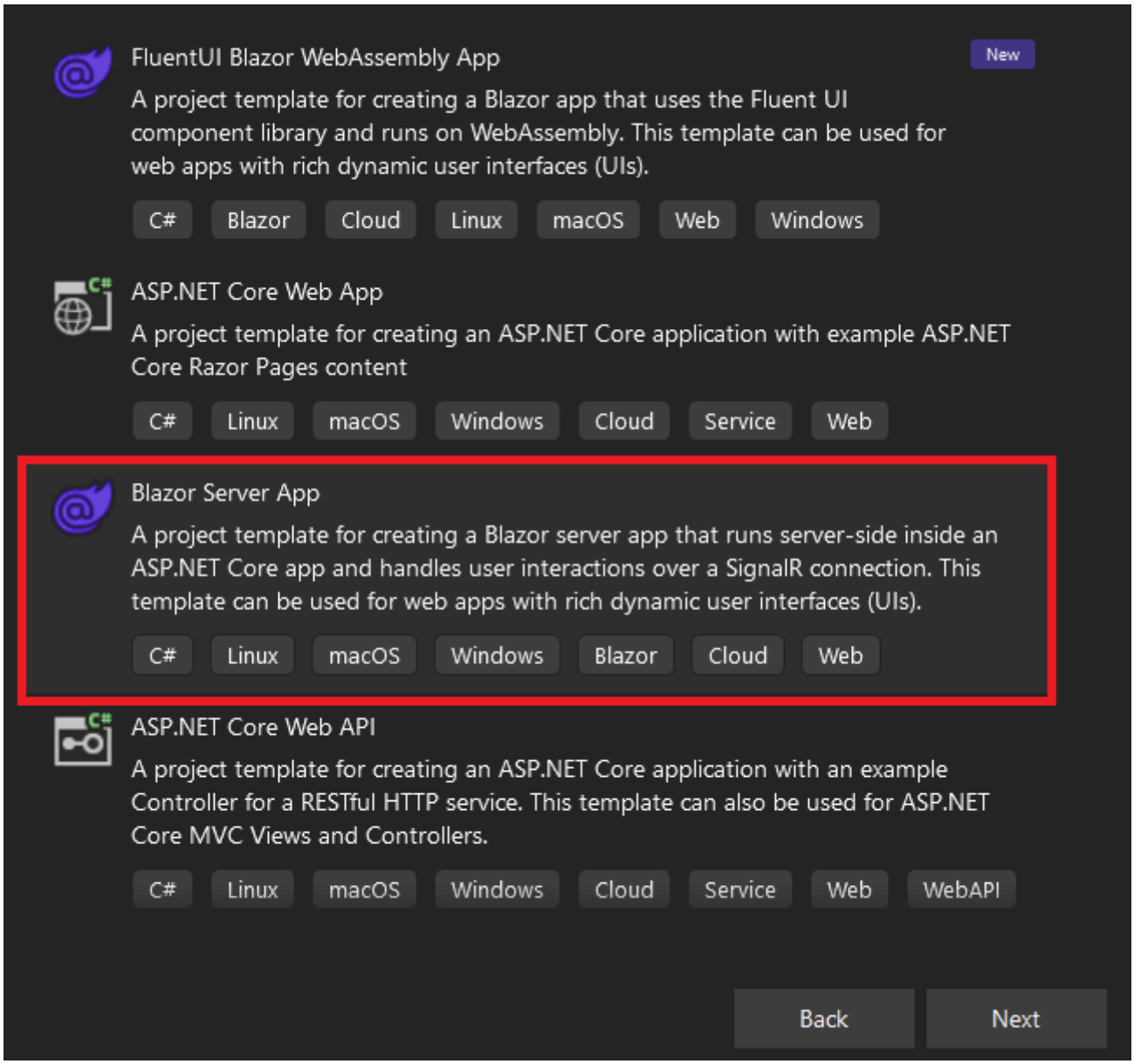
Crear una librería personalizada para TinyMCE con carga automática de archivos javascript

La idea detrás de este tutorial es la de mostrar como crear una librería Blazor para albergar un componente personalizado, asi como, una técnica que permita cargar los archivos javascript de forma automática. Para ello usaremos TinyMCE que es un editor de texto muy conocido y de grandes capacidades. Si bien, existe un componente oficial de TinyMCE para Blazor, el objetivo de este tutorial es el de mostrar unas técnicas que pueden ser útiles en otros contextos. Empecemos.

