



# .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}");  
    }  
}
```

Copiar el código del archivo  
11\_RecursosParaLaTeoria



# Codificar la interfaz IServicioX y la clase ServicioX



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

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

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

```
namespace DiContainer;  
public class ServicioX (ILogger logger): IServicioX  
{  
    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



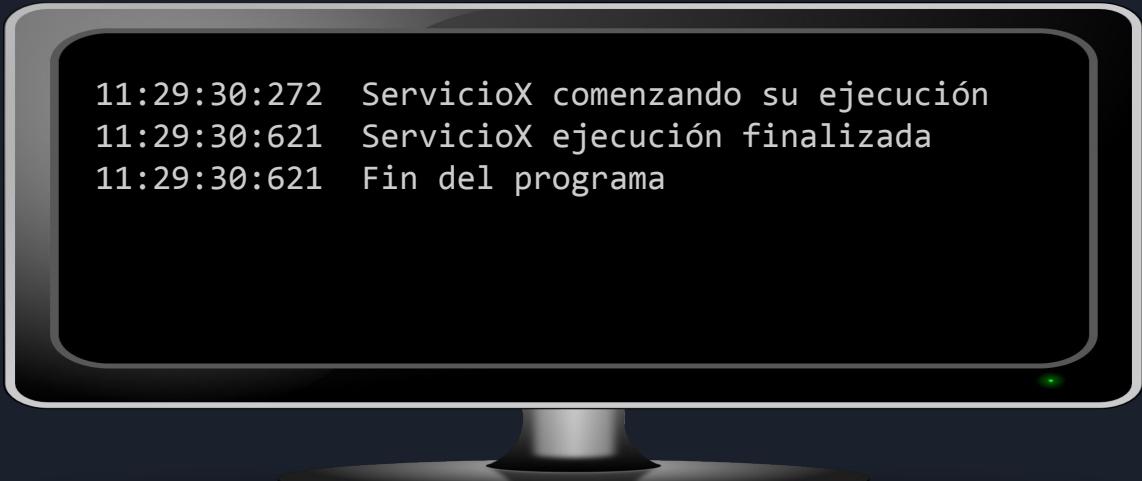
```
using DiContainer;
```

```
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

Se registran los servicios y se construye el proveedor

La clase  
ProveedorServicios  
ya no es necesaria

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.



## Nota sobre el patrón Builder

```
var servicios = new ServiceCollection();
. . . // aquí se configura el objeto que se construirá
var proveedor = servicios.BuildServiceProvider();
```

Aquí estamos utilizando el patrón **Builder** que permite construir un objeto complejo (en este caso un **ServiceProvider**) paso a paso, separando la construcción de un objeto complejo de su representación.

**ServiceCollection** actúa como el **builder**

## Nota sobre el patrón Builder

¿Cuál puede ser la ventaja de usar el patrón Builder?

¿Por qué no crear directamente el objeto que nos interesa construir y luego configurarlo por medio de métodos o propiedades suyas?



## Algunas ventajas de usar el patrón Builder

- **Inmutabilidad:** Una vez que el objeto final es construido, puede ser inmutable.
- **Validación:** El método `Build()` puede verificar que todas las configuraciones necesarias estén presentes y sean válidas antes de crear el objeto final.
- **Separación de Responsabilidades:** Separa el código de construcción del código de uso.





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}");
    }
}
```

Possible  
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

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

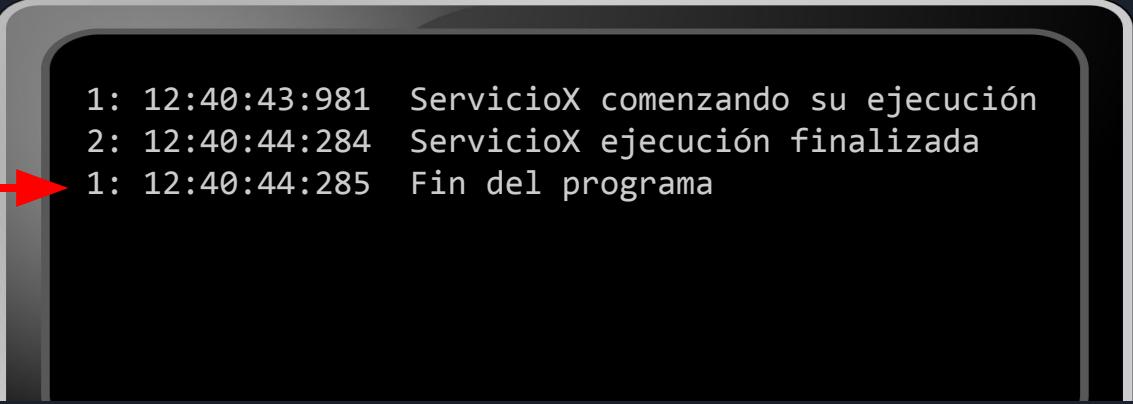


### Ejercicio práctico: Se requiere numerar las líneas de log



- 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



A terminal window showing log output. A red arrow points from the word "Mal" to the first line of the log. The log content is as follows:

```
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;

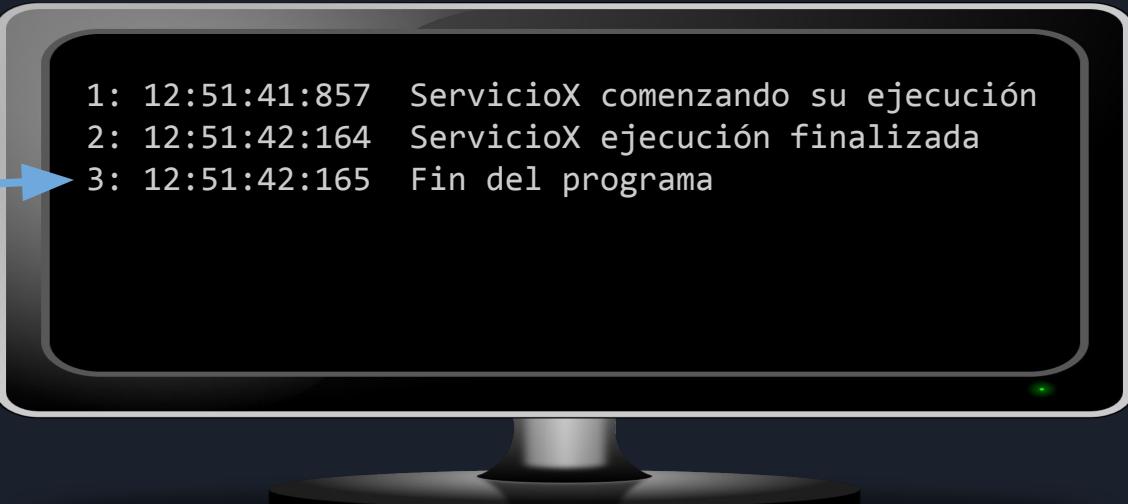
var servicios = new ServiceCollection();
servicios.AddSingleton<ILogger, LoggerNum>(); ←
servicios.AddTransient<IServicioX, ServicioX>();
var proveedor = servicios.BuildServiceProvider();

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

¡¡ Principio OPEN/CLOSE !!



## Registrando clases en el DI Container

### NOTA

También es posible registrar clases directamente, sin especificar la interfaz que implementan. Supongamos que queremos registrar la clase `CrearUsuarioUseCase`

