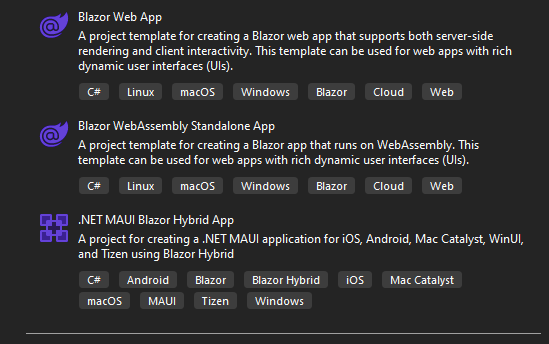
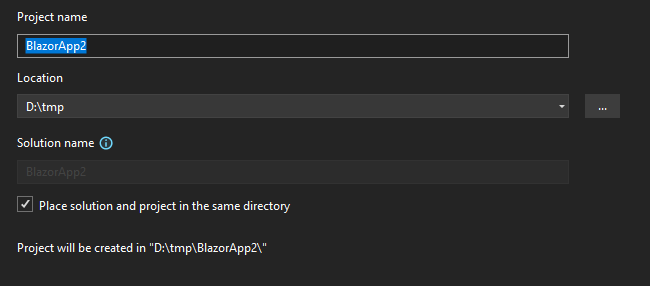
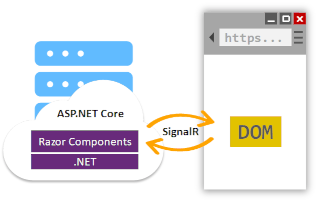
# Blazor 8

1. A whole bunch of great articles about Blazor have been published on the internet not so long ago, see the [Blazor University](https://blazor-university.com/) site. Microsoft has released a brand new version of Blazor. It has become a great WEB programming tool, but some of the statements and recommendations in the aforementioned series of articles are outdated and require corrections. I will describe here what, where and how it has changed.
2. First of all, it should be emphasized that the current version of **Blazor** requires:
3. Windows 11,
4. .NET 8
5. Visual Studio 2022 (Community Edition or any paid version).

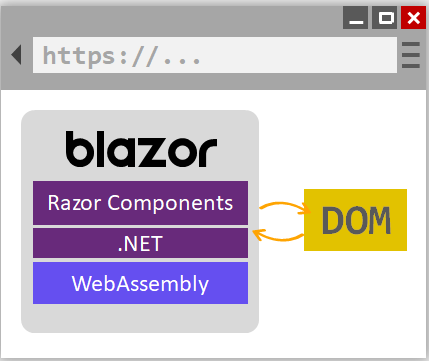
## Project templates

1. Visual Studio offers 3 different templates after creating new application:
2. 
3. **Blazor WebAssembly Standalone** application will create an application very similar to the one you had in your Windows 10 environment.
4. **.NET MAUI Blazor Hybrid App** is a Xamarin enhancement. This template creates applications that work as operating system applications and access the server directly, without a browser.
5. **Blazor Web App** is intended for the development of new WEB applications, and here the new **Blazor** variant works at full power. Visual studio asks for project parameters after selecting **Blazor Web App**. The dialog is the same as in the previous version of Blazor:
6. After setting the project parameters, the main dialog appears:
7. 
8. **Framework dropdown**: there are currently no options. More options will probably appear in this control later.
9. **Authentication type**: here you can choose **none** or **Individual Accounts**. Visual Studio will place the ..\Components\Account directory in the server part project after choosing the second option. The directory contains the files required for name/password authentication. The main menu is also expanded. We will talk about authentication and its configuration later, in a separate section.
10. **Configure for HTTPS**: this checkbox needs to be checked when you use standard authentication. The **SSL certificate** will be used by Blazor even after unchecking this control, as it is required when the page works in Server Side Rendering (SSR) mode. We'll talk about this later when we look at the different rendering modes.
11. **Interactive render mode** allows you to choose one of four options:
12. **None** - in this mode, applications will load extremely fast as there is no work required on the client, and no large WebAssembly assets to download. The server is just sending HTML that the browser then renders. This also means that each request for a new page results in a full page load. This kind of application is an enhancement of Razor pages: you can use Blazor components here, but you will have to use JavaScript for handling events.
13. **Server** - in this mode a page, or component, will be optionally pre-rendered on the server and then made interactive on the client via a **SignalR** connection. Once interactive, all events on the client will be transmitted back to the server over the SignalR connection to be processed on the server. Any updates required to the DOM will then be packaged up and sent to the client over the same SignalR connection where a small Blazor runtime will patch the updates into the DOM.
14. **WebAssembly** - this model is derived from the Blazor WebAssembly hosting model and fully capitalises on client-side capabilities, allowing C# code to run in the user’s browser. Initial data would be downloaded to the client along with the various framework DLLs and WebAssembly runtime. Once on the client, it would be bootstrapped and the page loaded. Any API calls to get data would be made and the UI would be re-rendered as necessary to display any data returned.
15. **Auto (Server and WebAssembly)** - it is main mode of Blazor applications in .NET 8. When setting a page or component to use Auto mode, the initial load of that component will be via server mode making it super fast. But in the background Blazor will download the necessary assets to the client so that on the next load it can be done using WebAssembly mode. This rendering mode will address the biggest pain point for developers when embarking on a new Blazor project, what hosting model should we use? Every component marked with **RenderMode.Auto** will need to execute on both the server and the client. Meaning that there will need to be some form of abstraction in place if the component needs to fetch any data.

