**Web aplikacija za unos podataka o kontaktima**

BLAZOR & C# - REST API

Dora Parmać

Kolovoz, 2021.

Contents

[INTRODUCTION – IN GENERAL 3](#_Toc79244470)

[CLASS MODELS 5](#_Toc79244471)

[DATA ACCESS 6](#_Toc79244472)

[REST API 6](#_Toc79244473)

[DATABASE SUPPORT 9](#_Toc79244474)

[REPOSITORY 10](#_Toc79244475)

[CONTROLLER 11](#_Toc79244476)

[MODEL VALIDATION 11](#_Toc79244477)

[CALL REST API FROM ASP NET CORE BLAZOR 12](#_Toc79244478)

[BUILDING BLAZOR FORM 12](#_Toc79244479)

# INTRODUCTION – IN GENERAL

For this assignment, I will use Blazor and C#.

Most common:

Server – C#, Java, PHP

Client – Angular, React, Vue

C# can be both Server and Client => Blazor (can run C# code directly in browser using WebAssembly)

Blazor WebAssembly (for compiling) and Blazor Server (app is executed on server from .NET)

When event occurs on client, info about event is sent to the server. Client doesn't recieve whole http, only DIFF (difference; meaning only part of the page that is changed is sent back => more responsive, faster)

BlazorServer i BlazorWebAssembly => both projects must be Start projects

Blazor WebAssembly:

* Pros:
  + active server connection not required
  + Client resources are used
  + ASP.NET Core web server not required
  + Can be hosted on own server, cloud, azure
* Cons:
  + First request usually takes longer
  + Restricted to the capabilities of thr browser
  + Capable client hardware and software is required

Files and folders in ASP.NET Core Blazor project

* + Program.cs => This file contains the Main() method which is the entry point for both the project types (i.e Blazor WebAssembly and Blazor Server). main method which is the entry point, runs on the server from within
  + Specifies startup class, startup.cs, app startup logic, configure services, DI services
  + Root app page: /\_Host (when first request hits app)
  + Components are the building blocks of blazor app
  + App is root component, built in router component to set up client side routing, it intercepts browser navigation and renders the page
  + Js file sets up real time signal connection between server and client
  + @page „/counter“ specifies page navigation, at the top of the file

Everything in blazor app is razor component, component driven framework.

Components can be nested, reused and shared across multiple projects.

Component files have the extension .razor.

Component name must start with an uppercase character

Two parts of the Blazor component:

1. @page is for user interface
2. @code is for C# code, and a reference to a variable from @code in @page part must start with „@“

Name of the generated class matches the name of the component file (Uppercase).

For including component: ie. <Counter />

Reusable, nonpages components are in the shared folder.

If we have longer code, good thing is to separate @page and @code into two different files. Two ways to do that:

1. partial files: Add new class file in pages => Counter.razor.cs, redefine as PUBLIC PARTIAL CLASS
2. base class: Add new class file => CounterBase.cs which inherits ComponentBase and variables must be at least protected, or public, not private

namespace: using Microsoft.AspNetCore.Components;

in Counter.razor must specifey @inherits Counterbase

A protected member is accessible within its class and by derived class instances.

# CLASS MODELS

* New project: Class library => UserManagment.Models

public class User

{

public int UserID { get; set; }

public string FirstName { get; set; }

public string LastName { get; set; }

public string Email { get; set; }

public int MobilePhone { get; set; }

public string Address { get; set; }

}

* Add new project to the solution => Blazor Server App: UserManagment.Web

Startup project

* Add reference to the model project (right click on dependencies in UserManagment.Web
* Add new Razor component on Pages folder: UserList.razor
* u UserListBase (Pages) promijenit u asinkrono izvrštavanje I dodat spinner

# DATA ACCESS

There are two ways for accessing data:

1. Dana access without REST Api: Use only if the Blazor project always runs on the server!

Diagram

Description automatically generated

1. Data access with REST Api

Diagram

Description automatically generated

# REST API

REST API allows applications to interact with each other and exchange data.

Third party that has data that I need. (weather ex)

The communication between the client and the server happens over HTTP.

A picture containing background pattern

Description automatically generated

Stateless: we should not be storing anything on the server related to the client.

Each request should be treated independently by the server.

Diagram

Description automatically generated with medium confidence

Common HTTP verbs.

* GET
* POST
* PUT
* PATCH
* DELETE

A picture containing diagram

Description automatically generatedTable

Description automatically generated with low confidence

PATCH is used when you want to do a partial update i.e only a subset of the properties.

POSTMAN is for testing!