```
var servicios = new ServiceCollection();
servicios.AddTransient<CrearUsuarioUseCase>();
ServiceProvider proveedor = servicios.BuildServiceProvider();
var crearUsuario = proveedor.GetService<CrearUsuarioUseCase>();
// aquí podemos utilizar la instancia provista por proveedor
```



## 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, LoggerNum>();`  
Registra el servicio `LoggerNum` como singleton. El proveedor devolverá siempre el mismo objeto cada vez que se lo requiera.

## 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. Esto corresponderá a la interacción que haga el usuario dentro de una única pestaña del navegador. Diferentes pestañas mantienen diferentes conexiones SignalR y por lo tanto representan distintos ámbitos.

# 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

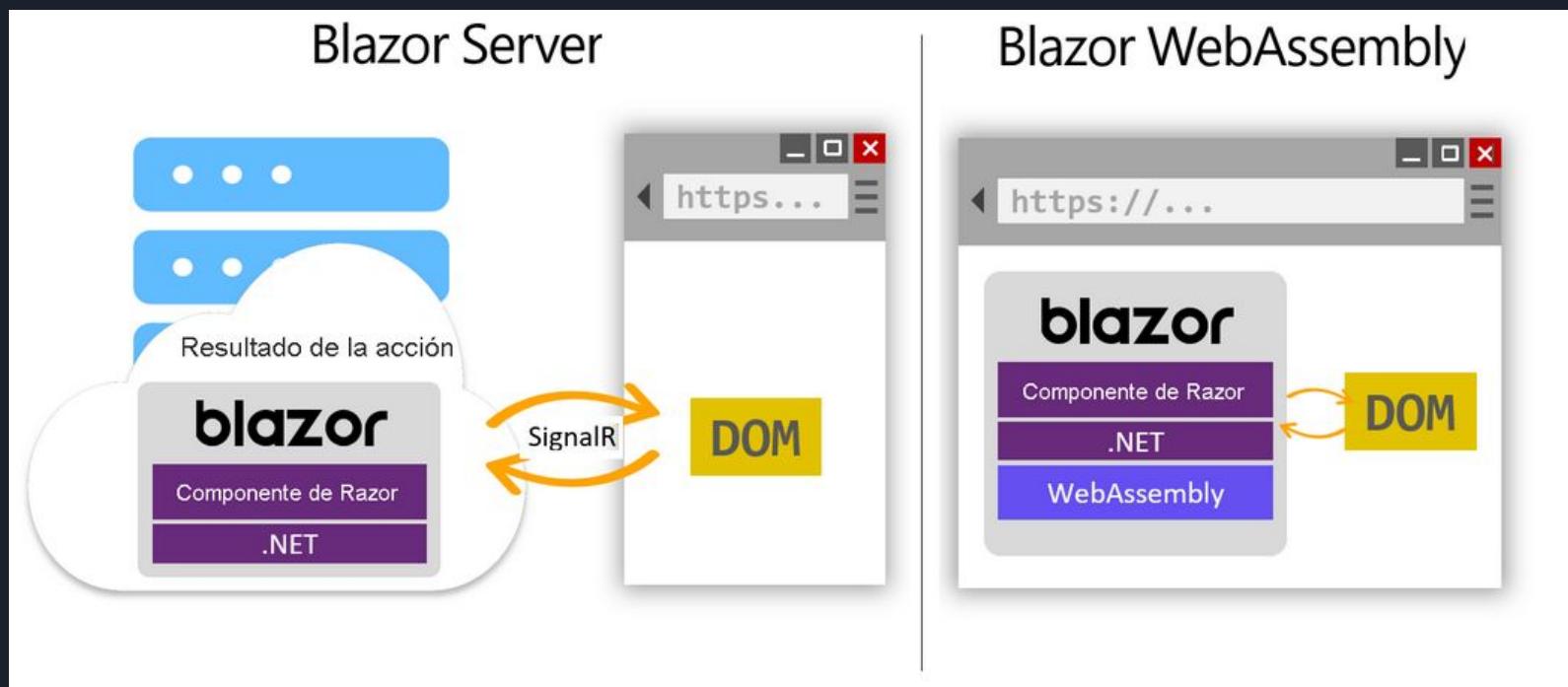
## ¿Qué es Razor?

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.json
- { appsettings.json
- HolaBlazor.csproj
- HolaBlazor.sln

C# Program.cs

En entorno de desarrollo

OUTLINE

No symbols found in document 'Program.cs'

Program.cs

