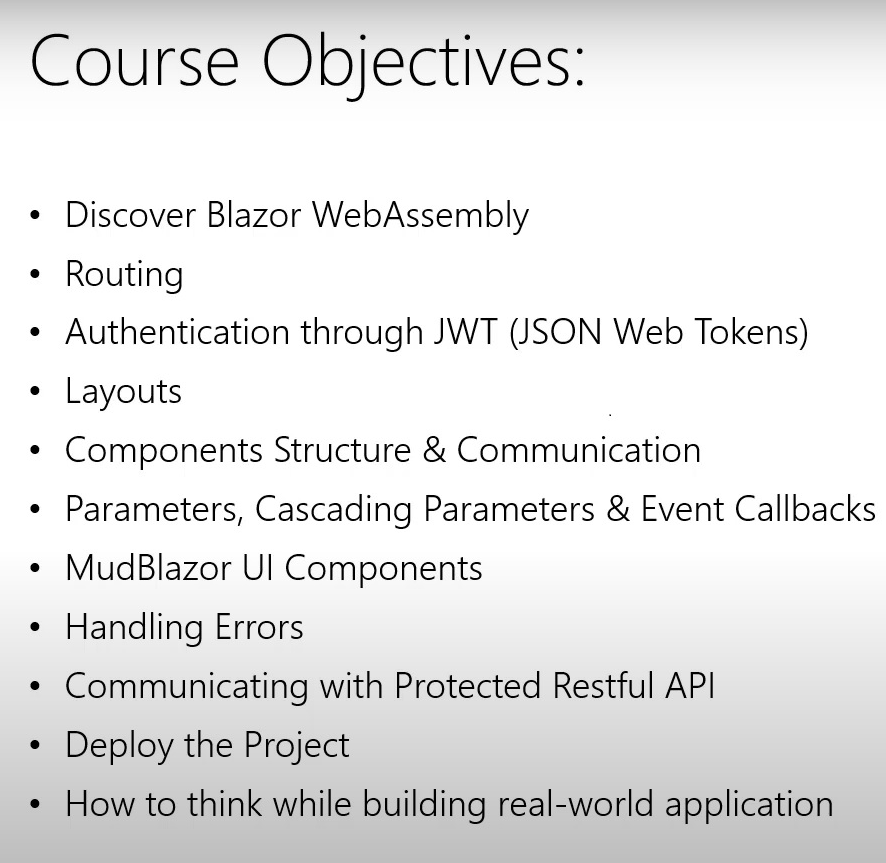
<https://www.youtube.com/playlist?list=PLFJQnCcZXWjsYlljYtU0oKTnJDcrLGgM2>

RegisterRequestValidatorRegisterRequestValidator

Contenido

[2. Apis 4](#_Toc90982399)

[3. Create the GitHub Repo & Initialize the Project with MudBlazor 7](#_Toc90982400)

[4. Discover the V2.0 API and Test It with 10](#_Toc90982401)

[Simular Login Postam 10](#_Toc90982402)

[Simular Registro de usuario en Postam 12](#_Toc90982403)

[Creación de un formulario de registro en Visual Studio 13](#_Toc90982404)

[5. Setup HttpClient & Create Authorization Messaging Handler 14](#_Toc90982405)

[6. Create Custom Authentication State Provider 17](#_Toc90982406)

[7. Implement the Auth Layout & Design the Login Form 21](#_Toc90982407)

[8. Implement the Logic of Login Form 29](#_Toc90982408)

[9. Implement LogOut Functionality and User Menu in App Bar 31](#_Toc90982409)

[10. Create Register Model With Validation using Fluent Validation 33](#_Toc90982410)

[11. Create HttpClientServices Project To Abstract the API Calls 36](#_Toc90982411)

[12. Design & Implement The Register Form 39](#_Toc90982412)

[13. Fix the Layout Of Register Form 43](#_Toc90982413)

[14. Fetch Plans 1. Create the Models 44](#_Toc90982414)

[15. Fetch Plans 2. Create the Plan Cards List Component 48](#_Toc90982415)

[16. Search and Pagination for the Plan Cards 60](#_Toc90982416)

[17. Show Plans in a DataGrid using MudBlazor Table with Server Side Reload 62](#_Toc90982417)

[18. Implement Plans Layout Switcher Between Cards List 67](#_Toc90982418)

[19. Implement NavMenu 68](#_Toc90982419)

[20. Implement Edit - Create Plan Client Service Logic 71](#_Toc90982420)

[21. Design the Plan Form 76](#_Toc90982421)

[22. Implement the Plan Create Form Submit Logic 80](#_Toc90982422)

[23. Implement Edit Functionality for Plan 81](#_Toc90982423)

[24. Finalizing Edit Plan 84](#_Toc90982424)

[25. Implement Delete Plan API Call 85](#_Toc90982425)

[26. Implementing Deleting the Plan Logic with Confirmation Dialog 86](#_Toc90982426)

[27. MessagingCenter with Blazor To Notify Plans List To Refresh After Delete 89](#_Toc90982427)

[28. Implement Plan Details Dialog to View Plan With Items 91](#_Toc90982428)

[29. Upgrade the Blazor App to .NET 6 And Implement New Features 97](#_Toc90982429)

[30. Designing the ToDo Item Card, Using Hot Reload And Setup Items Layout 98](#_Toc90982430)

[31. Implement the Logic of ToDo Items Http Client To Communicate With API 101](#_Toc90982431)

[Anexo 1 - Delegado Func<in T, out TResult> 102](#_Toc90982432)

[Anexo 2 - Messaging Center Blazor Utitlity 103](#_Toc90982433)

# 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/>

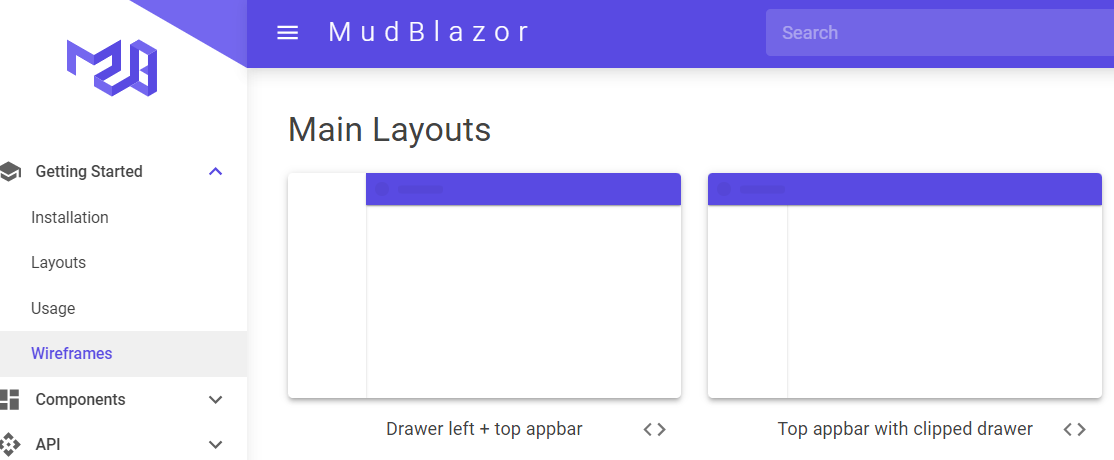
<MudDialogProvider/>

<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"

    },

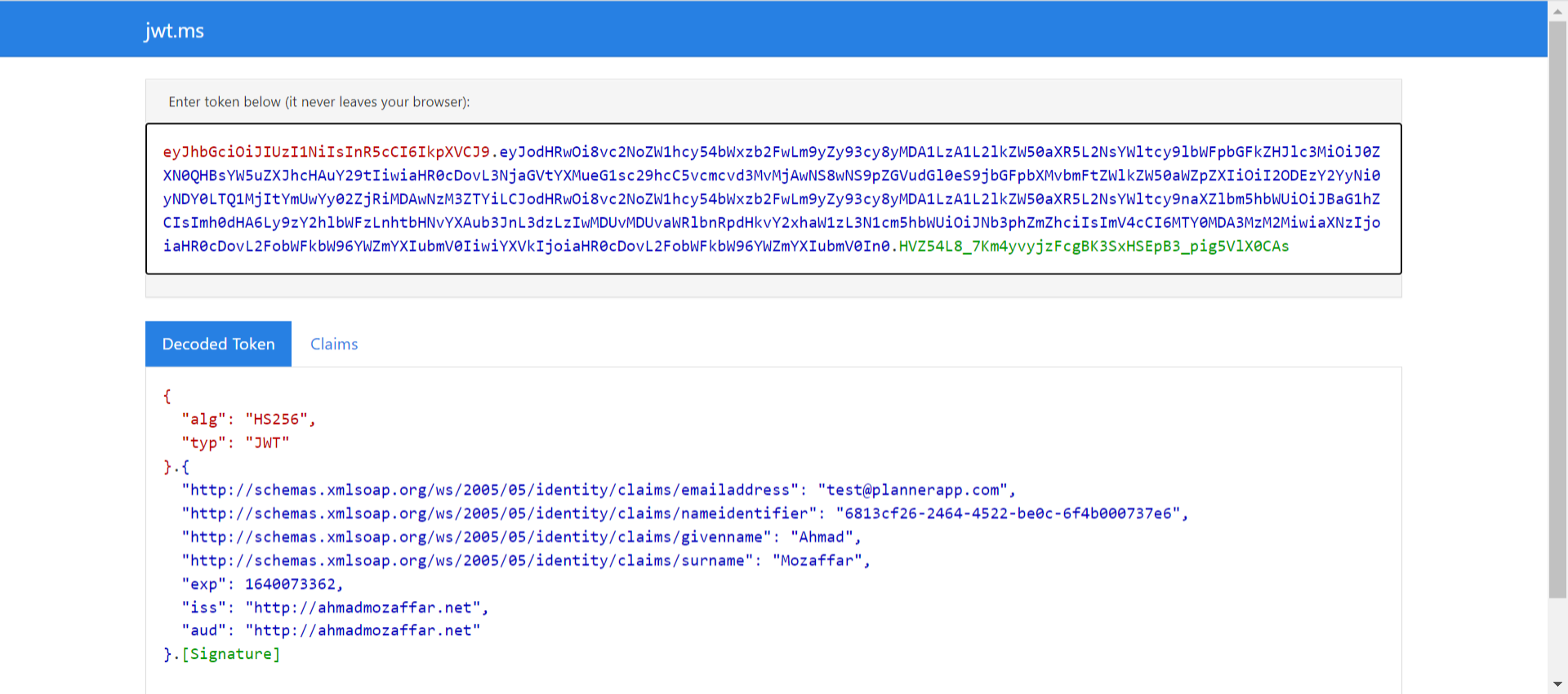
    "message": "Access token retrieved successfully",

    "isSuccess": **true**

}

Obtendremos el token + fecha de vencimiento.

1. Copiar el token en la url: [**https://jwt.ms/**](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 ApiErrorResponse.cs 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 MudBlazor, 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)">

1. 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 {

}

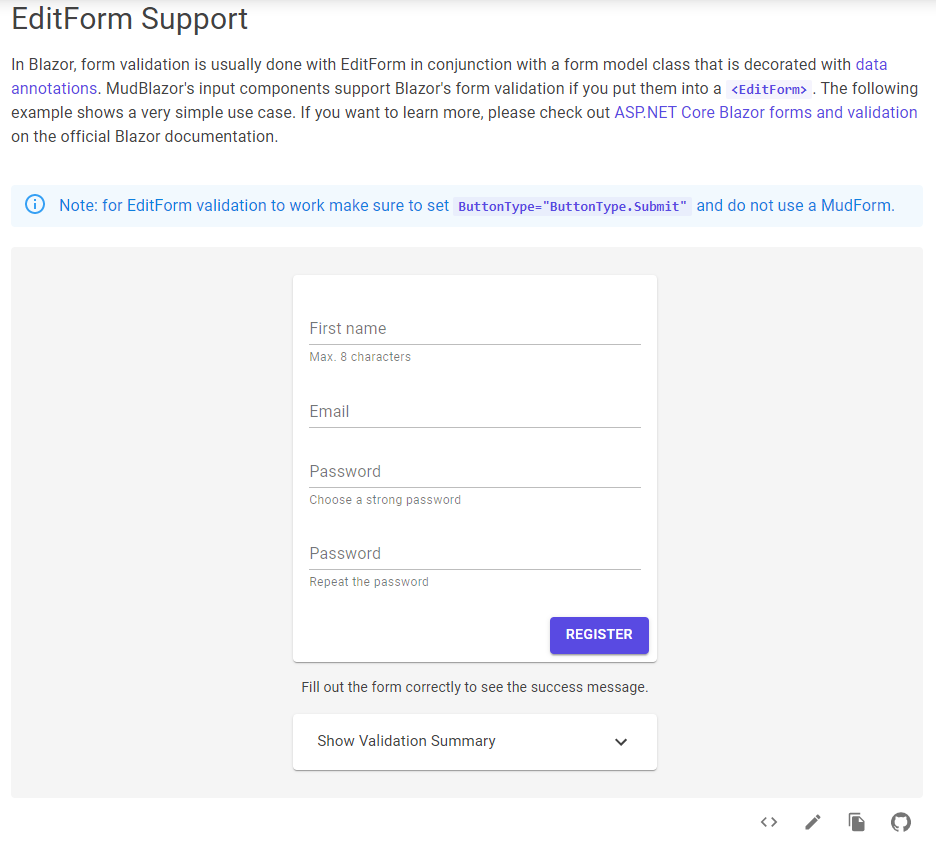
1. Las páginas están vacías en las que vamos añadiendo componentes, estos componentes se crearán en 1 nueva carpeta en la raíz que llamaremos Components + 1 subcarpeta Authentication.

