ZZZ Web Application

(…Screenshot…)

Students (Names & IDs):

Teacher:

Date:

Contents

[Project Overview 5](#_Toc165474926)

[Project Description 5](#_Toc165474927)

[Which This Project is Needed? 5](#_Toc165474928)

[Technology Stack 6](#_Toc165474929)

[HTML, CSS and JS 6](#_Toc165474930)

[1. HTML (HyperText Markup Language) 6](#_Toc165474931)

[2. CSS (Cascading Style Sheets) 6](#_Toc165474932)

[3. JavaScript (JS) 6](#_Toc165474933)

[4. How They Work Together: 6](#_Toc165474934)

[Blazor Server 8](#_Toc165474935)

[**Blazor Server: Server-Side Rendering with State Management** 8](#_Toc165474936)

[**Service Injection: Dependency Management Made Easy** 8](#_Toc165474937)

[Benefits of Using Service Injection: 9](#_Toc165474938)

[Types of Services: 9](#_Toc165474939)

[JS and C# interactions 10](#_Toc165474940)

[1. C# Functions Callable from JavaScript 10](#_Toc165474941)

[2. JSInterop: Low-Level Access to Browser APIs: 10](#_Toc165474942)

[3. Calling JavaScript Functions from C# (Optional): 11](#_Toc165474943)

[Model-View-View Model 12](#_Toc165474944)

[1. Model: 12](#_Toc165474945)

[2. ViewModel (ViewModel): 12](#_Toc165474946)

[3. View (Blazor Component): 12](#_Toc165474947)

[Benefits of MVVM: 12](#_Toc165474948)

[MVVM in Blazor Server: 13](#_Toc165474949)

[Summary: 13](#_Toc165474950)

[Entity Framework Core 14](#_Toc165474951)

[Benefits of EF Core Integration: 14](#_Toc165474952)

[Key Concepts: 14](#_Toc165474953)

[Integration with Blazor Server: 14](#_Toc165474954)

[Database Design 16](#_Toc165474955)

[SQL Table 16](#_Toc165474956)

[Entity Framework Integration 17](#_Toc165474957)

[Classes 17](#_Toc165474958)

[DbContext class 17](#_Toc165474959)

[App User class 17](#_Toc165474960)

[ZZZ Class 17](#_Toc165474961)

[Creating DB from EF classes 18](#_Toc165474962)

[Package Manager Console 18](#_Toc165474963)

[Examples of Using EF 20](#_Toc165474964)

[Create 20](#_Toc165474965)

[Update 20](#_Toc165474966)

[Select 20](#_Toc165474967)

[Delete 20](#_Toc165474968)

[Authentication 21](#_Toc165474969)

[Web App Pages 22](#_Toc165474970)

[Authentication 22](#_Toc165474971)

[Register 22](#_Toc165474972)

[Login 22](#_Toc165474973)

[Forget Password 22](#_Toc165474974)

[Reset Password 22](#_Toc165474975)

[Edit Profile/User 22](#_Toc165474976)

[List of All User (Admin) 22](#_Toc165474977)

[Contact Us 22](#_Toc165474978)

[Help 22](#_Toc165474979)

[About Us 22](#_Toc165474980)

[Contact Us 22](#_Toc165474981)

[Code 23](#_Toc165474982)

[Model Classes 23](#_Toc165474983)

[AppUser.cs 23](#_Toc165474984)

[Address.cs 23](#_Toc165474985)

[General Services 24](#_Toc165474986)

[Class 1 24](#_Toc165474987)

[Class 2 24](#_Toc165474988)

[General Classes 25](#_Toc165474989)

[Appsettings.json 25](#_Toc165474990)

[Program.cs 25](#_Toc165474991)

[\_Host.cshtml 25](#_Toc165474992)

[Components 26](#_Toc165474993)

[Register 26](#_Toc165474994)

[Login 26](#_Toc165474995)

[Project Challenges and Learning Outcomes 27](#_Toc165474996)

[Difficulties 27](#_Toc165474997)

[Learning Outcomes 27](#_Toc165474998)

