# Blazor 8

This course differs from the others in that it is designed for the Windows 11 operating system. Microsoft has provided a great description of ASP.NET Core and it is very difficult to say anything new. Here I will describe the creation of a WEB site, providing only links to the original [documentation](https://learn.microsoft.com/en-us/aspnet/core/blazor/?view=aspnetcore-8.0).

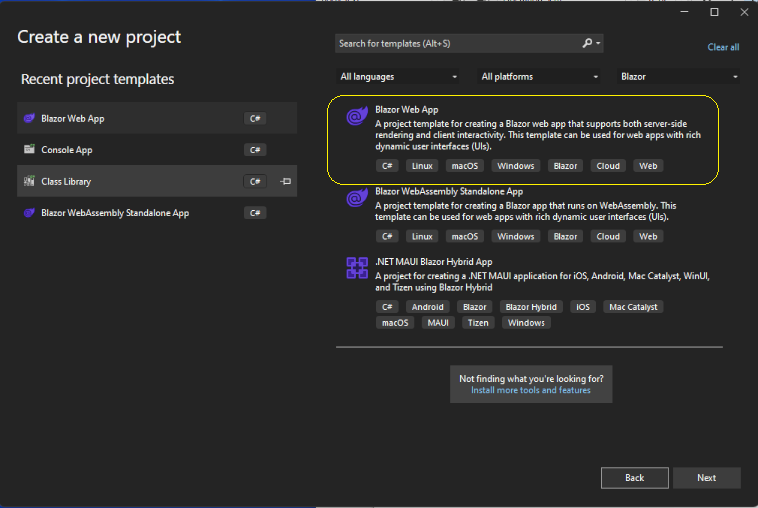
You will need:

1. a computer with the Windows 11 operating system installed,
2. Visual Studio; I am using Microsoft Visual Studio Community edition v4.8.09032; you can use the newer free or commercial version; download it fom Microsoft [site](https://visualstudio.microsoft.com/downloads/),
3. Tools for work with local database; I shall explain WEB programming with MS SQL Server; you can use MySQL, PostgreSQL, SQL Server or SQLite; the database itself can be packed inside a Docker container.
4. [NorthWind](https://en.wikiversity.org/wiki/Database_Examples/Northwind) database; you would install MS SQL server and NortWind variant adapted for this server.

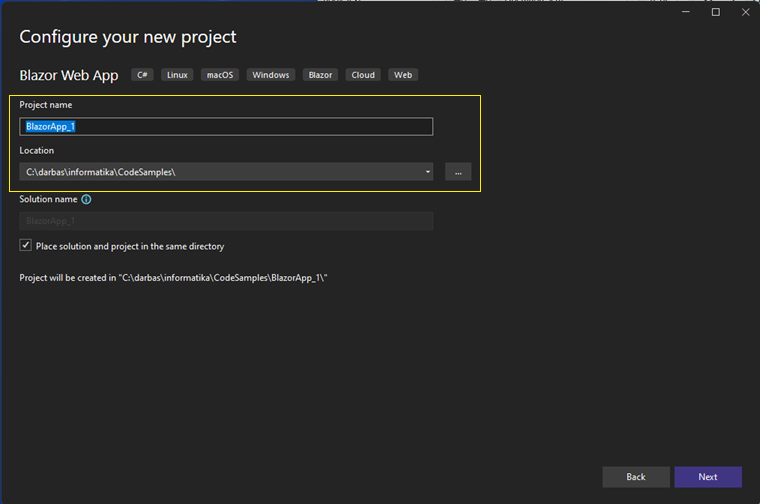
In the archive ‘https://github.com/gbukauskas/BlazorCodeDemo’ you will find the complete code of this WEB task. The branches in this archive correspond to the steps involved in developing this application.

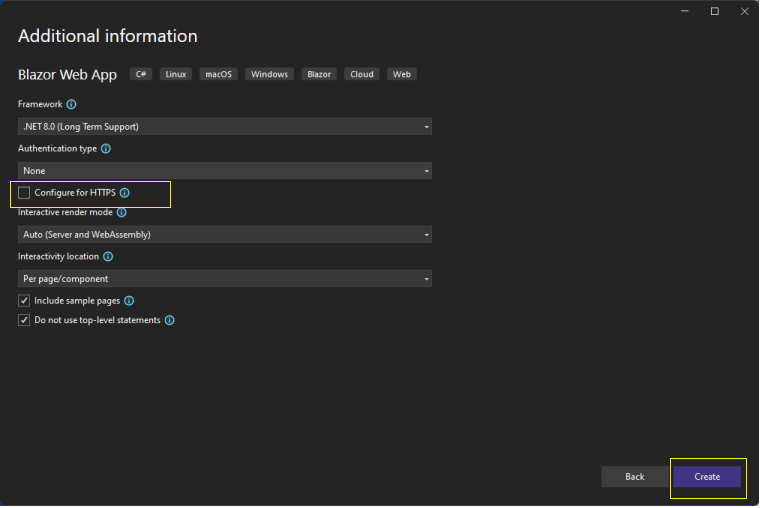
## Creating new application

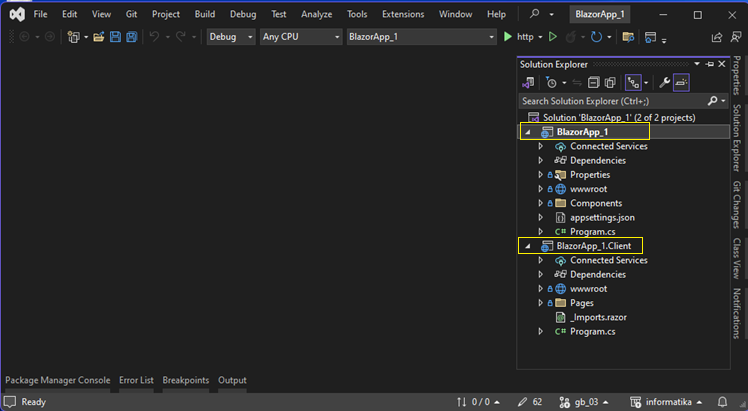
After launching Visual Studio, select "Create a new project". In the next dialog, select “Blazor Web App”:



Select project name and location on next screen:

Check off „Configure for HTTPS“ and click on „Create“ in the last screen:

Don’t worry about security: you will be enable it before the deployment. You will see two projects after a short pause:

Check that you have done everything correctly, .NET and Visual Studio versions if your project differs from this picture. Your .NET version should be at least 8.0.200 and Visual Studio - 17.9.2 or higher. Correct code is published on branch

https://github.com/gbukauskas/BlazorCodeDemo/tree/gb\_01

By default, the **Blazor Web App** template enables both **static** and **interactive server-side** rendering using a single project. If you also enable **Interactive WebAssembly** rendering, the project includes an additional client project (**.Client**) for your WebAssembly-based components. The built output from the client project is downloaded to the browser and executed on the client. Components using the Interactive WebAssembly or Interactive Auto render modes must be located in the **.Client** project.

## Create Database

For a full demonstration of the WEB task, you will need a database. I chose the free and well-known [NorthWind](https://code.google.com/archive/p/northwindextended/downloads) database. If you have Oracle, PostgreSql, MySql or MSSql server installed on your computer, download the version that suits you and install it. You can also use a remote SQL server, but make sure it is accessible from your computer. This is especially important working with PostgreSQL, since the standard installation blocks remote connections. I will work with MS SQL server, you can choose any other SQL database.

The **SQLite** is a C-language library that implements a [small](http://footprint.html/), [fast](http://fasterthanfs.html/), [self-contained](http://selfcontained.html/), [high-reliability](http://hirely.html/), [full-featured](http://fullsql.html/), SQL database engine. SQLite is the [most used](http://mostdeployed.html/) database engine in the world. SQLite is built into all mobile phones and most computers and comes bundled inside countless other applications that people use every day. All tools needed for work with this database can be found at the [SQLite](https://www.sqlite.org/index.html) link. You will also find the documentation there. Use this engine if your computer has no local database. Follow the [how-to](https://www.allhandsontech.com/programming/blazor/how-to-sqlite-blazor/) article installing **Northwind.Sqlite3.sql**. There is a nice [tutorial](https://www.tutorialspoint.com/sqlite/sqlite_create_database.htm) on WEB. Follow this short instruction:

1. Download sqlite-tools-win-x64-3450200.zip from [SQLite](https://www.sqlite.org/download.html) home site. Extract the content into any directory and put the path into environment variables.
2. Run CMD and navigate to the directory where you want to create the database. Run these commands:

> sqlite3 Northwind.db

sqlite> .read <path\_to\_Northwind.Sqlite3.sql>

sqlite> .exit

>

1. Launch these commands for to verifying the database:

> sqlite3 Northwind.db

sqlite> SELECT \* FROM sqlite\_master WHERE type='table';

sqlite> .exit

>

You will need **Browser for SQLite** for work with SQLite database. Download it from [home](https://sqlitebrowser.org/dl/) site and install this application.