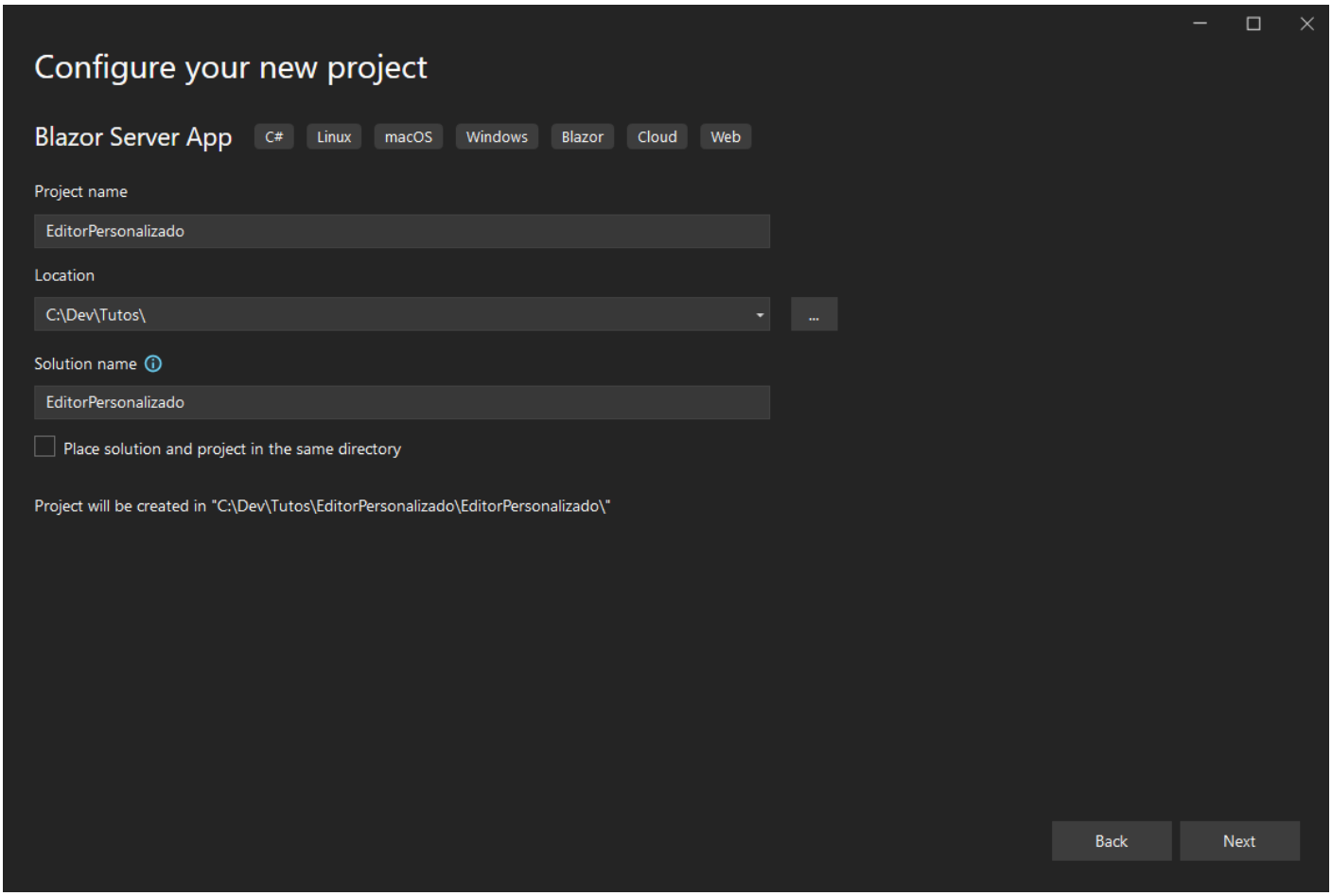
> Puedes encontrar el código fuente de este tutorial en nuestro repositorio de GitHub.

