WebAssembly-based .NET runtime. Blazor WebAssembly apps function in a similar way to front-end JavaScript frameworks like Angular or React. However, instead of writing JavaScript you write C#. The .NET runtime is downloaded with the app along with the app assembly and any required dependencies. No browser plugins or extensions are required.

The downloaded assemblies are normal .NET assemblies, like you would use in any other .NET app. Because the runtime supports .NET Standard, you can use existing .NET Standard libraries with your Blazor WebAssembly app. However, these assemblies will still execute in the browser security sandbox. Some functionality may throw a PlatformNotSupportedException, like trying to access the file system or opening arbitrary network connections.

When the app loads, the .NET runtime is started and pointed at the app assembly. The app startup logic runs, and the root components are rendered. Blazor calculates the UI updates based on the rendered output from the components. The DOM updates are then applied.



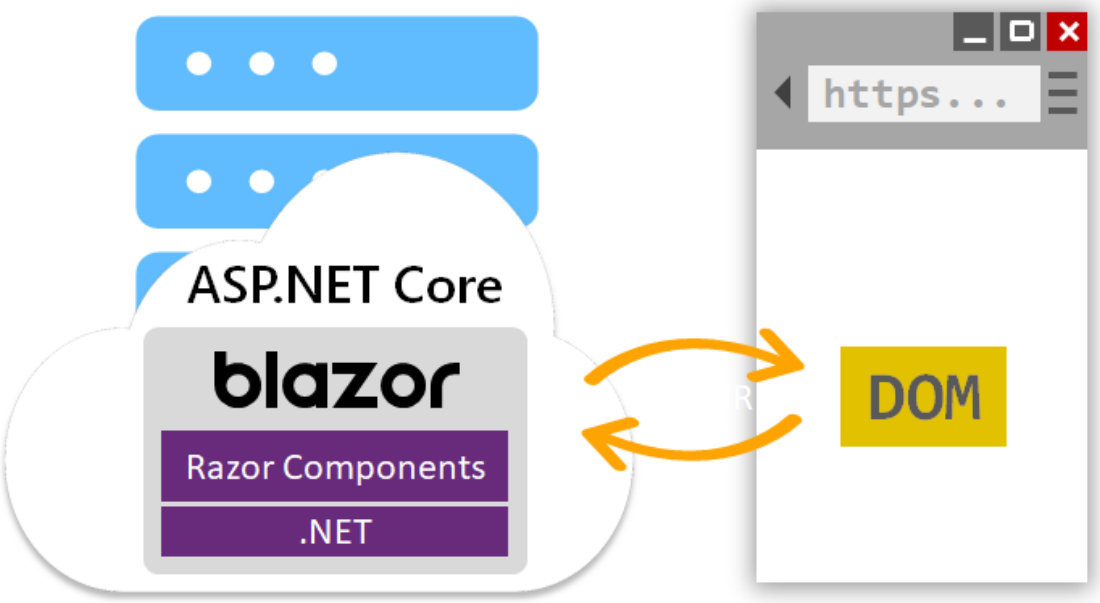
Blazor WebAssembly apps run purely client-side. Such apps can be deployed to static site hosting solutions like GitHub Pages or Azure Static Website Hosting. .NET isn't required on the server at all. Deep linking to parts of the app typically requires a routing solution on the server. The routing solution redirects requests to the root of the app. For example, this redirection can be handled using URL rewrite rules in IIS.

To get all the benefits of Blazor and full-stack .NET web development, host your Blazor WebAssembly app with ASP.NET Core. By using .NET on both the client and server, you can easily share code and build your app using one consistent set of languages, frameworks, and tools. Blazor provides convenient templates for setting up a solution that contains both a Blazor WebAssembly app and an ASP.NET Core host project. When the solution is built, the built static files from the Blazor app are hosted by the ASP.NET Core app with fallback routing already setup.

Blazor Server apps

Recall from the Blazor architecture discussion that Blazor components render their output to an intermediate abstraction called a `RenderTree`. The Blazor framework then compares what was rendered with what was previously rendered. The differences are applied to the DOM. Blazor components are decoupled from how their rendered output is applied. Consequently, the components themselves don't have to run in the same process as the process updating the UI. In fact, they don't even have to run on the same machine.

In Blazor Server apps, the components run on the server instead of client-side in the browser. UI events that occur in the browser are sent to the server over a real-time connection. The events are dispatched to the correct component instances. The components render, and the calculated UI diff is serialized and sent to the browser where it's applied to the DOM.



The Blazor Server hosting model may sound familiar if you've used ASP.NET AJAX and the UpdatePanel control. The `UpdatePanel` control handles applying partial page updates in response to trigger events on the page. When triggered, the `UpdatePanel` requests a partial update and then applies it without needing to refresh the page. The state of the UI is managed using `ViewState`. Blazor Server apps are slightly different in that the app requires an active connection with the client. Additionally, all UI state is maintained on the server. Aside from those differences, the two models are conceptually similar.

How to choose the right Blazor hosting model

As described in the Blazor hosting model docs, the different Blazor hosting models have different tradeoffs.

The Blazor WebAssembly hosting model has the following benefits:

- There's no .NET server-side dependency. The app is fully functioning after downloaded to the client.
- Client resources and capabilities are fully leveraged.
- Work is offloaded from the server to the client.
- An ASP.NET Core web server isn't required to host the app. Serverless deployment scenarios are possible (for example, serving the app from a CDN).

The downsides of the Blazor WebAssembly hosting model are:

- Browser capabilities restrict the app.
- Capable client hardware and software (for example, WebAssembly support) is required.
- Download size is larger, and apps take longer to load.

- .NET runtime and tooling support is less mature. For example, there are limitations in .NET Standard support and debugging.

Conversely, the Blazor Server hosting model offers the following benefits:

- Download size is much smaller than a client-side app, and the app loads much faster.
- The app takes full advantage of server capabilities, including use of any .NET compatible APIs.
- .NET on the server is used to run the app, so existing .NET tooling, such as debugging, works as expected.
- Thin clients are supported. For example, server-side apps work with browsers that don't support WebAssembly and on resource-constrained devices.
- The app's .NET/C# code base, including the app's component code, isn't served to clients.

The downsides to the Blazor Server hosting model are:

- Higher UI latency. Every user interaction involves a network hop.
- There's no offline support. If the client connection fails, the app stops working.
- Scalability is challenging for apps with many users. The server must manage multiple client connections and handle client state.
- An ASP.NET Core server is required to serve the app. Serverless deployment scenarios aren't possible. For example, you can't serve the app from a CDN.

The preceding list of trade-offs may be intimidating, but your hosting model can be changed later. Regardless of the Blazor hosting model selected, the component model is *the same*. In principle, the same components can be used with either hosting model. Your app code doesn't change; however, it's a good practice to introduce abstractions so that your components stay hosting model-agnostic. The abstractions allow your app to more easily adopt a different hosting model.

Deploy your app

ASP.NET Web Forms apps are typically hosted on IIS on a Windows Server machine or cluster. Blazor apps can also:

- Be hosted on IIS, either as static files or as an ASP.NET Core app.
- Leverage ASP.NET Core's flexibility to be hosted on various platforms and server infrastructures. For example, you can host a Blazor App using Nginx or Apache on Linux. For more information about how to publish and deploy Blazor apps, see the Blazor Hosting and deployment documentation.

In the next section, we'll look at how the projects for Blazor WebAssembly and Blazor Server apps are set up.

Project structure for Blazor apps

Despite their significant project structure differences, ASP.NET Web Forms and Blazor share many similar concepts. Here, we'll look at the structure of a Blazor project and compare it to an ASP.NET Web Forms project.

To create your first Blazor app, follow the instructions in the Blazor getting started steps. You can follow the instructions to create either a Blazor Server app or a Blazor WebAssembly app hosted in ASP.NET Core. Except for the hosting model-specific logic, most of the code in both projects is the same.

Project file

Blazor Server apps are .NET projects. The project file for the Blazor Server app is about as simple as it can get:

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>net7.0</TargetFramework>
  </PropertyGroup>

</Project>
```

The project file for a Blazor WebAssembly app looks slightly more involved (exact version numbers may vary):

```
<Project Sdk="Microsoft.NET.Sdk.BlazorWebAssembly">

  <PropertyGroup>
    <TargetFramework>net7.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.Components.WebAssembly" Version="7.0.0"
/>
    <PackageReference Include="Microsoft.AspNetCore.Components.WebAssembly.DevServer"
Version="7.0.0" PrivateAssets="all" />
  </ItemGroup>
```

```
</Project>
```

Blazor WebAssembly project targets `Microsoft.NET.Sdk.BlazorWebAssembly` instead of `Microsoft.NET.Sdk.Web` sdk because they run in the browser on a WebAssembly-based .NET runtime. You can't install .NET into a web browser like you can on a server or developer machine. Consequently, the project references the Blazor framework using individual package references.

By comparison, a default ASP.NET Web Forms project includes almost 300 lines of XML in its *.csproj* file, most of which is explicitly listing the various code and content files in the project. Since the release of .NET 5, both Blazor Server and Blazor WebAssembly apps can easily share one unified runtime.

Although they're supported, individual assembly references are less common in .NET projects. Most project dependencies are handled as NuGet package references. You only need to reference top-level package dependencies in .NET projects. Transitive dependencies are included automatically. Instead of using the *packages.config* file commonly found in ASP.NET Web Forms projects to reference packages, package references are added to the project file using the `<PackageReference>` element.

```
<ItemGroup>
  <PackageReference Include="Newtonsoft.Json" Version="13.0.2" />
</ItemGroup>
```

Entry point

The Blazor Server app's entry point is defined in the *Program.cs* file, as you would see in a Console app. When the app executes, it creates and runs a web host instance using defaults specific to web apps. The web host manages the Blazor Server app's lifecycle and sets up host-level services. Examples of such services are configuration, logging, dependency injection, and the HTTP server. This code is mostly boilerplate and is often left unchanged.

```
using BlazorApp3.Areas.Identity;
using BlazorApp3.Data;
using Microsoft.AspNetCore.Components.Authorization;
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
var connectionString = builder.Configuration.GetConnectionString("DefaultConnection");
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(connectionString));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();
builder.Services.AddDefaultIdentity<IdentityUser>(options =>
options.SignIn.RequireConfirmedAccount = true)
    .AddEntityFrameworkStores<ApplicationDbContext>();
builder.Services.AddRazorPages();
builder.Services.AddServerSideBlazor();
builder.Services.AddScoped<AuthenticationStateProvider,
RevalidatingIdentityAuthenticationStateProvider<IdentityUser>>();
builder.Services.AddSingleton<WeatherForecastService>();
```

```
var app = builder.Build();

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.UseMigrationsEndPoint();
}
else
{
    app.UseExceptionHandler("/Error");
    // The default HSTS value is 30 days. You may want to change this for production
scenarios, see https://aka.ms/aspnetcore-hsts.
    app.UseHsts();
}

app.UseHttpsRedirection();

app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapControllers();
app.MapBlazorHub();
app.MapFallbackToPage("/_Host");

app.Run();
```

Blazor WebAssembly apps also define an entry point in *Program.cs*. The code looks slightly different. The code is similar in that it's setting up the app host to provide the same host-level services to the app. The WebAssembly app host doesn't, however, set up an HTTP server because it executes directly in the browser.

Blazor apps don't use a *Global.asax* file to define the startup logic for the app. Instead, this logic is contained in *Program.cs* or in a related `Startup` class that is referenced from *Program.cs*. Either way, this code is used to configure the app and any app-specific services.

In a Blazor Server app, the *Program.cs* file shown is used to set up the endpoint for the real-time connection used by Blazor between the client browsers and the server.

In a Blazor WebAssembly app, the *Program.cs* file defines the root components for the app and where they should be rendered:

```
using BlazorApp1;
using Microsoft.AspNetCore.Components.Web;
using Microsoft.AspNetCore.Components.WebAssembly.Hosting;

var builder = WebAssemblyHostBuilder.CreateDefault(args);
builder.RootComponents.Add<App>("#app");
builder.RootComponents.Add<HeadOutlet>("head::after");

builder.Services.AddScoped(sp => new HttpClient { BaseAddress = new
Uri(builder.HostEnvironment.BaseAddress) });

await builder.Build().RunAsync();
```

Static files

Unlike ASP.NET Web Forms projects, not all files in a Blazor project can be requested as static files. Only the files in the *wwwroot* folder are web-addressable. This folder is referred to as the app's "web root". Anything outside of the app's web root *isn't* web-addressable. This setup provides an additional level of security that prevents accidental exposure of project files over the web.

Configuration

Configuration in ASP.NET Web Forms apps is typically handled using one or more *web.config* files. Blazor apps don't typically have *web.config* files. If they do, the file is only used to configure IIS-specific settings when hosted on IIS. Instead, Blazor Server apps use the ASP.NET Core configuration abstractions. (Blazor WebAssembly apps don't currently support the same configuration abstractions, but that may be a feature added in the future.) For example, the default Blazor Server app stores some settings in *appsettings.json*.

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*"
}
```

You'll learn more about configuration in ASP.NET Core projects in the Configuration section.

Razor components

Most files in Blazor projects are *.razor* files. Razor is a templating language based on HTML and C# that is used to dynamically generate web UI. The *.razor* files define components that make up the UI of the app. For the most part, the components are identical for both the Blazor Server and Blazor WebAssembly apps. Components in Blazor are analogous to user controls in ASP.NET Web Forms.

Each Razor component file is compiled into a .NET class when the project is built. The generated class captures the component's state, rendering logic, lifecycle methods, event handlers, and other logic. You'll learn more about authoring components in the Building reusable UI components with Blazor section.

The *_Imports.razor* files aren't Razor component files. Instead, they define a set of Razor directives to import into other *.razor* files within the same folder and in its subfolders. For example, a *_Imports.razor* file is a conventional way to add `using` directives for commonly used namespaces:

```
@using System.Net.Http
@using System.Net.Http.Json
@using Microsoft.AspNetCore.Components.Forms
@using Microsoft.AspNetCore.Components.Routing
```

```
@using Microsoft.AspNetCore.Components.Web
@using Microsoft.AspNetCore.Components.Web.Virtualization
@using Microsoft.AspNetCore.Components.WebAssembly.Http
@using Microsoft.JSInterop
@using BlazorApp1
@using BlazorApp1.Shared
```

Pages

Where are the pages in the Blazor apps? Blazor doesn't define a separate file extension for addressable pages, like the *.aspx* files in ASP.NET Web Forms apps. Instead, pages are defined by assigning routes to components. A route is typically assigned using the `@page` Razor directive. For example, the `Counter` component authored in the *Pages/Counter.razor* file defines the following route:

```
@page "/counter"
```

Routing in Blazor is handled client-side, not on the server. As the user navigates in the browser, Blazor intercepts the navigation and then renders the component with the matching route.

The component routes aren't currently inferred by the component's file location like they are with *.aspx* pages or ASP.NET Core Razor Pages. This feature may be added in the future. Each route must be specified explicitly on the component. Storing routable components in a *Pages* folder has no special meaning and is purely a convention.

You'll learn more about routing in Blazor in the Pages, routing, and layouts section.

Layout

In ASP.NET Web Forms apps, a common page layout is handled using master pages (*Site.Master*). In Blazor apps, the page layout is handled using layout components (*Shared/MainLayout.razor*). Layout components are discussed in more detail in the Page, routing, and layouts section.

Bootstrap Blazor

To bootstrap Blazor, the app must:

- Specify where on the page the root component (*App.Razor*) should be rendered.
- Add the corresponding Blazor framework script.

In the Blazor Server app, the root component's host page is defined in the *_Host.cshtml* file. This file defines a Razor Page, not a component. Razor Pages use Razor syntax to define a server-addressable page, very much like an *.aspx* page.

```
@page "/"
@namespace BlazorApp3.Pages
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@{
    Layout = "_Layout";
```

```
}

<component type="typeof(App)" render-mode="ServerPrerendered" />
```

The `render-mode` attribute is used to define where a root-level component should be rendered. The `RenderMode` option indicates the manner in which the component should be rendered. The following table outlines the supported `RenderMode` options.

Option	Description
`RenderMode.Server`	Rendered interactively once a connection with the browser is established
`RenderMode.ServerPrerendered`	First prerendered and then rendered interactively
`RenderMode.Static`	Rendered as static content

The *_Layout.cshtml* file includes the default HTML for the app and its components.

```
@using Microsoft.AspNetCore.Components.Web
@namespace BlazorApp3.Pages
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <base href="~/" />
    <link rel="stylesheet" href="css/bootstrap/bootstrap.min.css" />
    <link href="css/site.css" rel="stylesheet" />
    <link href="BlazorApp3.styles.css" rel="stylesheet" />
    <component type="typeof(HeadOutlet)" render-mode="ServerPrerendered" />
</head>
<body>
    @RenderBody()

    <div id="blazor-error-ui">
        <environment include="Staging,Production">
            An error has occurred. This application may no longer respond until reloaded.
        </environment>
        <environment include="Development">
            An unhandled exception has occurred. See browser dev tools for details.
        </environment>
        <a href="" class="reload">Reload</a>
        <a class="dismiss">✕</a>
    </div>

    <script src="_framework/blazor.server.js"></script>
</body>
</html>
```

The script reference to *_framework/blazor.server.js* establishes the real-time connection with the server and then deals with all user interactions and UI updates.

In the Blazor WebAssembly app, the host page is a simple static HTML file under *wwwroot/index.html*. The `<div>` element with id named `app` is used to indicate where the root component should be rendered.

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

<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0, maximum-
scale=1.0, user-scalable=no" />
    <title>BlazorApp1</title>
    <base href="/" />
    <link href="css/bootstrap/bootstrap.min.css" rel="stylesheet" />
    <link href="css/app.css" rel="stylesheet" />
    <link href="BlazorApp1.styles.css" rel="stylesheet" />
</head>

<body>
    <div id="app">Loading...</div>

    <div id="blazor-error-ui">
        An unhandled error has occurred.
        <a href="" class="reload">Reload</a>
        <a class="dismiss">✕</a>
    </div>
    <script src="_framework/blazor.webassembly.js"></script>
</body>

</html>
```

The root component to render is specified in the app's *Program.cs* file with the flexibility to register services through dependency injection. For more information, see ASP.NET Core Blazor dependency injection.