The **MS SQL** Server variant (Northwind.Ms.SQL.2005.sql) is quite old and has a couple of deprecated commands (**exec sp\_dboption**, lines 22-24). Comment them out before running the script. Open the Northwind.Ms.SQL.2005.sql file with Microsoft SQL Server Management Studio, comment lines with sp\_dboption and run the script. It will create Northwind database and populate it.

The **PostgreSQL** variant of the script (northwind.postgre.sql) does not create **NorthWind** database thus you will need to create the database before loading data. Owner of database should be your the most frequently used login. The process is described in the **02\_SQL.docx** document, section „**change root password**“.

You can load data using the **psql** command after creating **NorthWind database**:

>psql

Password for user xxxx:

<your password>

...

xxxx=#\**c NorthWind**

...

xxxx=#**\i <path to northwind.postgre.sql>**

The **MySQL** variant of the script (Northwind.MySQL5.sql) also does not create **NorthWind** database. The task can be done using **phpMyAdmin**, but on slow computers, use the **mysql** CLI tool instead:

docker container start mysql-server

docker cp <**NorthWind\_on\_your\_computer**> /tmp

docker exec -it mysql-server mysql -p

Enter password: <your root\_password>

mysql > CREATE DATABASE NorthWind;

mysql> USE NorthWind;

mysql> source /tmp/Northwind.MySQL5.sql;

**NorthWind\_on\_your\_computer** is a directory with downloaded files on your local computer. **tmp** is directory inside container. You have to copy the data into the internal directory of the container, because standard creation of the container from **mysql** image does not know how to work with volumes.

Now you need to create the DbContext and model. The steps depend on the type of selected database. I have described Below how to create data contexts for msSql, mySql, postgreSql and sqLite databases. You don't need to repeat all the processes: choose one of them. I've put the **gb\_02\_xxx** branches into the [GIT](https://github.com/gbukauskas/BlazorCodeDemo). The archive contains code samples with the corresponding data contexts. You can download the appropriate branch after failing to make the context yourself.

### Backup and restore

Restoring the original data is very important when running tests, so I will describe it for each database separately.

#### SQLite

This database stores all information in a single file: ~\Sqlite\NorthWind DB\Northwind.db. It is enough to make a copy of this file

cd <path\_to\_Northwind.db>

copy Northwind.db Northwind.db.orig

and later restore the Northwind.db file from this copy:

cd <path\_to\_Northwind.db>

copy Northwind.db.orig Northwind.db

#### MS SQL Server

Backup database:

1. Open Microsoft SQL Server Management Studio (SSMS),
2. Right click on Northwind database in Object Explorer and select Tasks\Export Data-tier Application.. .
3. In the next dialog, specify the file name and the directory where you will save the copy. Don’t write extension in the name: the tool will append **bacpac** extension itself.
4. Click “Finish” button in “Summary” tab and “Close” button in “Results” tab.

Use the same SSMS tool for restoring:

1. Right click on “Databases” folder and select “Import Data-tier Application … .
2. Click “Next” button inside “Introduction” tab.
3. Inside “Import Settings” tab browse to your \*.bacpac file and click “Open”. Click on “Text” button.
4. Click on “Next” button inside “Database Settings” tab. You can to restore data under another name if you wish.
5. Click “Finish” button on “Summary” tab.
6. Click on “Close” inside “Results” tab.

#### PostgreSQL

Use [pg\_dump](https://www.postgresql.org/docs/8.1/backup.html) utility for making backup file:

D:\>cd D:\darbas\BlazorCodeDemo\NorthWind

pg\_dump NorthWind > Northwind.postgreSQL.backup

Use the psql utility for restoring the database. The utility does not create database. You must create it yourself from template0 before restoring:

>psql

Password for user xxxx:

...

xxx=#createdb -T template0 NortWind\_1

\q

>psql NortWind\_1 < Northwind.postgreSQL.backup

Password for user xxx:

...

>

#### MySQL

Use [mysqldump](https://dev.mysql.com/doc/refman/8.0/en/mysqldump.html) utility for making backup file:

cd D:\darbas\BlazorCodeDemo\NorthWind

mysqldump -uroot -p northwind > Northwind.mySQL.backup.sql

The utility **mysql** does not create database. You must create it yourself:

cd D:\darbas\BlazorCodeDemo\NorthWind

mysql -uroot -p

...

mysql> create database NorthWind\_1;

mysql> use NorthWind\_1;

mysql> source db\_backup.dump;

mysql> \q

### Create DBContext

While writing the tests, I noticed that the **dbContext** for MS SQL Server does not specify the **primary key** in the tables. After creating the dbContext, download the file https://github.com/gbukauskas/BlazorCodeDemo/blob/gb\_03\_mssql/BlazorApp\_1/DataContext/NorthwindContext.cs

Compare this file with your variant and make corresponding corrections.

#### SQLite

Install two packages:

* Microsoft.EntityFrameworkCore.Sqlite, Microsoft.EntityFrameworkCore.Tools,
* Create folder Database inside server’s project and copy there the database (**Northwind.db3**).
* Create folder **DataContext\Models** inside server’s project,
* Open Package Manager Console and type the command:

PM> Scaffold-DbContext "Data Source=Database\Northwind.db3"

Microsoft.EntityFrameworkCore.Sqlite -OutputDir DataContext\Models

Write this command in a one row and click **Enter**.

The command will write NorthwindContext.cs and classes for every entity into the  **DataContext\Models** folder**. You can** move the NorthwindContext.cs file into a parent ( DataContext**)** folder**.** Read details of the process on site [Get Started With Entity Framework Core Using SQLite](https://www.c-sharpcorner.com/article/get-started-with-entity-framework-core-using-sqlite/).

#### MS SQL Server

Now let's try a simpler way:

* Install EF Core Power Tools via the **Extensions/Manage Extensions** menu inside Visual Studio,
* Right click a project in Solution Explorer, select **EF Core Power Tools/Reverse Engineer** or press Ctrl+Shift+A, select the Data folder, and select the **EF Core Database First Wizard**,
* Connect to your existing database via the Add button (or pick an existing connection from the drop down list),
* Choose database objects, you can use the top checkbox to select all objects,
* Click OK, and C# DbContext and entity classes will be generated in the current project.

Read [Reverse Engineering Quick Start](https://github.com/ErikEJ/EFCorePowerTools/wiki/Reverse-Engineering-Quick-Start) if something goes wrong.

#### PostgreSQL

Execute listed below steps:

* Create folder **DataContext\Models** inside server’s project,
* Install packages

Npgsql.EntityFrameworkCore.PostgreSQL,

Microsoft.EntityFrameworkCore.Design.

Use NuGet for this task.

* Ensure tat dotnet ef tool is installed on your computer. Type this command inside command prompt:

dotnet ef –help

Install the tool package if it was not found:

dotnet tool install --global dotnet-ef

* Check and resolve version conflicts inside dependencies of your server project.
* Open terminal and switch to the root directory of your server project.
* Type this command in one row:

dotnet ef dbcontext scaffold "Host=localhost;Database=NorthWind;Username=xxxx;Password=yyyy" Npgsql.EntityFrameworkCore.PostgreSQL -o DataContext\Models

#### MySQL

Current Blazor version on Windows 11 does not work with scaffolding. Follow this instruction:

1. Create new console application using commands:

mkdir mySQLef\northwindDB

cd mySQLef

dotnet new console --output .

**MySQLef** is my directory for the new application. You can use any other name.

1. Add the MySQL NuGet package for EF Core using the CLI:

dotnet add package MySql.EntityFrameworkCore

dotnet add package Microsoft.EntityFrameworkCore.Tools

dotnet add package MySql.Data

dotnet restore

1. Build objects for the **northwind** database:

dotnet ef dbcontext scaffold

"server=localhost;port=3306;user=root;password=xxx;database=northwind"

MySql.EntityFrameworkCore -o ./northwindDB -f

Write this command in **one** row.

1. Copy content of **mySQLef\northwindDB** directory into the **DataContext\Models** (inside your serer-side project.

## DbContextFactory

To register a DbContextFactory is the recommended way to use DbContext's from Blazor Server applications. When calling the **AddDbContext<DbContext>()** method the DbContext would be registerd with a scoped lifetime. This can lead to concurrency issues because server-side application keeps alive the connections. Resolve this issue registering **DbContextFactory** instead of **DbContext**. Add these rows into **Main** function (**program.cs** file):

builder.Services.AddDbContextFactory<NorthwindContext>(options =>

options.UseSqlServer(connectionString));

Services would inject DbContextFactory and follow this template:

public class EmployeeService

{

**private readonly IDbContextFactory<NorthwindContext> factory;**

public EmployeeService(**IDbContextFactory<NorthwindContext> factory**)

{

this.**factory** = factory;

}

public async Task<Employee[]> GetEmployeesAsync()

{

using (var context = **factory**.CreateDbContext()) {

return await context.Games.ToArrayAsync();

}

}

}

## IQueryable interface

Entity Framework Core uses Language-Integrated Query (LINQ) to query data from the database. LINQ allows you to use C# (or your .NET language of choice) to write strongly typed queries. It uses your derived context and entity classes to reference database objects. EF Core passes a representation of the LINQ query to the database provider. Database providers in turn translate it to database-specific query language (for example, SQL for a relational database).