Creación de un nuevo proyecto

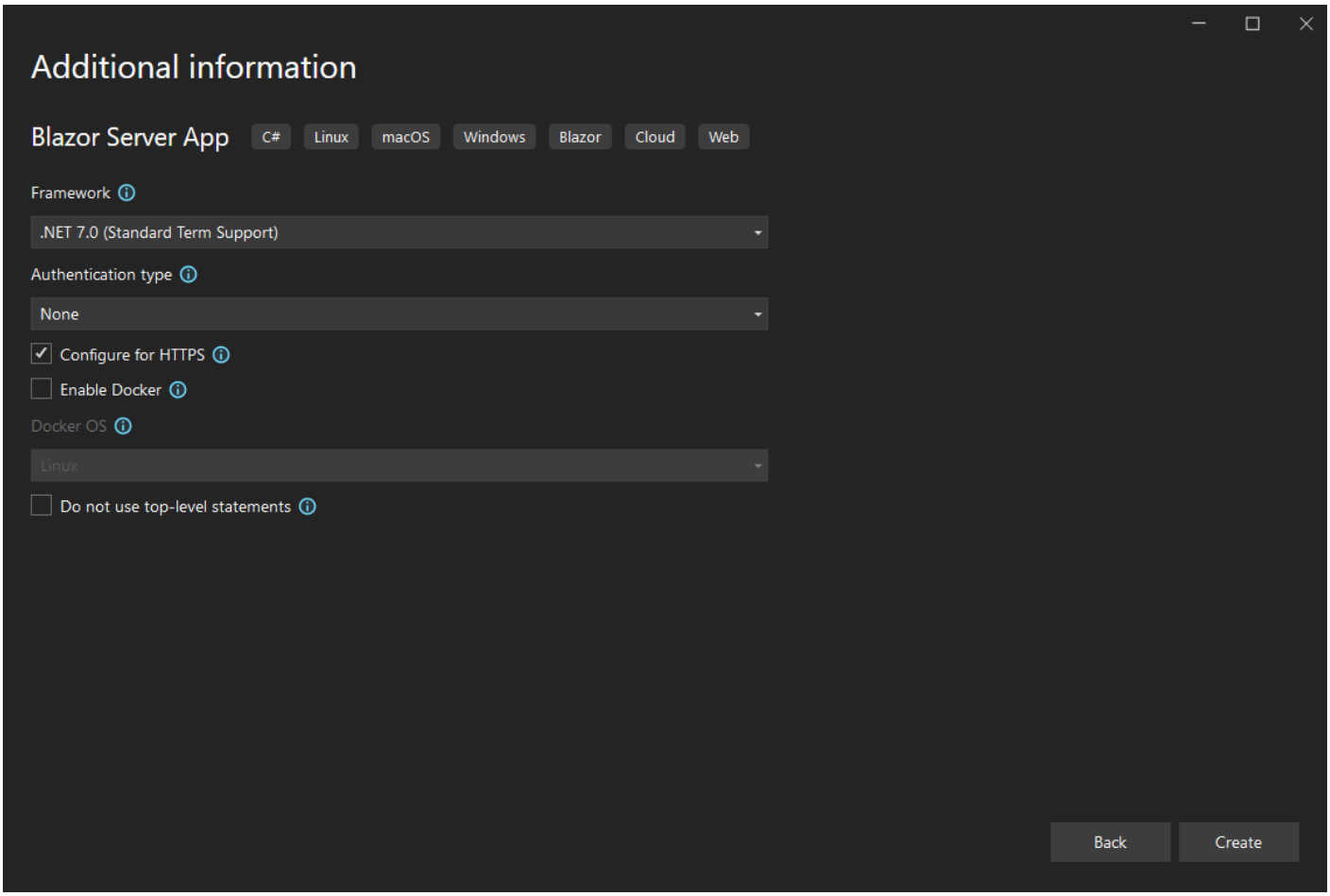
En primer lugar vamos a crear una nueva solucion y dos proyectos para albergar nuesto código. Para empezar abre Visual Studio 2022 y crea un nuevo proyecto de tipo **Blazor Server App** y pulsa en siguiente.



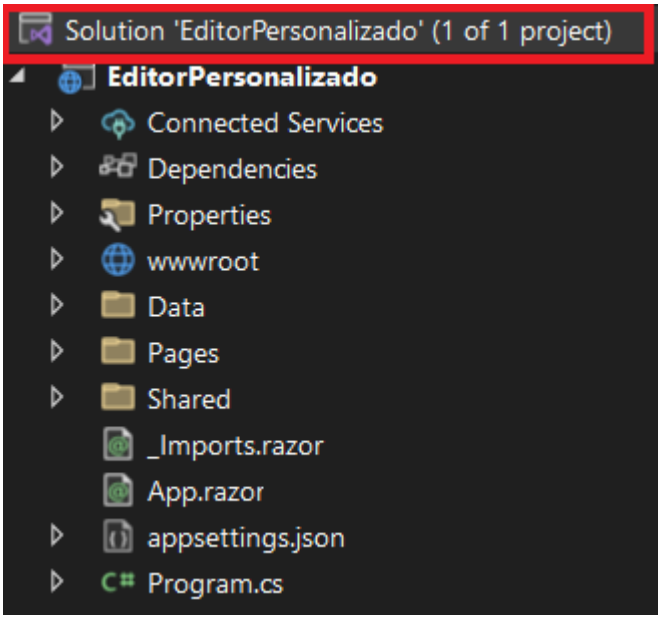
Rellena las propiedades del nuevo proyecto. Dale un nombre al proyecto y una ubicación y pulsa siguiente.