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 MudBlazor – 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 {

}

1. **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");

        }

}

}

1. 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. Mudblazor – 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");

}

}

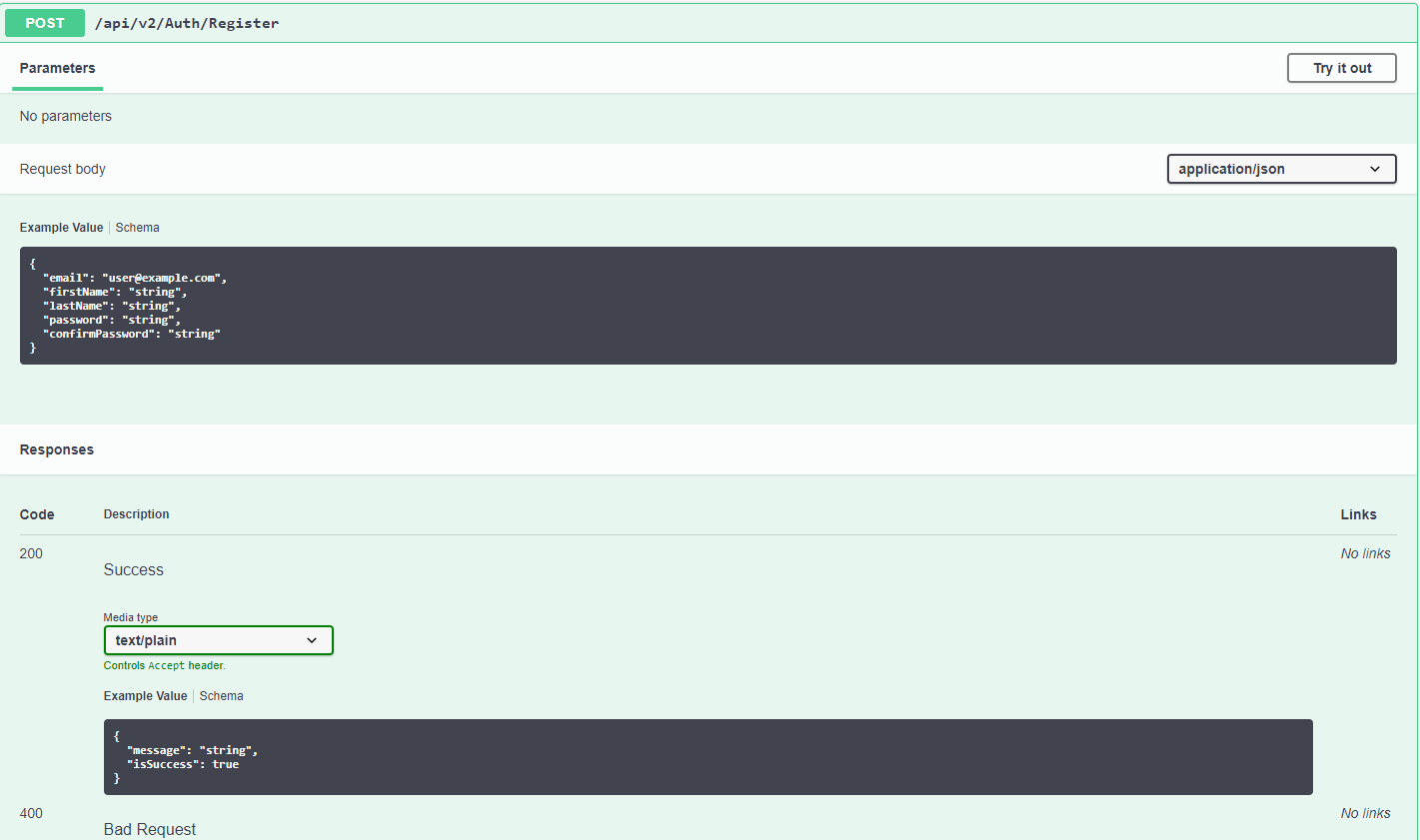
1. En MainLayout.razor añadiremos el componente creado anteriormente:

<**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

Componentes:

**PlansList** (eventos que llamarán al controlador httpClient en HttpPlansService.cs) compuesto de PlanCardsList.

**PlanCardsList** (formato tabla) componente grid mudblazor, los eventos se procesarán en el padre PlansList se definen como Parameter - EventCallback

**PlanCard** (componente individual para cada elemento de PlanCardsList), se definen los eventos cru como parámetros para que se puedan ejecutar en el padre.

Páginas:

Plans.razor: página enrutable que pueda llamar al controlar contiene el componente PlansList.

1. Agregar en las extensiones de inyección **DependencyInjectionExtensions.cs** del proyecto PlaneerApp.Client.Services los nuevos servicios:

return services.AddScoped<IAuthenticationService, HttpAuthenticationService>()

.AddScoped<IPlansService, HttpPlansService>();

1. Crear una nueva carpeta el PlaneerApp – Components – Plans, así que ahora tenemos 1 parte de los componentes: lista que recuperará los componentes y luego tendremos 2, 1 para la cuadricula y otro para la tarjeta de datos.

Crear el componente razor para la lista de planes en la carpeta Plans: PlansList.razor y la clase con el código c# PlansList.razor.cs

En vez de crear a mano código cs se podía haber hecho desde PlansList.razor clic en la parte izquierda – extraer bloque de código subyacente: genera 1 clase parcial.

**PlansList.razor**:

@namespace PlannerApp.Components

@\*UI View Selector \*@

//soporte para las múltiples vistas

//agrega los botones para que el usuario pueda seleccionar

<**MudIconButton** **Icon**="@Icons.Material.Filled.GridView" **Color**="Color.Primary" **Disabled**="\_isCardsViewEnabled" **OnClick**="SetCardsView"></**MudIconButton**>

<**MudIconButton** **Icon**="@Icons.Material.Filled.TableView" **Color**="Color.Primary" **Disabled**="@(!\_isCardsViewEnabled)" **OnClick**="SetTableView"></**MudIconButton**>

<div class="mt-4" />

@if (\_isCardsViewEnabled) {

<PlanCardsList FetchPlans="GetPlansAsync"

OnEditClicked="EditPlan"

OnDeleteClicked="DeletePlanAsync"/>

}

else {

<PlansTable OnEditClicked="EditPlan"

OnDeleteClicked="DeletePlanAsync" />

}

1. Código para la página principal: PlansList.razor.cs:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Microsoft.AspNetCore.Components;

using System.Net.Http;

using System.Net.Http.Json;

using Microsoft.AspNetCore.Components.Forms;

using Microsoft.AspNetCore.Components.Routing;

using Microsoft.AspNetCore.Components.Web;

using Microsoft.AspNetCore.Components.Web.Virtualization;

using Microsoft.AspNetCore.Components.WebAssembly.Http;

using Microsoft.AspNetCore.Components.Authorization;

using Microsoft.AspNetCore.Authorization;

using Microsoft.JSInterop;

using PlannerApp;

using PlannerApp.Shared;

using PlannerApp.Components;

using MudBlazor;

using Blazored.FluentValidation;

using PlaneerApp.Client.Services.Interfaces;

using PlaneerApp.Client.Services.Exceptions;

using PlannerApp.Shared.Models;

using AKSoftware.Blazor.Utilities;

namespace PlannerApp.Components {

public partial class PlansList {

[Inject] //controlador http

public IPlansService PlansService { get; set; }

[Inject]

public NavigationManager Navigation { get; set; }

[Inject]

public IDialogService DialogService { get; set; }

private bool \_isBusy = false;

private string \_errorMessage = string.Empty;

private int \_pageNumber = 1;

private int \_pageSize = 10;

//cadena de búsqueda a utilizar en la consulta

private string \_query = string.Empty;

private int \_totalPages = 1;

//resumen del plan

private List<PlanSummary> \_plans = new();

//vista de cuadricula de datos y vista de tarjetas en diferentes //componentes podrían utilizar esta funcion si utilizamos delegados?

//delegados:

//función más importante que devuelve una tarea asíncrona encargada //de obtener los planes con valores predeterminados

private async Task<PagedList<PlanSummary>> GetPlansAsync(string query = "", int pageNumber = 1, int pageSize = 10) {

\_isBusy = true;

Try {

//Esta función llama al proyecto PlaneerApp.Client.Services - //HttpAuthenticationService.cs que se encarga de centralizar las llamadas //Client y utiliza la función GetPlansAsync que se encarga de llamar al //controlador que, de momento está en la nube.

//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={pageNumbe// r}&pageSize={pageSize}");

var result = await PlansService.GetPlansAsync(query, pageNumber, pageSize);

//contendrá registros de la consulta

\_plans = result.Value.Records.ToList();

\_pageNumber = result.Value.Page;

\_pageSize = result.Value.PageSize;

\_totalPages = result.Value.TotalPages;

return result.Value;

}

//solicitud que viene del servicio del API cuando haya 1 incorrecta

catch (ApiException ex)

{

\_errorMessage = ex.ApiErrorResponse.Message;

}

catch (Exception ex) {

// TODO: Log this error

\_errorMessage = ex.Message;

}

\_isBusy = false;

return null;

}

#region View Toggler

private bool \_isCardsViewEnabled = true;

private void SetCardsView() {

\_isCardsViewEnabled = true;

}

private void SetTableView() {

\_isCardsViewEnabled = false;

}

#endregion

#region Edit

private void EditPlan(PlanSummary plan) {

Navigation.NavigateTo($"/plans/form/{plan.Id}");

}

#endregion

#region Delete

private async Task DeletePlanAsync(PlanSummary plan) {

var parameters = new DialogParameters();

parameters.Add("ContentText", $"Do you really want to delete the plan '{plan.Title}'?");

parameters.Add("ButtonText", "Delete");

parameters.Add("Color", Color.Error);

var options = new DialogOptions() { CloseButton = true, MaxWidth = MaxWidth.ExtraSmall };

var dialog = DialogService.Show<ConfirmationDialog>("Delete", parameters, options);

var confirmationResult = await dialog.Result;

if (!confirmationResult.Cancelled) {

// Confirmed to delete

Try {

await PlansService.DeleteAsync(plan.Id);

// Send a message about the deleted plan

MessagingCenter.Send(this, "plan\_deleted", plan);

}

catch (ApiException ex) {

// TODO: Log this error

}

catch (Exception ex) {

// TODO: Log this error

}

}

}

#endregion

}

}

1. Soporte **vista de lista de tarjetas**:

Crear un nuevo componente de razor en PlannerApp – Components – Plans: PlanCardsList.razor:

Nada mas crear el componente – en las acciones rápidas – extraer el bloque al código subyacente para que se cree el código c# **PlanCardsList.razor.cs**

@namespace PlannerApp.Components

<**MudPaper** **Class**="py-1 px-2">

<div class="d-flex">

<**MudTextField** @bind-Value="\_query" **Variant**="Variant.Outlined" />

<**MudIconButton** **Color**="Color.Primary" **Icon**="@Icons.Filled.Search" **OnClick**="async () => await GetPlansAsync(1)" />

</div>

</**MudPaper**>

<**MudDivider** **Class**="my-4"/>

//si está ocupado intentando presentar los datos de la API se mostrarán como esqueleto (mientras se cargan los datos), desde MudBlazor – Components – Skeleton

@if (\_isBusy) {

<**MudGrid** **Spacing**="2">

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

<PlanCard IsBusy="true"/>

</**MudItem**>

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

<PlanCard IsBusy="true"/>

</**MudItem**>

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

<PlanCard IsBusy="true"/>

</**MudItem**>

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

<PlanCard IsBusy="true"/>

</**MudItem**>

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

<PlanCard IsBusy="true"/>

</**MudItem**>

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

<PlanCard IsBusy="true"/>

</**MudItem**>

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

<PlanCard IsBusy="true"/>

</**MudItem**>

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

<PlanCard IsBusy="true"/>

</**MudItem**>

</**MudGrid**>

}

else {

<**MudGrid** **Spacing**="2"> //espacio entre los elementos de la cuadricula

@if (\_result != null && \_result.Records != null)

{

foreach (var plan in \_result.Records)

{

//mudblazor componte grid mostrará código para crear elementos de 1 tabla en este caso //MudItem

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

//nuevo componente PlanCard

<PlanCard PlanSummary="plan"

OnEditClicked="() => OnEditClicked.InvokeAsync(plan)"

OnDeleteClicked="() => OnDeleteClicked.InvokeAsync(plan)"/>

</**MudItem**>

}

}

</**MudGrid**>

}

<div class="d-flex justify-center py-2 px-1 mt-6">

@for(int i = 1;i <= \_result.TotalPages; i++) {

int index = i;

<**MudButton** **Variant**="Variant.Filled" **Color**="Color.Primary" **Class**="mr-1" **OnClick**="async () => await GetPlansAsync(index)" **Disabled**="@(\_pageNumber == index)">@index</**MudButton**>

}

</div>

1. PlanCardsList.razor.cs:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Microsoft.AspNetCore.Components;

using System.Net.Http;

using System.Net.Http.Json;

using Microsoft.AspNetCore.Authorization;

using Microsoft.JSInterop;

using PlannerApp;

using PlannerApp.Shared;

using PlannerApp.Components;

using MudBlazor;

using Blazored.FluentValidation;

using PlannerApp.Shared.Models;

using System.IO;

using AKSoftware.Blazor.Utilities;

namespace PlannerApp.Components {

public partial class PlanCardsList {

private bool \_isBusy { get; set; }

private int \_pageNumber = 1;

private int \_pageSize = 10;

private string \_query = string.Empty;

[Inject]

public NavigationManager Navigation { get; set; }

//creación del delegado del tipo func , este parámetro es un tipo de //función, variable o tipo de datos que reimprime o vuelve a //representar la función, pero su implementación está en el padre no //es su responsabilidad,por lo que se delega

//ver anexo 1 - Delegado Func o

<https://www.netmentor.es/entrada/delegados-csharp>

[Parameter]

public Func<string, int, int, Task<PagedList<PlanSummary>>> FetchPlans { get; set; }

[Parameter]

public EventCallback<PlanSummary> OnEditClicked { get; set; }

[Parameter]

public EventCallback<PlanSummary> OnDeleteClicked { get; set; }

private PagedList<PlanSummary> \_result = new();

protected override void OnInitialized() {

MessagingCenter.Subscribe<PlansList, PlanSummary>(this, "plan\_deleted", async (sender, args) => {

await GetPlansAsync(\_pageNumber);

StateHasChanged();

});

}

protected async override Task OnInitializedAsync() {

await GetPlansAsync();

}

//No hace falta repetir la implementación de la función GetPlansAsync, la //tenemos en el padre, desde el componente de vista en cuadricula también //se llamará, en este componente se ha creado (pasado) un delegado a la

//función padre FetchPlans que es el que se utiliza

//delegados: <https://www.netmentor.es/entrada/delegados-csharp>

//el componente necesita llamar a 1 función que devuelve una tarea con 3

//parámetros, esta clase no será responsable de buscar, llamarla o //escribir el código, solo hace falta definir la cabecera de la función //con los criterios, el padre es responsable de la lógica, así podemos pasar //dicha función a otro componente usando 1 delegado

private async Task GetPlansAsync(int pageNumber = 1) {

\_pageNumber = pageNumber;

\_isBusy = true;

\_result = await FetchPlans?.Invoke(\_query, \_pageNumber, \_pageSize);

\_isBusy = false;

}

}

}

1. Nuevo componente con el diseño de 1 tarjeta del plan – **PlanCard.razor**, para el diseño desde Mudblazor – components – cards – Conjunto, seleccionar el código <>

@namespace PlannerApp.Components

@if (IsBusy) {

<**MudCard** **Elevation**="0">

<**MudSkeleton** **SkeletonType**="SkeletonType.Rectangle" **Height**="200px" />

<**MudCardContent**>

<**MudSkeleton** **Width**="30%" **Height**="42px;" />

<**MudSkeleton** **Width**="80%" />

<**MudSkeleton** **Width**="100%" />

</**MudCardContent**>

<**MudCardActions**>

<**MudSkeleton** **Width**="64px" **Height**="40px" **Class**="ml-2" />

<**MudSkeleton** **Width**="105px" **Height**="40px" **Class**="ml-3" />

</**MudCardActions**>

</**MudCard**>

}

else {

<**MudCard**>

<**MudCardHeader**>

<**CardHeaderContent**>

<**MudText** **Typo**="Typo.body1">@PlanSummary?.Title</**MudText**>

</**CardHeaderContent**>

</**MudCardHeader**>

<**MudCardMedia** **Image**="@PlanSummary?.CoverUrl" **Height**="250" />

<**MudCardContent**>

<**MudText** **Typo**="Typo.body2">@PlanSummary?.Description</**MudText**>

</**MudCardContent**>

//acciones

<**MudCardActions**>

//acciones: vista previa, editar y eliminar, para buscar iconos, //desde mudblazor – iconos (diseño)

//listado de iconos mudblazor – caracteristicas – iconos – //buscar, posteriormente seleccionarlo y nos dará el enlace al //mismo en la parte superior derecha

<**MudIconButton** **Icon**="@Icons.Material.Filled.Preview" **Color**="Color.Success" **OnClick**="async () => await OnViewClicked.InvokeAsync(PlanSummary)" />