LINQ is able to query different data sources (e.g. databases, lists, ...) where the information does not reside in local memory. In fact, it may even exist on an entirely different machine. To support this very different requirement, LINQ introduces the following new concepts:

* Query provider (**IQueryProvider**) - This is specialized software that can interpret a query, so that it efficiently utilizes the underlying resources.
* Expression tree (**Expression**) - A tree is formed from elements of a query. This tree is later analyzed by the query provider. This should be familiar to anyone who has ever written a parser or compiler.
* Queryable sequence ([IQueryable](https://www.codeproject.com/Articles/1240553/LINQ-Part-An-Introduction-to-IQueryable)) - This is the approximate equivalent to **IEnumerable** in LINQ to Objects. It simply pairs a query provider with an expression tree.
* As with IEnumerable, LINQ provides a set of standard methods, defined in the [System.Linq.Queryable](https://learn.microsoft.com/en-us/dotnet/api/system.linq.queryable?view=net-8.0" \l "definition) class. These methods all extend **IQueryable**. They all share identical names and near identical syntax with their counterparts in **System.Linq.Enumerable**.

**DbSet** and **IDbSet** implement **IQueryable** and so can be used as the starting point for writing a LINQ query against the database. It is important to know that the transaction itself is deferred and you can always correct the request. Note that **DbSet** and **IDbSet** always create queries against the database and will always involve a round trip to the database even if the entities returned already exist in the context. A query is executed against the database when:

* It is enumerated by a foreach (C#) or For Each (Visual Basic) statement.
* It is enumerated by a collection operation such as ToArray, ToDictionary, or ToList.
* LINQ operators such as First or Any are specified in the outermost part of the query.
* The following methods are called: the Load extension method on a DbSet, DbEntityEntry.Reload, and Database.ExecuteSqlCommand.

When results are returned from the database, objects that do not exist in the context are attached to the context. If an object is already in the context, the existing object is returned (the current and original values of the object's properties in the entry are not overwritten with database values).

When you perform a query, entities that have been added to the context but have not yet been saved to the database **are not returned** as part of the result set.

## Services

We will move the NorthwindContext.cs class from the DataContext\Models folder one level up: to the DataContext folder. Only models will remain in the DataContext\Models directory. Next we will create two directories: **~\BlazorApp\_1\Services** and **~\BlazorApp\_1\Services\Interfaces**. We will create a generic interface in the second directory (see the INorthWind.cs file):

public interface INorthWind<T, K> where T : class

where K : IComparable

{

Task<T> CreateEntity(NorthwindContext ctx, T newEntity);

Task<Ienumerable<T>>

CreateEntities(NorthwindContext ctx, IEnumerable<T> collection);

Task<T> UpdateEntity(NorthwindContext ctx, T entity);

IQueryable<T> GetAllEntities(NorthwindContext ctx);

Task<T> GetEntityByIdAsync(NorthwindContext ctx, K id);

Task<K> DeleteEntityByIdAsync(NorthwindContext ctx, K id);

}

Note that the **GetAllEntities** method is not asynchronous and returns an **IQueryable<T>**. This means that it only creates the **Expression**, the actual reading of the data will be done by the calling function. This template will allow you to use all the functions defined inside the [IQueryable<T>](https://learn.microsoft.com/en-us/dotnet/api/system.linq.iqueryable-1?view=net-8.0) interface.

NorthWind database uses different types in the primary keys

Create your first service in the directory ~\BlazorApp\_1\Services (CustomerService.cs):

namespace BlazorApp\_1.Services

{

public class CustomerService : INorthWind<Customer>

{

public Task<Customer> CreateEntity(NorthwindContext ctx, Customer newEntity)

{

throw new NotImplementedException();

}

Task<Ienumerable<T>>

CreateEntities(NorthwindContext ctx, IEnumerable<T> collection);

{

throw new NotImplementedException();

}

Task<T> UpdateEntity(NorthwindContext ctx, T entity);

{

throw new NotImplementedException();

}

public IQueryable<Customer> GetAllEntities(NorthwindContext ctx)

{

throw new NotImplementedException();

}

Task<T> GetEntityByIdAsync(NorthwindContext ctx, K id);

{

throw new NotImplementedException();

}

Task<K> DeleteEntityByIdAsync(NorthwindContext ctx, K id);

{

throw new NotImplementedException();

}

}

}

This file was created by Visual Studio. Now we switch on creating tests without changing its content. We will implement the methods by writing tests.

### Paging

When viewing large tables, it is necessary to divide their content into pages. For this purpose, we will create a new interface in the directory ~\BlazorApp\_1\Services\Interfaces:

namespace BlazorApp\_1.Services.Interfaces

{

public interface **IPagedCollection<T>** where T : class

{

public Task<PageResponse<T>>

**GetPageAsync**(IQueryable<T> collection, int pageSize, int pageNumber);

}

}

We will describe the new class **PageResponse** inside the folder ~\BlazorApp\_1\DataContext\RequestResponse:

namespace BlazorApp\_1.DataContext.RequestResponse

{

public class PageResponse<T> where T : class

{

public int TotalRecords { get; set; }

public int TotalPages { get; set; } = 0;

public int PageSize { get; set; }

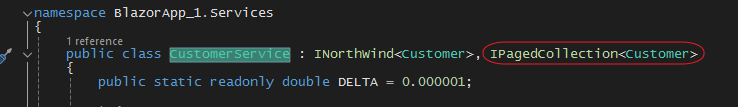
public int PageNumber { get; set; }

public IEnumerable<T>? Items { get; set; }

}

}

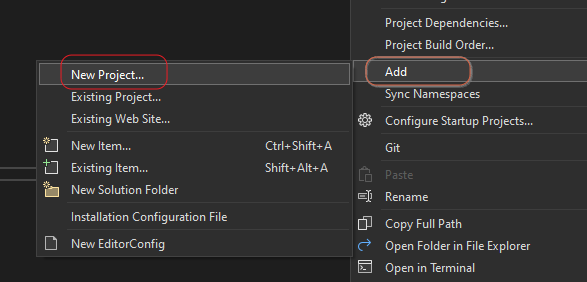
We will supplement CustomerService with this new interface:

The real implementation of this interface is quite easy but long. You can copy the code from my GIT [archive](https://github.com/gbukauskas/BlazorCodeDemo/blob/gb_03_mssql/BlazorApp_1/Services/CustomerService.cs).

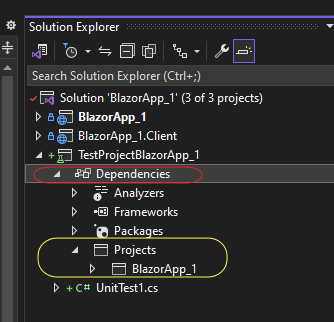
Download the UnitTest1.cs from my [archive](https://github.com/gbukauskas/BlazorCodeDemo/blob/gb_03_mssql/TestProjectBlazorApp_1/UnitTest1.cs). You will find tests for **CustomerService** here.

## Testing

Right click on solution inside “Solution Explorer” window and select “Add/New Project…” :



Select the **xUnit Test Project** template. Name it **TestProjectBlazorApp\_1**. Right click on „Dependencies“ in the project and add reference to the BlazorApp\_1:



It is necessary to create a dbContext for testing a database service. For that purpose you need to get the connectionString. Authors of WEB articles about testing often write it simply as a constant of the string type, but I followed the recommendation of the [stackoverflow](https://stackoverflow.com/questions/43010660/xunit-test-project-connection-string) site and read it from the **config** file. I added the **appsettings.json** file and the static class **DbOptionsFactory** to the TestProjectBlazorApp\_1 project.

File appsettings.json:

{

"ConnectionStrings": {

"DefaultConnection": "Server=localhost;Database=Northwind;Trusted\_Connection=True;MultipleActiveResultSets=true;Encrypt=False;"

}

}

File DbOptionsFactory.cs:

namespace TestProjectBlazorApp\_1.Tools

{

public static class DbOptionsFactory

{

static DbOptionsFactory()

{

var config = new ConfigurationBuilder()

.AddJsonFile("appsettings.json")

.Build();

var connectionString = config["ConnectionStrings:DefaultConnection"];

DbContextOptions = new DbContextOptionsBuilder<NorthwindContext>()

.UseSqlServer(connectionString)

.Options;

}

public static DbContextOptions<NorthwindContext> DbContextOptions { get; }

}

}

Our first test looks as follows:

namespace TestProjectBlazorApp\_1

{

public class UnitTest1

{

private INorthWind<Customer> CustomerSvc;

public UnitTest1()

{

CustomerSvc = new CustomerService();

}

[Fact]

public void Test1()

{

using (var context =

new NorthwindContext(DbOptionsFactory.DbContextOptions))

{

var allCustomers = CustomerSvc.GetAllEntities(context);

}

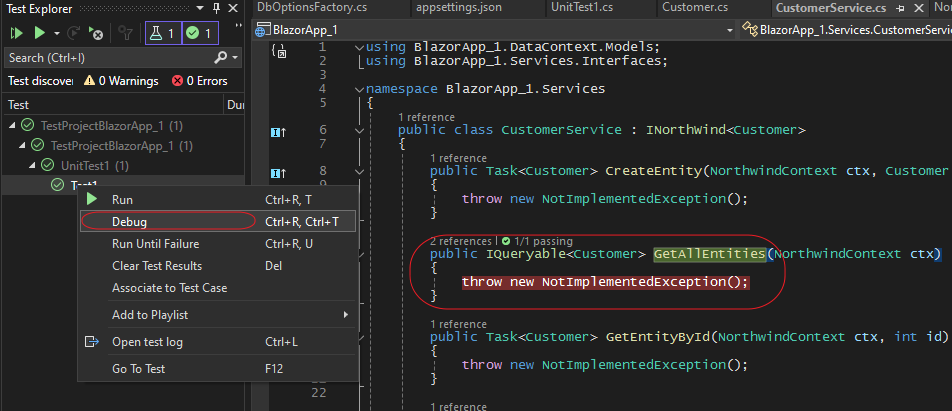
}

}

}

Interface **INorthWind** and class **CustomerService** are defined inside server-side project BlazorApp\_1 (see topics **Services**).