[References 28](#_Toc165474999)

# Project Overview

Bla Bla

## Project Description

Bla Bla

## Which This Project is Needed?

Bla Bla

# Technology Stack

## HTML, CSS and JS

n your Blazor Server web application, you'll interact with three core technologies that work together to create the user interface (UI) you see in your browser:

### 1. HTML (HyperText Markup Language)

* HTML is the foundation of web pages. It defines the structure and content of your UI using a set of tags and attributes.
* Think of HTML as a blueprint, outlining the different sections of your page like headings, paragraphs, forms, buttons, images, etc.
* It doesn't handle styling or interactivity, but provides a framework for these elements.

### 2. CSS (Cascading Style Sheets)

* CSS breathes life into your HTML structure by adding styles like colors, fonts, layouts, and positioning. It defines how the elements on your page should appear and behave visually.
* By using CSS selectors, you target specific HTML elements and apply styles to them. This allows you to create a consistent and visually appealing UI.

### 3. JavaScript (JS)

* JavaScript adds interactivity and dynamism to your web pages. It's a scripting language that executes code directly in the user's browser.
* You can use JavaScript for various purposes, such as:
  + Validating form input before submission.
  + Adding dynamic behavior to UI elements (e.g., showing/hiding content, animations).
  + Communicating with servers using AJAX requests (asynchronous data fetching).
* In Blazor Server applications, JavaScript can be used to interact with Blazor components and enhance the overall user experience

### 4. How They Work Together:

HTML, CSS, and JavaScript complement each other to create a complete web page:

1. **HTML** provides the structure.
2. **CSS** defines the appearance.
3. **JavaScript** adds interactivity.

Within Blazor Server, these technologies work alongside Blazor components. While Blazor handles most of the UI logic and data binding, you might still use HTML, CSS, and JavaScript for:

* Customizing the visual style of Blazor components.
* Adding basic client-side validation or interactivity.
* Integrating with external libraries for advanced functionality.

By mastering these core technologies, you have a solid foundation for building user-friendly interfaces in Blazor Server web applications and beyond.

## Blazor Server

Blazor Server, a web framework from Microsoft, offers a compelling approach to building interactive web UIs using C# and Razor syntax. This section delves into Blazor Server's architecture and how service injection empowers cleaner and more maintainable application development.

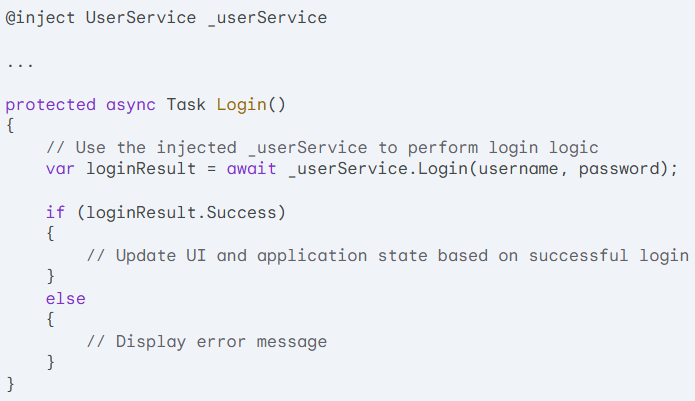
### **Blazor Server: Server-Side Rendering with State Management**

* **Server-Side Rendering:** Unlike Blazor WebAssembly, Blazor Server renders UI components on the server. This means the complete HTML, CSS, and JavaScript are generated on the server and sent to the user's browser.
* **User Interactions:** When a user interacts with the UI (e.g., clicks a button, submits a form), these events are sent back to the server.
* **State Management:** Since rendering happens on the server, Blazor Server manages the application's state centrally. This ensures consistency across all connected clients.

### **Service Injection: Dependency Management Made Easy**

* Service injection is a design pattern commonly used in web development to manage dependencies between objects. In Blazor Server, it simplifies how components access services they depend on.
* **@inject Directive:** You use the @inject directive within a Blazor component to declare that it requires an instance of a particular service.