<**MudIconButton** **Icon**="@Icons.Material.Filled.Edit" **Color**="Color.Default" **OnClick**="async () => await OnEditClicked.InvokeAsync(PlanSummary)" />

<**MudIconButton** **Icon**="@Icons.Material.Filled.Delete" **Color**="Color.Error" **OnClick**="async () => await OnDeleteClicked.InvokeAsync(PlanSummary)" />

</**MudCardActions**>

</**MudCard**>

}

1. Código c# asociado al componente anterior **PlanCard.razor.cs:**

using Microsoft.AspNetCore.Components;

using PlannerApp.Shared.Models;

namespace PlannerApp.Components {

public partial class PlanCard {

[Parameter]

public PlanSummary PlanSummary { get; set; }

[Parameter]

public bool IsBusy { get; set; }

[Parameter]

public EventCallback<PlanSummary> OnViewClicked { get; set; }

[Parameter]

public EventCallback<PlanSummary> OnDeleteClicked { get; set; }

[Parameter]

public EventCallback<PlanSummary> OnEditClicked { get; set; }

}

}

1. Creación 1 carpeta Plans en Pages, dentro de esta carpeta crearemos 1 componente razor (Plans.razor) junto a su clase con código c# (Plans.razor.cs) para que se puedan establecer rutas y acceso a futuros controladores:

Plans.razor:

@page "/plans"

<**PlansList** />

Plans.razor.cs, se implementará más adelante:

using MudBlazor;

namespace PlannerApp.Pages.Plans {

public partial class Plans {

private List<BreadcrumbItem> \_breadcrumbItems = new() {

new BreadcrumbItem("Home", "/index"),

new BreadcrumbItem("Plans", "/plans", true)

};

}

}

Para probar el funcionamiento, deberemos validarnos con [test@plannerapp.com](mailto:test@plannerapp.com) , password Test.123

<https://localhost:puerto/plans>

# 16. Search and Pagination for the Plan Cards

1. Agregar una barra de búsqueda en la parte superior + los botones de paginación, ya está implementado en el API

**GET​/api​/v2​/Plans**

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

En Components – Plants – PlansList.razor (componente padre) , en el componente de lista de tarjetas PlanCardsList.razor crearemos la barra de búsqueda (mudblazor – components – forms and inputs – Text field:

<**MudPaper** **Class**="py-1 px-2">

<div class="d-flex">

<**MudTextField** @bind-Value="\_query" **Variant**="Variant.Outlined" />

<**MudIconButton** **Color**="Color.Primary" **Icon**="@Icons.Filled.Search" **OnClick**="async () => await **GetPlansAsync**(1)" />

//queremos ir a la 1 pagina

</div>

</**MudPaper**>

<**MudDivider** **Class**="my-4"/> //linea de división

En

En PlanCardsList.razor.cs:

protected async override Task OnInitializedAsync() {

await GetPlansAsync();

}

private async Task **GetPlansAsync**(int pageNumber = 1) {

\_pageNumber = pageNumber;

\_isBusy = true;

\_result = await FetchPlans?.Invoke(\_query, \_pageNumber, \_pageSize);

\_isBusy = false;

}

Tenemos implementada la función en HttpPlansService.cs y en el API externo que de momento no se dispone del código.

Añadir la paginación a pie de página, consultar en mudblazor elemento flex en el apartado características para la justificación y llenado de espacio:

<div class="d-flex justify-center py-2 px-1 mt-6">

@for(int i = 1;i <= \_result.**TotalPages**; i++)

{

int index = i;

<**MudButton** **Variant**="Variant.Filled" **Color**="Color.Primary" **Class**="mr-1" **OnClick**="async () => await GetPlansAsync(index)" **Disabled**="@(\_pageNumber == index)">@index</**MudButton**>

//deshabilitamos pulsar en la página actual

}

</div>

# 17. Show Plans in a DataGrid using MudBlazor Table with Server Side Reload

1. Mudblazor – Components – Table – Escogeremos server side filtering, sorting and pagination – **ejemplo** de código fuente del componente de la tabla.

//función de datos del servidor, delegado de función que toma como parámetro la tarea de tablas y devuelve el tipo de datos Element, solo para el fragmento de esta tabla.

//Tenemos un proveedor de clasificación

<**MudTable** **ServerData**="@(new Func<TableState, Task<TableData<Element>>>(ServerReload))"

**Dense**="true" **Hover**="true" @ref="table">

<**ToolBarContent**>

<**MudText** **Typo**="Typo.h6">Periodic Elements</**MudText**>

<**MudSpacer** />

<**MudTextField** **T**="string" **ValueChanged**="@(s=>OnSearch(s))" **Placeholder**="Search" **Adornment**="Adornment.Start"

**AdornmentIcon**="@Icons.Material.Filled.Search" **IconSize**="Size.Medium" **Class**="mt-0"></**MudTextField**>

</**ToolBarContent**>

<**HeaderContent**>

<**MudTh**><**MudTableSortLabel** **SortLabel**="nr\_field" **T**="Element">Nr</**MudTableSortLabel**></**MudTh**>

<**MudTh**><**MudTableSortLabel** **SortLabel**="sign\_field" **T**="Element">Sign</**MudTableSortLabel**></**MudTh**>

<**MudTh**><**MudTableSortLabel** **SortLabel**="name\_field" **T**="Element">Name</**MudTableSortLabel**></**MudTh**>

<**MudTh**><**MudTableSortLabel** **SortLabel**="position\_field" **T**="Element">Position</**MudTableSortLabel**></**MudTh**>

<**MudTh**><**MudTableSortLabel** **SortLabel**="mass\_field" **T**="Element">Molar mass</**MudTableSortLabel**></**MudTh**>

</**HeaderContent**>

<**RowTemplate**>

<**MudTd** **DataLabel**="Nr">@context.Number</**MudTd**>

<**MudTd** **DataLabel**="Sign">@context.Sign</**MudTd**>

<**MudTd** **DataLabel**="Name">@context.Name</**MudTd**>

<**MudTd** **DataLabel**="Position">@context.Position</**MudTd**>

<**MudTd** **DataLabel**="Molar mass">@context.Molar</**MudTd**>

</**RowTemplate**>

<**NoRecordsContent**>

<**MudText**>No matching records found</**MudText**>

</**NoRecordsContent**>

<**LoadingContent**>

<**MudText**>Loading...</**MudText**>

</**LoadingContent**>

<**PagerContent**>

<**MudTablePager** />

</**PagerContent**>

</**MudTable**>

@code {

private IEnumerable<Element> pagedData;

private MudTable<Element> table;

private int totalItems;

private string searchString = null;

/// <summary>

/// Here we simulate getting the paged, filtered and ordered data from the server

/// </summary>

private async Task<TableData<Element>> ServerReload(TableState state) {

IEnumerable<Element> data = await httpClient.GetFromJsonAsync<List<Element>>("webapi/periodictable");

await Task.Delay(300);

data = data.Where(element => {

if (string.IsNullOrWhiteSpace(searchString))

return true;

if (element.Sign.Contains(searchString, StringComparison.OrdinalIgnoreCase))

return true;

if (element.Name.Contains(searchString, StringComparison.OrdinalIgnoreCase))

return true;

if ($"{element.Number} {element.Position} {element.Molar}".Contains(searchString))

return true;

return false;

}).ToArray();

totalItems = data.Count();

//ordenación

switch (state.SortLabel) {

case "nr\_field":

data = data.OrderByDirection(state.SortDirection, o => o.Number);

break;

case "sign\_field":

data = data.OrderByDirection(state.SortDirection, o => o.Sign);

break;

case "name\_field":

data = data.OrderByDirection(state.SortDirection, o => o.Name);

break;

case "position\_field":

data = data.OrderByDirection(state.SortDirection, o => o.Position);

break;

case "mass\_field":

data = data.OrderByDirection(state.SortDirection, o => o.Molar);

break;

}

pagedData = data.Skip(state.Page \* state.PageSize).Take(state.PageSize).ToArray();

return new TableData<Element>() {TotalItems = totalItems, Items = pagedData};

}

private void OnSearch(string text) {

searchString = text;

table.ReloadServerData();

}

}

1. Creación de 1 nuevo componente en Components – Plans - **PlansTable.razor** + la clase c# del componente, copiaremos y pegaremos en este componente el ejemplo de tabla de mudblazor del punto 1:

ServerReloadAsync: recarga del servidor

@namespace PlannerApp.Components

@using PlannerApp.Shared.Models

//recuperaremos el resumen del plan PlanSummary

<MudTable ServerData="@(new Func<TableState, Task<TableData<PlanSummary>>>(**ServerReloadAsync**))"

Dense="true" Hover="true" @ref="\_table">

<ToolBarContent>

<MudSpacer />

//llamada de búsqueda

<MudTextField T="string" ValueChanged="@(s=>OnSearch(s))" Placeholder="Search" Adornment="Adornment.Start" AdornmentIcon="@Icons.Material.Filled.Search" IconSize="Size.Medium" Class="mt-0"></MudTextField>

</ToolBarContent>

<HeaderContent>

<MudTh><MudTableSortLabel SortLabel="nr\_field" T="PlanSummary">Cover</MudTableSortLabel></MudTh>

<MudTh><MudTableSortLabel SortLabel="sign\_field" T="PlanSummary">Title</MudTableSortLabel></MudTh>

<MudTh><MudTableSortLabel SortLabel="name\_field" T="PlanSummary">Description</MudTableSortLabel></MudTh>

<MudTh><MudTableSortLabel SortLabel="position\_field" T="PlanSummary">Actions</MudTableSortLabel></MudTh>

</HeaderContent>

<RowTemplate>

//columnas para las fila de datos

<MudTd DataLabel="Nr"><img src="@context.CoverUrl" style="width: 80px" /></MudTd>

<MudTd DataLabel="Sign">@context.Title</MudTd>

<MudTd DataLabel="Name">@context.Description</MudTd>

//columna cru

<MudTd DataLabel="Position">

<MudIconButton Icon="@Icons.Material.Filled.Preview" Color="Color.Success" OnClick="async () => await OnViewClicked.InvokeAsync(context)" />

<MudIconButton Icon="@Icons.Material.Filled.Edit" Color="Color.Default" OnClick="async () => await OnEditClicked.InvokeAsync(context)" />

<MudIconButton Icon="@Icons.Material.Filled.Delete" Color="Color.Error" OnClick="async () => await OnDeleteClicked.InvokeAsync(context)" />

</MudTd>

</RowTemplate>

<PagerContent>

<MudTablePager />

</PagerContent>

</MudTable>

1. Creación de la clase **PlansTable.razor**.cs para el componente creado previamente:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Microsoft.AspNetCore.Components;

using System.Net.Http;

using System.Net.Http.Json;

using Microsoft.AspNetCore.Components.Forms;

using Microsoft.AspNetCore.Components.Routing;

using Microsoft.AspNetCore.Components.Web;

using Microsoft.AspNetCore.Components.Web.Virtualization;

using Microsoft.AspNetCore.Components.WebAssembly.Http;

using Microsoft.AspNetCore.Components.Authorization;

using Microsoft.AspNetCore.Authorization;

using Microsoft.JSInterop;

using PlannerApp;

using PlannerApp.Shared;

using PlannerApp.Components;

using MudBlazor;

using Blazored.FluentValidation;

using PlaneerApp.Client.Services.Interfaces;

using PlannerApp.Shared.Models;

using AKSoftware.Blazor.Utilities;

namespace PlannerApp.Components {

public partial class PlansTable {

[Inject]

public IPlansService PlansService { get; set; }

[Parameter]

public EventCallback<PlanSummary> OnViewClicked { get; set; }

[Parameter]

public EventCallback<PlanSummary> OnDeleteClicked { get; set; }

//va a permitir que la función retrasada se administren por el padre en //PlanList.razor.cs, no lo escribimos para todos

[Parameter]

public EventCallback<PlanSummary> OnEditClicked { get; set; }

private string \_query = string.Empty;

protected override void OnInitialized() {

MessagingCenter.Subscribe<PlansList, PlanSummary>(this, "plan\_deleted", async (sender, args) => {

await \_table.ReloadServerData();

StateHasChanged();

});

}

//función que debemos implementar que viene con el componente table, cada //vez que cambie el estado (nº de pagina, tamaño de página, …) se llamará a esta función

private async Task<TableData<PlanSummary>> **ServerReloadAsync**(TableState state) {

//obtener los planes con las variables del objeto de mudblazor

var result = await PlansService.GetPlansAsync(\_query, state.Page, state.PageSize);

//con la respuesta crearemos el nuevo objeto TableData, con sus //propiedades Items y TotlaItems

return new TableData<PlanSummary> {

Items = result.Value.Records,

TotalItems = result.Value.ItemsCount

};

}

private MudTable<PlanSummary> \_**table**;

//implementar onSearch del componente código ya facilitado en el ejemplo

private void OnSearch(string query) {

\_query = query;

//para que recargue los datos hay que hacer una referencia al objeto //\_table definido en el componente razor:

//<MudTable ServerData="@(new Func<TableState, //Task<TableData<PlanSummary>>>(**ServerReloadAsync**))"

// Dense="true" Hover="true" @ref="\_**table**">

\_**table**.ReloadServerData();

}

}

}

# 18. Implement Plans Layout Switcher Between Cards List

Interruptor para cambiar entre modo de vista de Table y Tarjetas:

1. MudBlazor – Components – Butons – TroggleIconButton – Basic Usage – On/off, copiar el código y pegarlo en PlanList.razor.
2. Iconos: Components – Butons – Icon Button – copiaremos el código de Delete y pegarlo en PlanList.razor, crearemos 2 iconos cambiando la imagen de Delete por GridView y TableView:

**PlanList.razor:**

@namespace PlannerApp.Components

@\*UI View Selector \*@

<**MudIconButton** **Icon**="@Icons.Material.Filled.GridView" **Color**="Color.Primary" **Disabled**="\_isCardsViewEnabled" **OnClick**="SetCardsView"></**MudIconButton**>

<**MudIconButton** **Icon**="@Icons.Material.Filled.TableView" **Color**="Color.Primary" **Disabled**="@(!\_isCardsViewEnabled)" **OnClick**="SetTableView"></**MudIconButton**>

<div class="mt-4" />

@if (\_isCardsViewEnabled) {

<**PlanCardsList** **FetchPlans**="GetPlansAsync"

**OnEditClicked**="EditPlan"

**OnDeleteClicked**="DeletePlanAsync"/>

}

else {

<**PlansTable** **OnEditClicked**="EditPlan"

**OnDeleteClicked**="DeletePlanAsync" />

}

En PlanList.razor.cs añadiremos este cambio:

#region View Toggler

private bool \_isCardsViewEnabled = true;

private void SetCardsView() {

\_isCardsViewEnabled = true;

}

private void SetTableView() {

\_isCardsViewEnabled = false;

}

#endregion

# 19. Implement NavMenu

Creacion de 1 menu de opciones al lado izquierdo, crearemos 1 componente padre para cada pagina que tenga titulo, una ruta de navegación y 1 contenedor.