Build the solution, open Text Explorer window an run your first test. The test will faill because function **GetAllEntities(NorthwindContext ctx)** was not implemented. You just need to set a breakpoint in the GetAllEntities function and make sure the function is called.



Correct files CustomerService.cs , UnitTest1.cs and the first test will be ready. Run it using Test Explorer window:

// CustomerService.cs

public IQueryable<Customer> GetAllEntities(NorthwindContext ctx)

{

**return ctx.Customers**;

}

...

// UnitTest1.cs

[Fact]

public void Test1()

{

using (var context = new NorthwindContext(DbOptionsFactory.DbContextOptions))

{

**int allCustomers = CustomerSvc.GetAllEntities(context).Count();**

**Assert.Equal(91, allCustomers);**

}

}

Read about other output checks on the [xUnit](https://hamidmosalla.com/2020/01/12/xunit-part-2-value-and-type-based-assertions-in-xunit/) site. Compare your version with my code on [GitHub](https://github.com/gbukauskas/BlazorCodeDemo/tree/gb_03_mssql) if something goes wrong. At this point, you should only be interested in the **GetAllEntities** method inside the **CustomerService.cs** file.

You should also examine the rest of the tests in the files CustomerService.cs and UnitTest1.cs. There I demonstrated how to use transactions and bulk operations when working with collections. You should also read the articles about [adding new records](https://www.tektutorialshub.com/entity-framework/add-record-add-multiple-records-entity-framework/) into the database and [work with collections](https://learn.microsoft.com/en-us/ef/core/saving/execute-insert-update-delete).

Realization of **CustomerService** and **unit test** is not very difficult, you can realize them yourself. Download the **gb\_03\_mssql** branch from [my archive](https://github.com/gbukauskas/BlazorCodeDemo/tree/gb_03_mssql) and compare files **CustomerService.cs**, **UnitTest1.cs** with your version.

// https://www.tektutorialshub.com/entity-framework/add-record-add-multiple-records-entity-framework/

// https://learn.microsoft.com/en-us/ef/core/saving/execute-insert-update-delete

// https://stackoverflow.com/questions/43010660/xunit-test-project-connection-string

// [XUnit – Part 2: Value and Type Based Assertions in xUnit - Hamid Mosalla](https://hamidmosalla.com/2020/01/12/xunit-part-2-value-and-type-based-assertions-in-xunit/)

// [Home > xUnit.net](https://xunit.net/" \l "documentation)

// [Shared Context between Tests > xUnit.net](https://xunit.net/docs/shared-context)

// [Unit testing C# code in .NET using dotnet test and xUnit - .NET | Microsoft Learn](https://learn.microsoft.com/en-us/dotnet/core/testing/unit-testing-with-dotnet-test)

## Components

**Blazor** apps are built using Razor components, informally known as Blazor components. A component is a self-contained portion of user interface (UI) with processing logic to enable dynamic behavior. Components can be nested, reused, shared among projects. Components render into an in-memory representation of the browser's Document Object Model (**DOM**) called a render tree, which is used to update the UI in a flexible and efficient way.

Components are implemented using a combination of C# and HTML markup in Razor component files with the **.razor** file extension.

Here is an example of Counter component generated by Visual Studio:

@page "/counter"

@rendermode InteractiveAuto

<PageTitle>Counter</PageTitle>

<h1>Counter</h1>

<p role="status">Current count: @currentCount</p>

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

@code {

private int currentCount = 0;

private void IncrementCount()

{

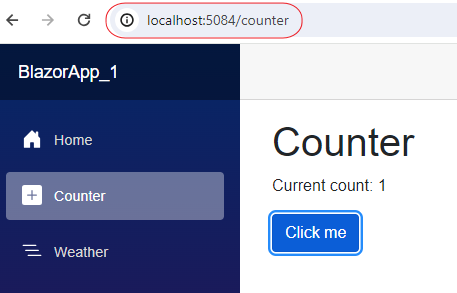
currentCount++;

}

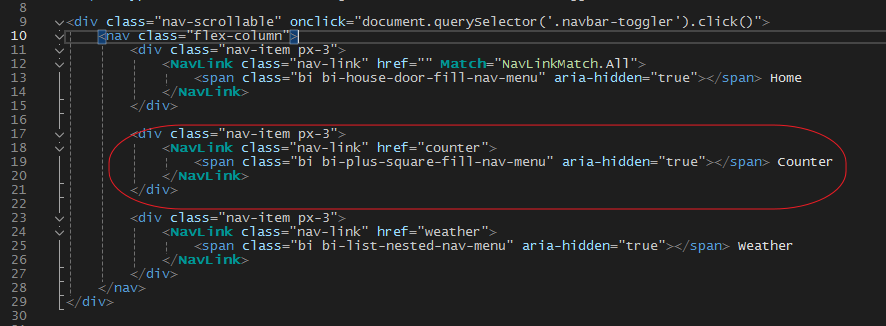
}

A standard component consists of three blocks: directives, HTML and C# code. An example demonstrates these blocks in different colors.

This component has an **@page** directive followed by a route template, which makes it a navigable page. A navigation page can be called by typing its address in the browser's address bar:



or by specifying its address in another component. Look at lines 17-21 of the NavMenu.razor component:



The component can also be called by writing a Custom HTML element. Look at the line 18 inside an example:

<**NavLink** class="nav-link" href="counter">

It calls component NavLink. This rule restricts component names: the name must start with an uppercase letter and cannot contain characters that are not accepted by the HTML standard. Since components are mapped into the Custom HTML Elements, make their name consist of at least two words and start each word with a capital letter. NavLink will be mapped into <nav-link> element.

During compilation, the Blazor system produces partial C# classes from the components. Current version of Blazor does not save auto-generated files to disk. If you wish to re-enable this feature then add the following code to your \*.csproj file:

<PropertyGroup>

**<EmitCompilerGeneratedFiles>true</EmitCompilerGeneratedFiles>**

</PropertyGroup>

By default, **ComponentBase** is the base class for components described by Razor component files. ComponentBase implements the lowest abstraction of components, the **IComponent** interface. **ComponentBase** defines component properties and methods for basic functionality, for example, to process a set of built-in component lifecycle events. [ComponentBase](https://github.com/dotnet/aspnetcore/blob/main/src/Components/Components/src/ComponentBase.cs) is published on WEB, you can use this code but keep in mind that the internal implementations of component features are subject to change at any time without notice.

Components can be of generic type, i.e. they can have a type parameter:

**@typeparam** **TValue**

<h2>Show Value: @TheValue.ToString() @TypeDescr</h2>

@code

{

[Parameter]

public **TValue** TheValue { get; set; }

protected string TypeDescr;

protected override Task OnInitializedAsync()

{

TypeDescr = $"(type is {typeof(TValue).Name})";

return Task.CompletedTask;

}

}

You must specify the type parameter **in both files** if code is written in the separate file:

// ---------- ShowValue.razor ----------

**@typeparam TValue**

<h2>Show Value: @TheValue.ToString() @TypeDescr</h2>

// ---------- ShowValue.razor.cs ----------

using Microsoft.AspNetCore.Components;

using System.Threading.Tasks;

namespace TestGenericComponent.Components

{

public partial class **ShowValue<TValue>** : ComponentBase

{

[Parameter]

public TValue TheValue { get; set; }

protected string TypeDescr;

protected override Task OnInitializedAsync()

{

TypeDescr = $"(type is {typeof(TValue).Name})";

return Task.CompletedTask;

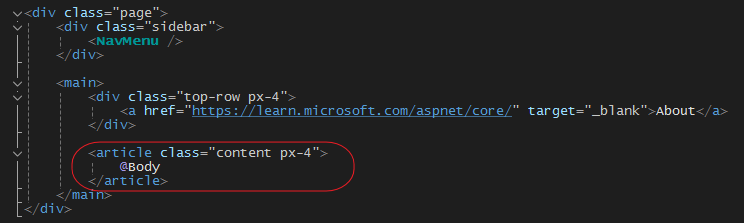
}

}

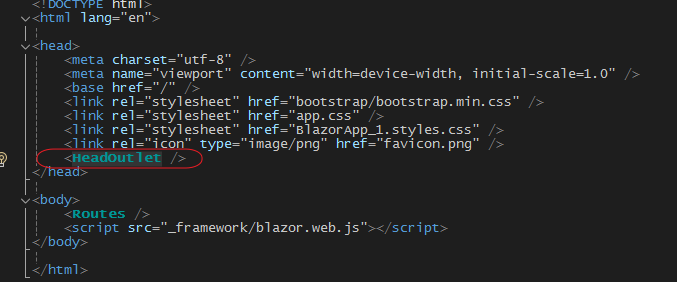
}

### HTML Sections

Components having **@page** are compiled into HTML fragments and Razor engine embeds them into the place marked by **@body** (see file MainLayout.razor):

Information for these tools can be provided in the head section of the **App.razor** file, but this section is identical for all files in the project. A component can change this content using JavaScript.