```
var builder = WebAssemblyHostBuilder.CreateDefault(args);
builder.RootComponents.Add<App>("#app");
builder.RootComponents.Add<HeadOutlet>("head::after");
```

Build output

When a Blazor project is built, all Razor component and code files are compiled into a single assembly. Unlike ASP.NET Web Forms projects, Blazor doesn't support runtime compilation of the UI logic.

Run the app with Hot Reload

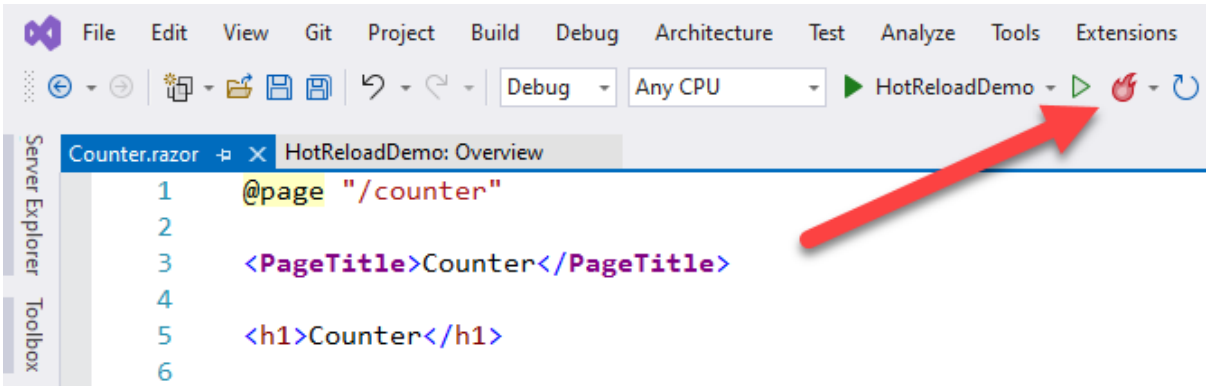
To run the Blazor Server app, press F5 in Visual Studio to run with the debugger attached, or Ctrl + F5 to run without the debugger attached.

To run the Blazor WebAssembly app, choose one of the following approaches:

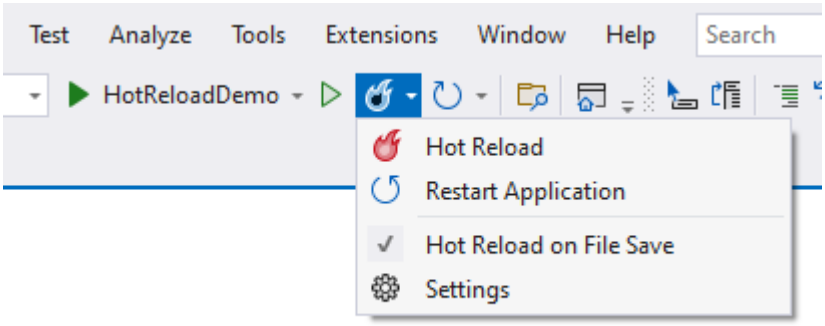
- Run the client project directly using the development server.
- Run the server project when hosting the app with ASP.NET Core.

Blazor WebAssembly apps can be debugged in both browser and Visual Studio. See Debug ASP.NET Core Blazor WebAssembly for details.

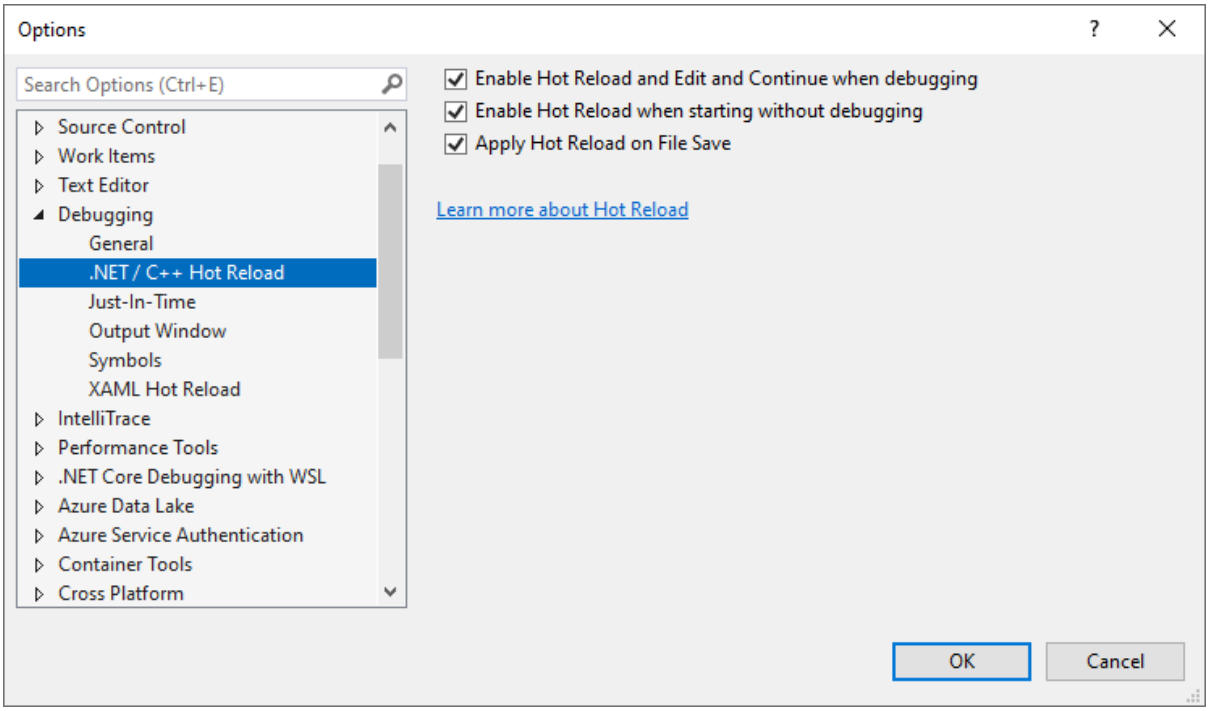
Both Blazor Server and Blazor WebAssembly apps support Hot Reload in Visual Studio. Hot Reload is a feature that automatically updates changes made to a Blazor app live, in the browser. You can toggle whether Hot Reload is enabled from its icon in the toolbar:



Selecting the caret beside the icon reveals additional options. You can toggle Hot Reload on or off, restart the application, and toggle whether Hot Reload should occur whenever a file is saved.



You can also access additional configuration options. The configuration dialog lets you specify whether Hot Reload should be enabled when debugging (along with Edit and Continue), when starting without debugging, or when a file is saved.



The "developer inner loop" has been greatly streamlined with Hot Reload. Without Hot Reload, a Blazor developer would typically need to restart and rerun the app after every change, navigating to the appropriate part of the app as required. With Hot Reload, changes can be made to the running app without the need to restart in most cases. Hot Reload even retains the state of pages, so there's no need to have to re-enter form values or otherwise get the app back where you need it.

CHAPTER 5

App startup

Applications that are written for ASP.NET typically have a `global.asax.cs` file that defines the `Application_Start` event that controls which services are configured and made available for both HTML rendering and .NET processing. This chapter looks at how things are slightly different with ASP.NET Core and Blazor Server.

Application_Start and Web Forms

The default web forms `Application_Start` method has grown in purpose over years to handle many configuration tasks. A fresh web forms project with the default template in Visual Studio 2022 now contains the following configuration logic:

- `RouteConfig` - Application URL routing
- `BundleConfig` - CSS and JavaScript bundling and minification

Each of these individual files resides in the `App_Start` folder and run only once at the start of our application. `RouteConfig` in the default project template adds the `FriendlyUrlSettings` for web forms to allow application URLs to omit the `.ASPX` file extension. The default template also contains a directive that provides permanent HTTP redirect status codes (HTTP 301) for the `.ASPX` pages to the friendly URL with the file name that omits the extension.

With ASP.NET Core and Blazor, these methods are either simplified and consolidated into the `Startup` class or they are eliminated in favor of common web technologies.

Blazor Server Startup Structure

Blazor Server applications reside on top of an ASP.NET Core 3.0 or later version. ASP.NET Core web applications are configured in *Program.cs*, or through a pair of methods in the `Startup.cs` class. A sample *Program.cs* file is shown below:

```
using BlazorApp1.Data;
using Microsoft.AspNetCore.Components;
using Microsoft.AspNetCore.Components.Web;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddRazorPages();
builder.Services.AddServerSideBlazor();
builder.Services.AddSingleton<WeatherForecastService>();
```

```
var app = builder.Build();

// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    // The default HSTS value is 30 days. You may want to change this for production
scenarios, see https://aka.ms/aspnetcore-hsts.
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();
app.UseRouting();
app.MapBlazorHub();
app.MapFallbackToPage("/_Host");

app.Run();
```

The app's required services are added to the `WebApplicationBuilder` instance's `Services` collection. This is how the various ASP.NET Core framework services are configured with the framework's built-in dependency injection container. The various `builder.Services.Add*` methods add services that enable features such as authentication, razor pages, MVC controller routing, SignalR, and Blazor Server interactions among many others. This method was not needed in web forms, as the parsing and handling of the ASPX, ASCX, ASHX, and ASMX files were defined by referencing ASP.NET in the web.config configuration file. More information about dependency injection in ASP.NET Core is available in the online documentation.

After the `app` has been built by the `builder`, the rest of the calls on `app` configure its HTTP pipeline. With these calls, we declare from top to bottom the Middleware that will handle every request sent to our application. Most of these features in the default configuration were scattered across the web forms configuration files and are now in one place for ease of reference.

No longer is the configuration of the custom error page placed in a `web.config` file, but now is configured to always be shown if the application environment is not labeled `Development`. Additionally, ASP.NET Core applications are now configured to serve secure pages with TLS by default with the `UseHttpsRedirection` method call.

Next, an unexpected configuration method call is made to `UseStaticFiles`. In ASP.NET Core, support for requests for static files (like JavaScript, CSS, and image files) must be explicitly enabled, and only files in the app's *wwwroot* folder are publicly addressable by default.

The next line is the first that replicates one of the configuration options from web forms: `UseRouting`. This method adds the ASP.NET Core router to the pipeline and it can be either configured here or in the individual files that it can consider routing to. More information about routing configuration can be found in the Routing section.

The final `app.Map*` calls in this section define the endpoints that ASP.NET Core is listening on. These routes are the web accessible locations that you can access on the web server and receive some content handled by .NET and returned to you. The first entry, `MapBlazorHub` configures a SignalR hub for use in providing the real-time and persistent connection to the server where the state and rendering of Blazor components is handled. The `MapFallbackToPage` method call indicates the web-accessible location of the page that starts the Blazor application and also configures the application to

handle deep-linking requests from the client-side. You will see this feature at work if you open a browser and navigate directly to Blazor handled route in your application, such as `/counter` in the default project template. The request gets handled by the *_Host.cshtml* fallback page, which then runs the Blazor router and renders the counter page.

The very last line starts the application, something that wasn't required in web forms (since it relied on IIS to be running).

Upgrading the BundleConfig Process

Technologies for bundling assets like CSS stylesheets and JavaScript files have changed significantly, with other technologies providing quickly evolving tools and techniques for managing these resources. To this end, we recommend using a Node command-line tool such as Grunt / Gulp / WebPack to package your static assets.

The Grunt, Gulp, and WebPack command-line tools and their associated configurations can be added to your application and ASP.NET Core will quietly ignore those files during the application build process. You can add a call to run their tasks by adding a `Target` inside your project file with syntax similar to the following that would trigger a gulp script and the `min` target inside that script:

```
<Target Name="MyPreCompileTarget" BeforeTargets="Build">
  <Exec Command="gulp min" />
</Target>
```

More details about both strategies to manage your CSS and JavaScript files are available in the Bundle and minify static assets in ASP.NET Core documentation.

CHAPTER 6

Build reusable UI components with Blazor

One of the beautiful things about ASP.NET Web Forms is how it enables encapsulation of reusable pieces of user interface (UI) code into reusable UI controls. Custom user controls can be defined in markup using *.ascx* files. You can also build elaborate server controls in code with full designer support.

Blazor also supports UI encapsulation through *components*. A component:

- Is a self-contained chunk of UI.
- Maintains its own state and rendering logic.
- Can define UI event handlers, bind to input data, and manage its own lifecycle.
- Is typically defined in a *.razor* file using Razor syntax.

An introduction to Razor

Razor is a light-weight markup templating language based on HTML and C#. With Razor, you can seamlessly transition between markup and C# code to define your component rendering logic. When the *.razor* file is compiled, the rendering logic is captured in a structured way in a .NET class. The name of the compiled class is taken from the *.razor* file name. The namespace is taken from the default namespace for the project and the folder path, or you can explicitly specify the namespace using the `@namespace` directive (more on Razor directives below).

A component's rendering logic is authored using normal HTML markup with dynamic logic added using C#. The `@` character is used to transition to C#. Razor is typically smart about figuring out when you've switched back to HTML. For example, the following component renders a `<p>` tag with the current time:

```
<p>@DateTime.Now</p>
```

To explicitly specify the beginning and ending of a C# expression, use parentheses:

```
<p>@(DateTime.Now)</p>
```

Razor also makes it easy to use C# control flow in your rendering logic. For example, you can conditionally render some HTML like this:

```
@if (value % 2 == 0)
{
    <p>The value was even.</p>
}
```

Or you can generate a list of items using a normal C# `foreach` loop like this:

```
<ul>
@foreach (var item in items)
{
    <li>@item.Text</li>
}
</ul>
```

Razor directives, like directives in ASP.NET Web Forms, control many aspects of how a Razor component is compiled. Examples include the component's:

- Namespace
- Base class
- Implemented interfaces
- Generic parameters
- Imported namespaces
- Routes

Razor directives start with the `@` character and are typically used at the start of a new line at the start of the file. For example, the `@namespace` directive defines the component's namespace:

```
@namespace MyComponentNamespace
```

The following table summarizes the various Razor directives used in Blazor and their ASP.NET Web Forms equivalents, if they exist.

Directive	Description	Example	Web Forms equivalent
`@attribute`	Adds a class-level attribute to the component	`@attribute [Authorize]`	None
`@code`	Adds class members to the component	`@code { ... }`	`<script runat="server">...</script>`
`@implements`	Implements the specified interface	`@implements IDisposable`	Use code-behind
`@inherits`	Inherits from the specified base class	`@inherits MyComponentBase`	`<%@ Control Inherits="MyUserControlBase" %>`
`@inject`	Injects a service into the component	`@inject IJSRuntime JS`	None

Directive	Description	Example	Web Forms equivalent
`@layout`	Specifies a layout component for the component	`@layout MainLayout`	`<%@ Page MasterPageFile="~/Site.Master" %>`
`@namespace`	Sets the namespace for the component	`@namespace MyNamespace`	None
`@page`	Specifies the route for the component	`@page "/product/{id}"`	`<%@ Page %>`
`@typeparam`	Specifies a generic type parameter for the component	`@typeparam TItem`	Use code-behind
`@using`	Specifies a namespace to bring into scope	`@using MyComponentNamespace`	Add namespace in *web.config*

Razor components also make extensive use of *directive attributes* on elements to control various aspects of how components get compiled (event handling, data binding, component & element references, and so on). Directive attributes all follow a common generic syntax where the values in parenthesis are optional:

```
@directive(-suffix(:name))(="value")
```

The following table summarizes the various attributes for Razor directives used in Blazor.

Attribute	Description	Example
`@attributes`	Renders a dictionary of attributes	`<input @attributes="ExtraAttributes" />`
`@bind`	Creates a two-way data binding	`<input @bind="username" @bind:event="oninput" />`
`@on{event}`	Adds an event handler for the specified event	`<button @onclick="IncrementCount">Click me!</button>`
`@key`	Specifies a key to be used by the diffing algorithm for preserving elements in a collection	`<DetailsEditor @key="person" Details="person.Details" />`
`@ref`	Captures a reference to the component or HTML element	`<MyDialog @ref="myDialog" />`

The various directive attributes used by Blazor (`@onclick`, `@bind`, `@ref`, and so on) are covered in the sections below and later chapters.

Many of the syntaxes used in *.aspx* and *.ascx* files have parallel syntaxes in Razor. Below is a simple comparison of the syntaxes for ASP.NET Web Forms and Razor.

Feature	Web Forms	Syntax	Razor	Syntax
Directives	`<%@ [directive] %>`	`<%@ Page %>`	`@[directive]`	`@page`
Code blocks	`<% %>`	`<% int x = 123; %>`	`@{ }`	`@{ int x = 123; }`
Expressions (HTML-encoded)	`<%: %>`	`<%:DateTime.Now %>`	Implicit: `@` Explicit: `@()`	`@DateTime.Now` `@(DateTime.Now)`
Comments	`<%-- --%>`	`<%-- Commented --%>`	`@* *@`	`@* Commented *@`
Data binding	`<%# %>`	`<%# Bind("Name") %>`	`@bind`	`<input @bind="username" />`

To add members to the Razor component class, use the `@code` directive. This technique is similar to using a `<script runat="server">...</script>` block in an ASP.NET Web Forms user control or page.