1. Creacion de 1 nuevo menú, mudblazor – components – Nav menú – copiar el código de Basic Usage y pegarlo en el menú de navegación de proyecto (shared - NavMenu.razor):

<**MudNavMenu**>

<**MudText** **Typo**="Typo.h6" **Class**="px-4">PlannerApp</**MudText**>

<**MudText** **Typo**="Typo.body2" **Class**="px-4 mud-text-secondary">V2.0 by AK Academy</**MudText**>

<**MudDivider** **Class**="my-2" />

<**MudNavLink** **Href**="/index">Home</**MudNavLink**>

<**MudNavLink** **Href**="/plans">Plans</**MudNavLink**>

<**MudNavLink** **Href**="/plans/add">Add Plan</**MudNavLink**>

<**MudNavLink** **Href**="/openitems">Not Done Items</**MudNavLink**>

<**MudNavLink** **Href**="/about">About</**MudNavLink**>

</**MudNavMenu**>

1. En Mainlayout.razor habilitar el menu:

<**NavMenu** />

1. Crear 1 carpeta en components – Layout para crear 1 componente en todas las páginas, donde se mostrara titulo, ruta de navegación y tiene 1 contenedor del sitio, asi que crearemos un nuevo componente en esta carpeta con el nombre: PAPage.razor + su archivo de clase
2. PAPage.razor:

@namespace PlannerApp.Components

<**MudContainer** **MaxWidth**="MaxWidth.ExtraLarge" **Class**="mt-2">

<**MudText** **Typo**="Typo.h4">@Title</**MudText**>

//lista de migas de pan (ruta de navegación)

//mudblazor – components – Breadcrumb – copiar el código del modelo custom //separator

<**MudBreadcrumbs** **Items**="BreadcrumbItems">

<**SeparatorTemplate**>

<**MudIcon** **Icon**="@Icons.Material.Filled.ArrowForward" **Size**="Size.Small" />

</**SeparatorTemplate**>

</**MudBreadcrumbs**>

@ChildContent

</**MudContainer**>

1. PAPage.razor.cs:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Microsoft.AspNetCore.Components;

using System.Net.Http;

using System.Net.Http.Json;

using Microsoft.AspNetCore.Components.Forms;

using Microsoft.AspNetCore.Components.Routing;

using Microsoft.AspNetCore.Components.Web;

using Microsoft.AspNetCore.Components.Web.Virtualization;

using Microsoft.AspNetCore.Components.WebAssembly.Http;

using Microsoft.AspNetCore.Components.Authorization;

using Microsoft.AspNetCore.Authorization;

using Microsoft.JSInterop;

using PlannerApp;

using PlannerApp.Shared;

using PlannerApp.Components;

using MudBlazor;

using Blazored.FluentValidation;

namespace PlannerApp.Components {

public partial class PAPage {

//fragmento de html o componente razor que puede ser renderizado, son componentes secundarios

[Parameter]

public RenderFragment ChildContent { get; set; }

[Parameter]

public string Title { get; set; }

[Parameter]

public List<BreadcrumbItem> BreadcrumbItems { get; set; } = new();

}

}

1. Ir a la 1 página Pages – Plans - **Plans.razor**

@page "/plans"

<**PAPage** **Title**="My Plans" **BreadcrumbItems**="\_breadcrumbItems">

<**PlansList** /> //se insertará en el contenedor del padre ChildContent

</**PAPage**>

1. Plans.razor.cs:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Microsoft.AspNetCore.Components;

using System.Net.Http;

using System.Net.Http.Json;

using Microsoft.AspNetCore.Components.Forms;

using Microsoft.AspNetCore.Components.Routing;

using Microsoft.AspNetCore.Components.Web;

using Microsoft.AspNetCore.Components.Web.Virtualization;

using Microsoft.AspNetCore.Components.WebAssembly.Http;

using Microsoft.AspNetCore.Components.Authorization;

using Microsoft.AspNetCore.Authorization;

using Microsoft.JSInterop;

using PlannerApp;

using PlannerApp.Shared;

using PlannerApp.Components;

using MudBlazor;

using Blazored.FluentValidation;

namespace PlannerApp.Pages.Plans {

public partial class Plans {

//migas de pan

private List<BreadcrumbItem> \_breadcrumbItems = new() {

new BreadcrumbItem("Home", "/index"), //nos lleva al indice

new BreadcrumbItem("Plans", "/plans", true) //desabilitado //porque ya estamos en esa página

};

}

}

# 20. Implement Edit - Create Plan Client Service Logic

CRU, edición y creación será básicamente los mismo con la excepción de que proporcionaremos id.

1. El método post de las api tiene esta estructura:

**POST​/api​/v2​/Plans**

Id string – no obligatorio en creación

Title \* required string

Description string

CoverFile string($binary) – fichero a cargar

CoverUrl string – url del achivo de la portada

ToDoItems array – Detalle completo del plan

**"toDoItems": [**

**{**

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

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

**"estimationDate": "2021-12-11T07:51:03.769Z",**

**"achievedDate": "2021-12-11T07:51:03.769Z",**

**"isDone": true,**

**"planId": "string"**

**}**

1. Simulación con postman creación de 1 plan:

* Logearnos en postman, new = post, url :

<https://plannerapp-api.azurewebsites.net/api/v2/auth/login>

Body – raw – JSON - Send

{

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

"password":"Test.123"

}

Copiar el token :

{

    "value": {

        "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..N2U--Jvd1OC0g2HnKN6QnZrUdyZa9o4n15zWG9EAdNw",

        "expiryDate": "2022-01-10T08:10:32Z"

    },

    "message": "Access token retrieved successfully",

    "isSuccess": **true**

Duplicar la pestaña en postam – botón derecha en la pestaña duplicate:

Cambiar la url: <https://plannerapp-api.azurewebsites.net/api/v2/plans>

Settings – Authorization – Type – Bearer – pegar el token

En body – form data – simularemos los campos del formulario para dar de alta un plan:

Title: Plan form Postman 69

Description: Example Postman

Antes de escribir CoverFile, cambiar el tipo de campo en vez de text – File – Select Files

Pulsar Send

{

    "value": {

        "id": "9f0a5a66-7238-4bd7-add3-c0676108bee8",

        "title": "Tennis Match",

        "description": "Tennis Match",

        "coverFile": **null**,

        "coverUrl": "https://akacademy.blob.core.windows.net/plannerapp/ballTennis-5e46495b-feee-478c-91e8-60bbb16161ff.jpg",

        "toDoItems": []

    },

    "message": "Plan created successfully",

    "isSuccess": **true**

}

Ahora en nuestra aplicación – plans estará el nuevo plan creado

1. Simulación con postman edición de 1 plan:

Put - <https://plannerapp-api.azurewebsites.net/api/v2/plans>

Title: Plan form Postman 69 -2

Description: Example Postman

Antes de escribir CoverFile, cambiar el tipo de campo en vez de text – File – Select Files

Id:9f0a5a66-7238-4bd7-add3-c0676108bee8 id copiado de la respuesta de la alta anterior

Al pulsar Send se guarda la modificación.

1. Modelo de datos del resumen del plan (PlanSummary.cs) en el proyecto PlannerApp.Shared – Models (contiene la información mínima para obtener la lista de planes):

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. Creación de la clase (modelo de datos) PlanDetail.cs en Models:

using Microsoft.AspNetCore.Http;

namespace PlannerApp.Shared.Models {

public class PlanDetail : PlanSummary {

//botón derecha IFormFile instalar esta interfaz que representa un //archivo que tiene 1 flujo y esto es lo que se envía al controlador

//como 1 archivo, irá del cliente al servidor

//<https://docs.microsoft.com/es-es/aspnet/core/mvc/models/file-uploads?view=aspnetcore-6.0>

//El archivo completo se lee en un [IFormFile](https://docs.microsoft.com/es-es/dotnet/api/microsoft.aspnetcore.http.iformfile), que es una representación de C# //del archivo que se usa para procesar o guardar el archivo.

//install package Microsoft.AspNetCore.Http.Features

public IFormFile CoverFile { get; set; }

// List of the to-dos

}

}

1. Creación de la clase (modelo de datos) FormFile.cs en Models que contendrá el archivo y su nombre:

using System.IO;

namespace PlannerApp.Shared.Models {

public class FormFile {

public FormFile(Stream fileStream, string fileName) {

FileStream = fileStream;

FileName = fileName;

}

//contendrá el archivo seleccionado por el usuario

public **Stream** FileStream { get; set; }

public string FileName { get; set; }

}

}

1. En el proyecto PlaneerApp.Client.Services – Interfaces - IPlansService.cs, crearemos 2 nuevas interficies:

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);

Task<ApiResponse<PlanDetail>> GetByIdAsync(string id);

//toma 1 modelo del detalle del plan PlanDetail y el archivo que //está en la clase FormFile creada en el paso anterior utilizando //stream con el archivo seleccionado por el usuario y con Http y //esta API se guardara en el servidor

Task<ApiResponse<PlanDetail>> **CreateAsync**(PlanDetail model, FormFile coverFile);

//única diferencia con Create es que se requiere id

Task<ApiResponse<PlanDetail>> **EditAsync**(PlanDetail model, FormFile coverFile);

Task DeleteAsync(string id);

}

}

1. Creación de 1 validador fluido (clase) en PlannerApp.Shared – Validators - PlanValidator.cs:

using FluentValidation;

using PlannerApp.Shared.Models;

//plan para el detalle del plan PlanDetail

namespace PlannerApp.Shared.Validators {

public class PlanValidator : AbstractValidator<**PlanDetail**> {

public PlanValidator() {

RuleFor(p => p.Title)

.NotEmpty()

.WithMessage("Title is required")

.MaximumLength(80)

.WithMessage("Title must be less than 80 characters");

RuleFor(p => p.Description)

.MaximumLength(500)

.WithMessage("Description must be less than 500 characters");

}

}

}

1. Implementar las 2 nuevas interficies creadas en el punto 7, CreateAsync y EditAsync, ir al archivo **HttpPlansService.cs** del proyecto PlaneerApp.Client.Services, en la definición de la clase:

public class HttpPlansService : IPlansService //botón derecho en IPlansService, implementar interficie para que nos cree los métodos asíncronos **CreateAsync y EditAsync**

public async Task<ApiResponse<PlanDetail>> **CreateAsync**(PlanDetail model, FormFile coverFile) {

var form = PreparePlanForm(model, coverFile, false);

var response = await \_httpClient.**PostAsync**("/api/v2/plans", form);

if (response.IsSuccessStatusCode) {

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

return result;

}

else {

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

throw new ApiException(errorResponse, response.StatusCode);

}

}

public async Task<ApiResponse<PlanDetail>> **EditAsync**(PlanDetail model, FormFile coverFile) {

//esta recuperando JSON y enviando un formulario

var form = PreparePlanForm(model, coverFile, **true**);

var response = await \_httpClient.**PutAsync**("/api/v2/plans", form);

if (response.IsSuccessStatusCode) {

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

return result;

}

Else {

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

throw new ApiException(errorResponse, response.StatusCode);

}

}

//devuelve el contenido Http con el cuerpo de la solicitud, parámetro //**isUpdate** para saber si es alta o modificacion

private HttpContent **PreparePlanForm**(PlanDetail model, FormFile coverFile, bool isUpdate) {

var form = new MultipartFormDataContent();

//nameof: obtener de forma rápida el nombre de una variable u operación

form.Add(new **StringContent**(model.Title), nameof(PlanDetail.Title));

if (!string.IsNullOrWhiteSpace(model.Description))

form.Add(new StringContent(model.Description), nameof(PlanDetail.Description));

**if (isUpdate)**

form.Add(new StringContent(**model.Id**), nameof(PlanDetail.Id));

if (coverFile != null)

form.Add(new **StreamContent**(coverFile.FileStream), nameof(model.CoverFile), coverFile.FileName);

return form;

}

# 21. Design the Plan Form

1. Creación del formulario, inicialmente para crear y luego para crear, en el proyecto PlannerApp – Components – Plans – agregar nuevo componente - **PlanForm.razor** + la clase:

MudBlazor.com – Components – Forms Inputs – Form – seleccionar el código de EditForm Support y pegar, también podemos copiar del formulario de LoginForm.razor

@namespace PlannerApp.Components

<**MudText** **Typo**="Typo.h6">Create a new Plan</**MudText**>

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

<**FluentValidationValidator** />

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

//para ver un cuadro de descripción mudblazor – components – Form input – //Text Field, copiar el texto de Multiline

<**MudTextField** **Label**="Description" **HelperText**="Optional" @bind-Value="@\_model.Description" **Lines**="4" **Variant**="Variant.Outlined" />

//para la carga de archivos mudblazor – components – file upload, en Use //any type of MudButton – Load Picture

//hay que poner en InputFile el archivo oculto, mudblazor utiliza para el //remplazo de este control el suyo MudFab, están relacionados, //posteriormente implementar la función OnChooseFileAsync

<**InputFile** id="fileInput" **OnChange**="OnChooseFileAsync" hidden multiple />

<**MudFab** **HtmlTag**="label"

**Color**="Color.Secondary"

**Icon**="@Icons.Filled.Image"

**Label**="Load picture"

for="fileInput" />

@if (\_isEditMode) {

<img src="@\_model.CoverUrl" style="width:75%;" class="m-2" />

}

//si hay archivo

@if (\_stream != null) {

<**MudText** **Typo**="Typo.subtitle2">@\_fileName</**MudText**>

}

//tratamiento de errores, mudblazor – components – Alert – Filled - Simple //alerts – copiar código del ejemplo: Meltdown is imminent

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

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

}

//linea

<**MudDivider** **DividerType**="DividerType.FullWidth" **Class**="my-2" />

<**ValidationSummary** /> //mostrará mensajes de error de validación fluida

<**MudButton** **Color**="Color.Primary" **Variant**="Variant.Filled" **ButtonType**="ButtonType.**Submit**" **FullWidth**="true" **Disabled**="\_isBusy">**Save**</**MudButton**>

//desabilitado mientras está ocupado para que el usuario no le de varias //veces

</**EditForm**>

1. PlanForm.razor.cs:

using Microsoft.AspNetCore.Components;

using Microsoft.AspNetCore.Components.Forms;

using MudBlazor;

using PlaneerApp.Client.Services.Interfaces;

using PlannerApp.Shared.Models;

using PlaneerApp.Client.Services.Exceptions;

using System.IO;

namespace PlannerApp.Components {

public partial class PlanForm {

[Inject]

public IPlansService PlansService { get; set; }

[Inject]

public NavigationManager Navigation { get; set; }

[Parameter]

public string Id { get; set; }

private bool \_isEditMode => Id != null;

private PlanDetail \_model = new PlanDetail();

private bool \_isBusy = false;

private Stream \_stream = null;

private string \_fileName = string.Empty;

private string \_errorMessage = string.Empty;

protected override async Task OnInitializedAsync() {

if (\_isEditMode)

await FetchPlanByIdAsync();

}

private async Task **SubmitFormAsync**() { //submit - envio del formulario

\_isBusy = true;

try {

**FormFile** formFile = null;

if (\_stream != null)

formFile = new FormFile(\_stream, \_fileName);

if (\_isEditMode)

await PlansService.**EditAsync**(\_model, formFile);

else

await PlansService.**CreateAsync**(\_model, formFile);

// Success

Navigation.NavigateTo("/plans");

}

catch (ApiException ex) {

\_errorMessage = ex.ApiErrorResponse.Message;

}

catch (Exception ex) {

// TODO: Log the error

\_errorMessage = ex.Message;

}

\_isBusy = false;

}

private async Task FetchPlanByIdAsync() {

\_isBusy = true;

try {

var result = await PlansService.GetByIdAsync(Id);

\_model = result.Value;

}

catch (ApiException ex) {

\_errorMessage = ex.ApiErrorResponse.Message;

}

catch (Exception ex) {

// TODO: Log the error

\_errorMessage = ex.Message;