The current version of Blazor introduced the **HeadOutlet** component in the **App.razor** file.



The **HeadOutlet** component analyzes content of the loading page and catches the **PageTitle** and **HeadContent** components in it. The content of captured components is moved to the location indicated by the **HeadOutlet** component. Each page can have its own information in the HTML header.

In .NET 8, ASP.NET Core introduced **sections**, a new feature that allows the control of content from child components. Sections can be used both in layouts and across nested parent-child components. Sections are like **ContentPlaceHolder** in Web Forms or **RenderSection** in ASP.NET MVC. Blazor defines two new components for this task: **SectionOutlet** and **SectionContent**. To bring these objects into scope, add the following statement to the application’s \_Imports.razor file:

@using Microsoft.AspNetCore.Components.Sections

**SectionOutlet** is a component that renders content provided by **SectionContent** components with matching **SectionName** or **SectionId** arguments. Two or more **SectionOutlet** components can’t have the same **SectionName** or **SectionId**.

**SectionContent** is a component that provides content as a RenderFragment to SectionOutlet components with a matching SectionName or SectionId.

If several SectionContent components have the same SectionName or SectionId, the matching SectionOutlet component renders the content of the **last rendered SectionContent**.

A section is created using a SectionOutlet and identified by either the section SectionName or SectionId parameter. The SectionName parameter is a string that is used to target the SectionOutlet from a SectionContent component. Although the argument passed to SectionName can use any type of casing, it is common to use kebab casing (for example, top-section) which is the casing choice for HTML element IDs.

For example, look at the lines 9-12 inside MainLayout.razor:

<div class="top-row px-4">

<a href="https://learn.microsoft.com/aspnet/core/" target="\_blank">About</a>

</div>

We can to improve this code adding section:

<div class="top-row px-4">

<**SectionOutlet SectionName="before-top-row"**/>

<a href="https://learn.microsoft.com/aspnet/core/" target="\_blank">About</a>

</div>

With the **SectionOutlet** added, any child component using the MainLayout can display content in before-top-row. It’s now possible to insert content before the About link, to demonstrate a dedicated help link for each page.

<**SectionContent SectionName="before-top-row"**>

<a href="/showdata-help" target="\_blank">ShowData Help</a>

</SectionContent>

One of the difficulties with this approach is discoverability. There is no way to know about the ContentSection without directly looking at the source code of MainLayout. For a better developer experience (DX), we can use the ContentId parameter instead of ContentName. The ContentId parameter accepts an object instead of a string. The name becomes strongly typed and recognized by the compiler. ContentId parameter, a static SectionOutlet object is declared in code. The SectionOutlet object is then assigned to the SectionOutlet component’s ContentId parameter.

<div class="top-row px-4">

**<SectionOutlet SectionId="BeforeTopRow" />**

<a href="https://learn.microsoft.com/aspnet/core/" target="\_blank">About</a>

</div>

...

@code{

/// <summary>

/// A section of the layout displayed at the top of the main layout.

/// </summary>

**public static SectionOutlet BeforeTopRow = new();**

}

Read the article on the [telerik](https://www.telerik.com/blogs/aspnet-8-blazor-sections-fully-explained) site. It provides many useful examples of the sections.

### HTML block

Components use Razor syntax. Razor is a markup syntax for embedding .NET based code into webpages. The Razor syntax consists of Razor markup, C#, and HTML. Files containing Razor generally have a .cshtml or .razor file extension. HTML markup in .cshtml Razor files is rendered by the server unchanged.

Razor supports C# and uses the @ symbol to transition from HTML to C#. A full description of Razor is available at [Microsoft's](https://learn.microsoft.com/en-us/aspnet/core/mvc/views/razor?view=aspnetcore-8.0) link, but this link deals with **ASP.NET Core** environment, the **Blazor** environment uses symbol "@" a bit [differently](https://blazor-university.com/components/literals-expressions-and-directives/).

### Parameters

Component parameters pass data to components and are defined using public C# properties on the component class with the **[Parameter]** attribute:

@code {

...

**[Parameter]**

public string **Name** { get; set; }

**[Parameter]**

public int **Age** { get; set; }

**[Parameter]**

public string **Address** { get; set; }

...

}

Parameter values can be used in the UI part. For this we use one-way binding (we simply write the name of the parameter with an **@** sign in front of it):

...

<h1>Person Information</h1>

<p>Name: **@Name**</p>

<p>Age: **@Age**</p>

<p>Address: **@Address**</p>

…

Component parameters can have initial values. User defined reference types can also be used:

// PanelBody.cs

namespace BlazorSample;

public class PanelBody

{

public string? Text { get; set; }

public string? Style { get; set; }

}

...

<!-- ParameterChild.razor -->

<div class="card w-25" style="margin-bottom:15px">

<div class="card-header font-weight-bold">**@Title**</div>

<div class="card-body" style="font-style:**@Body.Style**">

**@Body.Text**

</div>

</div>

...

@code {

**[Parameter]**

public string **Title** { get; set; } = "Set By Child";

**[Parameter]**

public **PanelBody** **Body** { get; set; } =

new()

{

Text = "Set by child.",

Style = "normal"

};

}

Apply the **[EditorRequired]** attribute to specify a required component parameter. If a parameter value isn't provided, editors or build tools may display warnings to the user. This attribute is only valid on properties also marked with the [Parameter] attribute. The EditorRequiredAttribute is enforced at design-time and when the app is built. The attribute isn't enforced at runtime, and it doesn't guarantee a non-null parameter value.

[Parameter]

**[EditorRequired]**

public string? Title { get; set; }

Values to the parameters can be assigned using **routing** or specifying their values in the **attributes** of the custom element.

#### Routing

Razor components that act as navigable endpoints or pages in a Blazor application must have either

* an **@page** directive followed by a route template

@\* Weather.razor \*@

**@page "/weather"**

@attribute [StreamRendering]

<PageTitle>Weather</PageTitle>

...

* an **@attribute** directive followed by a RouteAttribute that takes a constant representing a route template

**@attribute [Route(Constants.AboutRoute)]**

<h1>About Us</h1>

Constants.AboutRoute is defined as a constant value elsewhere within the application:

public static class Constants

{

public const string AboutRoute = "/about";

}

##### Enhanced navigation

Blazor Web Apps are capable of two types of routing for [page navigation and form](https://learn.microsoft.com/en-us/aspnet/core/blazor/fundamentals/routing?view=aspnetcore-8.0" \l "enhanced-navigation-and-form-handling) handling requests:

* **Normal navigation** (cross-document navigation): a full-page reload is triggered for the request URL.
* **Enhanced navigation** (same-document navigation): Blazor intercepts the request and performs a **fetch** request instead. Blazor then patches the response content into the page's DOM. Blazor's enhanced navigation and form handling avoid the need for a full-page reload and preserves more of the page state, so pages load faster, usually without losing the user's scroll position on the page.

Enhanced navigation is available when:

* The Blazor Web App script (**blazor.web.js**, see link in the Pages/App.razor file) is used, not the Blazor Server script (blazor.server.js) or Blazor WebAssembly script (blazor.webassembly.js).
* The feature isn't [explicitly disabled](https://learn.microsoft.com/en-us/aspnet/core/blazor/fundamentals/startup?view=aspnetcore-8.0" \l "disable-enhanced-navigation-and-form-handling).
* The destination URL is within the internal base URI space (the app's base path).

If server-side routing and enhanced navigation are enabled, location changing handlers are only invoked for programmatic navigation initiated from an **interactive runtime**. In future releases, additional types of navigation, such as following a link, may also invoke location changing handlers.