```
@code {
    int count = 0;

    void IncrementCount()
    {
        count++;
    }
}
```

Because Razor is based on C#, it must be compiled from within a C# project (*.csproj*). You can't compile *.razor* files from a Visual Basic project (*.vbproj*). You can still reference Visual Basic projects from your Blazor project. The opposite is true too.

For a full Razor syntax reference, see Razor syntax reference for ASP.NET Core.

Use components

Aside from normal HTML, components can also use other components as part of their rendering logic. The syntax for using a component in Razor is similar to using a user control in an ASP.NET Web Forms app. Components are specified using an element tag that matches the type name of the component. For example, you can add a `Counter` component like this:

```
<Counter />
```

Unlike ASP.NET Web Forms, components in Blazor:

- Don't use an element prefix (for example, `asp:`).
- Don't require registration on the page or in the *web.config*.

Think of Razor components like you would .NET types, because that's exactly what they are. If the assembly containing the component is referenced, then the component is available for use. To bring the component's namespace into scope, apply the `@using` directive:

```
@using MyComponentLib

<Counter />
```

As seen in the default Blazor projects, it's common to put `@using` directives into a *_Imports.razor* file so that they're imported into all *.razor* files in the same directory and in child directories.

If the namespace for a component isn't in scope, you can specify a component using its full type name, as you can in C#:

```
<MyComponentLib.Counter />
```

Modify page title from components

When building SPA-style apps, it's common for parts of a page to reload without reloading the entire page. Even so, it can be useful to have the title of the page change based on which component is currently loaded. This can be accomplished by including the `<PageTitle>` tag in the component's Razor page:

```
@page "/"
<PageTitle>Home</PageTitle>
```

The contents of this element can be dynamic, for instance showing the current count of messages:

```
<PageTitle>@MessageCount messages</PageTitle>
```

Note that if several components on a particular page include `<PageTitle>` tags, only the last one will be displayed (since each one will overwrite the previous one).

Component parameters

In ASP.NET Web Forms, you can flow parameters and data to controls using public properties. These properties can be set in markup using attributes or set directly in code. Razor components work in a similar fashion, although the component properties must also be marked with the `[Parameter]` attribute to be considered component parameters.

The following `Counter` component defines a component parameter called `IncrementAmount` that can be used to specify the amount that the `Counter` should be incremented each time the button is clicked.

```
<h1>Counter</h1>

<p>Current count: @currentCount</p>

<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>

@code {
```

```
    int currentCount = 0;

    [Parameter]
    public int IncrementAmount { get; set; } = 1;

    void IncrementCount()
    {
        currentCount+=IncrementAmount;
    }
}
```

To specify a component parameter in Blazor, use an attribute as you would in ASP.NET Web Forms:

```
<Counter IncrementAmount="10" />
```

Query string parameters

Razor components can also leverage values from the query string of the page they're rendered on as a parameter source. To enable this, add the `[SupplyParameterFromQuery]` attribute to the parameter. For example, the following parameter definition would get its value from the request in the form `?IncBy=2`:

```
[Parameter]
[SupplyParameterFromQuery(Name = "IncBy")]
public int IncrementAmount { get; set; } = 1;
```

If you don't supply a custom `Name` in the `[SupplyParameterFromQuery]` attribute, by default it will match the property name (`IncrementAmount` in this case).

Components and error boundaries

By default, Blazor apps will detect unhandled exceptions and show an error message at the bottom of the page with no additional detail. To constrain the parts of the app that are impacted by an unhandled error, for instance to limit the impact to a single component, the `<ErrorBoundary>` tag can be wrapped around component declarations.

For example, to protect against possible exceptions thrown from the `Counter` component, declare it within an `<ErrorBoundary>` and optionally specify a message to display if there is an exception:

```
<ErrorBoundary>
    <ChildContent>
        <Counter />
    </ChildContent>
    <ErrorContent>
        Oops! The counter isn't working right now; please try again later.
    </ErrorContent>
</ErrorBoundary>
```

If you don't need to specify custom error content, you can just wrap the component directly:

```
<ErrorBoundary>
  <Counter />
</ErrorBoundary>
```

A default message stating "An error as occurred." will be displayed if an unhandled exception occurs in the wrapped component.

Event handlers

Both ASP.NET Web Forms and Blazor provide an event-based programming model for handling UI events. Examples of such events include button clicks and text input. In ASP.NET Web Forms, you use HTML server controls to handle UI events exposed by the DOM, or you can handle events exposed by web server controls. The events are surfaced on the server through form post-back requests. Consider the following Web Forms button click example:

Counter.ascx

```
<asp:Button ID="ClickMeButton" runat="server" Text="Click me!"
OnClick="ClickMeButton_Click" />
```

Counter.ascx.cs

```
public partial class Counter : System.Web.UI.UserControl
{
    protected void ClickMeButton_Click(object sender, EventArgs e)
    {
        Console.WriteLine("The button was clicked!");
    }
}
```

In Blazor, you can register handlers for DOM UI events directly using directive attributes of the form `@on{event}`. The `{event}` placeholder represents the name of the event. For example, you can listen for button clicks like this:

```
<button @onclick="OnClick">Click me!</button>

@code {
    void OnClick()
    {
        Console.WriteLine("The button was clicked!");
    }
}
```

Event handlers can accept an optional, event-specific argument to provide more information about the event. For example, mouse events can take a `MouseEventArgs` argument, but it isn't required.

```
<button @onclick="OnClick">Click me!</button>

@code {
    void OnClick(MouseEventArgs e)
    {
        Console.WriteLine($"Mouse clicked at {e.ScreenX}, {e.ScreenY}.");
    }
}
```

Instead of referring to a method group for an event handler, you can use a lambda expression. A lambda expression allows you to close over other in-scope values.

```
@foreach (var buttonLabel in buttonLabels)
{
    <button @onclick="() => Console.WriteLine($"The {buttonLabel} button was
clicked!")">@buttonLabel</button>
}
```

Event handlers can execute synchronously or asynchronously. For example, the following `OnClick` event handler executes asynchronously:

```
<button @onclick="OnClick">Click me!</button>

@code {
    async Task OnClick()
    {
        var result = await Http.GetAsync("api/values");
    }
}
```

After an event is handled, the component is rendered to account for any component state changes. With asynchronous event handlers, the component is rendered immediately after the handler execution completes. The component is rendered *again* after the asynchronous `Task` completes. This asynchronous execution mode provides an opportunity to render some appropriate UI while the asynchronous `Task` is still in progress.

```
<button @onclick="ShowMessage">Get message</button>

@if (showMessage)
{
    @if (message == null)
    {
        <p><em>Loading...</em></p>
    }
    else
    {
        <p>The message is: @message</p>
    }
}

@code
{
    bool showMessage = false;
    string message;

    public async Task ShowMessage()
    {
        showMessage = true;
        message = await MessageService.GetMessageAsync();
    }
}
```

Components can also define their own events by defining a component parameter of type `EventCallback<TValue>`. Event callbacks support all the variations of DOM UI event handlers: optional arguments, synchronous or asynchronous, method groups, or lambda expressions.

```
<button class="btn btn-primary" @onclick="OnClick">Click me!</button>

@code {
```

```
    [Parameter]
    public EventCallback<MouseEventArgs> OnClick { get; set; }
}
```

Data binding

Blazor provides a simple mechanism to bind data from a UI component to the component's state. This approach differs from the features in ASP.NET Web Forms for binding data from data sources to UI controls. We'll cover handling data from different data sources in the Dealing with data section.

To create a two-way data binding from a UI component to the component's state, use the `@bind` directive attribute. In the following example, the value of the check box is bound to the `isChecked` field.

```
<input type="checkbox" @bind="isChecked" />

@code {
    bool isChecked;
}
```

When the component is rendered, the value of the checkbox is set to the value of the `isChecked` field. When the user toggles the checkbox, the `onchange` event is fired and the `isChecked` field is set to the new value. The `@bind` syntax in this case is equivalent to the following markup:

```
<input value="@isChecked" @onchange="(UIChangeEventArgs e) => isChecked = e.Value" />
```

To change the event used for the bind, use the `@bind:event` attribute.

```
<input @bind="text" @bind:event="oninput" />
<p>@text</p>

@code {
    string text;
}
```

Components can also support data binding to their parameters. To data bind, define an event callback parameter with the same name as the bindable parameter. The "Changed" suffix is added to the name.

PasswordBox.razor

```
Password: <input
    value="@Password"
    @oninput="OnPasswordChanged"
    type="@(showPassword ? "text" : "password")" />

<label><input type="checkbox" @bind="showPassword" />Show password</label>

@code {
    private bool showPassword;

    [Parameter]
    public string Password { get; set; }

    [Parameter]
```

```
    public EventCallback<string> PasswordChanged { get; set; }

    private Task OnPasswordChanged(ChangeEventArgs e)
    {
        Password = e.Value.ToString();
        return PasswordChanged.InvokeAsync(Password);
    }
}
```

To chain a data binding to an underlying UI element, set the value and handle the event directly on the UI element instead of using the `@bind` attribute.

To bind to a component parameter, use a `@bind-{Parameter}` attribute to specify the parameter to which you want to bind.

```
<PasswordBox @bind-Password="password" />

@code {
    string password;
}
```

State changes

If the component's state has changed outside of a normal UI event or event callback, then the component must manually signal that it needs to be rendered again. To signal that a component's state has changed, call the `StateHasChanged` method on the component.

In the example below, a component displays a message from an `AppState` service that can be updated by other parts of the app. The component registers its `StateHasChanged` method with the `AppState.OnChange` event so that the component is rendered whenever the message gets updated.

```
public class AppState
{
    public string Message { get; }

    // Lets components receive change notifications
    public event Action OnChange;

    public void UpdateMessage(string message)
    {
        Message = message;
        NotifyStateChanged();
    }

    private void NotifyStateChanged() => OnChange?.Invoke();
}
```

```
@inject AppState AppState

<p>App message: @AppState.Message</p>

@code {
    protected override void OnInitialized()
    {
```

```
        AppState.OnChange += StateHasChanged
    }
}
```

Component lifecycle

The ASP.NET Web Forms framework has well-defined lifecycle methods for modules, pages, and controls. For example, the following control implements event handlers for the `Init`, `Load`, and `UnLoad` lifecycle events:

Counter.ascx.cs

```
public partial class Counter : System.Web.UI.UserControl
{
    protected void Page_Init(object sender, EventArgs e) { ... }
    protected void Page_Load(object sender, EventArgs e) { ... }
    protected void Page_UnLoad(object sender, EventArgs e) { ... }
}
```

Razor components also have a well-defined lifecycle. A component's lifecycle can be used to initialize component state and implement advanced component behaviors.

All of Blazor's component lifecycle methods have both synchronous and asynchronous versions. Component rendering is synchronous. You can't run asynchronous logic as part of the component rendering. All asynchronous logic must execute as part of an `async` lifecycle method.

OnInitialized

The `OnInitialized` and `OnInitializedAsync` methods are used to initialize the component. A component is typically initialized after it's first rendered. After a component is initialized, it may be rendered multiple times before it's eventually disposed. The `OnInitialized` method is similar to the `Page_Load` event in ASP.NET Web Forms pages and controls.

```
protected override void OnInitialized() { ... }
protected override async Task OnInitializedAsync() { await ... }
```

OnParametersSet

The `OnParametersSet` and `OnParametersSetAsync` methods are called when a component has received parameters from its parent and the value are assigned to properties. These methods are executed after component initialization and *each time the component is rendered*.

```
protected override void OnParametersSet() { ... }
protected override async Task OnParametersSetAsync() { await ... }
```

OnAfterRender

The `OnAfterRender` and `OnAfterRenderAsync` methods are called after a component has finished rendering. Element and component references are populated at this point (more on these concepts below). Interactivity with the browser is enabled at this point. Interactions with the DOM and JavaScript execution can safely take place.

```
protected override void OnAfterRender(bool firstRender)
{
    if (firstRender)
    {
        ...
    }
}
protected override async Task OnAfterRenderAsync(bool firstRender)
{
    if (firstRender)
    {
        await ...
    }
}
```

`OnAfterRender` and `OnAfterRenderAsync` *aren't called when prerendering on the server*.

The `firstRender` parameter is `true` the first time the component is rendered; otherwise, its value is `false`.

IDisposable

Razor components can implement `IDisposable` to dispose of resources when the component is removed from the UI. A Razor component can implement `IDispose` by using the `@implements` directive:

```
@using System
@implements IDisposable

...

@code {
    public void Dispose()
    {
        ...
    }
}
```

Capture component references

In ASP.NET Web Forms, it's common to manipulate a control instance directly in code by referring to its ID. In Blazor, it's also possible to capture and manipulate a reference to a component, although it's much less common.

To capture a component reference in Blazor, use the `@ref` directive attribute. The value of the attribute should match the name of a settable field with the same type as the referenced component.

```
<MyLoginDialog @ref="loginDialog" ... />

@code {
    MyLoginDialog loginDialog = default!;

    void OnSomething()
    {
        loginDialog.Show();
```

```
    }
}
```

When the parent component is rendered, the field is populated with the child component instance. You can then call methods on, or otherwise manipulate, the component instance.

Manipulating component state directly using component references isn't recommended. Doing so prevents the component from being rendered automatically at the correct times.

Capture element references

Razor components can capture references to an element. Unlike HTML server controls in ASP.NET Web Forms, you can't manipulate the DOM directly using an element reference in Blazor. Blazor handles most DOM interactions for you using its DOM diffing algorithm. Captured element references in Blazor are opaque. However, they're used to pass a specific element reference in a JavaScript interop call. For more information about JavaScript interop, see ASP.NET Core Blazor JavaScript interop.

Templated components

In ASP.NET Web Forms, you can create *templated controls*. Templated controls enable the developer to specify a portion of the HTML used to render a container control. The mechanics of building templated server controls are complex, but they enable powerful scenarios for rendering data in a user customizable way. Examples of templated controls include `Repeater` and `DataList`.

Razor components can also be templated by defining component parameters of type `RenderFragment` or `RenderFragment<T>`. A `RenderFragment` represents a chunk of Razor markup that can then be rendered by the component. A `RenderFragment<T>` is a chunk of Razor markup that takes a parameter that can be specified when the render fragment is rendered.

Child content

Razor components can capture their child content as a `RenderFragment` and render that content as part of the component rendering. To capture child content, define a component parameter of type `RenderFragment` and name it `ChildContent`.

ChildContentComponent.razor

```
<h1>Component with child content</h1>

<div>@ChildContent</div>

@code {
    [Parameter]
    public RenderFragment ChildContent { get; set; }
}
```

A parent component can then supply child content using normal Razor syntax.

```
<ChildContentComponent>
    <ChildContent>
        <p>The time is @DateTime.Now</p>
    </ChildContent>
</ChildContentComponent>
```

Template parameters

A templated Razor component can also define multiple component parameters of type `RenderFragment` or `RenderFragment<T>`. The parameter for a `RenderFragment<T>` can be specified when it's invoked. To specify a generic type parameter for a component, use the `@typeparam` Razor directive.

SimpleListView.razor

```
@typeparam TItem

@Heading

<ul>
@foreach (var item in Items)
{
    <li>@ItemTemplate(item)</li>
}
</ul>

@code {
    [Parameter]
    public RenderFragment Heading { get; set; }

    [Parameter]
    public RenderFragment<TItem> ItemTemplate { get; set; }

    [Parameter]
    public IEnumerable<TItem> Items { get; set; }
}
```

When using a templated component, the template parameters can be specified using child elements that match the names of the parameters. Component arguments of type `RenderFragment<T>` passed as elements have an implicit parameter named `context`. You can change the name of this implement parameter using the `Context` attribute on the child element. Any generic type parameters can be specified using an attribute that matches the name of the type parameter. The type parameter will be inferred if possible:

```
<SimpleListView Items="messages" TItem="string">
    <Heading>
        <h1>My list</h1>
    </Heading>
    <ItemTemplate Context="message">
        <p>The message is: @message</p>
    </ItemTemplate>
</SimpleListView>
```

The output of this component looks like this:

```
<h1>My list</h1>
<ul>
    <li><p>The message is: message1</p></li>
    <li><p>The message is: message2</p></li>
<ul>
```

Code-behind

A Razor component is typically authored in a single *.razor* file. However, it's also possible to separate the code and markup using a code-behind file. To use a component file, add a C# file that matches the file name of the component file but with a *.cs* extension added (*Counter.razor.cs*). Use the C# file to define a base class for the component. You can name the base class anything you'd like, but it's common to name the class the same as the component class, but with a `Base` extension added (`CounterBase`). The component-based class must also derive from `ComponentBase`. Then, in the Razor component file, add the `@inherits` directive to specify the base class for the component (`@inherits CounterBase`).

Counter.razor

```
@inherits CounterBase

<h1>Counter</h1>

<p>Current count: @currentCount</p>

<button @onclick="IncrementCount">Click me</button>
```

Counter.razor.cs

```
public class CounterBase : ComponentBase
{
    protected int currentCount = 0;

    protected void IncrementCount()
    {
        currentCount++;
    }
}
```

The visibility of the component's members in the base class must be `protected` or `public` to be visible to the component class.

Additional resources

The preceding isn't an exhaustive treatment of all aspects of Razor components. For more information on how to Create and use ASP.NET Core Razor components, see the Blazor documentation.

CHAPTER 7

Pages, routing, and layouts

ASP.NET Web Forms apps are composed of pages defined in *.aspx* files. Each page's address is based on its physical file path in the project. When a browser makes a request to the page, the contents of the page are dynamically rendered on the server. The rendering accounts for both the page's HTML markup and its server controls.

In Blazor, each page in the app is a component, typically defined in a *.razor* file, with one or more specified routes. Routing mostly happens client-side without involving a specific server request. The browser first makes a request to the root address of the app. A root `Router` component in the Blazor app then handles intercepting navigation requests and forwards them to the correct component.

Blazor also supports *deep linking*. Deep linking occurs when the browser makes a request to a specific route other than the root of the app. Requests for deep links sent to the server are routed to the Blazor app, which then routes the request client-side to the correct component.

A simple page in ASP.NET Web Forms might contain the following markup:

Name.aspx

```
<%@ Page Title="Name" Language="C#" MasterPageFile="~/Site.Master" AutoEventWireup="true"
CodeBehind="Name.aspx.cs" Inherits="WebApplication1.Name" %>

<asp:Content ID="BodyContent" ContentPlaceHolderID="MainContent" runat="server">
    <div>
        What is your name?<br />
        <asp:TextBox ID="TextBox1" runat="server"></asp:TextBox>
        <asp:Button ID="Button1" runat="server" Text="Submit" OnClick="Button1_Click" />
    </div>
    <div>
        <asp:Literal ID="Literal1" runat="server" />
    </div>
</asp:Content>
```

Name.aspx.cs

```
public partial class Name : System.Web.UI.Page
{
    protected void Button1_Click1(object sender, EventArgs e)
    {
        Literal1.Text = "Hello " + TextBox1.Text;
    }
}
```

The equivalent page in a Blazor app would look like this:

Name.razor

```
@page "/Name"
@layout MainLayout

<div>
    What is your name?<br />
    <input @bind="text" />
    <button @onclick="OnClick">Submit</button>
</div>
<div>
    @if (name != null)
    {
        @:Hello @name
    }
</div>

@code {
    string text;
    string name;

    void OnClick() {
        name = text;
    }
}
```

Create pages

To create a page in Blazor, create a component and add the `@page` Razor directive to specify the route for the component. The `@page` directive takes a single parameter, which is the route template to add to that component.