**Example:**



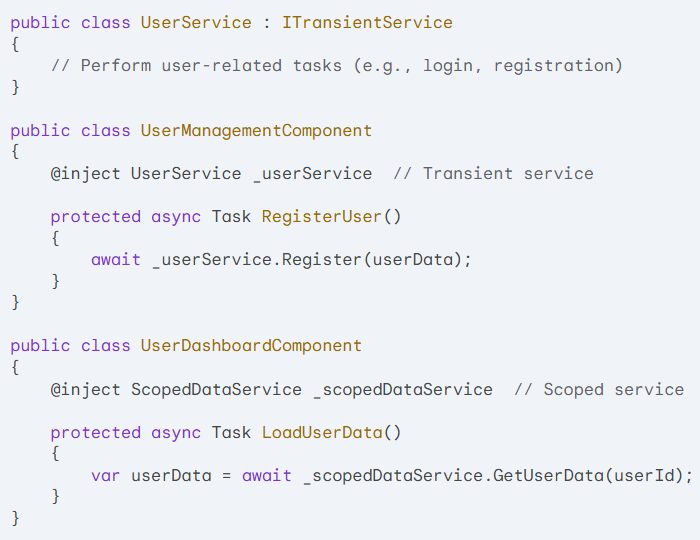
### Benefits of Using Service Injection:

* **Loose Coupling:** Components are not tightly coupled to specific service implementations. This makes them easier to test and reuse.
* **Dependency Injection Container:** Blazor Server leverages a dependency injection container (DIC) to manage the lifetime of services. This ensures that components always receive the correct service instance.
* **Maintainability:** As your application grows, managing dependencies through injection becomes crucial for cleaner code and easier future modifications.
* **Code Reusability:** You can create generic services that can be injected in different components, reducing code duplication.

### Types of Services:

* **Transient Services:** A new instance of the service is created for each request or component that injects it.
* **Scoped Services:** A single instance of the service is created for each web request (user session) and used by all components within that request.
* **Singleton Services:** A single instance of the service is created for the entire application's lifetime.

Examples:



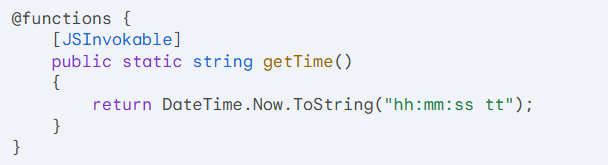
## JS and C# interactions

Blazor Server provides mechanisms for interaction between JavaScript (JS) code running in the user's browser and C# code executing on the server. This collaboration allows you to leverage the strengths of both environments, enabling richer user experiences in your Blazor Server applications. Here's a breakdown of the key approaches for JS and C# interaction:

### 1. C# Functions Callable from JavaScript

* Blazor Server components allow you to define C# methods using the @functions directive. These methods can be marked with the [JSInvokable] attribute, making them callable from JavaScript code running in the browser.
* When you call a C# function from JavaScript, the arguments are serialized and sent to the server. The server executes the C# function, and the return value (if any) is serialized back to JavaScript.

Example (C# function to be called by JS)

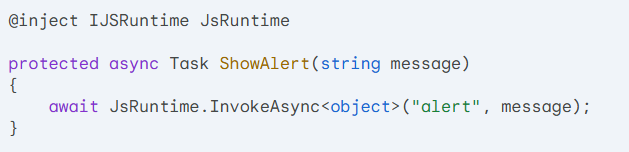


And then in your JS code,



### 2. JSInterop: Low-Level Access to Browser APIs:

* Blazor Server provides the IJSRuntime interface for low-level access to browser APIs from C# code. This can be useful when you need to interact with functionalities not directly exposed by Blazor's built-in components.
* Use caution with IJSRuntime as it bypasses Blazor's managed state and can introduce security risks if not used properly.



Same way you can access the browser cookies or by using a dedicated C# Service called: ProtectedLocalStorage

### 3. Calling JavaScript Functions from C# (Optional):

While less common, you can also invoke JavaScript functions from within C# code using IJSRuntime.InvokeAsync<T>(string identifier, object[] args). However, this approach should be used sparingly as it can lead to tighter coupling between your C# and JavaScript code.

