# Building Modern Web Applications Using Blazor ASP.NET Core

Blazor is a modern web framework from Microsoft that was included in .NET 5. It’s used for building interactive web applications using C# and .NET and it’s based on a flexible, modular component model that’s well-suited for building applications with rich, interactive web user interfaces. It should be noted that you can still use JavaScript if you’d like to, i.e., you can invoke your JavaScript functions from C# and vice versa.

This article will take a deep dive into Blazor and its components, and then demonstrate how to build modern web applications using it. It will also discuss the performance and security considerations, deployment using Docker and Kubernetes, and the best practices.

If you’re to work with the code examples discussed in this article, you need the following installed in your system:

* Visual Studio 2022
* .NET 9.0
* ASP.NET 9.0 Runtime

If you don’t already have Visual Studio 2022 installed on your computer, you can download it from here: <https://visualstudio.microsoft.com/downloads/>.

At the end of this journey, you’ll be able to build high-performance, scalable, and secure Blazor applications in ASP.NET Core 9 and Visual Studio 2022.

## Understanding the Problem

While building web applications, you would typically need server- and client-side components. To create the server-side components, you might want to use C#, Java, etc. On the other hand, when building the client-side components, you might typically want to use Angular, React, etc. You need to learn and master two different types of technologies, one for the server side and one for the client side. This makes it difficult to maintain the codebase over time. See **Figure 1** to understand a technology stack of a typical web application that doesn’t use Blazor.



**Figure 1:** The technology stack in a typical web application

Blazor is a web framework that allows web developers to use C# in lieu of JavaScript to create modern web applications with reusable components that can be executed on both the client-side and the server-side to provide optimal web solutions. The Blazor framework is part of the ASP.NET Core framework and helps in streamlining the web development process for .NET and C# developers targeting web applications.

## An Introduction to Blazor

Blazor, an open-source framework from Microsoft, empowers developers to create interactive web applications using C# and .NET. Blazor employs WebAssembly, a web standard that permits browser-based code execution from languages other than JavaScript, creating a fully featured, high-performance client-side development environment using C#. By using .NET, Blazor empowers developers to build full-stack applications encompassing both client and server components. Such flexibility permits developers to choose the hosting model best suited to their application’s needs, including client-side rendering for interactive interfaces and server-side rendering for improved performance and scalability.

Blazor provides a modern, efficient, and versatile approach to web application development. It allows developers to build interactive, high-performing applications using C# and the .NET and .NET Core frameworks. By using its features, tools, and best practices, developers can streamline development, improve application functionality, and create engaging user experiences across diverse platforms.

## Key Features of Blazor

Here’s a quick look at the key features of Blazor.

### Using C# in Lieu of JavaScript Throughout

When using Blazor, developers no longer need to leverage multiple development languages, libraries, and tools when building their applications. Instead, they can use C# throughout all application layers, both at the server side and the client-side, thereby eliminating the need to learn and use JavaScript. Consequently, this reduces the development effort, enables you to use the same language for building your client- and server-side components, thereby promoting code reuse across different platforms.

### Component-Based Architecture

Blazor follows a component-based architecture, facilitating code reusability, separation of concerns, modular design, easy maintenance, and enabling you to build applications in a structured manner. These components take the form of self-contained, reusable units of UI and logic written in razor syntax, which combines HTML, CSS, and C#. These components specify your application's logic and structure, making multilevel web development possible. In this approach, the source code is divided into smaller logical pieces that can be used again in the future while making the applications responsive and interactive.

### Authentication and Authorization

Blazor provides support for robust security and data protection to thwart malicious attackers. It comes with built-in support for authentication and authorization and easy integration with OAuth providers, IdentityServer, and Azure AD. Note that Blazor takes advantage of the ASP.NET security framework to establish a user's identity. In Blazor Server apps, the AuthenticationStateProvider service uses the HttpContext.User API to retrieve authentication state data.

Identity Server is an open-source framework for implementing identity and access control in your .NET applications. It implements OpenID Connect (OIDC) and OAuth 2.0 standards, integrates with ASP.NET Core Identity framework, and provides a common way to authenticate requests in ASP.NET Core applications.

### Integration with the .NET Ecosystem

Blazor integrates nicely with the .NET ecosystem, including ASP.NET Core, Entity Framework Core, etc. You can easily share code between the server and client components, thereby enhancing maintainability and reducing code redundancy. Additionally, you can call your JavaScript functions easily from Blazor and integrate Blazor with JavaScript frameworks and libraries.

### Support for Multiple Hosting Models

Blazor supports both client-side and server-side hosting models, providing flexibility and performance optimization based on project requirements. To achieve this, it supports multiple hosting models, such as the following:

* **Blazor WebAssembly (WASM):** These applications can be executed entirely on the client-side in the browser using WebAssembly.
* **Blazor Server:** These are applications that run on the server and can transmit UI updates to the client components over a SignalR connection.
* **Blazor Hybrid:** These applications combine the best of both worlds, i.e., they blend Blazor with .NET MAUI for building native desktop and mobile apps.

### Platform Independence

Applications built in Blazor are platform independent. For example, you can run Blazor WebAssembly apps in modern browsers without using plug-ins. Moreover, you can execute Blazor Hybrid apps on Windows, macOS, iOS, and Android platforms using .NET MAUI. Additionally, you can host your Blazor Server applications on any platform that provides support for ASP.NET Core.