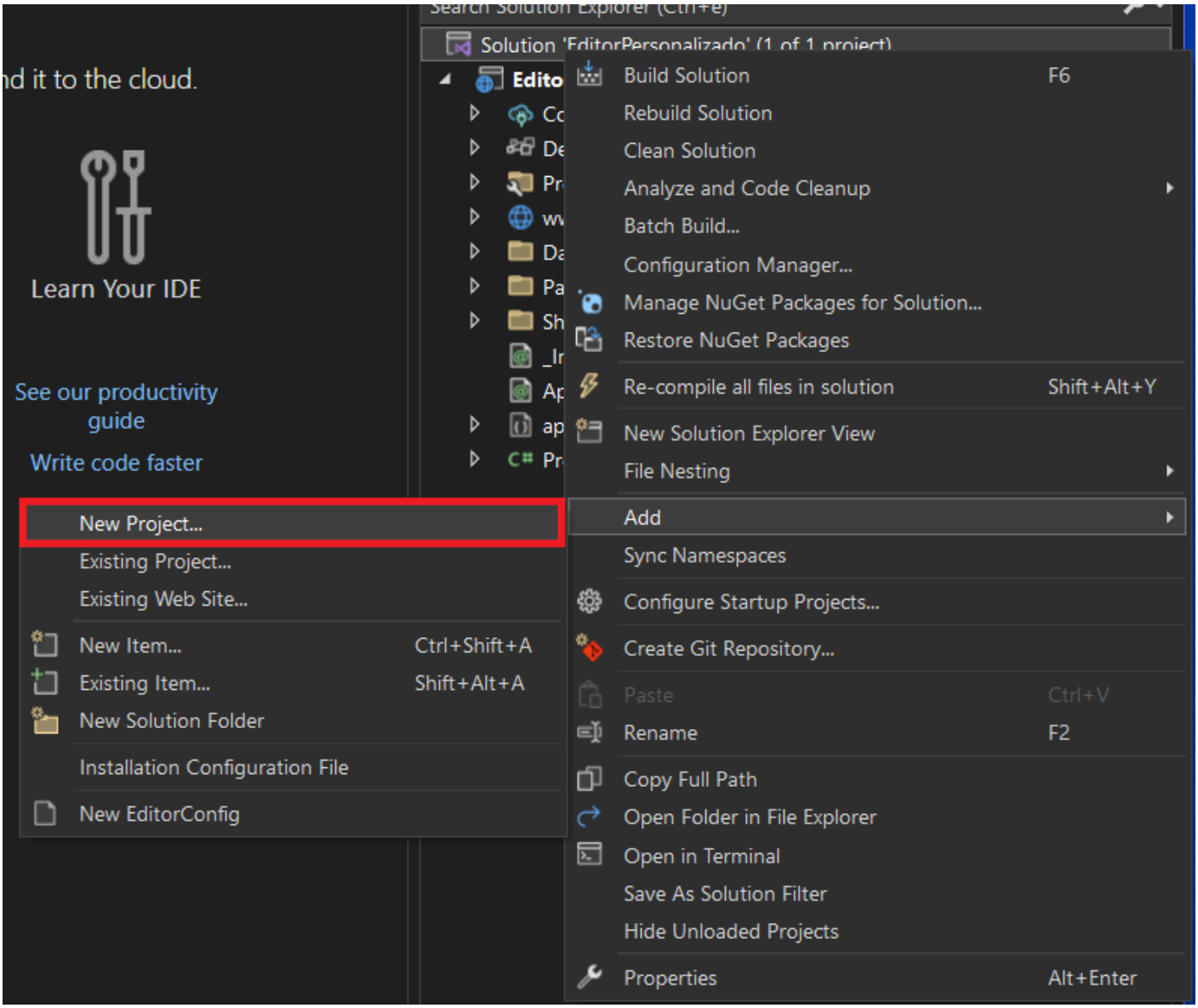


En la siguiente ventana puedes dejar los parámetros por defecto y pulsar el botón crear.