```
1 using System;
2 var port = int.Parse(
3     Environment.GetEnvironmentVariable("ASPNETCORE_PORT")
4     ?? "5097");
5 // ...
6 built...
```

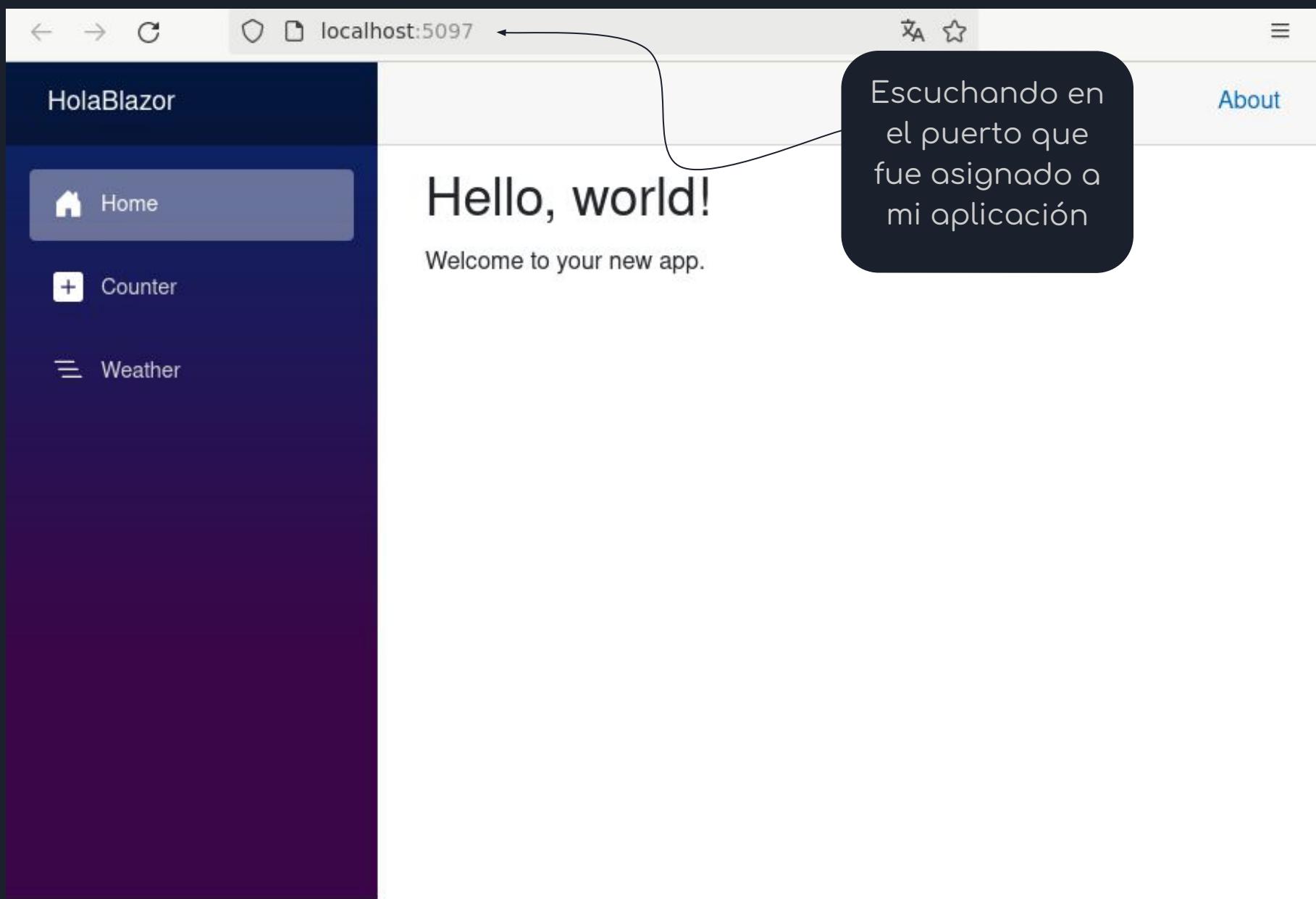
PROBLEMS

Filter (e.g. text,)

Escuchando en el puerto 5097.  
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

```
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
```

0 0 0 0 C#: HolaBlazor (HolaBlazor) Projects: 1 Spaces: 4 UTF-8 CRLF C# 40



Primero  
veamos algo  
de HTML

- > obj
- > Properties
- ✓ wwwroot
  - > bootstrap
  - # app.css
  - ↳ clientes.html
  - favicon.png
- { } appsettings.Devel...
- { } appsettings.json
- RSS HolaBlazor.csproj

#### ✓ OUTLINE

No symbols f  
in document  
'clientes.htm'

Program.cs    clientes.html

```
wwwroot > clientes.html
1 ! ←
2 ! !!
Emmet Abbreviation
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <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.



# HTML

The screenshot shows the Visual Studio Code interface with the following elements:

- EXPLORER** view on the left showing project files: Properties, wwwroot (bootstrap, app.css, clientes.html, favicon.png), appsettings.Development.json, appsettings.json, HolaBlazor.csproj, and OUTLINE.
- Program.cs** and **clientes.html** are open in the main editor area.
- DEBUG CONSOLE** at the bottom showing logs: "info: Microsoft.Hosting.Lifetime[14]" and "Now listening on: http://localhost:5001".
- Code Editor** showing the HTML code:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
</body>
</html>
```
- Annotations**:
  - A callout box in the top-left corner states: "Visual Studio Code nos crea el esquema básico de un documento html".
  - A callout box on the right side states: "Entre las etiquetas <body> y </body> colocaremos el contenido visible de la página web".

## HTML

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>
```

Document

localhost:5097/clientes.html

## Listado de clientes

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

info: Microsoft.Hosting.Lifetime[14]

Now listening on: http://[::]:5097

Microsoft.Hosting.Lifetime: [14]

info: Microsoft.Hosting.Lifetime[14]

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

ISOLE TERMINAL

C#: HolaBlazor (HolaBlazor) Projects:



# 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 -->
```

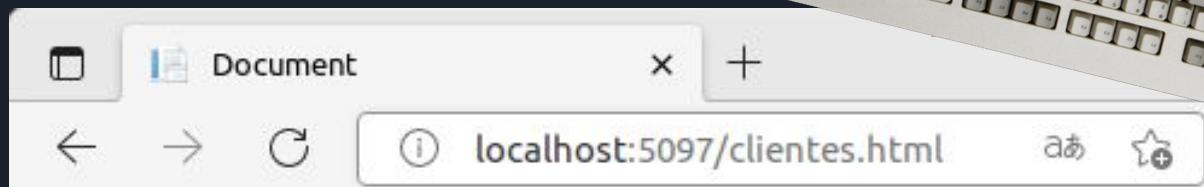
Realizar las siguientes pruebas

```
<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

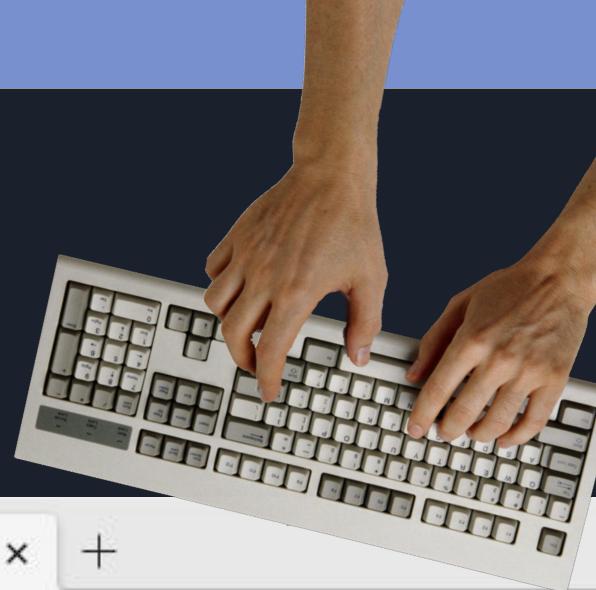
```
<body>
  <h1> Listado de clientes </h1>
  <p> Esto es un párrafo,
      otro renglón </p>
</body>
```



