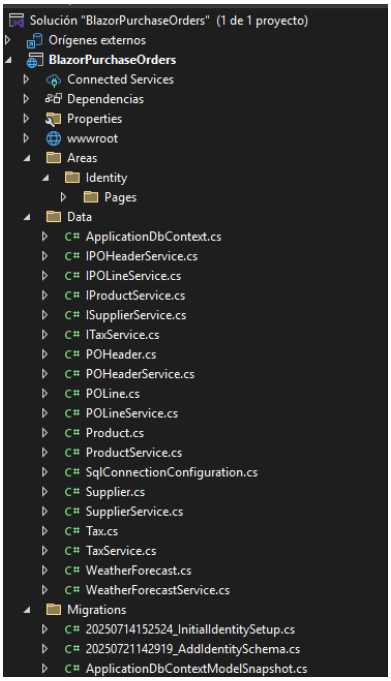
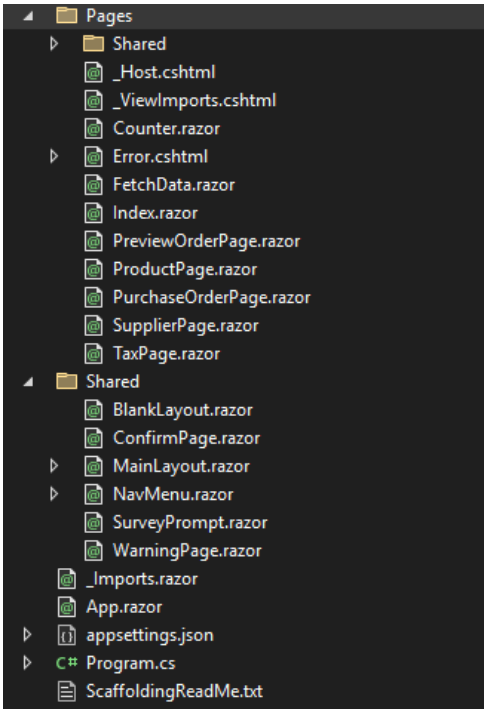
Explicación del proyecto - Purchase Orders





Blazor Server App es un modelo de aplicación web desarrollado por Microsoft como parte del framework **Blazor**, que permite construir interfaces web interactivas usando **C# en lugar de JavaScript**.

Dapper es un **micro ORM (Object-Relational Mapper)** para .NET. Su principal propósito es **mapear datos de una base de datos relacional a objetos C#.**

Acceso a Datos (Dapper/EF Core): Se usa una combinación. Identity usa Entity Framework Core (`ApplicationDbContext`), mientras que tus servicios de negocio (`POHeaderService`, etc.) usan Dapper para interactuar directamente con la base de datos a través de Stored Procedures (procedimientos almacenados). Esto

ultimo es un conjunto de instrucciones SQL predefinidas que se guarda en la base de datos y que se pueden ejecutar cuando se necesite, permitiendo **automatizar operaciones** comunes en la base de datos: Insertar, actualizar o eliminar datos.

Syncfusion Blazor: es una biblioteca de componente del framework Blazor que facilita la creación de aplicaciones web usando C#

———

EXPLICACION DE LA ESTRUCTURA ————

1. Carpeta Properties:

   Contienen el archivo que define como se ejecutara la aplicación en diferentes perfiles `launchSettings.json`

2. wwwroot:

   En este apartado se encuentra el contenido web estático, contiene CSS, JavaScript, Imágenes e iconos.

3. Areas:

   Capeta que se crea automáticamente al momento de crear el proyecto en Información Adicional →Authentication de campo → Cuentas individuales. Se utiliza para organizar la autenticación y gestión de usuarios de ASP.NET Core Identity.

   - pages: para acciones de usuario como registro, login, logout.

4. Carpeta Data: es una de las carpetas mas importantes para la lógica del proyecto y el acceso a datos.

   - Migrations: contiene los archivos generados por Entity Framework Core cuando se realizan migraciones de la base de datos, estos datos describen los cambios en el esquema de base de datos.
   - ApplicationDbContext.cs: gestiona la conexión a la base de datos, las tablas de identidad como usuarios, roles etc. Y cualquier otra tabla que se decida mapear.

- Interfaces de Servicio `(IPOHeaderService.cs, IPOLineService.cs, IProductService.cs, ISupplierService.cs, ITaxService.cs):` define las operaciones relacionadas con los encabezados de ordenes de compra, productos, proveedores e impuestos. Es clave para la inversión del control y la inyección de dependencias.
- Modelos de Datos: (`POHeader.cs` , `POLine.cs` , `Product.cs` , `Supplier.cs` , `Tax.cs` , `WeatherForecast.cs`): representan la estructura de los datos de la base de datos; como las tablas, columnas y filas.
- **Implementaciones de Servicio: (`POHeaderService.cs` , `POLineService.cs` , `ProductService.cs` , `SupplierService.cs` , `TaxService.cs` , `WeatherForecastService.cs`):** estas clases implementan las interfaces de servicio correspondientes y contienen la lógica real para interactuar con la BD, usando Dapper para realizar operaciones CRUD(Crear, leer, actualizar, eliminar).
- SqlConnectionConfiguration.cs: encapsula la conexión a la BD, permitiendo una fácil inyección de la configuración de conexión.