### Render Modes

Blazor leverages render modes to determine the hosting model to be used, whether the application should be rendered at the server or the client, and whether it’s interactive or non-interactive. Blazor supports the following types of render modes:

* Static server
* Interactive server
* Interactive WebAssembly
* Interactive auto

## How Does Blazor Work?

Blazor is a Microsoft framework that allows executing applications built using C# in a web browser without using any plug-ins. You can use Blazor to build modern-day applications using C# as a full-stack development tool. Applications developed using Blazor run inside the context of your web browser. Once you compile such an application, one or more files are loaded into the browser and executed. Unlike ASP.NET Core, you don't need a specific back-end component to run your Blazor application. Blazor applications can access web services using HTTP REST APIs.

A Blazor application is composed of optional reusable components containing a C#, HTML, and CSS conglomerate that can run both on the server and client sides. These components define the structure and behavior of the user interface. These are .NET C# classes enclosed in .NET assemblies supporting event handling logic and user events that can be reused and also be made available as Razor class libraries or NuGet packages. Components handle user interaction using events such as button clicks, which trigger updates to the state of the components.

### Blazor WebAssembly

Blazor WebAssembly (WASM) is a single-page application framework for building cutting-edge client-side web applications based on .NET that are compatible with all web browsers. With Blazor WASM, the entire application, from the application logic to UI elements, its dependencies, and the .NET Core runtime, is loaded in the web browser. Anytime you launch the web application or any web page, the code responsible for the client-side logic and all its dependencies is also fetched. The source code for .NET WASM and its dependencies, like C# and Razor files, are compiled into .NET assemblies, which are then transferred to your web browser and executed there. It’s preserved in bytecode format for fast download and execution and allows interaction with the browser through JavaScript, using a feature known as JavaScript Interop. **Figure 2** shows the components of a typical Blazor Web Assembly application.



**Figure 2**: The components of a Blazor Web Assembly application

### Blazor Server

In the Blazor server hosting model, components run on the server inside an ASP.NET Core application. Using the SignalR connection and the WebSockets protocol, you manage UI changes or updates, events, and JavaScript calls.

The Blazor application is hosted on the server, and all changes or events triggered on the application's browser are simplified and communicated to the server using SignalR. Although the user interface is rendered to the web browser, the UI updates and event handling are performed on the server side. Although this is analogous to traditional web applications, unlike a traditional web application, the client and the server communicate over a SignalR connection, as shown in **Figure 3**.



**Figure 3:** In a Blazor Server app, the client and the server applications communicate using SignalR

### Blazor Hybrid

The Blazor Hybrid model enables developers to create native applications for mobile or desktop using a hybrid model in which Razor components are executed directly in the .NET process instead of through WebAssembly. This approach doesn’t require web assembly to execute your application. You can take advantage of .NET native technologies, such as MAUI and WPF, together with Razor components to create your Blazor Hybrid applications. **Figure 4** shows the components of a Blazor Hybrid application.



**Figure 4:** Components of a Blazor Hybrid application

## Use Cases

Here are a few use cases for Blazor applications:

* Single-page applications (SPAs)
* Progressive web apps (PWAs)
* Cross-platform desktop applications
* Real-time applications
* Line-of-business (LOB) applications

## What Are Progressive Web Applications?

Progressive Web Applications (PWAs) are web applications that provide users with a rich experience regardless of the platform on which they execute.

The following are the features of PWAs:

* **Progressive enhancement:** PWAs can work on any device or platform, enabling an intuitive user experience on desktops and mobile devices.
* **Responsive design:** PWAs provide an enriched user experience by following responsive web design principles across devices and platforms.
* **Offline support:** PWAs can work offline or in areas with limited connectivity so that you can access the application even when there’s no connectivity.

Here are the key benefits of PWAs:

* Cross-platform compatibility
* Offline Accessibility
* Enhanced performance
* Cost-effective

There are certain downsides to using PWAs:

* Security concerns
* Limited native support
* Limited user engagement
* Limited discoverability
* Limited browser support

In the application you’ll create later in this article, you won’t use PWAs for simplicity and brevity.

### Create a New Blazor Web Assembly Project in .NET 9 and Visual Studio 2022

You can create a project in Visual Studio 2022 in several ways, such as from the Visual Studio 2022 Developer Command Prompt or by launching the Visual Studio 2022 IDE. When you launch Visual Studio 2022, you'll see the Start window. You can choose **Continue without code** to launch the main screen of the Visual Studio 2022 IDE.

Now that you know the basics, let’s start setting up the project. To create a new ASP.NET Core 8 Project in Visual Studio 2022:

1. Start the Visual Studio 2022 IDE.
2. In the **Create a new project** window, select **Blazor Web App** and click Next to move on.
3. Specify the project name as **Blazor\_WebAssembly\_Demo** and the path where it should be created in the **Configure your new project** window.
4. If you want the solution file and project to be created in the same directory, you can optionally check the **Place solution and project in the same directory** checkbox. Click Next to move on.
5. In the next screen, specify the target framework and authentication type as well. Ensure that the **Configure for HTTPS**, and **Do not use top-level statements** checkboxes are unchecked because you won’t use any of these in this example.
6. Next, specify the **Interactive render mode** and **Interactivity location**
7. You should ensure that the **Include sample pages** checkbox is checked if you would like to have sample pages added to your project
8. Click **Create** to complete the process.

A new Blazor Web App project is created. **Figure 5** shows the default solution structure.



**Figure 5:** The solution structure of a Blazor Web Assembly application

When you execute the application, the Home page will be displayed in the web browser, as shown in **Figure 6**.



**Figure 6:** A Blazor Web Assembly application in execution

## Integrating Blazor WebAssembly into an Existing ASP.NET Core Web Application

In this section, you’ll examine how you can integrate a Blazor WebAssembly application with an existing ASP.NET Core Web Application.

First, create an ASP.NET Core 9 application in Visual Studio by following the steps outlined in the next section.

