



.Net

Teoría 11

Contenedor de Inyección de dependencias (DI-Container)



Crear una aplicación de consola llamada DIContainer



1. Abrir una terminal del sistema operativo
2. Cambiar a la carpeta `proyectosDotnet`
3. Crear la aplicación de consola `DIContainer`
4. Abrir code en la carpeta `DIContainer`



Codificar la interfaz ILogger y la clase LoggerConsola



```
-----ILogger.cs-----
```

```
namespace DiContainer;  
public interface ILogger  
{  
    void Log(string mensaje);  
}
```

```
-----LoggerConsola.cs-----
```

```
namespace DiContainer;  
public class LoggerConsola : ILogger  
{  
    public void Log(string mensaje)  
    {  
        Console.WriteLine($"{DateTime.Now:hh:mm:ss:fff} {mensaje}");  
    }  
}
```



Codificar la interfaz IServicioX y la clase ServicioX



-----IServicioX.cs-----

```
namespace DiContainer;
public interface IServicioX
{
    void Ejecutar();
}
```

-----ServicioX.cs-----

```
namespace DiContainer;
public class ServicioX : IServicioX
{
    private readonly ILogger _logger;
    public ServicioX(ILogger logger)
    {
        this._logger = logger;
    }
    public void Ejecutar()
    {
        _logger.Log("ServicioX comenzando su ejecución");
        for (int i = 1; i <= 100_000_000; i++) ; //consumo tiempo simulando ejecución
        _logger.Log("ServicioX ejecución finalizada");
    }
}
```

Copiar el código del archivo
11_RecursosParaLaTeoria



Principio de inversión de dependencias Configuración de las dependencias

- Es conveniente agrupar la **configuración** de las dependencias en el código para facilitar futuros cambios en nuestra aplicación
- Idealmente para cambiar el comportamiento de la aplicación deberíamos:
 - 1) **Crear nuevas clases (dependencias)** que implementen determinadas interfaces
 - 2) **Configurar adecuadamente** la elección de las dependencias que se utilizarán



Principio de inversión de dependencias Configuración de las dependencias

- Para concentrar en nuestro código la configuración de las dependencias podemos delegar en una única clase la creación de las instancias de todas las dependencias.
- Como las dependencias pueden ser consideradas servicios, vamos a llamar a esa clase `ProveedorServicios`



Agregar la clase ProveedorServicios



```
-----ProveedorServicios.cs-----
```

```
namespace DiContainer;
```

```
class ProveedorServicios
```

```
{
```

```
    public ILogger GetLogger()
```

```
        => new LoggerConsola();
```

```
    public IServicioX GetServicioX()
```

```
        => new ServicioX(this.GetLogger());
```

```
}
```

Copiar el código del archivo
11_RecursosParaLaTeoria

Área concentrada del código donde
se configuran todas las dependencias

ProveedorServicios concentra la creación de todas las dependencias

```
class ProveedorServicios
{
    public ILogger GetLogger()
        => new LoggerConsola();
    public IServicioX GetServicioX()
        => new ServicioX(this.GetLogger());
}
```

Para crear un **ServicioX** se necesita un **Logger**. Esto no es problema para **ProveedorServicios** pues también puede autoproporcionárselo





Codificar Program.cs y ejecutar



```
using DiContainer;
```

```
var proveedor = new ProveedorServicios();  
var servicioX = proveedor.GetServicioX();  
servicioX.Ejecutar();
```

```
var logger = proveedor.GetLogger();  
logger.Log("Fin del programa");
```

Copiar el código del archivo
11_RecursosParaLaTeoria

```
11:29:30:272  ServicioX comenzando su ejecución  
11:29:30:621  ServicioX ejecución finalizada  
11:29:30:621  Fin del programa
```



Contenedor de Inyección de Dependencias

- En lugar de la clase `ProveedorServicios` utilizada en el ejemplo anterior, usaremos un `contenedor de inyección de dependencias`.
- Un contenedor de inyección de dependencias (`DI Container`) facilita configurar y obtener las dependencias que se usarán en la aplicación. También permite especificar si una dependencia debe usarse como `singleton` o debe crearse un nuevo objeto cada vez que se utilice

Contenedor de Inyección de Dependencias

Con “**Singleton**” nos referimos a una clase de la cual se va a instanciar un único objeto, por lo tanto la aplicación trabajará siempre con la misma instancia en cualquier parte del código.

Singleton es también un patrón de diseño





Contenedor de Inyección de Dependencias

- `.Net` provee un contenedor de inyección de dependencias por medio de las clases `ServiceCollection` y `ServiceProvider`
- Para utilizar estas clases en una aplicación de consola es necesario instalar un paquete `NuGet`
- `NuGet` es un administrador de paquetes gratuito y de código abierto diseñado para `.Net`



Contenedor de Inyección de Dependencias



- En la terminal del sistema operativo (o en la que provee el **Visual Studio Code**) posicionarse en la carpeta del proyecto (donde se encuentra el archivo **DiContainer.csproj**) y tipear el siguiente comando:

```
dotnet add package Microsoft.Extensions.Hosting
```

Este es el nombre del paquete
que se requiere instalar



Modificar Program.cs de la siguiente manera y ejecutar



```
using DiContainer;
using Microsoft.Extensions.DependencyInjection;

var servicios = new ServiceCollection();
servicios.AddTransient<ILogger, LoggerConsola>();
servicios.AddTransient<IServicioX, ServicioX>();
var proveedor = servicios.BuildServiceProvider();

var servicioX = proveedor.GetService<IServicioX>();
servicioX?.Ejecutar();

var logger = proveedor.GetService<ILogger>();
logger?.Log("Fin del programa");
```

Agregar esta
directiva using

La clase
ProveedorServicios
ya no es necesaria

Se registran los servicios y se
construye el proveedor

Copiar el código del archivo
11_RecursosParaLaTeoria

Contenedor de Inyección de Dependencias descripción del código presentado

```
servicios.AddTransient<IServicioX, ServicioX>();  
servicios.AddTransient<ILogger, LoggerConsola>();
```

Se registran los servicio `IServicioX` y `ILogger` en la colección de servicios, indicando que cuando se requiera un `IServicioX` debe proveerse una nueva instancia de la clase `ServicioX` y cuando se requiera un `ILogger` debe proveerse una nueva instancia de la clase `LoggerConsola`

Contenedor de Inyección de Dependencias descripción del código presentado

```
var proveedor = servicios.BuildServiceProvider();  
  
var servicio = proveedor.GetService<IServicioX>();
```

Se obtiene el proveedor de servicios a partir de la colección de servicios.

Se instancia y devuelve un objeto de la clase `ServicioX`. No debemos preocuparnos por las dependencias que requiere `ServicioX`, serán provistas por el contenedor

Contenedor de Inyección de Dependencias

Para que el **contenedor** pueda proveer los servicios requeridos se necesita:

1. Haber registrado el servicio y todas sus dependencias en el contenedor.
2. Utilizar en todos los casos inyección por medio del constructor.



Tiempo de vida de los servicios en un contenedor

- `servicios.AddTransient<ILogger, LoggerConsola>();`
Registra el servicio `LoggerConsola` como transitorio. El proveedor devolverá un nuevo objeto cada vez que se lo requiera.
- `servicios.AddSingleton<ILogger, LoggerConsola>();`
Registra el servicio `LoggerConsola` como singleton. El proveedor devolverá siempre el mismo objeto cada vez que se lo requiera.



Ejercicio práctico: Se requiere numerar las líneas de log



- Agregar un nuevo servicio `LoggerNum` que implemente la interfaz `ILogger` y que enumere las líneas que va imprimiendo en la consola



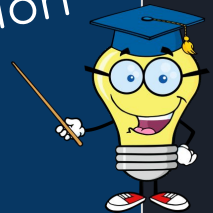
Ejercicio práctico: Se requiere numerar las líneas de log



- Agregar un nuevo servicio **LoggerNum** que implemente la interfaz **ILogger** y que enumere las líneas que va imprimiendo en la consola

```
namespace DiContainer;
class LoggerNum : ILogger
{
    private int _n;
    public void Log(string mensaje)
    {
        Console.WriteLine($"{{++_n}}:  {{DateTime.Now:hh:mm:ss:fff}}  {{mensaje}}");
    }
}
```

Posible
solución





Ejercicio práctico: Se requiere numerar las líneas de log



- Agregar un nuevo servicio **LoggerNum** que implemente la interfaz **ILogger** y que enumere las líneas que va imprimiendo en la consola
- En Program.cs realizar este único cambio: reemplazar **LoggerConsola** por **LoggerNum** en la instrucción de registro. Luego compilar y ejecutar para verificar el resultado





Ejercicio práctico: Se requiere numerar las líneas de log



- Agregar un nuevo servicio **LoggerNum** que implemente la interfaz **ILogger** y que enumere las líneas que va imprimiendo en la consola
- En Program.cs realizar este único cambio: reemplazar **LoggerConsola** por **LoggerNum** en la instrucción de registro. Luego compilar y ejecutar para verificar el resultado

Mal



```
1: 12:40:43:981  ServicioX comenzando su ejecución
2: 12:40:44:284  ServicioX ejecución finalizada
1: 12:40:44:285  Fin del programa
```



Ejercicio práctico: Se requiere numerar las líneas de log



- Agregar un nuevo servicio `LoggerNum` que implemente la interfaz `ILogger` y que enumere las líneas que va imprimiendo en la consola
- En `Program.cs` realizar este único cambio: reemplazar `LoggerConsola` por `LoggerNum` en la instrucción de registro. Luego compilar y ejecutar para verificar el resultado
- Registrar el servicio de Logger como singleton en lugar de transient con la instrucción

```
servicios.AddSingleton<ILogger, LoggerNum>();
```

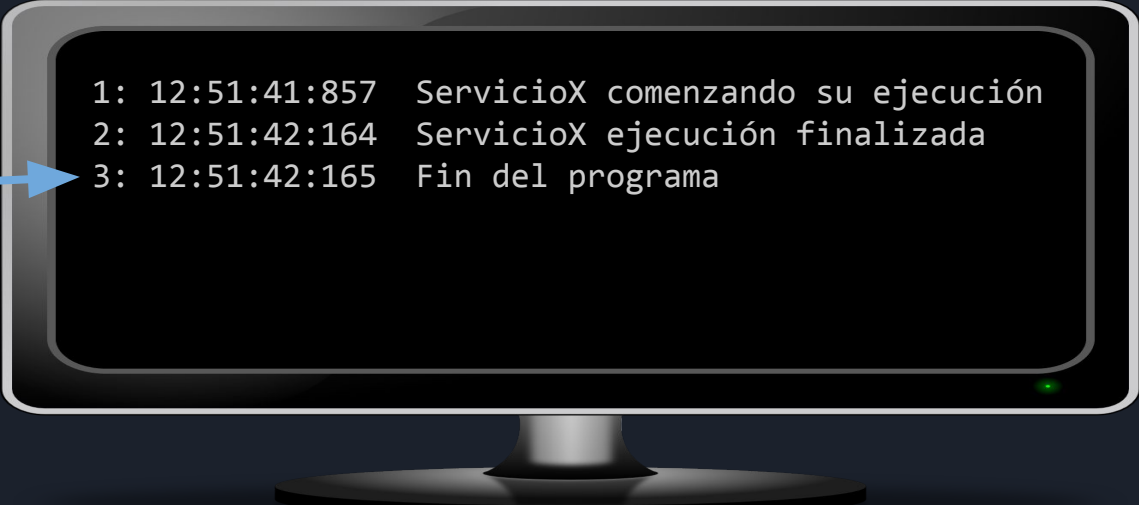


Principio de inversión de dependencias - Configuración de las dependencias - DI container

```
-----Program.cs-----  
using DiContainer;  
using Microsoft.Extensions.DependencyInjection;  
  
var servicios = new ServiceCollection();  
servicios.AddSingleton<ILogger, LoggerNum>();  
servicios.AddTransient<IServicioX, ServicioX>();  
var proveedor = servicios.BuildServiceProvider();  
  
var servicioX = proveedor.GetService<IServicioX>();  
servicioX?.Ejecutar();  
  
var logger = proveedor.GetService<ILogger>();  
logger?.Log("Fin del programa");
```

Se necesita siempre acceder a la misma instancia del servicio de log, por eso lo registramos como Singleton

Bien →



```
1: 12:51:41:857  ServicioX comenzando su ejecución  
2: 12:51:42:164  ServicioX ejecución finalizada  
3: 12:51:42:165  Fin del programa
```

Contenedor de Inyección de Dependencias

Hemos cambiado el comportamiento del programa agregando nuevo código y modificando sólo el área donde se registran los servicios

ii Principio OPEN/CLOSE !!



