

# PIZZA SHOP



Pablo Morato Polo  
Desarrollo Interfaces  
Febrero 2024

# INDICE

Objetivo .....	3
Problemática inicial.....	3
Etapa 1: Estructura y diseño .....	5
Estructura del proyecto: .....	5
Diseño: .....	6
Cambio Funcional (cambio estético e inclusión de un nuevo “Special”) .....	6
Etapa 2: Personalizar la pizza.....	8
Etapa 3: Mostrar pedidos y estado.....	10
Cambio funcional (pestaña Mis Promociones) .....	11
Etapa 4: Administrar estados.....	14
Etapa 5: Pago con validación .....	15
Etapa 6: Autenticación y autorización .....	17
Etapa 7: JavaScript .....	21
Etapa 8: Plantilla de componentes .....	22
Etapa 9: Aplicación Web Progresiva PWA .....	24

## Objetivo

Blazor es un marco de aplicación de una sola página para crear aplicaciones web del lado cliente mediante .NET y WebAssembly. En este taller, crearemos una aplicación completa de Blazor y aprenderemos sobre las diversas características del marco de Blazor a lo largo del camino.

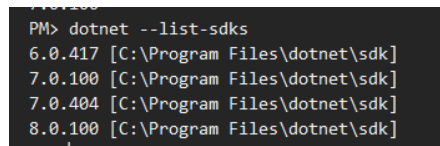
## Problemática inicial

El primer problema al que nos enfrentamos es la correcta configuración del proyecto y las dependencias, y la compatibilidad con la versión de Visual Studio que estamos utilizando.

Intenté actualizar las librerías y las dependencias del proyecto a .NET8.0, puesto que era con la última versión que he trabajado, pero tenía problemas de compatibilidad con .NET8.0, así que decidí replicar con las versiones indicadas en el taller, que aseguran compatibilidad ya que están testadas y aseguran su funcionamiento y compatibilidad.

En primer lugar, hay que verificar la versión de .NET y el SDK con el cual estamos trabajando en nuestro Visual Studio, para ello podemos abrir la consola Herramientas > Administrador De Paquetes > Consola del Administrador.

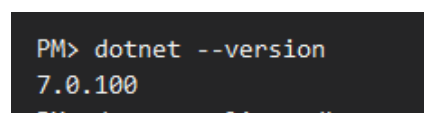
Una vez en la consola del administrador de paquetes, con el comando `"dotnet --list-sdks"` podemos ver el listado y versiones de los SDK que tenemos disponibles:



```
PM> dotnet --list-sdks
6.0.417 [C:\Program Files\dotnet\sdk]
7.0.100 [C:\Program Files\dotnet\sdk]
7.0.404 [C:\Program Files\dotnet\sdk]
8.0.100 [C:\Program Files\dotnet\sdk]
```

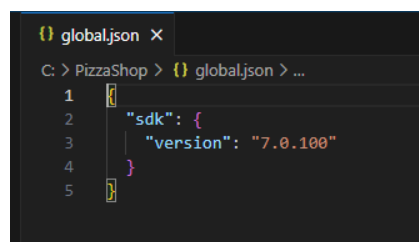
Como se puede ver en la anterior captura, tengo instalados diferentes versiones, que corresponden a las versiones .NET6.0, .NET7.0 (dos versiones distintas) y .NET8.0. En la web de Microsoft (<https://dotnet.microsoft.com/es-es/download/dotnet>) podemos obtener las distintas versiones con sus instaladores.

Con el comando `"dotnet --version"` podemos comprobar la versión con la que estamos trabajando:



```
PM> dotnet --version
7.0.100
```

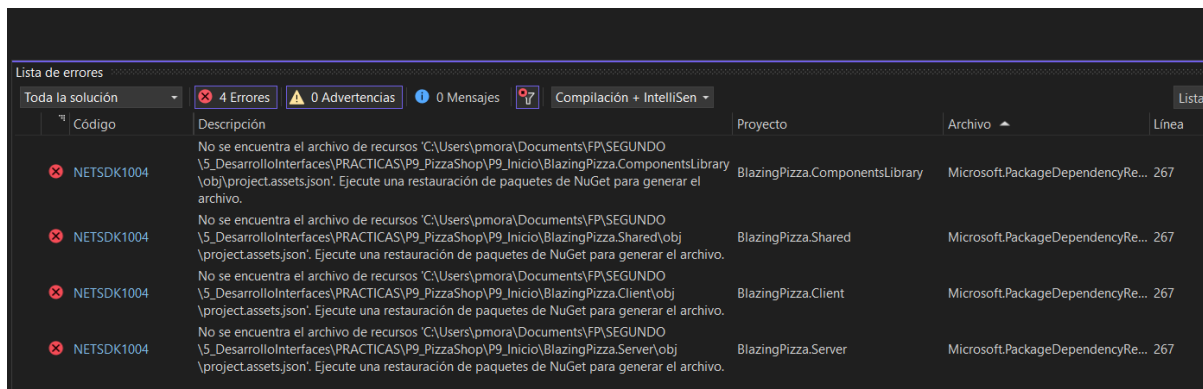
En este caso ya estamos trabajando con la versión .NET7.0, pero en el caso de estar en una distinta, podemos cambiar introduciendo el comando `"dotnet new globaljson --sdk-version 7.0.100"` en la consola, con lo que se crea un archivo en la raíz de nuestro proyecto que indica la versión de nuestro de proyecto.



```
{} global.json X
C: > PizzaShop > {} global.json > ...
1  {
2    "sdk": {
3      "version": "7.0.100"
4    }
5  }
```

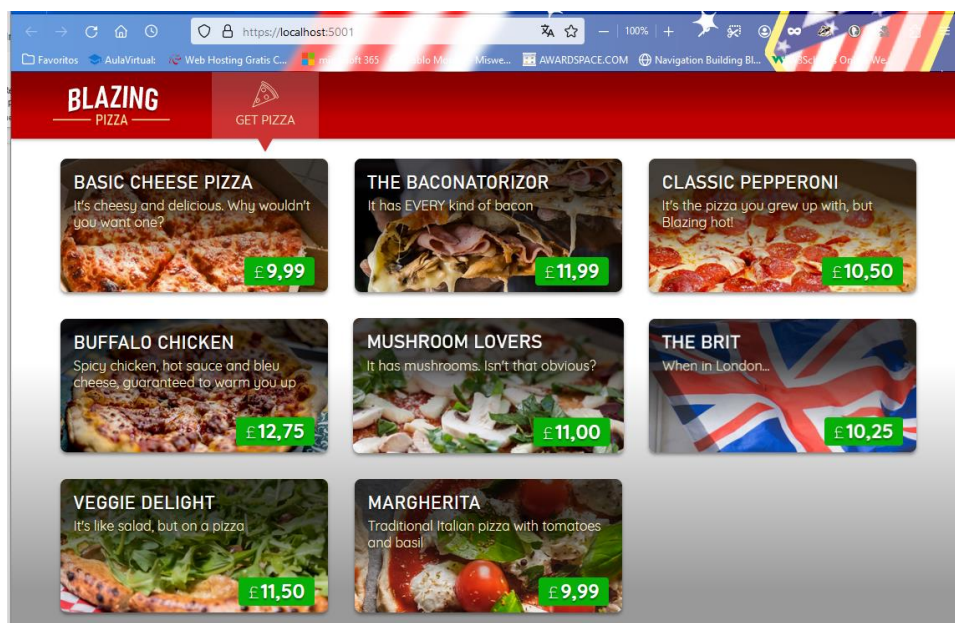
Como paso inicial, en el taller se indica que hay que descargar o clonar el repositorio, y luego ejecutar la solución del proyecto. Además, se indica que debemos de agregar a la raíz de nuestro proyecto los archivos correspondientes y necesarios para la configuración del entorno

Al realizar este primer paso, nos encontramos los primeros problemas de ejecución:



Estos problemas están relacionados con la ausencia del archivo, `project.assets.json`, archivo que define las dependencias de nuestro proyecto. Este es un archivo que se genera automáticamente, por lo que podemos recurrir a la consola del administrador de paquetes e introducir el comando `"dotnet restore"`, se produce una restauración de los archivos y se generan automáticamente los archivos necesarios, entre otros el `project.assets.json`.