### Create a New ASP.NET Core 9 Project in Visual Studio 2022

You can create a project in Visual Studio 2022 in several ways, such as from the Visual Studio 2022 Developer Command Prompt or by launching the Visual Studio 2022 IDE. When you launch Visual Studio 2022, you'll see the Start window. You can choose **Continue without code** to launch the main screen of the Visual Studio 2022 IDE.

Now that you know the basics, let’s start setting up the project. To create a new ASP.NET Core 8 Project in Visual Studio 2022:

1. Start the Visual Studio 2022 IDE.
2. In the **Create a new project** window, select **ASP.NET Core Web API** and click Next to move on.
3. Specify the project name as **ASPNETCoreBlazor** and the path where it should be created in the **Configure your new project** window.
4. If you want the solution file and project to be created in the same directory, you can optionally check the **Place solution and project in the same directory** checkbox. Click Next to move on.
5. In the next screen, specify the target framework and authentication type as well. Ensure that the **Configure for HTTPS**, **Enable Docker Support**, **Do not use top-level statements**, and the **Enable OpenAPI support** checkboxes are unchecked because you won’t use any of these in this example.
6. Remember to leave the **Use controllers** checkbox checked because you won’t use minimal API in this example.
7. Click Create to complete the process.

A new ASP.NET Core Web API project is created. You’ll use this project to implement the CQRS pattern in ASP.NET Core and C#.

Next, create a new Blazor WebAssembly application in the same solution using the steps outlined in an earlier section and follow the steps given below to integrate your Blazor WebAssembly into the ASP.NET Core Web project.

1. Right-click on the Solution Explorer and select the Add New Project option to create a new project into your solution.
2. Next, search for Blazor WebAssembly and select Blazor WebAssembly App from the search results found.
3. Configure the Blazor WebAssembly App using the **Configure your new project** dialog.
4. In the Program.cs file, comment out this line: **builder.RootComponents.Add<App>("#app");**.
5. To enable your ASP.NET Core Web application to use the Blazor Web Assembly components, add the Blazor application as a project reference using the Reference Manager dialog window.
6. Search for the NuGet project Microsoft.AspNetCore.Components.WebAssembly.Server in the ASP.NET Core Web application and add install it.
7. Add the following two lines in the Program.cs file:

app.UseBlazorFrameworkFiles();

app.MapFallbackToFile("index.html");

1. Add the following script reference as well:

<script src="\_framework/blazor.webassembly.js">

</script>

That’s all you need to do!

In the next section, you’ll create a new Blazor Server application in Visual Studio 2022.

### Create a New Blazor Server Application in .NET 9 and Visual Studio 2022

You can create a project in Visual Studio 2022 in several ways, such as from the Visual Studio 2022 Developer Command Prompt or by launching the Visual Studio 2022 IDE. When you launch Visual Studio 2022, you'll see the Start window. You can choose **Continue without code** to launch the main screen of the Visual Studio 2022 IDE.

Now that you know the basics, let’s start setting up the project. To create a new ASP.NET Core 8 Project in Visual Studio 2022:

1. Start the Visual Studio 2022 IDE.
2. In the **Create a new project** window, select **Blazor Server App** and click Next to move on.
3. Specify the project name as **Blazor\_Server\_Demo** and the path where it should be created in the **Configure your new project** window.
4. In the next screen, specify the target framework and authentication type as well. Ensure that the **Configure for HTTPS** and **Do not use top-level statements** checkboxes are unchecked because you won’t use any of these in this example.
5. If you want the solution file and project to be created in the same directory, you can optionally check the **Place solution and project in the same directory** checkbox. Click Next to move on.
6. Click Create to complete the process.

A new Blazor Server App project is created. **Figure 7** shows the default solution structure of the Blazor Server App you just created.



**Figure 7:** The solution structure of a Blazor Server application

Create a New Blazor Hybrid Application in .NET 9 and Visual Studio 2022

Set up a new Blazor Hybrid Application project in Visual Studio 2022 by following the steps outlined below:

1. Start the Visual Studio 2022 IDE.
2. In the **Create a new project** window, select **.NET MAUI Blazor Hybrid App** and click Next to move on.
3. Specify the project name as **Blazor\_Hybrid\_Demo** and the path where it should be created in the **Configure your new project** window.
4. In the next screen, specify the target framework and authentication type as well. Ensure that the **Configure for HTTPS** and **Do not use top-level statements** checkboxes are unchecked because you won’t use any of these in this example.
5. If you want the solution file and project to be created in the same directory, you can optionally check the **Place solution and project in the same directory** checkbox. Click Next to move on.
6. Click Create to complete the process.

**Figure 8** shows what the solution structure of the Blazor Hybrid Application looks like.



**Figure 8:** The solution structure of a Blazor Hybrid application

You can execute this application in Windows (default) or your Android and iOS devices. To run this application on an Android device, follow these steps:

1. From the menu in your Visual Studio IDE, select Tools > Android > Android Device Manager.
2. When the Android Device Manager is launched, create a new Android device to run your application.
3. Once the Android Device has been created, click Start.
4. If you've got multiple emulators created for your Android device, select your desired emulator and then click F5 to run the application.