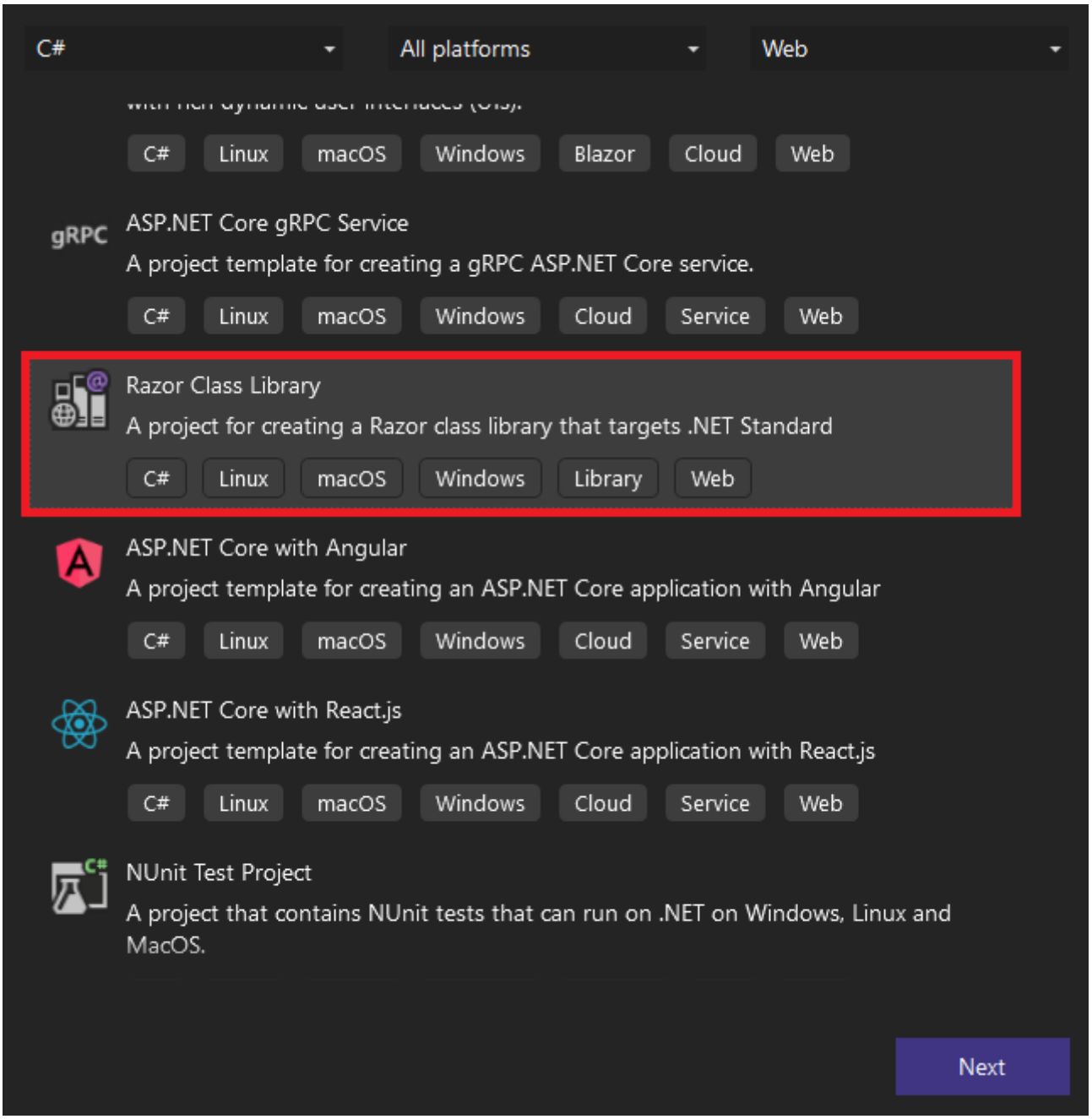


El proyecto creará una nueva solución y un nuevo proyecto en la ubicación que elegiste. El proyecto que acabamos de crear nos servirá para probar la librería en la que crearemos el componente. Ahora vamos a crear el proyecto de librería. En el explorador de soluciones, pulsa con el botón derecho del ratón en la raíz del proyecto para abrir el menú contextual y pulsa `Añadir > Nuevo proyecto...`.

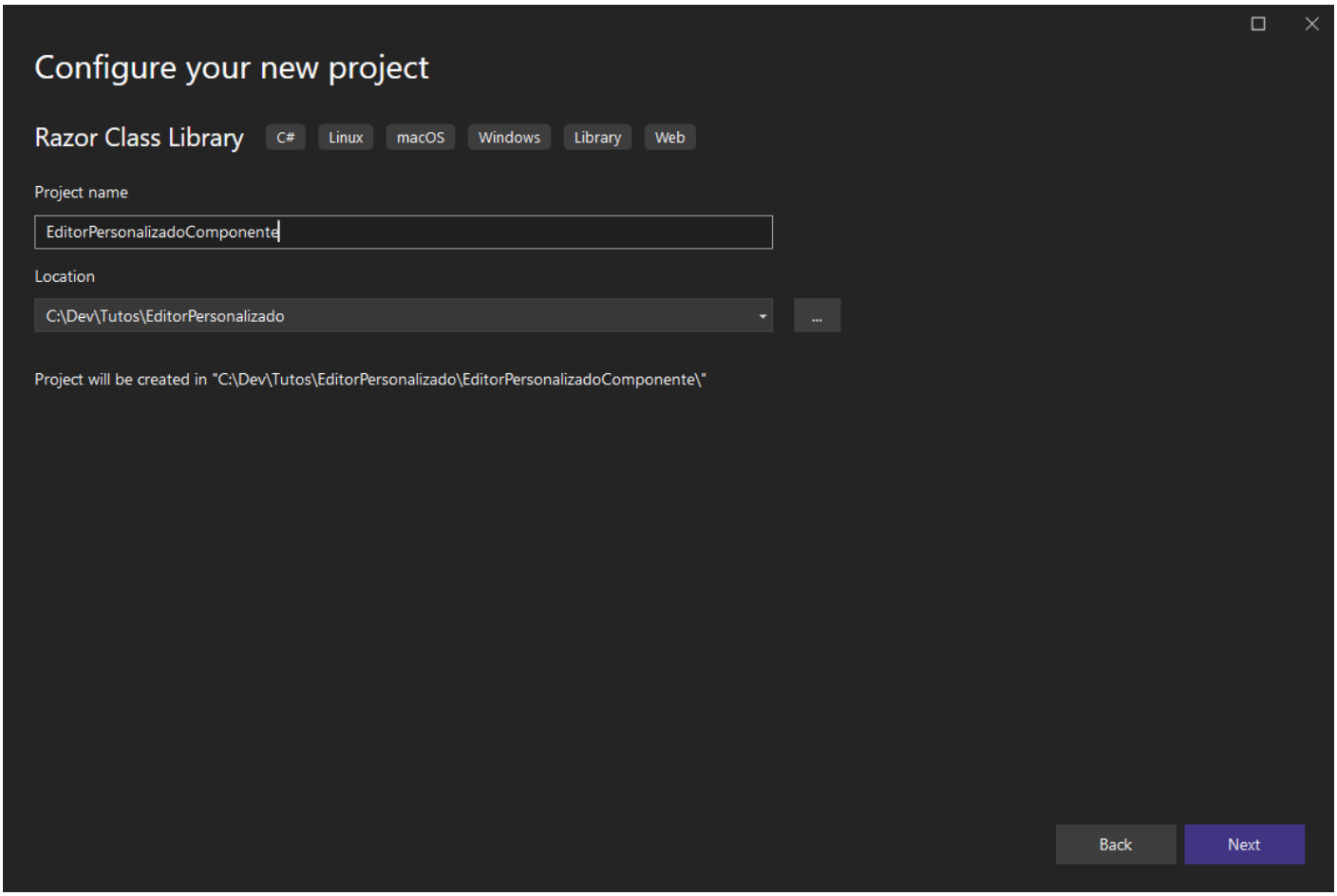




A continuación, selecciona el tipo de proyecto **Razor Class Library** y pulsa Siguiente.