Tiempo de vida de los servicios en un contenedor de DI

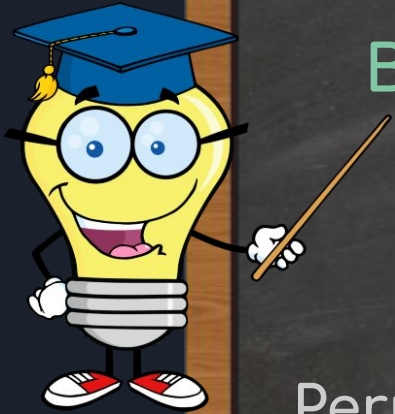
- Los servicios también se pueden registrar dentro de un scope (alcance o ámbito). Resulta útil en las aplicaciones web **ASP. NET Core**

```
servicios.AddScoped<IservicioA, ServicioA>();
```

- Se devuelve la misma instancia dentro del mismo ámbito. Para una aplicación **Blazor Server** se crea un ámbito por cada conexión **SignalR**, Las instancias se compartirán entre páginas y componentes para un mismo usuario, pero no entre diferentes usuarios y no entre diferentes pestañas del mismo explorador.

Aplicaciones Web con ASP.NET Core Blazor

¿Qué es Blazor?



Blazor es un framework de interfaz de usuario para .NET

Es parte de ASP NET Core

Permite crear SPAs (Single-page application) usando como lenguajes de programación C# y Razor Pages, haciendo nula la necesidad de programar en Javascript o frameworks derivados

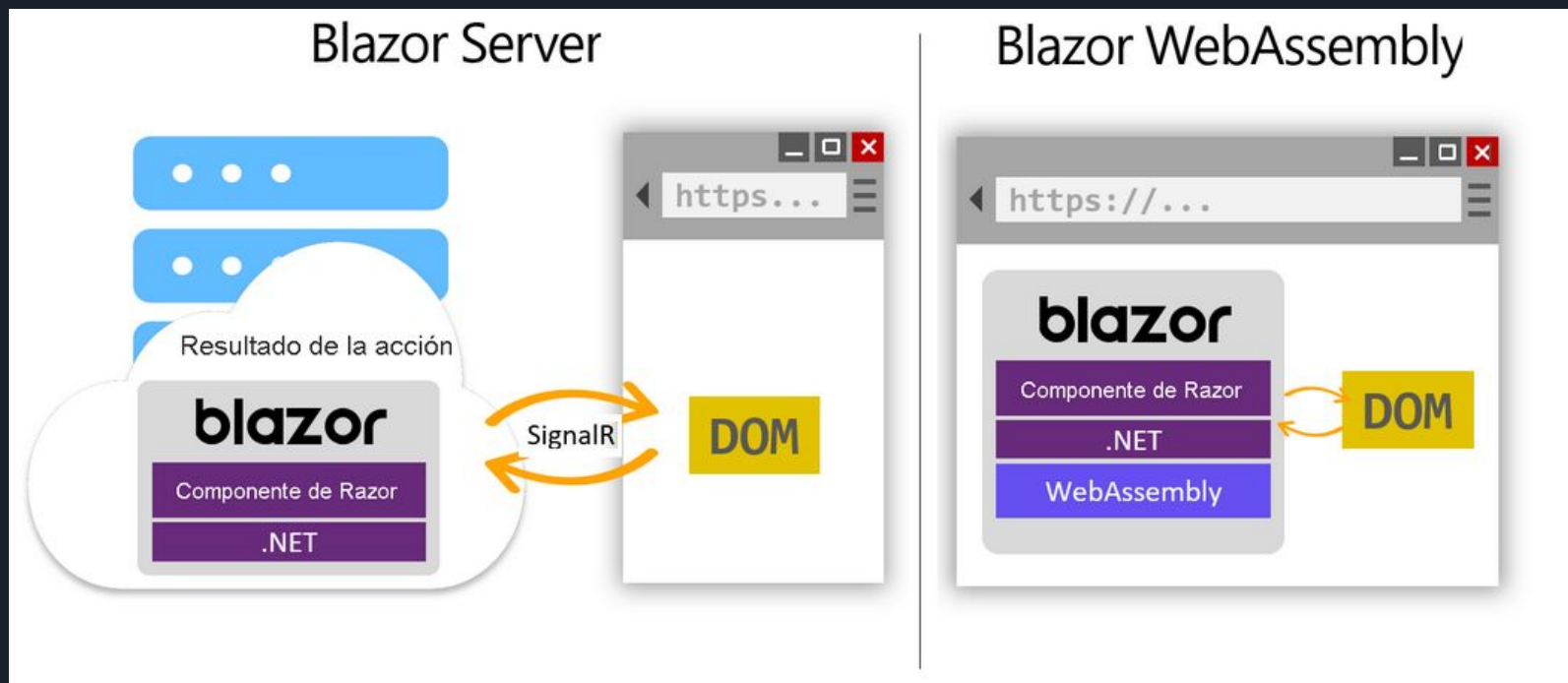
¿Qué es Razor?

Razor es un formato para generar contenido basado en texto como **HTML**. Los archivos **Razor** tienen una extensión de archivo **cshtml** o **razor** y contienen una combinación de código **C#** junto con **HTML**



¿Aplicación del lado del cliente o del servidor?

Las **aplicaciones Blazor** se pueden ejecutar en un servidor como parte de una aplicación ASP.NET o en el explorador del usuario.





Aplicación Blazor Server

- Una **aplicación Blazor Server** se implementa en un servidor web.
- El servidor mantiene con el navegador del usuario un canal de **comunicación bidireccional SignalR**.
- Las acciones de los usuarios sobre la **aplicación** se transmiten por esta conexión **SignalR** al servidor y, si es necesario actualizar la interfaz de usuario, el **framework de Blazor Server** envía en tiempo real al navegador los cambios para que se apliquen a la interfaz de usuario



Aplicación Blazor WebAssembly

- En una **aplicación Blazor WebAssembly**, las **DLL** de la aplicación se transmiten al navegador del usuario y se ejecutan sobre una versión de .NET optimizada para el entorno de ejecución **WebAssembly** del navegador.
- Se desplaza todo el procesamiento de la aplicación a la máquina del usuario. Para obtener datos o interactuar con otros servicios, la aplicación puede usar tecnologías web estándar para comunicarse con servicios HTTP.



Componentes

- Las aplicaciones **Blazor** se basan en componentes.
- Un componente es un elemento de la interfaz de usuario, como una página, un cuadro de diálogo o un formulario de entrada de datos.
- Utilizan sintaxis **Razor** (**C#** y **HTML**) y se escriben en archivos con extensión **.razor**
- Los componentes se compilan en clases **.NET**
- Se pueden anidar y reutilizar.
- Pueden ser “ruteables” (directiva **@page**)



Crear un proyecto Blazor Server



1. Abrir una terminal del sistema operativo
2. Cambiar a la carpeta `proyectosDotnet`
3. Crear la aplicación de Blazor Server con el comando:

Indicamos que no vamos a utilizar una conexión segura https para este proyecto

```
dotnet new blazor --no-https -o HolaBlazor
```

4. Abrir `Visual Studio Code` sobre este proyecto y ejecutar

EXPLORER

> SOLUTION EXPLORER

- > HOLABLAZOR
 - > bin
 - > Components
 - > obj
 - > Properties
 - > wwwroot
 - { } appsettings.Development.js...
 - { } appsettings.json
 - 🔥 HolaBlazor.csproj
 - ≡ HolaBlazor.sln
 - C# Program.cs
- > OUTLINE
 - No symbols found in document 'Program.cs'

Program.cs

```
1 using
2
3 var
4
5 // A
6 buil
```

PROBLEMS

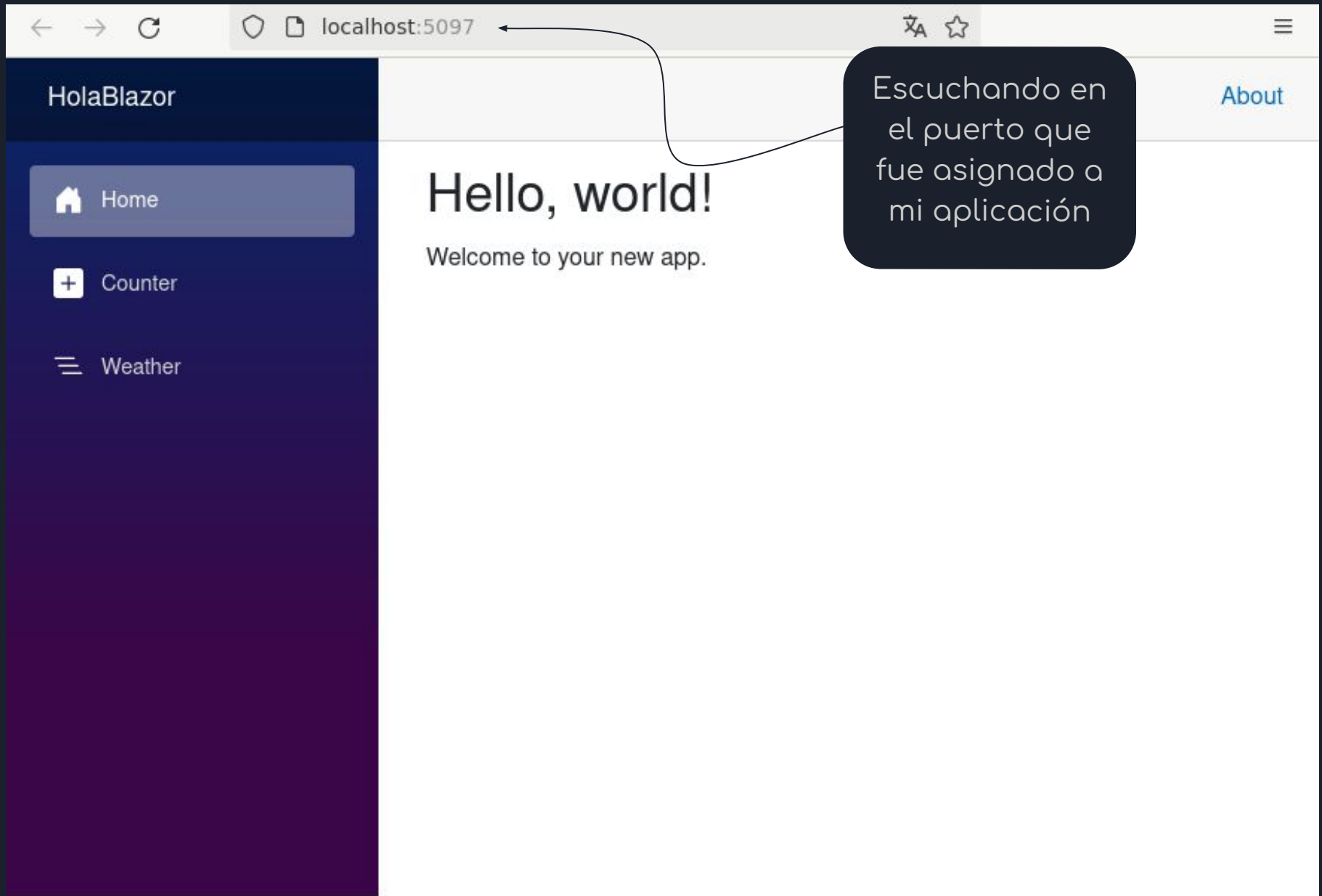
Filter (e.g. text, ...)

info: Microsoft.Hosting.Lifetime[14]
Now listening on: http://localhost:5097
Microsoft.Hosting.Lifetime: Information: Now listening on:
http://localhost:5097
info: Microsoft.Hosting.Lifetime[0]
Application started. Press Ctrl+C to shut down.
Microsoft.Hosting.Lifetime: Information: Application start
ed. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
Hosting environment: Development
Microsoft.Hosting.Lifetime: Information: Hosting environme

En entorno de desarrollo

Escuchando en el puerto 5040.
Un número que oscila entre 5000 y
5300 y se asigna automáticamente
en la creación del proyecto
Se guarda en el archivo
/Properties/launchSettings.json
junto a otras configuraciones

< 0 0 0 0 C#: HolaBlazor (HolaBlazor) Projects: 1 Spaces: 4 UTF-8 CRLF C# 36



Primero
veamos algo
de HTML

Program.cs clientes.html

wwwroot > <> clientes.html

1

2

!

