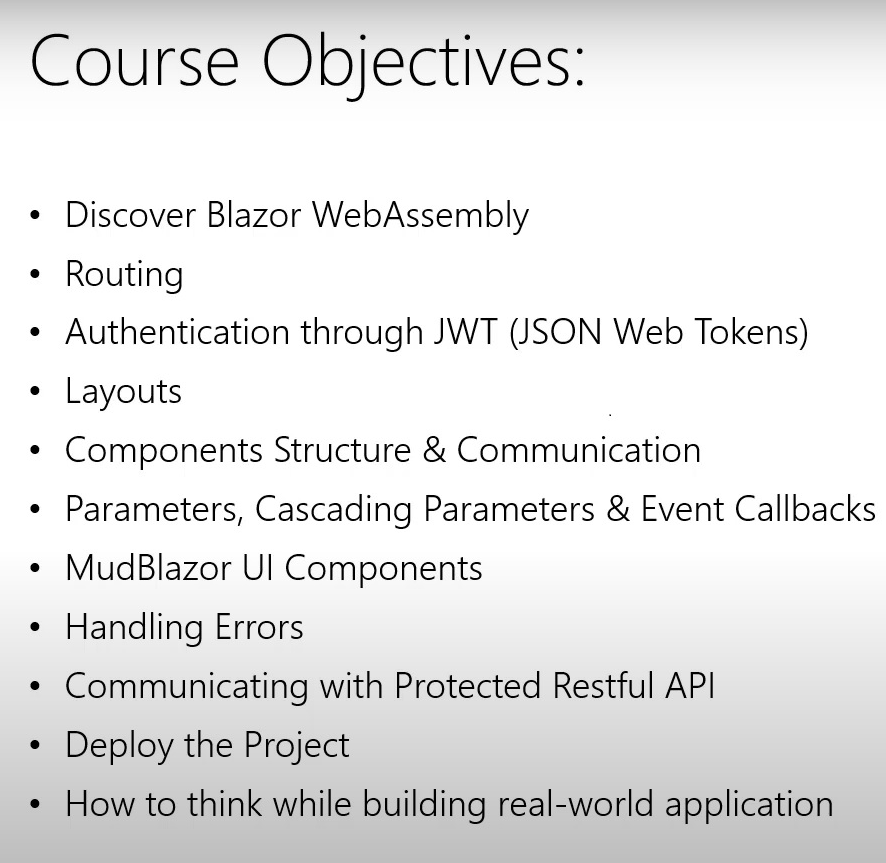
<https://www.youtube.com/playlist?list=PLFJQnCcZXWjsYlljYtU0oKTnJDcrLGgM2>

RegisterRequestValidatorRegisterRequestValidator

Contenido

[2. Apis 3](#_Toc89849790)

[3. Create the GitHub Repo & Initialize the Project with MudBlazor 6](#_Toc89849791)

[4. Discover the V2.0 API and Test It with 9](#_Toc89849792)

[Simular Login Postam 9](#_Toc89849793)

[Simular Registro de usuario en Postam 11](#_Toc89849794)

[Creación de un formulario de registro en Visual Studio 12](#_Toc89849795)

[5. Setup HttpClient & Create Authorization Messaging Handler 13](#_Toc89849796)

[6. Create Custom Authentication State Provider 16](#_Toc89849797)

[7. Implement the Auth Layout & Design the Login Form 20](#_Toc89849798)

[8. Implement the Logic of Login Form 28](#_Toc89849799)

[9. Implement LogOut Functionality and User Menu in App Bar 30](#_Toc89849800)

[10. Create Register Model With Validation using Fluent Validation 32](#_Toc89849801)

[11. Create HttpClientServices Project To Abstract the API Calls 35](#_Toc89849802)

[12. Design & Implement The Register Form 38](#_Toc89849803)

[13. Fix the Layout Of Register Form 42](#_Toc89849804)

[14. Fetch Plans 1 \_ Create the Models 43](#_Toc89849805)

[15. Fetch Plans 2. Create the Plan Cards List Component 47](#_Toc89849806)

# 2. Apis

<https://plannerapp-api.azurewebsites.net/>

<https://plannerapp-api.azurewebsites.net/swagger/index.html?urls.primaryName=V2>

Weather Web API  2.0 OAS3

[/swagger/v2/swagger.json](https://plannerapp-api.azurewebsites.net/swagger/v2/swagger.json)

**Auth**

**POST​/api​/v2​/Auth​/Register**

**POST​/api​/v2​/Auth​/Login**

**Plans**

**GET​/api​/v2​/Plans**

**POST​/api​/v2​/Plans**

**PUT​/api​/v2​/Plans**

**GET​/api​/v2​/Plans​/{id}**

**DELETE​/api​/v2​/Plans​/{id}**

**ToDos**

**GET​/api​/v2​/ToDos​/pendings**

**POST​/api​/v2​/ToDos**

**PUT​/api​/v2​/ToDos**

**PUT​/api​/v2​/ToDos​/Toggle​/{id}**

**DELETE​/api​/v2​/ToDos​/{id}**

**Schemas**

**RegisterRequest{**

|  |  |
| --- | --- |
| **email\*** | **string($email) *maxLength: 50 minLength: 0*** |
| **firstName\*** | **string *maxLength: 25 minLength: 0*** |
| **lastName\*** | **string *maxLength: 25 minLength: 0*** |
| **password\*** | **string *maxLength: 50 minLength: 5*** |
| **confirmPassword\*** | **string *maxLength: 50 minLength: 5*** |

**}**

**ApiResponse{**

|  |  |
| --- | --- |
| **message** | **string *nullable: true*** |
| **isSuccess** | **boolean** |

**}**

**ApiErrorResponse{**

|  |  |
| --- | --- |
| **message** | **string *nullable: true*** |
| **errors** | **[...]** |
| **isSuccess** | **boolean** |

**}**

**LoginRequest{**

|  |  |
| --- | --- |
| **email\*** | **string($email) *maxLength: 50 minLength: 0*** |
| **password\*** | **string *maxLength: 50 minLength: 5*** |

**}**

**AccessTokenResult{**

|  |  |
| --- | --- |
| **token** | **string *nullable: true*** |
| **expiryDate** | **string($date-time)** |

**}**

**AccessTokenResultApiResponse{**

|  |  |
| --- | --- |
| **value** | **{...}** |
| **message** | **string *nullable: true*** |
| **isSuccess** | **boolean** |

**}**

**ToDoItemDetail{**

|  |  |
| --- | --- |
| **id** | **string *nullable: true*** |
| **description\*** | **string** |
| **estimationDate** | **string($date-time) *nullable: true*** |
| **achievedDate** | **string($date-time) *nullable: true*** |
| **isDone** | **boolean** |
| **planId\*** | **string *maxLength: 50 minLength: 0*** |

**}**

**PlanDetail{**

|  |  |
| --- | --- |
| **id** | **string *nullable: true*** |
| **title\*** | **string *maxLength: 80 minLength: 0*** |
| **description** | **string *maxLength: 500 minLength: 0 nullable: true*** |
| **coverFile** | **string($binary) *nullable: true*** |
| **coverUrl** | **string *nullable: true*** |
| **toDoItems** | **[...]** |

**}**

**PlanDetailPagedList{**

|  |  |
| --- | --- |
| **totalPages** | **integer($int32) *readOnly: true*** |
| **page** | **integer($int32) *readOnly: true*** |
| **pageSize** | **integer($int32) *readOnly: true*** |
| **itemsCount** | **integer($int32) *readOnly: true*** |
| **records** | **[...]** |

**}**

**PlanDetailPagedListApiResponse{**

|  |  |
| --- | --- |
| **value** | **{...}** |
| **message** | **string *nullable: true*** |
| **isSuccess** | **boolean** |

**}**

**PlanDetailApiResponse{**

|  |  |
| --- | --- |
| **value** | **{...}** |
| **message** | **string *nullable: true*** |
| **isSuccess** | **boolean** |

**}**

**ToDoItemDetailPagedList{**

|  |  |
| --- | --- |
| **totalPages** | **integer($int32) *readOnly: true*** |
| **page** | **integer($int32) *readOnly: true*** |
| **pageSize** | **integer($int32) *readOnly: true*** |
| **itemsCount** | **integer($int32) *readOnly: true*** |
| **records** | **[...]** |

**}**

**ToDoItemDetailPagedListApiResponse{**

|  |  |
| --- | --- |
| **value** | **{...}** |
| **message** | **string *nullable: true*** |
| **isSuccess** | **boolean** |

**}**

**ToDoItemDetailApiResponse{**

|  |  |
| --- | --- |
| **value** | **{...}** |
| **message** | **string *nullable: true*** |
| **isSuccess** | **boolean** |

**}**

# 3. Create the GitHub Repo & Initialize the Project with MudBlazor

Software recomendado Gitkraken

Pasos a seguir:

1. Crear 1 nuevo repositorio en GitHub: plannerapp-v2, add readme file add git ignore del tipo Visual Studio.
2. En Github – code – copiar la url del repositorio.
3. Visual Studio 2022 clonar el repositorio anterior y añadiremos un nuevo proyecto en 1 nueva subcarpeta (src) que se creará en la carpeta del proyecto clonado anteriormente, el tipo de proyecto será del tipo Blazor para WebAssembly – Net Framework 6 – Https y Aplicación web progresiva marcados.
4. Instalar MudBlazor desde el administrador de paquetes Nuget – versión 6.
5. Instalar Blazored.LocalStorage
6. Para utilizar **MudBlazor** seguiremos estas recomendaciones:

<https://mudblazor.com/getting-started/installation#manual-install>

* En \_**Imports.razor** añadiremos: @using MudBlazor
* Add CSS & Font references en **index.html**:

<link href="https://fonts.googleapis.com/css?family=Roboto:300,400,500,700&display=swap" rel="stylesheet" />

<link href="\_content/MudBlazor/MudBlazor.min.css" rel="stylesheet" />

En este mismo archivo quitaremos la referencia de **Bootstrap**:

<link href="css/bootstrap/bootstrap.min.css" rel="stylesheet" />

Y añadiremos por el final:

<script src="\_content/MudBlazor/MudBlazor.min.js"></script>

* Registramos los servicios en **program.cs**:

using MudBlazor.Services;

builder.Services.AddMudServices();

* Añadir los componentes a MainLayout.razor:

<MudThemeProvider/>

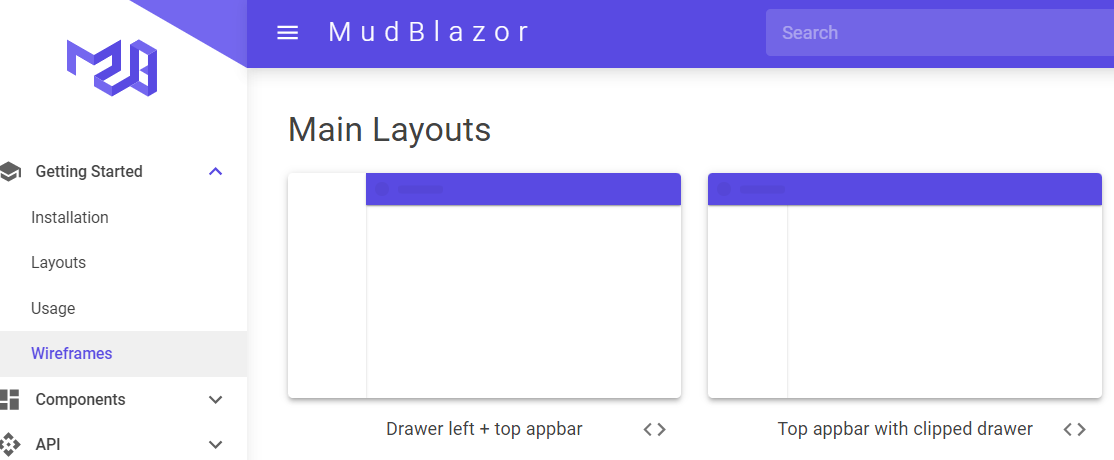
<MudDialogProvider/>

<MudSnackbarProvider/>

1. Añadir la plantilla de diseño desde MudBlazor:

<https://mudblazor.com/getting-started/wireframes#main-layouts>

En wireframes seleccionaremos la 2ª plantilla, en el símbolo <> se nos mostrará en código que copiaremos en nuestro MainLayout.razor sustituyendo al código anterior:



@inherits LayoutComponentBase

@\*<MudThemeProvider /> \*@

<**MudLayout**>

<**MudAppBar** **Elevation**="1">

<**MudIconButton** **Icon**="@Icons.Material.Filled.Menu" **Color**="Color.Inherit" **Edge**="Edge.Start" **OnClick**="@((e) => DrawerToggle())" />

<**MudText** **Typo**="Typo.h5" **Class**="ml-3">Application</**MudText**>

<**MudSpacer** />

<**MudIconButton** **Icon**="@Icons.Material.Filled.MoreVert" **Color**="Color.Inherit" **Edge**="Edge.End" />

</**MudAppBar**>

<**MudDrawer** @bind-Open="\_drawerOpen" **ClipMode**="DrawerClipMode.Always" **Elevation**="2">

<**NavMenu** /> //por defecto viene comentado, descomentar para mostrar nuestro menú

</**MudDrawer**>

<**MudMainContent**>

@Body

</**MudMainContent**>

</**MudLayout**>

@code {

bool \_drawerOpen = true;

void DrawerToggle()

{

\_drawerOpen = !\_drawerOpen;

}

}

<**MudThemeProvider**/>

<**MudDialogProvider**/>

<**MudSnackbarProvider**/>

En MundoBlazor – components – AppBar podríamos personalizar y adaptar nuestra barra de menús:

<https://mudblazor.com/components/appbar#api>

Para ver las propiedades, eventos, … de este componente:

<https://mudblazor.com/api/appbar#pages>

1. Probar el proyecto con el depurador de visual studio o desde 1 terminal de Windows desde la carpeta del proyecto en el explorador de Windows – botón de la derecha – abrir en terminal Windows en vista previa (powershell) – dotnet watch run

# 4. Discover the V2.0 API and Test It with

Depurar con Postam: <https://www.postman.com/>

## Simular Login Postam

1. Crear 1 nueva solicitud desde nuestro Workspace - botón + del tipo Post:

En <https://plannerapp-api.azurewebsites.net> (consultar las apis, el sitio de ejemplo está publicado en Azure). url del post: <https://plannerapp-api.azurewebsites.net/api/v2/auth/login>

Marcar en Body – **raw**, también **JSON**

En la documentación de las apis **LoginRequest** necesitará 1 email + password por lo que escribiremos en body – raw:

{

"email":"test@plannerapp.com",

"password":"Test.123"

}

Pulsar en Send

Respuesta:

{

    "value": {

        "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..HVZ54L8\_7Km4yvyjzFcgBK3SxHSEpB3\_pig5VlX0CAs",

        "expiryDate": "2021-12-21T07:56:02Z"

    },

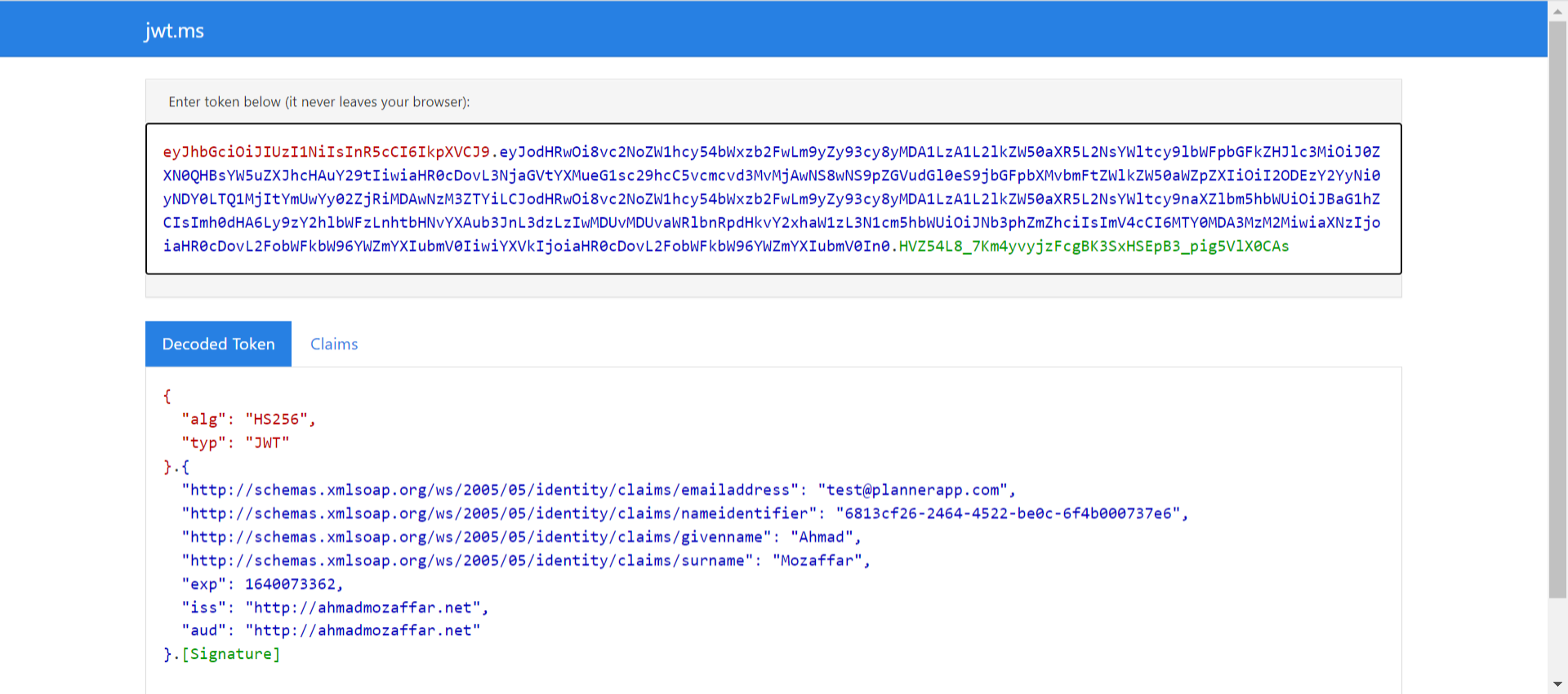
    "message": "Access token retrieved successfully",

    "isSuccess": **true**

}

Obtendremos el token + fecha de vencimiento.

