Microsoft

Blazor for ASP.NET Web Forms Developers

Daniel Roth

Jeff Fritz

Taylor Southwick

Preview Edition

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

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.

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 Core 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.

Send your feedback

This book and related samples are constantly evolving, so your feedback is welcomed! If you have comments about how this book can be improved, use the feedback section at the bottom of any page built on GitHub issues.

Contents

CHAPTER 1

An introduction to Blazor for ASP.NET Web Forms developers

ASP.NET Web Forms 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. A rich ecosystem of Web Forms UI controls provided by Microsoft and control vendors makes connecting to data sources and displaying rich visualizations easy. 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 new trends in web development, such as 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 and 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 this writing, almost half a million web developers use ASP.NET Web Forms every month. ASP.NET Web Forms 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 virtually 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, but ASP.NET Web Forms development remains a popular framework for creating web apps.

Even so, with the rapid pace of innovation and change in software development, all software developers need to stay abreast of new technologies and trends. This is true for ASP.NET Web Forms developers as well. Two trends in particular are worth considering:

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

An open-source and cross-platform .NET

When .NET and ASP.NET Web Forms first shipped, the platform ecosystem looked very different than it does today. The desktop and server markets were strongly dominated by Windows, with alternative platforms like macOS and Linux still struggling to get 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 are not 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. While there are risks associated with open-source, many open-source 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 variety of platforms, including Windows, macOS, and various Linux distributions. Xamarin provides Mono, an open-source version of .NET that runs on Android, iOS, and a variety of other form factors, including watches and smart TVs. Microsoft has announced that for .NET 5 both .NET Core and Mono will be reconciled 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 this 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 recently made it clear that ASP.NET Web Forms will not be brought to .NET Core or .NET 5. Why is that? Efforts were made in the early days of .NET Core to bring over ASP.NET Web Forms, but the number of breaking changes required were found to be too drastic. There is perhaps also an admission here that even for Microsoft, there is 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 continues to ship 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 5, then sticking with ASP.NET Web Forms on Windows is just

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 then executes some code (.NET code in ASP.NET apps) to produce a response that is sent back to the browser to handle. In this model, the browser is used as a thin rendering engine. All of the hard work of producing the UI, running the business logic, and managing state happens on the server.

However, browsers have become versatile platforms implementing 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 using web API calls or even over real-time protocols, like WebSockets. All of these benefits are available to web developers for free as long as they're willing to write some JavaScript. Client-side UI frameworks that simplify client-side web development, like Angular, React, and Vue, 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 having to bridge two different platforms and ecosystems (.NET and JavaScript) comes with a cost. You, as the web developer, have to become an expert in two parallel worlds with different languages, frameworks, and tools. Code and logic cannot 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, quickly changing preferences for different front-end frameworks (such as jQuery, Knockout, Angular, React, Vue, and so on) and build tools (such as grunt, gulp, webpack, and so on). 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 to the rescue!

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. The idea is that 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++, resulting in some pretty dramatic demonstrations of getting native 3D graphics engines running directly in the browser without any plugins. WebAssembly has since been standardized and implemented by all major browsers. Work on running .NET on WebAssembly was announced in late 2017 and is expected to ship in 2020, including support from

.NET 5. The ability to run .NET code directly in the browser enables full-stack web development with .NET.

Blazor: full-stack web development with .NET

Just being able to run .NET code in a browser doesn't give you an end-to-end experience building client-side web apps. That's where Blazor comes in. Blazor is a new client-side web UI framework based on C# instead of JavaScript that can run directly in the browser via WebAssembly. No browser plugins are required. Alternatively, Blazor apps can run server-side on .NET Core and handle all user interactions over a real-time connection with the browser. Blazor includes a full UI component model and built-in facilities for forms and validation, dependency injection, client-side routing, layouts, in-browser debugging, and JavaScript interop. Blazor also comes with great tooling support in Visual Studio and Visual Studio Code.

Blazor shares a lot in common with ASP.NET Web Forms. Both frameworks are component-based, event-driven, stateful UI programming models. The main architectural difference is that ASP.NET Web Forms runs on the server, while 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 very 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 should have a solid understanding of how to build Blazor apps, how Blazor works, how it ties in with .NET Core, and also some reasonable strategies for migrating existing ASP.NET Web Forms apps to Blazor where appropriate.