!

!!!

Emmet Abbreviation

<!DOCTYPE html>

<html lang="en">

<head>

<meta charse

<meta name= viewport content= width=device-

width, initial-scale=1.0">

<title>Document</title>

</head>

<body>

En **clientes.html**
Tipear **!** y presionar
la tecla [Enter]

Crear el archivo **clientes.html** en la carpeta
wwwroot que es la raíz web de la aplicación.
Acá se colocan los recursos estáticos
públicos de la aplicación.





EXPLORER

Program.cs

clientes.html



Visual Studio Code
nos crea el esquema
básico de un
documento **html**

wwwroot > clientes.html > ...

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <meta name="viewport" content="width=device-width, initial-scale=1">
6   <title>Document</title>
7 </head>
8 <body>
9
10 </body>
11 </html>
12
```

Entre las etiquetas
<body> y **</body>**
colocaremos el
contenido visible de
la página web

PROBLEMS

OUTPUT

DEBUG CONSOLE

Filter (e.g. text, !exclude, \escape)

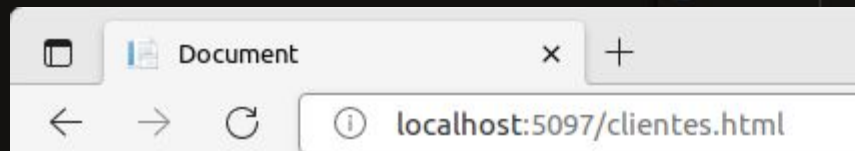
```
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5000
```

No cerrar la aplicación, que siga corriendo

```
4 <meta charset="UTF-8">
5 <meta name="viewport" content="width=device-width, initial-scale=1.0">
6 <title>Document</title>
7 </head>
8 <body>
9 <h1>Listado de clientes</h1>
10 </body>
11 </html>
12
```

Agregar, salvar el archivo y con el navegador acceder a </clientes.html>

Entre las etiquetas `<h1>` y `</h1>` se coloca un encabezado de nivel 1



Listado de clientes



Elementos HTML

- La mayoría de los elementos se definen con un tag de apertura y uno de cierre. Ejemplo:

```
<p></p>
```

- Hay algunas excepciones a esta regla. Ejemplo:

```
<br>, <img>
```

- Los tags pueden tener atributos. Ejemplo:

```
<a href="http://www.google.com">Google</a>
```

- Puede haber comentarios en el código HTML, el cuál no será procesado por el navegador. Ejemplo:

```
<!-- Esto es un comentario -->
```

Realizar las siguientes pruebas

```
<body>
```

```
  <h1> Listado de clientes </h1>
```

```
  <h2>encabezado 2</h2>
```

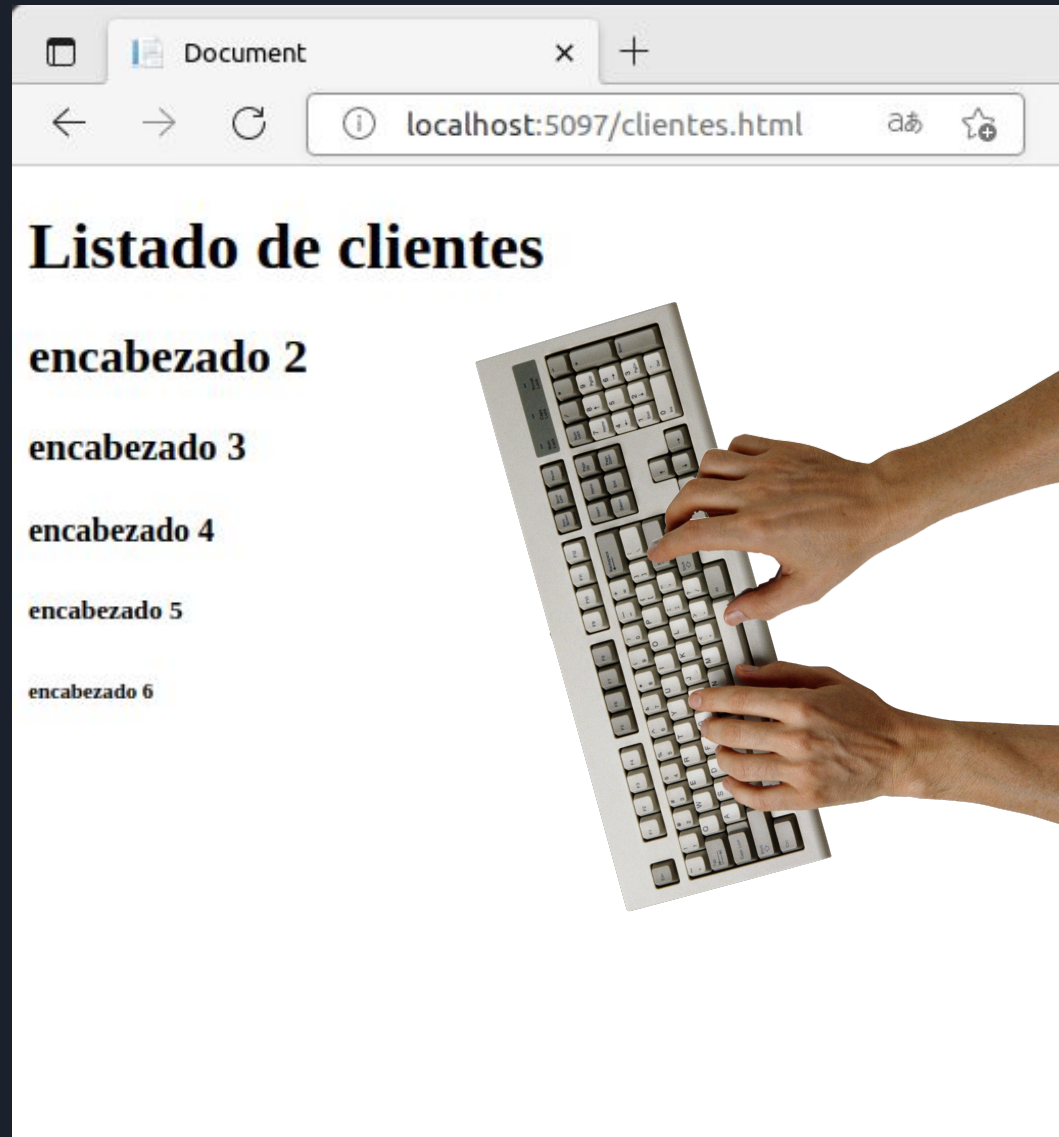
```
  <h3>encabezado 3</h3>
```

```
  <h4>encabezado 4</h4>
```

```
  <h5>encabezado 5</h5>
```

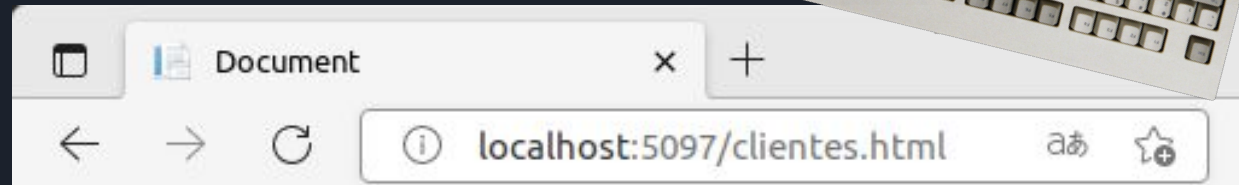
```
  <h6>encabezado 6</h6>
```

```
</body>
```



Probar el siguiente código

```
<body>  
  <h1> Listado de clientes </h1>  
  <p> Esto es un párafo,  
    otro renglón </p>  
</body>
```

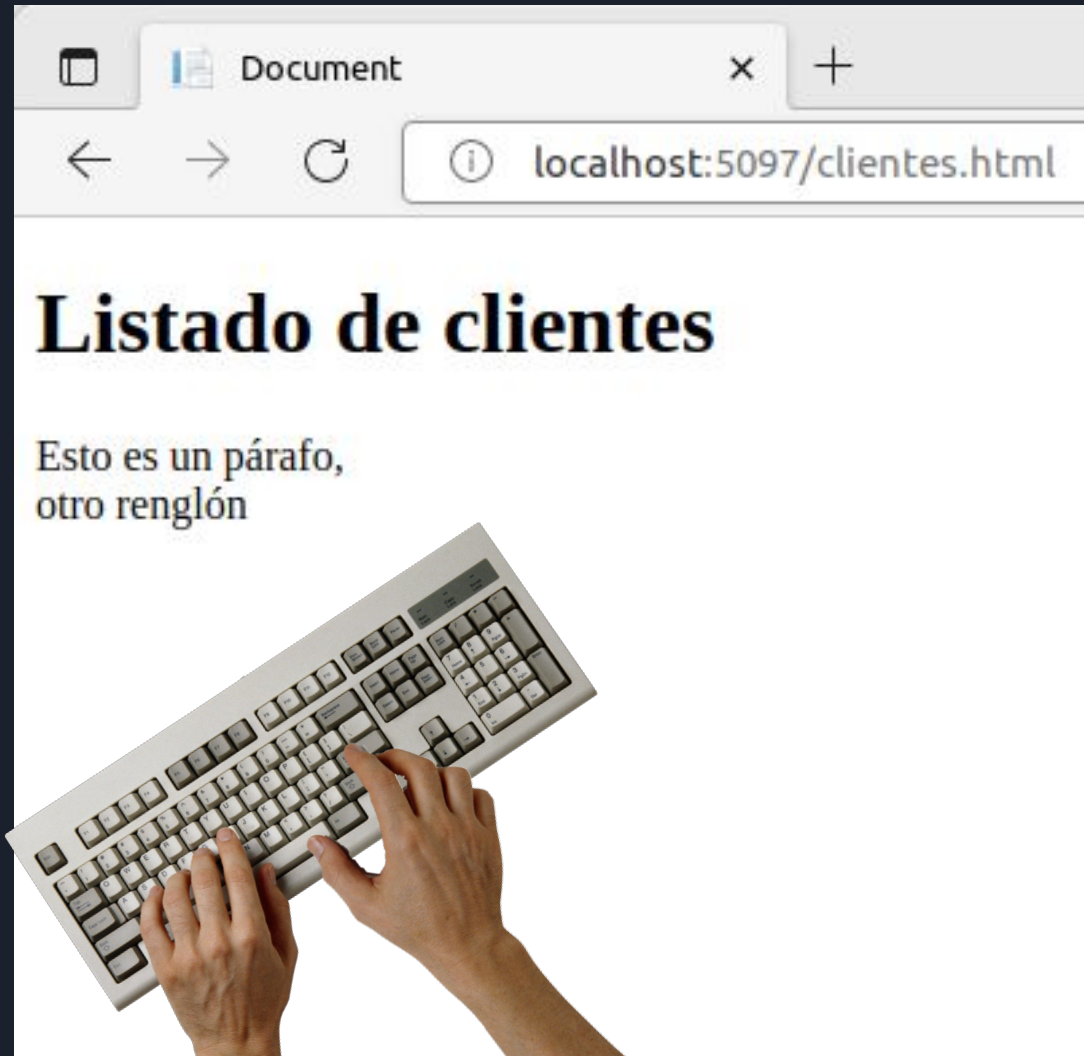


Se ignoran los fines de línea

Probar el siguiente código