When an enhanced navigation occurs, **LocationChanged** event handlers registered with **Interactive Server** and **WebAssembly** runtimes are typically invoked. There are cases when location changing handlers might not intercept an enhanced navigation. For example, the user might switch to another page before an interactive runtime becomes available. Therefore, it's important that app logic not rely on invoking a location changing handler, as there's no guarantee of the handler executing

When calling [NavigateTo](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.components.navigationmanager.navigateto?view=aspnetcore-8.0):

* If forceLoad is false, which is the default:
  + And enhanced navigation is available at the current URL, Blazor's enhanced navigation is activated.
  + Otherwise, Blazor performs a full-page reload for the requested URL.
* If forceLoad is true: Blazor performs a full-page reload for the requested URL, whether enhanced navigation is available or not.

You can refresh the current page by calling NavigationManager.Refresh(bool forceLoad = false), which always performs an enhanced navigation, if available. If enhanced navigation isn't available, Blazor performs a full-page reload.

You can refresh the current page by calling NavigationManager.Refresh.

Navigation.Refresh();

This command performs an enhanced navigation, if available. If enhanced navigation isn't available, Blazor performs a full-page reload.

Navigation.Refresh(true);

This command performsfull-page reload is always performed, even if enhanced navigation is available.

Enhanced navigation is enabled by default, but it can be controlled hierarchically and on a per-link basis using the **data-enhance-nav** HTML attribute:

<a href="redirect" **data-enhance-nav**="false">

GET without enhanced navigation

</a>

...

<ul **data-enhance-nav**="false">

<li>

<a href="redirect">GET without enhanced navigation</a>

</li>

<li>

<a href="redirect-2">GET without enhanced navigation</a>

</li>

</ul>

If the destination is a non-Blazor endpoint, enhanced navigation doesn't apply, and the client-side JavaScript retries as a full page load. To enable enhanced form handling, add the **Enhance** parameter to **EditForm** forms or the **data-enhance** attribute to HTML forms (<**form**>):

<**EditForm** ... **Enhance** ...>

...

</**EditForm**>

...

<**form** ... **data-enhance** ...>

...

</**form**>

Enhanced form handling isn't hierarchical and **doesn't flow to child forms**. Enhanced form posts only work with Blazor endpoints. Posting an enhanced form to non-Blazor endpoint results in an error.

Blazor's enhanced navigation and form handing may undo dynamic changes to the DOM if the updated content isn't part of the server rendering. To preserve the content of an element, use the data-permanent attribute:

<div data-permanent>

...

</div>

Once Blazor has started on the client, you can use the enhancedload event to listen for enhanced page updates. This allows for re-applying changes to the DOM that may have been undone by an enhanced page update.

//JavaScript

Blazor.addEventListener('enhancedload', () => console.log('Enhanced update!'));

Enhanced navigation with [static server-side rendering (static SSR)](https://learn.microsoft.com/en-us/aspnet/core/blazor/components/render-modes?view=aspnetcore-8.0" \l "static-server-side-rendering-static-ssr) requires special attention when loading JavaScript. For more information, see [ASP.NET Core Blazor JavaScript with static server-side rendering (static SSR)](https://learn.microsoft.com/en-us/aspnet/core/blazor/javascript-interoperability/static-server-rendering?view=aspnetcore-8.0).

##### Input components and forms

Visual Studio generates a WEB App project that contains the

@using Microsoft.AspNetCore.Components.Forms

command. This allows you to use WEB forms in the same way as in ASP.MVC projects. Open the file **\_Imports.razor** and include **Microsoft.AspNetCore.Components.Forms** if your project does not support forms.

The Blazor framework supports [forms](https://learn.microsoft.com/en-us/aspnet/core/blazor/forms/?view=aspnetcore-8.0) in Server Side Rendering mode. Interactive Server (SignalR), Interactive WebAssembly, or Auto **are not required**. Blazor forms provide built-in input components:

* Bound to an object or model that can use data annotations
  + HTML forms with the <form> element
  + EditForm components
* Built-in input components

The Microsoft.AspNetCore.Components.Forms namespace provides:

* Classes for managing form elements, state, and validation.
* Access to built-in Input\* components.

Standard HTML forms are supported. Create a form using the normal HTML <form> tag and specify an @onsubmit handler for handling the submitted form request.

@page "/starship-plain-form"

@inject ILogger<StarshipPlainForm> Logger

<**form method="post" @onsubmit="Submit" @formname="starship-plain-form"**>

<**AntiforgeryToken** />

<div>

<label>

Identifier:

<InputText @bind-Value="Model!.Id" />

</label>

</div>

<div>

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

</div>

</**form**>

@code {

[**SupplyParameterFromForm**]

public Starship? Model { get; set; }

protected override void OnInitialized() => Model ??= new();

private void Submit()

{

Logger.LogInformation("Id = {Id}", Model?.Id);

}

public class Starship

{

public string? Id { get; set; }

}

}

The **EditForm** component is Blazor's approach to managing user-input in a way that makes it easy to perform validation against user input. Each **EditForm** component acts as a parent component to any number of **input validation components** and optionally, validation message components. The **EditForm** is dependent on an **EditContext**, an object that holds information about the current state of the data editing process, such as which fields have been modified and the current validation state of the form together with any validation messages. Most often, you will pass a **model** directly to the form. You would only construct an instance of an **EditContext** when you want to access properties, methods, or more likely, events of the EditContext, which you might do if you want to take control over some aspect of validation, for example.

<**EditForm** **Model="Model" OnSubmit="Submit" FormName="Starship1"**>

<div>

<label>

Identifier:

<InputText @bind-Value="Model!.Id" />

</label>

</div>

<div>

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

</div>

</EditForm>

@code {

[**SupplyParameterFromForm**]

public Starship? Model { get; set; }

protected override void OnInitialized() => Model ??= new();

private void Submit()

{

Logger.LogInformation("Id = {Id}", Model?.Id);

}

public class Starship

{

public string? Id { get; set; }

}

}

EditForm component is is described on [Microsoft](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.components.forms.editform?view=aspnetcore-8.0) site. The component has listed below parameters:

* **AdditionalAttributes** - gets or sets a collection of additional attributes that will be applied to the created form element. The property has type **IReadOnlyDictionary<String,Object>**.

<EditForm Model="Model" ... **AdditionalAttributes="FormAttributes**">

...

</EditForm>

...

@code {

public **Dictionary<string, object> FormAttributes** { get; set; } =

new Dictionary<string, object>()

{

{ "required", "required" },

{ "placeholder", "Child Component Placeholder" },

{ "size", "100" },

{ "maxlength", "15" }

};

}

...

* **ChildContent** - specifies the content to be rendered inside this EditForm. This is standard **RenderFragment**. Render fragments are described on [BlazorUniversity](https://blazor-university.com/templating-components-with-renderfragements/) site. Anything written between **EditForm** tags goes into **ChildContent**. You will write here **HTML elements** and **input validation components**:
  + [InputCheckbox](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.components.forms.inputcheckbox?view=aspnetcore-8.0) - an input component for editing Boolean values.
  + [InputDate<TValue>](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.components.forms.inputdate-1?view=aspnetcore-8.0) - an input component for editing date values. The supported types for the date value are: DateOnly, DateOnly?, DateTime, DateTime? . Use the **TValue** parameter to specify the type:

...

<**InputDate** **TValue="DateOnly"** @bind-Value="@**\_endDate**" />

@code {

private DateOnly \_**endDate** = DateOnly.FromDateTime(DateTime.Today);

...

}

The date type can also be specified with InputDateType:

<InputDate Type="**InputDateType.Date**" @bind-Value="@\_endDate" />

Enum **InputDateType** has the following values: **Date**, **DateTimeLocal**, **Month**, **Time**. They are described in the [Blazor documentation](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.components.forms.inputdatetype?view=aspnetcore-8.0).

* + [InputFile](https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.components.forms.inputfile?view=aspnetcore-8.0) - this component wraps the HTML file input element and supplies a Stream for each file's contents. The **InputFile** component renders an HTML <input> element of type file. By default, the user selects single files. Add the **multiple** attribute to permit the user to upload multiple files at once.

<InputFile OnChange="LoadFiles" **multiple** />

...

@code {

private void LoadFiles(InputFileChangeEventArgs e)

{

...

}

}

To read data from a user-selected file, call IBrowserFile.OpenReadStream on the file and read from the returned stream. OpenReadStream enforces a maximum size in bytes of its Stream. Reading one file or multiple files larger than 500 KB results in an exception. The maxAllowedSize parameter of OpenReadStream can be used to specify a larger size if required.

* + bbb
* Aaa
* bbb

##### Input validation components

// [.NET 8 Blazor New Forms Functionality Explained - YouTube](https://www.youtube.com/watch?v=UkSBN1Lsxr0&ab_channel=CodingwithTom)

##### Route parameters

Route parameters are placeholders for values that you want to pass to a specific component via the URL. The place holder is represented in a route template as a token within curly braces: **{ token }**. The token or parameter name must match a public property within the component that is decorated with the **[Parameter]** attribute. URL is case-insensitive, so the following example is completely correct:

@page "/details/**{id}**"

<h1>Details</h1>

@code{

**[Parameter]** public string **Id** { get; set; }

In the current version of Blazor, optional parameters are marked with a question mark at the end of the name:

@page "/counter\_1/{**start?**}"

@rendermode InteractiveServer

<PageTitle>Counter</PageTitle>

<h1>Counter</h1>

<p role="status">Current count: @currentCount</p>

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

@code {

**[Parameter]**

**public string? Start { get; set; } = null;**

private int currentCount = 0;

**protected override void OnParametersSet()**

{

base.OnParametersSet();

**currentCount = String.IsNullOrEmpty(Start) ? -5 : Int32.Parse(Start);**

}

private void IncrementCount()

{

currentCount++;

}

}

Handle the parameters in the **OnParametersSet()** or **OnParametersSetAsync()** function. These two functions are always executed, while OnInitialized() or OnInitializedAsync() may not be executed. Download branch [gb\_04\_mssql](https://github.com/gbukauskas/BlazorCodeDemo/tree/gb_04_mssql) and look at file **Counter\_1.razor** if you met some problems.