```
@page "/counter"
```

The route template parameter is required. Unlike ASP.NET Web Forms, the route to a Blazor component *isn't* inferred from its file location (although that may be a feature added in the future).

The route template syntax is the same basic syntax used for routing in ASP.NET Web Forms. Route parameters are specified in the template using braces. Blazor will bind route values to component parameters with the same name (case-insensitive).

```
@page "/product/{id}"

<h1>Product @Id</h1>

@code {
    [Parameter]
    public string Id { get; set; }
}
```

You can also specify constraints on the value of the route parameter. For example, to constrain the product ID to be an `int`:

```
@page "/product/{id:int}"

<h1>Product @Id</h1>

@code {
    [Parameter]
    public int Id { get; set; }
}
```

For a full list of the route constraints supported by Blazor, see Route constraints.

Router component

Routing in Blazor is handled by the `Router` component. The `Router` component is typically used in the app's root component (*App.razor*).

```
<Router AppAssembly="@typeof(Program).Assembly">
    <Found Context="routeData">
        <RouteView RouteData="@routeData" DefaultLayout="@typeof(MainLayout)" />
    </Found>
    <NotFound>
        <LayoutView Layout="@typeof(MainLayout)">
            <p>Sorry, there's nothing at this address.</p>
        </LayoutView>
    </NotFound>
</Router>
```

The `Router` component discovers the routable components in the specified `AppAssembly` and in the optionally specified `AdditionalAssemblies`. When the browser navigates, the `Router` intercepts the navigation and renders the contents of its `Found` parameter with the extracted `RouteData` if a route matches the address, otherwise the `Router` renders its `NotFound` parameter.

The `RouteView` component handles rendering the matched component specified by the `RouteData` with its layout if it has one. If the matched component doesn't have a layout, then the optionally specified `DefaultLayout` is used.

The `LayoutView` component renders its child content within the specified layout. We'll look at layouts more in detail later in this chapter.

Navigation

In ASP.NET Web Forms, you trigger navigation to a different page by returning a redirect response to the browser. For example:

```
protected void NavigateButton_Click(object sender, EventArgs e)
{
    Response.Redirect("Counter");
}
```

Returning a redirect response isn't typically possible in Blazor. Blazor doesn't use a request-reply model. You can, however, trigger browser navigations directly, as you can with JavaScript.

Blazor provides a `NavigationManager` service that can be used to:

- Get the current browser address
- Get the base address
- Trigger navigations
- Get notified when the address changes

To navigate to a different address, use the `NavigateTo` method:

```
@page "/"
@inject NavigationManager NavigationManager

<button @onclick="Navigate">Navigate</button>

@code {
    void Navigate() {
        NavigationManager.NavigateTo("counter");
    }
}
```

For a description of all `NavigationManager` members, see URI and navigation state helpers.

Base URLs

If your Blazor app is deployed under a base path, then you need to specify the base URL in the page metadata using the `<base>` tag for routing to work property. If the host page for the app is server-rendered using Razor, then you can use the `~/` syntax to specify the app's base address. If the host page is static HTML, then you need to specify the base URL explicitly.

```
<base href="~/" />
```

Page layout

Page layout in ASP.NET Web Forms is handled by Master Pages. Master Pages define a template with one or more content placeholders that can then be supplied by individual pages. Master Pages are defined in *.master* files and start with the `<%@ Master %>` directive. The content of the *.master* files is coded as you would an *.aspx* page, but with the addition of `<asp:ContentPlaceHolder>` controls to mark where pages can supply content.

Site.master

```
<%@ Master Language="C#" AutoEventWireup="true" CodeBehind="Site.master.cs"
Inherits="WebApplication1.SiteMaster" %>

<!DOCTYPE html>
<html lang="en">
<head runat="server">
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title><%: Page.Title %> - My ASP.NET Application</title>
    <link href="~/favicon.ico" rel="shortcut icon" type="image/x-icon" />
</head>
```

```
<body>
    <form runat="server">
        <div class="container body-content">
            <asp:ContentPlaceHolder ID="MainContent" runat="server">
            </asp:ContentPlaceHolder>
            <hr />
            <footer>
                <p>&copy; <%: DateTime.Now.Year %> - My ASP.NET Application</p>
            </footer>
        </div>
    </form>
</body>
</html>
```

In Blazor, you handle page layout using layout components. Layout components inherit from `LayoutComponentBase`, which defines a single `Body` property of type `RenderFragment`, which can be used to render the contents of the page.

MainLayout.razor

```
@inherits LayoutComponentBase
<h1>Main layout</h1>
<div>
    @Body
</div>
```

When the page with a layout is rendered, the page is rendered within the contents of the specified layout at the location where the layout renders its `Body` property.

To apply a layout to a page, use the `@layout` directive:

```
@layout MainLayout
```

You can specify the layout for all components in a folder and subfolders using an *_Imports.razor* file. You can also specify a default layout for all your pages using the Router component.

Master Pages can define multiple content placeholders, but layouts in Blazor only have a single `Body` property. This limitation of Blazor layout components will hopefully be addressed in a future release.

Master Pages in ASP.NET Web Forms can be nested. That is, a Master Page may also use a Master Page. Layout components in Blazor may be nested too. You can apply a layout component to a layout component. The contents of the inner layout will be rendered within the outer layout.

ChildLayout.razor

```
@layout MainLayout
<h2>Child layout</h2>
<div>
    @Body
</div>
```

Index.razor

```
@page "/"
@layout ChildLayout
<p>I'm in a nested layout!</p>
```

The rendered output for the page would then be:

```
<h1>Main layout</h1>
<div>
    <h2>Child layout</h2>
    <div>
        <p>I'm in a nested layout!</p>
    </div>
</div>
```

Layouts in Blazor don't typically define the root HTML elements for a page (`<html>`, `<body>`, `<head>`, and so on). The root HTML elements are instead defined in a Blazor app's host page, which is used to render the initial HTML content for the app (see Bootstrap Blazor). The host page can render multiple root components for the app with surrounding markup.

Components in Blazor, including pages, can't render `<script>` tags. This rendering restriction exists because `<script>` tags get loaded once and then can't be changed. Unexpected behavior may occur if you try to render the tags dynamically using Razor syntax. Instead, all `<script>` tags should be added to the app's host page.

CHAPTER 8

State management

State management is a key concept of Web Forms applications, facilitated through ViewState, Session State, Application State, and Postback features. These stateful features of the framework helped to hide the state management required for an application and allow application developers to focus on delivering their functionality. With ASP.NET Core and Blazor, some of these features have been relocated and some have been removed altogether. This chapter reviews how to maintain state and deliver the same functionality with the new features in Blazor.

Request state management with ViewState

When discussing state management in Web Forms application, many developers will immediately think of ViewState. In Web Forms, ViewState manages the state of the content between HTTP requests by sending a large encoded block of text back and forth to the browser. The ViewState field could be overwhelmed with content from a page containing many elements, potentially expanding to several megabytes in size.

With Blazor Server, the app maintains an ongoing connection with the server. The app's state, called a *circuit*, is held in server memory while the connection is considered active. State will only be disposed when the user navigates away from the app or a particular page in the app. All members of the active components are available between interactions with the server.

There are several advantages of this feature:

- Component state is readily available and not rebuilt between interactions.
- State isn't transmitted to the browser.

However, there are some disadvantages to in-memory component state persistence to be aware of:

- If the server restarts between request, state is lost.
- Your application web server load-balancing solution must include sticky sessions to ensure that all requests from the same browser return to the same server. If a request goes to a different server, state will be lost.
- Persistence of component state on the server can lead to memory pressure on the web server.

For the preceding reasons, don't rely on just the state of the component to reside in-memory on the server. Your application should also include some backing data store for data between requests. Some simple examples of this strategy:

- In a shopping cart application, persist the content of new items added to the cart in a database record. If the state on the server is lost, you can reconstitute it from the database records.

- In a multi-part web form, your users will expect your application to remember values between each request. Write the data between each of your user's posts to a data store so that they can be fetched and assembled into the final form response structure when the multi-part form is completed.

For additional details on managing state in Blazor apps, see ASP.NET Core Blazor state management.

Maintain state with Session

Web Forms developers could maintain information about the currently acting user with the Microsoft.AspNetCore.Http.ISession dictionary object. It's easy enough to add an object with a string key to the `Session`, and that object would be available at a later time during the user's interactions with the application. In an attempt to eliminate managing interacting with HTTP, the `Session` object made it easy to maintain state.

The signature of the .NET Framework `Session` object isn't the same as the ASP.NET Core `Session` object. Consider the documentation for the new ASP.NET Core Session before deciding to migrate and use the new session state feature.

Session is available in ASP.NET Core and Blazor Server, but is discouraged from use in favor of storing data in a data repository appropriately. Session state is also not functional if visitors decline the use HTTP cookies in your application due to privacy concerns.

Configuration for ASP.NET Core and Session state is available in the Session and state management in ASP.NET Core article.

Application state

The `Application` object in the Web Forms framework provides a massive, cross-request repository for interacting with application-scope configuration and state. Application state was an ideal place to store various application configuration properties that would be referenced by all requests, regardless of the user making the request. The problem with the `Application` object was that data didn't persist across multiple servers. The state of the application object was lost between restarts.

As with `Session`, it's recommended that data move to a persistent backing store that could be accessed by multiple server instances. If there is volatile data that you would like to be able to access across requests and users, you could easily store it in a singleton service that can be injected into components that require this information or interaction.

The construction of an object to maintain application state and its consumption could resemble the following implementation:

```
public class MyApplicationState
{
    public int VisitorCounter { get; private set; } = 0;

    public void IncrementCounter() => VisitorCounter += 1;
}
```

```
app.AddSingleton<MyApplicationState>();
```

```
@inject MyApplicationState AppState

<label>Total Visitors: @AppState.VisitorCounter</label>
```

The `MyApplicationState` object is created only once on the server, and the value `VisitorCounter` is fetched and output in the component's label. The `VisitorCounter` value should be persisted and retrieved from a backing data store for durability and scalability.

In the browser

Application data can also be stored client-side on the user's device so that is available later. There are two browser features that allow for persistence of data in different scopes of the user's browser:

- `localStorage` - scoped to the user's entire browser. If the page is reloaded, the browser is closed and reopened, or another tab is opened with the same URL then the same `localStorage` is provided by the browser
- `sessionStorage` - scoped to the user's current browser tab. If the tab is reloaded, the state persists. However, if the user opens another tab to your application or closes and reopens the browser the state is lost.

You can write some custom JavaScript code to interact with these features, or there are a number of NuGet packages that you can use that provide this functionality. One such package is Microsoft.AspNetCore.ProtectedBrowserStorage.

For instructions on utilizing this package to interact with `localStorage` and `sessionStorage`, see the Blazor State Management article.

CHAPTER 9

Forms and validation

The ASP.NET Web Forms framework includes a set of validation server controls that handle validating user input entered into a form (`RequiredFieldValidator`, `CompareValidator`, `RangeValidator`, and so on). The ASP.NET Web Forms framework also supports model binding and validating the model based on data annotations (`[Required]`, `[StringLength]`, `[Range]`, and so on). The validation logic can be enforced both on the server and on the client using unobtrusive JavaScript-based validation. The `ValidationSummary` server control is used to display a summary of the validation errors to the user.

Blazor supports the sharing of validation logic between both the client and the server. ASP.NET provides pre-built JavaScript implementations of many common server validations. In many cases, the developer still has to write JavaScript to fully implement their app-specific validation logic. The same model types, data annotations, and validation logic can be used on both the server and client.

Blazor provides a set of input components. The input components handle binding field data to a model and validating the user input when the form is submitted.

Input component	Rendered HTML element
`InputCheckbox`	`<input type="checkbox">`
`InputDate`	`<input type="date">`
`InputNumber`	`<input type="number">`
`InputSelect`	`<select>`
`InputText`	`<input>`
`InputTextArea`	`<textarea>`

The `EditForm` component wraps these input components and orchestrates the validation process through an `EditContext`. When creating an `EditForm`, you specify what model instance to bind to using the `Model` parameter. Validation is typically done using data annotations, and it's extensible. To enable data annotation-based validation, add the `DataAnnotationsValidator` component as a child of the `EditForm`. The `EditForm` component provides a convenient event for handling valid (`OnValidSubmit`) and invalid (`OnInvalidSubmit`) submissions. There's also a more generic `OnSubmit` event that lets you trigger and handle the validation yourself.

To display a validation error summary, use the `ValidationSummary` component. To display validation messages for a specific input field, use the `ValidationMessage` component, specifying a lambda expression for the `For` parameter that points to the appropriate model member.

The following model type defines several validation rules using data annotations:

```
using System;
using System.ComponentModel.DataAnnotations;

public class Starship
{
    [Required]
    [StringLength(16,
        ErrorMessage = "Identifier too long (16 character limit).")]
    public string Identifier { get; set; }

    public string Description { get; set; }

    [Required]
    public string Classification { get; set; }

    [Range(1, 100000,
        ErrorMessage = "Accommodation invalid (1-100000).")]
    public int MaximumAccommodation { get; set; }

    [Required]
    [Range(typeof(bool), "true", "true",
        ErrorMessage = "This form disallows unapproved ships.")]
    public bool IsValidatedDesign { get; set; }

    [Required]
    public DateTime ProductionDate { get; set; }
}
```

The following component demonstrates building a form in Blazor based on the `Starship` model type:

```
<h1>New Ship Entry Form</h1>

<EditForm Model="@starship" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <ValidationSummary />

    <p>
        <label for="identifier">Identifier: </label>
        <InputText id="identifier" @bind-Value="starship.Identifier" />
        <ValidationMessage For="() => starship.Identifier" />
    </p>
    <p>
        <label for="description">Description (optional): </label>
        <InputTextArea id="description" @bind-Value="starship.Description" />
    </p>
    <p>
        <label for="classification">Primary Classification: </label>
        <InputSelect id="classification" @bind-Value="starship.Classification">
            <option value="">Select classification ...</option>
            <option value="Exploration">Exploration</option>
            <option value="Diplomacy">Diplomacy</option>
            <option value="Defense">Defense</option>
        </InputSelect>
        <ValidationMessage For="() => starship.Classification" />
    </p>
    <p>
        <label for="accommodation">Maximum Accommodation: </label>
        <InputNumber id="accommodation" @bind-Value="starship.MaximumAccommodation" />
        <ValidationMessage For="() => starship.MaximumAccommodation" />
    </p>
```

```
    <p>
        <label for="valid">Engineering Approval: </label>
        <InputCheckbox id="valid" @bind-Value="starship.IsValidatedDesign" />
        <ValidationMessage For="() => starship.IsValidatedDesign" />
    </p>
    <p>
        <label for="productionDate">Production Date: </label>
        <InputDate id="productionDate" @bind-Value="starship.ProductionDate" />
        <ValidationMessage For="() => starship.ProductionDate" />
    </p>

    <button type="submit">Submit</button>
</EditForm>

@code {
    private Starship starship = new Starship();

    private void HandleValidSubmit()
    {
        // Save the data
    }
}
```

After the form submission, the model-bound data hasn't been saved to any data store, like a database. In a Blazor WebAssembly app, the data must be sent to the server. For example, using an HTTP POST request. In a Blazor Server app, the data is already on the server, but it must be persisted. Handling data access in Blazor apps is the subject of the Dealing with data section.

Additional resources

For more information on forms and validation in Blazor apps, see the Blazor documentation.

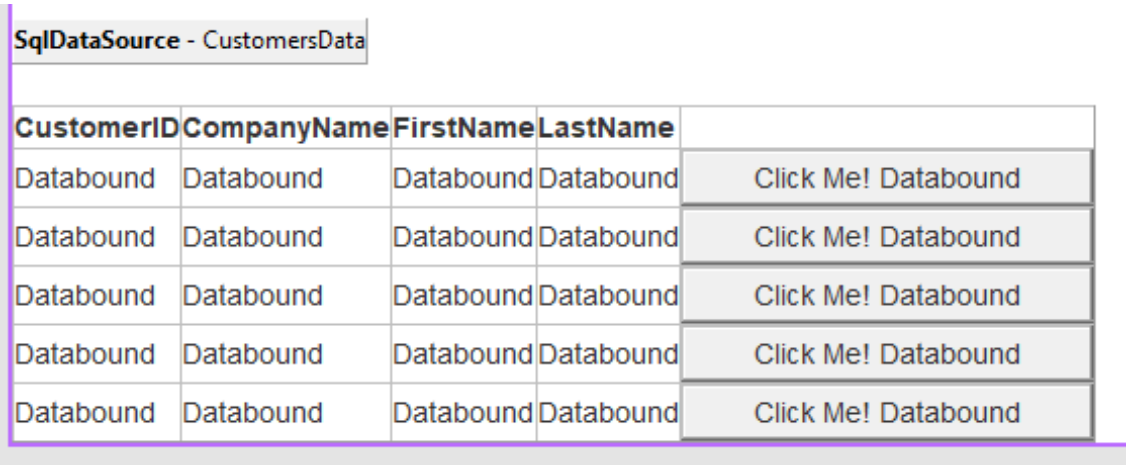
CHAPTER 10

Work with data

Data access is the backbone of an ASP.NET Web Forms app. If you're building forms for the web, what happens to that data? With Web Forms, there were multiple data access techniques you could use to interact with a database:

- Data Sources
- ADO.NET
- Entity Framework

Data Sources were controls that you could place on a Web Forms page and configure like other controls. Visual Studio provided a friendly set of dialogs to configure and bind the controls to your Web Forms pages. Developers who enjoy a "low code" or "no code" approach preferred this technique when Web Forms was first released.



ADO.NET is the low-level approach to interacting with a database. Your apps could create a connection to the database with Commands, Datatables, and Datasets for interacting. The results could then be bound to fields on screen without much code. The drawback of this approach was that each set of ADO.NET objects (`Connection`, `Command`, and `DataTable`) was bound to libraries provided by a database vendor. Use of these components made the code rigid and difficult to migrate to a different database.

Entity Framework

Entity Framework (EF) is the open source object-relational mapping framework maintained by the .NET Foundation. Initially released with .NET Framework, EF allows for generating code for the database connections, storage schemas, and interactions. With this abstraction, you can focus on your

app's business rules and allow the database to be managed by a trusted database administrator. In .NET, you can use an updated version of EF called EF Core. EF Core helps generate and maintain the interactions between your code and the database with a series of commands that are available for you using the `dotnet ef` command-line tool. Let's take a look at a few samples to get you working with a database.

EF Code First

A quick way to get started building your database interactions is to start with the class objects you want to work with. EF provides a tool to help generate the appropriate database code for your classes. This approach is called "Code First" development. Consider the following `Product` class for a sample storefront app that we want to store in a relational database like Microsoft SQL Server.