Ahora, después de estos pasos, nuestro entorno está preparado y no muestra errores al compilar el proyecto, al ejecutar se abre la aplicación y el navegador con la página correspondiente en nuestro localhost:



## Etapa 1: Estructura y diseño

### Estructura del proyecto:

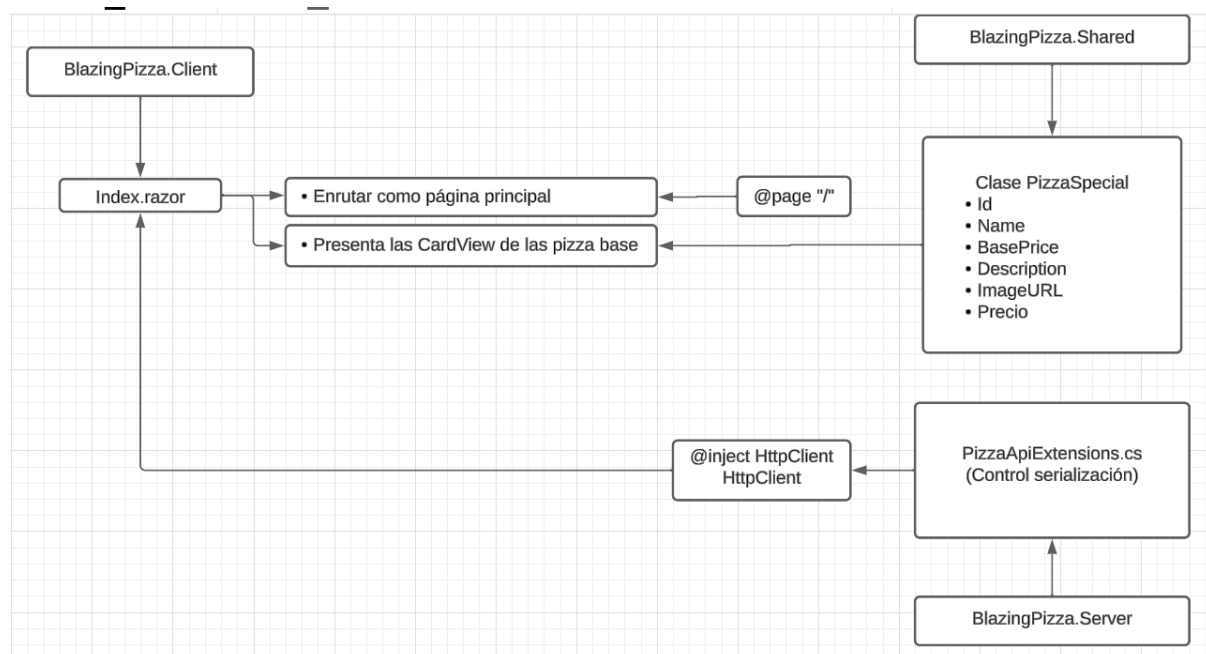
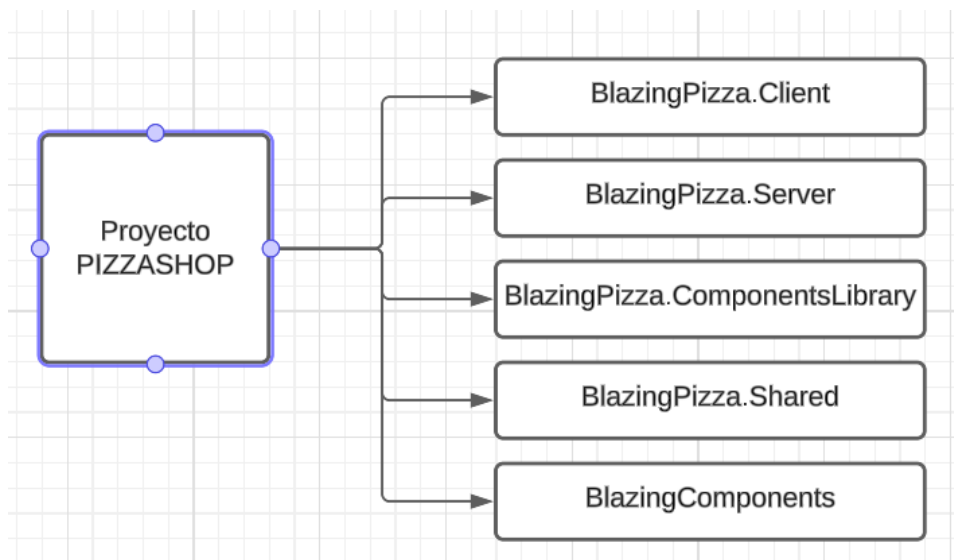
Comprobamos que inicialmente nuestro proyecto se compone de otros cuatro proyectos (y posteriormente se añadirá otro):

BlazingPizza.Client: Contiene los componentes de la interfaz de usuario

BlazingPizza.Server: Proyecto de ASP:NET que aloja la app Blazor y servicios backend

BlazingPizza.Shared: Contiene los modelos compartidos

BlazingPizza.ComponentsLibrary: Biblioteca de componentes y código auxiliar



En el proyecto BlazingPizza.Client buscamos en Pages el Index.razor.

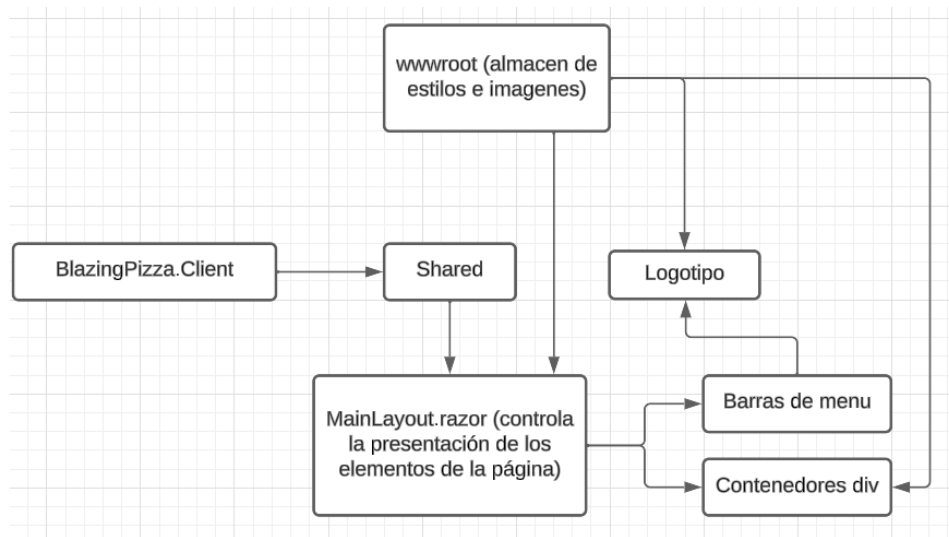
En este archivo definimos `@page "/"` para definir y enrutar esta página como la página principal.

Agregamos un bloque con @code, donde definimos la lista de pizzas especiales. En el proyecto de BlazingPizza.Shared tenemos definida la clase PizzaSpecial (Id, Name, BasePrice, Description, ImageURL, Precio).

La lista la vamos a extraer de una API, por lo que debemos de inyectar en nuestra cabecera el componente con @inject HttpClient HttpClient, con el cual nos podremos hacer las peticiones http.

En el bloque @code generado antes, introducimos un método que controle la deserialización de los objetos. La API se define en BlazingPizza.Server en la clase PizzaApiExtensions.cs.