1. Copiar el token en la url: <https://jwt.ms/> :



El token es un conjunto de valores claves que representa al usuario actual o al usuario de inicio de la aplicación, dentro de este token tenemos email, id, nombre, apellido, …

El funcionamiento de validación ocurrirá de la siguiente manera:

* El usuario solicita un token de acceso al registro después de logearse.
* Consultaremos el token almacenado en LocalStorage.
* Posteriormente la verificación de inicio de sesión se realizará a través de 1 proveedor de autenticación que verificará si hay 1 token, posteriormente lo decodificará, obtiene los tokens, crea 1 usuario y lo entrega a la aplicación Blazor para saber que el usuario a iniciado sesión.

## Simular Registro de usuario en Postam

1. Crear 1 nueva solicitud desde nuestro Workspace - botón + del tipo Post, previamente consultar las api en <https://plannerapp-api.azurewebsites.net/> , seleccionar v2, hacer un click en el post de register para copiar la estructura de ejemplo de datos:

{

"email": "user@example.com",

"firstName": "string",

"lastName": "string",

"password": "string",

"confirmPassword": "string"

}

En la url del post: url del post: <https://plannerapp-api.azurewebsites.net/api/v2/Auth/Register> (consultar las apis, el sitio de ejemplo está publicado en Azure). Marcar en Body – raw, también JSON

Seleccionar raw en body y JSON.

{

  "email": "user6@example.com",

  "firstName": "Antonio",

  "lastName": "Vargas",

  "password": "Ak.123",

  "confirmPassword": "Ak.123"

}

{

    "message": "User created successfully!",

    "isSuccess": **true**

}

## Creación de un formulario de registro en Visual Studio

1. Crear el modelo en 1 biblioteca de clases compartida para que se pueda usar desde cualquier cliente:

Botón derecho en la solución – agregar – nuevo proyecto – Biblioteca de clases de c# - PlannerApp.Shared – Net 6 – Crear

1. Borrar class1.cs, crear las carpetas Responses y Models.
2. Crear la clase ApiResponse.cs en la carpeta Responses que contendrá el objeto de respuesta, se puede consultar la estructura en el api de Login:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace PlannerApp.Shared.Responses {

public class ApiResponse {

public string Message { get; set; }

public bool IsSuccess { get; set; }

}

public class ApiResponse<T> : ApiResponse {

public T Value { get; set; }

}

}

1. Crear la clase en la carpeta Responses para el tratamiento de errores, en vez de escribirla podemos copiar de la api el ejemplo y pegar en Visual Studio desde Edición – Pegado especial - Pegar JSON como clases:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace PlannerApp.Shared.Responses

{

public class ApiErrorResponse

{

public string Message { get; set; }

public string[] Errors { get; set; }

public bool IsSuccess { get; set; }

}

}

# Setup HttpClient & Create Authorization Messaging Handler

<https://www.netmentor.es/entrada/implementar-httpclient>

1. Crearemos 1 instancia del cliente http para que se pueda inyectar en cualquier componente o servicio c#, de forma predeterminada la plantilla que se crea con el proyecto en program.cs es:

builder.Services.AddScoped(sp => new HttpClient { BaseAddress = new Uri(builder.HostEnvironment.BaseAddress) });

La dirección base es la misma que nuestra aplicación cliente que no tiene 1 API, por lo que no es útil, lo cambiaremos para que acepte nuestra API porque necesitamos tener 1 forma mas eficiente de administrar nuestras instancias de http client por si queremos implementar Microsoft Graph, por lo que necesitamos implementar varias instancias, también si queremos que el cliente http haga alguna cosa automática por nosotros, como cada vez que enviamos 1 solicitud para ir al almacenamiento local para obtener el token de acceso, configúrelo en el encabezado y envíelo sin la necesidad de hacer esto por nosotros mismos cada vez y para hacer esto necesitamos usar algo llamado **manejador de mensajes** donde podremos adjuntar ese manejador de mensajes a un cliente http, así que la solicitud antes de que pase pasa por todos los manejadores de mensajes disponibles o adjúntela para este y simplemente configúrelo y cada manejador de mensajes va a hacer algo en nuestro proyecto, solo tendremos un manejador de mensajes que se va a configurar el token de acceso en el encabezado de la solicitud, así para poder configurarlo usaremos el servicio definido en program.cs

1. Para disponer de AddHttpClient hay que instalar el paquete de nuget:

**System.Extensions.http**

1. Para configurar la instancia y darle un nombre:

builder.Services.AddHttpClient("PlannerApp.Api", client => {

//establecer la dirección base de nuestro api, será bastante simple adjuntar un controlador de mensajes

client.BaseAddress = new Uri("https://plannerapp-api.azurewebsites.net");

//adjuntamos el controlador de mensajes al cliente http, cada vez que se envíe una solicitud, post o se elimine algo pasará por aquí y llamará al método asíncrono del controlador

}).AddHttpMessageHandler<AuthorizationMessageHandler>();

1. Registrar el servicio de almacenamiento en program.cs:

builder.Services.AddBlazoredLocalStorage();

1. Crear un controlador de mensajes de autorización: AuthorizationMessageHandler.cs

using Blazored.LocalStorage;

using System;

using System.Net.Http;

using System.Threading;

using System.Threading.Tasks;

namespace PlannerApp {

public class AuthorizationMessageHandler : DelegatingHandler {

//registrar el servicio de almacenamiento local

private readonly ILocalStorageService \_storage;

public AuthorizationMessageHandler(ILocalStorageService storage) {

//mirar el servicio de almacenamiento local

\_storage = storage;

}

protected async override Task<HttpResponseMessage> SendAsync(HttpRequestMessage request, CancellationToken cancellationToken) {

//manipularemos la respuesta para recuperar el token de acceso desde el almacenamiento local en el caso de que esté allí y configurarlo en el encabezado.

//Comprobar en el almacenamiento local si hay algún token de acceso que contiene la clave asíncrona access\_token

if (await \_storage.ContainKeyAsync("access\_token")) {

//lo obtenemos y lo ponenmos en el encabezado

var token = await \_storage.GetItemAsStringAsync("access\_token");

request.Headers.Authorization = new System.Net.Http.Headers.AuthenticationHeaderValue("Bearer", token);

}

Console.WriteLine("Authorization Message Handler Called");

//enviar asincronamene la respuesta y el token de cancelación

return await base.**SendAsync**(request, cancellationToken);

}

}

}

1. Registrar el controlador de mensajes en program.cs:

builder.Services.AddTransient<AuthorizationMessageHandler>();

1. HttpClientFactory para controlar las instancias del cliente http como se explica aquí:

<https://www.netmentor.es/entrada/implementar-httpclient>

builder.Services.AddScoped(sp => sp.GetService<IHttpClientFactory>().CreateClient("PlannerApp.Api"));

1. Prueba en FetchData.razor:

@inject HttpClient Http

forecasts = await Http.GetFromJsonAsync<WeatherForecast[]>("sample-data/weather.json");

Se ejecutar el controlador de mensajes porque todas las solicitudes pasan por el y se verá un mensaje en la consola ???

<https://localhost:7097/fetchdata>

# Create Custom Authentication State Provider

Creación de 1 mecanismo de autenticación, así nuestra aplicación sabrá quien está autenticado, conectado, que es, propiedades del usuario, roles, …, autorización basada en reglas.

Para obtener el estado de autenticación, si hay un token buscaremos todos sus claims, … Microsoft proporciona una clase base llamada proveedor de datos de autenticación, posteriormente Blazor sabrá que el inicio de sesión es correcto y cuáles son sus datos.

1. Crear una nueva clase abstracta JwtAuthenticationStateProvider.cs

<https://www.netmentor.es/entrada/clase-abstracta>

Implementaremos un mecanismo de autenticación que se basa en el token de acceso existente de LocalStorage.

using Blazored.LocalStorage;

using Microsoft.AspNetCore.Components.Authorization;

using System.IdentityModel.Tokens.Jwt;

using System.Security.Claims;

using System.Threading.Tasks;

namespace PlannerApp

{

//ctrl + click en JwtAuthenticationStateProvider implementar clase abstracta

public class JwtAuthenticationStateProvider : AuthenticationStateProvider {

private readonly ILocalStorageService \_storage;

//inyectar el servicio de almacenamiento local

public JwtAuthenticationStateProvider(ILocalStorageService storage) {

\_storage = storage;

}

//en una clase abstracta debemos sobrescribir todos sus métodos

public async override Task<AuthenticationState> GetAuthenticationStateAsync() {

//si hay un token de acceso, verificación de la palabra de acceso

if (await \_storage.ContainKeyAsync("access\_token")) {

// The user is logged in

var tokenAsString = await \_storage.GetItemAsStringAsync("access\_token");

//ctrl + punto **Install package System.IdentityModel.TokensJwt**

var tokenHandler = new JwtSecurityTokenHandler();

//deberíamos hacer lo mismo que jkt.ms hizo al decodificar ese token y cambiar //los claims

var token = tokenHandler.ReadJwtToken(tokenAsString);

var identity = new ClaimsIdentity(token.Claims, "Bearer");

//crear el objeto de usuario

var user = new ClaimsPrincipal(identity);

//estado de autenticación

var authState = new AuthenticationState(user);

//evento disponible cuando se produce un cambio de estado, como parámetro la //tarea del estado