### Blazor Server

1. 
2. **ASP.NET Core** apps and **Blazor Server** use the Razor language to describe HTML content for rendering, but they significantly differ in how markup is rendered.
3. When a Razor Page or view is rendered, every line of Razor code emits HTML in text form. After rendering, the server disposes of the page or view instance, including any state that was produced. When another request for the page occurs, the entire page is rerendered to HTML again and sent to the client.
4. Blazor Server produces a graph of components to display similar to an HTML or XML DOM. The component graph includes state held in properties and fields. Blazor evaluates the component graph to produce a binary representation of the markup, which is sent to the client for rendering. After the connection is made between the client and the server, the component's static prerendered elements are replaced with interactive elements. Prerendering content on the server in order to load HTML content on the client quickly makes the app feel more responsive to the client.
5. After the components are interactive on the client, UI updates are triggered by user interaction and app events. When an update occurs, the component graph is rerendered, and a UI diff (difference) is calculated. This diff is the smallest set of DOM edits required to update the UI on the client. The diff is sent to the client in a binary format and applied by the browser.

### Blazor WebAssembly

1. 
2. Running .NET code inside web browsers is made possible by **WebAssembly** (abbreviated wasm). **WebAssembly** is a compact bytecode format optimized for fast download and maximum execution speed. WebAssembly is an open web standard and supported in web browsers without plugins. WebAssembly works in all modern web browsers, including mobile browsers.
3. **WebAssembly** code can access the full functionality of the browser via JavaScript, called **JavaScript interoperability**, often shortened to JavaScript interop or JS interop. .NET code executed via WebAssembly in the browser runs in the browser's JavaScript sandbox with the protections that the sandbox provides against malicious actions on the client machine.
4. When a Blazor WebAssembly app is built and run:
5. C# code files and Razor files are compiled into .NET assemblies.
6. The assemblies and the [.NET runtime](https://learn.microsoft.com/en-us/dotnet/framework/get-started/overview) are downloaded to the browser.
7. Blazor WebAssembly bootstraps the .NET runtime and configures the runtime to load the assemblies for the app. The Blazor WebAssembly runtime uses JavaScript interop to handle DOM manipulation and browser API calls.
8. The size of the published app, its payload size, is a critical performance factor for an app's usability. A large app takes a relatively long time to download to a browser, which diminishes the user experience.
9. For apps that require third-party JavaScript libraries and access to browser APIs, components interoperate with JavaScript. Components are capable of using any library or API that JavaScript is able to use. C# code can call into JavaScript code, and JavaScript code can call into C# code.
10. Blazor implements the .NET Standard, which enables Blazor projects to reference libraries that conform to .NET Standard specifications. .NET Standard is a formal specification of .NET APIs that are common across .NET implementations. .NET Standard class libraries can be shared across different .NET platforms, such as Blazor, .NET Framework, .NET Core, Xamarin, Mono, and Unity.APIs that aren't applicable inside of a web browser (for example, accessing the file system, opening a socket, and threading) throw a **PlatformNotSupportedException**.