```
public class Product
{
    public int Id { get; set; }

    [Required]
    public string Name { get; set; }

    [MaxLength(4000)]
    public string Description { get; set; }

    [Range(0, 99999,99)]
    [DataType(DataType.Currency)]
    public decimal Price { get; set; }
}
```

Product has a primary key and three additional fields that would be created in our database:

- EF will identify the `Id` property as a primary key by convention.
- `Name` will be stored in a column configured for text storage. The `[Required]` attribute decorating this property will add a `not null` constraint to help enforce this declared behavior of the property.
- `Description` will be stored in a column configured for text storage, and have a maximum length configured of 4000 characters as dictated by the `[MaxLength]` attribute. The database schema will be configured with a column named `MaxLength` using data type `varchar(4000)`.
- The `Price` property will be stored as currency. The `[Range]` attribute will generate appropriate constraints to prevent data storage outside of the minimum and maximum values declared.

We need to add this `Product` class to a database context class that defines the connection and translation operations with our database.

```
public class MyDbContext : DbContext
{
    public DbSet<Product> Products { get; set; }
}
```

The `MyDbContext` class provides the one property that defines the access and translation for the `Product` class. Your application configures this class for interaction with the database using the following entries in the `Startup` class's `ConfigureServices` method (or appropriate location in *Program.cs* using the `builder.Services` property instead of `services`):

```
services.AddDbContext<MyDbContext>(options =>
    options.UseSqlServer("MY DATABASE CONNECTION STRING"));
```

The preceding code will connect to a SQL Server database with the specified connection string. You can place the connection string in your *appsettings.json* file, environment variables, or other configuration storage locations and replace this embedded string appropriately.

You can then generate the database table appropriate for this class using the following commands:

```
dotnet ef migrations add 'Create Product table'
dotnet ef database update
```

The first command defines the changes you're making to the database schema as a new EF Migration called `Create Product table`. A Migration defines how to apply and remove your new database changes.

Once applied, you have a simple `Product` table in your database and some new classes added to the project that help manage the database schema. You can find these generated classes, by default, in a new folder called *Migrations*. When you make changes to the `Product` class or add more related classes you would like interacting with your database, you need to run the command-line commands again with a new name of the migration. This command will generate another set of migration classes to update your database schema.

EF Database First

For existing databases, you can generate the classes for EF Core by using the .NET command-line tools. To scaffold the classes, use a variation of the following command:

```
dotnet ef dbcontext scaffold "CONNECTION STRING" Microsoft.EntityFrameworkCore.SqlServer -c
MyDbContext -t Product -t Customer
```

The preceding command connects to the database using the specified connection string and the `Microsoft.EntityFrameworkCore.SqlServer` provider. Once connected, a database context class named `MyDbContext` is created. Additionally, supporting classes are created for the `Product` and `Customer` tables that were specified with the `-t` options. There are many configuration options for this command to generate the class hierarchy appropriate for your database. For a complete reference, see the command's documentation.

More information about EF Core can be found on the Microsoft Docs site.

Interact with web services

When ASP.NET was first released, SOAP services were the preferred way for web servers and clients to exchange data. Much has changed since that time, and the preferred interactions with services have shifted to direct HTTP client interactions. With ASP.NET Core and Blazor, you can register the configuration of your `HttpClient` in *Program.cs* or in the `Startup` class's `ConfigureServices` method. Use that configuration when you need to interact with the HTTP endpoint. Consider the following configuration code:

```
// in Program.cs
builder.Services.AddHttpClient("github", client =>
{
    client.BaseAddress = new Uri("http://api.github.com/");
    // Github API versioning
    client.DefaultRequestHeaders.Add("Accept", "application/vnd.github.v3+json");
    // Github requires a user-agent
    client.DefaultRequestHeaders.Add("User-Agent", "BlazorWebForms-Sample");
});
```

Whenever you need to access data from GitHub, create a client with a name of `github`. The client is configured with the base address, and the request headers are set appropriately. Inject the `IHttpClientFactory` into your Blazor components with the `@inject` directive or an `[Inject]` attribute on a property. Create your named client and interact with services using the following syntax:

```
@inject IHttpClientFactory factory

...

@code {
    protected override async Task OnInitializedAsync()
    {
        var client = factory.CreateClient("github");
        var response = await client.GetAsync("repos/dotnet/docs/issues");
        response.EnsureStatusCode();
        var content = await response.Content.ReadAsStringAsync();
    }
}
```

This method returns the string describing the collection of issues in the *dotnet/docs* GitHub repository. It returns content in JSON format and is deserialized into appropriate GitHub issue objects. There are many ways that you can configure the `HttpClientFactory` to deliver preconfigured `HttpClient` objects. Try configuring multiple `HttpClient` instances with different names and endpoints for the various web services you work with. This approach will make your interactions with those services easier to work with on each page. For more information, see Make HTTP requests using IHttpClientFactory.

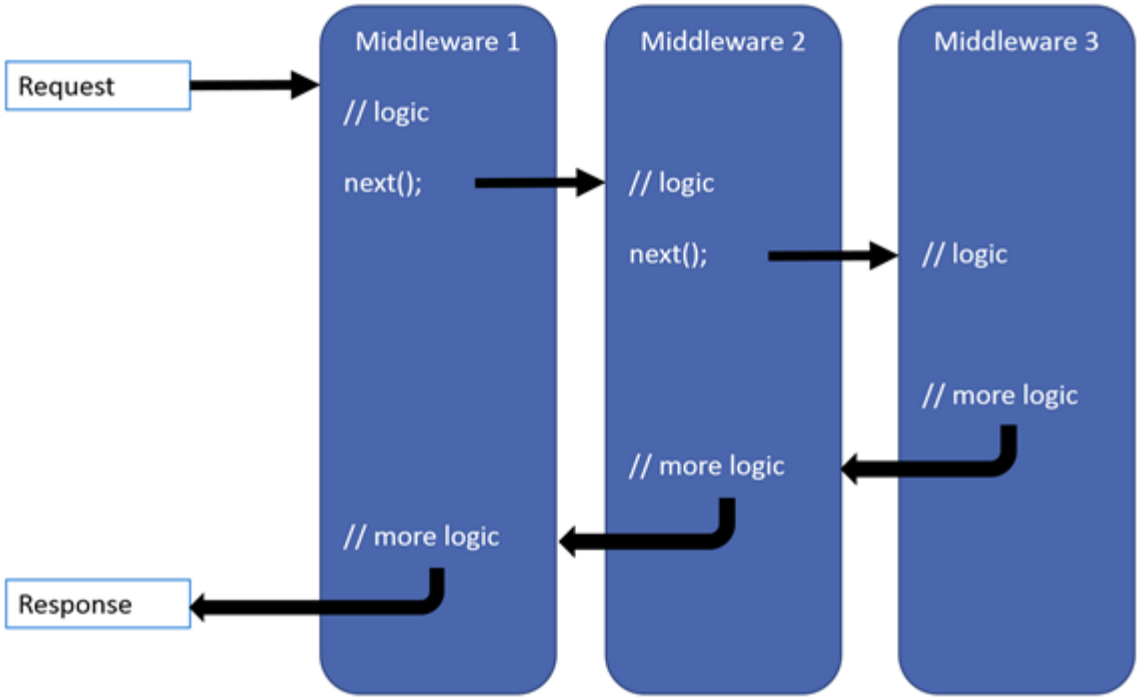
CHAPTER 11

Modules, handlers, and middleware

An ASP.NET Core app is built upon a series of *middleware*. Middleware is handlers that are arranged into a pipeline to handle requests and responses. In a Web Forms app, HTTP handlers and modules solve similar problems. In ASP.NET Core, modules, handlers, *Global.asax.cs*, and the app life cycle are replaced with middleware. In this chapter, you'll learn about middleware in the context of a Blazor app.

Overview

The ASP.NET Core request pipeline consists of a sequence of request delegates, called one after the other. The following diagram demonstrates the concept. The thread of execution follows the black arrows.



The preceding diagram lacks a concept of lifecycle events. This concept is foundational to how ASP.NET Web Forms requests are handled. This system makes it easier to reason about what process is occurring and allows middleware to be inserted at any point. Middleware executes in the order in which it's added to the request pipeline. They're also added in code instead of configuration files, usually in *Startup.cs*.

Katana

Readers familiar with Katana will feel comfortable in ASP.NET Core. In fact, Katana is a framework from which ASP.NET Core derives. It introduced similar middleware and pipeline patterns for ASP.NET 4.x. Middleware designed for Katana can be adapted to work with the ASP.NET Core pipeline.

Common middleware

ASP.NET 4.x includes many modules. In a similar fashion, ASP.NET Core has many middleware components available as well. IIS modules may be used in some cases with ASP.NET Core. In other cases, native ASP.NET Core middleware may be available.

The following table lists replacement middleware and components in ASP.NET Core.

Module	ASP.NET 4.x module	ASP.NET Core option
HTTP errors	`CustomErrorModule`	Status Code Pages Middleware
Default document	`DefaultDocumentModule`	Default Files Middleware

Module	ASP.NET 4.x module	ASP.NET Core option
Directory browsing	`DirectoryListingModule`	Directory Browsing Middleware
Dynamic compression	`DynamicCompressionModule`	Response Compression Middleware
Failed requests tracing	`FailedRequestsTracingModule`	ASP.NET Core Logging
File caching	`FileCacheModule`	Response Caching Middleware
HTTP caching	`HttpCacheModule`	Response Caching Middleware
HTTP logging	`HttpLoggingModule`	ASP.NET Core Logging
HTTP redirection	`HttpRedirectionModule`	URL Rewriting Middleware
ISAPI filters	`IsapiFilterModule`	Middleware
ISAPI	`IsapiModule`	Middleware
Request filtering	`RequestFilteringModule`	URL Rewriting Middleware IRule
URL rewriting†	`RewriteModule`	URL Rewriting Middleware
Static compression	`StaticCompressionModule`	Response Compression Middleware
Static content	`StaticFileModule`	Static File Middleware
URL authorization	`UrlAuthorizationModule`	ASP.NET Core Identity

This list isn't exhaustive but should give an idea of what mapping exists between the two frameworks. For a more detailed list, see IIS modules with ASP.NET Core.

Custom middleware

Built-in middleware may not handle all scenarios needed for an app. In such cases, it makes sense to create your own middleware. There are multiple ways of defining middleware, with the simplest being a simple delegate. Consider the following middleware, which accepts a culture request from a query string:

```
public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        app.Use(async (context, next) =>
        {
            var cultureQuery = context.Request.Query["culture"];

            if (!string.IsNullOrWhiteSpace(cultureQuery))
            {
                var culture = new CultureInfo(cultureQuery);

                CultureInfo.CurrentCulture = culture;
                CultureInfo.CurrentUICulture = culture;
            }

            // Call the next delegate/middleware in the pipeline
            await next();
        });
```

```
        app.Run(async (context) =>
            await context.Response.WriteAsync(
                $"Hello {CultureInfo.CurrentCulture.DisplayName}"));
    }
}
```

Middleware can also be defined as class, either by implementing the `IMiddleware` interface or by following middleware convention. For more information, see Write custom ASP.NET Core middleware.

CHAPTER 12

App configuration

The primary way to load app configuration in Web Forms is with entries in the *web.config* file—either on the server or a related configuration file referenced by *web.config*. You can use the static `ConfigurationManager` object to interact with app settings, data repository connection strings, and other extended configuration providers that are added into the app. It's typical to see interactions with app configuration as seen in the following code:

```
var configurationValue = ConfigurationManager.AppSettings["ConfigurationSettingName"];
var connectionString =
ConfigurationManager.ConnectionStrings["MyDatabaseConnectionName"].ConnectionString;
```

With ASP.NET Core and server-side Blazor, the *web.config* file MAY be present if your app is hosted on a Windows IIS server. However, there's no `ConfigurationManager` interaction with this configuration, and you can receive more structured app configuration from other sources. Let's take a look at how configuration is gathered and how you can still access configuration information from a *web.config* file.

Configuration sources

ASP.NET Core recognizes there are many configuration sources you may want to use for your app. The framework attempts to offer you the best of these features by default. Configuration is read and aggregated from these various sources by ASP.NET Core. Later loaded values for the same configuration key take precedence over earlier values.

ASP.NET Core was designed to be cloud-aware and to make the configuration of apps easier for both operators and developers. ASP.NET Core is environment-aware and knows if it's running in your `Production` or `Development` environment. The environment indicator is set in the `ASPNETCORE_ENVIRONMENT` system environment variable. If no value is configured, the app defaults to running in the `Production` environment.

Your app can trigger and add configuration from several sources based on the environment's name. By default, the configuration is loaded from the following resources in the order listed:

1. *appsettings.json* file, if present
2. *appsettings.{ENVIRONMENT_NAME}.json* file, if present
3. User secrets file on disk, if present
4. Environment variables
5. Command-line arguments

appsettings.json format and access

The *appsettings.json* file can be hierarchical with values structured like the following JSON:

```
{
  "section0": {
    "key0": "value",
    "key1": "value"
  },
  "section1": {
    "key0": "value",
    "key1": "value"
  }
}
```

When presented with the preceding JSON, the configuration system flattens child values and references their fully qualified hierarchical paths. A colon (`:`) character separates each property in the hierarchy. For example, the configuration key `section1:key0` accesses the `section1` object literal's `key0` value.

User secrets

User secrets are:

- Configuration values that are stored in a JSON file on the developer's workstation, outside of the app development folder.
- Only loaded when running in the `Development` environment.
- Associated with a specific app.
- Managed with the .NET CLI's `user-secrets` command.

Configure your app for secrets storage by executing the `user-secrets` command:

```
dotnet user-secrets init
```

The preceding command adds a `UserSecretsId` element to the project file. The element contains a GUID, which is used to associate secrets with the app. You can then define a secret with the `set` command. For example:

```
dotnet user-secrets set "Parent:ApiKey" "12345"
```

The preceding command makes the `Parent:ApiKey` configuration key available on a developer's workstation with the value `12345`.

For more information about creating, storing, and managing user secrets, see the Safe storage of app secrets in development in ASP.NET Core document.

Environment variables

The next set of values loaded into your app configuration is the system's environment variables. All of your system's environment variable settings are now accessible to you through the configuration API.

Hierarchical values are flattened and separated by colon characters when read inside your app. However, some operating systems don't allow the colon character environment variable names. ASP.NET Core addresses this limitation by converting values that have double-underscores (`__`) into a colon when they're accessed. The `Parent:ApiKey` value from the user secrets section above can be overridden with the environment variable `Parent__ApiKey`.

Command-line arguments

Configuration can also be provided as command-line arguments when your app is started. Use the double-dash (`--`) or forward-slash (`/`) notation to indicate the name of the configuration value to set and the value to be configured. The syntax resembles the following commands:

```
dotnet run CommandLineKey1=value1 --CommandLineKey2=value2 /CommandLineKey3=value3
dotnet run --CommandLineKey1 value1 /CommandLineKey2 value2
dotnet run Parent:ApiKey=67890
```

The return of web.config

If you've deployed your app to Windows on IIS, the *web.config* file still configures IIS to manage your app. By default, IIS adds a reference to the ASP.NET Core Module (ANCM). ANCM is a native IIS module that hosts your app in place of the Kestrel web server. This *web.config* section resembles the following XML markup:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <location path="." inheritInChildApplications="false">
    <system.webServer>
      <handlers>
        <add name="aspNetCore" path="*" verb="*" modules="AspNetCoreModuleV2"
resourceType="Unspecified" />
      </handlers>
      <aspNetCore processPath=".\MyApp.exe"
                  stdoutLogEnabled="false"
                  stdoutLogFile=".\logs\stdout"
                  hostingModel="inprocess" />
    </system.webServer>
  </location>
</configuration>
```

App-specific configuration can be defined by nesting an `environmentVariables` element in the `aspNetCore` element. The values defined in this section are presented to the ASP.NET Core app as environment variables. The environment variables load appropriately during that segment of app startup.

```
<aspNetCore processPath="dotnet"
      arguments=".\MyApp.dll"
      stdoutLogEnabled="false"
      stdoutLogFile=".\logs\stdout"
      hostingModel="inprocess">
  <environmentVariables>
    <environmentVariable name="ASPNETCORE_ENVIRONMENT" value="Development" />
```

```
    <environmentVariable name="Parent:ApiKey" value="67890" />
  </environmentVariables>
</aspNetCore>
```

Read configuration in the app

ASP.NET Core provides app configuration through the IConfiguration interface. This configuration interface should be requested by your Blazor components, Blazor pages, and any other ASP.NET Core-managed class that needs access to configuration. The ASP.NET Core framework will automatically populate this interface with the resolved configuration configured earlier. On a Blazor page or a component's Razor markup, you can inject the `IConfiguration` object with an `@inject` directive at the top of the *.razor* file like this:

```
@inject IConfiguration Configuration
```

This preceding statement makes the `IConfiguration` object available as the `Configuration` variable throughout the rest of the Razor template.