Dale un nombre al nuevo proyecto dejando la ubicación como está y pulsa Siguiente. En la ventana siguiente deja los valores por defecto y pulsa Crear.

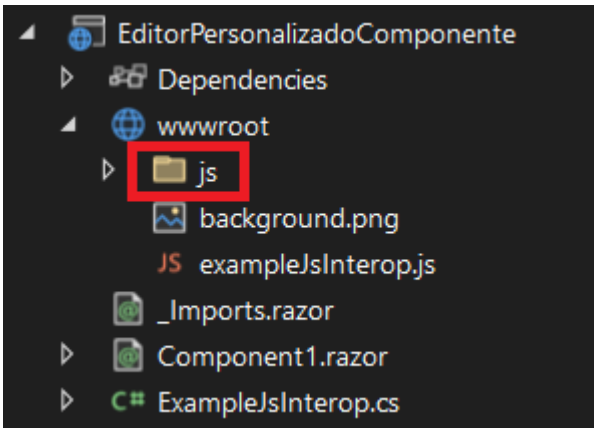


El nuevo proyecto que acabamos de crear permite la creación de una librería de componentes y es el proyecto que usaremos para crear nuestro componente personalizado.

Descarga de TinyMCE

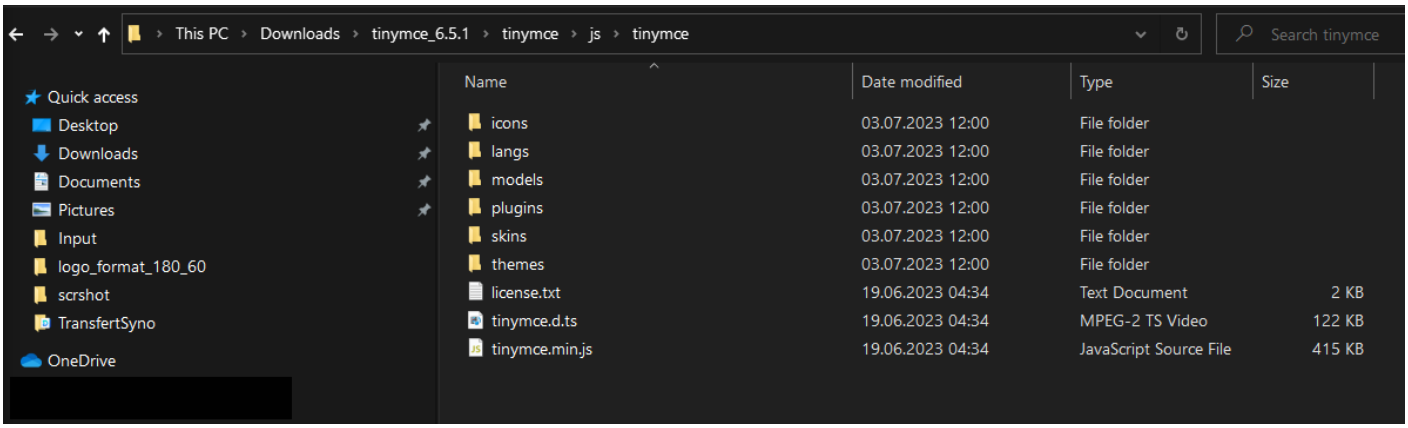
Nuestro componente personalizado usa el editor TinyMCE que está desarrollado en javascipt. Para tener el máximo control posible necesitamos obtener los archivos que componen el editor. Estos archivos los podemos obtener en la siguiente direccion: https://www.tiny.cloud/get-tiny/. Pulsa en ***Download TinyMCE SDK Now*** para descargar TinyMCE.

En Visual Studio, en el proyecto *EditorPersonalizadoComponente*, dentro de la carpeta `wwwroot` crea una nueva carpeta `js`.