}

\_isBusy = false;

}

//función a la que se llamara cuando se seleccione un formulario

private async Task **OnChooseFileAsync**(InputFileChangeEventArgs e) {

\_errorMessage = string.Empty;

var file = e.File;

if (file != null) {

if (file.Size >= 2097152) {

\_errorMessage = "The file must be equal or less than 2MB";

return;

}

string[] allowedExtensions = new[] { ".jpg", ".png", ".bmp", ".svg" };

string extension = Path.GetExtension(file.Name).ToLower();

if (!allowedExtensions.Contains(extension)) {

\_errorMessage = "Please choose a valid image file";

return;

}

//no leerá 1 archivo mas grande de 2 mg

using (var stream = file.OpenReadStream(2097152)) {

var buffer = new byte[file.Size];

await stream.ReadAsync(buffer, 0, (int)file.Size);

\_stream = new MemoryStream(buffer);

\_stream.Position = 0;

\_fileName = file.Name;

}

}

}

}

}

1. Crear 1 página del cru de planes por enrutamiento/controladores, añadir nuevo componente en Pages – Plans - **CreateEditPlan.razor** + clase:

@page "/plans/form"

@page "/plans/form/{id}"

<**PAPage** **Title**="Create/Edit a Plan" **BreadcrumbItems**="\_breadcrumbItems">

<**MudGrid**>

//en pantalla grande lg solo la mitad

<**MudItem** **lg**="6" **md**="8" **sm**="12">

<**PlanForm** **Id**="@Id"/>

</**MudItem**>

</**MudGrid**>

</**PAPage**>

1. **CreateEditPlan.razor.cs:**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Microsoft.AspNetCore.Components;

using System.Net.Http;

using System.Net.Http.Json;

using Microsoft.AspNetCore.Components.Forms;

using Microsoft.AspNetCore.Components.Routing;

using Microsoft.AspNetCore.Components.Web;

using Microsoft.AspNetCore.Components.Web.Virtualization;

using Microsoft.AspNetCore.Components.WebAssembly.Http;

using Microsoft.AspNetCore.Components.Authorization;

using Microsoft.AspNetCore.Authorization;

using Microsoft.JSInterop;

using PlannerApp;

using PlannerApp.Shared;

using PlannerApp.Components;

using MudBlazor;

using Blazored.FluentValidation;

namespace PlannerApp.Pages.Plans {

public partial class CreateEditPlan {

[Parameter]

public string Id { get; set; }

private List<BreadcrumbItem> \_breadcrumbItems = new() {

new BreadcrumbItem("Home", "/index"),

new BreadcrumbItem("Plans", "/plans"),

new BreadcrumbItem("Create", "/plans/form", true)

};

}

}

Hacer una prueba en la url: <https://localhost:7097/plans/form>

# 22. Implement the Plan Create Form Submit Logic

Enviar los datos del formulario con el método **SubmitFormAsync** de la clase PlanForm.razor.cs.

[Inject]

public NavigationManager Navigation { get; set; }

private async Task **SubmitFormAsync**() { //submit - envio del formulario

\_isBusy = true;

try {

**FormFile** formFile = null;

if (\_stream != null)

formFile = new FormFile(\_stream, \_fileName);

//enviar el modelo de datos vinculado al formulario + archivo

if (\_isEditMode)

await PlansService.**EditAsync**(\_model, formFile);

else

await PlansService.**CreateAsync**(\_model, formFile);

// Success

Navigation.NavigateTo("/plans");

}

//la API se encarga de devolver 1 excepción de este tipo en caso de error

catch (ApiException ex) {

\_errorMessage = ex.ApiErrorResponse.Message;

}

catch (Exception ex) {

// TODO: Log the error

\_errorMessage = ex.Message;

}

\_isBusy = false;

}

# 23. Implement Edit Functionality for Plan

Implementaremos la edición, desde la documentación de la API – Plans – PUT:

**PUT​/api​/v2​/Plans**

Id string

Title \* string

Description string

CoverFile string($binary)

CoverUrl string

ToDoItems array

Desde …/plans/ al pulsar en el botón editar se creará la url: …/plans/id abriendo el formulario CreateEditPlan.razor y rellenando los campos del mismo, posteriormente enviaremos los datos con el API Put.

Antes de empezar hay que implementar el API para obtener el plan que se quiere obtener por su id:

**GET​/api​/v2​/Plans​/{id}**

1. En PlaneerApp.Client.Services – Interfaces - IPlansService.cs, añadir:

Task<ApiResponse<PlanDetail>> GetByIdAsync(string id);

1. En HttpPlansService.cs implementar el cambio de esta interfaz:

public class HttpPlansService : IPlansService – botón derecha – implementar

public async Task<ApiResponse<PlanDetail>> GetByIdAsync(string id) {

var response = await \_httpClient.GetAsync($"/api/v2/plans/{id}");

if (response.IsSuccessStatusCode) {

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

return result;

}

Else {

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

throw new ApiException(errorResponse, response.StatusCode);

}

}

1. Cambios en PlanForm.razor.cs:

[Parameter]

public string Id { get; set; }

private bool \_isEditMode => Id != null;

[//implementar](file:///\\implementar) la búsqueda

private async Task FetchPlanByIdAsync() {

\_isBusy = true;

try {

var result = await PlansService.GetByIdAsync(Id);

\_model = result.Value;

}

catch (ApiException ex) {

\_errorMessage = ex.ApiErrorResponse.Message;

}

catch (Exception ex) {

// TODO: Log the error

\_errorMessage = ex.Message;

}

\_isBusy = false;

}

protected override async Task OnInitializedAsync() {

if (\_isEditMode)

await FetchPlanByIdAsync();

}

1. Cambios en PlanForm.razor:

Tenemos que mostrar la imagen si existe y si estamos editando:

@if (\_isEditMode) {

<img src="@\_model.CoverUrl" style="width:75%;" class="m-2" />

}

private async Task SubmitFormAsync() {

…

if (\_isEditMode)

await PlansService.EditAsync(\_model, formFile);

else

await PlansService.CreateAsync(\_model, formFile);

…

1. En PlanCardList.razor implementar la acción del botón editar:

<**MudGrid** **Spacing**="2">

@if (\_result != null && \_result.Records != null) {

foreach (var plan in \_result.Records) {

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

<**PlanCard** **PlanSummary**="plan"

**OnEditClicked**="() => OnEditClicked.InvokeAsync(plan)"

**OnDeleteClicked**="() => OnDeleteClicked.InvokeAsync(plan)"/>

</**MudItem**>

}

}

</**MudGrid**>

1. En PlanCardList.razor.cs:

[Inject]

public NavigationManager Navigation { get; set; }

[Parameter]

public EventCallback<PlanSummary> OnEditClicked { get; set; }

1. En Pages – Plans – Plans.razor añadir la nueva ruta:

@page "/plans"

//@page "/plans/{id}" //error no funciona

1. En CreateEditPlan.razor:

@page "/plans/form"

@page "/plans/form/{id}"

<**PAPage** **Title**="Create/Edit a Plan" **BreadcrumbItems**="\_breadcrumbItems">

<**MudGrid**>

<**MudItem** **lg**="6" **md**="8" **sm**="12">

<**PlanForm** **Id**="@Id"/> //parámetro id

</**MudItem**>

</**MudGrid**>

</**PAPage**>

# 24. Finalizing Edit Plan

Refactorizaremos para que la edición del modo de vista en tablas y tarjetas se haga desde el mismo sitio, después de editar y salvar volveremos a la vista desde donde partia el usuario.

Ahora mismo la funcionalidad de la vista en tarjetas está en la propia vista (PlanCardList.razor) por lo que se tendría que volver a implementar en la vista de tablas.

1. Actualmente en **PlanCardList.razor**.**cs** al hacer un click en editar tiene su propia funcionalidad, vamos a hacer que se ejecute en su componente padre (PlanList.razor) para aprovechar el código (escribirlo 1 unica vez) para vista en tablas:

Sustituiremos esto:

private void EditPlan(PlanSummary plan) {

Navigation.NavigateTo($"/plans/form/{plan.Id}");

}

Por esto:

[Parameter]

public EventCallback<PlanSummary> OnEditClicked { get; set; }

1. Modificar el enlace de **PlanCardList.razor:**

<**MudIconButton** **Icon**="@Icons.Material.Filled.Edit" **Color**="Color.Default" **OnClick**="async () => await OnEditClicked.InvokeAsync(PlanSummary)" />

1. Hacer lo mismo en PlanTableRazor.cs:

[Parameter]

public EventCallback<PlanSummary> OnEditClicked { get; set; }

1. Implementar en el padre PlanList.razor:

@if (\_isCardsViewEnabled) {

<**PlanCardsList** **FetchPlans**="GetPlansAsync"

**OnEditClicked**="**EditPlan**"

**OnDeleteClicked**="DeletePlanAsync"/>

}

else {

<**PlansTable** **OnEditClicked**="**EditPlan**"

**OnDeleteClicked**="DeletePlanAsync" />

}

1. En PlanList.razor.cs:

[Inject]

public NavigationManager Navigation { get; set; }

#region Edit

private void **EditPlan**(PlanSummary plan) {

Navigation.NavigateTo($"/plans/form/{plan.Id}");

}

#endregion

# 25. Implement Delete Plan API Call

Implementar el código que se comunicará con la API con la eliminación del lado del servidor:

**DELETE​/api​/v2​/Plans​/{id}**

Success

{

"message": "string",

"isSuccess": true

}

1. Implementar la interface IPlansService porque es donde se agregar las abstracciones de las comunicaciones de la API, no nos interesa la respuesta, solo que se ha realizado correctamente:

Task DeleteAsync(string id);

1. Implementar la interficie en HttpPlansService.cs con f12 o editando la clase:

public async Task DeleteAsync(string id) {

var response = await \_httpClient.DeleteAsync($"/api/v2/plans/{id}");

if (!response.IsSuccessStatusCode) {

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

throw new ApiException(errorResponse, response.StatusCode);

}

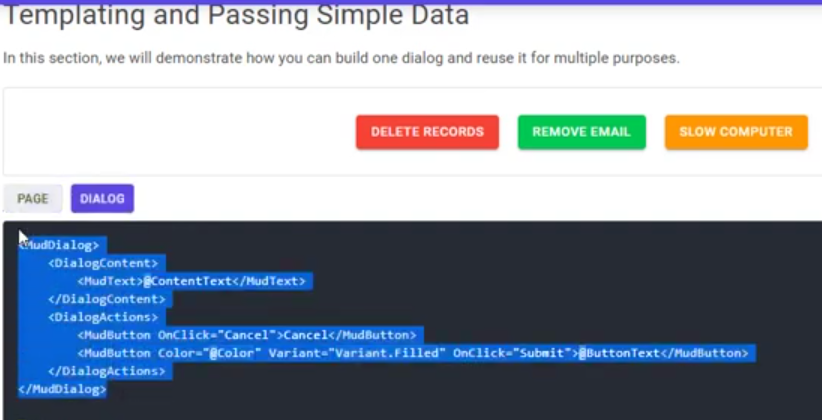
}

Posteriormente llamaremos a esta función anterior con confirmación y recargaremos la página una vez eliminado el registro.

# 26. Implementing Deleting the Plan Logic with Confirmation Dialog

Implementar la eliminación de la lista de planes cuando haga click en eliminar se mostrará 1 cuadro de dialogo de mudblazor, después de la eliminación del registro – se notificará a la lista y tabla de tarjetas para que se actualice, cuando hagamos click en eliminar llamada a la API.

1. Mudblazor – Components – Dialog – Templating and passing simple data – código del botón Delete Records, apartado Dialog copiar el código de diseño del componente y pegar en 1 nuevo componente **ConfirmationDialog.razor** que se creará en una nueva carpeta Components - **Dialogs**



@namespace PlannerApp.Components

<**MudDialog**>

<**DialogContent**>

<**MudText**>@ContentText</**MudText**> //contexto del contenido

</**DialogContent**>

<**DialogActions**>

<**MudButton** **OnClick**="Cancel">Cancel</**MudButton**>

<**MudButton** **Color**="@Color" **Variant**="Variant.Filled" **OnClick**="Submit">@ButtonText</**MudButton**>

</**DialogActions**>

</**MudDialog**>

@code {

//instancia como parámetro de cascada, este cuadro de dialogo sirve también para advertencias.

[CascadingParameter] MudDialogInstance MudDialog { get; set; }

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

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

[Parameter] public Color Color { get; set; }

//si el resultado está bien se elimina

void Submit() => MudDialog.Close(DialogResult.Ok(true));

void Cancel() => MudDialog.Cancel();

}

1. Ahora tenemos el diseño del cuadro de dialogo, del mismo componente de Mudblazor – cuadro de dialogo – seleccionaremos el código de **Page** y lo pegaremos en el Proyecto PlannerApp – Components – Plans – **PlanList.razor.cs**:

[Inject]

public IDialogService DialogService { get; set; }

#region Delete

private async Task DeletePlanAsync(PlanSummary plan) {

var parameters = new DialogParameters();

//3 parámetros: texto del contenido interior, acción y color

parameters.Add("ContentText", $"Do you really want to delete the plan '{plan.Title}'?");

parameters.Add("ButtonText", "Delete");

parameters.Add("Color", Color.Error);

//opción de cierre del cuadro de dialogo

var options = new DialogOptions() { CloseButton = true, MaxWidth = MaxWidth.ExtraSmall };

//llamada al componente ConfirmationDialog creado previamente

var dialog = DialogService.Show<**ConfirmationDialog**>("Delete", parameters, options);

//en mudblazor – Components – Dialog – Passing Data – copiar el código

//para el control de la acción seleccionada por el usuario

var confirmationResult = await dialog.Result;

if (!confirmationResult.Cancelled) {

// Confirmed to delete

Try {

await PlansService.DeleteAsync(plan.Id);

// Send a message about the deleted plan

MessagingCenter.Send(this, "plan\_deleted", plan);

}

catch (ApiException ex) {

// TODO: Log this error

}

catch (Exception ex) {

// TODO: Log this error

}

}

}

#endregion

1. Componente hijo de tarjetas PlanCardList.razor.cs:

[Parameter]

public EventCallback<PlanSummary> OnDeleteClicked { get; set; }

1. PlanCardList.razor:

foreach (var plan in \_result.Records) {

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

<**PlanCard** **PlanSummary**="plan"

OnEditClicked="() => OnEditClicked.InvokeAsync(plan)"

**OnDeleteClicked**="() => **OnDeleteClicked**.InvokeAsync(plan)"/>

</**MudItem**>

}

1. Componente hijo de tabla PlansTable.razor.cs que se ejecutará en el padre:

[Parameter]

public EventCallback<PlanSummary> OnDeleteClicked { get; set; }

1. PlansTable.razor:

<**MudIconButton** **Icon**="@Icons.Material.Filled.Delete" **Color**="Color.Error"

**OnClick**="async () => await OnDeleteClicked.InvokeAsync(context)" />

1. Por último en el componente padre PlansList.razor:

@if (\_isCardsViewEnabled) {

<**PlanCardsList** **FetchPlans**="GetPlansAsync"

**OnEditClicked**="EditPlan"

**OnDeleteClicked**="DeletePlanAsync"/>

}

else {

<**PlansTable** **OnEditClicked**="EditPlan"

**OnDeleteClicked**="DeletePlanAsync" />

}

Funciona, pero queda pendiente notificar a los componentes hijos que se actualicen para que no se muestre el plan eliminado.