**Figure 9** shows the Blazor Hybrid application running in an Android emulator.



**Figure 9:** A Blazor Hybrid application running in an Android emulator

## Implement a Supply Chain Management Application

Microservices architecture is a structural approach that organizes an application as a collection of small independent services modeled around a business domain that can communicate with one another, if need be. In this section, you’ll implement a Supply Chain Management Application using ASP.NET Core and Blazor. To do this, follow these steps:

1. Create the Supply Chain Management System database
2. Build the Solution Structure
3. Create the Product Microservice
	1. Create the Product model
	2. Create the Product Data Context
	3. Seed the Product database table
	4. Create the Product Repository
	5. Create the Product Controller
4. Create the Supplier Microservice
	1. Create the Supplier model
	2. Create the Supplier Data Context
	3. Seed the Supplier database table
	4. Create the Supplier Repository
	5. Create the Supplier Controller
5. Install Entity Framework Core
6. Configure the application
	1. Specify the database connection string
7. Execute the application

### Create the Supply Chain Management System Database

Create a new database called SupplyChainManagementSystem using the following script:

Create database SupplyChainManagementSystem

Next, create the Product, Order, Supplier, and the Shipment database tables inside the SupplyChainManagementSystem database using the script given in **Listing 1**.

**Figure 10** demonstrates the database diagram of the ShoppingCartSystem database.



**Figure 10:** The database design of the SupplyChainManagementSystem database

In this next section, you’ll create a new Blazor Server Application named SupplyChainManagementSystem by following the steps outlined earlier.

### The Solution Structure

As evident from the database design, the SupplyChainManagementSystem application is comprised of the Supplier, Product, Shipment, and Order microservices. Each of these microservices correspond to a module. There are four modules in this application.

For the sake of simplicity, you’ll create only two of them: **Supplier** and **Product**. **Figure 11** shows what the solution structure looks like.



**Figure 11:** The solution structure of our Supply Chain Management System

There are several files and folders in the generated project that you can observe when you look at the solution structure:

* **Pages:** This solution folder contains Razor components or Blazor UI building blocks.
* **wwwroot:** This solution folder contains static files like images, CSS, JavaScript etc.
* **Program.cs:** This file marks the entry point for the application.
* **App.razor:** This file represents the root component that configures routing and layout for your application.
* **\_Imports.razor:** This file represents a Shared Razor imports file for the entire application.

In the sections that follow, you’ll create the related classes and interfaces pertaining to the application you’ll be building here.

## Reorganizing the Project

When you create a standalone Blazor WebAssembly application in .NET 9, you’ll not see any option to host it in ASP.NET Core. Microsoft has removed this option from .NET 8 onward. If you use .NET 7, you’ll be able to see an option to host your project in ASP.NET Core, as shown in **Figure 12.**



**Figure 12:** Creating a standalone Blazor WebAssembly App in DotNet7

When you create a standalone Blazor WebAssembly application in .NET 7, you’ll observe three projects created automatically for you. These are the Server/API project, the Client project, and a Shared project.

You can, however, create these projects in .NET 9 manually as a workaround. In this example, you’ll create the following three projects in the solution:

* SupplyChainManagementSystem.Server
* SupplyChainManagementSystem.Client
* SupplyChainManagementSystem.Shared

The server project is where we usually write the back-end logic (API, services, etc.). The shared project typically contains libraries that are used across both the server and client projects. The client project is where you build your user interface components.

### Install Entity Framework Core

So far, so good. The next step is to install the necessary NuGet Package(s) for working with Entity Framework Core and SQL Server. To install these packages into your project, right-click on the solution and then select **Manage NuGet Packages for Solution….**

Once the window pops up, search for the NuGet packages to add to your project. To do this, type in Microsoft.EntityFrameworkCore, Microsoft.EntityFrameworkCore.Design, Microsoft.EntityFrameworkCore.Tools, and Microsoft.EntityFrameworkCore.SqlServer in the search box and install them one after the other. Alternatively, you can type the commands shown below at the NuGet Package Manager Command Prompt:

PM> Install-Package

Microsoft.EntityFrameworkCore

PM> Install-Package

Microsoft.EntityFrameworkCore.Design

PM> Install-Package

Microsoft.EntityFrameworkCore.Tools

PM> Install-Package

Microsoft.EntityFrameworkCore.SqlServer

You can also install these packages by executing the following commands at the Windows Shell:

dotnet add package

Microsoft.EntityFrameworkCore

dotnet add package

 Microsoft.EntityFrameworkCore.Design

dotnet add package

 Microsoft.EntityFrameworkCore.Tools

dotnet add package

 Microsoft.EntityFrameworkCore.

SqlServer

Install the NuGet packages in both the SupplyChainManagementSystem.Server and SupplyChainManagementSystem.Client (if you need to use EF Core components) projects.

## Create the Product Microservice

In this example, you’ll build the Product microservice application or the Product API. The product microservice application is composed of the following files:

* **Product.cs:** The product model that contains domain-specific data and (optionally) business logic
* **IProductRepository.cs:** The IProductRepository interface that contains the declaration of the operations supported by the product repository
* **ProductRepository.cs:** The product repository class that implements the members of the IProductRepository interface
* **ProductDbContext.cs:** The product data context used to perform CRUD operations for the Product table in the database
* **appsettings.json:** The application’s settings file where you can configure the database connection string, logging metadata, etc.
* **Program.cs:** Any ASP.NET Core application contains a file where the startup code required by the application resides. This file is named Program.cs where the services required by your application are configured. You can specify dependency injection (DI), configuration, middleware, and much more information in this file.

### Specify the Database Connection String

Your application requires a connection string to establish a connection to the database which, in turn, contains the necessary information about the database connection and any initialization parameters sent by a data provider to a data source. Typically, a connection string contains the name of the database to connect to, the instance name of the database server where the database resides, and some other settings pertaining to security of the database.

In ASP.NET Core, the application's settings are stored in a file known as appsettings.json. This file is created by default when you create a new ASP.NET Core project. You can take advantage of the ConnectionString property to retrieve or store the connection string for a database. You can specify the connection string in the appsettings.json file, as shown below:

{

 "Logging": {

 "LogLevel": {

 "Default": "Information",

 "Microsoft.AspNetCore": "Warning"

 }

 },

 "ConnectionStrings": {

 " DefaultConnection":

 "Write your connection string here."