## Listado de clientes

Esto es un párrafo, otro renglón

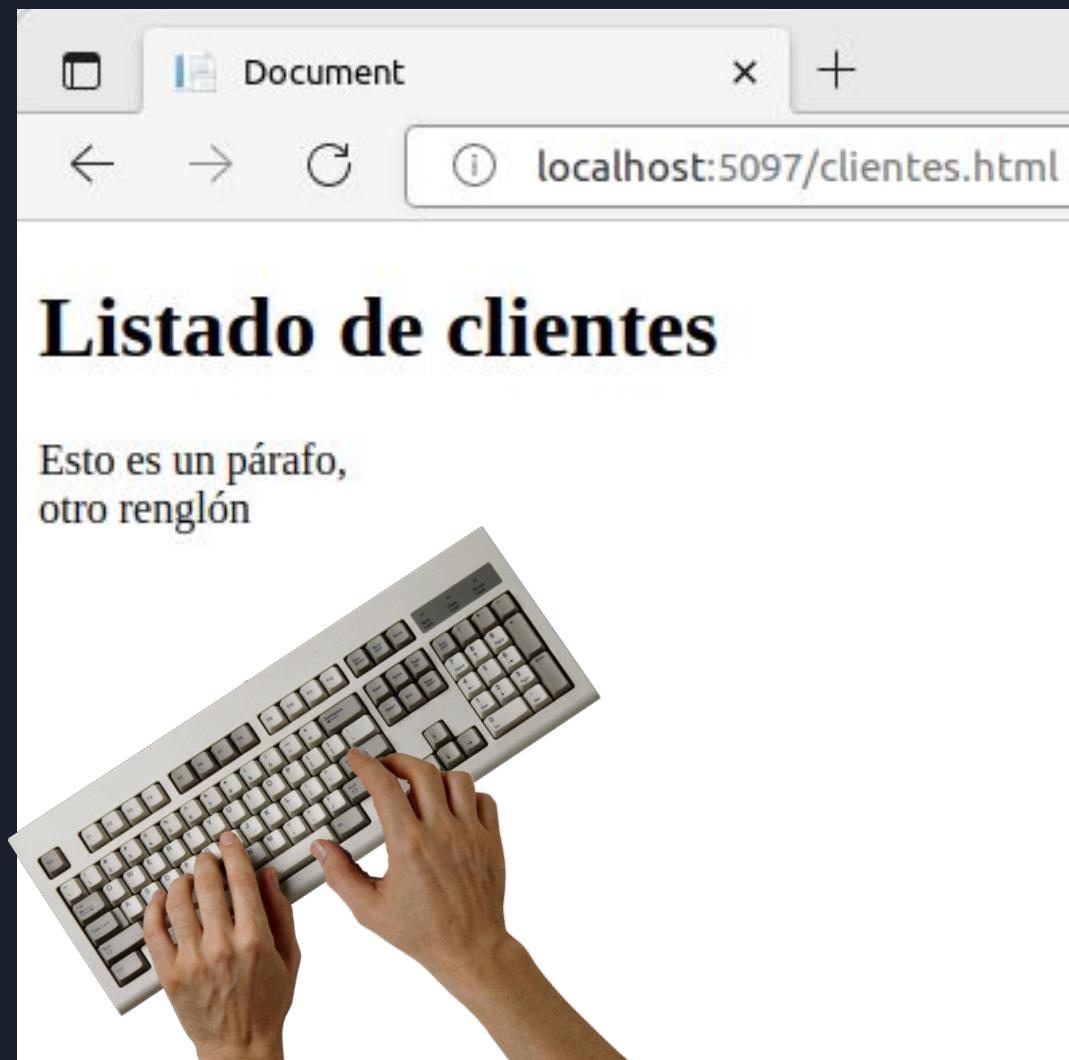
Se ignoran los fines de línea



Probar el siguiente código

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

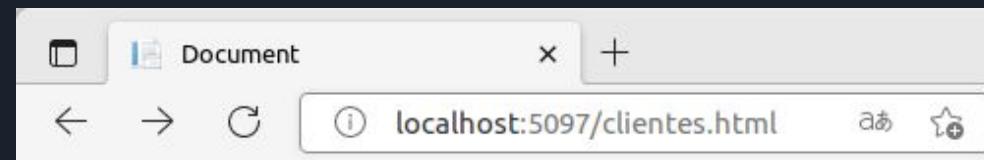
Agregar la etiqueta  
de fin de línea



Probar el siguiente código

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

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



## Listado de clientes

Esto es un párrafo,  
otro renglón

[ir a google](http://www.google.com)





## 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 <**img**> 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:

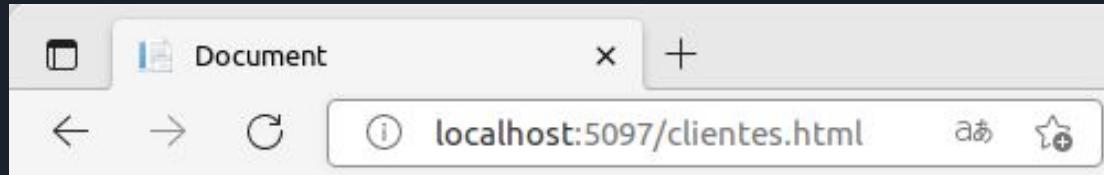
```
<html lang="es">
```

## 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

<li> se usa para especificar un ítem dentro de una lista  
¿Cuál es la diferencia entre </ul> y <ol>?  
¿Para qué sirve el atributo title?