```
<body>  
  <h1> Listado de clientes </h1>  
  <p> Esto es un párafo, <br>  
    otro renglón </p>  
</body>
```

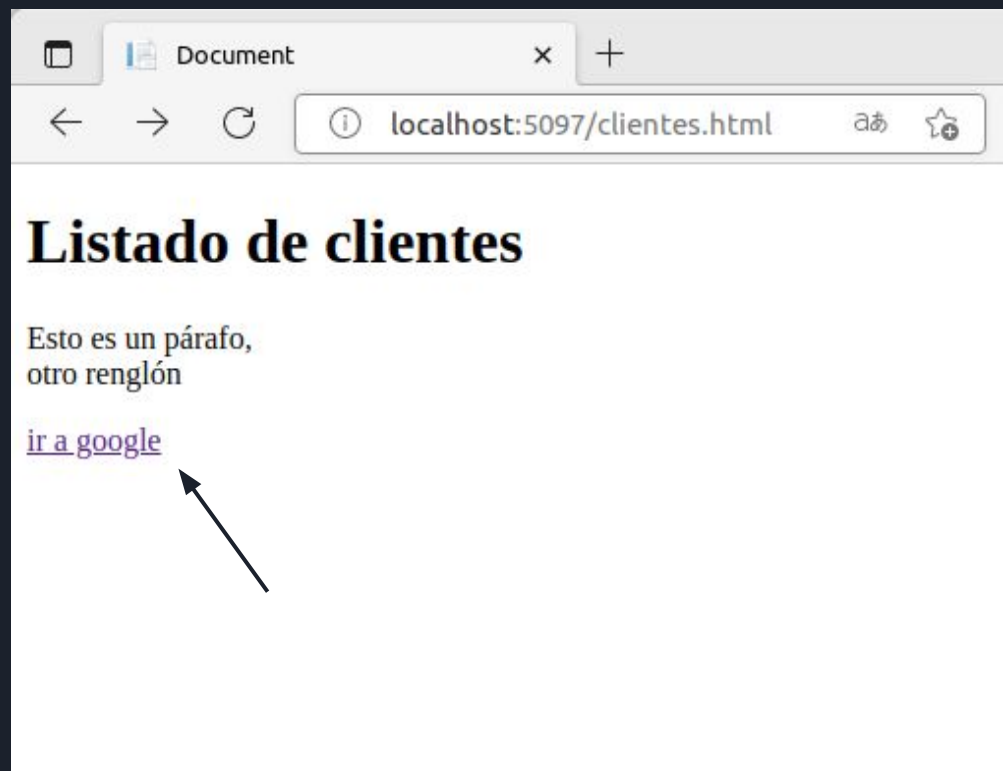
Agregar la etiqueta
de fin de línea



Probar el siguiente código

```
<body>  
  <h1> Listado de clientes </h1>  
  <p> Esto es un párafo,<br>  
    otro renglón </p>  
  <a href="http://www.google.com">ir a google</a>  
</body>
```

La etiqueta `<a>` se
utiliza para colocar
un link





Atributos HTML

- Los atributos HTML proporcionan información adicional sobre los elementos HTML.
- Todos los elementos HTML pueden tener atributos
- Los atributos siempre se especifican en la etiqueta de inicio
- Los atributos generalmente vienen en pares de la forma: `nombre="valor"`

Atributos HTML

- La etiqueta `` se utiliza para incrustar una imagen en una página HTML. El atributo `src` especifica la ruta a la imagen que se mostrará. También puede contener los atributos `width` y `height`, que especifican el ancho y el alto de la imagen (en píxeles). Ejemplo:

```

```

- Siempre se debe incluir el atributo `lang` en la etiqueta `<html>`, para declarar el idioma de la página web. Esto está destinado a ayudar a los motores de búsqueda y navegadores. Ejemplo:

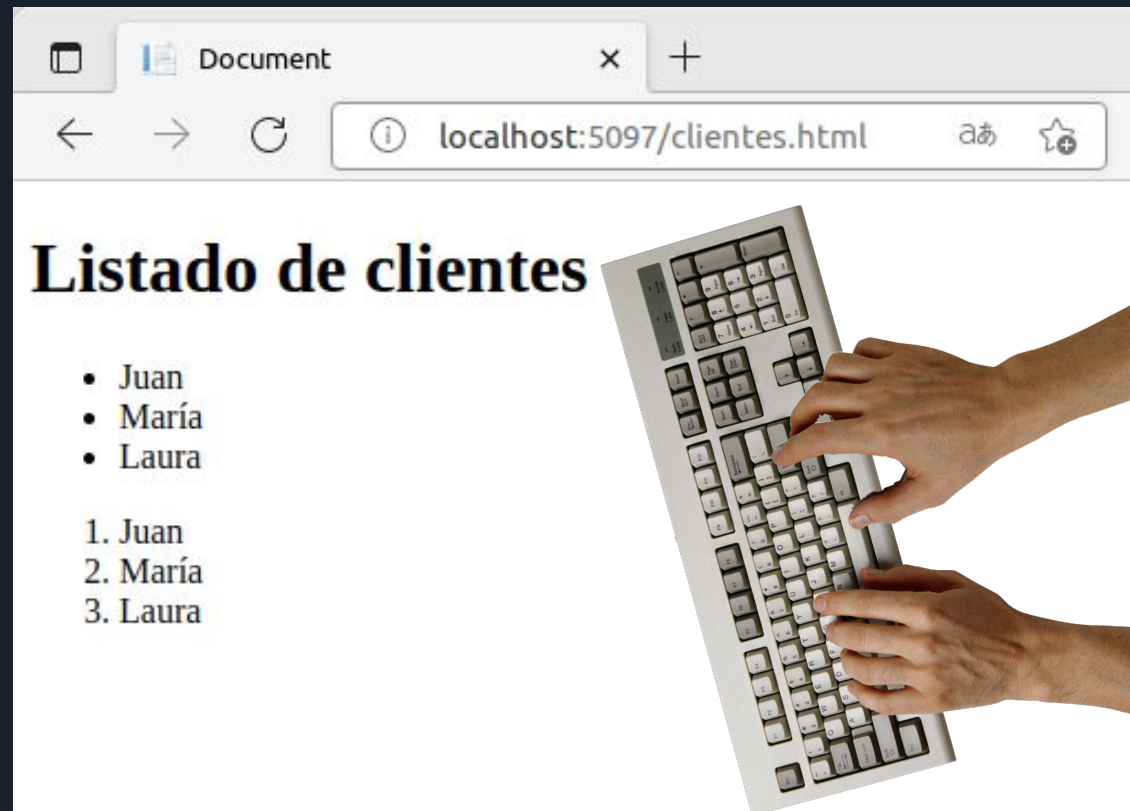
```
<html lang="es">
```

Probar el siguiente código

```
<body>
  <h1> Listado de clientes </h1>
  <ul>
    <li title="El primero">Juan</li>
    <li>María</li>
    <li>Laura</li>
  </ul>
  <ol>
    <li>Juan</li>
    <li>María</li>
    <li>Laura</li>
  </ol>
</body>
```

Copiar el código del archivo
11_RecursosParaLaTeoria

`` se usa para especificar un ítem dentro de una lista
¿Cual es la diferencia entre `` y ``?
¿Para qué sirve el atributo `title`?



Etiquetas <div> y

- <div>

Elemento utilizado para agrupar otros elementos HTML. Es un **elemento a nivel bloque**, por lo tanto, el navegador por defecto mostrará un salto de línea antes y después de él

-

Elemento utilizado para agrupar elementos de texto. Es un **elemento a nivel línea**, por lo tanto, el navegador por defecto NO mostrará un salto de línea antes y después de él

Probar el siguiente código

```
<head>
  . . .
  <style>
    #encabezado {
      background-color: green;
      font-size: xx-large;
    }
    #blanco {
      color: white;
    }
  </style>
</head>
<body>
  <div id="encabezado">
    <p>primer párrafo</p>
    <p>segundo párrafo</p>
    <span id="blanco">este texto es blanco</span> pero este no
  </div>
  <p>esto está fuera del encabezado</p>
</body>

</html>
```

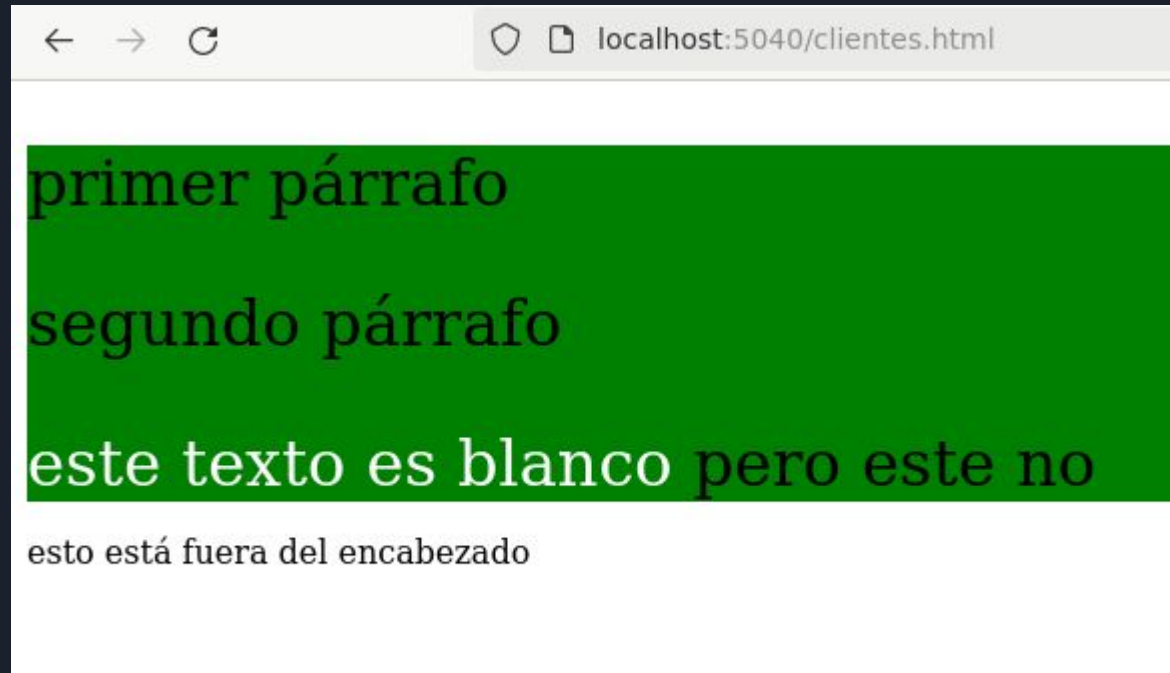
Estilos CSS que se aplican a los
elementos con atributo
`id="encabezado"` y `id="rojo"`



Probar el siguiente código