 },

 "AllowedHosts": "\*"

}

You’ll use this connection string to enable the application to connect to the database in a section later in this article.

Typically, you should access the database components in the server project only. If you’re to implement authentication and authorization in your application, you can (optionally) do that in the client project.

### Create the Model Classes

First off, create two solution folders named Models and DataAccess. The former will contain one or more model classes, and the latter will have the data context and repository interfaces and classes. It should be noted that you can always create multiple data context classes in the same project. If your data context class contains many entity references, it’s a good practice to split the data context among multiple data context classes rather than having one large data context class.

Create a new class called Product in a file named Product.cs inside the Models folder and write the following code in there:

namespace SCMS.Product.Models

{

 public record Product

 {

 public Guid Product\_Id

 { get; set; }

 public string Product\_Name

 { get; set; } = default!;

 public string Product\_Description

 { get; set; } = default!;

 public string Product\_Category

 { get; set; } = default!;

 public decimal Product\_Price

 { get; set; } = default!;

 public int Product\_Quantity

 { get; set; } = default!;

 public DateTime Created\_At

 { get; set; } = DateTime.Now;

 public DateTime Modified\_At

 { get; set; } = DateTime.Now;

 }

}

In this implementation, you’ll use only one model class: Product.

### Create the Data Context

In Entity Framework Core (EF Core), a data context is a component used by an application to interact with the database and manage database connections, and to query and persist data in the database. Let’s now create the data context class to enable the application to interact with the database to perform CRUD (Create, Read, Update, and Delete) operations.

To do this, create a new class named ProductDbContext that extends the DbContext class of EF Core and write the following code in there:

public class ProductDbContext: DbContext

{

 public DbSet <Product>

 Products { get; set; }

 protected override void OnConfiguring

 (DbContextOptionsBuilder optionsBuilder)

 {

 base.OnConfiguring (optionsBuilder);

 }

}

In the preceding piece of code, the statement base.OnConfiguring(optionsBuilder) calls the OnConfiguring method of the base class of your ProductDbContext. Because the base class of the ProductDbContext class is DbContext, the call does nothing in particular.

You can specify your database connection string in the OnConfiguring overloaded method of the ProductDbContext class. However, in this implementation, you’ll store the database connection settings in the AppSettings.json file and read it in the Program.cs file to establish a database connection.

Note that your custom data context class (the ProductDbContext class, in this example), must expose a public constructor that accepts an instance of type DbContextOptions<ApplicationDbContext> as an argument. This is needed to enable the runtime to pass the context configuration using a call to the AddDbContext() method to your custom DbContext class. The following code snippet illustrates how you can define a public constructor for your data context class.

public ProductDbContext

(DbContextOptions

<ProductDbContext> options,

IConfiguration configuration) :

base(options)

{

 \_configuration = configuration;

}

### Seed the Database

You might often want to work with data seeding when using Entity Framework Core (EF Core) to populate a blank database with an initial or minimal data set. Data seeding is a one-time process of loading data into a database. The EF Core framework provides an easy way to seed the data using the OnModelCreating() method of the DbContext class.

To generate fake data in your ASP.NET Core application, you can take advantage of the Bogus open-source library. It helps you to seed your database by taking advantage of randomly generated but realistic data. To use this library, you install the Bogus library from NuGet into your project. You can get it from here: <https://www.nuget.org/packages/bogus>.

The following code snippet illustrates how you can generate data using random data from the Bogus library.

private Product[]

GenerateProductData()

{

 var productFaker = new Faker

 <SCS.Product.Models.Product>()

 .RuleFor(x => x.Product\_Id,

 f => Guid.NewGuid())

 .RuleFor(x =>

 x.Product\_Name, f =>

 f.Commerce.ProductName())

 .RuleFor(x => x.Product\_Description,

 f => f.Commerce.ProductDescription())

 .RuleFor(x => x.Product\_Category,

 f => f.Commerce.ProductMaterial())

 .RuleFor(x => x.Product\_Price,

 f => Math.Round(f.Random.Decimal

 (1000, 5000), 2));

 return productFaker.

 Generate(count: 5).ToArray();

}

Invoke the GenerateProductData method in the OnModelCreating method to populate the database with randomly generated data, as shown in the following piece of code:

protected override void

OnModelCreating

(ModelBuilder modelBuilder)

{

 modelBuilder.Entity

 <Models.Product> ().

 ToTable("Product");

 modelBuilder.Entity

 <Models.Product> ().

 HasKey(p => p.Product\_Id);

 var products =

 GenerateProductData();

 modelBuilder.Entity

 <Models.Product> ().

 HasData(products);

}

The complete source code of the ProductDbContext class is given in **Listing 2**.

Register the ProductDb data context instance as a service with the services container of ASP.NET Core using the following piece of code in the Program.cs file:

builder.Services.AddDbContext

<ProductDbContext>(options =>

{

 options.UseSqlServer(

 builder.Configuration

 ["ConnectionStrings:DefaultConnection"]);

});

If your application needs to perform multiple units of work, it’s advisable to use a DbContext factory instead. To do this, register a factory by calling the AddDbContextFactory method in the Program.cs file of your project, as shown in the following code example:

builder.Services.AddDbContextFactory

<ProductDbContext>(options =>

{

 options.UseSqlServer

(builder.Configuration

["ConnectionStrings: DefaultConnection"]);

});

## Create, Update, and Delete Products

Now that you know how to query data from the Product database table, let’s understand how you can create a new product record, update an existing product record, and delete a product record from the database. To do this, you need to create commands to handle each of the Create, Update, and Delete operations.