### Auto

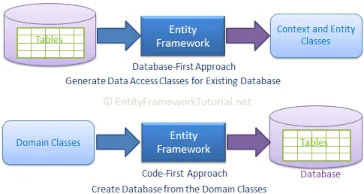
1. Set these parameters on next screen:
2. **Framework** - .NET 8.x,
3. **Authentication type** – Individual accounts,
4. **Configure for HTTPS** – true,
5. **Interactive render mode** – Auto (Server and WebAssembly),
6. **Interactivity location** – Per page/component,
7. **Include sample pages** – true,
8. **Do not use top-level statements** – false.
9. Visual Studio will create two projects after setting these parameters: server and client. Here I’ll demonstrate how these two projects may be used for processing any database.
10. The ~/Data directory has the ApplicationDbContext.cs file, which contains DB context for the Entity Framework:
    1. public class ApplicationDbContext(DbContextOptions<**ApplicationDbContext**> options)
    2. : IdentityDbContext<**ApplicationUser**>(options)
    3. {
    4. }
    5. Second file in this directory is **ApplicationUser.cs**. This file defines **ApplicationUser** class:
    6. public class ApplicationUser : IdentityUser
    7. {
    8. }
11. In the generated version, this class contains only those fields that are described in the standard class **Microsoft.AspNetCore.Identity**. You can add any number of additional fields.
12. The **appsettings.json** file contains the database connection: see to the "ConnectionStrings" section. The local database is specified in the generated task:
13. "ConnectionStrings": {
14. "DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=aspnet-BlazorApp4-0c7655a7-1f05-43cd-921a-72502c56eb35;Trusted\_Connection=True;MultipleActiveResultSets=true"
15. },
16. **"Server=…**" is one long string. JSON doesn't allow breaking lines for readability.

#### Entity Framework (EF)

Blazor server uses **EF Core 8** for work with SQL databases. EF Core supports two development approaches 1) Code-First 2) Database-First. EF Core mainly targets the code-first approach and provides little support for the database-first approach because the visual designer or wizard for DB model is not supported as of EF Core.

In the code-first approach, EF Core API creates the database and tables using migration based on the conventions and configuration provided in your domain classes. This approach is useful in Domain Driven Design (DDD).

In the database-first approach, EF Core API creates the domain and context classes based on your existing database using EF Core commands. This has limited support in EF Core as it does not support visual designer or wizard.



##### EF Core Database Providers

Entity Framework Core uses a provider model to access many different databases. EF Core includes providers as NuGet packages which you need to install.

Here is a list of database providers and NuGet packages for EF Core:

1. **SQL Server** Microsoft.EntityFrameworkCore.SqlServer
2. **MySQL** MySql.Data.EntityFrameworkCore
3. **PostgreSQL** Npgsql.EntityFrameworkCore.PostgreSQL
4. **SQLite** Microsoft.EntityFrameworkCore.SQLite
5. **SQL Compact** EntityFrameworkCore.SqlServerCompact40
6. **In-memory** Microsoft.EntityFrameworkCore.InMemory

**EF Core** is not included as a default package in .NET 7. We need to install it via the NuGet package.

The **Microsoft.EntityFrameworkCore** is the base package for all the basic operations of EF Core. However, you have to install a database provider package from NuGet for the database you use in your project. For example, to use the MS SQL Server database, you need to install **Microsoft.EntityframeworkCore.SqlServer** package.

The database provider package of EF Core includes all the other dependent packages it needs. So, it includes **Microsoft.EntityFrameworkCore** too. So, no need to install it separately.

EF can be installed from Visual Studio by opening the '**Package manager**' console or by running the '**dotnet add package**' command from the terminal. Before running the latter command, change to the directory containing the \*.csproj file.