```
<head>
  . . .
  <style>
    #encabezado {
      background-color: green;
      font-size: xx-large;
    }
    #blanco {
      color: white;
    }
  </style>
</head>
<body>
  <div id="encabezado">
    <p>primer párrafo</p>
    <p>segundo párrafo</p>
    <span id="blanco">este texto es blanco</span> pero este no
  </div>
  <p>esto está fuera del encabezado</p>
</body>

</html>
```

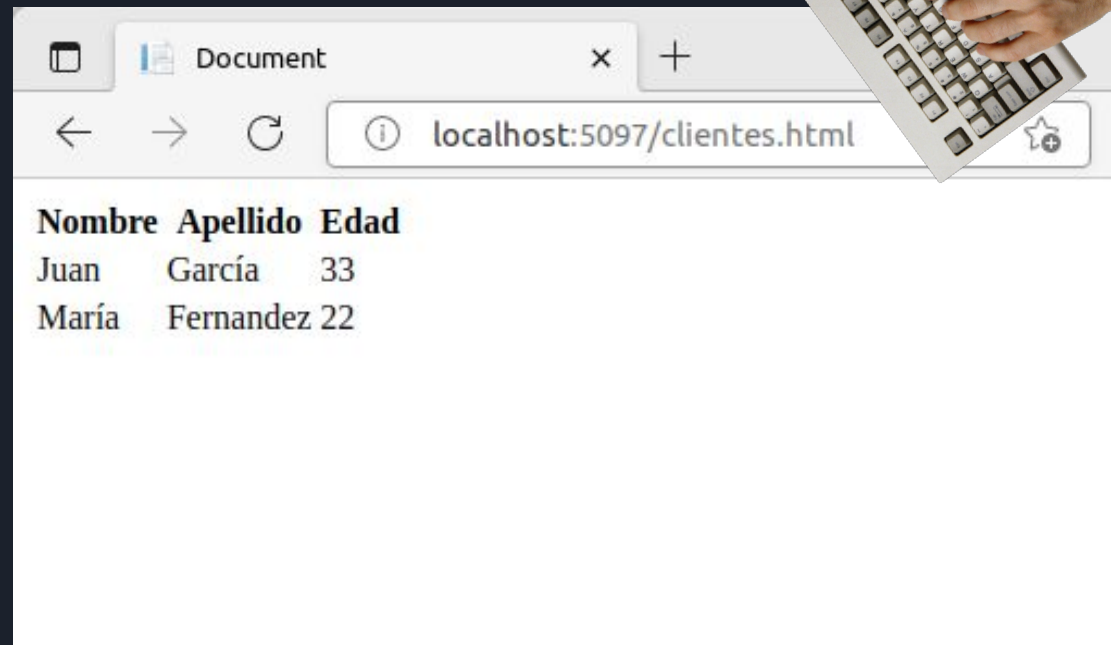


Tablas

- La etiqueta `<table>` se utiliza colocar una tabla
- Las tablas están conformadas por filas demarcadas con etiquetas `<tr>` (table row)
- Cada fila tendrá una cierta cantidad de celdas demarcadas con etiquetas `<td>` (table data)
- Usualmente las celdas de la primera fila se indican con etiquetas `<th>` (table header)

Probar el siguiente código

```
<body>
  <h1> Listado de clientes </h1>
  <table>
    <tr>
      <th>Nombre</th>
      <th>Apellido</th>
      <th>Edad</th>
    </tr>
    <tr>
      <td>Juan</td>
      <td>García</td>
      <td>33</td>
    </tr>
    <tr>
      <td>María</td>
      <td>Fernandez</td>
      <td>22</td>
    </tr>
  </table>
</body>
```



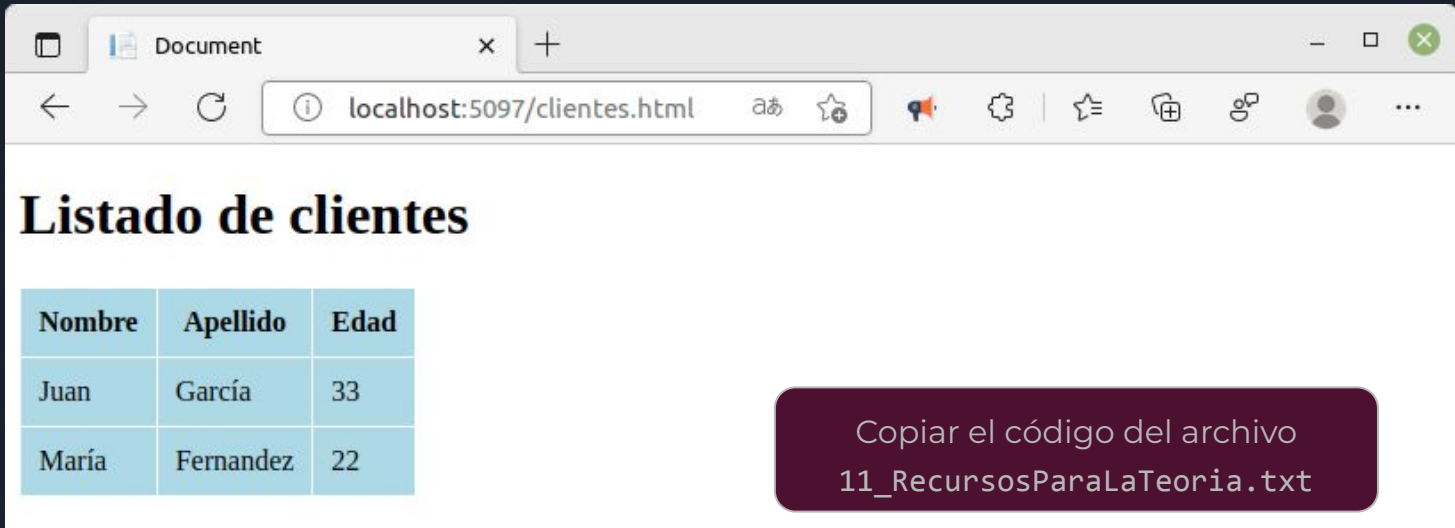
Copiar el código del archivo
11_RecursosParaLaTeoria.txt

Probar el siguiente código

```
<style>
  table, th, td {
    border: 1px solid white;
    border-collapse: collapse;
    padding: 10px;
  }
  th, td {
    background-color: lightblue;
  }
</style>
```

Estilos CSS que se aplican a todos los elementos `table`, `th` y `td`

Estilos CSS que se aplican a todos los elementos `th` y `td`



The screenshot shows a web browser window with the title 'Document' and the address bar displaying 'localhost:5097/clientes.html'. The page content features a heading 'Listado de clientes' followed by a table with three columns: 'Nombre', 'Apellido', and 'Edad'. The table contains two rows of data. A dark purple button is located at the bottom right of the page.

Nombre	Apellido	Edad
Juan	García	33
María	Fernandez	22

Copiar el código del archivo
11_RecursosParaLaTeoria.txt

Componentes Blazor

The screenshot shows the Visual Studio IDE with the following components:

- EXPLORER:** Displays the project structure. Under **SOLUTION EXPLORER**, the **HolaBlazor** project is expanded, showing **Components** > **Pages**. The file **Hola.razor** is highlighted in the list.
- CLIENTES.HTML:** The active file in the editor, showing the content of **Hola.razor**:

```
1 @page "/hola"
2
3 <h1>Hola Mundo</h1>
4
5
```
- DEBUG CONSOLE:** Shows the output of the application, including the message "Application started. Press Ctrl+C to shut down." and "Microsoft.Hosting.Lifetime: Information: Application started. Press Ctrl+C to shut down."

A yellow box highlights the code in **Hola.razor**, and a white arrow points from the **Hola.razor** file in the Explorer to the code in the editor.

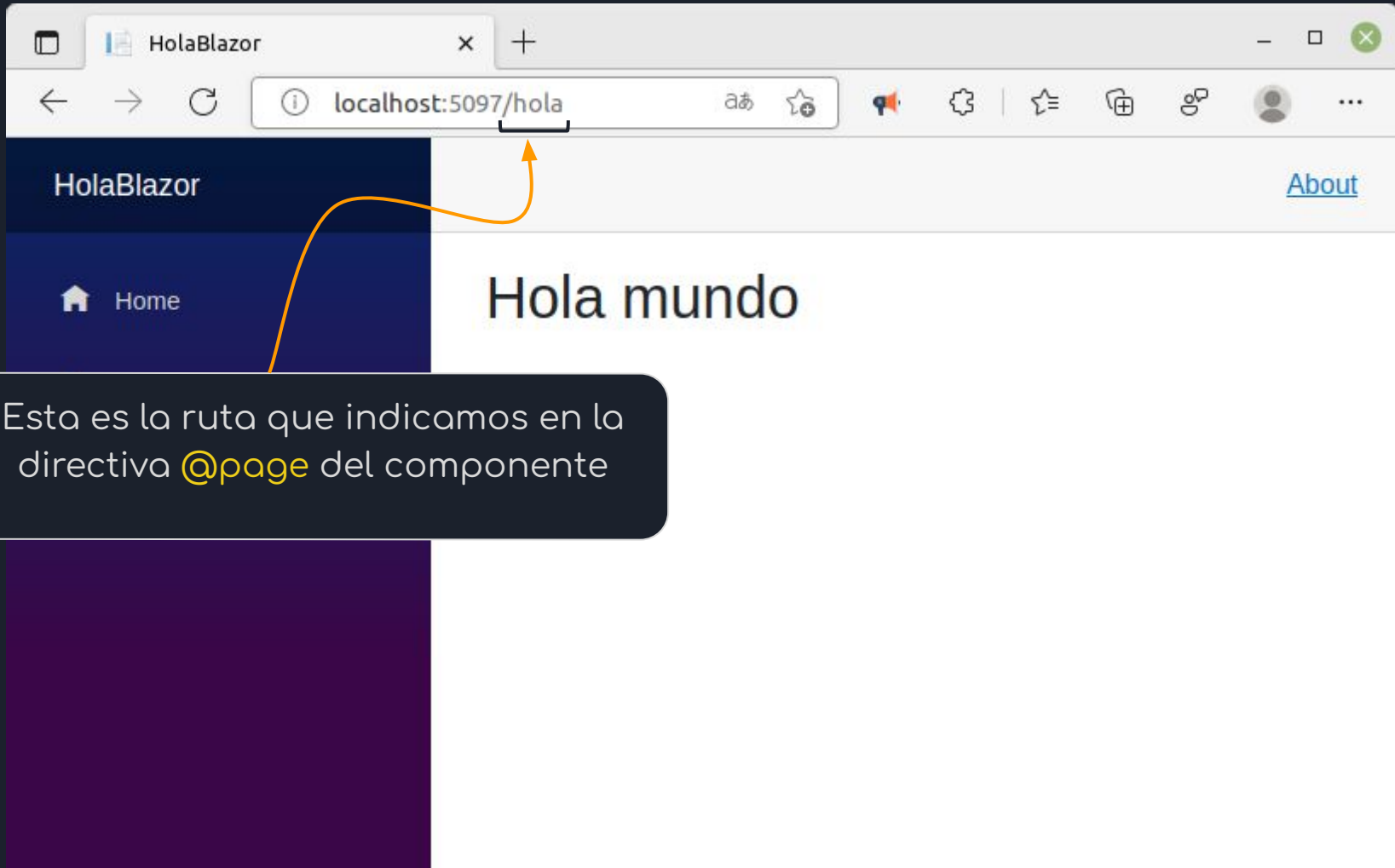
Crear el archivo **Hola.razor** en la carpeta **Components/Pages** con este contenido

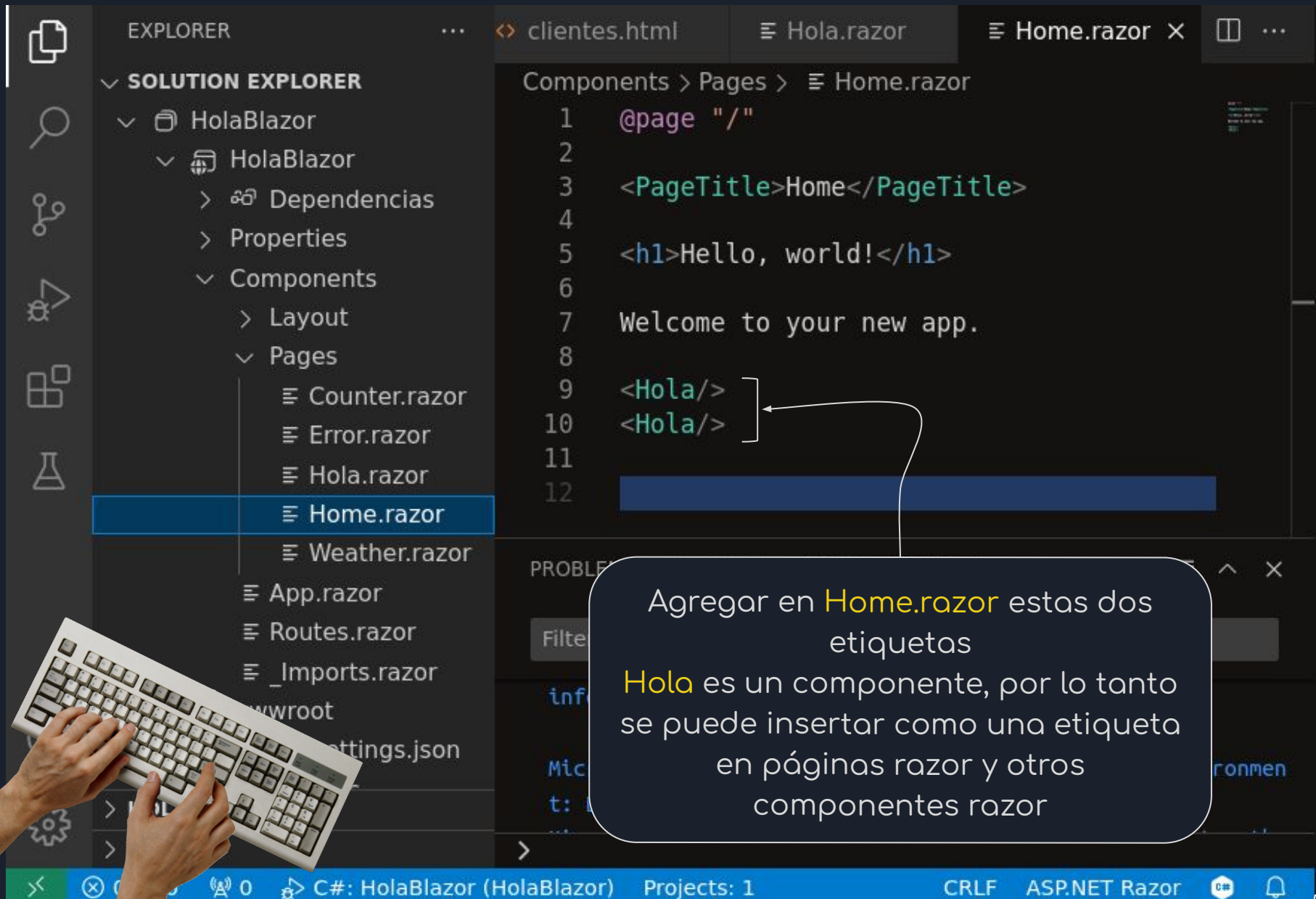
Por convención en esta carpeta se colocan los componentes "ruteables"

Application started. Press Ctrl+C to shut down.
Microsoft.Hosting.Lifetime: Information: Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
Hosting environment: Development



Detener la aplicación y ejecutarla nuevamente.
Acceder desde el navegador a /hola





EXPLORER

SOLUTION EXPLORER

- HolaBlazor
 - HolaBlazor
 - Dependencies
 - Properties
 - Components
 - Layout
 - Pages
 - Counter.razor
 - Error.razor
 - Hola.razor
 - Home.razor**
 - Weather.razor
 - App.razor
 - Routes.razor
 - _Imports.razor
 - wwwroot
 - settings.json

clientes.html

Hola.razor

Home.razor

Components > Pages > Home.razor