Individual configuration settings can be read by specifying the configuration setting hierarchy sought as an indexer parameter:

```
var mySetting = Configuration["section1:key0"];
```

You can fetch entire configuration sections by using the GetSection method to retrieve a collection of keys at a specific location with a syntax similar to `GetSection("section1")` to retrieve the configuration for section1 from the earlier example.

Strongly typed configuration

With Web Forms, it was possible to create a strongly typed configuration type that inherited from the ConfigurationSection type and associated types. A `ConfigurationSection` allowed you to configure some business rules and processing for those configuration values.

In ASP.NET Core, you can specify a class hierarchy that will receive the configuration values. These classes:

- Don't need to inherit from a parent class.
- Should include `public` properties that match the properties and type references for the configuration structure you wish to capture.

For the earlier *appsettings.json* sample, you could define the following classes to capture the values:

```
public class MyConfig
{
    public MyConfigSection section0 { get; set;}

    public MyConfigSection section1 { get; set;}
}

public class MyConfigSection
{
```

```
    public string key0 { get; set; }

    public string key1 { get; set; }
}
```

This class hierarchy can be populated by adding the following line to the `Startup.ConfigureServices` method (or appropriate location in *Program.cs* using the `builder.Services` property instead of `services`):

```
services.Configure<MyConfig>(Configuration);
```

In the rest of the app, you can add an input parameter to classes or an `@inject` directive in Razor templates of type `IOptions<MyConfig>` to receive the strongly typed configuration settings. The `IOptions<MyConfig>.Value` property will yield the `MyConfig` value populated from the configuration settings.

```
@inject IOptions<MyConfig> options
@code {
    var MyConfiguration = options.Value;
    var theSetting = MyConfiguration.section1.key0;
}
```

More information about the Options feature can be found in the Options pattern in ASP.NET Core document.

CHAPTER 13

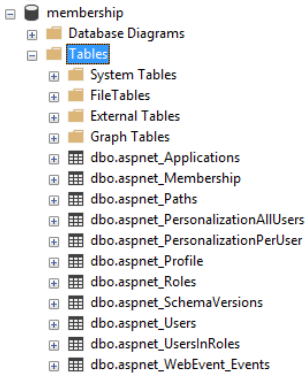
Security: Authentication and Authorization in ASP.NET Web Forms and Blazor

Migrating from an ASP.NET Web Forms application to Blazor will almost certainly require updating how authentication and authorization are performed, assuming the application had authentication configured. This chapter will cover how to migrate from the ASP.NET Web Forms universal provider model (for membership, roles, and user profiles) and how to work with ASP.NET Core Identity from Blazor apps. While this chapter will cover the high-level steps and considerations, the detailed steps and scripts may be found in the referenced documentation.

ASP.NET universal providers

Since ASP.NET 2.0, the ASP.NET Web Forms platform has supported a provider model for a variety of features, including membership. The universal membership provider, along with the optional role provider, is commonly deployed with ASP.NET Web Forms applications. It offers a robust and secure way to manage authentication and authorization that continues to work well today. The most recent offering of these universal providers is available as a NuGet package, Microsoft.AspNet.Providers.

The Universal Providers work with a SQL database schema that includes tables like `aspnet_Applications`, `aspnet_Membership`, `aspnet_Roles`, and `aspnet_Users`. When configured by running the aspnet_regsql.exe command, the providers install tables and stored procedures that provide all of the necessary queries and commands to work with the underlying data. The database schema and these stored procedures are not compatible with newer ASP.NET Identity and ASP.NET Core Identity systems, so existing data must be migrated into the new system. Figure 1 shows an example table schema configured for universal providers.



The universal provider handles users, membership, roles, and profiles. Users are assigned globally unique identifiers and basic information like userId, userName etc. are stored in the `aspnet_Users` table. Authentication information, such as password, password format, password salt, lockout counters and details, etc. are stored in the `aspnet_Membership` table. Roles consist simply of names and unique identifiers, which are assigned to users via the `aspnet_UsersInRoles` association table, providing a many-to-many relationship.

If your existing system is using roles in addition to membership, you will need to migrate the user accounts, the associated passwords, the roles, and the role membership into ASP.NET Core Identity. You will also most likely need to update your code where you're currently performing role checks using if statements to instead leverage declarative filters, attributes, and/or tag helpers. We will review migration considerations in greater detail at the end of this chapter.

Authorization configuration in Web Forms

To configure authorized access to certain pages in an ASP.NET Web Forms application, typically you specify that certain pages or folders are inaccessible to anonymous users. This configuration is done in the web.config file:

```
<?xml version="1.0"?>
<configuration>
    <system.web>
      <authentication mode="Forms">
        <forms defaultUrl="~/home.aspx" loginUrl="~/login.aspx"
          slidingExpiration="true" timeout="2880"></forms>
      </authentication>

      <authorization>
        <deny users="?" />
      </authorization>
    </system.web>
</configuration>
```

The `authentication` configuration section sets up the forms authentication for the application. The `authorization` section is used to disallow anonymous users for the entire application. However, you can provide more granular authorization rules on a per-location basis as well as apply role-based authorization checks.

```
<location path="login.aspx">
  <system.web>
```

```
    <authorization>
      <allow users="*" />
    </authorization>
  </system.web>
</location>
```

The above configuration, when combined with the first one, would allow anonymous users to access the login page, overriding the site-wide restriction on non-authenticated users.

```
<location path="/admin">
  <system.web>
    <authorization>
      <allow roles="Administrators" />
      <deny users="*" />
    </authorization>
  </system.web>
</location>
```

The above configuration, when combined with the others, restricts access to the `/admin` folder and all resources within it to members of the "Administrators" role. This restriction could also be applied by placing a separate `web.config` file within the `/admin` folder root.

Authorization code in Web Forms

In addition to configuring access using `web.config`, you can also programmatically configure access and behavior in your Web Forms application. For instance, you can restrict the ability to perform certain operations or view certain data based on the user's role.

This code can be used both in code-behind logic as well as in the page itself:

```
<% if (HttpContext.Current.User.IsInRole("Administrators")) { %>
  <a href="/admin">Go To Admin</a>
<% } %>
```

In addition to checking user role membership, you can also determine if they are authenticated (though often this is better done using the location-based configuration covered above). Below is an example of this approach.

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!User.Identity.IsAuthenticated)
    {
        FormsAuthentication.RedirectToLoginPage();
    }
    if (!Roles.IsUserInRole(User.Identity.Name, "Administrators"))
    {
        MessageLabel.Text = "Only administrators can view this.";
        SecretPanel.Visible = false;
    }
}
```

In the code above, role-based access control (RBAC) is used to determine whether certain elements of the page, such as a `SecretPanel`, are visible based on the current user's role.

Typically, ASP.NET Web Forms applications configure security within the `web.config` file and then add additional checks where needed in `.aspx` pages and their related `.aspx.cs` code-behind files. Most applications leverage the universal membership provider, frequently with the additional role provider.

ASP.NET Core Identity

Although still tasked with authentication and authorization, ASP.NET Core Identity uses a different set of abstractions and assumptions when compared to the universal providers. For example, the new Identity model supports third party authentication, allowing users to authenticate using a social media account or other trusted authentication provider. ASP.NET Core Identity supports UI for commonly needed pages like login, logout, and register. It leverages EF Core for its data access, and uses EF Core migrations to generate the necessary schema required to support its data model. This introduction to Identity on ASP.NET Core provides a good overview of what is included with ASP.NET Core Identity and how to get started working with it. If you haven't already set up ASP.NET Core Identity in your application and its database, it will help you get started.

Roles, claims, and policies

Both the universal providers and ASP.NET Core Identity support the concept of roles. You can create roles for users and assign users to roles. Users can belong to any number of roles, and you can verify role membership as part of your authorization implementation.

In addition to roles, ASP.NET Core identity supports the concepts of claims and policies. While a role should specifically correspond to a set of resources a user in that role should be able to access, a claim is simply part of a user's identity. A claim is a name value pair that represents what the subject is, not what the subject can do.

It is possible to directly inspect a user's claims and determine based on these values whether a user should be given access to a resource. However, such checks are often repetitive and scattered throughout the system. A better approach is to define a *policy*.

An authorization policy consists of one or more requirements. Policies are registered as part of the authorization service configuration in the `ConfigureServices` method of `Startup.cs`. For example, the following code snippet configures a policy called "CanadiansOnly", which has the requirement that the user has the Country claim with the value of "Canada".

```
services.AddAuthorization(options =>
{
    options.AddPolicy("CanadiansOnly", policy => policy.RequireClaim(ClaimTypes.Country,
"Canada"));
});
```

You can learn more about how to create custom policies in the documentation.

Whether you're using policies or roles, you can specify that a particular page in your Blazor application requires that role or policy with the `[Authorize]` attribute, applied with the `@attribute` directive.

Requiring a role:

```
@attribute [Authorize(Roles ="administrators")]
```

Requiring a policy be satisfied:

```
@attribute [Authorize(Policy ="CanadiansOnly")]
```

If you need access to a user's authentication state, roles, or claims in your code, there are two primary ways to achieve this functionality. The first is to receive the authentication state as a cascading parameter. The second is to access the state using an injected `AuthenticationStateProvider`. The details of each of these approaches are described in the Blazor Security documentation.

The following code shows how to receive the `AuthenticationState` as a cascading parameter:

```
[CascadingParameter]
private Task<AuthenticationState> authenticationStateTask { get; set; }
```

With this parameter in place, you can get the user using this code:

```
var authState = await authenticationStateTask;
var user = authState.User;
```

The following code shows how to inject the `AuthenticationStateProvider`:

```
@using Microsoft.AspNetCore.Components.Authorization
@inject AuthenticationStateProvider AuthenticationStateProvider
```

With the provider in place, you can gain access to the user with the following code:

```
AuthenticationState authState = await
AuthenticationStateProvider.GetAuthenticationStateAsync();
ClaimsPrincipal user = authState.User;

if (user.Identity.IsAuthenticated)
{
  // work with user.Claims and/or user.Roles
}
```

Note: The `AuthorizeView` component, covered later in this chapter, provides a declarative way to control what a user sees on a page or component.

To work with users and claims (in Blazor Server applications) you may also need to inject a `UserManager<T>` (use `IdentityUser` for default) which you can use to enumerate and modify claims for a user. First inject the type and assign it to a property:

```
@inject UserManager<IdentityUser> MyUserManager
```

Then use it to work with the user's claims. The following sample shows how to add and persist a claim on a user:

```
private async Task AddCountryClaim()
{
    var authState = await AuthenticationStateProvider.GetAuthenticationStateAsync();
    var user = authState.User;
    var identityUser = await MyUserManager.FindByNameAsync(user.Identity.Name);

    if (!user.HasClaim(c => c.Type == ClaimTypes.Country))
    {
```

```
        // stores the claim in the cookie
        ClaimsIdentity id = new ClaimsIdentity();
        id.AddClaim(new Claim(ClaimTypes.Country, "Canada"));
        user.AddIdentity(id);

        // save the claim in the database
        await MyUserManager.AddClaimAsync(identityUser, new Claim(ClaimTypes.Country,
"Canada"));
    }
}
```

If you need to work with roles, follow the same approach. You may need to inject a `RoleManager<T>` (use `IdentityRole` for default type) to list and manage the roles themselves.

Note: In Blazor WebAssembly projects, you will need to provide server APIs to perform these operations (instead of using `UserManager<T>` or `RoleManager<T>` directly). A Blazor WebAssembly client application would manage claims and/or roles by securely calling API endpoints exposed for this purpose.

Migration guide

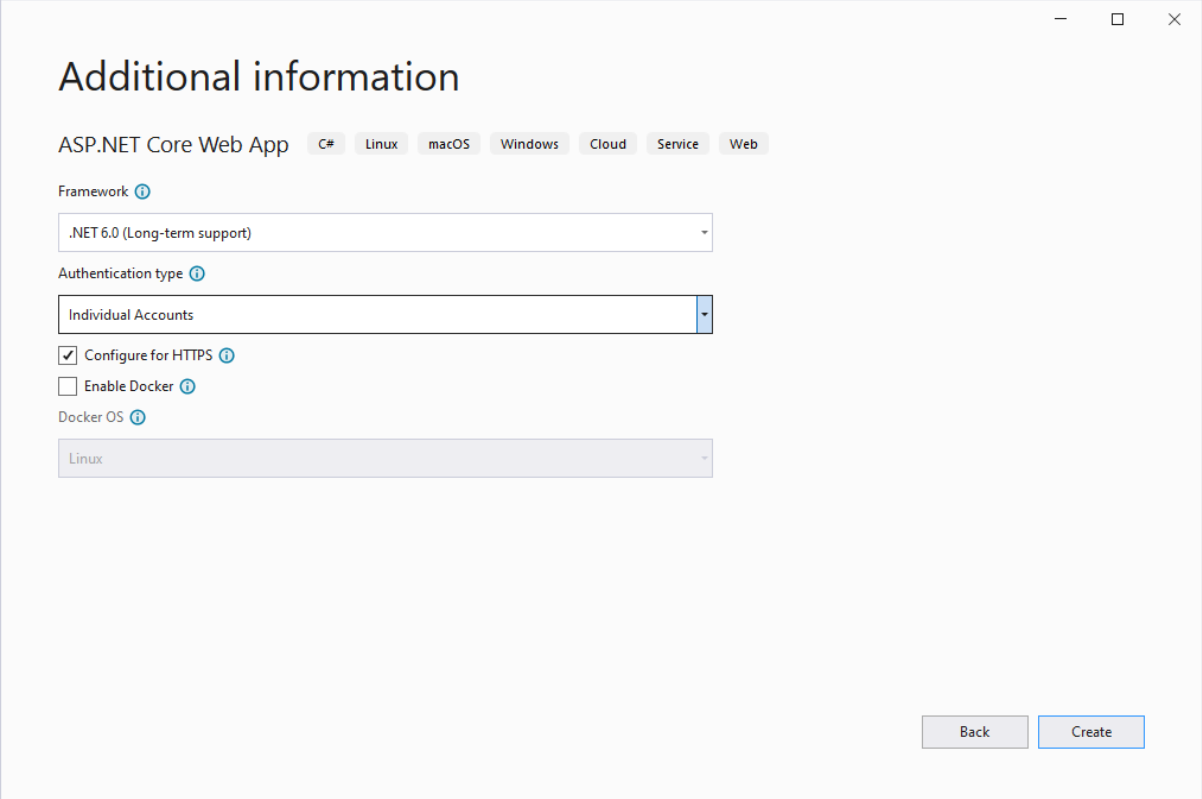
Migrating from ASP.NET Web Forms and universal providers to ASP.NET Core Identity requires several steps:

1. Create ASP.NET Core Identity database schema in the destination database
2. Migrate data from universal provider schema to ASP.NET Core Identity schema
3. Migrate configuration from the `web.config` to middleware and services, typically in *Program.cs* (or a `Startup` class)
4. Update individual pages using controls and conditionals to use tag helpers and new identity APIs.

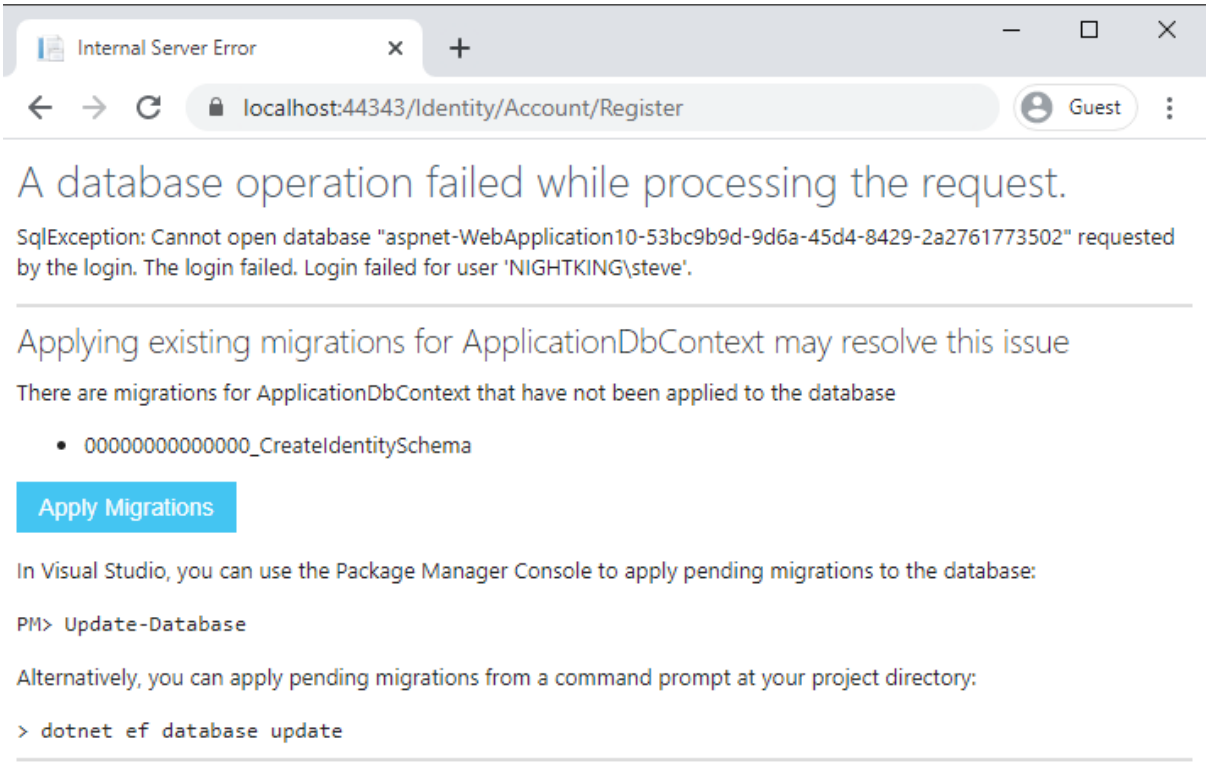
Each of these steps is described in detail in the following sections.

Creating the ASP.NET Core Identity schema

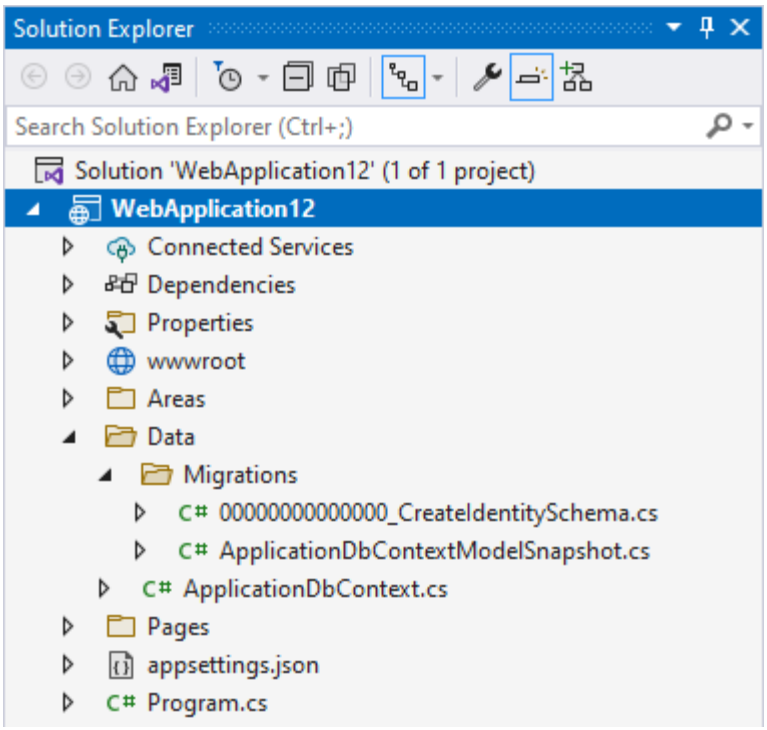
There are several ways to create the necessary table structure used for ASP.NET Core Identity. The simplest is to create a new ASP.NET Core Web application. Choose Web Application and then change Authentication type to use Individual Accounts.



From the command line, you can do the same thing by running `dotnet new webapp -au Individual`. Once the app has been created, run it and register on the site. You should trigger a page like the one shown below:



Click on the "Apply Migrations" button and the necessary database tables should be created for you. In addition, the migration files should appear in your project, as shown:



You can run the migration yourself, without running the web application, using this command-line tool:

```
dotnet ef database update
```

If you would rather run a script to apply the new schema to an existing database, you can script these migrations from the command-line. Run this command to generate the script:

```
dotnet ef migrations script -o auth.sql
```

The above command will produce a SQL script in the output file `auth.sql`, which can then be run against whatever database you like. If you have any trouble running `dotnet ef` commands, make sure you have the EF Core tools installed on your system.

In the event you have additional columns on your source tables, you will need to identify the best location for these columns in the new schema. Generally, columns found on the `aspnet_Membership` table should be mapped to the `AspNetUsers` table. Columns on `aspnet_Roles` should be mapped to `AspNetRoles`. Any additional columns on the `aspnet_UsersInRoles` table would be added to the `AspNetUserRoles` table.

It's also worth considering putting any additional columns on separate tables. So that future migrations won't need to take into account such customizations of the default identity schema.

Migrating data from universal providers to ASP.NET Core Identity

Once you have the destination table schema in place, the next step is to migrate your user and role records to the new schema. A complete list of the schema differences, including which columns map to which new columns, can be found here.

To migrate your users from membership to the new identity tables, you should follow the steps described in the documentation. After following these steps and the script provided, your users will need to change their password the next time they log in.

It is possible to migrate user passwords but the process is much more involved. Requiring users to update their passwords as part of the migration process, and encouraging them to use new, unique passwords, is likely to enhance the overall security of the application.

Migrating security settings from web.config to app startup

As noted above, ASP.NET membership and role providers are configured in the application's `web.config` file. Since ASP.NET Core apps are not tied to IIS and use a separate system for configuration, these settings must be configured elsewhere. For the most part, ASP.NET Core Identity is configured in the *Program.cs* file. Open the web project that was created earlier (to generate the identity table schema) and review its *Program.cs* (or *Startup.cs*) file.

This code adds support for EF Core and Identity:

```
// Add services to the container.
var connectionString = builder.Configuration.GetConnectionString("DefaultConnection");
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(connectionString));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();
builder.Services.AddDefaultIdentity<IdentityUser>(options =>
    options.SignIn.RequireConfirmedAccount = true)
    .AddEntityFrameworkStores<ApplicationDbContext>();
```

The `AddDefaultIdentity` extension method is used to configure Identity to use the default `ApplicationDbContext` and the framework's `IdentityUser` type. If you're using a custom `IdentityUser`, be sure to specify its type here. If these extension methods aren't working in your application, check that you have the appropriate using statements and that you have the necessary NuGet package references. For example, your project should have some version of the `Microsoft.AspNetCore.Identity.EntityFrameworkCore` and `Microsoft.AspNetCore.Identity.UI` packages referenced.

Also in *Program.cs* you should see the necessary middleware configured for the site. Specifically, `UseAuthentication` and `UseAuthorization` should be set up, and in the proper location.

```
// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.UseMigrationsEndPoint();
}
else
{
    app.UseExceptionHandler("/Error");
    // The default HSTS value is 30 days. You may want to change this for production
scenarios, see https://aka.ms/aspnetcore-hsts.
    app.UseHsts();
}

app.UseHttpsRedirection();

app.UseStaticFiles();
```

```
app.UseRouting();

app.UseAuthentication();
app.UseAuthorization();

//app.MapControllers();
app.MapBlazorHub();
app.MapFallbackToPage("/_Host");
```

ASP.NET Identity does not configure anonymous or role-based access to locations from *Program.cs*. You will need to migrate any location-specific authorization configuration data to filters in ASP.NET Core. Make note of which folders and pages will require such updates. You will make these changes in the next section.

Updating individual pages to use ASP.NET Core Identity abstractions

In your ASP.NET Web Forms application, if you had `web.config` settings to deny access to certain pages or folders to anonymous users, you would migrate these changes by adding the `[Authorize]` attribute to such pages:

```
@attribute [Authorize]
```

If you further had denied access except to those users belonging to a certain role, you would likewise migrate this behavior by adding an attribute specifying a role:

```
@attribute [Authorize(Roles ="administrators")]
```

The `[Authorize]` attribute only works on `@page` components that are reached via the Blazor Router. The attribute does not work with child components, which should instead use `AuthorizeView`.

If you have logic within page markup for determining whether to display some code to a certain user, you can replace this with the `AuthorizeView` component. The AuthorizeView component selectively displays UI depending on whether the user is authorized to see it. It also exposes a `context` variable that can be used to access user information.

```
<AuthorizeView>
    <Authorized>
        <h1>Hello, @context.User.Identity.Name!</h1>
        <p>You can only see this content if you are authenticated.</p>
    </Authorized>
    <NotAuthorized>
        <h1>Authentication Failure!</h1>
        <p>You are not signed in.</p>
    </NotAuthorized>
</AuthorizeView>
```

You can access the authentication state within procedural logic by accessing the user from a `Task<AuthenticationState` configured with the `[CascadingParameter]` attribute. This configuration will get you access to the user, which can let you determine if they are authenticated and if they belong to a particular role. If you need to evaluate a policy procedurally, you can inject an instance of the `IAuthorizationService` and calls the `AuthorizeAsync` method on it. The following sample code demonstrates how to get user information and allow an authorized user to perform a task restricted by the `content-editor` policy.

```
@using Microsoft.AspNetCore.Authorization
@inject IAuthorizationService AuthorizationService

<button @onclick="@DoSomething">Do something important</button>

@code {
    [CascadingParameter]
    private Task<AuthenticationState> authenticationStateTask { get; set; }

    private async Task DoSomething()
    {
        var user = (await authenticationStateTask).User;

        if (user.Identity.IsAuthenticated)
        {
            // Perform an action only available to authenticated (signed-in) users.
        }

        if (user.IsInRole("admin"))
        {
            // Perform an action only available to users in the 'admin' role.
        }

        if ((await AuthorizationService.AuthorizeAsync(user, "content-editor"))
            .Succeeded)
        {
            // Perform an action only available to users satisfying the
            // 'content-editor' policy.
        }
    }
}
```

The `AuthenticationState` first need to be set up as a cascading value before it can be bound to a cascading parameter like this. That's typically done using the `CascadingAuthenticationState` component. This configuration is typically done in `App.razor`:

```
<CascadingAuthenticationState>
    <Router AppAssembly="@typeof(Program).Assembly">
        <Found Context="routeData">
            <AuthorizeRouteView RouteData="@routeData"
                DefaultLayout="@typeof(MainLayout)" />
        </Found>
        <NotFound>
            <LayoutView Layout="@typeof(MainLayout)">
                <p>Sorry, there's nothing at this address.</p>
            </LayoutView>
        </NotFound>
    </Router>
</CascadingAuthenticationState>
```

Summary

Blazor uses the same security model as ASP.NET Core, which is ASP.NET Core Identity. Migrating from universal providers to ASP.NET Core Identity is relatively straightforward, assuming not too much customization was applied to the original data schema. Once the data has been migrated, working

with authentication and authorization in Blazor apps is well documented, with configurable as well as programmatic support for most security requirements.

References

- Introduction to Identity on ASP.NET Core
- Migrate from ASP.NET Membership authentication to ASP.NET Core 2.0 Identity
- Migrate Authentication and Identity to ASP.NET Core
- ASP.NET Core Blazor authentication and authorization

CHAPTER 14

Migrate from ASP.NET Web Forms to Blazor

Migrating a code base from ASP.NET Web Forms to Blazor is a time-consuming task that requires planning. This chapter outlines the process. Something that can ease the transition is to ensure the app adheres to an *N-tier* architecture, wherein the app model (in this case, Web Forms) is separate from the business logic. This logical separation of layers makes it clear what needs to move to .NET Core and Blazor.

For this example, the eShop app available on GitHub is used. eShop is a catalog service that provides CRUD capabilities via form entry and validation.

Why should a working app be migrated to Blazor? Many times, there's no need. ASP.NET Web Forms will continue to be supported for many years. However, many of the features that Blazor provides are only supported on a migrated app. Such features include:

- Performance improvements in the framework such as `Span<T>`
- Ability to run as WebAssembly
- Cross-platform support for Linux and macOS
- App-local deployment or shared framework deployment without impacting other apps

If these or other new features are compelling enough, there may be value in migrating the app. The migration can take different shapes; it can be the entire app, or only certain endpoints that require the changes. The decision to migrate is ultimately based on the business problems to be solved by the developer.

Server-side versus client-side hosting

As described in the hosting models chapter, a Blazor app can be hosted in two different ways: server-side and client-side. The server-side model uses ASP.NET Core SignalR connections to manage the DOM updates while running any actual code on the server. The client-side model runs as WebAssembly within a browser and requires no server connections. There are a number of differences that may affect which is best for a specific app:

- Running as WebAssembly doesn't support all features (such as threading) at the current time
- Chatty communication between the client and server may cause latency issues in server-side mode

- Access to databases and internal or protected services require a separate service with client-side hosting

At the time of writing, the server-side model more closely resembles Web Forms. Most of this chapter focuses on the server-side hosting model, as it's production-ready.

Create a new project

This initial migration step is to create a new project. This project type is based on the SDK style projects of .NET and simplifies much of the boilerplate that was used in previous project formats. For more detail, please see the chapter on Project Structure.

Once the project has been created, install the libraries that were used in the previous project. In older Web Forms projects, you may have used the *packages.config* file to list the required NuGet packages. In the new SDK-style project, *packages.config* has been replaced with `<PackageReference>` elements in the project file. A benefit to this approach is that all dependencies are installed transitively. You only list the top-level dependencies you care about.

Many of the dependencies you're using are available for .NET, including Entity Framework 6 and log4net. If there's no .NET or .NET Standard version available, the .NET Framework version can often be used. Your mileage may vary. Any API used that isn't available in .NET causes a runtime error. Visual Studio notifies you of such packages. A yellow icon appears on the project's **References** node in **Solution Explorer**.

In the Blazor-based eShop project, you can see the packages that are installed. Previously, the *packages.config* file listed every package used in the project, resulting in a file almost 50 lines long. A snippet of *packages.config* is:

```
<?xml version="1.0" encoding="utf-8"?>
<packages>
  ...
  <package id="Microsoft.ApplicationInsights.Agent.Intercept" version="2.4.0"
targetFramework="net472" />
  <package id="Microsoft.ApplicationInsights.DependencyCollector" version="2.9.1"
targetFramework="net472" />
  <package id="Microsoft.ApplicationInsights.PerfCounterCollector" version="2.9.1"
targetFramework="net472" />
  <package id="Microsoft.ApplicationInsights.Web" version="2.9.1" targetFramework="net472"
/>
  <package id="Microsoft.ApplicationInsights.WindowsServer" version="2.9.1"
targetFramework="net472" />
  <package id="Microsoft.ApplicationInsights.WindowsServer.TelemetryChannel"
version="2.9.1" targetFramework="net472" />
  <package id="Microsoft.AspNet.FriendlyUrls" version="1.0.2" targetFramework="net472" />
  <package id="Microsoft.AspNet.FriendlyUrls.Core" version="1.0.2" targetFramework="net472"
/>
  <package id="Microsoft.AspNet.ScriptManager.MSAjax" version="5.0.0"
targetFramework="net472" />
  <package id="Microsoft.AspNet.ScriptManager.WebForms" version="5.0.0"
targetFramework="net472" />
  ...
  <package id="System.Memory" version="4.5.1" targetFramework="net472" />
  <package id="System.Numerics.Vectors" version="4.4.0" targetFramework="net472" />
  <package id="System.Runtime.CompilerServices.Unsafe" version="4.5.0"
```

```
targetFramework="net472" />
  <package id="System.Threading.Channels" version="4.5.0" targetFramework="net472" />
  <package id="System.Threading.Tasks.Extensions" version="4.5.1" targetFramework="net472"
/>
  <package id="WebGrease" version="1.6.0" targetFramework="net472" />
</packages>
```

The `<packages>` element includes all necessary dependencies. It's difficult to identify which of these packages are included because you require them. Some `<package>` elements are listed simply to satisfy the needs of dependencies you require.

The Blazor project lists the dependencies you require within an `<ItemGroup>` element in the project file:

```
<ItemGroup>
    <PackageReference Include="Autofac" Version="4.9.3" />
    <PackageReference Include="EntityFramework" Version="6.4.4" />
    <PackageReference Include="log4net" Version="2.0.12" />
    <PackageReference Include="Microsoft.Extensions.Logging.Log4Net.AspNetCore"
Version="2.2.12" />
</ItemGroup>
```

One NuGet package that simplifies the life of Web Forms developers is the Windows Compatibility Pack. Although .NET is cross-platform, some features are only available on Windows. Windows-specific features are made available by installing the compatibility pack. Examples of such features include the Registry, WMI, and Directory Services. The package adds around 20,000 APIs and activates many services with which you may already be familiar. The eShop project doesn't require the compatibility pack; but if your projects use Windows-specific features, the package eases the migration efforts.

Enable startup process

The startup process for Blazor has changed from Web Forms and follows a similar setup for other ASP.NET Core services. When hosted server-side, Razor components are run as part of a normal ASP.NET Core app. When hosted in the browser with WebAssembly, Razor components use a similar hosting model. The difference is the components are run as a separate service from any of the backend processes. Either way, the startup is similar.

The *Global.asax.cs* file is the default startup page for Web Forms projects. In the eShop project, this file configures the Inversion of Control (IoC) container and handles the various lifecycle events of the app or request. Some of these events are handled with middleware (such as `Application_BeginRequest`). Other events require the overriding of specific services via dependency injection (DI).

By way of example, the *Global.asax.cs* file for eShop, contains the following code:

```
public class Global : HttpApplication, IContainerProviderAccessor
{
    private static readonly ILog _log =
LogManager.GetLogger(System.Reflection.MethodBase.GetCurrentMethod().DeclaringType);

    static IContainerProvider _containerProvider;
    IContainer container;
```

```
    public IContainerProvider ContainerProvider
    {
        get { return _containerProvider; }
    }

    protected void Application_Start(object sender, EventArgs e)
    {
        // Code that runs on app startup
        RouteConfig.RegisterRoutes(RouteTable.Routes);
        BundleConfig.RegisterBundles(BundleTable.Bundles);
        ConfigureContainer();
        ConfigDataBase();
    }

    /// <summary>
    /// Track the machine name and the start time for the session inside the current
session
    /// </summary>
    protected void Session_Start(Object sender, EventArgs e)
    {
        HttpContext.Current.Session["MachineName"] = Environment.MachineName;
        HttpContext.Current.Session["SessionStartTime"] = DateTime.Now;
    }