NotifyAuthenticationStateChanged(Task.FromResult(authState));

return authState;

}

//creación de 1 nuevo claim, inicialmente vacío, significa que no hay identidad, el usuario no ha iniciado sesión

return new AuthenticationState(new ClaimsPrincipal());

// Empty claims principal means no identity and the user is not logged in

}

}

}

1. Registrar en program.cs:

//permite redirigir a la página de inicio de sesión al usuario si no está autorizado:

builder.Services.AddAuthorizationCore();

//instalar el paquete **Microsoft.AspNetCore.Components.Authorization**

builder.Services.AddScoped<AuthenticationStateProvider, JwtAuthenticationStateProvider>();

1. Cambios en \_imports.razor, añadir:

@using Microsoft.AspNetCore.Components.Authorization

@using Microsoft.AspNetCore.Authorization

1. Cambios en el enrutador app.razor (parámetro de cascada), es un valor que el componente padre tiene y podría ser accedido desde el hijo, la autenticación está disponible para todos los hijos:

<**CascadingAuthenticationState**>

<**Router** **AppAssembly**="@typeof(Program).Assembly" **PreferExactMatches**="@true">

<**Found** **Context**="routeData">

<**AuthorizeRouteView** **RouteData**="@routeData" **DefaultLayout**="@typeof(MainLayout)">

<**NotAuthorized**>

//redirigir al usuario al inicio de sesión

<h3>You don't have access for this page</h3>

</**NotAuthorized**>

</**AuthorizeRouteView**>

</**Found**>

<**NotFound**>

<**LayoutView** **Layout**="@typeof(MainLayout)">

<p>Sorry, there's nothing at this address.</p>

</**LayoutView**>

</**NotFound**>

</**Router**>

</**CascadingAuthenticationState**>

1. Prueba de autorización en Fetchdata.razor: @attribute [Authorize] y ya está protegido.
2. Prueba con Postman, vamos a generar un token y volveremos a probar Fetchdata.razor

Crear un nuevo Post con la url: <https://plannerapp-api.azurewebsites.net/api/v2/auth/login>

Body – Raw:

{

"email":"test@plannerapp.com",

"password":"Test.123"

}

Pulsar en Send

Respuesta:

{

    "value": {

        "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..HVZ54L8\_7Km4yvyjzFcgBK3SxHSEpB3\_pig5VlX0CAs",

        "expiryDate": "2021-12-21T07:56:02Z"

    },

    "message": "Access token retrieved successfully",

    "isSuccess": **true**

}

Desde Chrome – Herramientas del desarrollador – Application – Key, añadir Access\_token y pegar el token obtenido anteriormente con Postman en Value, ahora al refrescar fetchdata debería funcionar.

1. En index.razor:

<**AuthorizeView**>

<**Authorized**>

<h2>Welcome</h2>

//ó

<h2>Welcome

@context.User.FindFirst(System.Security.Claims.ClaimTypes.GivenName).Value

</h2>

</**Authorized**>

<**NotAuthorized**>

<h2>Por favor logueese</h2>

//o utilizando MudBlazor:

<**MudLink** **Href**="/authentication/login">Login</**MudLink**>

</**NotAuthorized**>

</**AuthorizeView**>

Esta API <https://plannerapp-api.azurewebsites.net/api/v2/auth/login> obtiene el token existente en su navegador, hay que eliminar el token creado anteriormente manualmente, desde Chrome herramientas del desarrollador – Application – suprimir Key Access\_token

# 7. Implement the Auth Layout & Design the Login Form

Implementar la interfaz de usuario para que el usuario pueda iniciar sesión en las credenciales y cuando encajemos con el token, lo almacenamos en LocalStorage y luego llamaremos al estado de autenticación de obtención asíncrono para actualizar el estado de la aplicación.

Si el usuario no ha iniciado la sesión debería ser redirigido hacia la página de inicio.

1. Crear un nuevo diseño (MubBlazor).
2. Implementar el formulario.
3. Implementar el cliente http.
4. Crear un nuevo diseño (MubBlazor):

En MainLayout.razor está el diseño principal con la plantilla obtenida de MubBlazor:

@inherits LayoutComponentBase

@\*<MudThemeProvider /> \*@

<MudLayout>

    <MudAppBar Elevation="1">

        <MudIconButton Icon="@Icons.Material.Filled.Menu" Color="Color.Inherit" Edge="Edge.Start" OnClick="@((e) => DrawerToggle())" />

        <MudText Typo="Typo.h5" Class="ml-3">Application</MudText>

        <MudSpacer />

        <MudIconButton Icon="@Icons.Material.Filled.MoreVert" Color="Color.Inherit" Edge="Edge.End" />

    </MudAppBar>

//cuadro de dialogo

    <MudDrawer @bind-Open="\_drawerOpen" ClipMode="DrawerClipMode.Always" Elevation="2">

       <NavMenu />

    </MudDrawer>

    <MudMainContent>

        @Body

    </MudMainContent>

</MudLayout>

@code {

    bool \_drawerOpen = true;

    void DrawerToggle()    {

        \_drawerOpen = !\_drawerOpen;

    }

}

<MudThemeProvider/>

<MudDialogProvider/>

<MudSnackbarProvider/>

Cambiaremos este diseño creando un nuevo componente Layout desde Shared – Agregar componente – Razor component – **AuthLayout.razor**:

No mostraremos ningún cuadro de diálogo, bocadillo si creo?, el proveedor de temas se usará para modo oscuro, copiar Layout de MainLayout.razor y personalizarlo:

@inherits LayoutComponentBase

<MudLayout>

    <MudAppBar Elevation="1">

        <MudText Typo="Typo.h5" Class="ml-3">PlannerApp</MudText>

        <MudAppBarSpacer />

        <MudIconButton Icon="@Icons.Material.Filled.MoreVert" Color="Color.Inherit" Edge="Edge.End" />

    </MudAppBar>

    <MudMainContent>

        <MudGrid Justify="Justify.Center" Class="mt-3">

            <MudItem sm="12" md="4" lg="3">

                @Body

            </MudItem>

        </MudGrid>

    </MudMainContent>

</MudLayout>

<MudThemeProvider />

@code {

}

1. Implementar el formulario.:

Desde mudblazor copiaremos diseño de formulario que no ocupe toda la página, compuesto de 1 tarjeta en el medio con sombra, desde Mudblazor – Components – Grid tenemos varios diseños y en Paper podemos posteriormente ensamblar más código.

Desde Card testa compuesto de Título + 1 parte de acción + 1 contenido, mientras que el papel de barro es una caja con sombra que es el que se utilizará.

//El centro de las columnas deben de estar en el centro

<MudGrid Justify="Justify.Center" Class="mt-3"> //mt-3 conjunto de clases disponibles por ModBlazor, se pueden consultar en MudBlazor – features – spacing, utilizaremos clases en vez de pixeles

//Agregar 1 elemento, en la pantalla pequeña no necesitaremos espacio por la //izquierda sm=”12” se cogerán las 12 columnas, pantalla grande la 3 parte:

//lg="3"

            <MudItem sm="12" md="4" lg="3">

                @Body

            </MudItem>

        </MudGrid>

* El **diseño principal** no está en la página de inicio index.razor, sino que, como se define en **App.Razor**, está en MainLayout:

<Found Context="routeData">

            <AuthorizeRouteView RouteData="@routeData" DefaultLayout="@typeof(MainLayout)">

* Implementar el **inicio de sesión**, crear una carpeta en Pages – Authentication, en la que se crearán 2 componentes de razor: **Login.razor** y Register.razor, agregaremos el diseño de AuthLayout

Login.razor:

**@layout AuthLayout**  // diseño heredado de AuthLayout.razor

@page "/authentication/login" //enrutador

<LoginForm />

@code {

}

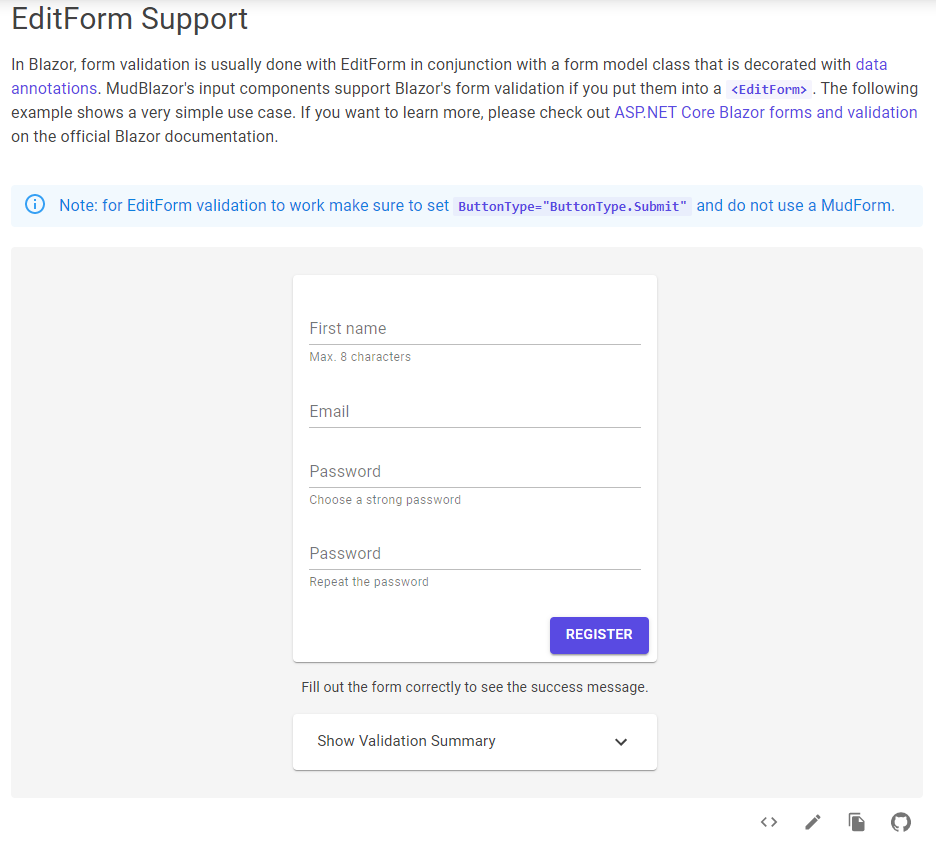
* Las páginas están vacías en las que vamos añadiendo componentes, estos componentes se crearán en 1 nueva carpeta en la raíz que llamaremos Components + 1 subcarpeta Authentication.