### Create the ProductRepository Class

A repository class is an implementation of the Repository design pattern and is one that manages data access. The application takes advantage of the repository instance to perform CRUD operations against the database. Now, create a new class named ProductRepository in a file having the same name with a .cs extension. Then write the following code in there:

public class ProductRepository :

IProductRepository

{

}

The ProductRepository class, illustrated in the code snippet below, implements the methods of the IProductRepository interface. Here is how the IProductRepository interface should look:

using SupplyChainManagementSystem.Shared.Models;

using System.Collections.Generic;

namespace SupplyChainManagementSystem.Server.Data

{

 public interface IProductRepository

 {

 public Task<List<Product>>

 GetAllProductsAsync();

 public Task<Product>

 GetProductByIdAsync(Guid id);

 public Task CreateAsync (Product product);

 public Task UpdateAsync (Product product);

 public Task DeleteAsync (Product product);

 }

}

In the model classes (both Product and Supplier), you’ve used record type. The reason is that a record type is an immutable, lightweight data type and much more efficient compared to a class type or even a struct type as far as performance and data integrity is concerned.

The complete source code of the ProductRepository class is given in **Listing 3**.

Create the ProductController Class

Now, create a new controller named ProductController in the Controllers folder of the project. The following code snippet shows how you can take advantage of constructor injection to pass an instance of type IProductRepository interface using the constructor and then use it to retrieve all product records from the database.

public class ProductController : Controller

{

 private IProductRepository \_productRepository;

 public ProductController(IProductRepository

 productRepository)

 {

 \_productRepository = productRepository;

 }

}

The complete source of the ProductController class is given in **Listing 4**.

## Create the Supplier Microservice

In this example, you’ll build the Product microservice application or the Product API. The product microservice application is composed of the following files:

* **Supplier.cs:** The supplier model that contains domain-specific data and (optionally) business logic
* **ISupplierRepository.cs:** The ISupplierRepository interface that contains the declaration of the operations supported by the product repository
* **SupplierRepository.cs:** The supplier repository class that implements the members of the ISupplierRepository interface
* **SupplierDbContext.cs:** The supplier data context used to perform CRUD operations for the Supplier table in the database
* **appsettings.json:** The application’s settings file where you can configure the database connection string, logging metadata, etc.
* **Program.cs:** Any ASP.NET Core application contains a file where the startup code required by the application resides. This file is named Program.cs where the services required by your application are configured. You can specify dependency injection (DI), configuration, middleware, and much more information in this file.

### Create the Supplier Class

Create a new class named Order in a file having the same name with a .cs extension and write the following code in there:

public class Supplier

{

 public Guid Id { get; set; }

 public string FirstName { get; set; }

 public string LastName { get; set; }

 public string Address { get; set; }

 public string Email { get; set; }

 public string Phone { get; set; }

}

The other entity classes are not being shown here for brevity and also because this is a minimalistic implementation to illustrate a microservices-based application with just two modules: Product and Supplier.

### Seed the Database

The following code snippet illustrates how you can generate random supplier data using random data from the Bogus library:

private Supplier[]

GenerateSupplierData()

{

 var supplierFaker =

 new Faker<Supplier>()

 .RuleFor(x => x.Supplier\_Id,

 f => Guid.NewGuid())

 .RuleFor(x => x.Supplier\_Name,

 f => f.Person.FullName)

 .RuleFor(x => x.Supplier\_Address,

 f => f.Person.Address.ToString())

 RuleFor(x => x.Supplier\_Phone,

 f => f.Person.Phone)

 .RuleFor(x => x.Supplier\_Email,

 f => f.Person.Email);

 return supplierFaker.Generate

 (count: 5).ToArray();

}

Next, invoke the GenerateSupplierData method in the OnModelCreating method to populate the database with randomly generated supplier data, as shown in the following piece of code:

protected override void

OnModelCreating

(ModelBuilder modelBuilder)

{

 modelBuilder.Entity<Supplier>

 ().ToTable("Supplier");

 modelBuilder.Entity

 <Supplier>().HasKey(s =>

 s.Supplier\_Id);

 var suppliers = GenerateSupplierData();

 modelBuilder.Entity<Supplier>

 ().HasData(suppliers);

}

The complete source code of the SupplierDbContext class is given in **Listing 5**.

Lastly, register the SupplierDb data context instance with the services container of ASP.NET Core using the following piece of code in the Program.cs file:

builder.Services.AddDbContext

<SupplierDbContext>

(options =>

{

 options.UseSqlServer(

 builder.Configuration

 ["ConnectionStrings:DefaultConnection"]);

});

### Create the ISupplierRepository Interface

Create a new .cs file named ISupplierRepository in your project and replace the default generated code with the following code snippet:

public interface ISupplierRepository

{

 public Task<List<Supplier>> GetSuppliers();

 public Task<Supplier> GetSupplier(int Id);

}

### Create the SupplierRepository Class

Now create a new class named SupplierRepository in a file having the same name with a .cs extension. Write the following code in there:

public class SupplierRepository :

ISupplierRepository

{

}

The SupplierRepository class, illustrated in the code snippet below, implements the methods of the ISupplierRepository interface.

public async Task<List<Supplier>> GetSuppliers()

{

 return await Task.FromResult(suppliers);

}

public async Task<Order> GetSupplier(int Id)

{

 return await Task.FromResult(suppliers.

 FirstOrDefault(x => x.Id == Id));

}

The complete source code of the SupplierRepository class is given in **Listing 6**.