Getting started with Blazor

Getting started with Blazor is easy. Go to https://blazor.net and follow the links to install the appropriate .NET Core 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

ASP.NET Web Forms

Blazor

Blazor app hosting models

Blazor apps can be hosted in IIS just like ASP.NET Web Forms apps. Blazor apps can also be hosted in one of the following ways:

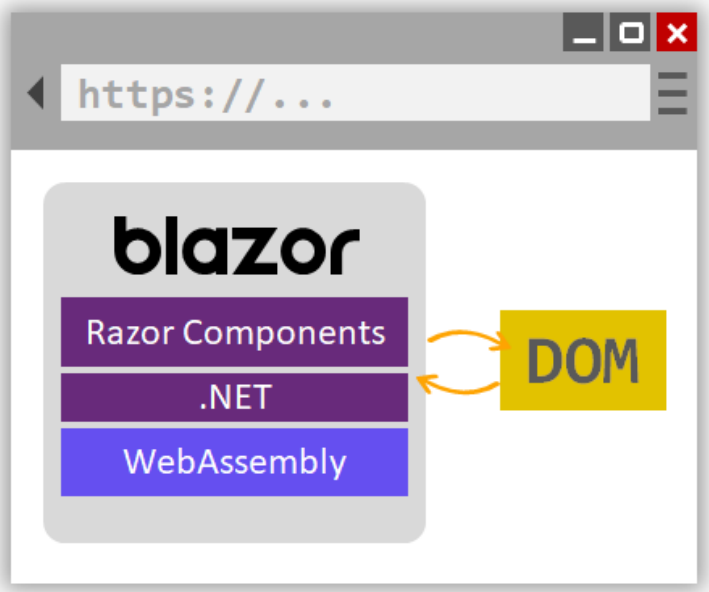
- Client-side in the browser on WebAssembly.
- Server-side in an ASP.NET Core app.

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.



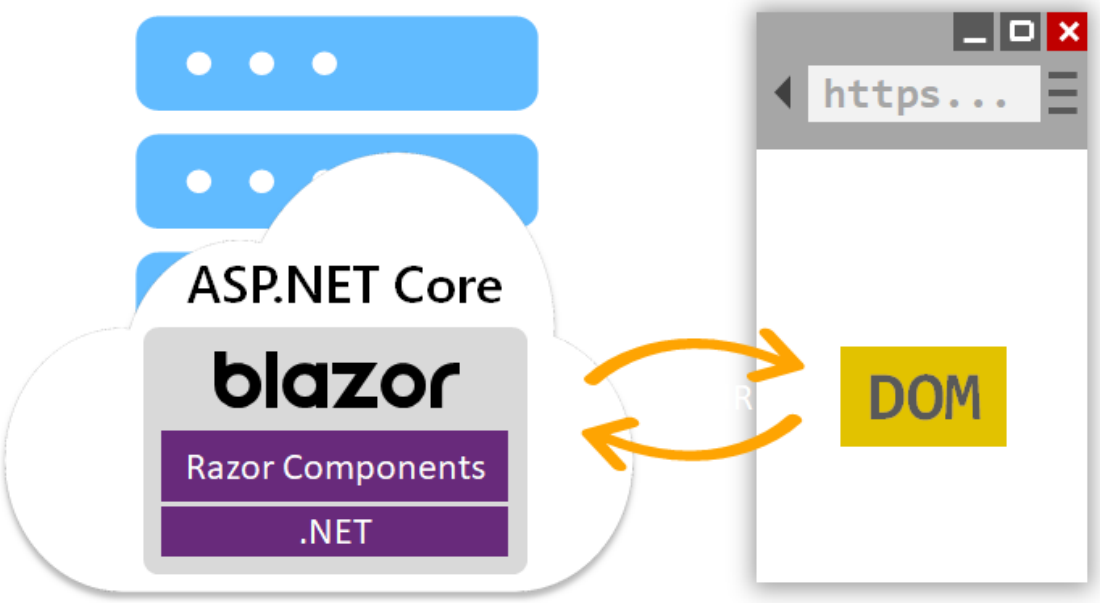
Blazor WebAssembly apps run purely client-side. Such apps can be deployed to static site hosting solutions like GitHub Pages or Azure Static Website Hosting. .NET isn't required on the server at all. Deep linking to parts of the app typically requires a routing solution on the server. The routing solution 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 Core-compatible APIs.
- .NET Core 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 will 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. Most of the code in both projects is the same; although, the hosting model-specific logic will differ.

Project file

Blazor Server apps are .NET Core 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>netcoreapp3.0</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.Web">

  <PropertyGroup>
    <TargetFramework>netstandard2.0</TargetFramework>
    <RazorLangVersion>3.0</RazorLangVersion>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.Blazor" Version="3.1.0" />
    <PackageReference Include="Microsoft.AspNetCore.Blazor.Build" Version="3.1.0"
PrivateAssets="all" />
    <PackageReference Include="Microsoft.AspNetCore.Blazor.HttpClient" Version="3.1.0" />
    <PackageReference Include="Microsoft.AspNetCore.Blazor.DevServer" Version="3.1.0"
PrivateAssets="all" />
  </ItemGroup>
```

```
  <ItemGroup>
    <ProjectReference Include="..\Shared\BlazorWebAssemblyApp1.Shared.csproj" />
  </ItemGroup>