The default data type of the parameter’s property is a string. By default, all route data values are **strings**. If you want to work with different types, you must apply a constraint to the parameter in the route template. You do this by adding a colon, followed by the data type that you want to work with:

@page "/details/{**id:int**}"

<h1>Details</h1>

@code{

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

}

Note that the URL pattern **cannot contain spaces**:

@page "/details/{**id: int**}"

is **incorrect** and causes **error message**:

The constraint reference ' int' could not be resolved to a type. Register the constraint type with 'Microsoft.AspNetCore.Routing.RouteOptions.ConstraintMap'.

The following data types can be used in routing parameters:

|  |  |  |
| --- | --- | --- |
| **Constraint** | **Description** | **Example** |
| bool | Boolean value | {isActive:bool} |
| int | 32-bit integer value | {id:int} |
| datetime | DateTime value | {startdate:datetime} |
| decimal | decimal value | {cost:decimal} |
| double | 64-bit floating-point value | {latitude:double} |
| float | 32-bit floating-point value | {x:float} |
| long | 64-bit integer value | {x:long} |
| guid | GUID value | {id:guid} |

A URL template can contain multiple parameters. Place optional parameters at the end of the URL template, otherwise you may get unexpected results.

A component can contain multiple **@page** directives:

@**page** "/details

@**page** "/details/{id:int}"

@**page** "/details/{title}" // string data type

<h1>Details</h1>

@code{

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

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

}

Catch-all route parameter is a name that starts with asterisk symbol (\*). Place int at the end of URL template because it captures all non-processed symbols from the link:

@page "/catch-all/{\*pageRoute}"

<PageTitle>Catch All</PageTitle>

<h1>Catch All Parameters Example</h1>

<p>Add some URI segments to the route and request the page again.</p>

<p>

PageRoute: @PageRoute

</p>

@code {

[Parameter]

public string? PageRoute { get; set; }

}

For the URL **/catch-all/this/is/a/test** with a route template of /catch-all/{\*pageRoute}, the value of **PageRoute** is set to **this/is/a/test**. Slashes and segments of the captured path are decoded.

The simplest way to link to a route within a Blazor component is to use an HTML hyperlink.

<a href="/Counter">This works just fine</a>

Rad about hyperlinks on [w3schools](https://www.w3schools.com/html/html_links.asp) site.

File NavMenu.razor demonstrates NavLink component:

<**NavLink** class="nav-link" href="counter\_1">

<span class="bi bi-list-nested-nav-menu" aria-hidden="true"></span> Counter

</**NavLink**>

Read about this component on [c-sharpcorner](https://www.c-sharpcorner.com/article/understanding-navlink-in-blazor/) site.

##### Query strings

Use HTML Query when the component has two or more parameters. Query strings are essentially an instance or collection of key value pairs encoded into a URL:

<a href="/Counter?initialCount=10&cssStyle=bold">Start value is 10</a>

The structure of HTML Query is

{URL}?{Query\_element}&{Query\_element}...

The structure of each Query\_element is

{name\_of\_the\_parameter}={value}

Use the **[SupplyParameterFromQuery**] attribute to specify that a component parameter comes from the query string. Specify the [SupplyParameterFromQuery(Name = "xxx") )] when name in the query string is different from parameter's name the component:

@page "/search"

<h1>Search Example</h1>

<p>Filter: @Filter</p>

<p>Page: @Page</p>

@if (Stars is not null)

{

<p>Stars:</p>

<ul>

@foreach (var name in Stars)

{

<li>@name</li>

}

</ul>

}

@code {

**[SupplyParameterFromQuery]**

public string? **Filter** { get; set; }

**[SupplyParameterFromQuery]**

public int? **Page** { get; set; }

**[SupplyParameterFromQuery(Name = "star")]**

public string[]? **Stars** { get; set; }

}

URL of /search?filter=scifi%20stars&page=3&star=LeVar%20Burton&star=Gary%20Oldman will be parsed into:

* The Filter property resolves to **scifi stars**.
* The Page property resolves to **3**.
* The Stars array is filled from query parameters named star (Name = "star") and resolves to **LeVar Burton** and **Gary Oldman**.

Component parameters supplied from the query string support the following types:

* **bool, DateTime, decimal, double, float, Guid, int, long, string**.
* Nullable variants of the preceding types.
* Arrays of the preceding types, whether they're nullable or not nullable.

Query elements with the same names are collected into an array, see to the **star** element in the example.

Call **NavigationManager.GetUriWithQueryParameters** to create a URI constructed from Uri with multiple parameters added, updated, or removed. For each value, the framework uses value?.GetType() to determine the runtime type for each query parameter and selects the correct culture-invariant formatting. The framework throws an error for unsupported types.

@inject NavigationManager Navigation

...

**Navigation.GetUriWithQueryParameters**("{URI}", {PARAMETERS})

The {URI} placeholder is the URI with or without a query string.

The {PARAMETERS} placeholder is an IReadOnlyDictionary<string, object>

More information on constructing or using a query string you will find on the [Microsoft](./For%20more%20information%20on%20constructing%20or%20using%20a%20query%20string,%20see%20the%20Microsoft%20site.) site.

### Blazor Render Modes in .NET 8

Blazor v8 allows you to specify the render mode for each component separately. **Render mode** can be specified as a directive at the top of the page:

@page "..."

**@rendermode** InteractiveServer

or as a parameter:

<Dialog **@rendermode**="InteractiveServer" />

The available render modes are listed in the microsoft.AspNetCore.Components.Web.RenderMode class:

public static class RenderMode

{

public static InteractiveServerRenderMode **InteractiveServer** { get; }

public static InteractiveWebAssemblyRenderMode **InteractiveWebAssembly** { get; }

public static InteractiveAutoRenderMode **InteractiveAuto** { get; }

}

Component rendering modes in .NET 8 are as follows :

* **Static server-side** (SSR); this is default render mode; it has no property in the RenderMode class.
* Interactive server (SSRI),
* Interactive WASM (CLI),
* Auto interactive,
* **Streaming**; It is an option for SSR mode.

The article [Blazor Render Modes](https://dvo.ms/2024/01/23/blazor-render-modes/) explains these different modes.

#### SSR (Static server-side) render mode

The server produces static HTML that doesn't provide for user interactivity or maintaining Razor component state. Static or static rendering is a server-side scenario that means the component is rendered without the capacity for interplay between the user and .NET/C# code. JavaScript and HTML DOM events remain unaffected, but no user events on the client can be processed with .NET running on the server.

Static server-side rendering (static SSR) typically performs a full page refresh whenever the user navigates to a new page or submits a form. In .NET 8, Blazor can enhance page navigation and form handling by intercepting the request and performing a fetch request instead. Blazor then handles the rendered response content by patching it into the browser DOM. Enhanced navigation and form handling avoids the need for a full page refresh and preserves more of the page state, so pages load faster and more smoothly.

The diffing algorithm must find the additions/editions/deletions, and generate the right commands to update the view. By default, it uses the element index to compare elements. While this works great in most cases, it is sometimes not optimal.

The **@key** directive allows instructing Blazor to use a specific key to compare elements instead of using the index. Blazor will compare the existing items with the new ones using the value of the key. This way it will better detect additions/modifications/deletions.

Typically, it makes sense to use **@key** whenever you a rendering a list (e.g., in a @foreach block) and you have a suitable value to define the @key.

You can also use @key as a way of forcing Blazor not to preserve a certain element or component subtree when something changes. For example:

<div **@key="@currentPerson"**>

...some content that depends on @currentPerson...

</div>

In this example, if @currentPerson changes, then the @key will force Blazor to discard the entire <div> and its descendants, and rebuild that subtree within the UI with new elements and components. This can be useful if you need to guarantee that no UI state is preserved when @currentPerson changes. Read about @key on [AspNetCore](https://github.com/dotnet/AspNetCore.Docs/issues/12755) site.

@key directive generates no HTML attributes in the element.

##### List of customers (scrolled list)

Now we will consider the case where all records are displayed. This strategy is perfect for this case, because the dbo.Customers table is small: it has 91 records. Tables with many records require pagination, which we will discuss separately.