The '**Powershell**' window can also be opened from Visual Studio: **'right click**' on the project and select '**Open in terminal**'. Type this command in the window:

dotnet add package Microsoft.EntityFrameworkCore.SqlServer

EF can also be installed after modifying the \*.csproj file. Open this file in any text editor and add the command into ItemGroup:

<ItemGroup>

<ProjectReference Include="..\BlazorApp4.Client\BlazorApp4.Client.csproj" />

<ProjectReference Include="..\StudentsManagement\StudentsManagement.csproj" />

<PackageReference Include="Microsoft.AspNetCore.Components.WebAssembly.Server" Version="8.0.0" />

<PackageReference Include="Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore" Version="8.0.0" />

<PackageReference Include="Microsoft.AspNetCore.Identity.EntityFrameworkCore" Version="8.0.0" />

<PackageReference Include="Microsoft.EntityFrameworkCore" Version="8.0.1" />

<**PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer" Version="8.0.0"** />

<PackageReference Include="Microsoft.EntityFrameworkCore.Tools" Version="8.0.0" />

<PackageReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Design" Version="8.0.0" />

<PackageReference Include="System.Diagnostics.Debug" Version="4.3.0" />

</ItemGroup>

##### The model

With EF Core, data access is performed using a **model**. A model is made up of **entity classes** and a **context** object that represents a session with the database. The context object allows querying and saving data.

EF supports the following model development approaches:

* Generate a model from an existing database.
* Hand code a model to match the database. Once a model is created, use EF Migrations to create a database from the model. Migrations allow evolving the database as the model changes:

using System.Collections.Generic;

using Microsoft.EntityFrameworkCore;

namespace Intro;

public class BloggingContext : DbContext

{

public DbSet<Blog> Blogs { get; set; }

public DbSet<Post> Posts { get; set; }

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

optionsBuilder.UseSqlServer(

@"Server=(localdb)\mssqllocaldb;Database=Blogging;Trusted\_Connection=True");

}

}

public class Blog

{

public int BlogId { get; set; }

public string Url { get; set; }

public int Rating { get; set; }

public List<Post> Posts { get; set; }

}

public class Post

{

public int PostId { get; set; }

public string Title { get; set; }

public string Content { get; set; }

public int BlogId { get; set; }

public Blog Blog { get; set; }

}

This template is only suitable for small databases. For large databases with many relations, place a configuration into static class of each entity:

using Microsoft.EntityFrameworkCore;

namespace BlazorApp4.Models

{

public class Student

{

public int Id { get; set; }

public string? FirstName { get; set; }

public string? MiddleName { get; set; }

public string? LastName { get; set; }

public string? EmailAddress { get; set; }

public string? Address { get; set; }

public string? PhoneNumber { get; set; }

public string? Country { get; set; }

public Guid Version { get; set; }

public static void **Configure**(ModelBuilder builder)

{

builder.Entity<Student>().HasKey(s => s.Id);

builder.Entity<Student>().Property(s => s.Id).ValueGeneratedOnAdd();

builder.Entity<Student>().Property(s => s.Version).IsConcurrencyToken();

}

}

}

The static **Configure** function will be called from the same OnModelCreating event:

namespace BlazorApp4.Data

{

public class ApplicationDbContext(DbContextOptions<ApplicationDbContext> options) :

IdentityDbContext<ApplicationUser>(options)

{

public DbSet<Student> Students { get; set; }

protected override void **OnModelCreating**(ModelBuilder builder)

{

base.OnModelCreating(builder);

**Student.Configure(builder)**;

}

}

}

#### MS SQL Server

1. It is enough to change the **DefaultConnection** element for connecting to the MS SQL server. Here is connection to server on my computer:
2. "ConnectionStrings": {
3. "DefaultConnection": "Data Source=GedoDell;Initial Catalog=BlazorApp4;Integrated Security=True;Encrypt=False;MultipleActiveResultSets=true"
4. },
5. The connection string can be specified in several different ways. All they are all listed in the [ConnectionStrings](https://www.connectionstrings.com/sql-server/) site.