## Create the SupplierController Class

Now, create a new controller named SupplierController in the Controllers folder of the project. The SupplierController exposes a couple of action methods to enable consuming one or more supplier records from a client application. Similar to the ProductController class, you’ll leverage constructor injection to build an instance of the SupplierRepository class and then use it to retrieve supplier data from the database, as shown in the code snippet given below:

public class SupplierController :

Controller

{

 private ISupplierRepository

 \_supplierRepository;

 public SupplierController

 (ISupplierRepository

 supplierRepository)

 {

 \_supplierRepository =

 supplierRepository;

 }

}

The complete source of the SupplierController class is given in **Listing 7**.

## Register the Service Instances with IServiceCollection

The following code snippet illustrates how you can register the IRequestHandler instances added as a transient service to the IServiceCollection:

builder.Services.AddScoped

<IProductRepository,

ProductRepository>();

Register the IProductRepository instance with the service collection, as shown below:

builder.Services.AddScoped

<IProductRepository, ProductRepository>();

The following code snippet illustrates how an instance of type ISupplierRepository is added as a scoped service to the IServiceCollection:

builder.Services.AddScoped

<ISupplierRepository,

SupplierRepository>();

The complete source code of the Program.cs file is given in **Listing 8**.

## Execute the SupplierChainManagementSystem Application

Finally, run the application by pressing the F5 key on your keyboard. When the application executes, you’ll observe the Home page displayed, as shown in **Figure 13**.



**Figure 13:** The SupplyChainManagementSystem application in execution

You can implement authentication in your Blazor application using the built-in authentication provided by Microsoft. If you already have a Blazor application, you can’t follow this approach to implement authentication in your Blazor applications. It’s preferable to implement an out-of-the-box authentication to overcome the limitations of this approach.

## Where Do We Go from Here?

Blazor is a feature-rich contemporary framework and cutting-edge technology that avoids dealing with the nitty-gritty of developing interactive web apps. It supports both client-side and server-side hosting models, providing flexibility, enhanced performance, and the ability to build full-stack applications in .NET and C#.

Hosting Models in Blazor

Blazor offers a range of hosting models and gives flexibility to developers in terms of whether to host the application on the client-side or on the server-side. The supported hosting models are Blazor Server and Blazor WebAssembly.

Sponsored Sidebar

AI Executive Briefing

Experience the game-changing impact of AI through CODE Consulting's Executive Briefing service. Uncover the immense potential and wide-ranging benefits of AI in every industry. Our briefing provides strategic guidance for seamless implementation, covering crucial aspects such as infrastructure, talent acquisition, and leadership.

Discover how to effectively integrate AI and propel your organization into future success.

Contact us today to schedule your executive briefing and embark on a journey of AI-powered growth. [www.codemag.com/ai](http://www.codemag.com/ai)

Listing 1: The Database Script

 -- Supplier table

CREATE TABLE Supplier (

 Supplier\_Id UniqueIdentifier PRIMARY KEY,

 Supplier\_Name VARCHAR(255) NOT NULL,

 Supplier\_Address VARCHAR(255) NOT NULL,

 Supplier\_Contact VARCHAR(255),

 Supplier\_Phone VARCHAR(20),

 Supplier\_Email VARCHAR(20)

);

-- Product table

CREATE TABLE Product (

 Product\_Id UniqueIdentifier PRIMARY KEY,

 Product\_Name VARCHAR(255) NOT NULL,

 Product\_Description TEXT,

 Unit\_Price DECIMAL(10, 2) NOT NULL,

 Product\_Quantity INT NOT NULL

);

-- Order table

CREATE TABLE [Order] (

 Order\_Id UniqueIdentifier PRIMARY KEY,

 Product\_Id UniqueIdentifier NOT NULL,

 Supplier\_Id UniqueIdentifier NOT NULL,

 Order\_date DATE NOT NULL,

 Order\_Quantity INT NOT NULL,

 FOREIGN KEY (Product\_Id) REFERENCES Product(Product\_Id),

 FOREIGN KEY (Supplier\_Id) REFERENCES Supplier(Supplier\_Id)

);

-- Shipment table

CREATE TABLE Shipment (

 Shipment\_Id UniqueIdentifier PRIMARY KEY,

 Order\_Id UniqueIdentifier NOT NULL,

 Shipment\_Date DATE NOT NULL,

 Estimated\_Arrival\_Date DATE NOT NULL,

 Actual\_Arrival\_Date DATE,

 FOREIGN KEY (Order\_Id) REFERENCES [Order](Order\_Id)

);

Listing 2: The ProductDbContext class

using Bogus;

using Microsoft.EntityFrameworkCore;

using SupplyChainManagementSystem.Shared.Models;

namespace SupplyChainManagementSystem.Server.Data

{

 public class ProductDbContext : DbContext

 {

 private readonly IConfiguration \_configuration;

 public ProductDbContext(DbContextOptions

 <ProductDbContext> options,

 IConfiguration configuration) : base(options)

 {

 \_configuration = configuration;

 }

 protected override void

 OnConfiguring(DbContextOptionsBuilder

 optionsBuilder)

 {

 \_ = optionsBuilder.UseSqlServer(\_configuration.

 GetConnectionString("DefaultConnection")).