## Model-View-View Model

The Model-View-ViewModel (MVVM) pattern is a software design pattern commonly used in building user interfaces (UIs) that separates concerns and promotes maintainability. Here's a breakdown of the three components in an MVVM context for Blazor Server applications:

### 1. Model:

* Represents the data objects (e.g., User class) and business logic of your application.
* This layer encapsulates the data structures and operations that manipulate that data.
* It should be independent of any specific UI presentation or technology.

### 2. ViewModel (ViewModel):

* Acts as an intermediary between the model and the view (Blazor component).
* The view model exposes relevant data from the model in a format suitable for the view, handling data transformation and validation as needed.
* It might also contain logic for presenting data in a way that's conducive to the UI, such as formatting dates or converting values to user-friendly units.

### 3. View (Blazor Component):

* This is the visual representation of your UI, built using Blazor components (Razor syntax and C# code).
* It interacts with the view model for data binding and user interactions.
* The view doesn't directly access the model; it relies on the view model to provide the necessary data and functionality.

### Benefits of MVVM:

* **Separation of Concerns:** Keeps UI logic, data logic, and presentation separate, leading to cleaner, more maintainable code.
* **Testability:** Easier to write unit tests for view models and views in isolation.
* **Flexibility:** The view model can be reused across different views, promoting code reusability.

### MVVM in Blazor Server:

* In a Blazor Server application, the view model acts as a bridge between the server-side model and the Blazor component.
* Blazor Server components use data binding to directly access properties and methods exposed by the view model.
* You can use change notification mechanisms within the view model to automatically update the UI whenever the underlying data changes.

### Summary:

MVVM promotes a clean separation of concerns in your Blazor Server applications, leading to more maintainable and testable code. By understanding these roles and their interactions, you can effectively structure your Blazor projects with well-defined responsibilities in each layer.

## Entity Framework Core

Entity Framework Core (EF Core) integration is a powerful feature in Blazor Server applications that allows you to interact with relational databases in a more structured and efficient way. Here's a brief overview:

### Benefits of EF Core Integration:

* **Simplified Data Access:** EF Core provides a layer of abstraction between your application code and the underlying database. You can interact with data using C# objects (entities) that map to your database tables. This simplifies data access and reduces the need for writing complex SQL queries directly.
* **Automatic Change Tracking:** EF Core automatically tracks changes made to your entities in memory. This simplifies data updates and allows you to save changes back to the database with minimal code.
* **Migrations and Schema Management:** EF Core provides a powerful migration system that allows you to evolve your database schema over time. You can define migrations to represent changes to your entities and tables, and EF Core handles the process of applying those changes to your database.

### Key Concepts:

* **DbContext:** This class serves as the bridge between your application and the database. It manages connections, entity sets, and change tracking.
* **Entity:** C# classes that represent database tables. You define properties within the entity class that map to table columns.
* **Migrations:** Scripts that track changes to your database schema over time. You can use EF Core's migration tools to generate and apply these scripts.

### Integration with Blazor Server:

* You can inject the DbContext instance into your Blazor Server components using dependency injection.
* Components can then use the DbContext to perform CRUD (Create, Read, Update, Delete) operations on your data.
* EF Core provides features like eager loading and lazy loading to efficiently retrieve related data from the database.

By combining Blazor Server's UI capabilities with EF Core's data access functionality, you can build data-driven web applications with cleaner code and efficient database interaction.

# Database Design

Bla Bla

SQL Database

(…Screen shot…Entire project)

## SQL Table

Bla Bla

(…Screen shot for User Table…)

(…Screen shot for Address Table…)

More…

# Entity Framework Integration

Bla Bla

## Classes

### DbContext class

Bla Bla

(…Code…)

### App User class

Bla Bla

(…Code…)

### ZZZ Class

Bla Bla

(…Code…)

## Creating DB from EF classes

Bla Bla

### Package Manager Console

PMC is a powerful tool within Visual Studio that allows you to manage NuGet packages and execute various commands related to your project. In the context of creating a database for your Blazor Server application, PMC plays a crucial role in managing Entity Framework Core migrations.