</Project>
```

Blazor WebAssembly projects target .NET Standard instead of .NET Core 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. Many of the simplifications in the .NET Core- and .NET Standard-based projects come from the default targets and properties imported by referencing the `Microsoft.NET.Sdk.Web` SDK, often referred to as simply the "Web SDK". The Web SDK includes wildcards and other conveniences that simplify inclusion of code and content files in the project. You don't need to list the files explicitly. When targeting .NET Core, the Web SDK also adds framework references to both the .NET Core and ASP.NET Core shared frameworks. The frameworks are visible from the **Dependencies** > **Frameworks** node in the Solution Explorer window. The shared frameworks are collections of assemblies that were installed on the machine when installing .NET Core.

Although they're supported, individual assembly references are less common in .NET Core projects. Most project dependencies are handled as NuGet package references. You only need to reference top-level package dependencies in .NET Core 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 element.

```
<ItemGroup>
  <PackageReference Include="Newtonsoft.Json" Version="12.0.2" />
</ItemGroup>
```

Entry point

The Blazor Server app's entry point is defined in the *Program.cs* file, just like 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 like configuration, logging, dependency injection, and the HTTP server. This code is mostly boilerplate and is often left unchanged.

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

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
```

```
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

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 have a `Startup` class instead of a *Global.asax* file to define the startup logic for the app. The `Startup` class is used to configure the app and any app-specific services. In the Blazor Server app, the `Startup` class is used to set up the endpoint for the real-time connection used by Blazor between the client browsers and the server. In the Blazor WebAssembly app, the `Startup` class defines the root components for the app and where they should be rendered. We'll take a deeper look at the `Startup` class in the App startup section.

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 the app's "web root". Anything outside of the app's web root is *not* web-addressable. This setup provides an additional level of security that prevents accidental exposing 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": "*"
}
```

We'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. We'll look at 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` statements for commonly used namespaces:

```
@using System.Net.Http
@using Microsoft.AspNetCore.Authorization
@using Microsoft.AspNetCore.Components.Authorization
@using Microsoft.AspNetCore.Components.Forms
@using Microsoft.AspNetCore.Components.Routing
@using Microsoft.AspNetCore.Components.Web
@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]{custom-style=Code} 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. 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.

We'll look in greater detail at routing in Blazor in the Pages and routing section.

Layout

In ASP.NET Web Forms apps, common page layout is handled using master pages (*Site.Master*). In Blazor apps, page layout is handled using layout components (*Shared/MainLayout.razor*). Layout components will be discussed in more detail in Page layout 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. The `Html.RenderComponent(RenderMode)` method 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 options exist:

- Rendered as static content (`RenderModel.Static`)
- Rendered interactively once a connection with the browser has been established (`RenderMode.Server`)
- First prerendered and then rendered interactively (`RenderMode.ServerPrerendered`)

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.

```
@page "/"
@namespace BlazorApp1.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" />
    <title>BlazorApp1</title>
    <base href="~/" />
    <link rel="stylesheet" href="css/bootstrap/bootstrap.min.css" />
    <link href="css/site.css" rel="stylesheet" />
</head>
<body>
    <app>
        @(await Html.RenderComponentAsync<App>(RenderMode.ServerPrerendered))
    </app>

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

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

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width" />
    <title>BlazorApp2</title>
    <base href="/" />
    <link href="css/bootstrap/bootstrap.min.css" rel="stylesheet" />
    <link href="css/site.css" rel="stylesheet" />
</head>
<body>
    <app>Loading...</app>

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

The specific component to render is configured in the app's `Startup.Configure` method with a corresponding CSS selector indicating where the component should be rendered.

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
    }

    public void Configure(IComponentsApplicationBuilder app)
    {
        app.AddComponent<App>("app");
    }
}
```

Build output

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

Run the app

To run the Blazor Server app, press `F5` in Visual Studio. Blazor apps don't support runtime compilation. To see the results of code changes, including component markup changes, rebuild and restart the app when running with the debugger. If you run without the debugger attached (`Ctrl+F5`), Visual Studio watches for file changes and restarts the app as changes are made. You manually refresh the browser as changes are made.

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 don't support debugging using Visual Studio. To run the app, use `Ctrl+F5` instead of `F5`. You can instead debug Blazor WebAssembly apps directly in the browser. See Debug ASP.NET Core Blazor for details.

App startup

Build reusable UI components with Blazor

- Event handling
- Post backs
- Client interactivity

CHAPTER 7

Pages and routing

Page layout

Managing state

- View state
- Session state
- Local storage
- App state

Dealing with data

- Entity Framework
- Forms and validation
- Data sources and controls
- Calling Web APIs

CHAPTER 11

Modules, handlers, and middleware

App configuration

CHAPTER 13

Security: authentication and authorization in ASP.NET Web Forms and Blazor

Migrating from ASP.NET Web Forms to Blazor

- Doing it piecemeal
- Move to .NET Standard
- API portability scanner
- Project file format
- ...