 EnableSensitiveDataLogging();

 }

 public DbSet<Product> Products { get; set; }

 protected override void OnModelCreating

 (ModelBuilder modelBuilder)

 {

 modelBuilder.Entity<Product>().ToTable("Product");

 modelBuilder.Entity<Product>().HasKey(p =>

 p.Product\_Id);

 var products = GenerateProductData();

 modelBuilder.Entity<Product>().HasData(products);

 }

 private Product[] GenerateProductData()

 {

 var productFaker = new Faker<Product>()

 .RuleFor(x => x.Product\_Id, f => Guid.NewGuid())

 .RuleFor(x => x.Product\_Name, f =>

 f.Commerce.ProductName())

 .RuleFor(x => x.Product\_Description, f =>

 f.Commerce.ProductDescription())

 .RuleFor(x => x.Product\_Category, f =>

 f.Commerce.ProductMaterial())

 .RuleFor(x => x.Product\_Price, f =>

 Math.Round(f.Random.Decimal(1000, 5000), 2));

 return productFaker.Generate(count: 5).ToArray();

 }

 }

}

Listing 3: The ProductRepository class

using SupplyChainManagementSystem.Shared.Models;

namespace SupplyChainManagementSystem.Server.Data

{

 public class ProductRepository

 {

 private readonly

 ProductDbContext \_productDbContext;

 public ProductRepository(

 ProductDbContext productDbContext)

 {

 \_productDbContext = productDbContext;

 }

 public async Task<List<Product>> GetProducts()

 {

 return await Task.FromResult(

 \_productDbContext.Products.ToList());

 }

 public async Task<Product> GetProduct(Guid Id)

 {

 return await Task.FromResult

 (\_productDbContext.Products.FirstOrDefault(

 x => x.Product\_Id == Id));

 }

 }

}

Listing 4: The ProductController class

[Route("api/[controller]")]

[ApiController]

public class ProductController : Controller

{

 private IProductRepository \_productRepository;

 public ProductController

 (IProductRepository productRepository)

 {

 \_productRepository = productRepository;

 }

 [HttpGet("GetProducts")]

 public async Task<List<Product>> GetProducts()

 {

 return await \_productRepository.GetProducts();

 }

}

Listing 5: The SupplierDbContext class

using Bogus;

using Microsoft.EntityFrameworkCore;

using SupplyChainManagementSystem.Shared.Models;

namespace SupplyChainManagementSystem.Server.Data

{

 public class SupplierDbContext : DbContext

 {

 private readonly IConfiguration \_configuration;

 public SupplierDbContext(DbContextOptions

 <SupplierDbContext> options,

 IConfiguration configuration) : base(options)

 {

 \_configuration = configuration;

 }

 protected override void

 OnConfiguring(DbContextOptionsBuilder

 optionsBuilder)

 {

 \_ = optionsBuilder.UseSqlServer

 (\_configuration.

 GetConnectionString("DefaultConnection")).

 EnableSensitiveDataLogging();

 }

 public DbSet<Supplier> Suppliers { get; set; }

 protected override void OnModelCreating

 (ModelBuilder modelBuilder)

 {

 modelBuilder.Entity<Supplier>().ToTable("Supplier");

 modelBuilder.Entity<Supplier>().HasKey(s =>

 s.Supplier\_Id);

 var suppliers = GenerateSupplierData();

 modelBuilder.Entity<Supplier>().HasData(suppliers);

 }

 private Supplier[] GenerateSupplierData()

 {

 var supplierFaker = new Faker<Supplier>()

 .RuleFor(x => x.Supplier\_Id,

 f => Guid.NewGuid())

 .RuleFor(x => x.FirstName, f =>

 f.Person.FirstName)

 .RuleFor(x => x.LastName, f =>

 f.Person.LastName)

 .RuleFor(x => x.Address, f =>

 f.Person.Address.ToString())

 .RuleFor(x => x.Phone, f =>

 f.Person.Phone)

 .RuleFor(x => x.Email, f =>

 f.Person.Email);

 return supplierFaker.Generate

 (count: 5).ToArray();

 }

 }

}

Listing 6: The SupplierRepository class

using SupplyChainManagementSystem.Shared.Models;

namespace SupplyChainManagementSystem.Server.Data

{

 public class SupplierRepository : ISupplierRepository

 {

 private readonly

 SupplierDbContext \_supplierDbContext;

 public SupplierRepository(

 SupplierDbContext supplierDbContext)

 {

 \_supplierDbContext = supplierDbContext;

 }

 public async Task<List<Supplier>> GetSuppliers()

 {

 return await Task.FromResult(

 \_supplierDbContext.Suppliers.ToList());

 }

 public async Task<Supplier> GetSupplier(Guid Id)

 {

 return await Task.FromResult

 (\_supplierDbContext.Suppliers.

 FirstOrDefault(

 x => x.Supplier\_Id == Id));

 }

 }

}

Listing 7: The SupplierController class

[Route("api/[controller]")]

[ApiController]

public class SupplierController : ControllerBase {

 private ISupplierRepository \_supplierRepository;

 public SupplierController

 (ISupplierRepository supplierRepository)

 {

 \_supplierRepository = supplierRepository;

 }

 [HttpGet("{id}")]

 public async Task<Supplier> GetSupplier(int id)

 {

 return await \_supplierRepository.GetSupplier(id);

 }

 [HttpGet("GetSuppliers")]

 public async Task<List<Supplier>> GetSuppliers()

 {

 return await \_supplierRepository.GetSuppliers();

 }

}

Listing 8: The Program.cs file (Server/API project)

using Microsoft.AspNetCore.Identity;

using Microsoft.EntityFrameworkCore;

using SupplyChainManagementSystem.Server.Data;

using SupplyChainManagementSystem.Server.Models;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddControllers();

builder.Services.AddDbContext<ApplicationDbContext>(

 options => options.UseSqlServer(

 builder.Configuration.GetConnectionString

("DefaultConnection")));

builder.Services.AddIdentity<ApplicationUser,

IdentityRole>().AddEntityFrameworkStores

<ApplicationDbContext>();

builder.Services.AddScoped<IProductRepository,

ProductRepository>();

builder.Services.AddScoped<ISupplierRepository,

SupplierRepository>();

var app = builder.Build();

// Configure the HTTP request pipeline.

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();