### Diseño:

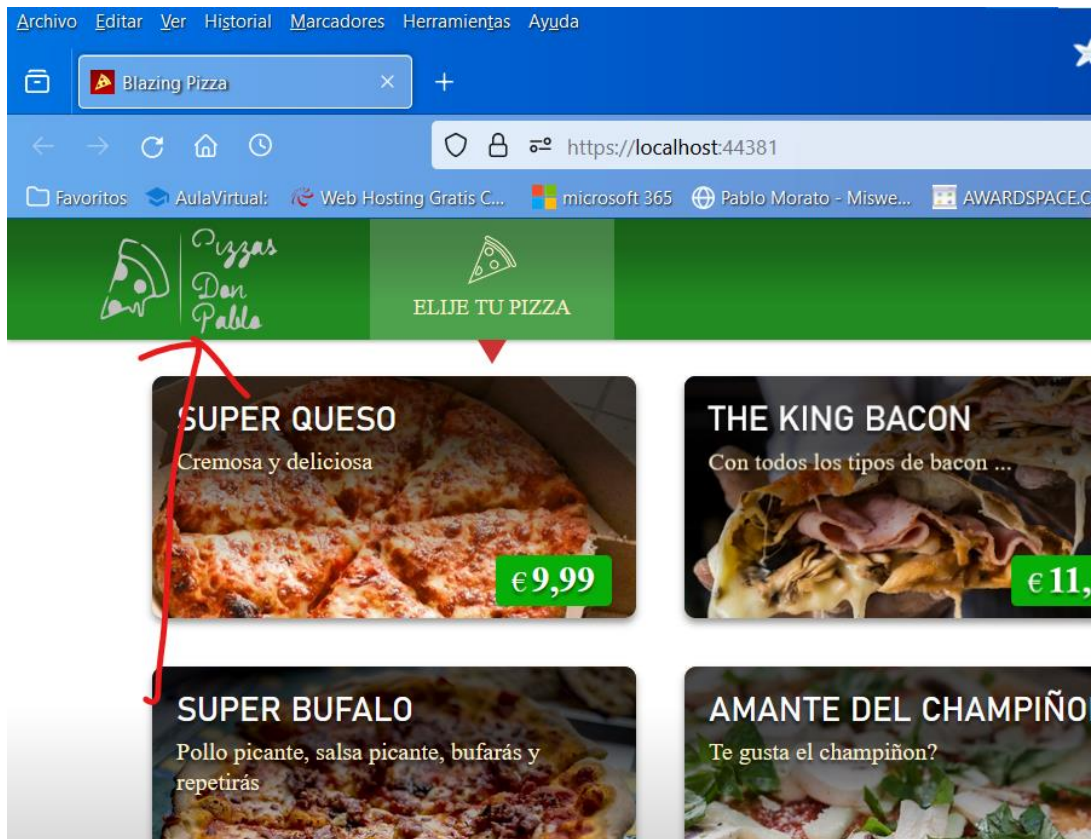


En el proyecto BlazingPizza.Client buscamos en Shared el MainLayout.razor, que controla los elementos de nuestra página (barras de menú, y contenedores div).

Se define la barra superior de la página con el logotipo y un link, dentro de wwwroot tenemos la carpeta donde se almacenan las imágenes.

### Cambio Funcional (cambio estético e inclusión de un nuevo “Special”)

En esta primera etapa del taller de la aplicación se tratan aspectos muy básicos, por lo que los cambios que vamos a realizar serán poco importantes y muy básicos. El cambio consistirá en un cambio de logotipo de la página, para dicho cambio debemos tener nuestro logo en un formato svg, y la foto la guardamos en la carpeta wwwroot del proyecto BlazingPizza.Client. Como hemos visto, el componente Mainlayout.razor, es el que controla la presentación de los elementos de la página, por lo que debemos de buscar en la parte de la barra superior donde se encuentra el logo y cambiar la ruta hasta la foto de nuestro logo. En el pantallazo se muestra el nuevo logo:



Otro cambio que realizamos es introducir una nueva pizza, para realizar este cambio accedemos al fichero pizza.db, en donde se gestionan los datos de la base de datos que utiliza la aplicación. Se elige la opción de acceder al archivo con la aplicación DB Browser for SQLite, accedemos a la pestaña correspondiente donde se guardan las pizzas (nombre, descripción, precio y ruta de la fotografía).

DB Browser for SQLite - C:\Users\pmora\Documents\FP\SEGUNDO\5\_DesarrolloInterfaces\PRACTICAS\P9\_PizzaShop\P9\_01\_Inicio\Blazing

Archivo Editar Ver Herramientas Ayuda

Nueva base de datos Abrir base de datos Guardar cambios Deshacer cambios Abrir proyecto Guardar proyecto

Estructura Hoja de datos Editar pragmas Ejecutar SQL

Tabla: Specials

	Id	Name	BasePrice	Description	ImageUrl
	Filtro	Filtro	Filtro	Filtro	Filtro
1	1	Super Queso	9.99	Cremosa y deliciosa	img/pizzas/cheese.jpg
2	2	The king Bacon	11.99	Con todos los tipos de bacon ...	img/pizzas/bacon.jpg
3	3	Clasica de Peperoni	10.5	La original de todos los tiempos	img/pizzas/pepperoni.jpg
4	4	Super Bufalo	12.75	Pollo picante, salsa picante, bufarás ...	img/pizzas/meaty.jpg
5	5	Amante del Champiñon	11.0	Te gusta el champiñon?	img/pizzas/mushroom.jpg
6	6	The Brit	10.25	When in London...	img/pizzas/brit.jpg
7	7	Delicia Vegetariana	11.5	Olvidate de la carne	img/pizzas/salad.jpg
8	8	Margarita	9.99	La pizza tradicional italiana	img/pizzas/margherita.jpg
9	9	Marinera	13	La del mar	img/pizzas/marinera.jpg

Utilizando la misma aplicación, accedemos a los ingredientes y traducimos de idioma todos los ingredientes:

DB Browser for SQLite - C:\Users\pmora\Documents\FP\SEGUNDO\5\_DesarrolloInterfaces\PRACTICAS\P9\_PizzaShop\P9\_01\_Inicio\Blazing

Archivo Editar Ver Herramientas Ayuda

Nueva base de datos Abrir base de datos Guardar cambios Deshacer cambios Abrir proyecto Guardar proyecto

Estructura Hoja de datos Editar pragmas Ejecutar SQL

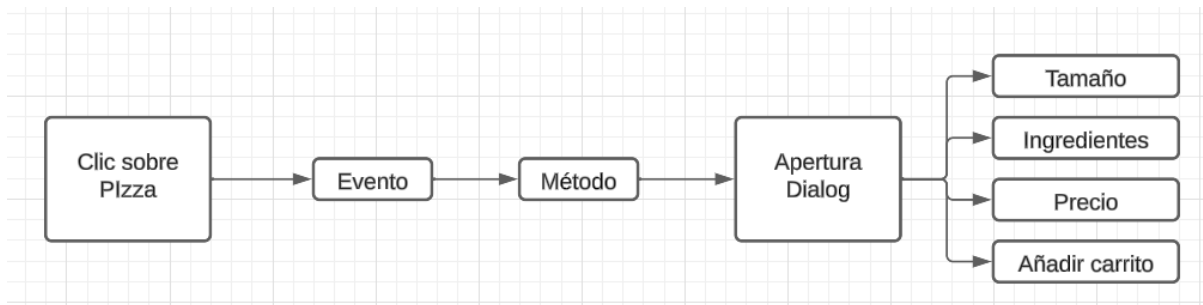
Tabla: Toppings

	Id	Name	Price
	Filtro	Filtro	Filtro
1	1	Extra de queso	2.5
2	2	Bacon americano	2.99
3	3	Bacon ingles	2.99
4	4	Bacon canadiense	2.99
5	5	Te y bollos	5.0
6	6	Carne de avestruz	4.5
7	7	Dimientes	1.0

## Etapa 2: Personalizar la pizza

En esta etapa vamos a gestionar como personalizar una pizza. Cuando hagamos clic sobre una de las pizzas especiales que se muestran en la página, se abrirá un dialog.





Para ello, en primer lugar, debemos crear el evento producido al apretar el botón, que llame a un método que nos muestre el dialogo de configuración, por lo que desarrollamos en la etiqueta `@code` el código del método, al cual pasamos como parámetro la pizza especial sobre la que hemos clicado.

Ahora tenemos que implementar el dialog de personalización, será un nuevo dialog, en donde seleccionaremos el tamaño y los ingredientes, mostrando el precio y permitiendo añadirlo al carrito de pedidos.

Sobre la carpeta Shared clicamos y añadimos un nuevo componente `ConfigurePizzaDialog.razor`, el cual estará formado por el nombre y descripción de la pizza, y dos botones (cancelar y pedir). Más tarde añadimos un slider que controla el tamaño de la pizza, y un cuadro de texto que nos permitirá añadir ingredientes.

En la página principal, `index.razor`, incluimos un condicional para mostrar el dialog en caso de clicar sobre alguna de las pizzas.

