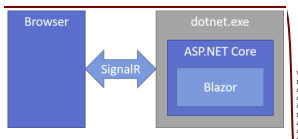
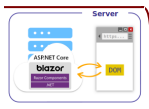
# Blazor Server




With this hosting model, the application is executed on the server. Between the client and the server a SignalR connection is established. When an event occurs on the client such as a button click for example, the information about the event is sent to the server over the SignalR connection. The server handles the event and for the generated HTML a diff (difference) is calculated. The entire HTML is not sent again to the client, it's only the diff that is sent to the client over the SignalR connection. The browser then updates the UI. Since only the diff is applied to update the UI, the application feels faster and more responsive to the user.




## Blazor server hosting model

### Benefits

The app loads much faster as the download size is significantly smaller than a Blazor WebAssembly app
Since the app runs on the server, it can take full advantage of server capabilities, including use of any .NET Core compatible APIs.
All the client needs, to use the app is a browser. Even browsers that don't support WebAssembly can be used.
More secure because the app's .NET/C# code isn't served to clients.

Advantages of Blazor Server
- Smaller download size
- The app takes full advantage of server capabilities
- Thin clients are supported

### Downsides

A full-blown ASP.NET Core server is required to host the application. Serverless deployment scenarios such as serving the app from a CDN aren't possible.
An active connection to the server is always required. This means there is a need to keep the server up and running 24X7. If the server is down, the application stops working.
As every user interaction involves a round trip to the server a higher latency usually exists when compared with Blazor WebAssembly hosting.
Scalability can be challenging especially for the apps that have many users as the server must manage multiple client connections and handle client state. However, we can overcome this scalability issue, by using Azure SignalR Service with a Blazor Server app.
This service allows a Blazor Server app to scale really well by supporting a large number of concurrent SignalR connections.

Disadvantages of Blazor Server
- Higher UI latency
- There's no offline support
- Scalability is challenging for apps with many users

https://www.pragimtech.com/blog/blazor/blazor-hosting-models/

## Blazor project structure

### Program.cs

This file contains the Main() method which is the entry point for both the project types (i.e Blazor WebAssembly and Blazor Server).

In a Blazor server project, the Main() method calls CreateHostBuilder() method which sets up the ASP.NET Core host.

### wwwroot

Contains the static files like images, stylesheets etc.

### App.razor

This is the root component of the application. It uses the built-in Router component and sets up client-side routing. It is this Router component that intercepts browser navigation and renders the page that matches the requested address.

### Pages folder

Index component (Index.razor) – Rendered when we navigate to the root application URL.
Counter component (Counter.razor) – Rendered when we navigate to the path /counter.
FetchData component (FetchData.razor) – Rendered when we navigate to the path /fetchdata.
Error component (Error.razor) – Rendered when an unhandled exception occurs in the blazor app.

Contains the _Host razor page and the routable components that make up the Blazor app. The components have the .razor extension.

### Shared folder

Contains the shared components

MainLayout component (MainLayout.razor)

Implements the navigation menu on the sidebar.
NavLink component, renders navigation links to other Razor components like the index, counter and fetchdata components. This NavLink component is intelligent enough to highlight the navigation menu item, if it's component is currently displayed.

NavMenu component (NavMenu.razor)

### Imports.razor

This file contains the common namespaces so we do not have to include them in every razor component. (This is like the _ViewImports.cshtml file in an asp.net core MVC project.)

### Startup.cs

Configures the applications DI i.e dependency injection services. For example AddServerSideBlazor() method adds Blazor server side services. On the IServiceCollection interface there are several methods that start with the word Add. These methods add different services for the Blazor application. We can even add our own services to the DI container.

#### ConfigureServices

It contains the applications's startup logic. The Startup class contains the following two methods.

Configures the app's request processing pipeline. Depending on what we want the Blazor application to be capable of doing we add or remove the respective middleware components from request processing pipeline.

#### Configure

MapBlazorHub sets up the endpoint for the SignalR connection with the client browser.