#### Migrations

1. Entity Framework knows how to create a database from C# classes. Create a Data directory and add the following class to it:
2. public class Student
3. {
4. public int Id { get; set; }
5. public string? FirstName { get; set; }
6. public string? MiddleName { get; set; }
7. public string? LastName { get; set; }
8. public string? EmailAddress { get; set; }
9. public string? Address { get; set; }
10. public string? PhoneNumber { get; set; }
11. public string? Country { get; set; }
12. public Guid Version { get; set; }
13. public static void Configure(ModelBuilder builder)
14. {
15. builder.Entity<Student>().HasKey(s => s.Id);
16. builder.Entity<Student>().Property(s => s.Id).ValueGeneratedOnAdd();
17. builder.Entity<Student>().Property(s => s.Version).IsConcurrencyToken();
18. }
19. }
20. The Configure static function provides the [Fluent API](https://www.learnentityframeworkcore.com/configuration/fluent-api) functions that describe the table columns. This example describes the primary key (**Id**) and a field for catching concurrency errors (**Version**). The **IsConcurrencyToken** method is used to specify that a property should be included in a WHERE clause in an **UPDATE** or **DELETE** statement as part of [concurrency management](https://www.learnentityframeworkcore.com/concurrency). Some properties of the model can be described by [attributes](https://www.learnentityframeworkcore.com/configuration/data-annotation-attributes) but **FluentAPI** provides much more configuration options.
21. Now all that remains is to modify the **ApplicationDbContext** class: list all data sets and the Configure functions of these sets (the **Data/ApplicationDbContext.cs** file ):
22. public class ApplicationDbContext(DbContextOptions<ApplicationDbContext> options) :
23. IdentityDbContext<ApplicationUser>(options)
24. {
25. public DbSet<Student> Students { get; set; }
26. protected override void OnModelCreating(ModelBuilder builder)
27. {
28. base.OnModelCreating(builder);
29. **Student.Configure**(builder);
30. }
31. }
32. With this data, you can use [data migration tools](https://www.infoworld.com/article/3691114/how-to-work-with-ef-core-migrations-in-aspnet-core.html) and create a new database.
33. Now install two packages: **Microsoft.VisualStudio.Web.CodeGeneration.Design** and **Microsoft.EntityFrameworkCore**.
34. If you don't want a type to be included in the model, you can exclude it:

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Ignore<BlogMetadata>();

}

With this configuration migrations will not create the AspNetUsers table, but IdentityUser is still included in the model and can be used normally.

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<IdentityUser>()

.ToTable("AspNetUsers", t => t.ExcludeFromMigrations());

}

##### Table name

By convention, each entity type will be set up to map to a database table with the same name as the **DbSet** property that exposes the entity. If no DbSet exists for the given entity, the class name is used. You can manually configure the table name:

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Blog>()

.ToTable("blogs");

}

##### Table schema

When using a relational database, tables are by convention created in your database's default schema. For example, Microsoft SQL Server will use the **dbo** schema (SQLite does not support schemas).You can configure tables to be created in a specific schema as follows:

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Blog>()

.ToTable("blogs", schema: "blogging");

}

Rather than specifying the schema for each table, you can also define the default schema at the model level with the fluent API:

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.HasDefaultSchema("blogging");

}

##### View mapping

Entity types can be mapped to database views using the Fluent API:

modelBuilder.Entity<Blog>()

.ToView("blogsView", schema: "blogging");

EF will assume that the referenced view already exists in the database, it will not create it automatically in a migration. Mapping to a view will remove the default table mapping, but the entity type can also be mapped to a table explicitly. In this case the query mapping will be used for queries and the table mapping will be used for updates.

1. https://learn.microsoft.com/en-us/ef/core/modeling/entity-types?tabs=fluent-api
2. https://www.entityframeworktutorial.net/efcore/create-entities.aspx
3. [EPISODE 1 On How to Create Simple CRUD .NET 8.0 Blazor With Auto Render Magic,EF Core, SQL Server. (youtube.com)](https://www.youtube.com/watch?v=xgeoNVsIwug)
4. [EPISODE 2 On How to Create Simple CRUD .NET 8.0 Blazor With Auto Render Magic,EF Core, SQL Server.](https://www.youtube.com/watch?v=7M6Pz86Vw3Q)