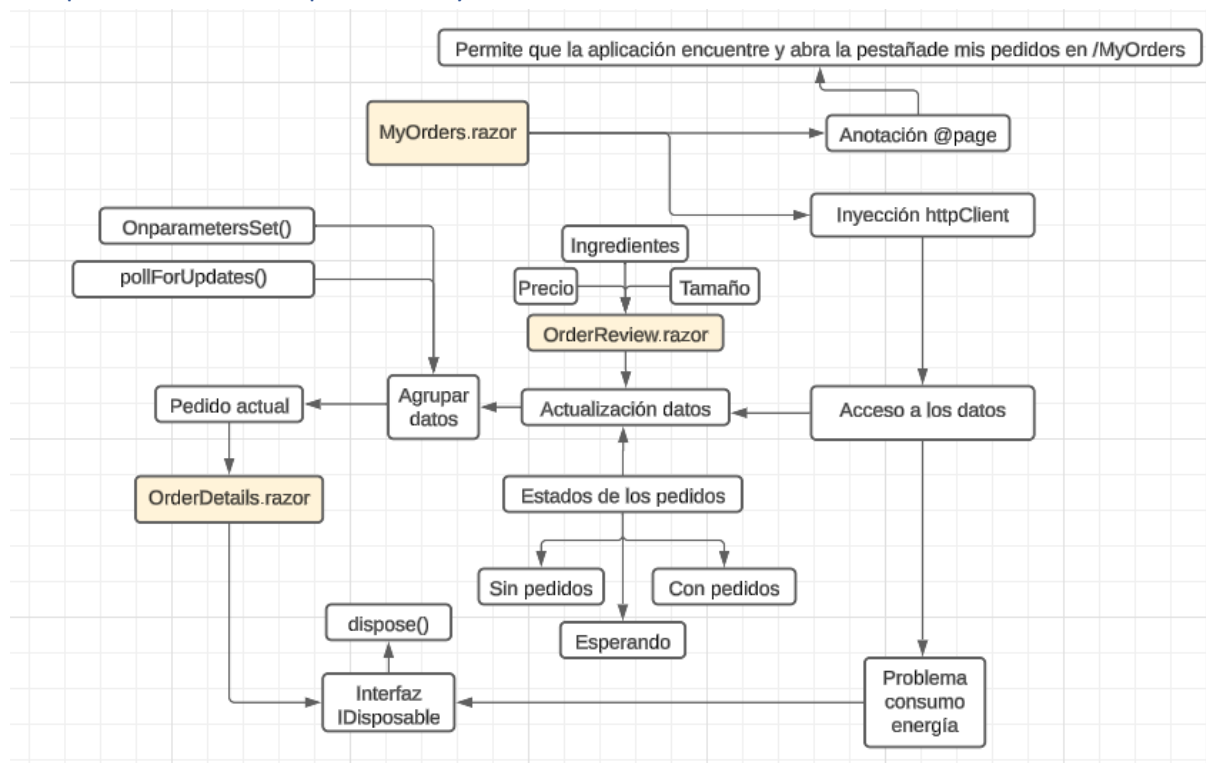
Para mostrar el pedido actual:

Creamos un componente dentro de la carpeta Shared llamado `ConfiguredPizzaltem`.

En este nuevo componente incluimos las etiquetas necesarias para el botón para borrar, el nombre de la pizza, los ingredientes y el precio, y en el bloque `@code` dos parámetros, la pizza configurada y el evento cuando la pizza es eliminada.

En el componente `index.razor`, creamos una barra lateral que nos permita alojar los pedidos, con una opción para eliminar, y otra opción que muestra el total y nos permite realizar el pedido.

### Etapa 3: Mostrar pedidos y estado



En la barra de nuestra aplicación, en el archivo contenido en Shared llamado MainLayout.razor, añadimos un link/pestaña llamada Mis Pedidos. Tenemos que hacer:

- 1 Una parrilla para mostrar pedidos, añadir una presentación de los detalles de los pedidos.
- 2 Agrupar los detalles de los pedidos.
- 3 Hacer y pintar en nuestra página los detalles.
- 4 Ver la información en tiempo real.
- 5 Optimizar la memoria.
- 6 Navegar automáticamente a los detalles.

Ahora, creamos en Pages un nuevo componente llamado MyOrders.razor:

- Añadimos anotación para enrutar la página (@page "/myorders")
- Para mostrar los pedidos debemos incorporar la inyección (@inject HttpClient HttpClient)
- Añadimos en el bloque de código (@code) la petición asíncrona de los datos.
- En el main establecemos las 3 posibilidades de salida:
  - Esperando carga
  - Sin pedidos
  - Con pedidos

1.- Ahora establecemos las etiquetas necesarias en MyOrders.razor para mostrar los pedidos.

Al clicar en el detalle de pedido para conocer el estado, debe redirigirnos a otra página, por lo que debemos de crear OrderDetails.razor en Pages.

En OrderDetails.razor añadimos el enrutamiento @page "/myorders/{orderId:int}", definiendo el número de pedido como parámetro.

2.- Para poder agrupar los detalles de los pedidos

Añadir directiva `@using` e `@inject` (`@using` se utiliza para importar namespaces y hacerlos accesibles en el archivo Razor, e `@inject` se usa para inyectar dependencias y obtener acceso a objetos específicos en el componente Razor, simplifican código al permitir acceder a elementos y servicios sin repetir código o lidiar con la creación manual de instancias.)

Implementar el agrupamiento dentro del bloque `@code`, con los dos métodos `OnParametersSet()` y `PollForUpdates()`.

3.- En el div del main se controla y muestra si el valor es invalido, si se están cargando los datos y si hay datos que mostrar.

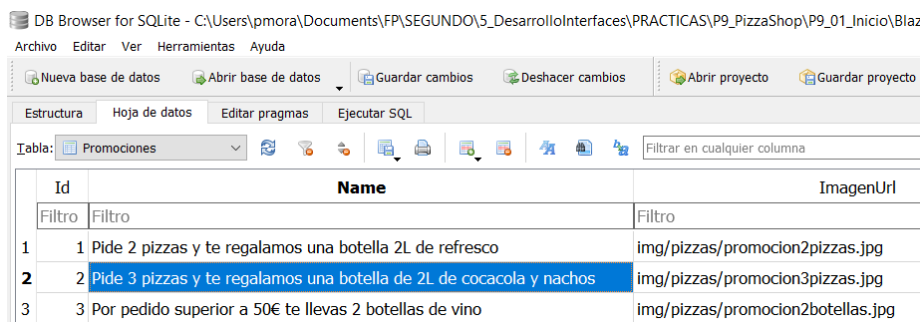
A continuación, debemos crear un nuevo fichero `OrderReview.razor` en la carpeta `Shared` de nuestro proyecto, en donde controlaremos el contenido del pedido actual. Haremos un bucle `foreach` en donde recorreremos las pizzas de nuestro pedido, extrayendo la información de tamaño, nombre y precio, y luego haremos otro bucle `foreach` para recorrer y mostrar los ingredientes.

Cuando hacemos una petición de los detalles del pedido, se inicia un proceso de consulta, pero no se finaliza, si hacemos muchas peticiones de distintos pedidos se consumirá memoria puesto que no se cierran los procesos. Para optimizar este consumo de memoria, haciendo que se detenga el proceso una vez salgamos de la pantalla, utilizamos la interfaz `IDisposable`. Para ello añadimos `@implements IDisposable` en `OrderDetails.razor.`, e incorporamos el método `Dispose()` en el bloque de código.

### Cambio funcional (pestaña Mis Promociones)

En este cambio vamos a introducir una nueva pestaña a la que va a poder acceder el usuario cuando está logueado, además de la pestaña de mis pedidos aparecerá la pestaña de “Mis Promociones”.

En esta pestaña se cargarán las promociones existentes en la base de datos en la tabla que hemos creado de promociones:



Id	Name	ImagenUrl
1	Pide 2 pizzas y te regalamos una botella 2L de refresco	img/pizzas/promocion2pizzas.jpg
2	Pide 3 pizzas y te regalamos una botella de 2L de cocacola y nachos	img/pizzas/promocion3pizzas.jpg
3	Por pedido superior a 50€ te llevas 2 botellas de vino	img/pizzas/promocion2botellas.jpg

Para realizar esta funcionalidad, además de crear la tabla promociones en la base de datos, tendremos que realizar varios pasos:

- Crear una clase `Promociones.cs` en donde definimos los atributos de las promociones junto con los getter y setters.

```

1 namespace BlazingPizza;
2
3 /// <summary>
4 /// Representa las promociones
5 /// </summary>
6 public class Promocion
7 {
8     public int Id { get; set; }
9
10    public string Name { get; set; } = string.Empty;
11
12    public string ImagenUrl { get; set; } = string.Empty;
13
14 }
15

```

- Dentro de la clase PizzaApiExtensions.cs, tenemos que mapear y establecer el endpoint para poder serializar la información de la base de datos y poder acceder con las peticiones httpClient:

```

// Promociones
app.MapGet("/promociones", async (PizzaStoreContext db) => {
    var promociones = await db.Promociones.ToListAsync();
    return Results.Ok(promociones);
});

```

- Se define una propiedad en el contexto de la base de datos que representa una colección de entidades de tipo Promocion. Esta propiedad se puede utilizar para poder acceder a las promociones en la base de datos desde las distintas partes del código de la aplicación.

```

// Nueva llamada para coger los datos de la tabla de la base de datos
1 referencia
public DbSet<Promocion> Promociones { get; set; }

2 referencias
public DbSet<Topping> Toppings { get; set; }