### Pages/_Host.cshtml

This is the root page of the application and is specified by calling MapFallBackToPage("/_Host") method. It is implemented as a razor page. It is this page, that is initially served when a first request hits the application. It has the standard HTML, HEAD and BODY tags. It also specifies where the root application component, App component (App.razor) must be rendered. Finally, it also loads the blazor.server.js JavaScript file, which sets up the real-time SignalR connection between the server and the client browser. This connection is used to exchange information between the client and the server. SignalR, is a great framework for adding real-time web functionality to apps.

### Data folder (Blazor Server)

#### appsettings.json (Blazor Server)

A blazor project also uses this file to store the Configuration settings.

### Conclusion

Remember everything in a Blazor application is a razor component. Components are the fundamental building blocks of a Blazor application.

https://www.pragimtech.com/blog/blazor/blazor-project-structure/

## ASP.NET Core razor components

Blazor is a component-driven framework, meaning components are the fundamental building blocks of a Blazor application.
They can be nested, reused, and if implemented properly, can even be shared across multiple projects.
Component files have the extension .razor

HTML markup which defines the user interface of the component i.e the look and feel.
C# code which defines the processing logic.

When the application is compiled, the HTML and C# code converted into a component class. The name of the generated class matches the name of the component file. A component file name must start with an uppercase character. If you add a component file that starts with a lower case character, the code will fail to compile and you get the following compiler error.

**Component names cannot start with a lowercase character**

A SignalR connection is established between the server and the client browser. After the component is initially rendered and when the user clicks the button, The information about the click event is sent to the server over the SignalR connection.
In response to the event, the component is regenerated, but the entire HTML is not sent back to the client, it's only the diff, i.e the difference in the render tree, in this case, the new counter value that is sent to the client browser.
Since only the changed part of the page is updated instead of reloading and updating the entire page, the application feels faster and more responsive to the user.

### Nesting razor components

This path is specified by the @page directive at the top of the component.
A component can also be nested inside another component using HTML syntax.

### Where to place components

Components can be placed anywhere within a blazor project. It's a good practice to place components that produce webpages in the Pages folder and reusable non-page components in the Shared folder. If you want to, you can also place them in a completely different custom folder within your project.

### Split component HTML and C# code

In this example, both the HTML and C# are in a single file. This is fine for a simple component like this counter, but it is usually a good practice to separate HTML and C# code into their own files. It is not only good from maintenance standpoint, but it is also easy to unit test. There are 2 approaches, to split component HTML and C# code into their own separate files.

#### Partial files approach

The HTML markup remains in Counter.razor file.
Remember, when the component is compiled a class with the same name as the component file is generated. Create another class file with name Counter.razor.cs and include the following code in it. Notice the class in this file is implemented as a partial class.

#### Base class approach

Just like the partial class approach, even with the base class approach, the HTML markup remains in Counter.razor file.
Move the C# code to a separate class. I named it CounterBase. You can name the class anything you want, but it is a common convention to have the same name as the component but suffixed with the word Base.
The class has to inherit from the built-in ComponentBase class. This class is in Microsoft.AspNetCore.Components namespace.
The access modifier must be at least protected if you wish to access the class member from the HTML
Finally, in Counter.razor file do not forget to include the following inherits directive.

#### Single file or Mixed file approach

Both the HTML markup and C# code are in a single file.

## Razor vs Blazor

### Introduction to ASP.NET Core Blazor

https://docs.microsoft.com/en-us/aspnet/core/blazor/?view=aspnetcore-5.0

### SPA

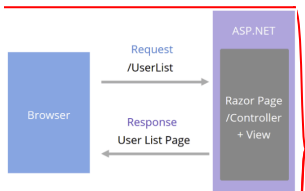
Blazor is an alternative to MVC and Razor Pages but with a twist: It's a single page app framework (SPA) that just happens to use C# instead of JavaScript.

#### single page app framework

### Blazor lifecycle

The components you build for your Blazor applications are officially called Razor Components. You write them using Razor! This is still the Razor we're used to. You can do everything you could before (use loops, conditionals and so on). But there is one crucial difference with Blazor we're dealing in components, not pages.