    /// <summary>
    /// https://autofaccn.readthedocs.io/en/latest/integration/webforms.html
    /// </summary>
    private void ConfigureContainer()
    {
        var builder = new ContainerBuilder();
        var mockData = bool.Parse(ConfigurationManager.AppSettings["UseMockData"]);
        builder.RegisterModule(new ApplicationModule(mockData));
        container = builder.Build();
        _containerProvider = new ContainerProvider(container);
    }

    private void ConfigDataBase()
    {
        var mockData = bool.Parse(ConfigurationManager.AppSettings["UseMockData"]);

        if (!mockData)
        {

Database.SetInitializer<CatalogDBContext>(container.Resolve<CatalogDBInitializer>());
        }
    }

    protected void Application_BeginRequest(object sender, EventArgs e)
    {
        //set the property to our new object
        LogicalThreadContext.Properties["activityid"] = new ActivityIdHelper();

        LogicalThreadContext.Properties["requestinfo"] = new WebRequestInfo();

        _log.Debug("Application_BeginRequest");
    }
}
```

The preceding file becomes the *Program.cs* file in server-side Blazor:

```
using eShopOnBlazor.Models;
using eShopOnBlazor.Models.Infrastructure;
using eShopOnBlazor.Services;
using log4net;
using System.Data.Entity;
using eShopOnBlazor;

var builder = WebApplication.CreateBuilder(args);

// add services

builder.Services.AddRazorPages();
builder.Services.AddServerSideBlazor();

if (builder.Configuration.GetValue<bool>("UseMockData"))
{
    builder.Services.AddSingleton<ICatalogService, CatalogServiceMock>();
}
else
{
    builder.Services.AddScoped<ICatalogService, CatalogService>();
    builder.Services.AddScoped<IDatabaseInitializer<CatalogDBContext>,
CatalogDBInitializer>();
    builder.Services.AddSingleton<CatalogItemHiLoGenerator>();
    builder.Services.AddScoped(_ => new
CatalogDBContext(builder.Configuration.GetConnectionString("CatalogDBContext")));
}

var app = builder.Build();

new LoggerFactory().AddLog4Net("log4Net.xml");

if (app.Environment.IsDevelopment())
{
    app.UseDeveloperExceptionPage();
}
else
{
    app.UseExceptionHandler("/Home/Error");
}

// Middleware for Application_BeginRequest
app.Use((ctx, next) =>
{
    LogicalThreadContext.Properties["activityid"] = new ActivityIdHelper(ctx);
    LogicalThreadContext.Properties["requestinfo"] = new WebRequestInfo(ctx);
    return next();
});

app.UseStaticFiles();

app.UseRouting();

app.UseEndpoints(endpoints =>
{
    endpoints.MapBlazorHub();
    endpoints.MapFallbackToPage("/_Host");
});

ConfigDataBase(app);
```

```
void ConfigDataBase(IApplicationBuilder app)
{
    using (var scope = app.ApplicationServices.CreateScope())
    {
        var initializer =
scope.ServiceProvider.GetService<IDatabaseInitializer<CatalogDBContext>>();

        if (initializer != null)
        {
            Database.SetInitializer(initializer);
        }
    }
}

app.Run();
```

One significant change you may notice from Web Forms is the prominence of dependency injection (DI). DI has been a guiding principle in the ASP.NET Core design. It supports customization of almost all aspects of the ASP.NET Core framework. There's even a built-in service provider that can be used for many scenarios. If more customization is required, it can be supported by many community projects. For example, you can carry forward your third-party DI library investment.

In the original eShop app, there's some configuration for session management. Since server-side Blazor uses ASP.NET Core SignalR for communication, the session state isn't supported as the connections may occur independent of an HTTP context. An app that uses the session state requires rearchitecting before running as a Blazor app.

For more information about app startup, see App startup.

Migrate HTTP modules and handlers to middleware

HTTP modules and handlers are common patterns in Web Forms to control the HTTP request pipeline. Classes that implement `IHttpModule` or `IHttpHandler` could be registered and process incoming requests. Web Forms configure modules and handlers in the *web.config* file. Web Forms is also heavily based on app lifecycle event handling. ASP.NET Core uses middleware instead. Middleware is registered in the `Configure` method of the `Startup` class. Middleware execution order is determined by the registration order.

In the Enable startup process section, a lifecycle event was raised by Web Forms as the `Application_BeginRequest` method. This event isn't available in ASP.NET Core. One way to achieve this behavior is to implement middleware as seen in the *Startup.cs* file example. This middleware does the same logic and then transfers control to the next handler in the middleware pipeline.

For more information on migrating modules and handlers, see Migrate HTTP handlers and modules to ASP.NET Core middleware.

Migrate static files

To serve static files (for example, HTML, CSS, images, and JavaScript), the files must be exposed by middleware. Calling the `UseStaticFiles` method enables the serving of static files from the web root path. The default web root directory is *wwwroot*, but it can be customized. As included in the *Program.cs* file:

```
...

app.UseStaticFiles();

...
```

The eShop project enables basic static file access. There are many customizations available for static file access. For information on enabling default files or a file browser, see Static files in ASP.NET Core.

Migrate runtime bundling and minification setup

Bundling and minification are performance optimization techniques for reducing the number and size of server requests to retrieve certain file types. JavaScript and CSS often undergo some form of bundling or minification before being sent to the client. In ASP.NET Web Forms, these optimizations are handled at run time. The optimization conventions are defined an *App_Start/BundleConfig.cs* file. In ASP.NET Core, a more declarative approach is adopted. A file lists the files to be minified, along with specific minification settings.

For more information on bundling and minification, see Bundle and minify static assets in ASP.NET Core.

Migrate ASPX pages

A page in a Web Forms app is a file with the *.aspx* extension. A Web Forms page can often be mapped to a component in Blazor. A Razor component is authored in a file with the *.razor* extension. For the eShop project, five pages are converted to a Razor page.

For example, the details view comprises three files in the Web Forms project: *Details.aspx*, *Details.aspx.cs*, and *Details.aspx.designer.cs*. When converting to Blazor, the code-behind and markup are combined into *Details.razor*. Razor compilation (equivalent to what's in *.designer.cs* files) is stored in the *obj* directory and isn't, by default, viewable in **Solution Explorer**. The Web Forms page consists of the following markup:

```
<%@ Page Title="Details" Language="C#" MasterPageFile="~/Site.Master"
AutoEventWireup="true" CodeBehind="Details.aspx.cs"
Inherits="eShopLegacyWebForms.Catalog.Details" %>

<asp:Content ID="Details" ContentPlaceHolderID="MainContent" runat="server">
    <h2 class="esh-body-title">Details</h2>

    <div class="container">
        <div class="row">
```

```
            <asp:Image runat="server" CssClass="col-md-6 esh-picture" ImageUrl='<%#"/Pics/"
+ product.PictureFileName%>' />
            <dl class="col-md-6 dl-horizontal">
                <dt>Name
                </dt>

                <dd>
                    <asp:Label runat="server" Text='<%#product.Name%>' />
                </dd>

                <dt>Description
                </dt>

                <dd>
                    <asp:Label runat="server" Text='<%#product.Description%>' />
                </dd>

                <dt>Brand
                </dt>

                <dd>
                    <asp:Label runat="server" Text='<%#product.CatalogBrand.Brand%>' />
                </dd>

                <dt>Type
                </dt>

                <dd>
                    <asp:Label runat="server" Text='<%#product.CatalogType.Type%>' />
                </dd>
                <dt>Price
                </dt>

                <dd>
                    <asp:Label CssClass="esh-price" runat="server"
Text='<%#product.Price%>' />
                </dd>

                <dt>Picture name
                </dt>

                <dd>
                    <asp:Label runat="server" Text='<%#product.PictureFileName%>' />
                </dd>

                <dt>Stock
                </dt>

                <dd>
                    <asp:Label runat="server" Text='<%#product.AvailableStock%>' />
                </dd>

                <dt>Restock
                </dt>

                <dd>
                    <asp:Label runat="server" Text='<%#product.RestockThreshold%>' />
                </dd>

                <dt>Max stock
                </dt>
```

```
                <dd>
                    <asp:Label runat="server" Text='<%#product.MaxStockThreshold%>' />
                </dd>

            </dl>
        </div>

        <div class="form-actions no-color esh-link-list">
            <a runat="server" href='<%# GetRouteUrl("EditProductRoute", new {id
=product.Id}) %>' class="esh-link-item">Edit
            </a>
            |
            <a runat="server" href="~" class="esh-link-item">Back to list
            </a>
        </div>

    </div>
</asp:Content>
```

The preceding markup's code-behind includes the following code:

```
using eShopLegacyWebForms.Models;
using eShopLegacyWebForms.Services;
using log4net;
using System;
using System.Web.UI;

namespace eShopLegacyWebForms.Catalog
{
    public partial class Details : System.Web.UI.Page
    {
        private static readonly ILog _log =
LogManager.GetLogger(System.Reflection.MethodBase.GetCurrentMethod().DeclaringType);

        protected CatalogItem product;

        public ICatalogService CatalogService { get; set; }

        protected void Page_Load(object sender, EventArgs e)
        {
            var productId = Convert.ToInt32(Page.RouteData.Values["id"]);
            _log.Info($"Now loading... /Catalog/Details.aspx?id={productId}");
            product = CatalogService.FindCatalogItem(productId);

            this.DataBind();
        }
    }
}
```

When converted to Blazor, the Web Forms page translates to the following code:

```
@page "/Catalog/Details/{id:int}"
@inject ICatalogService CatalogService
@inject ILogger<Details> Logger

<h2 class="esh-body-title">Details</h2>

<div class="container">
    <div class="row">
```

```
        <img class="col-md-6 esh-picture" src="@($"/Pics/{_item.PictureFileName}")">

        <dl class="col-md-6 dl-horizontal">
            <dt>
                Name
            </dt>

            <dd>
                @_item.Name
            </dd>

            <dt>
                Description
            </dt>

            <dd>
                @_item.Description
            </dd>

            <dt>
                Brand
            </dt>

            <dd>
                @_item.CatalogBrand.Brand
            </dd>

            <dt>
                Type
            </dt>

            <dd>
                @_item.CatalogType.Type
            </dd>
            <dt>
                Price
            </dt>

            <dd>
                @_item.Price
            </dd>

            <dt>
                Picture name
            </dt>

            <dd>
                @_item.PictureFileName
            </dd>

            <dt>
                Stock
            </dt>

            <dd>
                @_item.AvailableStock
            </dd>

            <dt>
                Restock
            </dt>
```

```
            <dd>
                @_item.RestockThreshold
            </dd>

            <dt>
                Max stock
            </dt>

            <dd>
                @_item.MaxStockThreshold
            </dd>

        </dl>
    </div>

    <div class="form-actions no-color esh-link-list">
        <a href="@($"/Catalog/Edit/{_item.Id}")" class="esh-link-item">
            Edit
        </a>
        |
        <a href="/" class="esh-link-item">
            Back to list
        </a>
    </div>

</div>

@code {
    private CatalogItem _item;

    [Parameter]
    public int Id { get; set; }

    protected override void OnInitialized()
    {
        Logger.LogInformation("Now loading... /Catalog/Details/{Id}", Id);

        _item = CatalogService.FindCatalogItem(Id);
    }
}
```

Notice that the code and markup are in the same file. Any required services are made accessible with the `@inject` attribute. Per the `@page` directive, this page can be accessed at the `Catalog/Details/{id}` route. The value of the route's `{id}` placeholder has been constrained to an integer. As described in the routing section, unlike Web Forms, a Razor component explicitly states its route and any parameters that are included. Many Web Forms controls may not have exact counterparts in Blazor. There's often an equivalent HTML snippet that will serve the same purpose. For example, the `<asp:Label />` control can be replaced with an HTML `<label>` element.

Model validation in Blazor

If your Web Forms code includes validation, you can transfer much of what you have with little-to-no changes. A benefit to running in Blazor is that the same validation logic can be run without needing custom JavaScript. Data annotations enable easy model validation.

For example, the *Create.aspx* page has a data entry form with validation. An example snippet would look like this:

```
<div class="form-group">
    <label class="control-label col-md-2">Name</label>
    <div class="col-md-3">
        <asp:TextBox ID="Name" runat="server" CssClass="form-control"></asp:TextBox>
        <asp:RequiredFieldValidator runat="server" ControlToValidate="Name"
Display="Dynamic"
            CssClass="field-validation-valid text-danger" ErrorMessage="The Name field is
required." />
    </div>
</div>
```

In Blazor, the equivalent markup is provided in a *Create.razor* file:

```
<EditForm Model="_item" OnValidSubmit="@...">
    <DataAnnotationsValidator />

    <div class="form-group">
        <label class="control-label col-md-2">Name</label>
        <div class="col-md-3">
            <InputText class="form-control" @bind-Value="_item.Name" />
            <ValidationMessage For="(() => _item.Name)" />
        </div>
    </div>

    ...
</EditForm>
```

The `EditForm` context includes validation support and can be wrapped around an input. Data annotations are a common way to add validation. Such validation support can be added via the `DataAnnotationsValidator` component. For more information on this mechanism, see ASP.NET Core Blazor forms and validation.

Migrate configuration

In a Web Forms project, configuration data is most commonly stored in the *web.config* file. The configuration data is accessed with `ConfigurationManager`. Services were often required to parse objects. With .NET Framework 4.7.2, composability was added to the configuration via `ConfigurationBuilders`. These builders allowed developers to add various sources for the configuration that was then composed at run time to retrieve the necessary values.

ASP.NET Core introduced a flexible configuration system that allows you to define the configuration source or sources used by your app and deployment. The `ConfigurationBuilder` infrastructure that you may be using in your Web Forms app was modeled after the concepts used in the ASP.NET Core configuration system.

The following snippet demonstrates how the Web Forms eShop project uses *web.config* to store configuration values:

```
<configuration>
  <configSections>
    <section name="entityFramework"
```

```
type="System.Data.Entity.Internal.ConfigFile.EntityFrameworkSection, EntityFramework,
Version=6.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
requirePermission="false" />
  </configSections>
  <connectionStrings>
    <add name="CatalogDBContext" connectionString="Data Source=(localdb)\MSSQLLocalDB;
Initial Catalog=Microsoft.eShopOnContainers.Services.CatalogDb; Integrated Security=True;
MultipleActiveResultSets=True;" providerName="System.Data.SqlClient" />
  </connectionStrings>
  <appSettings>
    <add key="UseMockData" value="true" />
    <add key="UseCustomizationData" value="false" />
  </appSettings>
</configuration>
```

It's common for secrets, such as database connection strings, to be stored within the *web.config*. The secrets are inevitably persisted in unsecure locations, such as source control. With Blazor on ASP.NET Core, the preceding XML-based configuration is replaced with the following JSON:

```
{
  "ConnectionStrings": {
    "CatalogDBContext": "Data Source=(localdb)\\MSSQLLocalDB; Initial
Catalog=Microsoft.eShopOnContainers.Services.CatalogDb; Integrated Security=True;
MultipleActiveResultSets=True;"
  },
  "UseMockData": true,
  "UseCustomizationData": false
}
```

JSON is the default configuration format; however, ASP.NET Core supports many other formats, including XML. There are also several community-supported formats.

You can access configuration values from the builder in *Program.cs*:

```
if (builder.Configuration.GetValue<bool>("UseMockData"))
{
    builder.Services.AddSingleton<ICatalogService, CatalogServiceMock>();
}
else
{
    builder.Services.AddScoped<ICatalogService, CatalogService>();
    builder.Services.AddScoped<IDatabaseInitializer<CatalogDBContext>,
CatalogDBInitializer>();
    builder.Services.AddSingleton<CatalogItemHiLoGenerator>();
    builder.Services.AddScoped(_ => new
CatalogDBContext(builder.Configuration.GetConnectionString("CatalogDBContext")));
}
```

By default, environment variables, JSON files (*appsettings.json* and *appsettings.{Environment}.json*), and command-line options are registered as valid configuration sources in the configuration object. The configuration sources can be accessed via `Configuration[key]`. A more advanced technique is to bind the configuration data to objects using the options pattern. For more information on configuration and the options pattern, see Configuration in ASP.NET Core and Options pattern in ASP.NET Core, respectively.

Migrate data access

Data access is an important aspect of any app. The eShop project stores catalog information in a database and retrieves the data with Entity Framework (EF) 6. Since EF 6 is supported in .NET 5, the project can continue to use it.

The following EF-related changes were necessary to eShop:

- In .NET Framework, the `DbContext` object accepts a string of the form *name=ConnectionString* and uses the connection string from `ConfigurationManager.AppSettings[ConnectionString]` to connect. In .NET Core, this isn't supported. The connection string must be supplied.
- The database was accessed in a synchronous way. Though this works, scalability may suffer. This logic should be moved to an asynchronous pattern.

Although there isn't the same native support for dataset binding, Blazor provides flexibility and power with its C# support in a Razor page. For example, you can perform calculations and display the result. For more information on data patterns in Blazor, see the Data access chapter.

Architectural changes

Finally, there are some important architectural differences to consider when migrating to Blazor. Many of these changes are applicable to anything based on .NET Core or ASP.NET Core.

Because Blazor is built on .NET Core, there are considerations in ensuring support on .NET Core. Some of the major changes include the removal of the following features:

- Multiple AppDomains
- Remoting
- Code Access Security (CAS)
- Security Transparency

For more information on techniques to identify necessary changes to support running on .NET Core, see Port your code from .NET Framework to .NET Core.

ASP.NET Core is a reimagined version of ASP.NET and has some changes that may not initially seem obvious. The main changes are:

- No synchronization context, which means there's no `HttpContext.Current`, `Thread.CurrentPrincipal`, or other static accessors
- No shadow copying
- No request queue

Many operations in ASP.NET Core are asynchronous, which allows easier off-loading of I/O-bound tasks. It's important to never block by using `Task.Wait()` or `Task.GetResult()`, which can quickly exhaust thread pool resources.

Migration conclusion

At this point, you've seen many examples of what it takes to move a Web Forms project to Blazor. For a full example, see the eShopOnBlazor project.