## Listado de clientes

- Juan
  - María
  - Laura
1. Juan
  2. María
  3. Laura





## Etiquetas <div> y <span>

- <**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

- <**span**>

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"`

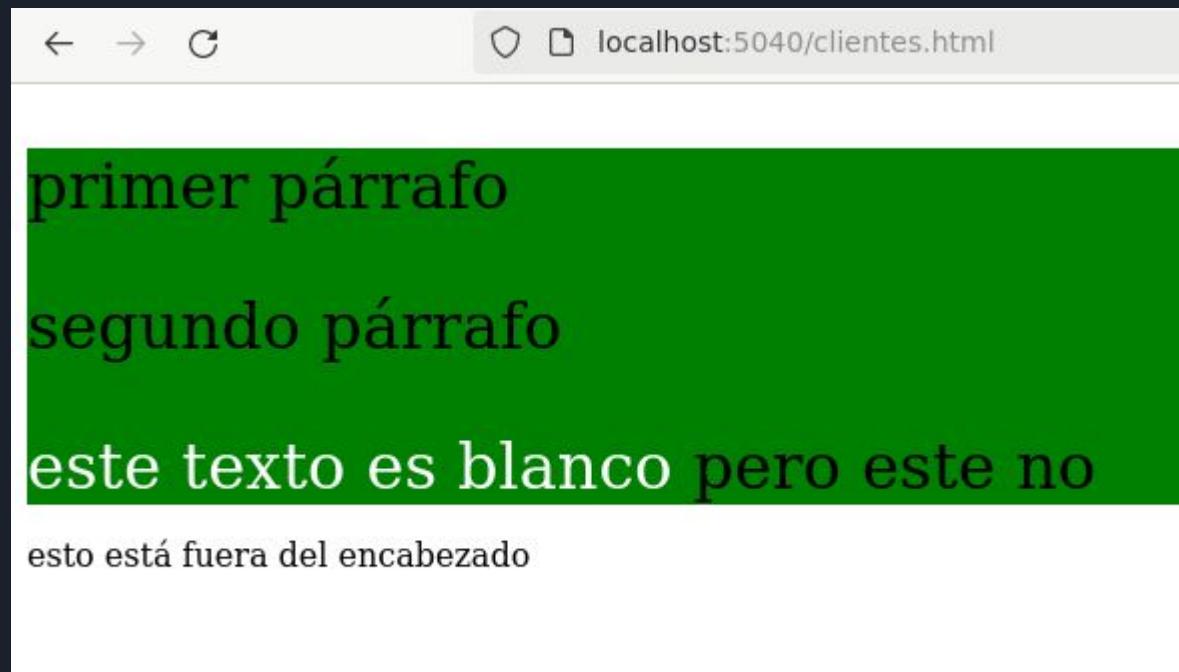


Copiar el código del archivo  
11\_RecursosParaLaTeoria.txt

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>
```



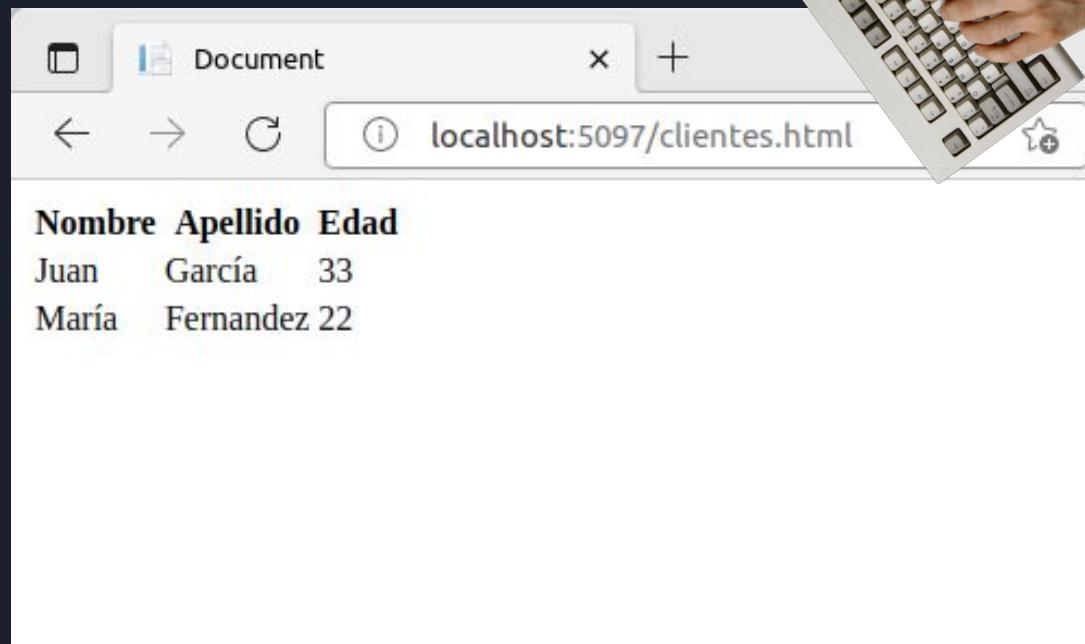


## Tablas

- 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`

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 Code interface with the following details:

- Solution Explorer:** Shows the project structure for "HolaBlazor". The "Components" folder contains "Layout", "Pages", and "Components". The "Pages" folder contains "Counter.razor", "Error.razor", "Hola.razor" (which is selected), "Home.razor", and "Weather.razor". Other files like "App.razor", "Routes.razor", and "\_Imports.razor" are also listed.
- Editor:** The file "Hola.razor" is open in the editor. It contains the following code:

```
1 @page "/hola"
2
3 <h1>Hola Mundo</h1>
4
5
```
- Bottom Status Bar:** Shows icons for file status (0 errors, 0 warnings, 0 info), a C# icon, the project name "C#: HolaBlazor (HolaBlazor)", the number of projects (1), and file endings ("CRLF", "A", "zor").
- Bottom Right:** An illustration of hands typing on a keyboard.
- Callout:** A callout box points from the "Hola.razor" file in the Solution Explorer to the "Hola.razor" component in the editor, containing the following text:

Crear el archivo **Hola.razor** en la carpeta **Components/Pages** con este contenido  
Por convención en esta carpeta se colocan los componentes “ruteables”



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



A screenshot of a web browser window titled "HolaBlazor". The address bar shows "localhost:5097/hola". The page content displays "Hola mundo". A callout bubble points to the address bar with the text: "Esta es la ruta que indicamos en la directiva @page del componente".



The screenshot shows a Microsoft Visual Studio IDE window. On the left is the Solution Explorer, which lists a project named "HolaBlazor" containing files like "Dependencias", "Properties", "Components", "Layout", "Pages", and several Razor components such as "Counter.razor", "Error.razor", "Hola.razor", "Home.razor" (which is selected and highlighted in blue), and "Weather.razor". Below these are "App.razor", "Routes.razor", and "\_Imports.razor". The status bar at the bottom indicates the project is "C#: HolaBlazor (HolaBlazor)" with "Projects: 1".

The main code editor window displays the file "Home.razor" with the following content:

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

A callout bubble with a black border and rounded corners points from the text "Hola" in the second line of the code to the explanatory text below. The bubble contains the following text:

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



A screenshot of a web browser window displaying a Blazor application. The URL in the address bar is `localhost:5097`. The page content includes the text "Hello, world!" and "Welcome to your new app." followed by two "Hola Mundo" messages. On the left side, there is a dark sidebar with a navigation menu. The "Home" item is highlighted with a blue background, while "Counter" and "Weather" are shown with white backgrounds and gray icons. The sidebar has a gradient background transitioning from dark purple at the bottom to dark blue at the top.

- Home
- Counter
- Weather

Hello, world!

Welcome to your new app.

Hola Mundo

Hola Mundo



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



```
@page "/hola"