```
1 @page "/"
2
3 <PageTitle>Home</PageTitle>
4
5 <h1>Hello, world!</h1>
6
7 Welcome to your new app.
8
9 <Holo/>
10 <Holo/>
11
12
```

PROBLEMS

Filter

inf

Mic

t:

ronmen

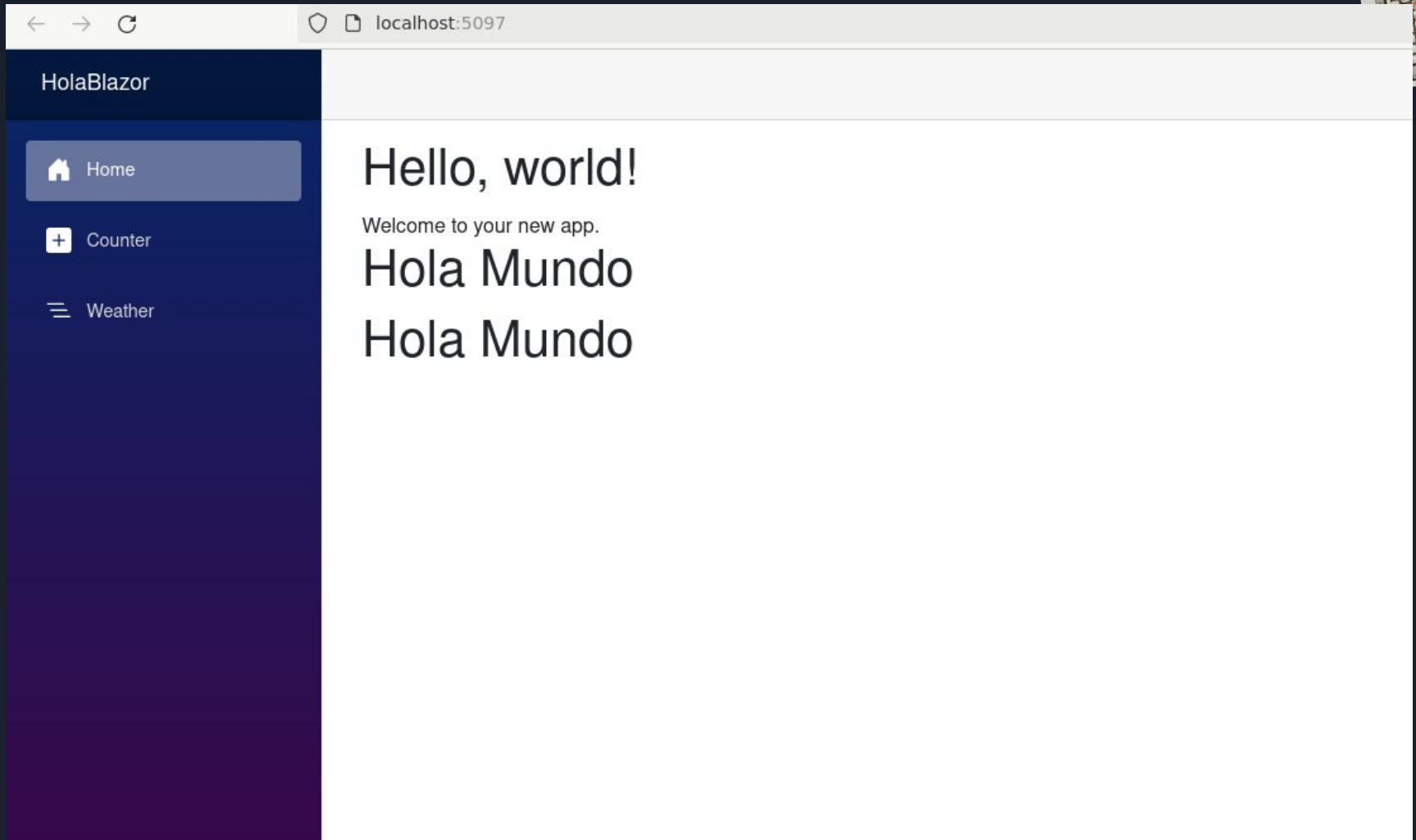
C#: HolaBlazor (HolaBlazor) Projects: 1 CRLF ASP.NET Razor

Agregar en **Home.razor** estas dos etiquetas

Holo es un componente, por lo tanto se puede insertar como una etiqueta en páginas razor y otros componentes razor



Detener la aplicación y ejecutarla nuevamente





Modificar Hola.razor de la siguiente manera y volver a ejecutar



```
@page "/hola"
```

```
<h1>Hola @nombre</h1>
```

```
@code{
```

```
    string nombre="Juan";
```

```
}
```

Esta es la sección de la vista.
Utilizamos la @ para cambiar de
código HTML a C#, en este caso
para acceder a la variable
nombre

Sección de código C#, en este caso
sólo estamos definiendo la variable
nombre

La sección se identifica con

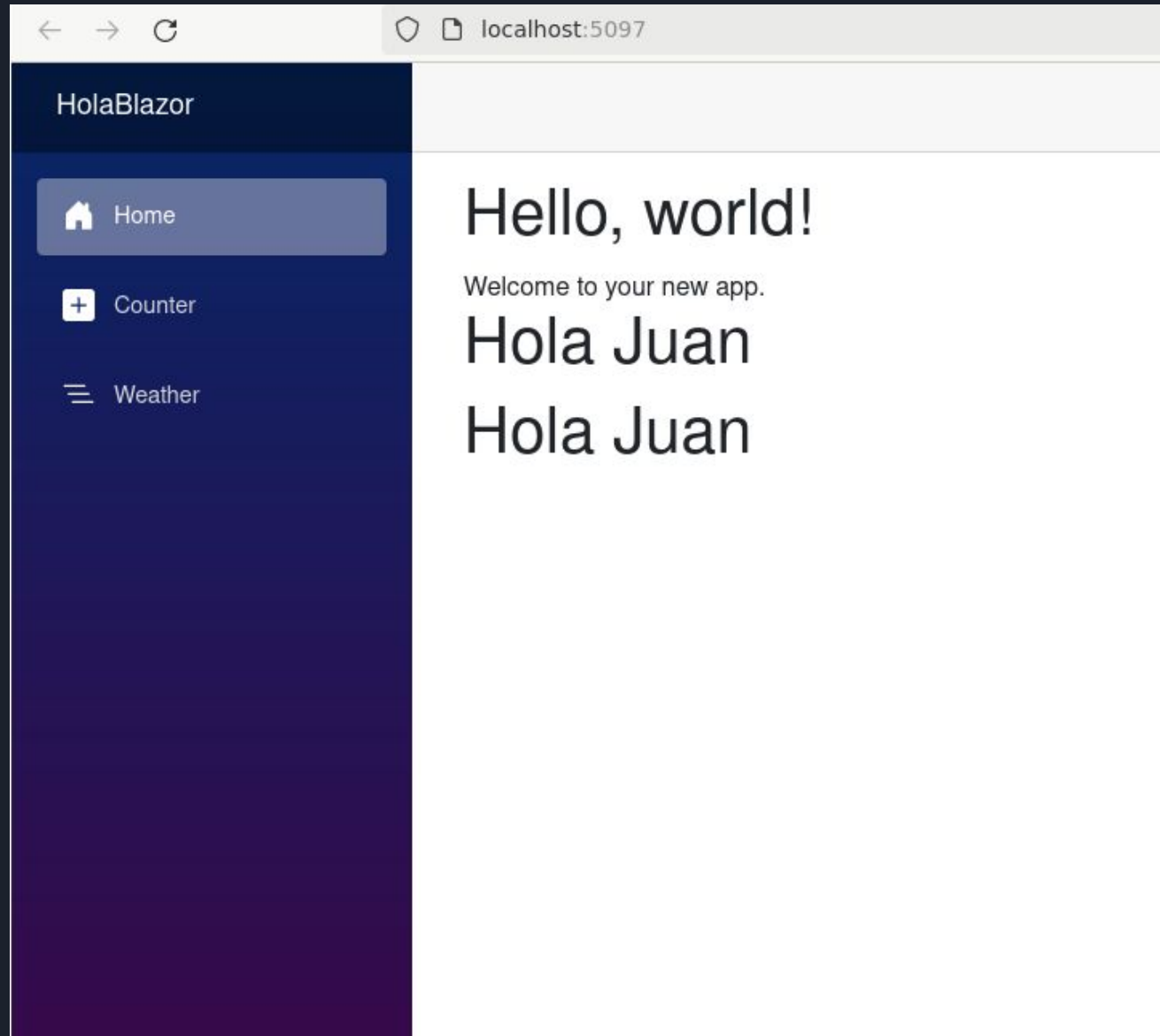
@code{...}

HTML