```

- Por último podemos crear nuestro componente Promociones.razor en donde tendremos lo que nos pintará cuando entremos en la pestaña de Mis Promociones, para ello debemos de realizar varias anotaciones.
  - Anotación @page que enrutará la página con la dirección de acceso.
  - Anotación del atributo [Authorize] que limitará el acceso y solo permitirá acceder a los usuarios logueados.
  - Inyección de HttpClient que permitirá hacer las peticiones get.
  - Luego establecemos un contenedor div en el cual introducimos un bucle tipo foreach que nos permite recorrer la lista de las promociones que hemos obtenido de la base de datos, en donde mostraremos la descripción y las fotografías.
  - En el apartado @code, se declara una lista promociones que contendrá objetos de tipo Promocion, y con el método OnInitializedAsync, que se ejecuta cuando el componente está siendo inicializado, se realiza la solicitud HTTP utilizando HttpClient para obtener los datos de promociones desde el endpoint "promociones". Los datos obtenidos se asignan a la lista de promociones para que puedan ser recorridos y mostrados por el bucle foreach.

```

@page "/promociones"
@attribute [Authorize]
@inject HttpClient HttpClient

@using System.Net.Http.Json

<PageTitle>Don Pablo - Mis Promociones</PageTitle>

<div style="display: flex; flex-wrap: wrap; justify-content: center;">
    @if (promociones != null)
    {
        @foreach (var promocion in promociones)
        {
            <div style="width: 300px; margin: 10px; border: 5px solid #ccc; border-radius: 5px;">
                <div style="background-color: #f0f0f0; border-radius: 5px 5px 0 0; box-shadow: 0 4px 8px rgba(0, 0, 0, 0.1); height: 100px; position: relative;">
                    <h6 style="text-align: center; padding: 10px; margin: 0;">@promocion.Name</h6>
                    
                </div>
            </div>
        }
    }
</div>

@code {
    List<Promocion>? promociones;

    protected override async Task OnInitializedAsync()
    {
        promociones = await HttpClient.GetFromJsonAsync<List<Promocion>>("promociones");
    }
}

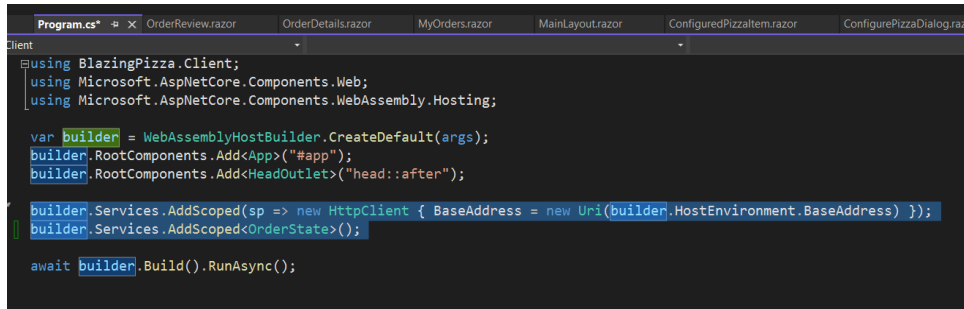
```

## Etapla 4: Administrar estados

Si realizamos el pedido de una pizza y no clicamos en pedir, y volvemos a la página inicial, al volver a la página el pedido desaparece.

Para solucionar este fallo tendremos que implantar un patrón AppState, que coordinará el estado.

En Program.cs tenemos que agregar el servicio, modificando el existente



```

Program.cs
Client
using BlazingPizza.Client;
using Microsoft.AspNetCore.Components.Web;
using Microsoft.AspNetCore.Components.WebAssembly.Hosting;

var builder = WebAssemblyHostBuilder.CreateDefault(args);
builder.RootComponents.Add<App>("#app");
builder.RootComponents.Add<HeadOutlet>("head::after");

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

await builder.Build().RunAsync();

```

Y tenemos que crear un fichero OrderState en la raíz para que no nos arroje un error.

Tenemos que inyectar el OrderState en el index (@inject OrderState OrderState)

Como OrderState será el que controle los estados del pedido tenemos que mover los métodos (ShowConfigurePizzaDialog(), CancelConfigurePizzaDialog(), ConfirmConfigurePizzaDialog(), RemoveConfiguredPizza y añadir el ResetOrder) a la nueva clase de OrderState. Además, debemos de modificar las llamadas de estos métodos desde el index.razor.

## Etapa 5: Pago con validación

En esta etapa vamos a agregar la pantalla de pago, para lo cual tenemos que agregar a la carpeta Pages de nuestro proyecto el archivo Checkout.razor .

En este archivo Checkout.razor debemos de incorporar:

- Enrutamientos @Pages “/chekout”

- Inyección de dependencias

@inject HttpClient HttpClient – Gestiona las peticiones Http

@inject OrderState OrderState – Gestiona el estado de los pedidos

@inject NavigationManager NavigationManager – Gestiona la navegación entre páginas

- Movemos el método PlaceOrder() desde index.razor a Checkout.razor

Además, tenemos que modificar el botón que tenemos en Index.razor que envía los pedidos

```
<div class="order-total @(Order.Pizzas.Any() ? "" : "hidden")">
  Total:
  <span class="total-price">@Order.GetFormattedTotalPrice()</span>
  <a href="/checkout" class="@(Order.Pizzas.Count == 0 ? "btn btn-warning disabled" : "btn btn-warning")">
    Pedir >
  </a>
</div>
```

El siguiente paso es gestionar la dirección de entrega del pedido, para lo que debemos de crear un componente dentro de la carpeta Shared llamado AddressEditor.razor. Dentro incluimos los contenedores con los campos necesarios para introducir las direcciones de entrega, junto con un bloque de código donde definimos la clase Address que será utilizada como un componente, con un getter y un setter, y que tiene un constructor.

```
AddressEditor.razor*  x ConfigurePizzaDialog.razor Checkout.razor Index.razor
43
44 @code {
45     [Parameter, EditorRequired] public Address Address { get; set; } = new();
46 }
47
```

Ahora ya se puede instalar en la clase Checkout.razor los datos proporcionados en los formularios de la clase AddressEditor.razor. En la página web del checkout aparecerán y se capturarán los datos de la dirección del pedido, serializando conjuntamente el pedido y los datos de la dirección.

Nota: En la documentación se indica que se puede descargar la herramienta “DB Browser for SQLite” para ver el contenido serializado.

Ahora hay que hacer la validación de datos del lado del servidor, y también del lado del cliente, puesto que ahora mismo se puede hacer un pedido dejando la dirección de entrega en blanco.

En la clase Address.cs del proyecto BlazingPizza.Shared debemos de incluir las anotaciones correspondientes a los datos que son obligatorios y la longitud de los campos, en el caso de que no se rellenen los campos “required” no se procesará el pedido:

```

Address.cs*  AddressEditor.razor  ConfigurePizzaDialog.razor  Checkout.razor*  Index.razor
BlazingPizza.Shared  BlazingPizza.Address  Name
1  using System.ComponentModel.DataAnnotations; // Incorporamos las anot
2  namespace BlazingPizza;
3
4  public class Address
5  {
6      public int Id { get; set; }
7
8      [Required, MaxLength(100)]
9      public string Name { get; set; } = string.Empty;
10
11     [Required, MaxLength(100)]
12     public string Line1 { get; set; } = string.Empty;
13
14     [MaxLength(100)]
15     public string Line2 { get; set; } = string.Empty;

```

Ahora para hacer la validación del lado cliente, en el Checkout.razor quitamos el Onvalidate="PlaceOrder" del EditForm y añadimos al botón @onClick="PlaceOrder" quitando el dissabled. Además, añadimos los componentes DataAnnotationsValidator y ValidationSummary.

```

AddressEditor.razor  ConfigurePizzaDialog.razor  Checkout.razor*  Index.razor
<button class="checkout-button btn btn-warning" @onclick="PlaceOrder">
    Realizar pedido
</button>

<DataAnnotationsValidator /> <!-- Componentes para hacer la validacion del formulario-->
<ValidationSummary />
</EditForm>

```

Para personalizar los mensajes de validación, tendremos que hacer modificaciones en la clase Address.cs y en AddressEditor.razor del proyecto BlazingPizza.shared:

```

Address.cs
1  public class Address
2  {
3      public int Id { get; set; }
4
5      [Required(ErrorMessage = "Debe introducir un nombre.")]
6      [MaxLength(100)]
7      [Display(Name = "Nombre")]
8      public string Name { get; set; } = string.Empty;
9
10     [Required(ErrorMessage = "Debe introducir una dirección de envío.")]
11     [MaxLength(100)]

```

```

AddressEditor.razor
<!-- Formulario con campos para introducir las direcciones d
<div class="form-field">
    <label>Nombre:</label>
    <div>
        <input @bind="Address.Name" />
        <ValidationMessage For="@(() => Address.Name)" />
    </div>
</div>

<div class="form-field">
    <label>Dirección 1:</label>
    <div>

```

Podemos modificar el input @bin por el InputText @bin-Value quedando más estético.

Añadiendo, en Checkout.razor dentro del bloque @code, un bool isSubmitting evitamos que el formulario se envíe más veces en caso de apretar varias veces al botón si es que tarda en cargar la acción de apretar el botón.