En **\_imports.razor**, añadir: @using PlannerApp.Components

Nuestro primer componente en Components\Authentication será **LoginForm.razor**:

En mudblazor – components – card hay ejemplos de tarjetas.

El código del componente lo obtendremos de Componentes – Form Inputs & Controls – Form, escogeremos el módelo: EditForm Support



Seleccionaremos el símbolo <> para obtener el código que podremos copiar a nuestro componente adaptándolo posteriormente a nuestro modelo de datos:

**LoginForm.razor:**

@namespace PlannerApp.Components

<EditForm Model="\_model" OnValidSubmit="LoginUserAsync">

    <DataAnnotationsValidator />

    <MudCard>

//Copiar del componente card

        <MudCardHeader>

            <CardHeaderContent>

                <MudText Typo="Typo.h6">Welcome to PlannerApp v2.0</MudText>

            </CardHeaderContent>

        </MudCardHeader>

        <MudCardContent>

            <MudTextField Label="Email" HelperText="Required" @bind-Value="@\_model.Email" />

            <MudTextField InputType="InputType.Password" Label="Password" HelperText="Required" @bind-Value="@\_model.Password" />

            <ValidationSummary />

            @if (!string.IsNullOrWhiteSpace(\_errorMessage))   {

//mostrar mensajes de error, componentes de MudRazor – Alert

//class=”my-2” establece espacio

                <MudAlert Severity="Severity.Error" Class="my-2" Variant="Variant.Filled">@\_errorMessage</MudAlert>

            }

        </MudCardContent>

        <**MudCardActions**>

//deshabilitar los botones para que no se pueda hacer clic varias veces

           <MudButton ButtonType="ButtonType.Submit" Variant="Variant.Filled" Color="Color.Primary" Disabled="\_isBusy">Login</MudButton>

//class=”m1-1” – margen 1

          <MudButton Variant="Variant.Outlined" Color="Color.Secondary" Class= "ml-1" Disabled="\_isBusy" OnClick="RedirectToRegister">Create Account</MudButton>

        </MudCardActions>

    </MudCard>

</EditForm>

@code {

}

* **Crear** una **clase** para no escribir código dentro del componente razor creado anteriormente, blazor permite clases parciales por lo que es mejor así, con el mismo nombre y en la misma carpeta **LoginForm.cs**:

using Blazored.LocalStorage;

using Microsoft.AspNetCore.Components;

using Microsoft.AspNetCore.Components.Authorization;

using **PlannerApp.Shared.Models**;

using PlannerApp.Shared.Responses;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Net.Http;

using System.Net.Http.Json;

using System.Threading.Tasks;

namespace PlannerApp.Components{

//clase parcial que hereda de ComponentBase

    public partial class LoginForm : ComponentBase    {

        [Inject]

        public HttpClient HttpClient { get; set; }

        [Inject]

        public NavigationManager Navigation { get; set; }

        [Inject]

        public AuthenticationStateProvider AuthenticationStateProvider { get; set; }

//almacenar la respuesta que llegue del servidor

        [Inject]

        public ILocalStorageService Storage { get; set; }

// \_model crea 1 instancia de nuestro modelo: LoginRequest

        private LoginRequest \_model = new LoginRequest();

        private bool \_isBusy = false;

        private string \_errorMessage = string.Empty;

        private async Task LoginUserAsync()  {

            \_isBusy = true;

            \_errorMessage = string.Empty; //cada vez que se hace inicio de sesión

//enviar los datos al servidor, coge el objeto \_**model** con los datos //del formulario traduciéndolos a JSon

            var response = await HttpClient.PostAsJsonAsync("/api/v2/auth/login", \_model);

if (response.IsSuccessStatusCode)   {

//la respuesta del api devuelve un objeto del tipo de inicio de sesión //(LoginResult : Message , IsSuccess, Value), ejemplo de respuesta:

**{**

**"value": {**

**"token": "string",**

**"expiryDate": "2021-11-27T08:36:34.873Z"**

**},**

**"message": "string",**

**"isSuccess": true**

**}**

                var result = await response.Content.ReadFromJsonAsync<ApiResponse<LoginResult>>();

                // Store it in local storage, almacenaremos el token en local

// crear la variable y fecha de expiración

                await Storage.SetItemAsStringAsync("access\_token", result.Value.Token);

                await Storage.SetItemAsync<DateTime>("expiry\_date", result.Value.ExpiryDate);

//notificar a la aplicación que el inicio de sesión ha cambiado

                await AuthenticationStateProvider.GetAuthenticationStateAsync();

                Navigation.NavigateTo("/");

            }

            else {

//capturamos la respuesta del error del API,lee el contenido HTTP y //devuelve el valor que resulta de deserializar el contenido como JSON //en una operación asincrónica.

//Este código se utilizará como estándar en todas las respuestas erroneas

                var errorResult = await response.Content.ReadFromJsonAsync<ApiErrorResponse>();

                \_errorMessage = errorResult.Message;

            }

            \_isBusy = false;

        }

        private void RedirectToRegister()   {

            Navigation.NavigateTo("/authentication/register");

        }

}

}

* Crear 1 **modelo de datos** (clase) de inicio de solicitud en PlannerApp.Shared – Models que se llamará: **LoginRequest**.cs:

Importante: Para poder utilizar en PlannerApp, desde este proyecto en Dependencias – **Agregar dependencia** y marcar PlannerApp.Shared.

using System;

using System.Collections.Generic;

using System.ComponentModel.DataAnnotations;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace PlannerApp.Shared.Models {

    public class LoginRequest    {

        [Required]

        [EmailAddress]

        public string Email { get; set; }

        [Required]

        [StringLength(20, MinimumLength = 6)]