<h1>Hola @nombre</h1>

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

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

La sección se identifica con  
@code{...}

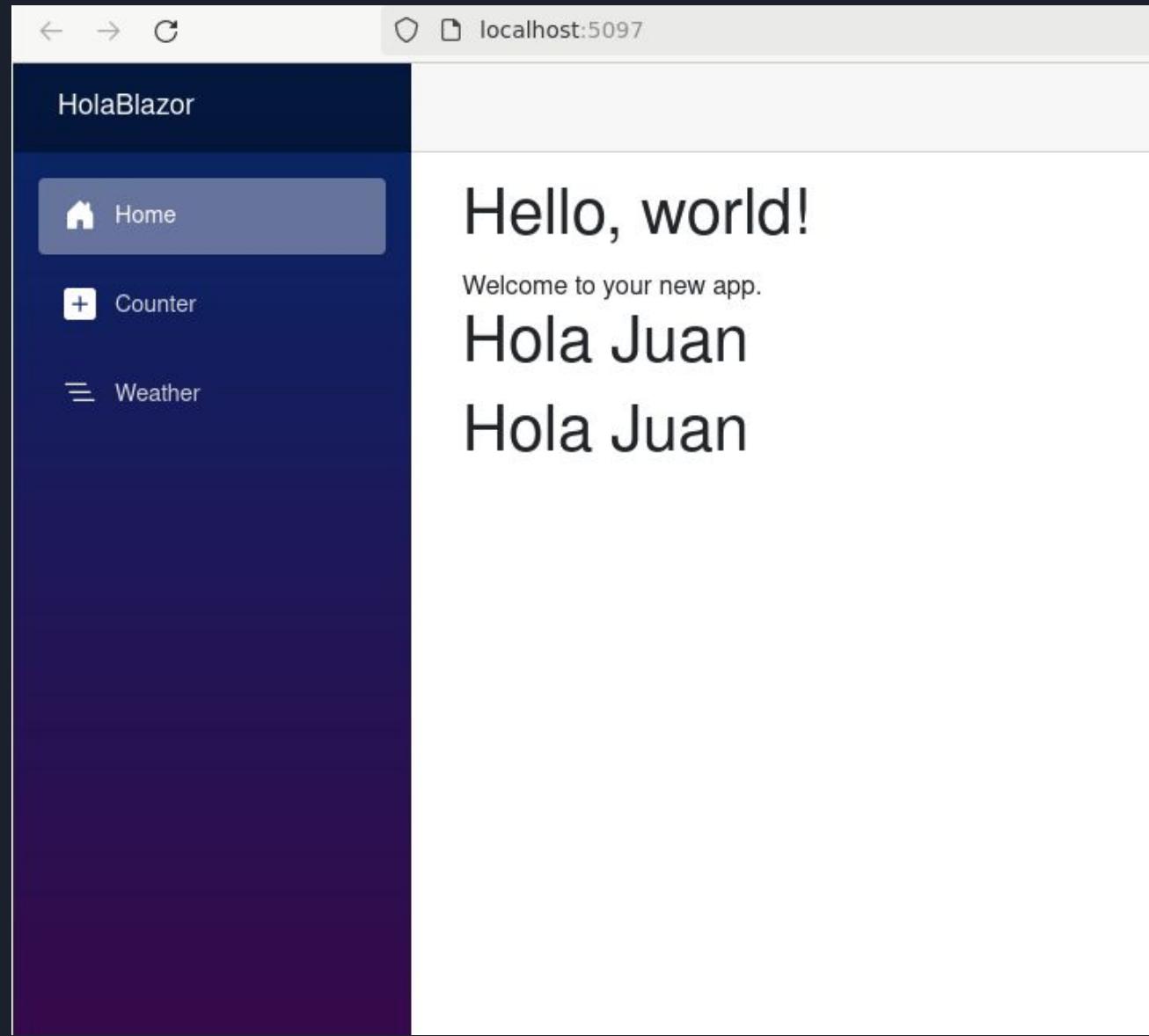
Esta es la sección de la vista. Utilizamos la @ para cambiar de código HTML a C#, en este caso para acceder a la variable nombre

## HTML

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

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

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



RUN AND DEBUG C#: HolaBlazor ... clientes.html ⚡ 🔍 ⏪ ⏴ ⏵

VARIABLES

Locals > this: {HolaBlazor.Components.Pages.Hola}

WATCH

CALL STACK

BREAKPOINTS

Components > Pages > Hola.razor

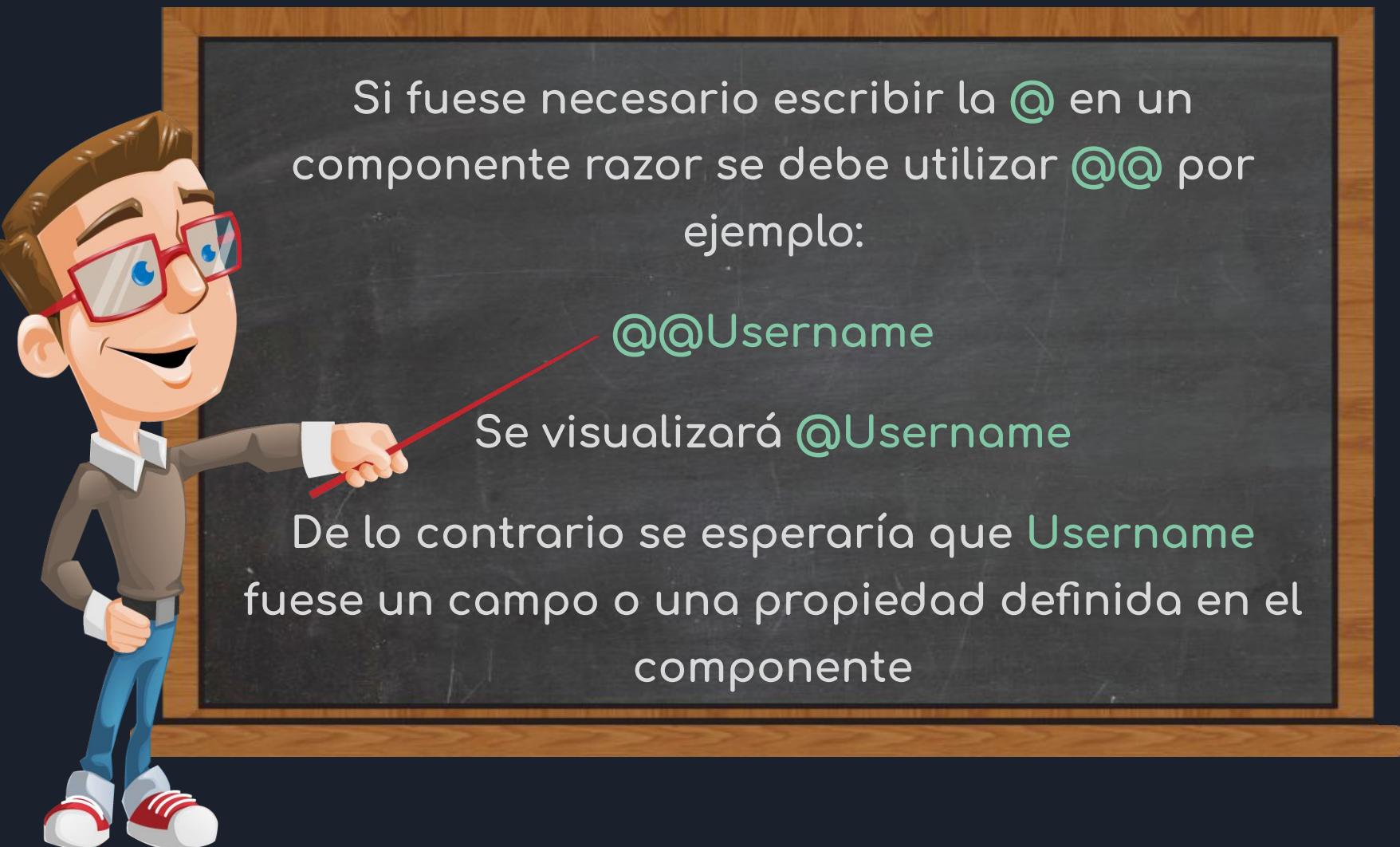
```
1 @page "/hola"
2
3 <h1>Hola @nombre</h1>
4
5 @code{
6     string nombre="Juan";
7 }
```