## Etapa 6: Autenticación y autorización

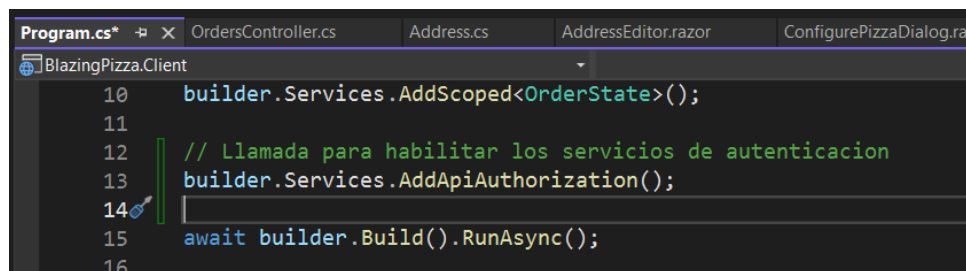
En esta etapa insertaremos el código necesario para que los usuarios puedan iniciar sesión.

Como inicio tenemos que incorporar la directiva `using Microsoft.AspNetCore.Authorization` y el atributo `[Authorize]`, que controla que solo los roles autorizados puedan acceder al controlador. La etiqueta `[Authorize]` hay que incluirla en todas las páginas en las que va a ver datos privados de pedidos y donde habrá que presentar un token de autenticación, posteriormente habrá que gestionar las autorizaciones, para que solo un usuario autenticado y autorizado pueda ver sus pedidos.

El proceso de autenticación hará lo siguiente:

- El proceso de autenticación nos redirigirá a la página de inicio de sesión
- En el inicio de sesión, la app se prepara para redirigir al punto de conexión de autorización del proveedor de identidades. La app proporciona una devolución de llamada de inicio de sesión y se procesa la respuesta de autenticación, si la autenticación es correcta se redirige a la página solicitada, si no es correcto se redirige a una página de inicio fallido.
- WebAssembly carga el punto de conexión procesando la respuesta, si la autenticación es correcta se redirige a la página solicitada, si no es correcto se redirige a una página de inicio fallido mostrando un error.

Primero, en el proyecto en el archivo `Program.cs` del cliente, se realiza una llamada a la API de autorización



```

10 builder.Services.AddScoped<OrderState>();
11
12 // Llamada para habilitar los servicios de autenticación
13 builder.Services.AddApiAuthorization();
14
15 await builder.Build().RunAsync();
16

```

El proyecto server ya incorpora la configuración para poder usar `IdentityServer`, y también se ha configurado el `appsettings.json` para emitir los tokens a la aplicación cliente. Por lo que nos faltaría crear dentro de Pages en el proyecto cliente una nueva página llamada `Authentication.razor`, que gestionará la autenticación de los usuarios.

En esta nueva página `Authentication.razor` incluimos:

- Direccionamiento con `@page "/authentication/{action}"`
- Recogemos el parámetro de la Action mediante el componente `RemoteAuthenticatorView`:  
`<RemoteAuthenticatorView Action="@Action">`
- Incluimos un bloque de código en donde marcamos como parámetro la "Action" para que pueda ser llamada desde fuera de nuestro componente.

```

Authentication.razor*
1  @page "/authentication/{action}"
2
3  <RemoteAuthenticatorView Action="@Action" />
4
5  @code {
6      [Parameter]
7      public string Action { get; set; } = string.Empty;
8  }
9

```

En el proyecto cliente, dentro del archivo App.razor, envolvemos todo el contenido con el `CascadingAuthenticationState`, para que fluya el estado de autenticación por toda la app:

```

App.razor*
1  <!-- Envolvemos todo con CascadingAuthenticationState para que fluya el estado de autent
2  <CascadingAuthenticationState>
3      <Router AppAssembly="typeof(Program).Assembly" Context="routeData">
4          <Found>
5              <RouteView RouteData="routeData" DefaultLayout="typeof(MainLayout)" />
6          </Found>
7          <NotFound>
8              <LayoutView Layout="typeof(MainLayout)">
9                  <div class="main">Lo sentimos, no hay nada en esta direccion.</div>
10             </LayoutView>
11         </NotFound>
12     </Router>
13 </CascadingAuthenticationState>
14

```

Ahora, en la carpeta Shared del proyecto Cliente, tenemos que crear un nuevo componente llamado `LoginDisplay.razor`, en donde se mostraran los vínculos para registrarse, iniciar sesión, ver el perfil y el botón de cierre de sesión. Añadimos también un bloque de código en donde incluiremos la acción del botón para desloguearnos.

Nos faltaría incluir en la vista principal, situada en la carpeta Shared del proyecto cliente, el `<LoginDisplay>`, y además rodear el enlace de `MyOrders` con un `<AuthorizeView>`

En este punto, ya podemos registrar un usuario y acceder, el siguiente paso es el de solicitar un token de acceso.

El controlador de mensajes adquiere el token y los adjunta a las solicitudes en el encabezado, para ello, dentro del proyecto cliente, en la clase `Program.cs`, añadimos el servicio para configurar el `httpClient` (`OrdersClient`), estableciendo su `BaseAddress` y agregando el gestor `BaseAddressAuthorizationMessageHandler` (el cual nos permite manejar las autorizaciones).

```

Program.cs
13 // Añadimos un servicio que nos permita configurar e inyectar el httpClient y la direccion, con el gestor que nos permita
14 // agregar los encabezados de autorizacion
15 builder.Services.AddHttpClient<OrdersClient>(client => client.BaseAddress = new Uri(builder.HostEnvironment.BaseAddress))
16     .AddHttpMessageHandler<BaseAddressAuthorizationMessageHandler>();
17
18 // Llamada para habilitar los servicios de autentificacion
19 builder.Services.AddApiAuthorization();
20
21 await builder.Build().RunAsync();
22

```

Ahora toca implementar la gestión de los pedidos en las diferentes páginas que tenemos y visualizan los pedidos de los clientes:

`Checkout.razor`:

- En primer lugar, inyectamos el `ordersClient` `@inject OrdersClient ordersClient`
- Modificamos el método `PlaceOrder` para poder manejar la excepción en el caso de que el token de autenticación no sea válido y que nos lleve a la pantalla de error.

MyOrders.razor:

- En primer lugar, inyectamos el ordersClient `@inject OrdersClient ordersClient`
- Modificamos para que en el caso de que el `getOrders` de error se pueda manejar la excepción en el caso de que el token de autenticación no sea válido y que nos lleve a la pantalla de error.

OrdersDetails.razor:

- En primer lugar inyectamos el ordersClient `@inject OrdersClient ordersClient`
- Modificamos para que en el caso de que el `getOrder` de error se pueda manejar la excepción en el caso de que el token de autenticación no sea válido y que nos lleve a la pantalla de error.

En este estado todos los usuarios autenticados pueden acceder a las consultas de pedidos, tenemos que implementar la autorización.

Para autorizar, buscamos en el proyecto Server dentro del `OrdersController.cs`, en los métodos `getOrders` y `getOrderWithStatus` las cláusulas `where` que asocian los usuarios con su id, garantizando que solo listan los pedidos de su id.

Ahora, en el componente que controla la app en el lado cliente, `App.razor`, modificamos la vista de la ruta para que la navegación sea posible en el caso de que el usuario este autorizado (cambiando el `RouteView` por un `AuthorizeRouteView`).

```

<!-- Envolvemos todo con CascadingAuthenticationState para que fluya el estado de autenticación a través de la app -->
<CascadingAuthenticationState>
  <Router AppAssembly="typeof(Program).Assembly" Context="routeData">
    <Found>
      <!-- Modificamos el RouteView por AuthorizeRouteView para gestionar el acceso de usuarios autorizados -->
      <AuthorizeRouteView RouteData="routeData" DefaultLayout="typeof(MainLayout)" >
        <NotAuthorized>
          <p>No está autorizado para acceder al recurso.</p>
        </NotAuthorized>
        <Authorizing>
          <div class="main">Por favor espere...</div>
        </Authorizing>
      </AuthorizeRouteView>
    </Found>
    <NotFound>
      <LayoutView Layout="typeof(MainLayout)">
        <div class="main">Lo sentimos, no hay nada en esta dirección.</div>
      </LayoutView>
    </NotFound>
  </Router>
</CascadingAuthenticationState>

```

Si el cliente no está autorizado, lo redireccionaremos a la página de inicio de sesión. Para ello, creamos un nuevo componente dentro de la carpeta `Shared` del proyecto cliente, llamado `RedirectToLogin.razor`.

```

1 <!-- Página que redireccionará al inicio de sesión en el caso de que el usuario no esté autorizado a visitar la página -->
2 <!-- Inyectamos el navegador -->
3 @inject NavigationManager Navigation
4 @code {
5   protected override void OnInitialized()
6   {
7     // Navegamos a la página de inicio de sesión cuando se inicializa la página
8     Navigation.NavigateToLogin($"authentication/login?returnUrl={Navigation.Uri}");
9   }

```

Antes de realizar el siguiente paso para persistir datos, tenemos que crear la clase `PizzaAuthenticationState.cs`, que hereda de `RemoteAuthenticationState`, en donde se recogerá la información del pedido junto con el estado de autenticación, que nos permitirá realizar la persistencia de los pedidos que hemos configurado antes de realizar el login.