# 27. MessagingCenter with Blazor To Notify Plans List To Refresh After Delete

Actualizar componentes hijos después de la eliminación, llamando al centro de mensajería simplificando la comunicación entre mensajes incluso sin usar devoluciones de llamada o parámetros.

En <https://github.com/aksoftware98/blazor-utilities> está disponible Messaging Center Blazor Utitlity junto con 1 explicación para el envío de datos a través de sus componentes o en el anexo 2.

Después de eliminar desde PlanCardsList.razor lo tiene que notificar y actualizarse también PlansTable.razor.

Pasos a seguir:

1. En el proyecto PlannerApp – Dependencias – Manage Nuget Package – Examinar: AkSoftware.Blazor.Utilities – Instalar
2. PlannerList.razor.cs (encargado de enviar el mensaje):

using AKSoftware.Blazor.Utilities;

#region Delete

private async Task DeletePlanAsync(PlanSummary plan) {

var parameters = new DialogParameters();

parameters.Add("ContentText", $"Do you really want to delete the plan '{plan.Title}'?");

parameters.Add("ButtonText", "Delete");

parameters.Add("Color", Color.Error);

var options = new DialogOptions() { CloseButton = true, MaxWidth = MaxWidth.ExtraSmall };

var dialog = DialogService.Show<ConfirmationDialog>("Delete", parameters, options);

var confirmationResult = await dialog.Result;

if (!confirmationResult.Cancelled) {

// Confirmed to delete

Try {

await PlansService.DeleteAsync(plan.Id);

// Centro de mensajería: El código toma como parámetro el objeto remitente this, el filtro del mensaje como en este ejemplo "plan\_deleted" que será utilizado por otros componentes para recibir el mensaje, y el último parámetro es el valor a enviar (plan que se ha eliminado).

// Send a message about the deleted plan

MessagingCenter.Send(this, "plan\_deleted", plan);

}

catch (ApiException ex) {

// TODO: Log this error

}

catch (Exception ex) {

// TODO: Log this error

}

}

}

#endregion

1. Receptor de mensajes (tabla **PlansTable.razor.cs**):

using AKSoftware.Blazor.Utilities;

protected override void OnInitialized() {

//remitente PlansList + tipo de argumento PlanSummary

MessagingCenter.Subscribe<PlansList, PlanSummary>(this, "plan\_deleted", async

//definimos el remitente

(sender, args) => {

//recargamos los datos de la tabla

await \_table.ReloadServerData();

StateHasChanged();

});

}

1. Receptor de mensajes en la lista de tarjetas (PlanCardsList.razor.cs):

using AKSoftware.Blazor.Utilities;

protected override void OnInitialized() {

MessagingCenter.Subscribe<PlansList, PlanSummary>(this, "plan\_deleted", async (sender, args) => {

await GetPlansAsync(\_pageNumber);

StateHasChanged();

});

}

# 28. Implement Plan Details Dialog to View Plan With Items

Lista de elementos de tareas pendientes, recuperaremos la estructura de la documentación de la API:

**"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"**

**}**

En el proyecto PlaneerApp.Client.Services – Interfaces - IPlansService.cs:

using PlannerApp.Shared.Models;

using PlannerApp.Shared.Responses;

namespace PlaneerApp.Client.Services.Interfaces {

public interface IPlansService {

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

Task<ApiResponse<PlanDetail>> GetByIdAsync(string id);

Task<ApiResponse<PlanDetail>> CreateAsync(PlanDetail model, FormFile coverFile);

Task<ApiResponse<PlanDetail>> EditAsync(PlanDetail model, FormFile coverFile);

Task DeleteAsync(string id);

}

}

Tenemos 2 tipos de datos PlanSummary (resumen) y **PlanDetail**.

1. Copiar la estructura de toDoItems del API, en 1 nueva clase o en la clase del modelo de datos PlanDetail.cs:

{ "id": "string",

"description": "string",

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

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

"isDone": true,

"planId": "string" }

1. Pegaremos desde el menú Edit – Pegado especial – Pegar

public class ToDoItemDetail {

public string Id { get; set; }

public string Description { get; set; }

//public DateTime EstimationDate { get; set; }

//public DateTime AchievedDate { get; set; }

public bool IsDone { get; set; }

public string PlanId { get; set; }

}

Provisionalmente comentamos los 2 campos fechas y creamos la nueva clase desde las suguerencias de la parte izquierda – Mover tipo a TodoItemDetail.cs

1. En PlanDetail.cs de la carpeta Models:

using Microsoft.AspNetCore.Http;

namespace PlannerApp.Shared.Models {

public class PlanDetail : PlanSummary {

public IFormFile CoverFile { get; set; }

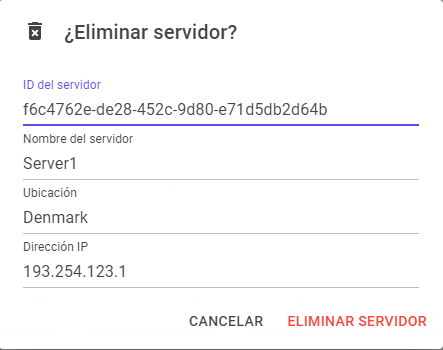
// List of the to-dos

public List<ToDoItemDetail> toDoItems { get; set; }

}

}

1. Mostraremos la página de detalle en un cuadro de dialogo, desde MudBlazor – Components – Dialog – seleccionar el código del ejemplo asociado al botón Server1 Delete desde el apartado Dialog en la parte inferior



1. Crear 1 nuevo componente en PlannerApp – Components – Plans - **PlanDetailsDialog**.**razor**:

(o shift f2 para crear 1 nuevo archivo) ¿?

Dividir **PlanDetailsDialog**.**razor** en 2 secciones, usaremos el elemento cuadricula de MudBlazor <MudGrid> + <MudItem>, este cuado de dialogo se le llamará desde varios lugares.

@namespace PlannerApp.Components

<**MudDialog**>

<**TitleContent**>

<**MudText** **Typo**="Typo.h6">

//no acaba de funcionar siempre cargando ¿??

@if (\_isBusy){

<text>Loading Plan Details ...</text>

}

else {

<text>@\_plan?.Title</text>

}

</**MudText**> </**TitleContent**>

<**DialogContent**>

<**MudGrid**>

<**MudItem** **sm**="12" **md**="6">

<!-- Utilizar un skeleton mientras aun está cargando -->

@if (\_isBusy) {

<**MudSkeleton** **Animation**="Animation.Wave" **SkeletonType**="SkeletonType.Text" **Style**="min-width:250px"/>

<**MudSkeleton** **Animation**="Animation.Wave" **SkeletonType**="SkeletonType.Text" **Style**="min-width:250px"/>

<**MudSkeleton** **Animation**="Animation.Wave" **SkeletonType**="SkeletonType.Text" **Style**="min-width:250px"/>

<**MudSkeleton** **Animation**="Animation.Wave" **SkeletonType**="SkeletonType.Text" **Style**="min-width:250px"/>

<**MudSkeleton** **Animation**="Animation.Wave" **SkeletonType**="SkeletonType.Text" **Style**="min-width:250px"/>

}

else {

<!-- Plan details obsoleto -->

@\*<img style="width:100%;border-radius:5px" src="@\_plan.CoverUrl" />

<MudText Typo="Typo.h5">@\_plan?.Title</MudText>

<MudText Typo="Typo.body1">@\_plan?.Description</MudText>

<MudText Typo="Typo.body1">@\_plan?.Description</MudText>\*@

}

</**MudItem**>

<**MudItem** **sm**="12" **md**="6">

<!-- Items go here -->

</**MudItem**>

</**MudGrid**>

</**DialogContent**>

<**DialogActions**>

<**MudButton** **OnClick**="Close">Close</**MudButton**>

</**DialogActions**>

</**MudDialog**>

1. Crear la clase del componente anterior: **PlanDetailsDialog**.**razor.cs:**

using Blazored.FluentValidation;

using Microsoft.AspNetCore.Components;

using MudBlazor;

using PlaneerApp.Client.Services.Exceptions;

using PlaneerApp.Client.Services.Interfaces;

using PlannerApp.Shared.Models;

using System.Reflection.Metadata;

namespace PlannerApp.Components {

public partial class PlanDetailsDialog {

[CascadingParameter]

MudDialogInstance MudDialog { get; set; }

[Inject]

public IPlansService PlansService { get; set; }

//parametro utilizado para recuperar el plan

[Parameter]

public string PlanId { get; set; }

private PlanDetail \_plan;

private bool \_isBusy;

private string \_errorMessage = string.Empty;

private void Close() {

MudDialog.Cancel();

}

//OnParametersSet y OnParametersSetAsync: son ejecutados cuando el componente ha recbido todos los parámetros y sus valores han sido asignado

//a sus respectivas propiedades, aunque esto de tener los parámetros asignados ocurre también en la etapa de inicialización,

//los métodos OnParametersSet se ejecutan cada vez que se actualizan los parámetros a diferencia de los métodos OnInitialized

//que se ejecutan 1 única vez.

protected override void OnParametersSet() {

//verificar la identificación del plan

if (PlanId == null)

throw new ArgumentNullException(nameof(PlanId));

base.OnParametersSet();

}

protected async override Task OnInitializedAsync() {

await FetchPlanAsync();

}

private async Task FetchPlanAsync() {

\_isBusy = true;

try {

var result = await PlansService.GetByIdAsync(PlanId);

\_plan = result.Value;

StateHasChanged();

}

catch (ApiException ex) {

//todo

}

catch (Exception ex) {

//todo: Log this error

}

\_isBusy = false;

}

} }

1. Llamar al cuadro de dialogo creado anteriormente desde la lista de tarjetas al pulsar en el botón detalle, la funcionalidad se creará en su componente padre **PlanList.razor.cs** para centralizarlo, abriremos 1 nueva sección y consistirá en mostrar un cuadro de dialogo:
2. En PlansTable.razor comprobar que se llama al método OnViewClicked que se ejecutará en el padre:

<**MudIconButton** **Icon**="@Icons.Material.Filled.Preview" **Color**="Color.Success"

**OnClick**="async () => await **OnViewClicked**.InvokeAsync(context)" />

1. En PlansTable.razor.cs:

[Parameter]

public EventCallback<PlanSummary> OnViewClicked { get; set; }

1. Comprobar lo mismo para PlanCardList.razor

<**MudGrid** **Spacing**="2">

@if (\_result != null && \_result.Records != null) {

foreach (var plan in \_result.Records) {

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

<**PlanCard** **PlanSummary**="plan"

**OnEditClicked**="() => OnEditClicked.InvokeAsync(plan)"

**OnDeleteClicked**="() => OnDeleteClicked.InvokeAsync(plan)"

**OnViewClicked**="() => OnViewClicked.InvokeAsync(plan)"

/>

</**MudItem**>

}

}

</**MudGrid**>

1. En PlanCardsList.razor.cs:

[Parameter]

public EventCallback<PlanSummary> OnViewClicked { get; set; }

1. PlanList.razor:

@if (\_isCardsViewEnabled) {

<**PlanCardsList** **FetchPlans**="GetPlansAsync"

**OnEditClicked**="EditPlan"

**OnDeleteClicked**="DeletePlanAsync"

**OnViewClicked**="ViewPlan"/>

}

else {

<**PlansTable** **OnEditClicked**="EditPlan"

**OnDeleteClicked**="DeletePlanAsync"

**OnViewClicked**="ViewPlan"/>/>

}

1. En PlanList.razor.cs:

#region View

private void ViewPlan(PlanSummary plan) {

var parameters = new DialogParameters();

parameters.Add("PlanId", plan.Id);

var options = new DialogOptions() { CloseButton = true, MaxWidth = MaxWidth.Medium };

var dialog = DialogService.Show<PlanDetailsDialog>("Details", parameters, options);

}

#endregion

# 29. Upgrade the Blazor App to .NET 6 And Implement New Features

En el cuadro de dialogo de la izquierda se mostraron los detalles y en el de la derecha implementaremos la funcionalidad de crear, editar y eliminar.

Actualizaremos la aplicación a Net 6 desde el nombre del proyecto PlanneApp doble clic :

<TargetFramework>net**6**.0</TargetFramework>

Tambien cambiar las bibliotecas de Microsoft a 6.0:

<ItemGroup>

<PackageReference Include="AKSoftware.Blazor.Utilities" Version="1.0.0" />

<PackageReference Include="Blazored.FluentValidation" Version="2.0.3" />

<PackageReference Include="Blazored.LocalStorage" Version="4.1.5" />

<PackageReference Include="Microsoft.AspNetCore.Components.Authorization"

Version="6.0.0" />

<PackageReference Include="Microsoft.AspNetCore.Components.WebAssembly"

Version="6.0.0" />

<PackageReference

Include="Microsoft.AspNetCore.Components.WebAssembly.DevServer"

Version="6.0.0" PrivateAssets="all" />

<PackageReference Include="Microsoft.Extensions.Http" Version="6.0.0" />

<PackageReference Include="MudBlazor" Version="6.0.2" />

<PackageReference Include="System.IdentityModel.Tokens.Jwt" Version="6.14.1" />

</ItemGroup>

Haremos lo mismo con el resto de proyectos.

Gracias a esta configuración de program.cs: builder.RootComponents.Add<HeadOutlet>("head::after");

Podemos añadir a las paginas razor esta etiqueta, en index.razor:

<**PageTitle**>Home | PlannerApp</**PageTitle**>

//al ejecutar pone el titulo en la pestaña

En CreateEditPlan.razor:

<**PageTitle**>@(Id == null ? "Create" : "Edit") | PlannerApp</**PageTitle**>

# 30. Designing the ToDo Item Card, Using Hot Reload And Setup Items Layout

|  |  |
| --- | --- |
|  |  |

Al hacer un clic en el botón botón verde de vista se pueden ver y mantener los detalles del plan.