        public string Password { get; set; }

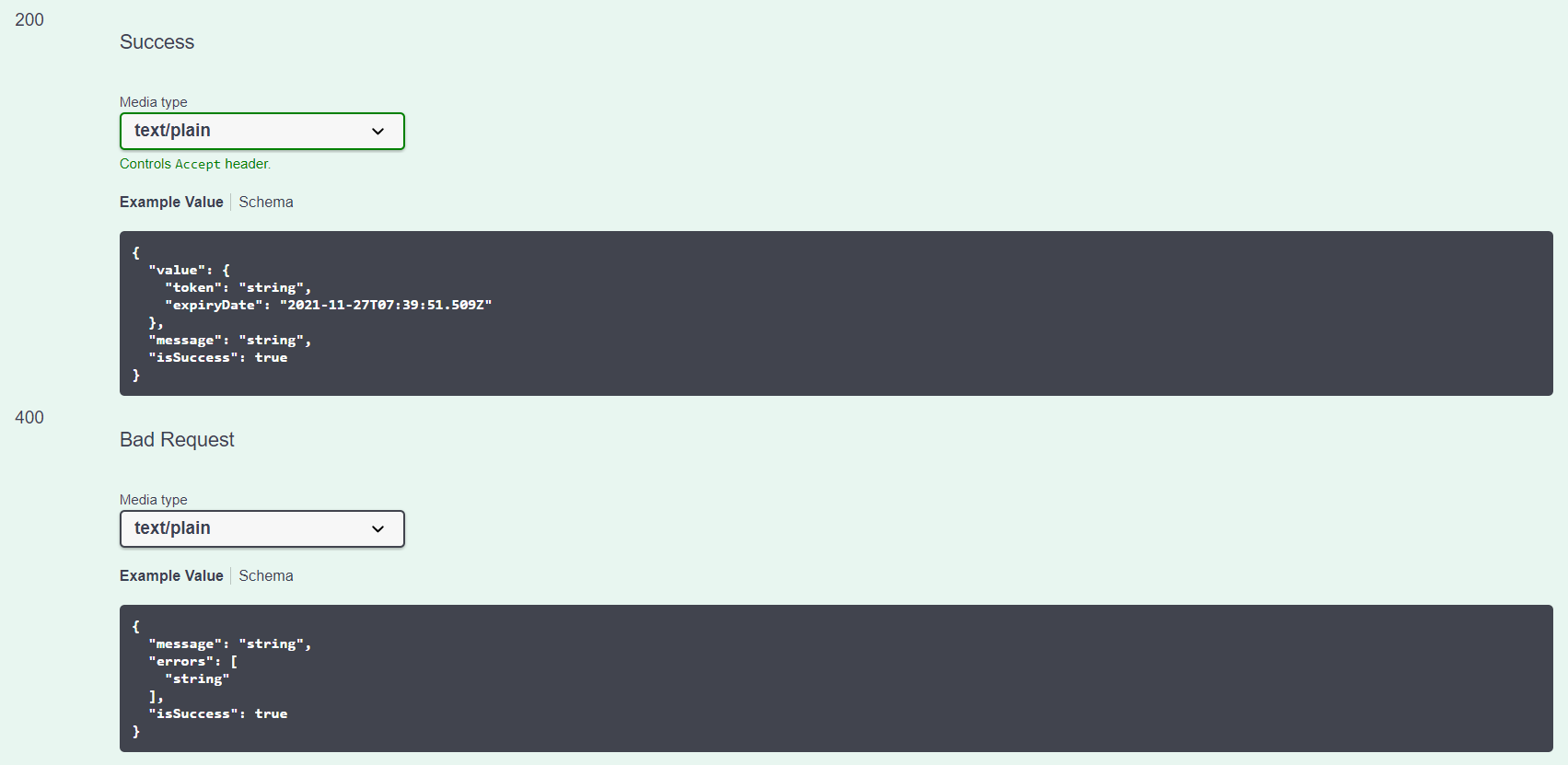
    }

}

# 8. Implement the Logic of Login Form

Implementar la lógica del formulario creado anteriormente y comenzar a enviar la solicitud al servidor para recoger el intento de inicio de sesión.

Tenemos 2 tipos de respuestas globales a la aplicación, en la referencia de la API en POST​/api​/v2​/Auth​/Login:



1. Crear la clase con el modelo de datos en PlanerApp.Shared (LoginResult.cs):

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace PlannerApp.Shared.Models {

public class LoginResult {

public string Token { get; set; }

public DateTime ExpiryDate { get; set; }

}

}

1. En index.razor:

@page "/"

<h1>Hello, world!</h1>

Welcome to your new app.

<**SurveyPrompt** **Title**="How is Blazor working for you?" />

<**AuthorizeView**>

<**Authorized**>

<h2>Welcome</h2>

</**Authorized**>

<**NotAuthorized**>

//enlace para validarse en nuestra aplicación

<**MudLink** **Href**="/authentication/login">Login</**MudLink**>

</**NotAuthorized**>

</**AuthorizeView**>

Posteriormente implementaremos la funcionalidad que si el usuario no está validado será redirigido automáticamente a la página de inicio.

Si la validación es correcta se tiene acceso al objeto de usuario.

# 9. Implement LogOut Functionality and User Menu in App Bar

Cerrar sesión eliminará el token de acceso de LocalStorage, refrescaremos el estado de sesión e iremos a la página de inicio.

Mostraremos en la parte superior un menú desplegable del usuario conectado para que pueda cerrar sesión:

1. Mudrazor – Components – Menu – buscar un diseño (por ejemplo account en custom activator):

<MudMenu FullWidth="true" Disabled="true">

<ActivatorContent>

<MudChip Icon="@Icons.Material.Filled.Person" Disabled="true" Color="Color.Primary">Disabled</MudChip>

</ActivatorContent>

<ChildContent>

<MudMenuItem>Usage</MudMenuItem>

<MudMenuItem>Sign Out</MudMenuItem>

</ChildContent>

</MudMenu>

Nota: en el ejemplo del curso han copiado el tipo Offset botón que ya no existe, en el apartado placament diferentes tipos de dirección para el menú desplegable, podremos copiar su código, ejemplo: Option="Origin.TopRight”

1. Crear un nuevo componente: LoginDisplay.razor en PlannerApp.razor:

@inject Blazored.LocalStorage.ILocalStorageService Storage

@inject AuthenticationStateProvider AuthenticationStateProvider

@inject NavigationManager Navigation

@using System.Security.Claims

//verificación de inicio de sesión correcto

<**AuthorizeView**>

<**Authorized**>

//icono puntos suspensivos

<**MudMenu** **Icon**="@Icons.Filled.Person" **Color**="Color.Secondary" **Direction**="Direction.Right" **OffsetY**="true">

//datos del usuario, buscaremos el 1 claim

<**MudMenuItem**>Welcome @context.User.FindFirst(ClaimTypes.GivenName).Value</**MudMenuItem**>

<**MudMenuItem** **OnClick**="**LogOutAsync**">LogOut</**MudMenuItem**>

</**MudMenu**>

</**Authorized**>

<**NotAuthorized**>

<**MudIconButton** **Icon**="@Icons.Filled.Login" **Color**="Color.Secondary" **OnClick**='() => Navigation.NavigateTo("/authentication/**login**")' />

</**NotAuthorized**>

</**AuthorizeView**>

@code {

private async Task LogOutAsync() {

//eliminar token de LocalStorage

await Storage.RemoveItemAsync("access\_token");

await Storage.RemoveItemAsync("expiry\_date");

//refrescar estado de autenticación

await AuthenticationStateProvider.GetAuthenticationStateAsync();

Navigation.NavigateTo("/authentication/login");

}

}

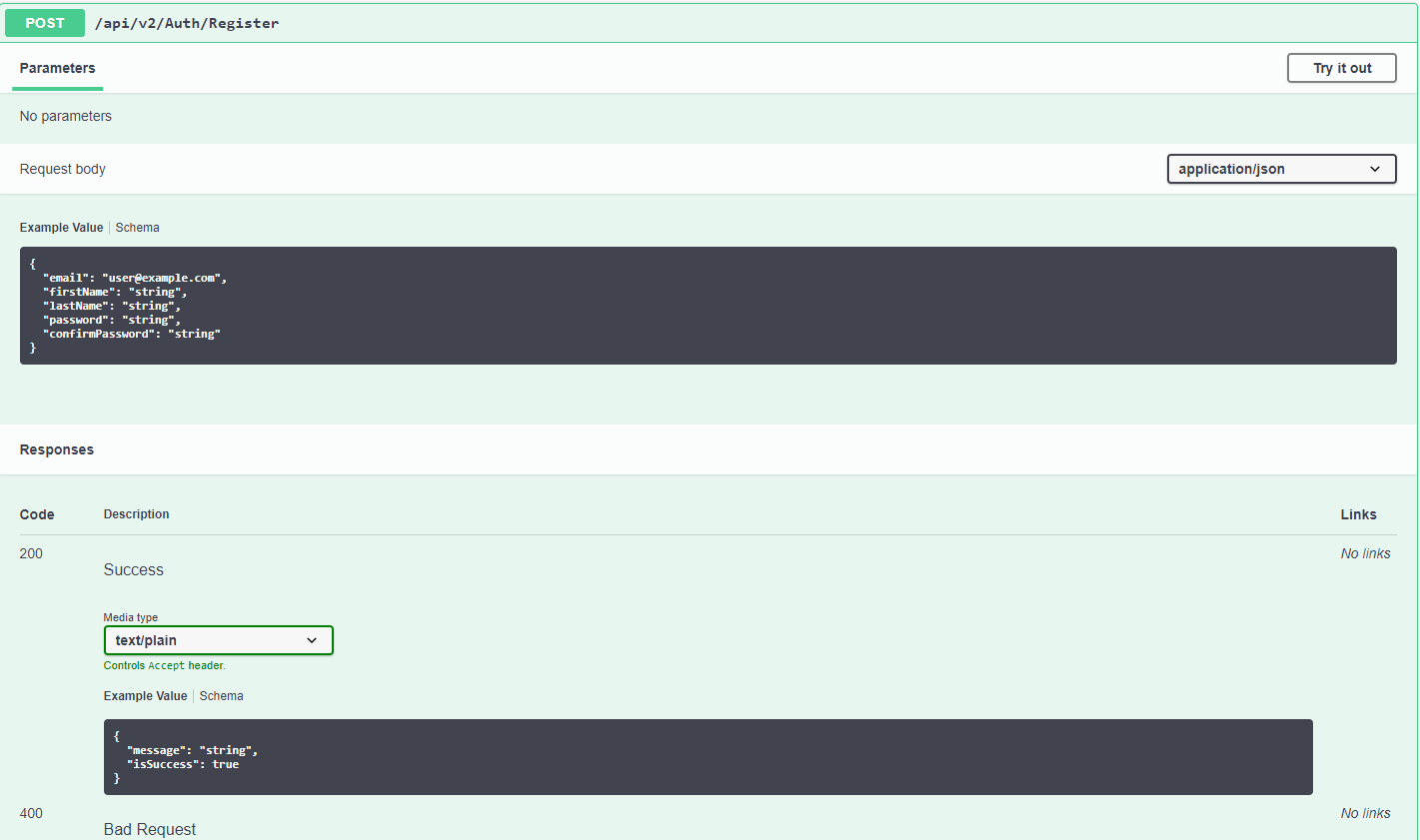
1. En MainLayout.razor añadiremos el componente creado anteriormente:

<**MudSpacer** />

<**LoginDisplay** />

# 10. Create Register Model With Validation using Fluent Validation

Volvemos a las API registro:



El objeto de solicitud de registro tiene el siguiente esquema:

**RegisterRequest{**

|  |  |
| --- | --- |
| **email\*** | **string($email) *maxLength: 50 minLength: 0*** |
| **firstName\*** | **string *maxLength: 25 minLength: 0*** |
| **lastName\*** | **string *maxLength: 25 minLength: 0*** |
| **password\*** | **string *maxLength: 50 minLength: 5*** |
| **confirmPassword\*** | **string *maxLength: 50 minLength: 5*** |

**}**

1. Crear la clase RegisterRequest.cs en PlannerApp.Shared – Shared - Models, según modelo de solicitud de registro:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace PlannerApp.Shared.Models {

public class RegisterRequest {

public string Email { get; set; }

public string FirstName { get; set; }

public string LastName { get; set; }

public string Password { get; set; }

public string ConfirmPassword { get; set; }

}

}

Nota: para no crear la clase manualmente copiaremos el ejemplo del modelo de datos, crearemos la clase y edición pegado especial – pegar JSON como clases.