En el siguiente paso, vamos a realizar que el estado del pedido persista, puesto que si hacemos un pedido sin habernos logueado, al autenticarnos perdemos el pedido que teníamos seleccionado previamente. Para ello, modificamos el componente de la página `Authentication.razor` (situado en el proyecto cliente dentro de la carpeta `Pages`), añadiendo:

- Las inyecciones del OrderState y del NavigationManager
- El direccionamiento de la página
- Se define el RemoteAuthenticatorViewCore, en donde se pasan parámetros , entre ellos el estado de autenticación y el método que se aplica cuando se inicializa la autenticación, en este caso el RestorePizza.
- Definimos un método ReplaceOrder en OrderState.cs que será llamado cuando se autentique el usuario
- En la clase Program.cs debemos incluir el estado de autenticación cuando implementamos el servicio de la autorización de la API

Solo nos faltaría realizar el cierre de sesión:

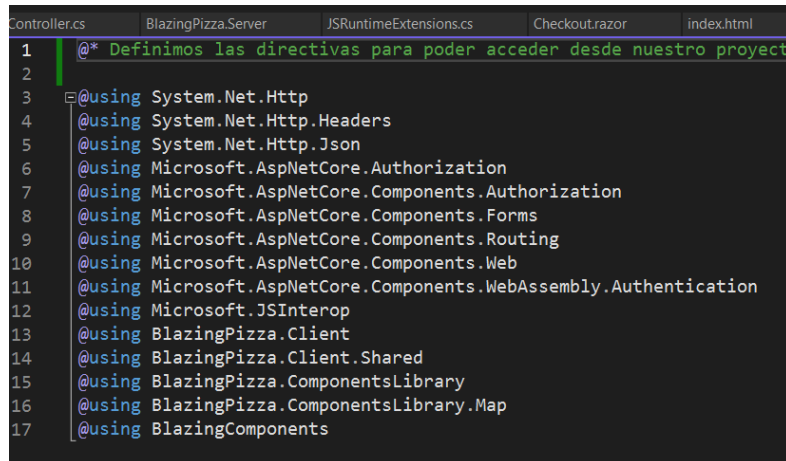
Para realizar el cierre de sesión tan solo debemos de añadir en Program.cs la opción para el logout:

```
17
18 // Llamada para habilitar los servicios de autenticacion, incluimoss el <PizzaAuthenticationState>
19 // La parte options implica el codigo necesario para poder hacer el logout
20 builder.Services.AddApiAuthorization<PizzaAuthenticationState>(options => {
21     options.AuthenticationPaths.LogOutSucceededPath = "";
22 });
23
24 await builder.Build().RunAsync();
25
```

## Etapa 7: JavaScript

Con JavaScript vamos a implementar un mapa para hacer un seguimiento en tiempo real y poder localizar el pedido.

En la raíz de nuestro proyecto cliente añadimos el componente `_Imports.razor`, para poder acceder desde nuestro cliente a los archivos que incluyamos en el proyecto `ComponentsLibrary`.

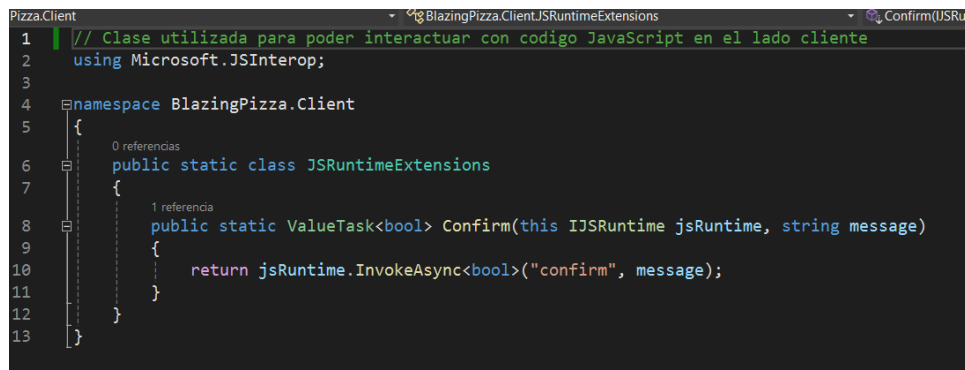


```

1  @* Definimos las directivas para poder acceder desde nuestro proyecto
2
3  @using System.Net.Http
4  @using System.Net.Http.Headers
5  @using System.Net.Http.Json
6  @using Microsoft.AspNetCore.Authorization
7  @using Microsoft.AspNetCore.Components.Authorization
8  @using Microsoft.AspNetCore.Components.Forms
9  @using Microsoft.AspNetCore.Components.Routing
10 @using Microsoft.AspNetCore.Components.Web
11 @using Microsoft.AspNetCore.Components.WebAssembly.Authentication
12 @using Microsoft.JSInterop
13 @using BlazingPizza.Client
14 @using BlazingPizza.Client.Shared
15 @using BlazingPizza.ComponentsLibrary
16 @using BlazingPizza.ComponentsLibrary.Map
17 @using BlazingComponents
  
```

En el componente `OrderDetails.razor` del proyecto cliente dentro de `Pages`, tendremos que añadir un apartado para que se nos muestre el detalle de la localización en el mapa del pedido.

En el proyecto cliente, añadimos una clase `JSRuntimeExtensions.cs` que implemente el método para llamar a JavaScript. Añadimos la directiva `using Microsoft.JSInterop`.



```

1  // Clase utilizada para poder interactuar con código JavaScript en el lado cliente
2  using Microsoft.JSInterop;
3
4  namespace BlazingPizza.Client
5  {
6      public static class JSRuntimeExtensions
7      {
8          public static ValueTask<bool> Confirm(this IJSRuntime jsRuntime, string message)
9          {
10             return jsRuntime.InvokeAsync<bool>("confirm", message);
11          }
12      }
13  }
  
```

Ahora añadiremos en nuestro `Index.razor`, situado en la carpeta `Pages` del proyecto cliente, la inyección de para realizar las llamadas de JavaScript mediante `@inject IJSRuntime JS`. Además, incluimos el método `RemovePizza`.

## Etapa 8: Plantilla de componentes

En esta etapa se creará un nuevo proyecto con la biblioteca para componentes, y haremos plantillas para poder reutilizar esos componentes en los distintos proyectos. Con las plantillas podremos mostrar un cuadro de dialogo o realizar la carga de datos.

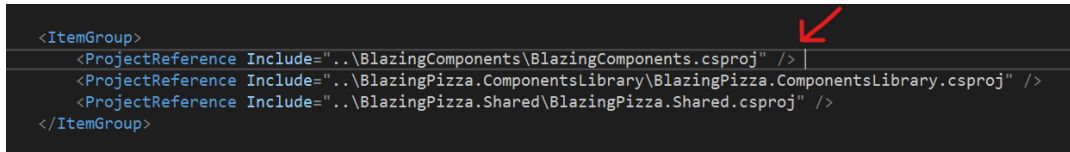
Para hacer un nuevo proyecto, haremos clic sobre el contenedor de proyectos, y elegiremos un proyecto de Biblioteca de clases Razor. En el archivo BlazingComponents.csproj podremos ver las características.

PROBLEMA: Debido a la configuración del Visual Studio, aparecen errores en cuanto a la compatibilidad de dependencias al igual que me ocurría al inicio del taller. Para solucionarlo, en administración de nuggets (también se puede hacer en el BlazingComponents.csproj), incorporo la dependencia de Microsoft.Extensions.DependencyInjection, posteriormente se realiza en la consola de administrador de nuggets un “dotnet restore”, y en administración de paquetes nuggets desinstalo el Microsoft.AspNetCore.Components.Web que es el que está dándome problemas, reinstalándolo, pero en este caso la versión 7.0 que es la compatible con el proyecto.

La primera plantilla que crearemos será la de TemplatedDialog.razor. Haciendo clic sobre el proyecto creamos el nuevo componente e incluimos:

- Un contenedor div para el dialog.
- Un bloque de código donde agregamos un parámetro ChildContent, del tipo RenderFragment que nos permitirá para la personalización de los componentes.