```
@page "/hola"
```

```
<h1>Hola @nombre</h1>
```

```
@code{  
    string nombre="Juan";  
}
```



Si colocamos un breakpoint, podemos ver en ejecución que la clase C# que se crea a partir del archivo **.razor** coincide con el nombre del archivo y su namespace se determina por la carpeta en la que está incluido

Now listening on: <http://localhost:5097>
Microsoft.Hosting.Lifetime: Information: Now listening on: <http://localhost:5097>

Secuencia de escape para @

Si fuese necesario escribir la @ en un componente razor se debe utilizar @@ por ejemplo:

@@Username

Se visualizará @Username

De lo contrario se esperaría que Username fuese un campo o una propiedad definida en el componente





Modificar Hola.razor de la siguiente manera y volver a ejecutar



```
@page "/hola"
```

```
<h1>Hola @nombre.ToUpper()</h1>
```

```
<p>El doble de 5 es @(5*2) y la Sumatoria  
de 1 a 10 es @Sumatoria(10) </p>
```

```
@code {  
    string nombre = "Juan";  
    int Sumatoria(int n) =>  
        Enumerable.Range(1, n).Sum();  
}
```

Copiar el código del archivo
11_RecursosParaLaTeoria.txt

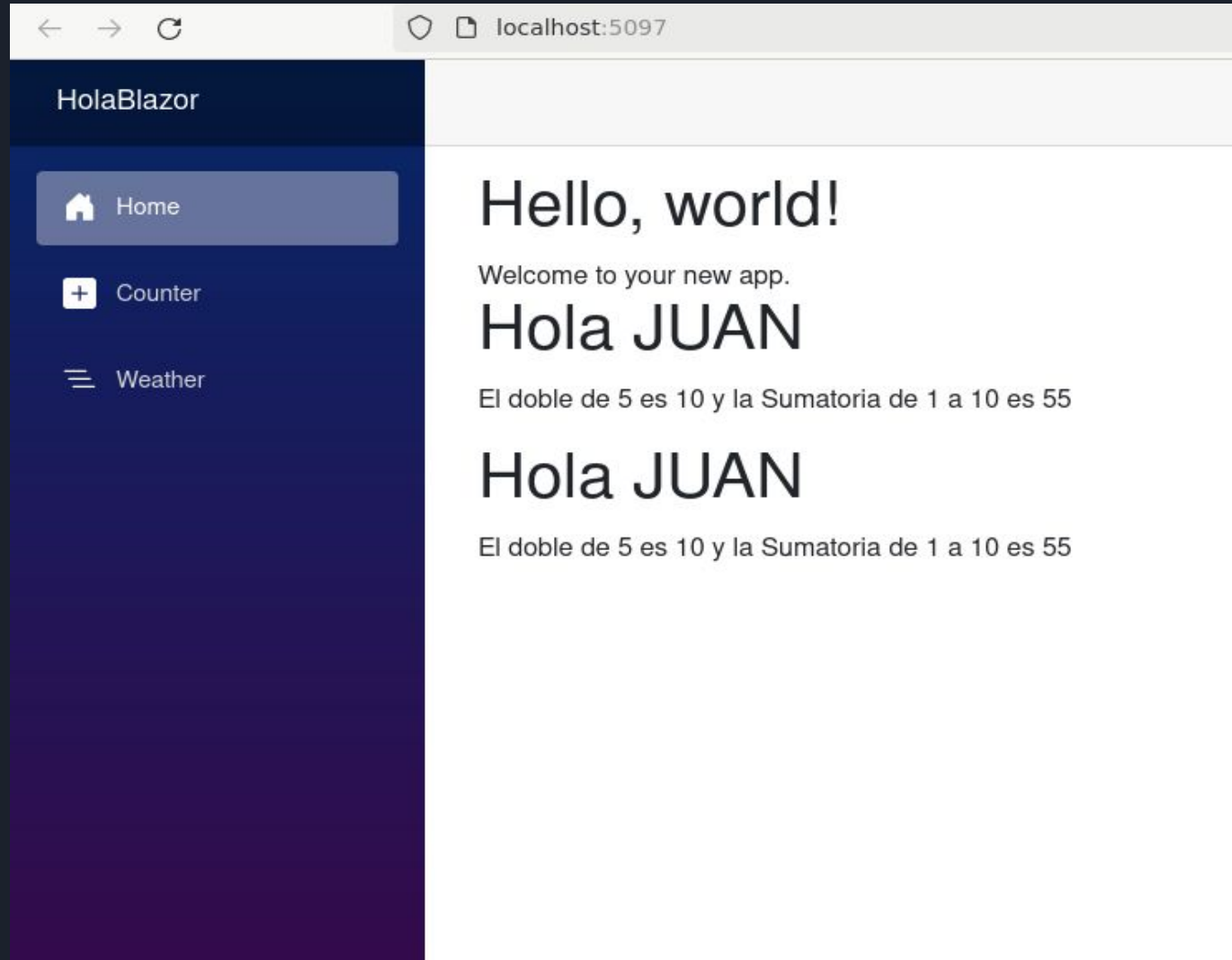
HTML

```
@page "/hola"
```

```
<h1>Hola @nombre.ToUpper()</h1>
```

```
<p>El doble de 5 es @(5*2) y la Sumatoria  
de 1 a 10 es @Sumatoria(10) </p>
```

```
@code {  
    string nombre = "Juan";  
    int Sumatoria(int n) =>  
        Enumerable.Range(1, n).Sum();  
}
```



Expresiones implícitas y explícitas en Razor

Las **expresiones implícitas Razor** comienzan por @ seguida de código de C#. Generalmente no admiten espacios y no se indica dónde terminan, se trata de expresiones simples, por ejemplo

`@nombre.ToUpper()`

En algunas expresiones es necesario indicar cuál es el comienzo y fin de la misma, se llaman **expresiones explícitas** y se denotan entre paréntesis, por ejemplo:

`@(5*2)`

De lo contrario, `@5*2` provocaría error de compilación



Modificar los componentes Home.razor y Hola.razor y volver a ejecutar



```
----- Home.razor -----
```

```
@page "/"  
<Hola/>
```

```
----- Hola.razor -----
```

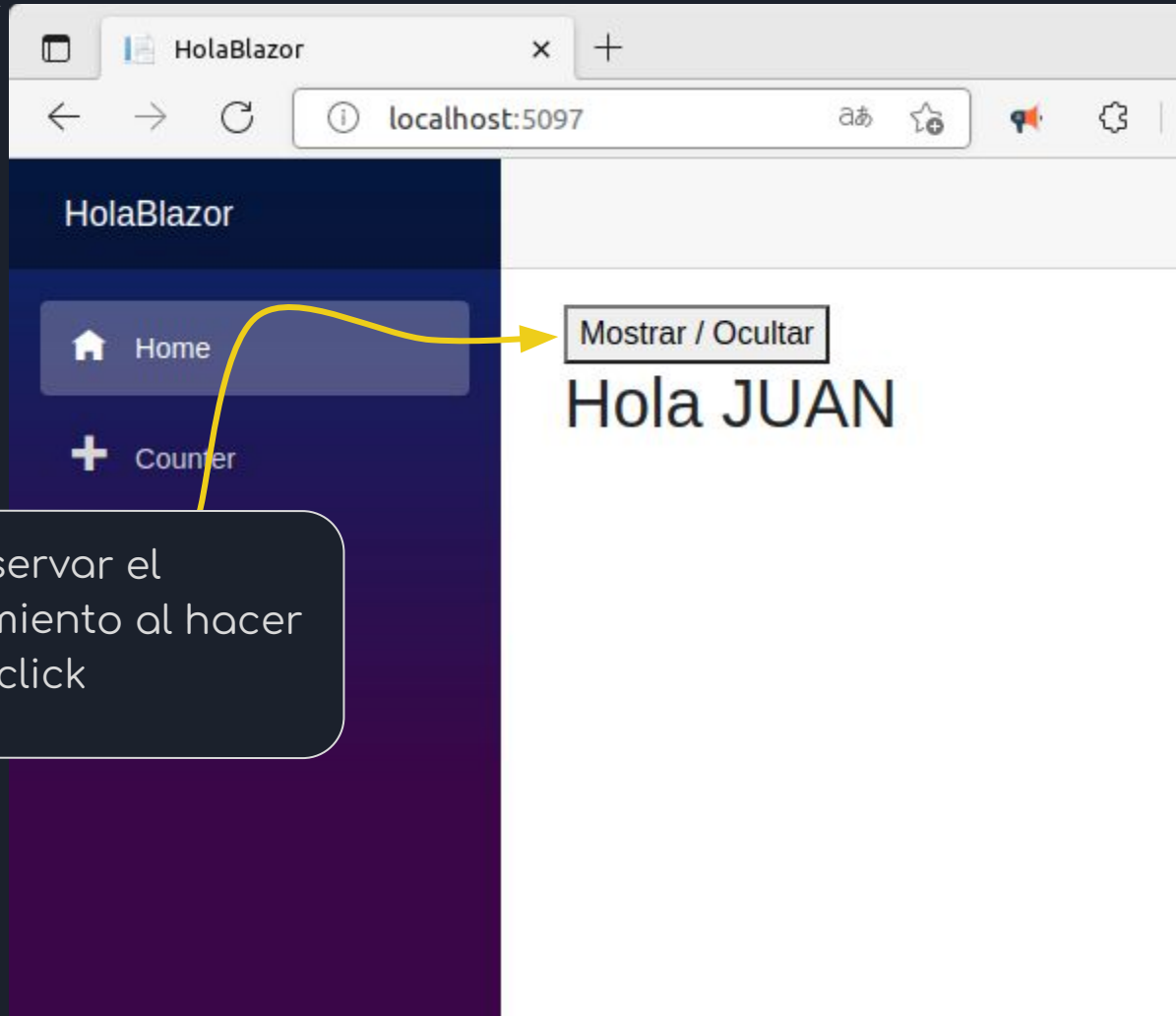
```
@page "/hola"  
@rendermode InteractiveServer  
<button @onclick="Cambiar">Mostrar / Ocultar</button>  
@if (EsVisible)  
{  
    <h1>Hola @nombre.ToUpper()</h1>  
}  
@code {  
    string nombre = "Juan";  
    bool EsVisible = true;  
    void Cambiar() {  
        EsVisible = !EsVisible;  
    }  
}
```

Es necesario si queremos que el componente sea interactivo

Copiar el código del archivo
11_RecursosParaLaTeoria.txt

HTML

```
@page "/hola"
@rendermode InteractiveServer
<button @onclick="Cambiar">Mostrar / Ocultar</button>
@if (EsVisible)
{
    <h1>Hola @nombre.ToUpper()</h1>
}
@code {
    string nombre = "Juan";
    bool EsVisible = true;
    void Cambiar() {
        EsVisible = !EsVisible;
    }
}
```



Observar el
comportamiento al hacer
click

EXPLORER

SOLUTION EXPLORER

- ✓ HolaBlazor
 - ✓ HolaBlazor
 - > Dependencias
 - > Properties
 - > Components
 - ✓ Entidades
 - Persona.cs
 - > wwwroot
 - > {} appsettings.json
 - Program.cs

Persona.cs