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

✖ 0 ⚠ 0 ⚡ 0 C#: HolaBlazor (HolaBlazor) Projects: 1 CRLF ASP.NET Razor ⚡ 66

## Secuencia de escape para @

A cartoon illustration of a teacher with brown hair and glasses, wearing a brown sweater over a white collared shirt, standing next to a chalkboard. He is holding a red pointer stick and pointing it towards the chalkboard. The chalkboard has a wooden frame and contains text in Spanish.

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();
}
```

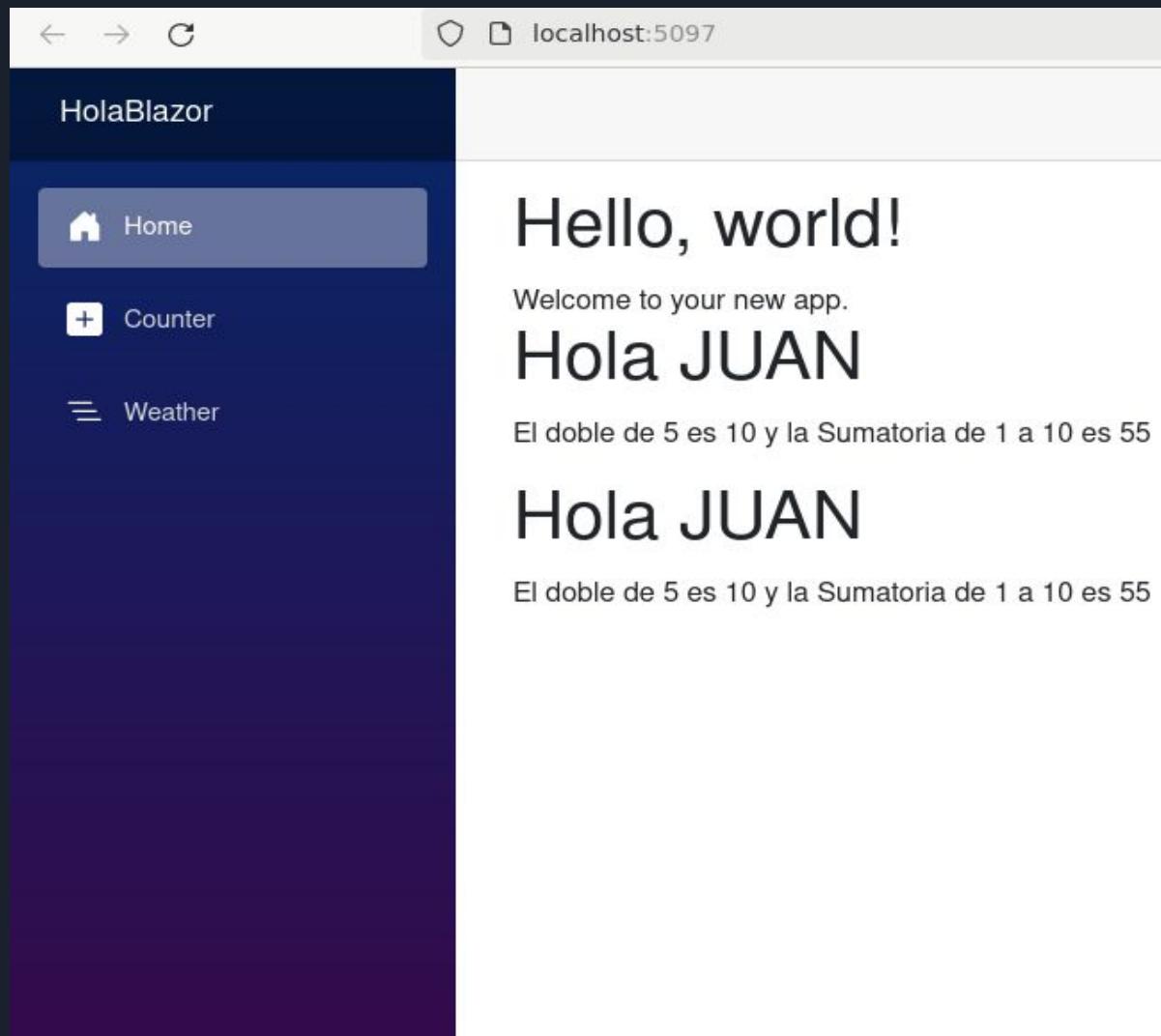
Copiar el código del archivo  
11\_RecursosParaLaTeoria.txt

# 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 C#. Suelen ser simples y no admiten espacios. El final de la expresión lo determina el contexto (por ejemplo, un espacio o un carácter HTML). Ejemplo:

```
<p>@nombre.ToUpper()</p>
```

En casos donde la ambigüedad podría causar errores (como operaciones matemáticas), se usan expresiones explícitas, delimitadas por paréntesis, por ejemplo:

```
<p>@(5 * 2)</p>  <!-- Correcto: expresión explícita -->
<p>@5 * 2</p>      <!-- Error: Razor intentaría interpretar "*" como HTML
                        provocando error de compilación: "5" is not valid
                        at the start of a code block-->
```



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



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

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

Es necesario si queremos que el componente sea interactivo

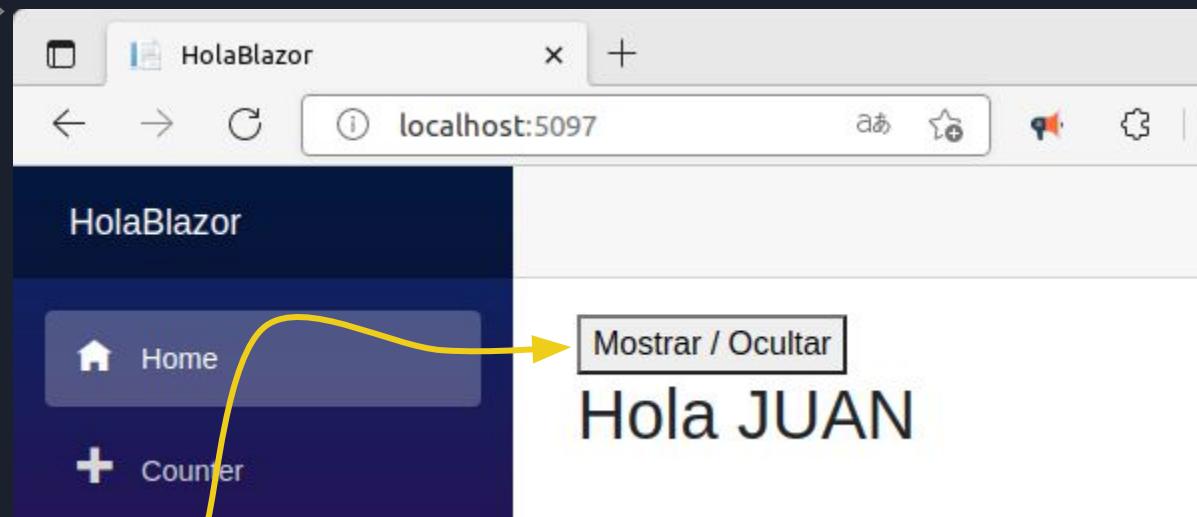
----- Hola.razor -----

```
@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;
    }
}
```

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

The screenshot shows the Visual Studio IDE interface. On the left, the Solution Explorer pane displays a project named "HolaBlazor" with files like "Program.cs", "Person.cs" (selected), and "appsettings.json". A yellow arrow points from a callout box labeled "Crear la carpeta Entidades y dentro de ella la clase Persona" to the "Person.cs" file in the Solution Explorer. Another yellow arrow points from the same callout box to the namespace declaration in the code. On the right, the code editor shows the "Person.cs" file with the following C# code:

```
namespace HolaBlazor.Entidades;
class Persona
{
    public string Nombre { get; set; } = "";
    public string Apellido { get; set; } = "";
    public int? Edad { get; set; } //podría faltar el dato de la edad

    // vamos a hardcodear una lista de personas
    // que usaremos en los siguientes ejemplos
    // para ello definimos el siguiente método estático
    public static List<Persona> GetLista()
    {
        return new List<Persona>()
        {
            new Persona() {Nombre="Pablo",Apellido="Perez", Edad=34},
            new Persona() {Nombre="Laura",Apellido="García", Edad=30},
            new Persona() {Nombre="José",Apellido="Lopez", Edad=45},
            new Persona() {Nombre="Ana",Apellido="Colombo", Edad=21},
            new Persona() {Nombre="María",Apellido="Suarez", Edad=15},
        };
    }
}
```

A callout box on the right side of the code editor contains the text "Definirla en el namespace HolaBlazor.Entidades". At the bottom of the screen, there is a keyboard icon with hands, a status bar showing "C# .NET Blazor (HolaBlazor) Projects: 1", and a callout box containing the text "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 screenshot shows a web browser window displaying a Blazor application. The address bar indicates the site is running on localhost:5097. The page title is "HolaBlazor". On the left, there is a dark sidebar menu with three items: "Home" (selected), "Counter", and "Weather". The main content area has a heading "Listado de personas" and a bulleted list of five people with their names and ages.

← → ⌂ ⚡ localhost:5097 ⚡ ⭐ ⌄

HolaBlazor

About

Home Counter Weather

## Listado de personas

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

The screenshot shows the Visual Studio IDE interface. On the left, the Solution Explorer pane displays the project structure for "HolaBlazor". It includes a "Components" folder containing "Layout", "Pages", and files like "Counter.razor", "Error.razor", "Hola.razor", "Home.razor", and "Weather.razor". Below these are "App.razor" and "Routes.razor". A "Entidades" folder is also present. In the center, the code editor window is open to the file "\_Imports.razor". The code listed is:

```
Components > _Imports.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
```

A callout bubble with a wavy arrow points from the text "ahora se puede borrar la directiva using agregada en Hola.razor" to the line "11 @using HolaBlazor.Entidades". The bubble contains the following text:

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)

At the bottom of the screen, there is a decorative graphic of a hand pointing at a keyboard.



# 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

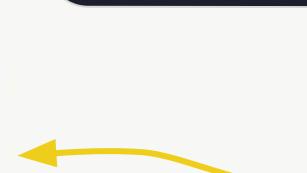
HomeCounterWeather

## Listado de personas

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

Nombre
Apellido
Edad
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>  
}  
  
@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();  
    }  
}
```

Mostramos estos elementos sólo si la propiedad SoloLectura es false

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 />
```

The screenshot shows a Blazor application running at `localhost:5097`. The application has a dark blue sidebar on the left with three items: "Home", "Counter", and "Weather". The main content area displays two identical lists of people, each enclosed in a large curly brace. The first list is annotated with a callout box containing the code `<Hola SoloLectura="true"/>`. The second list is annotated with another callout box containing the code `<Hola />`. At the bottom of the main content area, there is a form with four input fields: "Nombre", "Apellido", "Edad" (with a dropdown arrow), and a button labeled "Agregar".

HolaBlazor

About

Home

Counter

Weather

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

# Práctica sobre la teoría 11

El contenido de esta práctica, está alineado con el trabajo final de codificación que se solicitará próximamente.

Se desea aplicar los conocimientos adquiridos sobre persistencia de datos con Entity Framework Core (EF Core) utilizando SQLite como motor de base de datos, y la gestión avanzada de dependencias a través del Contenedor de Inyección de Dependencias (DI Container) provisto por .NET.

Se espera que los alumnos tomen como base la solución desarrollada en el Primer Trabajo Práctico de Codificación (aquel donde implementaron una capa de repositorios basada en archivos de texto y una arquitectura limpia con casos de uso) y la evolucionen para incorporar estas tecnologías y patrones.

- 1) Implementación de Repositorios con Entity Framework Core y SQLite: Adaptar el proyecto de repositorio para implementar la persistencia en una base de datos SQLite. Reimplementar cada una de las clases de repositorio concretas (aquellas que implementaban las interfaces de repositorio definidas en la capa de aplicación). Realizar las operaciones de estos repositorios traduciéndolas a operaciones de EF Core (ej. context.Add(), context.SaveChanges(), consultas LINQ, etc.)
- 2) Integración del Contenedor de Inyección de Dependencias (DI Container) de .NET: Refactorizar el Punto de Entrada de la Aplicación (Program.cs en el proyecto de consola) eliminando la creación manual (con new) de las dependencias. Registrar también a los casos de uso en el contenedor. Configurar `ServiceCollection` y obtener el `ServiceProvider`. Probar su funcionamiento ejecutando los casos de uso a través de las instancias provistas por el `ServiceProvider`.