- OnInitialized() / OnInitializedAsync() — It is executed when the component is completely loaded.
- OnParametersSet() / OnParametersSetAsync() — When a component is first initialized, and each time new or updated parameters are received from parent in the render tree.
- OnAfterRender(bool firstRender) / OnAfterRenderAsync(bool firstRender) — Called after each render of the component.
- ShouldRender() — This method returns a Boolean value, if true, it refreshes the UI, otherwise changes are not sent to UI.
- StateHasChanged() — This method notifies the component that its state has changed.




MVC and Razor Pages have always focused on serving entire pages to the browser.

Every time the user clicks a button, or a link, the browser makes a request to the server, which hits the database, gets the .cshtml Razor Views (or Pages), mashes the data and markup together then returns the whole lot back to the browser (which re-renders the entire page).

Razor takes care of rendering your HTML based on the data in your model, while also supporting various conditionals and loops.

With Blazor, you are able to build your user interface as a series of components instead.

#### Components All the Way Down

https://www.telerik.com/blogs/difference-between-blazor-vs-razor

Blazor is a technology similar to ASP.NET Core & ASP.NET MVC in that:

It powers web applications
It uses Razor as its template syntax for the creation of UI.

A common point of misconception is that Blazor uses Razor. This further exacerbated by two other similar terms – Blazor components and Razor components. Those are widely used interchangeably but the correct terminology is Razor Component. A component is the combination of markup (written in HTML) and logic (in C#) in a single Razor file (with the .cshtml extension).

Razor is a popular template markup syntax for .NET. Blazor (Browser + Razor) is a .NET based web framework which can run on the client using WebAssembly or running on the server via SignalR.

https://www.telerik.com/faqs/blazor-ui/what-is-the-difference-between-blazor-vs-razor

https://espressocoder.com/2019/02/21/whats-the-difference-between-razor-and-blazor/

Simply putting, Blazor is a free open source Single Page Application development framework. It uses C# instead of Javascript to create rich interactive web applications.

#### Blazor

https://hexishub.medium.com/blazor-vs-razor-1dd7546bcf29

https://hexis-hub.com/meanshore-portugal-insights/articles/blazor-razor

https://www.c-sharpcorner.com/article/asp-net-mvc-razor-vs-blazor/

## Models & data

Create clean model classes and easily bind them to your database. Declaratively define validation rules, using C# attributes, which are applied on the client and server.

The Model is responsible for managing the data of the application. It contains no business logic. In simple term, models are the entity representation of the database table that consists of the properties of table columns.

## Controllers

Simply route requests to controller actions, implemented as normal C# methods. Data from the request path, query string, and request body are automatically bound to method parameters.

Controller handles the user interaction and it is reponsible for invoking the action based on the route and fetching data from the model to render on the view.

## Views

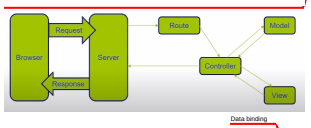
The Razor syntax provides a simple, clean and lightweight way to render HTML content based on your view. Razor lets you render a page using C#, producing fully HTML5 compliant web pages.

View is the user inderface. It contains the Markup (Razor syntax). It uses the Razor engine to render the view. View renders the model data passed to it via any controller. We can also write the C# code in the view.

## MVC

### What is MVC?

- Model View Controller or MVC is a design pattern for developing applications.
- MVC isolates the application logic from the user interface layer and supports the separation of concerns.
- In MVC architecture, the controller receives the HTTP request and then works with the model to get the data and render it on the view.
- The controller specifies which view to render based on the HTTP request.




## Blazor basics

- Data binding
- Components & Routing
- Parameters / Cascading Parameters
- Javascript in Blazor

### Blazor Server

- Admin Management Portal
- Authentication & Authorization
- Quill JS
- Seed Database

### .NET 5 API

- Endpoint for Blazor Client
- Bearer Token
- CORS
- Data Transfer Object

## ASP.NET Core fundamentals

### History

As we know, ASP.NET is the framework that has been used to develop data-driven web applications for many years. Since then, the ASP.NET Framework went through a steady evolutionary change, and finally, the most decent evolution is ASP.NET Core.