1. Crear 1 nueva carpeta en Components – ToDoItems: donde crearemos los componentes relacionados con el plan.
2. Nuevo componente en Components – ToDoItems – **ToDoItem.razor:**

Este componente que se visualizará en la parte de la derecha de la vista de detalle, se verá como 1 tarjeta con 1 checkbox, texto modificable al hacer un click, botón eliminar, cancelar y guardar:

@namespace PlannerApp.Components

[//mudblazor](file:///\\mudblazor) – features – flex (justificar, alineación, …) permite poner elementos en vertical u horizontal, envuelto de 1 manera muy flexible, necesitaremos que esté alineado en el centro, fila flexible, …

//margenes y padding en [mudblazor](file:///\\mudblazor) – features – spacing, shrink = reducir

<**MudPaper** **Elevation**="1" **Class**="mt-2 px-3">

<div class="d-flex **flex-row align-center**">

[//mudblazor](file:///\\mudblazor) – components – form inputs & controls – checkbox , escogeremos de la categoría basic checkboxes el 2 copiar código + clases como mr-2 para mantener margen por la derecha a la etiqueta

<**MudCheckBox** @bind-Checked="@\_isChecked" **Color**="Color.Primary"

**Class**="flex-grow-0 flex-shrink-0 mr-2"></**MudCheckBox**>

//si estamos en modo de edición

@if (\_isEditMode) {

//textfiel de mudblazor – dense – outline

<**MudTextField** @bind-Value="\_description" **Label**="Outlined"

**Variant**="Variant.Outlined" **Margin**="Margin.Dense" **Class**="flex-grow-1 flex-shrink-0 mr-2" ></**MudTextField**>

<**MudIconButton** **Icon**="@Icons.Material.Filled.Cancel" **Color**="Color.Primary"

**Class**="flex-grow-0 flex-shrink-0 mr-1"

**OnClick**="() => ToggleEditMode(true)" /> //cancela modo edición

<**MudIconButton** **Icon**="@Icons.Material.Filled.Save" **Color**="Color.Success"

**Class**="flex-grow-0 flex-shrink-0 mr-2"/> }

else {

//la etiqueta descripción al hacer un clic pasa a modo edición

<**MudText** **Typo**="Typo.body1" **Class**="flex-grow-1 flex-shrink-0 mr-2"

**Style**="cursor:pointer" @onclick="() => ToggleEditMode(false)">

@Item.Description

</**MudText**>

@if (\_isBusy) {

<**MudProgressCircular** **Color**="Color.Primary" **Indeterminate**="true" **Class**="flex-grow-0 flex-shrink-0 mr-2"/>

}

else {

//mudblazor – components – buttons – iconButtons – simple icons buttons – código delete

<**MudIconButton** **Icon**="@Icons.Material.Filled.Delete" **Color**="Color.Error" **Class**="flex-grow-0 flex-shrink-0 mr-2"

**OnClick**="RemoveItemAsync"/>

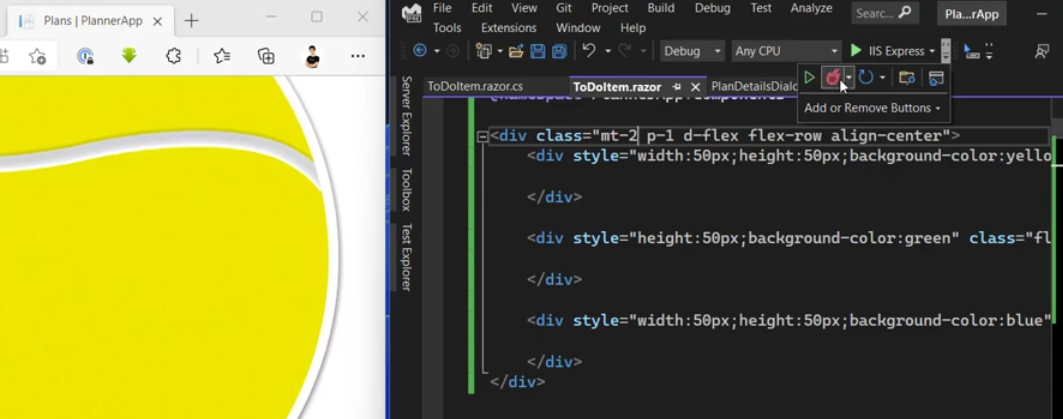
}

}

</div>

</**MudPaper**>

Nota: podemos ir diseñando y visualizando el contenido sin necesidad de compilar, en visual studio 2022 recarga activa (botón rojo):



1. Implementar **ToDoItem.razor.cs:**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Microsoft.AspNetCore.Components;

using System.Net.Http;

using System.Net.Http.Json;

using Microsoft.AspNetCore.Components.Forms;

using Microsoft.AspNetCore.Components.Routing;

using Microsoft.AspNetCore.Components.Web;

using Microsoft.AspNetCore.Components.Web.Virtualization;

using Microsoft.AspNetCore.Components.WebAssembly.Http;

using Microsoft.AspNetCore.Components.Authorization;

using Microsoft.AspNetCore.Authorization;

using Microsoft.JSInterop;

using PlannerApp;

using PlannerApp.Shared;

using PlannerApp.Components;

using MudBlazor;

using Blazored.FluentValidation;

using PlannerApp.Shared.Models;

using PlaneerApp.Client.Services.Interfaces;

using PlaneerApp.Client.Services.Exceptions;

namespace PlannerApp.Components {

public partial class ToDoItem {

[Inject]

public IToDoItemsService ToDoItemsService { get; set; }

[Parameter] //tarea pendientes, detalle del plan

public ToDoItemDetail Item { get; set; }

[Parameter]

public EventCallback<ToDoItemDetail> OnItemDeleted { get; set; }

private bool \_isChecked = true;

private bool \_isEditMode = false;

private bool \_isBusy = false;

private string \_description = String.Empty;

private void **ToggleEditMode**(bool isCancel) {

if (\_isEditMode) {

\_isEditMode = false;

\_description = isCancel ? Item.Description : \_description;

}

else {

\_isEditMode = true;

\_description = Item.Description;

}

}

private async Task **RemoveItemAsync**() {

try {

\_isBusy = true;

// Call the API to add the item

await ToDoItemsService.DeleteAsync(Item.Id);

// Notify the parent about the newly added item

await OnItemDeleted.InvokeAsync(Item);

}

catch (ApiException ex) {

// TODO: Handle errors globally

}

catch (Exception ex)

{

// TODO: Handle errors globally

}

\_isBusy = false;

}

}

}

# 31. Implement the Logic of ToDo Items Http Client To Communicate With API

Implementar la lógica de los elementos del plan, desde la documentación del web api v2 – ToDos (sección tareas pendientes),

#### ToDos

**GET​/api​/v2​/ToDos​/pendings**

**PUT​/api​/v2​/ToDos**

**PUT​/api​/v2​/ToDos​/Toggle​/{id}**

**DELETE​/api​/v2​/ToDos​/{id}**

**POST​/api​/v2​/ToDos: para dar de alta una tarea necesitaremos el id del plan y 1 descripción, lo mismo que para Put**

{

"id": "string",

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

"estimationDate": "2021-12-21T11:31:09.566Z",

"achievedDate": "2021-12-21T11:31:09.566Z",

"isDone": true,

"**planId**": "string"

}

1. Creación de la interficie que servirá para implementar la API para los elementos del plan en PlaneerApp.Client.Services – Interfaces - IToDoItemsService.cs:

using PlannerApp.Shared.Models;

using PlannerApp.Shared.Responses;

using System.Threading.Tasks;

namespace PlaneerApp.Client.Services.Interfaces {

public interface IToDoItemsService {

Task<ApiResponse<ToDoItemDetail>> **CreateAsync**(string description,

string planId);

Task<ApiResponse<ToDoItemDetail>> **EditAsync**(string id, string

newDescription, string planId);

Task ToggleAsync(string id);

Task **DeleteAsync**(string id);

}

}

1. Creación de servicio http de tareas (clase): **HttpToDoItemsService.cs** en la raíz del proyecto PlaneerApp.Client.Services que implementa la interfaz creada en el punto anterior, es una copia del servicio creado para planes: HttpPlansService.cs:

using PlaneerApp.Client.Services.Exceptions;

using PlaneerApp.Client.Services.Interfaces;

using PlannerApp.Shared.Models;

using PlannerApp.Shared.Responses;

using System.Net.Http;

using System.Net.Http.Json;

using System.Threading.Tasks;

namespace PlaneerApp.Client.Services {

public class HttpToDoItemsService : IToDoItemsService {

private readonly HttpClient \_httpClient;

public HttpToDoItemsService(HttpClient httpClient) {

\_httpClient = httpClient;

}

public async Task<ApiResponse<ToDoItemDetail>> **CreateAsync**

(string description, string planId) {

var response = await \_httpClient.PostAsJsonAsync("/api/v2/todos",

new {

PlanId = planId,

Description = description

});

if (response.IsSuccessStatusCode) {

var result = await

response.Content.ReadFromJsonAsync<ApiResponse<ToDoItemDetail>>();

return result;

}

else {

var errorResponse = await

response.Content.ReadFromJsonAsync<ApiErrorResponse>();

throw new ApiException(errorResponse, response.StatusCode);

}

}

public async Task<ApiResponse<ToDoItemDetail>> **EditAsync**(string id,

string description, string planId) {

var response = await \_httpClient.PutAsJsonAsync("/api/v2/todos",

new {

PlanId = planId,

Description = description,

Id = id

});

if (response.IsSuccessStatusCode) {

var result = await

response.Content.ReadFromJsonAsync<ApiResponse<ToDoItemDetail>>();

return result;

}

else {

var errorResponse = await

response.Content.ReadFromJsonAsync<ApiErrorResponse>();

throw new ApiException(errorResponse, response.StatusCode);

}

}

public async Task **DeleteAsync**(string id) {

var response = await

\_httpClient.DeleteAsync($"/api/v2/todos/{id}");

if (!response.IsSuccessStatusCode) {

var errorResponse = await

response.Content.ReadFromJsonAsync<ApiErrorResponse>();

throw new ApiException(errorResponse, response.StatusCode);

}

}

public async Task **ToggleAsync**(string id) {

var response = await

\_httpClient.PutAsJsonAsync<object>($"/api/v2/todos/toggle/{id}", null);

if (!response.IsSuccessStatusCode) {

var errorResponse = await

response.Content.ReadFromJsonAsync<ApiErrorResponse>();

throw new ApiException(errorResponse, response.StatusCode);

}

}

}

}

1. Inyectar el servicio de elementos como nueva dependencia en: DependencyInjectionExtensions.cs:

public static IServiceCollection AddHttpClientServices

(this IServiceCollection services) {

return services.AddScoped<IAuthenticationService,

HttpAuthenticationService>()

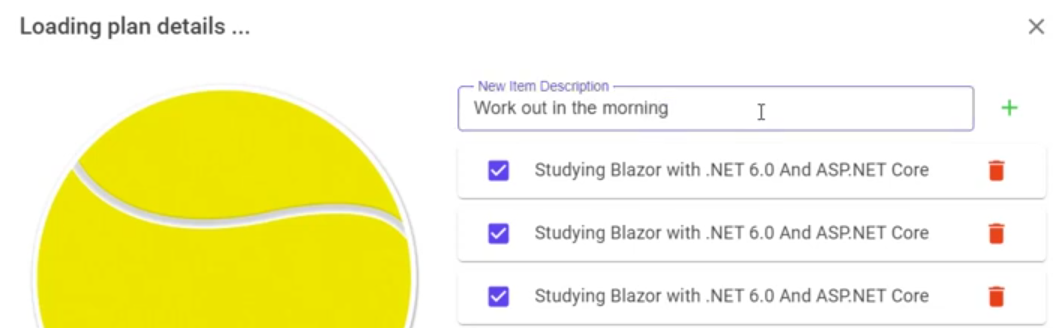
.AddScoped<IPlansService, HttpPlansService>()

.AddScoped<**IToDoItemsService, HttpToDoItemsService**>();

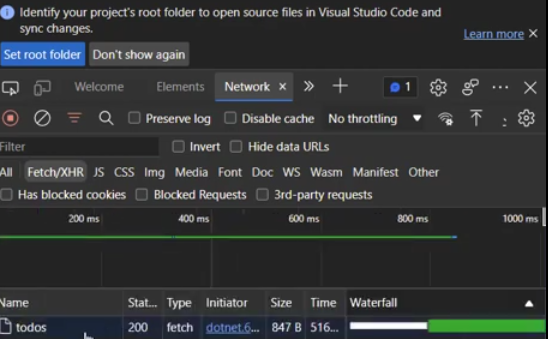
}

Esta función se inyecta porque ya estaba llamada desde program.cs

# 32. Design And Implement Add To-Do Item Form And Use It In Plan Dialog



Implementaremos la lógica para dar de alta nuevos elementos, con el botón más enviaremos la solicitud al servidor y obtendremos el resultado, se podrá depurar en Google Chrome:



1. Diseño del **formulario de altas** que consistirá en 1 descripción y el botón de + que añadirá el detalle, desde PlannerApp – Components - **CreateToDoItemForm.razor** + la clase:

Se puede crear rápidamente un nuevo componente en Visual Studio con shift f12, copiaremos el diseño de TodoItem.razor

@namespace PlannerApp.Components

<div class="d-flex flex-row align-center my-2">

<**MudTextField** @bind-Value="\_description"

**Label**="New Item Description"

**Placeholder**="Going to the gym"

**Variant**="Variant.Outlined"

**Margin**="Margin.Dense" **Error**="@(!string.IsNullOrWhiteSpace(\_errorMessage))"

**ErrorText** ="@\_errorMessage"

**Class**="flex-grow-1 flex-shrink-0 mr-2" ></**MudTextField**>

@if (\_isBusy) {

//mostrar al usuario barra progreso bola indicando que esta haciendo algo, ocupado el servidor dando de alta.

//mudblazor – components – progress – circular - primary

<**MudProgressCircular** **Color**="Color.Primary"

**Indeterminate**="true" **Class**="flex-grow-0 flex-shrink-0 mr-2"/>

}

else {

<**MudIconButton** **Icon**="@Icons.Material.Filled.Add"

**Color**="Color.Success" **Class**="flex-grow-0 flex-shrink-0 mr-2"

**OnClick**="**AddToItemAsync**"></**MudIconButton**>

}

</div>

1. **CreateToDoItemForm.razor.cs:**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Microsoft.AspNetCore.Components;

using System.Net.Http;

using System.Net.Http.Json;

using Microsoft.AspNetCore.Components.Forms;

using Microsoft.AspNetCore.Components.Routing;

using Microsoft.AspNetCore.Components.Web;

using Microsoft.AspNetCore.Components.Web.Virtualization;

using Microsoft.AspNetCore.Components.WebAssembly.Http;

using Microsoft.AspNetCore.Components.Authorization;

using Microsoft.AspNetCore.Authorization;

using Microsoft.JSInterop;

using PlannerApp;

using PlannerApp.Shared;

using PlannerApp.Components;

using MudBlazor;

using Blazored.FluentValidation;

using PlaneerApp.Client.Services.Interfaces;

using PlaneerApp.Client.Services.Exceptions;

using PlannerApp.Shared.Models;

namespace PlannerApp.Components {

public partial class CreateToDoItemForm {

[Inject]

public IToDoItemsService ToDoItemsService { get; set; }

[Parameter]

public string PlanId { get; set; }

[Parameter] //seguirá la ejecución en el padre

public **EventCallback<ToDoItemDetail> OnToDoItemAdded** { get; set; }

private bool \_isBusy = false;

private string \_description { get; set; }

private string \_errorMessage = string.Empty;

private async Task **AddToItemAsync**() {

\_errorMessage = string.Empty;

try {

if (string.IsNullOrWhiteSpace(\_description)) {

\_errorMessage = "Description is required";

return;

}

\_isBusy = true;

// Call the **API** to add the item

var result = await **ToDoItemsService.CreateAsync**

(\_description, PlanId);

\_description = String.Empty;

// **Notify the parent** about the newly added item

await OnToDoItemAdded.InvokeAsync(result.Value);

}

catch (ApiException ex) {

\_errorMessage = ex.Message;

}

catch (Exception ex) {

// TODO: Handle errors globally, falta implementarlo

}

\_isBusy = false;

}

}

}

1. En PlanDetailsDialog.razor (padre, alojar al componente):

<**MudItem** **sm**="12" **md**="7">

<!-- Items go here -->

@if (\_isBusy) {

<**MudSkeleton** **Animation**="Animation.Wave"

**SkeletonType**="SkeletonType.Text" **Style**="min-width:250px"/>

<**MudSkeleton** **Animation**="Animation.Wave"

**SkeletonType**="SkeletonType.Text" **Style**="min-width:250px"/>

<**MudSkeleton** **Animation**="Animation.Wave" **SkeletonType**="SkeletonType.Text"

**Style**="min-width:250px"/>

<**MudSkeleton** **Animation**="Animation.Wave" **SkeletonType**="SkeletonType.Text"

**Style**="min-width:250px"/>

<**MudSkeleton** **Animation**="Animation.Wave" **SkeletonType**="SkeletonType.Text"