No aplicaremos validación tradicional de email, utilizaremos validación fluida que es mejor para las anotaciones de datos, la desventaja que tiene es que más códigos, mayor flexibilidad (más lógica).

1. Añadiremos la lógica a la carpeta Models, previamente en el administrador de paquetes nuget desde el proyecto PlannerApp.Shared – Dependencias – Botón derecha – administrador paquetes nuget, añadiremos: FluentValitation
2. Crear la carpeta Validators en PlannerApp.Shared y 1 clase dentro de esta carpeta: RegisterRequestValidator.cs

using FluentValidation;

using PlannerApp.Shared.Models;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace PlannerApp.Shared.Validators {

public class RegisterRequestValidator : AbstractValidator<RegisterRequest> {

public RegisterRequestValidator() {

RuleFor(p => p.Email)

.NotEmpty()

.WithMessage("Email is required")

.EmailAddress()

.WithMessage("Email is not a valid email address");

RuleFor(p => p.FirstName)

.NotEmpty()

.WithMessage("First name is required")

.MaximumLength(25)

.WithMessage("First name must be less than 25 characters");

RuleFor(p => p.LastName)

.NotEmpty()

.WithMessage("Last name is required")

.MaximumLength(25)

.WithMessage("Last name must be less than 25 characters");

RuleFor(p => p.Password)

.NotEmpty()

.WithMessage("Password is required")

.MinimumLength(5)

.WithMessage("Password must at least 5 characters");

RuleFor(p => p.ConfirmPassword)

.Equal(p => p.Password)

.WithMessage("Confirm password doesn't match the password");

}

}

}

1. Implementar el formulario, moveremos el formulario **LoginForm**.razor.cs a un nuevo proyecto app.services que se creará para compartir interfaces, así en lugar de inyectar el cliente http, inyectaremos el servicio de autenticación, habrá que trasladar el código responsable de comunicar con la api, para separar el proyecto, así se mantendrá en 1 único lugar y si vamos a crear diferentes clientes tendríamos que estar escribiendo el mismo código 1 y otra vez:

private async Task LoginUserAsync() {

\_isBusy = true;

\_errorMessage = string.Empty;

var response = await HttpClient.PostAsJsonAsync("/api/v2/auth/login", \_model);

if (response.IsSuccessStatusCode) {

var result = await response.Content.ReadFromJsonAsync<ApiResponse<LoginResult>>();

// Store it in local storage

await Storage.SetItemAsStringAsync("access\_token", result.Value.Token);

await Storage.SetItemAsync<DateTime>("expiry\_date", result.Value.ExpiryDate);

await AuthenticationStateProvider.GetAuthenticationStateAsync();

Navigation.NavigateTo("/");

}

else {

var errorResult = await response.Content.ReadFromJsonAsync<ApiErrorResponse>();

\_errorMessage = errorResult.Message;

}

\_isBusy = false;

}

# 11. Create HttpClientServices Project To Abstract the API Calls

Separar la lógica de la comunicación de la API en otro proyecto para reducir el uso de código, no hace falta escribirlo varias veces, además de facilitar las pruebas unitarias podemos crear 1 abstracción evitando tener formularios mezclados con la API. Tampoco hace falta tener la API dentro del mismo proyecto incluso tenerla en la nube en otro servidor.

1. Crear 1 nuevo proyecto donde pondremos el código de comunicación con la API, proyecto compartido si creamos varios clientes, desde la solución – agregar – nuevo proyecto – **librería de clases** - PlaneerApp.Client.Services – Net 6.
2. Eliminar la clase por defecto que se crea Class1.cs y crear una carpeta para los interfaces, dentro de esta la clase para implementar el servicio de **autenticación** IAuthenticationService.cs:

En autenticación se deberán cubrir iniciar sesión y registrarse, ya hemos iniciado sesión implementar el registro.

Si consultamos la API registro del anterior tema hay que facilitarle el modelo de datos RegisterRequest:

public string Email { get; set; }

public string FirstName { get; set; }

public string LastName { get; set; }

public string Password { get; set; }

public string ConfirmPassword { get; set; }

Devolver un objeto respuesta **ApiResponse** con 1 mensaje para saber si ha tenido éxito:

public string Message { get; set; }

public bool IsSuccess { get; set; }

IAuthenticationService.cs:

using PlannerApp.Shared.Models;

using PlannerApp.Shared.Responses;

using System.Threading.Tasks;

namespace PlaneerApp.Client.Services.Interfaces {

public interface IAuthenticationService {

Task<ApiResponse> RegisterUserAsync(RegisterRequest model);

// TODO: Migrate login to IAuthenticationService (Pendiente)

}

}

1. En dependencias agregar la referencia al proyecto compartido desde dependencias botón de la derecha – agregar referencia del proyecto - PlannerApp.Shared para tener acceso a todos los modelos.
2. En caso de fracaso (solicitudes incorrectas), se lanzará 1 excepción y se tratará aparte, crearemos 1 carpeta de Excepciones (Exceptions) y agregar 1 nueva clase ApiException.cs:

**Modelo de datos errores API:**

**{**

**"message": "string”,**

**"errors":: ["string"],**

**"isSuccess": true**

**}**

using PlannerApp.Shared.Responses;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Net;

using System.Text;

using System.Threading.Tasks;

namespace PlaneerApp.Client.Services.Exceptions {

//hacer f12 en exception para ver la clase con sus metadatos

public class ApiException : Exception {

//propiedad obtener respuesta error, agregaremos este error para que se pueda acceder desde 1 declaración catch en el cliente o desde la sintaxis o declaración try catch

public ApiErrorResponse ApiErrorResponse { get; set; }

public HttpStatusCode StatusCode { get; set; } //400, …

public ApiException(ApiErrorResponse error, HttpStatusCode statusCode) : this(error) {

StatusCode = statusCode;

}

//lanzaremos la excepción

public ApiException(ApiErrorResponse error) {

ApiErrorResponse = error;

}

}

}

1. Crear la implementación del servicio de registro botón derecho en la raíz del proyecto PlaneerApp.Client.Services – agregar – clase - HttpAuthenticationService.cs:

using PlaneerApp.Client.Services.Exceptions;

using PlaneerApp.Client.Services.Interfaces;

using PlannerApp.Shared.Models;

using PlannerApp.Shared.Responses;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Net.Http;

using System.Net.Http.Json;

using System.Text;

using System.Threading.Tasks;

namespace PlaneerApp.Client.Services {

public class HttpAuthenticationService : IAuthenticationService {

private readonly HttpClient \_client;

public HttpAuthenticationService(HttpClient client) {

\_client = client;

}

public async Task<ApiResponse> RegisterUserAsync(RegisterRequest model) {

var response = await \_client.PostAsJsonAsync("/api/v2/auth/register", model);

if (response.IsSuccessStatusCode) {

var result = await response.Content.ReadFromJsonAsync<ApiResponse>();

return result;

}

Else {

var errorResponse = await response.Content.ReadFromJsonAsync<ApiErrorResponse>();

//lanzar excepción obteniendo la respuesta al mismo

throw new ApiException(errorResponse, response.StatusCode);

}

}

}

}

# 12. Design & Implement The Register Form

Migrar el inicio de sesión, registro al proyecto creado anteriormente, que es responsable de la comunicación con las APIs.

Diseñaremos el formulario de registro y enviar la lógica de detrás.