The ASP.NET Core is not a continuous part of the ASP.NET 4.x Framework. Instead, it is a completely new framework.
This Framework is an actual re-write of the current ASP.NET 4.x Framework, but with much smaller and a lot more modular.
Some people think that many things remain the same, but that is not completely true. The ASP.NET Core is actually a big fundamental change to the ASP.NET Framework.

ASP.NET Core is the new version of the ASP.NET Web Framework mainly targeted to run on .NET Core Platform.

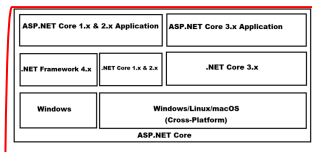
ASP.NET Core is a free, open-source, high-performance, light-weight, and cross-platform framework for building cloud-based applications, such as web apps, IoT apps, and mobile backend. It is designed to run on the cloud as well as on-premises.

Same as .NET Core, it was architected modular with minimum overhead, and then other more advanced features can be added as NuGet Packages as per application requirement. This results in high performance, require less memory, less deployment size, and easy to maintain.

ASP.NET Core Framework is completely rewritten of existing ASP.NET 4.x, with architectural changes that make it more modular, scalable, open-source, high-performance, lightweight, and a cross-platform web framework.

The ASP.NET Core is an open-source framework supported by Microsoft and the community, so you can also contribute or download the source code from the ASP.NET Core Repository on GitHub (https://github.com/dotnet/aspnetcore)

ASP.NET Core 3.x application can be targeted to develop and run on top of the .NET Core only whereas ASP.NET Core 1.x and 2.x application can be targeted to develop and run on top of the .NET Core (Cross-platform) as well as .NET Framework (Windows only). For better understanding, please have a look at the following image.




Open Source:
The ASP.NET Core framework is Open Source which is the main reason behind its popularity. The Entire Source Code of this .NET Core Framework is available at https://github.com/aspnet and you are free to download the source code and even if you want then you can also modify and compile your own version of it.

.NET Core team is always there to support your effort in the seamless development of the application. It receives bug fixing and improvement updates on a regular basis usually within a short time period. You don't have to wait longer for updates. The flexibility in framework code is keeping it at the top of the developer's choice. Depending upon the project's requirements libraries and framework's component can be added later.

Cross-Platform:
The ASP.NET Core Framework is designed from scratch to keep in mind to be Cross-Platform for both development and deployment. So, we don't need to build different applications for different platforms using different frameworks. Let us discuss what Cross-Platform is from the ASP.NET Core point of view by comparing it with the earlier versions of the ASP.NET Framework.

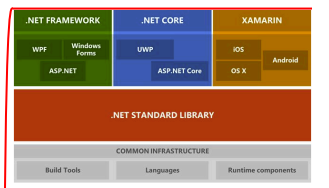
The earlier versions of ASP.NET Framework applications can only run on windows platforms whereas the ASP.NET Core applications can develop and run on different platforms such as Windows, Mac, or Linux operating systems. We can host the earlier ASP.NET Framework 4.x applications only on IIS whereas we can host the ASP.NET Core applications on IIS, Nginx, Docker, Apache, or even self-host deployment.

To develop ASP.NET Core applications, you have multiple options like you can use either Visual Studio or Visual Studio Code. If you want, then you can also use any third-party editors as per your choice.

CLI Support: Using CLI commands you can develop and run .NET applications as well as you can also publish the application using CLI command.

Fast: ASP.NET Core no longer depends on System.Web.dll for browser-server communication. ASP.NET Core allows us to include packages that we need for the application. This reduces the request pipeline and improves performance and scalability.

IoC Container: One of the most important used design patterns in the real-time application is the Dependency Injection Design Pattern. It includes the built-in IoC (Inversion of Control) container for automatic dependency injection which makes it maintainable and testable.




https://www.c-sharpcorner.com/article/difference-between-net-framework-and-net-core/

.NET Core is a better choice if you:
Want to target your apps on Windows, Linux, and Mac operating systems.
Are not afraid of learning new things.
Are not afraid of breaking and fixing things since .NET Core is not fully matured yet.
A student who is just learning .NET.
Love open source.