5. Carpeta Pages: Contiene los componentes Blazor que actúan como paginas navegables de la aplicación.
   C
   ada archivo `.razor` con una directiva `@page` es una ruta URL accesible.
   - `_Host.cshtml` : página que configura la aplicación Blazor en el lado del servidor.
   - `PreviewOrderPage.razor` : página para ver los detalles de la orden de compra.
   - `ProductPage.razor` : Página que gestiona los productos.
   - `PurchaseOrderPage.razor` : Página principal para la gestión de órdenes de compra.
   - `SupplierPage.razor` : Página que gestiona los proveedores.
   - `TaxPage.razor` : Página que gestiona tasas de impuestos.

6. Shared: contiene **componentes Blazor reutilizables** que no son páginas navegables por sí mismas. Estos componentes pueden ser insertados en varias páginas o en otros componentes.
   - `NavMenu.razor` : El menú de navegación principal de la aplicación.

- `WarningPage.razor`: Un componente reutilizable para mostrar diálogos de advertencia al usuario.
- `ConfirmPage.razor`: Un componente reutilizable para mostrar diálogos de confirmación al usuario.

7. `_Imports.razor`: contiene directivas `@using` que se aplican **globalmente** a todos los componentes `.razor` Esto evita tener que añadir las mismas directivas `@using` repetidamente en cada archivo `.razor`.
8. `App.razor`: es el componente raíz de la aplicación.
9. `appsettings.json`: Almacena configuraciones como cadenas de conexión a la base de datos.
10. Program.cs: entorno en el que se ejecuta una aplicación, configuración de servicios.

Los `Servicios` son componentes claves que encapsulan lógica y funcionalidades especificas del proyecto, haciéndolas disponibles para otras partes de la aplicación a través de un patrón llamado inyección de dependencias

Una interfaz define un conjunto de métodos que una clase debe implementar, los archivos como `IPOHeaderService.cs`, `IPOLineService.cs`, `ITaxService.cs` son interfaces y en este caso `ITaxService` tiene métodos como: `TaxList()`, `TaxInsert()`, `TaxUpdate()` luego TaxList( ) se usa dentro de la clase TaxService, como un método publico asincrónico. su propósito es obtener una lista de todos los registros de impuestos de la base de datos y devolverlos como una colección de objetos Tax:

```
public async Task<IEnumerable<Tax>> TaxList()
{
IEnumerable<Tax> taxes;
using (var conn = new SqlConnection(_configuration.Value))
{
taxes = await conn.QueryAsync<Tax>("spTax_List", commandType: Comm
}
```

```
return taxes;
}
```

`UPDATE:` modifica datos de una tabla

`Procedimientos Almacenados (Store Procedures):` son un conjunto de sentencias SQL que se compilan y se almacenan directamente en una base de datos
Ejemplo: El procedimiento almacenado spTax_Update se encarga de actualizar una fila en la tabla `dbo.Tax`. Cuando el código C# llama a `await conn.ExecuteAsync("spTax_Update", parameters, commandType: CommandType.StoredProcedure);` en `TaxService.cs` está invocando este código SQL almacenado en la base de datos.

Agregar Roles a los usuarios:

- En program.cs en el método que registra los servicios para la gestión de usuarios: `AddDefaultIdentity<IdentityUser>()` después agrego `.AddRoles<IdentityRole>()` habilita la funcionalidad de gestión de roles.
- Luego en NavMenu.razor agrego la autorización para la visualización del menú dependiendo del rol que desempeñe el usuario.
- En las siguientes clases TaxPage.razor, SupplierPage.razor, ProductPage.razor agrego:

```
@attribute [Authorize(Roles = "Admin")]
```

es una directiva de Blazor que se utiliza para **restringir el acceso a un componente o página solo a usuarios que poseen el rol especificado**.

NOTA: La autenticación es saber quien es el usuario y la autorización es conceder permiso para ver datos o realizar alguna acción

---

Como funcia la vista previa de un pedido:

En PreviewOrderPage.razor esta todo el diseño de la orden del pedido y

Obtencion de los datos

La siguiente linea de código en Index.razor esta vincula con la vista previa del pedido y se encarga de abrir una nueva ventana del navegador que muestra la previsualización de la orden de compra seleccionada, utilizando el ID de la orden para construir la URL específica. Esto se logra llamando a una función de JavaScript desde el código C# de Blazor.

```
await IJS.InvokeVoidAsync("open", new object[] { "/previeworder/" + selecte
```

código en el ciclo @foreach en el HTML de PreviewOrderPage.razor:
Los
`String.Format` se utilizan para formatear los números y monedas correctamente (`{0:N0}` para números sin decimales, `{0:C2}` para moneda con dos decimales, `{0:P2}` para porcentaje).

La
"vista previa del pedido" se logra al abrir una nueva pestaña del navegador (`_blank` que carga una página Blazor dedicada (`PreviewOrderPage`. Esta página utiliza el ID de la orden pasado en la URL para recuperar los detalles completos del encabezado y las líneas del pedido de la base de datos y luego renderiza esa información en un formato visualmente agradable (simulando una factura o recibo) utilizando HTML y CSS.