1. Agregar una referencia al proyecto creado anteriormente desde Dependencias – botón derecho – Agregar referencia del proyecto – marcar PlannerApp.Client.Services.
2. Creación de 1 metodo de extensión responsable de agregar todos los servicios y el proyecto de datos a este, crea 1 clase DependencyInjectionExtensions.cs en el proyecto PlannerApp.Client.Services:

using Microsoft.Extensions.DependencyInjection;

using PlaneerApp.Client.Services.Interfaces;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace PlaneerApp.Client.Services

{

public static class DependencyInjectionExtensions {

//control punto en IServiceCollection Instalar Microsoft.Extensions.DependencyInjection.Abstractions

//métodos de extensión nos permite extender la funcionalidad de un objeto o tipo con métodos estáticos. Con las únicas dos condiciones que son, el método tiene que ser estático y en el primer parámetro, debemos indicar la palabra clave “this”, este método se podrá llamar desde otros servicios

public static IServiceCollection AddHttpClientServices(this IServiceCollection services) {

return services.AddScoped<IAuthenticationService, HttpAuthenticationService>()

.AddScoped<IPlansService, HttpPlansService>();

}

}

}

1. En program.cs del proyecto PlannerApp agregamos los servicios http creados en el punto anterior:

builder.Services.AddHttpClientServices();

1. En PlannerApp, como estamos usando validación fluida, instalaremos la biblioteca responsable del flujo y validación Blazored.FluentValidation
2. Ir a la autenticación de componentes y crear el formulario de registro, proyecto PlannerApp – Components – Authentication – botón derecha agregar – Componente razor - RegisterForm.razor, copiaremos el código de mudblazor del formulario LoginFom.razor:

@namespace PlannerApp.Components

<**EditForm** **Model**="\_model" **OnValidSubmit**="RegisterUserAsync">

//<**DataAnnotationsValidator** /> cambiaremos el validadorque viene por defecto porque no estamos definiendo ninguno en el modelo por fluentvalidator

<**FluentValidationValidator** />

<**MudCard**>

<**MudCardHeader**>

<**CardHeaderContent**>

<**MudText** **Typo**="Typo.h6">Welcome to PlannerApp v2.0</**MudText**>

</**CardHeaderContent**>

</**MudCardHeader**>

<**MudCardContent**>

<**MudTextField** **Label**="Email" **HelperText**="Required" @bind-Value="@\_model.Email" />

<**MudTextField** **Label**="First Name" **HelperText**="Required" @bind-Value="@\_model.FirstName" />

<**MudTextField** **Label**="Last Name" **HelperText**="Required" @bind-Value="@\_model.LastName" />

<**MudTextField** **InputType**="InputType.Password" **Label**="Password" **HelperText**="Required" @bind-Value="@\_model.Password" />

<**MudTextField** **InputType**="InputType.Password" **Label**="Confirm Password" **HelperText**="Required" @bind-Value="@\_model.ConfirmPassword" />

<**ValidationSummary** />

@if (!string.IsNullOrWhiteSpace(\_errorMessage)) {

<**MudAlert** **Severity**="Severity.Error" **Class**="my-2" **Variant**="Variant.Filled">@\_errorMessage</**MudAlert**>

}

</**MudCardContent**>

<**MudCardActions**>

<**MudButton** **ButtonType**="ButtonType.Submit" **Variant**="Variant.Filled" **Color**="Color.Primary" **Disabled**="\_isBusy">Create Account</**MudButton**>

<**MudButton** **Variant**="Variant.Outlined" **Color**="Color.Secondary" **OnClick**="RedirectToLogin" **Class**="ml-1" **Disabled**="\_isBusy">Login</**MudButton**>

</**MudCardActions**>

</**MudCard**>

</**EditForm**>

@code {

}

Crear código detrás del archivo – botón derecho en la misma carpeta – agregar clase - RegisterForm.razor.cs, copiaremos el código de LoginForm.razor.cs:

using Microsoft.AspNetCore.Components;

using PlaneerApp.Client.Services.Exceptions;

using PlaneerApp.Client.Services.Interfaces;

using PlannerApp.Shared.Models;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

namespace PlannerApp.Components {

public partial class RegisterForm {

[Inject] //servicio de autenticación

public IAuthenticationService AuthenticationService { get; set; }

[Inject]

public NavigationManager Navigation { get; set; }

private RegisterRequest \_model = new();

private bool \_isBusy = false;

private string \_errorMessage = string.Empty;

private async Task RegisterUserAsync() {

\_isBusy = true;

\_errorMessage = string.Empty;

try {

await AuthenticationService.RegisterUserAsync(\_model);

Navigation.NavigateTo("/authentication/login");

}

catch (ApiException ex) {

// Hanlde the errors of the aPI

// TODO: Log those errors

\_errorMessage = ex.ApiErrorResponse.Message;

}

catch(Exception ex) {

// Handle errors

\_errorMessage = ex.Message;

}

\_isBusy = false;

}

private void RedirectToLogin() {

Navigation.NavigateTo("/authentication/login");

}

}

}

1. Agregar la validación fluida:

En \_imports.razor añadir @using Blazored.FluentValidation

La validación fluida se realizará automáticamente, solo hay 1 validador de solicitud, ejecutar el proyecto para comprobarlo.

# 13. Fix the Layout Of Register Form

Hay un error en el formulario para registrarse dentro de la carpeta de componentes, esta carpeta tiene solo componentes que son páginas sin enrutamiento, hay que solucionarlo moviendo el enrutador y la autenticación, para ello crearemos una **página** (componente razor) en PlannerApp – Pages – Authentication - Register.razor:

@page "/authentication/register" //enrutamiento

@layout AuthLayout

<RegisterForm />

Quitaremos @page "/authentication/register" del componente: Components – Authentication - RegisterForm.razor

# 14. Fetch Plans 1. Create the Models

Se ha acabado la autenticación, infraestructura, la configuración del cliente http, la implementación de inicio de sesión y el formulario de registro. El siguiente paso son los planes, para ello nos iremos a la documentación de la API V2, el método Get requiere los siguientes parámetros:

#### Plans

**GET​/api​/v2​/Plans**

Parameters query (string), pageNumber (int), pageSize (int)

Respuesta en el caso de éxito (200) devuelve un message con value que contiene un objeto llamado lista? con paginas, total registros, …, y récords que contiene 1 lista de elementos donde cada elemento es 1 plan con titulo, archivo de portada, …, el histórico de ítems (toDoItems) no se recupera en esta API, solo id. Esta solicitud se recuperará en la URL de la portada de la descripción del título de identificación

**{**

**"value": {**

**"totalPages": 0,**

**"page": 0,**

**"pageSize": 0,**

**"itemsCount": 0,**

**"records": [**

**{**

**"id": "string",**

**"title": "string",**

**"description": "string",**

**"coverFile": "string",**

**"coverUrl": "string",**

**"toDoItems": [**

**{**

**"id": "string",**

**"description": "string",**

**"estimationDate": "2021-12-08T07:36:36.625Z",**

**"achievedDate": "2021-12-08T07:36:36.625Z",**

**"isDone": true,**

**"planId": "string"**

**}**

**]**

**}**

**]**

**},**

**"message": "string",**

**"isSuccess": true**

**}**

En cuanto al diseño, el usuario va a poder cambiar entre 1 de estos 2, depende de la cantidad de registros que tenga:



1. Creación de modelos, en el proyecto PlannerApp.Shared – Models – agregar nueva clase: PagedList.cs:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace PlannerApp.Shared.Models {

//genérico para cualquier tipo de datos, en nuestro proyecto planes, pero puede ser cualquier modelo de datos coches, …

public class PagedList<T> {

public int TotalPages { get; set; }

public int ItemsCount { get; set; }

public int Page { get; set; }

public int PageSize { get; set; }

public IEnumerable<T> Records { get; set; }

}

}

1. Nuevo modelo de datos (clase) PlanSummary.cs, tendremos 2 resultados diferentes este modelo implementará el resumen del plan, tenemos 1 api para obtener el plan por id (detalle).

namespace PlannerApp.Shared.Models {

public class PlanSummary {

public string Id { get; set; }

public string Title { get; set; }

public string Description { get; set; }

public string CoverUrl { get; set; }

}

}

}

1. En el proyecto PlaneerApp.Client.Services crear la interfaz o el servicio de cliente http para comunicarnos con los planes, siendo fácil (por ejemplo pruebas unitarias) aislar el cliente http de los componentes y esa lógica de verificación de conmutación es el código de estado ?, la interfaz se llamará IPlansService.cs en la carpeta Interfaces:

using PlannerApp.Shared.Models;

using PlannerApp.Shared.Responses;

using System;

using System.Collections.Generic;

using System.IO;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace PlaneerApp.Client.Services.Interfaces {

public interface IPlansService {

Task<ApiResponse<PagedList<PlanSummary>>> GetPlansAsync(string query = null, int pageNumber = 1, int pageSize = 10);

}

}

1. Crear 1 nueva clase en la raíz del proyecto PlaneerApp.Client.Services que implemente la interfaz del punto anterior, se llamará HttpPlansService.cs:

using PlaneerApp.Client.Services.Exceptions;

using PlaneerApp.Client.Services.Interfaces;

using PlannerApp.Shared.Models;

using PlannerApp.Shared.Responses;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Net.Http;

using System.Net.Http.Json;

using System.Text;

using System.Threading.Tasks;

namespace PlaneerApp.Client.Services {

public class HttpPlansService : IPlansService {

private readonly HttpClient \_httpClient;

public HttpPlansService(HttpClient httpClient) {

\_httpClient = httpClient;

}

//botón derecho en IplansService implementar la interfaz y agregar using de las interfaces

public async Task<ApiResponse<PagedList<PlanSummary>>> GetPlansAsync(string query = null, int pageNumber = 1, int pageSize = 10)

{

var response = await \_httpClient.GetAsync($"/api/v2/plans?query={query}&pageNumber={pageNumber}&pageSize={pageSize}");

if (response.IsSuccessStatusCode) {

var result = await response.Content.ReadFromJsonAsync<ApiResponse<PagedList<PlanSummary>>>();

return result;

}

Else {

var errorResponse = await response.Content.ReadFromJsonAsync<ApiErrorResponse>();

throw new ApiException(errorResponse, response.StatusCode);

}

}

}

}

# 15. Fetch Plans 2. Create the Plan Cards List Component