We want Blazor App calling REST Api (storing and retrieving data from database)

Steps:

* New solution project: ASP.NET Core Web Api (this provides data to the Blazor App)
* …api must start before …web

Table

Description automatically generated

* Want to use models in API, so add reference

# DATABASE SUPPORT

* Entity Framework Core is an ORM (i.e an Object-Relational Mapper)

One of the very important classes in Entity Framework Core is the DbContext class. This is the class that we use in our application code to interact with the underlying database. It is this class that manages the database connection and is used to retrieve and save data in the database.

* Include DB Connection string in appSetings.json
* Install nuget packages:

1. Microsoft.EntityFrameworkCore.SqlServer
2. Microsoft.EntityFrameworkCore.Tools (for migrations) IN API (watch for this ERROR)

* To the API project add new folder: Models => DBContext class will be in this folder => add new class AppDbContext.cs

public class AppDbContext : DbContext

{

public AppDbContext(DbContextOptions<AppDbContext> options) base(options)

{

}

public DbSet<User> Users { get; set; }

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

base.OnModelCreating(modelBuilder);

modelBuilder.Entity<User>().HasData( new User

{

UserID = 1,

FirstName = "Ana",

LastName = "Anić",

Email = "ana.anic@gmail.com",

MobilePhone = "+385 91 234 5678",

Address = "Ulica grada 1"

}

}

Configure server: Startup.cs u API

services.AddDbContext<AppDbContext>(options =>

options.UseSqlServer(Configuration.GetConnectionString("DBConnection")));

services.AddControllers();

* Add-Migration InitialCreate u Package Manager Console
* Update-Database in PM

# REPOSITORY

* Repository Pattern is an abstraction of the Data Access Layer. It hides the details of how exactly the data is saved or retrieved from the underlying data source. The details of how the data is stored and retrieved is in the respective repository.

The interface in the repository pattern specifies

* What operations (i.e methods) are supported by the repository
* The data required for each of the operations i.e the parameters that need to be passed to the method and the data the method returns
* The repository interface contains what it can do, but not, how it does, what it can do
* The implementation details are in the respective repository class that implements the repository Interface
* Add new class in Api models: IUserRepository.cs
* Add new class in Api models : UserRepository.cs
* Tie the repository interface and the implementation classes together => Startup.cs : Configure services

services.AddScoped<IUserRepository, UserRepository>();

# CONTROLLER

* Create controller class that derives from the built-in ControllerBase class. ControllerBase is in Microsoft.AspNetCore.Mvc namespace.
* Controller is for MVC Web application
* ControllerBase is for MVC Web
* For both (MVC Web application and MVC Web), derive it from Controller class
* Add new class in controller in api: UsersController.cs

Graphical user interface

Description automatically generated with low confidence

* A GET request to the URI /api/employees returns the list of Employees
* A GET request to the URI /api/employees/1 should return the Employee whose ID is 1

When a new resource is created the following 3 things usually happen

* Return the http status code 201 to indicate that the resource is successfully created.
* Return the newly created resource. In our case, the newly created employee.
* Add a Location header to the response. The Location header specifies the URI of the newly created employee object.

# MODEL VALIDATION

* Install-Package System.ComponentModel.Annotations
* Api controller checks if the model is valid

# CALL REST API FROM ASP NET CORE BLAZOR

Create a service to call REST API: Add a folder with name Services to the Blazor web application project. Add the following 2 class files to this folder.

1. IEmployeeService.cs
2. EmployeeService.cs

* In ConfigureServices method of the Startup class register HttpClient Services using AddHttpClient method.
* ERROR: Watch out for the port number!
* We are using HttpClient class to call the REST API service.
* This class is in System.Net.Http namespace.
* HttpClient is injected into the EmployeeService using dependency injection.
* We have not registered HttpClient service with the dependency injection container yet. We will do that in just a bit.
* We are using httpClient.GetJsonAsync to call the REST API. This method is in Microsoft.AspNetCore.Blazor.HttpClient Nuget package. Install this package and do not forget to include the namespace Microsoft.AspNetCore.Components.

NOTE: NET.5 Install-Package System.Net.Http.Json -Version 5.0.0

Pass the REST API endpoint (api/employees) to httpClient.GetJsonAsync method.

* Properties that get their data from different database tables are called Navigation properties

# BUILDING BLAZOR FORM

* Add new Razor component in Pages: AddUser.razor
* Add new Class component in Pages: AddUserBase.cs
* We can connect controller directly from Blazor app, but using a Service is smarter
* Form validation
* Edit user: New folder Models in Web => EditUserModel.cs
* Mapping! => required when we passing data between different layers
* Configure in Startup.cs: services.AddAutoMapper(typeof(UserProfile));
* Add new class in Models: UserProfile.cs
* Inject AutoMapper in EditUserBase()
* Create new User: IUserServices:

Task<User> CreateUser(User newUser)