**1. appsettings.json**

* This file, located in the root directory of your project, stores configuration information for your application, including the connection string for your SQL database. The connection string specifies the server address, database name, and credentials needed to connect to the database.

**Example appsettings.json (replace with your actual values):**

{

"ConnectionStrings": {

"DefaultConnection": "Server=localhost;Database=YourDatabaseName;Trusted\_Connection=True;"

}

}

**2. Program.cs**

* This file serves as the entry point for your application. In a Blazor Server project that uses EF Core, Program.cs is responsible for injecting the DbContext instance into the application's dependency injection container. This DbContext is configured to use the connection string information from appsettings.json.

**(Replace with your Code…)**

C#

using Microsoft.AspNetCore.Hosting;

using Microsoft.Extensions.Configuration;

using Microsoft.Extensions.DependencyInjection;

using Microsoft.Extensions.Hosting;

using Microsoft.Extensions.Logging;

using YourProjectName.Data; // Replace with your project namespace

var builder = WebApplication.CreateBuilder(args);

// Configure DbContext with connection string from appsettings.json

builder.Services.AddDbContext<YourDbContext>(options =>

{

options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection"));

});

**3. Add-Migration**

* This PMC command is used to create a new migration file. Migration files track changes made to your data models (e.g., adding or modifying database tables or columns). Each time you modify your model classes, you need to run Add-Migration to generate a new migration file that reflects these changes.

**Example Usage in PMC:**

Command line: Add-Migration InitialCreate

This command will create a migration file named InitialCreate.cs in your project's Migrations folder.

**4. Update-Database**

* Once you've created a migration file using Add-Migration, you can apply the changes defined in the migration to your actual SQL database using the Update-Database command. This command executes the SQL statements contained in the migration file, creating or modifying tables and columns in your database according to your model definitions.

**Example Usage in PMC:**

Command line: Update-Database

This command will apply the changes described in the latest migration file (InitialCreate.cs in this example) to your SQL database.

## Examples of Using EF

Bla Bla

### Create

Bla Bla

(…Code snippet…)

### Update

Bla Bla

(…Code snippet…)

### Select

Bla Bla

(…Code snippet…)

### Delete

Bla Bla

(…Code snippet…)

# Authentication

Bla Bla..using email & password

When registering, checking if same email exists

When log-in, checking if email and password are the same

Password Hash?

Saving zzz in Cookies?

# Web App Pages

## Authentication

### Register

(…Screen shot placeholder…)

Bla Bla

### Login

(…Screen shot placeholder…)

Bla Bla

### Forget Password

(…Screen shot placeholder…)

Bla Bla

### Reset Password

(…Screen shot placeholder…)

Bla Bla

### Edit Profile/User

(…Screen shot placeholder…)

Bla Bla

### List of All User (Admin)

(…Screen shot placeholder…)

Bla Bla

## Contact Us

### Help

(…Screen shot placeholder…)

Bla Bla

### About Us

(…Screen shot placeholder…)

Bla Bla

### Contact Us

(…Screen shot placeholder…)

Bla Bla

# Code

## Model Classes

### AppUser.cs

(…Code…)

Bla

### Address.cs

(…Code…)

Bla

## General Services

### Class 1

(…Code…)

Bla Bla

### Class 2

(…Code…)

Bla Bla

## General Classes

### Appsettings.json

(…Code…)

Bla

### Program.cs

(…Code…)

Bla

### \_Host.cshtml

(…Code…)

Bla

## Components

### Register

#### Register.razor

(…Code…)

Bla

#### Register.cs

(…Code…)

Bla

### Login

#### Login.razor

(…Code…)

#### Login.cs

(…Code…)

# Project Challenges and Learning Outcomes

## Difficulties

Bla…

Dev Environment

Things didn’t work

Things didn't look nice

## Learning Outcomes

Bla…

Help from teacher

Help from YouTube channels

Help from Stack Overflow

Help from LLMs (ChatGPT, Gemini,…)

# References