```
Entidades > Persona.cs > ...
1 namespace HolaBlazor.Entidades;
2 class Persona
3 {
4     public string Nombre { get; set; } = "";
5     public string Apellido { get; set; } = "";
6     public int? Edad { get; set; } //podría faltar el dato de la edad
7
8     // vamos a hardcodear una lista de personas
9     // que usaremos en los siguientes ejemplos
10    // para ello definimos el siguiente método estático
11    public static List<Persona> GetLista()
12    {
13        return new List<Persona>() {
14            new Persona() {Nombre="Pablo",Apellido="Perez", Edad=34},
15            new Persona() {Nombre="Laura",Apellido="García", Edad=30},
16            new Persona() {Nombre="José",Apellido="Lopez", Edad=45},
17            new Persona() {Nombre="Ana",Apellido="Colombo", Edad=21},
18            new Persona() {Nombre="María",Apellido="Suarez", Edad=15},
19        };
20    }
21 }
22
23
24
```

Definirla en el namespace `HolaBlazor.Entidades`

Crear la carpeta `Entidades` y dentro de ella la clase `Persona`

Copiar el código del archivo `11_RecursosParaLaTeoria.txt`



Modificar el componente Hola.razor y volver a ejecutar



```
@page "/hola"
```

```
@using HolaBlazor.Entidades
```

```
<h1>Listado de personas</h1>
```

```
<ul>
```

```
@foreach (var p in lista)
```

```
{
```

```
    <li>@p.Apellido, @p.Nombre (@p.Edad)</li>
```

```
}
```

```
</ul>
```

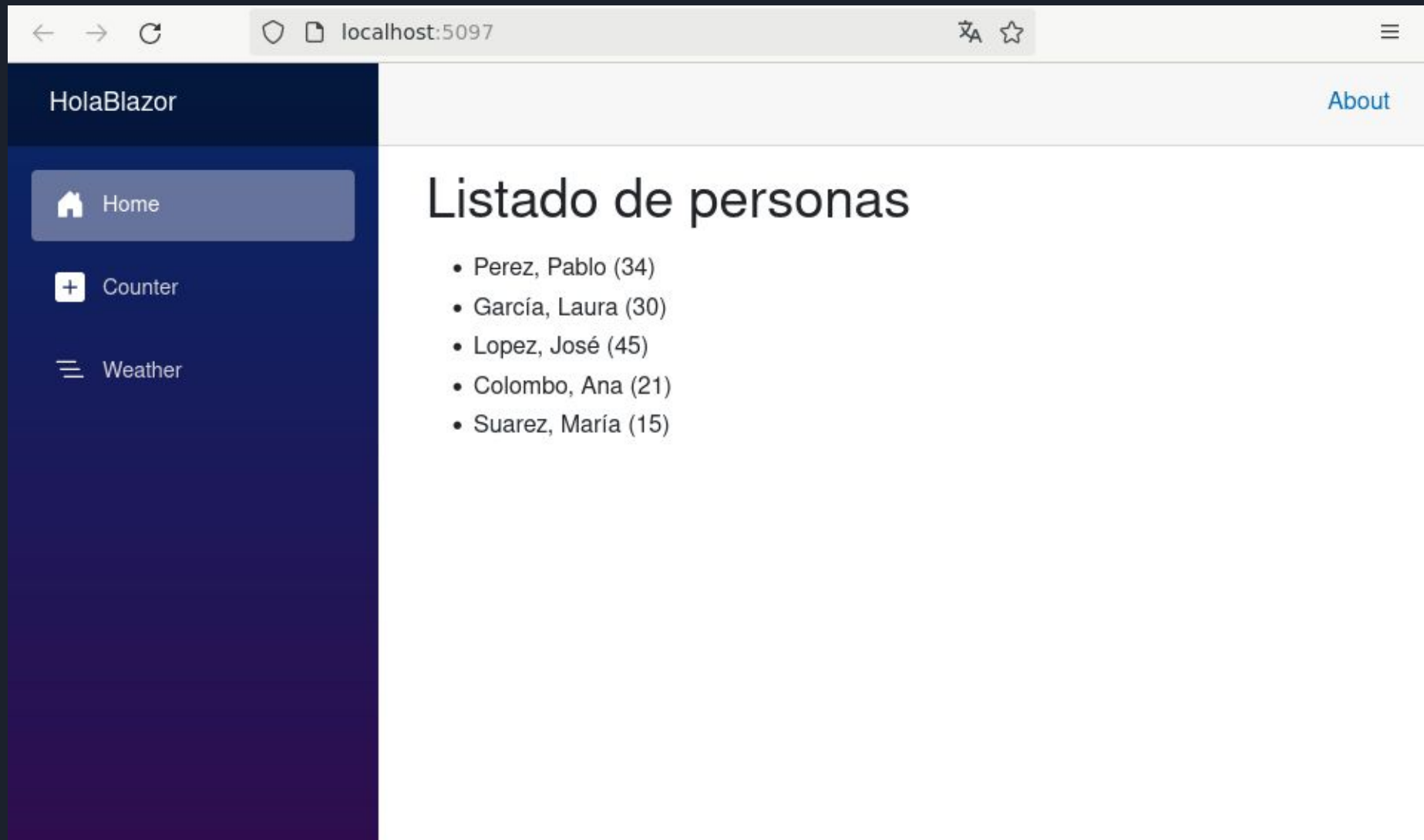
```
@code {
```

```
    List<Persona> lista = Persona.GetLista();
```

```
}
```

Directiva `@using`
Observar que no es necesario
el punto y coma (;) final

Copiar el código del archivo
11_RecursosParaLaTeoria.txt



The image shows a Visual Studio interface with the following components:

- EXPLORER:** Displays the project structure. Under 'Components' > 'Pages', the file '_Imports.razor' is selected. An arrow points from this file to a text box.
- Code Editor:** Shows the content of '_Imports.razor'. The file contains a list of @using directives. Line 11, '@using HolaBlazor.Entidades', is highlighted in blue. An arrow points from this line to the same text box.
- Text Box:** Contains the instruction: 'Agregar en _Imports.razor la directiva `@using HolaBlazor.Entidades` para que se aplique a todos los componentes razor de la aplicación (ahora se puede borrar la directiva using agregada en Hola.razor)'. The code snippet is highlighted in green.
- Bottom Bar:** Shows the active solution 'C#: HolaBlazor (HolaBlazor)', 'Projects: 1', and encoding 'UTF-8 with BOM'. The file type is 'ASP.NET Razor'.

```
1 @using System.Net.Http
2 @using System.Net.Http.Json
3 @using Microsoft.AspNetCore.Components.Forms
4 @using Microsoft.AspNetCore.Components.Routing
5 @using Microsoft.AspNetCore.Components.Web
6 @using static Microsoft.AspNetCore.Components.Web.RenderMode
7 @using Microsoft.AspNetCore.Components.Web.Virtualization
8 @using Microsoft.JSInterop
9 @using HolaBlazor
10 @using HolaBlazor.Components
11 @using HolaBlazor.Entidades
12
```




Modificar el componente Hola.razor y volver a ejecutar



```
@page "/hola"
@rendermode InteractiveServer

<h1>Listado de personas</h1>
<ul>
    @foreach (var p in lista)
    {
        <li>@p.Apellido, @p.Nombre (@p.Edad)</li>
    }
</ul>
<button @onclick="Agregar">Agregar a Carlos</button>

@code {
    List<Persona> lista = Persona.GetLista();
    void Agregar() => lista.Add(new Persona() {
        Nombre = "Carlos",
        Apellido = "Maldini",
        Edad = 66
    });
}
```

Copiar el código del archivo
11_RecursosParaLaTeoria.txt

HolaBlazor

About

🏠 Home

⛶ Counter

☰ Weather

Listado de personas

- Perez, Pablo (34)
- García, Laura (30)
- Lopez, José (45)
- Colombo, Ana (21)
- Suarez, María (15)

Agregar a Carlos

Cada vez que se hace click se agrega Carlos a la lista



Modificar el componente Hola.razor y volver a ejecutar



```
@page "/hola"
@rendermode InteractiveServer

<h1>Listado de personas</h1>
<ul>
    @foreach (var p in lista)
    {
        <li>@p.Apellido, @p.Nombre (@p.Edad)</li>
    }
</ul>
<input placeholder="Nombre" @bind="p.Nombre" /><br>
<input placeholder="Apellido" @bind="p.Apellido" /><br>
<input type="number" placeholder="Edad" @bind="p.Edad" /><br>
<button @onclick="Agregar">Agregar</button>

@code {
    List<Persona> lista = Persona.GetLista();
    Persona p = new Persona();
    void Agregar()
    {
        lista.Add(p);
        p = new Persona();
    }
}
```

Copiar el código del archivo
11_RecursosParaLaTeoria.txt

HolaBlazor

About

🏠 Home

+ Counter

☰ Weather

Listado de personas

- Perez, Pablo (34)
- García, Laura (30)
- Lopez, José (45)
- Colombo, Ana (21)
- Suarez, María (15)



Agregar

Ahora se puede editar el
nombre, apellido y edad
de la nueva persona



```

. . .
</ul>
<input @ref="input_01" placeholder="Nombre" @bind="p.Nombre" /><br>
<input placeholder="Apellido" @bind="p.Apellido" /><br>
<input type="number" placeholder="Edad" @bind="p.Edad" /><br>
<button @onclick="Agregar">Agregar</button>

```

```

@code {
    List<Persona> lista = Persona.GetLista();
    ElementReference input_01;
    Persona p = new Persona();
    void Agregar()
    {
        lista.Add(p);
        p = new Persona();
        input_01.FocusAsync();
    }
}

```

Con estos cambios, luego de agregar una nueva persona a la lista se establece el foco en el primer input para poder ingresar el nombre de la próxima persona a agregar



Modificar el componente Hola.razor



```
...  
@if (!SoloLectura)  
{  
    <input @ref="input_01" placeholder="Nombre" @bind="p.Nombre" /><br>  
    <input placeholder="Apellido" @bind="p.Apellido" /><br>  
    <input type="number" placeholder="Edad" @bind="p.Edad" /><br>  
    <button @onclick="Agregar">Agregar</button>  
}
```

Mostramos estos
elementos sólo si la
propiedad
SoloLectura es false

```
@code {  
    [Parameter]  
    public bool SoloLectura { get; set; } = false;  
    List<Persona> lista = Persona.GetLista();  
    ElementReference input_01;  
    Persona p = new Persona();  
    void Agregar()  
    {  
        lista.Add(p);  
        p = new Persona();  
        input_01.FocusAsync();  
    }  
}
```

El atributo **[Parameter]**
aplicado a una propiedad
pública permite establecer su
valor en la etiqueta del
componente **<Hola>** cuando
sea utilizada



Modificar el componente Home.razor y ejecutar



```
@page "/"  
<Hola SoloLectura="true"/>  
<Hola />
```

←

→

↺

localhost:5097

🔍

☆

☰

HolaBlazor

🏠 Home

+ Counter

☰ Weather

About

Listado de personas

- Perez, Pablo (34)
- García, Laura (30)
- Lopez, José (45)
- Colombo, Ana (21)
- Suarez, María (15)

Listado de personas

- Perez, Pablo (34)
- García, Laura (30)
- Lopez, José (45)
- Colombo, Ana (21)
- Suarez, María (15)

Nombre

Apellido

Edad

Agregar

<Hola SoloLectura="true"/>

<Hola />

Fin teoría 11

No se proporcionan ejercicios prácticos sobre esta teoría.

El contenido de esta teoría será aplicado en la resolución del trabajo final del curso.