.NET Core Advantages

If you're building a new application and have a choice between .NET Core and .NET Framework, .NET Core is the way to go.

Microsoft just announced .NET Core v 3.0, which is a much-improved version of .NET Core. If you want to learn and build for the future, .NET Core is the way.

.NET 3.0 now supports WPF and Windows Forms. .NET Core 3.0 also supports cross-development between UWP, WPF, and Windows Forms. This provides developers the flexibility to bring modern interfaces of UWP into Windows Forms and WPF.

If you're building a new app and your company has approved it, .NET Core is the way to go. .NET Core will continue to get better and better.

.NET Core is better suited for cross-platform needs. .NET Core apps are supported on Windows, Linux, and macOS. Microsoft's popular open-source code editor, Visual Studio Code, is supported on Windows, Linux, and macOS. VS Code supports the modern needs of code editors including IntelliSense and debugging. Most third-party editors, such as Sublime, Emacs, and Vi, work with .NET Core.

Microservices architecture is supported in .NET Core, which allows cross-platform services to work with .NET Core including services developed with .NET Framework, Java, Ruby, or others.

Containers are the VMs of today. .NET Core's modularity, lightweight, and flexibility makes it easier to deploy .NET Core apps in containers. Containers can be deployed on any platform, cloud, Linux, and Windows. . NET Core works well with both Docker and Azure Kubernetes Service.

A need for high performance and scalable systems.

Performance and scalability were the two major key focus areas when .NET Core was developed. .NET Core and ASP.NET Core are the top-performing web frameworks according to some benchmarks.

.NET Core versions play well together. You can easily run multiple versions and apps side by side on the same machine.

Power of C#

C# is the most versatile programming languages among all and it is changing every few month. Every minor release of the languages introduces some cool and modern features. The current versions C# 8.0 and C# 9.0 have introduced several new features to make programmers' lives easier. The language is designed to write clean, faster, and modern code. The language is designed to be productive. Check out C# 8 Features and Deep Dive into C# 9 to learn more about these feature.

Note: ASP.NET Core was initially launched as ASP.NET S but later it was renamed to ASP.NET Core.

.Net Core:
.NET Core is open-source and cross-platform
.NET Core is a runtime to execute applications which ate build on it.
Install .NET Core Runtime to run applications and install .NET Core SDK to build applications.
.NET Core 3.1 – latest version

ASP.NET Core:
Asp.net core is Open-source and Cross-platform
ASP.NET Core is a web framework to build web apps, IoT apps, and mobile backend on top of .NET Core.
There is no separate runtime and SDK are available for ASP.NET Core. .NET Core runtime and SDK includes ASP.NET Core libraries.
ASP.NET Core 3.1 – latest version
There is no separate versioning for ASP.NET Core. It is the same as the .NET Core versions.

### The Startup class

Services required by the app are configured.
The app's request handling pipeline is defined, as a series of middleware components.

https://docs.microsoft.com/en-us/aspnet/core/fundamentals/startup?view=aspnetcore-5.0

### Dependency injection (services)

ASP.NET Core includes a built-in dependency injection (DI) framework that makes configured services available throughout an app.

Services are typically resolved from DI using constructor injection. With constructor injection, a class declares a constructor parameter of either the required type or an interface. The DI framework provides an instance of this service at runtime.

If the built-in Inversion of Control (IoC) container doesn't meet all of an app's needs, a third-party IoC container can be used instead.

https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-5.0

### Middleware

The request handling pipeline is composed as a series of middleware components. Each component performs operations on an HttpContext and either invokes the next middleware in the pipeline or terminates the request.