**Style**="min-width:250px"/>

}

else {

<!-- Plan's items go here -->

//componente razor para dar de altas detalles del plan

//OnToDoItemAddedCallback devolución del elemento agregado para que se acabe de procesar en el padre que es donde estamos

<**CreateToDoItemForm** **PlanId**="@PlanId"

**OnToDoItemAdded**="**OnToDoItemAddedCallback**" />

@foreach(var item in \_items) {

//formulario cru detalles del plan

<**ToDoItem** **Item**="item" **OnItemDeleted**="OnToDoItemDeletedCallback" />

}

}

</**MudItem**>

1. En PlanDetailsDialog.razor.cs:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Microsoft.AspNetCore.Components;

using System.Net.Http;

using System.Net.Http.Json;

using Microsoft.AspNetCore.Components.Forms;

using Microsoft.AspNetCore.Components.Routing;

using Microsoft.AspNetCore.Components.Web;

using Microsoft.AspNetCore.Components.Web.Virtualization;

using Microsoft.AspNetCore.Components.WebAssembly.Http;

using Microsoft.AspNetCore.Components.Authorization;

using Microsoft.AspNetCore.Authorization;

using Microsoft.JSInterop;

using PlannerApp;

using PlannerApp.Shared;

using PlannerApp.Components;

using MudBlazor;

using Blazored.FluentValidation;

using PlaneerApp.Client.Services.Interfaces;

using PlannerApp.Shared.Models;

using PlaneerApp.Client.Services.Exceptions;

namespace PlannerApp.Components {

public partial class PlanDetailsDialog {

[CascadingParameter]

MudDialogInstance MudDialog { get; set; }

[Inject]

public IPlansService PlansService { get; set; }

[Parameter]

public string PlanId { get; set; }

private PlanDetail \_plan;

private bool \_isBusy;

private string \_errorMessage = string.Empty;

private List<ToDoItemDetail> \_items = new();

private void Close() {

MudDialog.Cancel();

}

protected override void OnParametersSet() {

if (PlanId == null)

throw new ArgumentNullException(nameof(PlanId));

base.OnParametersSet();

}

protected async override Task OnInitializedAsync() {

await FetchPlanAsync();

}

private async Task FetchPlanAsync() {

\_isBusy = true;

try {

var result = await PlansService.GetByIdAsync(PlanId);

\_plan = result.Value;

\_items = \_plan.ToDoItems;

StateHasChanged();

}

catch (ApiException ex) {

// TODO

}

catch (Exception ex) {

// TODO: Log this error

}

\_isBusy = false;

}

private void **OnToDoItemAddedCallback**(ToDoItemDetail todoItem) {

\_items.Add(todoItem);

}

private void **OnToDoItemDeletedCallback**(ToDoItemDetail todoItem) {

\_items.Remove(todoItem);

}

}

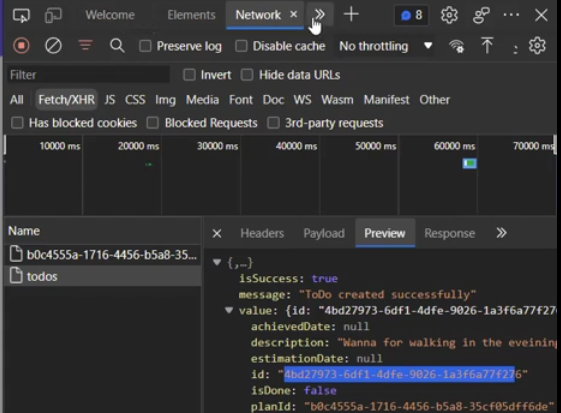
}

1. Depurar en Edge (herramienta del desarrollador) los datos enviados desde el formulario de altas al servidor-API:

Herramienta del desarrollador - Netwoork – para que muestre la solicitud del cliente que se está enviado al servidor, hay que deshabilitar css, images, … por eso marcaremos Fetch/XHR

Al dar de alta 1 nuevo detalle del plan en el depurador de Edge aparecerá un elemento todos con el json correspondiente que se ha enviado + todos los detalles (Headers, Payloads (json que hemos enviado), Preview, …), al lado de Network tenemos el panel de Console por si queremos enviar algún mensaje desde nuestro formulario para ir comprobando al depurar que se van ejecuntado las acciones implementadas por ejemplo:

Console.WriteLine(todoItem.Id);



# 33. List the Items In Plan And Improve To-Do Item Card Component

# Anexo 1 - Delegado Func<in T, out TResult>

Similar al caso anterior, acepta de 1 a 16 parámetros y los tipos son genéricos. Con la diferencia de que en este caso, Func<in T, out TResult> debe devolver un valor, **y el último tipo** que se le asigna a Func es el tipo de retorno.

En otras palabras un Func<int, string> es una función que recibe un int y devuelve un string:

Func<int, string> resultado = v => $"el resultado es{v}";

Console.WriteLine(resultado(5));

Como hemos indicado acepta múltiples parámetros por ejemplo si realizamos una multiplicación.

Func<int, int, int> multiplicacion = (v1, v2) => v1 \* v2;

int valor = multiplicacion(3, 2);

Console.WriteLine($"El resultado es {valor}");

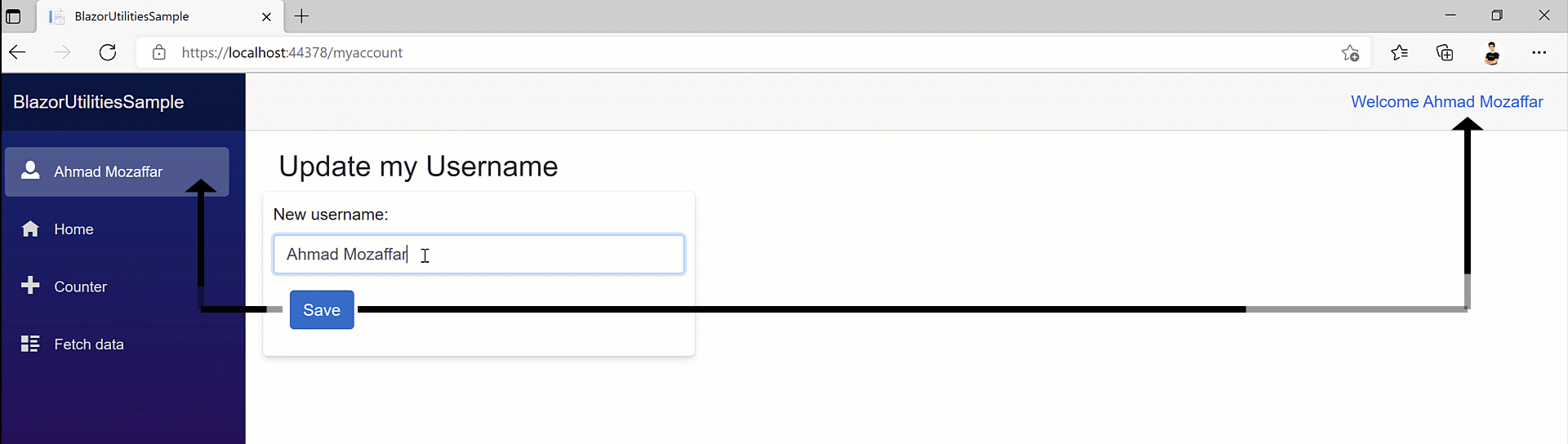
# Anexo 2 - Messaging Center Blazor Utitlity

[](https://github.com/aksoftware98/blazor-utilities/blob/main/Assets/Blazor%20Utitlies.png?raw=true)

**AKSoftware.Blazor.Utilities**

**Envíe datos a través de sus componentes con fluidez con Messaging Center**

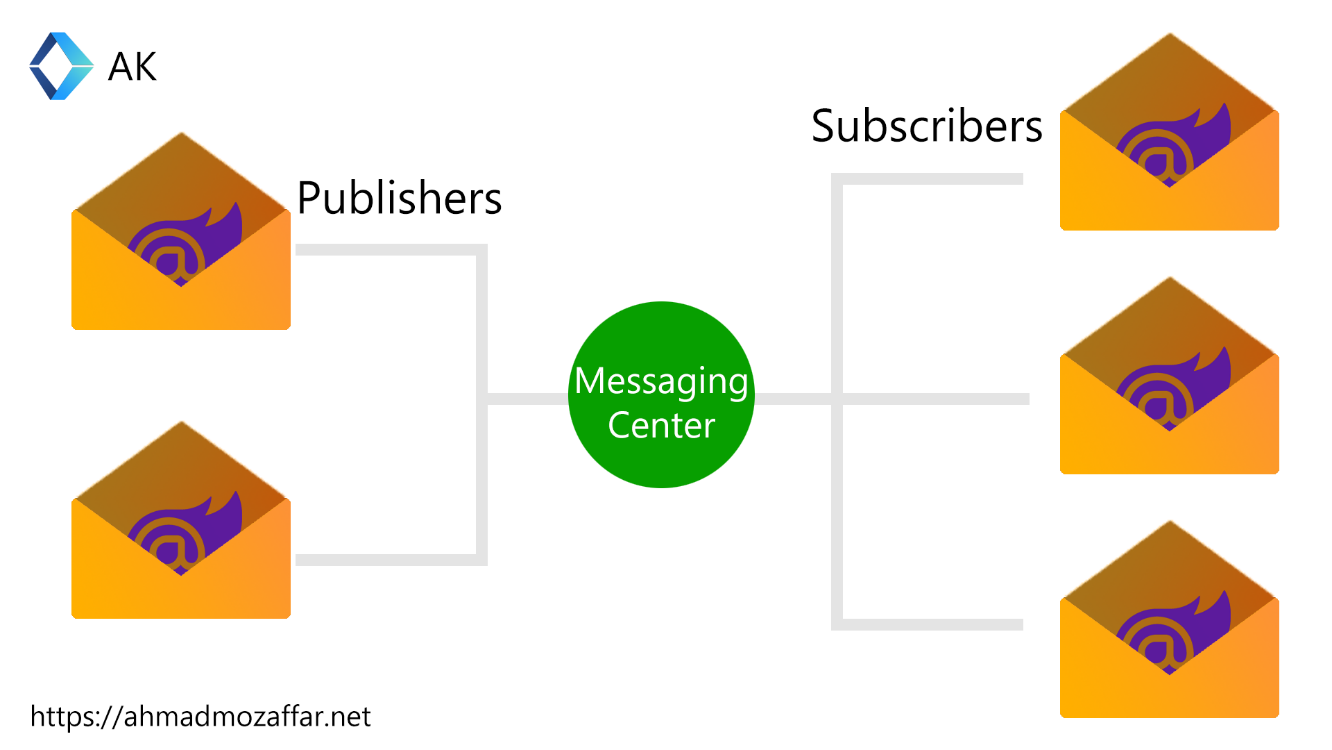
Los parámetros, los parámetros en cascada y las devoluciones de llamada de eventos son formas de enviar datos a través de componentes, pero MessagingCenter es mucho más fácil y directo

[](https://github.com/aksoftware98/blazor-utilities/blob/main/Assets/MessagingCenter%20Sample.gif?raw=true)

En este ejemplo cada vez que se escribe algo en el componente secundario y hacemos un clic en Save estamos notificando a 2 componentes diferentes que se produjo algún cambio y que se está respondiendo por esta acción, pueden ser varios.

Si es un desarrollador de Xamarin.Forms, definitivamente pasó a través de la clase MessagingCenter que se estaba usando para enviar mensajes a través de objetos, esta clase ayuda mucho al enviar datos a través de los componentes del sistema de manera eficiente. Y ahora, Blazor Messaging Center es una gran herramienta para resolver el problema de enviar datos a través de componentes y actualizarlos. Con MessagingCenter, podrá enviar datos de un componente a otro o a un conjunto de otros componentes con solo una línea de código a pesar de la relación entre esos componentes (Padre - Hijo .etc ...). La siguiente ilustración muestra los beneficios de utilizando MessagingCenter para enviar datos entre los componentes.

Explicación rápida del centro de mensajes:

[](https://github.com/aksoftware98/blazor-utilities/blob/main/Assets/MessagingCenter.png?raw=true)

**Contenido del paquete:**

Por ahora, el paquete **AKSoftware.Blazor.Utilities** solo contiene el servicio ***MessagingCenter*** que es el mismo que se implementa en Xamarin.Forms para ser utilizado en proyectos Blazor como un servicio para enviar los datos a través de los componentes usando el patrón Publicar - Suscripción además de el modelo existente de parámetros y devoluciones de llamada de eventos.

**Nota:** La clase del centro de mensajería con el paquete actual es la misma clase y está clonada del repositorio de Xamarin.Forms desde el siguiente enlace [Xamarin.Forms](https://github.com/xamarin/Xamarin.Forms/blob/5.0.0/Xamarin.Forms.Core/MessagingCenter.cs)

**Mira el proyecto de muestra:**

En el repositorio actual puede encontrar un proyecto de muestra para usar el servicio MessagingCenter en Blazor que muestra cómo enviar un valor de cadena de un componente a otros dos componentes sin una relación directa entre ellos y actualizar sus estados. El ejemplo muestra el uso de la actualización del nombre de usuario en un componente de formulario y luego, usando MessagingCenter, podemos enviar esa actualización al componente NavMenu y MainLayout y actualizar el valor allí. Consulte la siguiente carpeta:

**Comience con Blazor MessagingCenter**

Asegúrese de instalar el paquete NuGet de la biblioteca con el ***comando NuGet***

Install-Package AKSoftware.Blazor.Utilities

O a través de ***.NET CLI***

dotnet package install AKSoftware.Blazor.Utilities

Ahora en **\_imports.razor** y agregue el siguiente nombre

@using AKSoftware.Blazor.Utilities

**Publicar un mensaje**

Desde el componente del que desea enviar datos del ejemplo, una cadena de muestra como el ejemplo en el repositorio

El código toma como parámetro el objeto remitente, el filtro del mensaje como en este ejemplo "mensaje\_saludo" que será utilizado por otros componentes para recibir el mensaje, y el último parámetro es el valor a enviar.

public void SendMessage() {

string valueToSend = "Hi from Component 1";

MessagingCenter.Send(this, "greeting\_message", valueToSend);

}

**Recibir un mensaje del componente de destino**

En el componente de destino, todo lo que puede hacer es llamar al método de suscripción para suscribirse a un objetivo de mensajería.

Blazor WebAssembly

public void SubscribeToMessage() {

MessagingCenter.Subscribe<Component1, string>(this, "greeting\_message", (sender, value) =>

{

// Do actions against the value

// If the value is updating the component make sure to call

string greeting = $"Welcome {value}";

StateHasChanged(); // To update the state of the component

});

}

## 

**Aviso del servidor Blazor**

Debido a que Blazor Server está trabajando en el hilo, asegúrese de llamar a ese InvokeAsync dentro de la devolución de llamada **Nota:** Esto se resolverá y actualizará en la versión 1.0.1 Agradecimiento especial a [**Ferenc Czirok**](https://github.com/czirok) por mencionar la nota y el siguiente fragmento de código

public void SubscribeToMessage() {

MessagingCenter.Subscribe<Component1, string>(this, "greeting\_message", async (sender, value) => {

// Do actions against the value

// If the value is updating the component make sure to call

// Use InvokeAsync() to switch execution to the Dispatcher when triggering rendering or component state

await InvokeAsync(() => {

string greeting = $"Welcome {value}";

StateHasChanged(); // To update the state of the component

});

});

}

### 