Download code from branch <https://github.com/gbukauskas/BlazorCodeDemo/tree/gb_04_mssql_1>. In the code of this branch, the file NavMenu.razor has an additional menu item:

<div class="nav-item px-3">

<NavLink class="nav-link" href="**showcustomers**">

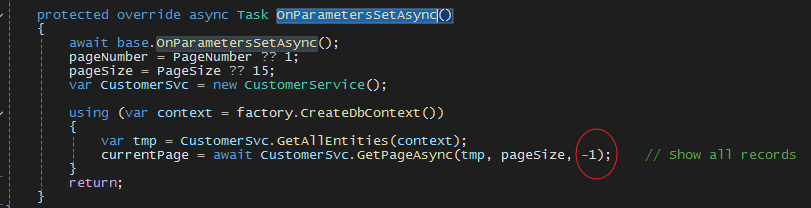
<span class="bi bi-list-nested-nav-menu" aria-hidden="true"></span> All Customers

</NavLink>

</div>

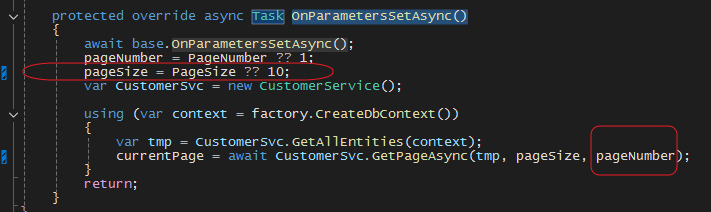
This menu item displays list of customers reading them from **Northwind** database.

The list of customers is displayed by the **ShowCustomers.razor** component. Take a look at the **OnParametersSetAsync** function: it reads all the records from the dbo.Customers table. I changed the **GetPageAsync** function in **CustomerService.cs**. This function returns all records when **pageNumber** is -1. Of course, the **GetAllEntities** function can also be used, but its response is different from that of **GetPageAsync**. This difference would complicate the HTML part of the component. The **ShowCustomers.razor.css** file creates **isolated CSS**. Read about CSS isolation and its advantages on the [Microsoft](https://learn.microsoft.com/en-us/aspnet/core/blazor/components/css-isolation?view=aspnetcore-8.0) website.

The provided CSS does not allow the table header to "escape" during scrolling. Comment out the first two CSS tags and see for yourself how it changes table scrolling. The Chrome browser caches pages, which greatly hinders debugging. Clear your history or use a different browser for testing this fix.

##### List of customers (paged list)

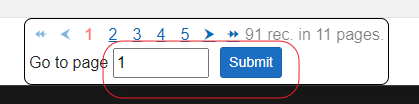
The ShowCustomers.razor file has two parameters (pagenumber and pagesize) that allow you to display the list in pages. Just fix the OnParametersSetAsync() function in this file and the application will show the first page.

Now you need to add component that will allow you to go to the next or previous page. Blazor Bootstrap has [pagination](https://docs.blazorbootstrap.com/components/pagination) component but you can use it inside interactive page only (SSRI or CLI). We need a component where HTML links are used instead of buttons.

The file [ShowCustomers.razor](https://github.com/gbukauskas/BlazorCodeDemo/blob/gb_04_mssql_2/BlazorApp_1/Components/CustomPager.razor) contains the first version of this pager. In this file, the PrepareButtonList function does the main work. It forms a list of links whose length does not exceed the value specified by **MaxButtons**. The list is shifted to the left so that the link to the selected page is visible (**PageNumber** parameter). Here is a screenshot of the component:



This variant will work well for a small number of pages. In the case of a large number of pages, it is necessary to give a opportunity for directly entering a number of the desired page.



Blazor's documentation claims that HTML forms work and don't require a WEB socket connection, but you will meet a surprise working with Static Server Side Rendering pages: the system calls the target function but does not pass it any POST parameters. I believe this is a bug which will be fixed in future versions of the Blazor. The error itself is easily bypassed, you just need to switch into interactive mode: add command **@rendermode InteractiveServer** to the start of the **CustomPager.razor** component:

@namespace BlazorApp\_1

**@rendermode InteractiveServer**

@inject NavigationManager NavManager

@using BlazorApp\_1.DataContext.Models

@using System.Text

…

You can find the corrected version of the application in [my archive](https://github.com/gbukauskas/BlazorCodeDemo/tree/gb_04_mssql_2a). Two files were fixed in GIT branch gb\_04\_mssql\_2a:

* The **ShowCustomers.razor** component received an additional parameter which is used for sending error messages from server.

@page "/showcustomers/{pagenumber:int?}/{pagesize:int?}/{**errormessage?**}"

@attribute [StreamRendering(true)]

...

@code {

...

[Parameter]

**public string ErrorMessage { get; set; } = "";**

...

}

* The CustomPager.razor component has the same additional parameter

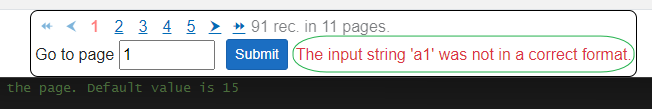
...

[Parameter]

public string ErrorMessage { get; set; } = "";

…

The component displays a content of this parameter in the pager:



// [Blazor in .NET 8: Server-side and Streaming Rendering | Chris Sainty - Building with Blazor](https://chrissainty.com/blazor-in-dotnet-8-server-side-and-streaming-rendering/)

// [Introduction · Bootstrap v5.1 (getbootstrap.com)](https://getbootstrap.com/docs/5.1/getting-started/introduction/)

// https://getbootstrap.com/docs/5.3/getting-started/introduction/

[Bing Videos](https://www.bing.com/videos/riverview/relatedvideo?q=ef+core+add+postgresql&mid=B6BDC4CF7B0083AA5401B6BDC4CF7B0083AA5401&FORM=VIRE)

[Blazor | Что такое Blazor (metanit.com)](https://metanit.com/sharp/blazor/1.1.php)

[Blazor | Управление элементом head и компонент HeadOutlet (metanit.com)](https://metanit.com/sharp/blazor/2.13.php)

[Get Your Head Together: Blazor’s HeadContent and PageTitle (telerik.com)](https://www.telerik.com/blogs/get-your-head-together-blazor-new-headcontent-pagetitle)

[Blazor Render Modes in .NET 8 (dvo.ms)](https://dvo.ms/2024/01/23/blazor-render-modes/) ++

[Blazor — New .NET 8 Render Modes | DevExpress Technical (medium.com)](https://medium.com/devexpress-technical/blazor-new-net-8-render-modes-7be850029174) ++

https://www.youtube.com/watch?v=QZKi81F24H0&ab\_channel=DevExpress

https://www.youtube.com/watch?v=YwZdtLEtROA&ab\_channel=dotnet

https://www.youtube.com/watch?v=EqSgqm4qWjo&ab\_channel=PatrickGod

[Getting Started with Blazor’s New Render Modes in .NET 8 (telerik.com)](https://www.telerik.com/blogs/getting-started-blazor-new-render-modes-net-8?ref=dailydev)

[ASP.NET Core Blazor render modes | Microsoft Learn](https://learn.microsoft.com/en-us/aspnet/core/blazor/components/render-modes?view=aspnetcore-8.0)

[.NET 8: Blazor Render Modes Explained — SitePoint](https://www.sitepoint.com/net-8-blazor-render-modes-explained/)

[Front-end Web Development with .NET for Beginners | Microsoft Learn](https://learn.microsoft.com/en-us/shows/frontend-web-development-with-dotnet-for-beginners/) → video collection

[Exploring Blazor Changes in .NET 8 - Auto Render Mode (jonhilton.net)](https://jonhilton.net/blazor-auto-render-mode/)

https://www.youtube.com/watch?v=lrYBq3nOQPM&ab\_channel=AuthorisedTerritory

[Routing And Navigation In Blazor | Learn Blazor](https://www.learnblazor.com/routing)

[Forms with .NET 8 Blazor SSR (Static Server-Side Rendering) & Enhanced Navigation🔥 (youtube.com)](https://www.youtube.com/watch?v=SwmDg4Rmq3U&ab_channel=PatrickGod)

[https://learn.microsoft.com/en-us/aspnet/core/blazor/components/?view=aspnetcore-8.0#component-parameters](https://learn.microsoft.com/en-us/aspnet/core/blazor/components/?view=aspnetcore-8.0" \l "component-parameters)

https://www.learnblazor.com/routing

[https://learn.microsoft.com/en-us/aspnet/core/blazor/components/?view=aspnetcore-8.0#component-parameters](https://learn.microsoft.com/en-us/aspnet/core/blazor/components/?view=aspnetcore-8.0" \l "component-parameters)

https://learn.microsoft.com/en-us/aspnet/core/blazor/hybrid/routing?view=aspnetcore-8.0&pivots=maui

https://www.telerik.com/blogs/blazor-basics-blazor-routing-navigation-fundamentals

[Exploring Blazor Changes in .NET 8 - Auto Render Mode (jonhilton.net)](https://jonhilton.net/blazor-auto-render-mode/)

[LINQ Part 3: An Introduction to IQueryable - CodeProject](https://www.codeproject.com/Articles/1240553/LINQ-Part-An-Introduction-to-IQueryable)

https://www.w3schools.com/howto/howto\_js\_get\_current\_window.asp