By convention, a middleware component is added to the pipeline by invoking a Use... extension method in the Startup.Configure method. For example, to enable rendering of static files, call UseStaticFiles.

https://docs.microsoft.com/en-us/aspnet/core/fundamentals/middleware/?view=aspnetcore-5.0

### Host

On startup, an ASP.NET Core app builds a host. The host encapsulates all of the app's resources

- An HTTP server implementation
- Middleware components
- Logging
- Dependency injection (DI) services
- Configuration

There are two different hosts:

- .NET Generic Host — The .NET Generic Host is recommended. The ASP.NET Core Web Host is available only for backwards compatibility.
- ASP.NET Core Web Host

The CreateDefaultBuilder and ConfigureWebHostDefaults methods configure a host with a set of default options

- Use Kestrel as the web server and enable IIS integration.
- Load configuration from appsettings.json, appsettings.{Environment Name}.json, environment variables, command line arguments, and other configuration sources.
- Send logging output to the console and debug providers.

https://docs.microsoft.com/en-us/aspnet/core/fundamentals/host/generic-host?view=aspnetcore-5.0

### Servers

An ASP.NET Core app uses an HTTP server implementation to listen for HTTP requests. The server surfaces requests to the app as a set of request features composed into an HttpContext.

https://docs.microsoft.com/en-us/aspnet/core/fundamentals/servers/?view=aspnetcore-5.0

### Configuration

ASP.NET Core provides a configuration framework that gets settings as name-value pairs from an ordered set of configuration providers. Built-in configuration providers are available for a variety of sources, such as .json files, .xml files, environment variables, and command-line arguments. Write custom configuration providers to support other sources.

By default, ASP.NET Core apps are configured to read from appsettings.json, environment variables, the command line, and more. When the app's configuration is loaded, values from environment variables override values from appsettings.json.

The preferred way to read related configuration values is using the options pattern. For more information, see Bind hierarchical configuration data using the options pattern.

https://docs.microsoft.com/en-us/aspnet/core/fundamentals/configuration/?view=aspnetcore-5.0

### Environments

Execution environments, such as Development, Staging, and Production, are a first-class notion in ASP.NET Core. Specify the environment an app is running in by setting the ASPNETCORE_ENVIRONMENT environment variable. ASP.NET Core reads that environment variable at app startup and stores the value in an IWebHostEnvironment implementation. This implementation is available anywhere in an app via dependency injection (DI).

### Logging

ASP.NET Core supports a logging API that works with a variety of built-in and third-party logging providers.

To create logs, resolve an ILogger<TCategoryName> service from dependency injection (DI) and call logging methods such as LogInformation.

### Routing

A route is a URL pattern that is mapped to a handler. The handler is typically a Razor page, an action method in an MVC controller, or a middleware. ASP.NET Core routing gives you control over the URLs used by your app.

https://docs.microsoft.com/en-us/aspnet/core/fundamentals/routing?view=aspnetcore-5.0

### Error handling

ASP.NET Core has built-in features for handling errors.

- A developer exception page
- Custom error pages
- Static status code pages
- Startup exception handling

https://docs.microsoft.com/en-us/aspnet/core/fundamentals/error-handling?view=aspnetcore-5.0

### Content root

The executable hosting the app (.exe).
Compiled assemblies that make up the app (.dll).
Content files used by the app, such as:
- Razor files (.cshtml, .razor)
- Configuration files (.json, .xml)
- Data files (.db)
- The Web root, typically the wwwroot folder.

The content root is the base path for:

During development, the content root defaults to the project's root directory. This directory is also the base path for both the app's content files and the Web root. Specify a different content root by setting its path when building the host.

https://docs.microsoft.com/en-us/aspnet/core/fundamentals/host/generic-host?view=aspnetcore-5.0#contentroot

### Web root

The web root is the base path for public, static resource files, such as:

- Stylesheets (.css)
- JavaScript (.js)
- Images (.png, .jpg)

By default, static files are served only from the web root directory and its sub-directories. The web root path defaults to {content root}/wwwroot. Specify a different web root by setting its path when building the host.

https://docs.microsoft.com/en-us/aspnet/core/fundamentals/host/generic-host?view=aspnetcore-5.0#webroot

Prevent publishing files in wwwroot with the <Content> project item in the project file. The following example prevents publishing content in wwwroot/local and its sub-directories:

In Razor .cshtml files, tilde-slash (~/) points to the web root. A path beginning with ~/ is referred to as a virtual path.

https://docs.microsoft.com/en-us/aspnet/core/fundamentals/static-files?view=aspnetcore-5.0