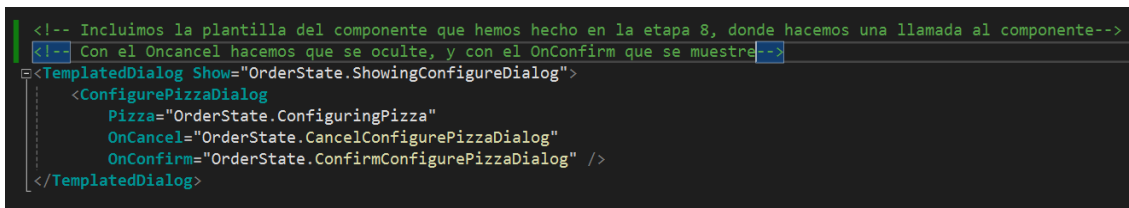
Para poder utilizar el componente TemplatedDialog.razor en nuestros proyectos debemos de agregar la referencia al Cliente.csproj:



```
<ItemGroup>
  <ProjectReference Include="..\BlazingComponents\BlazingComponents.csproj" />
  <ProjectReference Include="..\BlazingPizza.ComponentsLibrary\BlazingPizza.ComponentsLibrary.csproj" />
  <ProjectReference Include="..\BlazingPizza.Shared\BlazingPizza.Shared.csproj" />
</ItemGroup>
```

Ahora, en nuestro componente ConfigurePizzaDialog.razor, situado en la carpeta Shared del proyecto cliente, realizamos una simplificación de etiquetas para que sea compatible con la plantilla que hemos hecho (eliminamos las dos carpetas que rodean el dialog).

Para usar el nuevo componente, editamos el componente Index.razor de nuestro proyecto cliente, incluimos la llamada al componente, y las acciones para que se muestre o desaparezca.



```
<!-- Incluimos la plantilla del componente que hemos hecho en la etapa 8, donde hacemos una llamada al componente-->
<!-- Con el OnCancel hacemos que se oculte, y con el OnConfirm que se muestre-->
<TemplatedDialog Show="OrderState.ShowingConfigureDialog">
  <ConfigurePizzaDialog
    Pizza="OrderState.ConfiguringPizza"
    OnCancel="OrderState.CancelConfigurePizzaDialog"
    OnConfirm="OrderState.ConfirmConfigurePizzaDialog" />
</TemplatedDialog>
```

Ahora en el taller se plantea una plantilla que gestione la lista y estado de los pedidos. Crearemos en el proyecto BlazorComponents un componente llamado TemplatedList.razor.

- En primer lugar, establecemos una directiva indicando el parámetro Titem
- Incluimos una estructura condicional que gestione los estados de la lista de pedidos (nula, vacía o con pedidos).

- Un bloque de código que declara la variable `items`, y que define los distintos parámetros que establece en bloque condicional, y que serán necesarias para rellenar el fragmento que se mostrará.
- Por último se incluye un método, asíncrono, que realiza la carga de los datos.

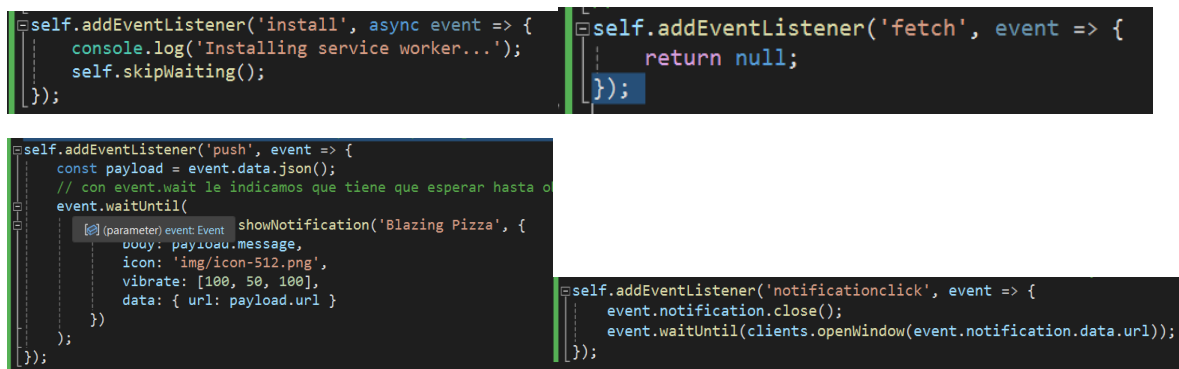
## Etaapa 9: Aplicación Web Progresiva PWA

En esta etapa se plantea el uso de las API del navegador para instalar nuestra aplicación en la barra de tareas, trabajar sin conexión o recibir notificaciones push.

Para realizar estas funciones es necesario la instalación de un servicio, será un archivo de código JavaScript, el cual crearemos dentro del proyecto cliente en la carpeta wwwroot el archivo service-worker.js.

En este archivo incluimos varias acciones relativas a los eventos que pueden suceder:

- Evento de instalación, indicando que inmediatamente pasea un estado activo.
- Evento de solicitud de red, controla si utilizamos los datos en cache si estamos offline.
- Evento de petición push, guardando en la variable payload los datos de la petición transformándolos en un Json.
- Evento de clic sobre notificación, donde se inicializa cuando hacemos clic sobre una notificación, entonces cierra la notificación y abre el navegador con la url.



```

self.addEventListener('install', async event => {
  console.log('Installing service worker...');
  self.skipWaiting();
});

self.addEventListener('fetch', event => {
  return null;
});

self.addEventListener('push', event => {
  const payload = event.data.json();
  // con event.wait le indicamos que tiene que esperar hasta o
  event.waitUntil(
    (parameter) event: Event    showNotification('Blazing Pizza', {
      body: payload.message,
      icon: 'img/icon-512.png',
      vibrate: [100, 50, 100],
      data: { url: payload.url }
    })
  );
});

self.addEventListener('notificationclick', event => {
  event.notification.close();
  event.waitUntil(clients.openWindow(event.notification.data.url));
});

```

Ahora vamos a hacer que nuestra aplicación sea instalable en un sistema operativo.

- En primer lugar, creamos un manifiesto en la carpeta wwwroot, que indica como tiene que actuar el navegador.
- En el archivo index.html incluimos una referencia indicando donde encontrar el manifiesto.
- Además, en el Index.html incluimos el script que habilita el service.worker.js.

Ahora, cuando entremos en la web a través del PC o móvil tendremos la posibilidad de instalar la app.

El siguiente paso es el envío de notificaciones push, obtener suscripciones, etc. Antes de poder enviar notificaciones hay que pedir permiso, el navegador genera una suscripción.

- En el componente Checkout.razor incorporamos una llamada a la petición de suscripción a la inicialización del componente (en el método onInitialized()).
- Y añadimos también el método que define como será la petición de la notificación de suscripción. Este método hace que antes de continuar esperemos a que la clase OrdersClient lance el método de suscribirse a las notificaciones.
- Importante insertar el servicio en el componente @inject IJSRuntime IJSRuntime.

Por lo tanto, tendremos que incorporar en la clase OrdersClient.cs el método SubscribeNotifications método que permitirá suscribirnos a las notificaciones (llamado desde Checkout.razor), se le pasa



una notificación, el método no tiene que esperar (await), liberando el hilo, mientras se hace una petición http del tipo Put a la dirección notification/susbscribe, en función de la respuesta, arrojará un código de error y lanzará excepción:

```
// Metodo que permite subscribirnos a las notificaciones, se le pasa una notificación
// El metodo no tiene que esperar (await) mientras se hace una petición http del tipo put a la direc
1 referencia
public async Task SubscribeToNotifications(NotificationSubscription subscription)
{
    var response = await httpClient.PutAsJsonAsync("notifications/subscribe", subscription);
    // En función de la respuesta arrojará un código de error y lanzará excepción
    response.EnsureSuccessStatusCode();
}
```

Ahora tenemos suscripciones, por lo tanto, podemos enviar notificaciones. Pasos a tener en cuenta:

- Incorporar paquete Nugget WebPush en el proyecto server.
- En OrderController.cs hay que incluir los métodos TrackAndSendNotificationAsync y SendNotificationAsync, ambos métodos se encargan del seguimiento y envío de notificaciones.

Por último, tenemos que incluir dos métodos en el service-worker.js:

- Self.addEventListener('push', event), que indica a la aplicación que hacer con las notificaciones que entran.
- Self.addEventListener('notificationclick', event), que indica a la aplicación que hacer cuando el usuario clicla la notificación (cierra la notificación y abre una ventana en donde se encontrara la información sobre el estado del pedido.

Bibliografia:

[Estructura del proyecto de Blazor de ASP.NET Core | Microsoft Learn](#)