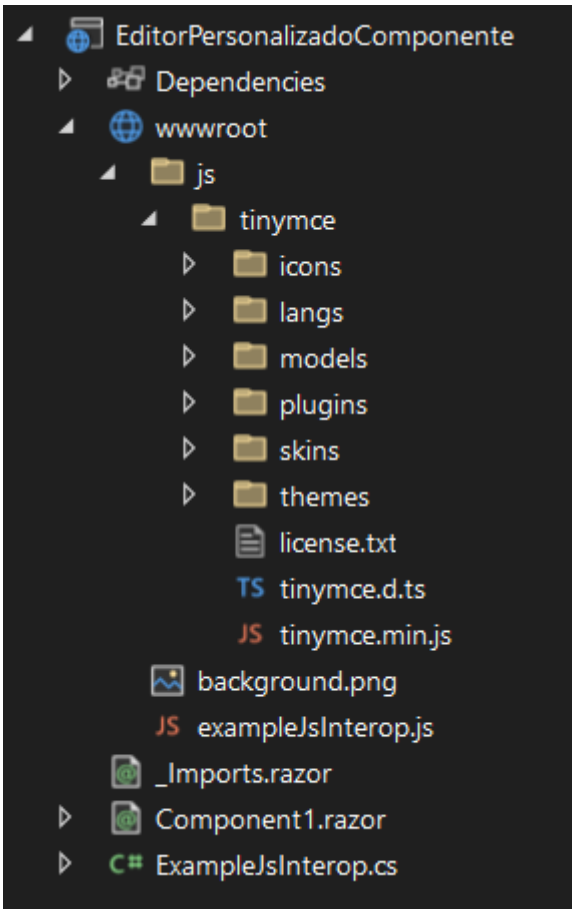


Extrae el archivo comprimido de TinyMCE en un directorio, por ejemplo en *Descargas*. Abre la carpeta y recorre los directorios hasta que encuentres el directorio `tinymce` que contiene todos los archivos de TinyMCE. Copia esta carpeta y todos los archivos que contiene dentro a la carpeta `wwwroot/js` .

> En Visual Studio puedes 'pegar' una carpeta en un directorio del Explorador de Soluciones como si se tratara del Explorador de Windows para hacer la copia.



Después de copiar los archivos, en el explorador de soluciones deberías tener algo similar a esta imagen:



Hemos instalado los archivos de TinyMCE dentro de la carpeta `wwwroot`, que es la carpeta que contiene los archivos que serán accesibles desde internet, del mismo modo que las imágenes y otros recursos de una página web, como las hojas de estilos en cascada (css).

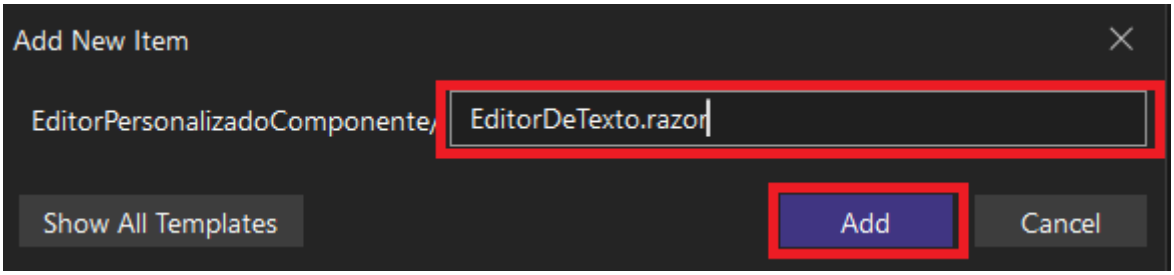
> Ten en cuenta que todos los archivos que se encuentren dentro de la carpeta wwwroot de una librería de Razor, son accesibles a través de la ruta `./_content/NombreLibreria`. En nuestro caso, el archivo `tinymce.min.js` es accesible en la ruta `./content/EditorPersonalizadoComponente/js/tinymce/tinymce.min.js`.

Creación del componente en la librería personalizada

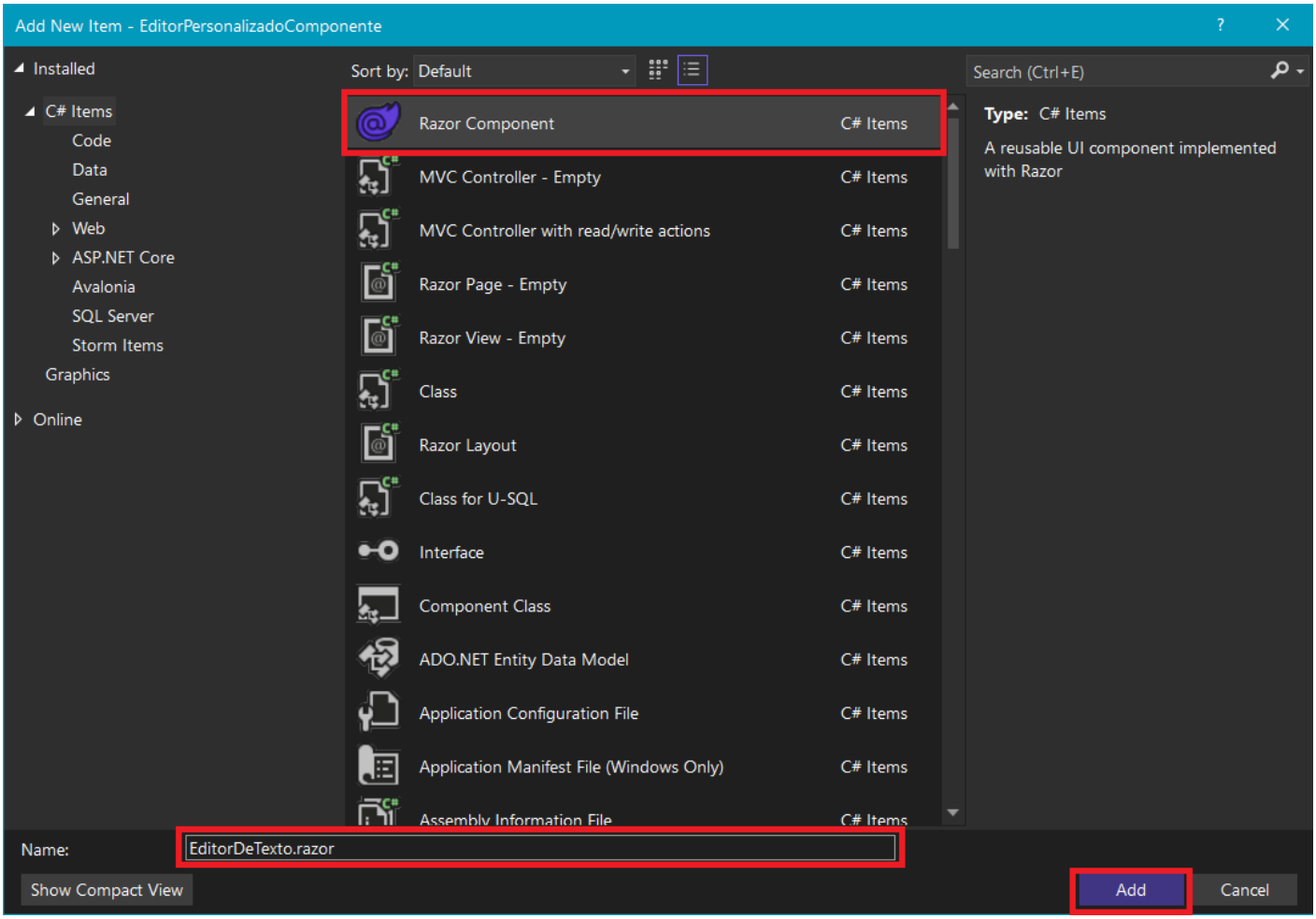
Para empezar, vamos a hacer un poco de limpieza. En el proyecto *EditorPersonalizadoComponente* elimina los archivos siguientes:

- wwwroot/exampleJsInterop.js
- Component1.razor
- ExampleJsInterop.cs

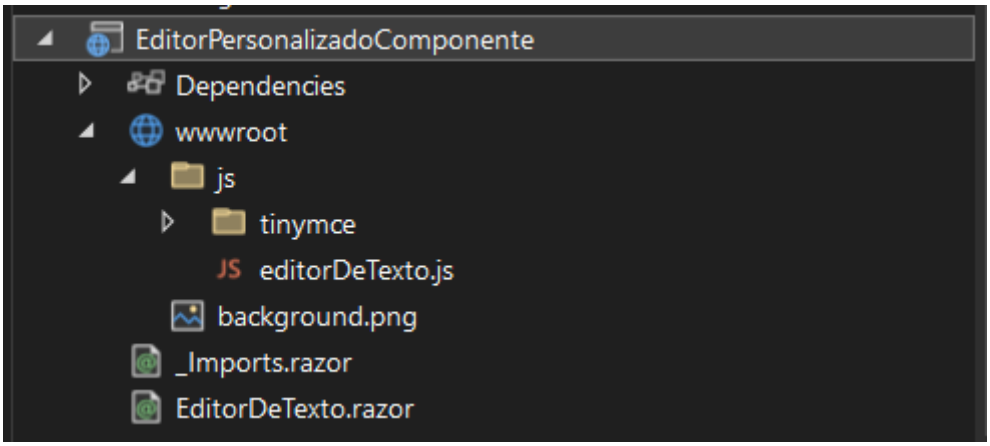
Nuestro nuevo componente necesitará dos archivos nuevos: un archivo de componente Blazor y un archivo javascript para la inicialización del componente. Vamos a crear en primer lugar el archivo de componente. En el Explorador de Soluciones selecciona el nodo del proyecto *EditorPersonalizadoComponente* y haz clic derecho con el ratón sobre él. Selecciona la opción `Añadir > Nuevo elemento`. Según como hayas configurado tu Visual Studio pueden mostrarse dos interfaces diferentes para la creación de un nuevo componente. Si se abre una ventana en la que te pide el nombre del archivo, esta es la nueva interfaz simplificada, escribe `EditorDeTexto.razor` para el nuevo componente, y pulsa el botón *Añadir*.



Si no, se abrirá la ventana clásica de creación de elementos. Selecciona el tipo *Componente de Razor*, como nombre pon EditorDeTexto.razor y pulsa el botón *Añadir*.



Ahora repite el proceso, pero esta vez para crear un archivo **javascript** dentro de la carpeta `wwwroot/js` al cual llamaremos `editorDeTexto.js`. Después de añadir los dos archivos el árbol del proyecto debería verse así:



Una vez creados los archivos, vamos a proceder a escribir el código. Abre el archivo `EditorDeTexto.razor` y escribe el código a continuación:

```
@inject IJSRuntime _jsruntime

<textarea id="editor"></textarea>

@code {
    private Lazy<Task<IJSObjectReference>> moduleTask;

    protected override async Task OnAfterRenderAsync(bool firstRender)
    {
        await _jsruntime.InvokeVoidAsync("import", "./js/tinymce/tinymce.min.js");
        moduleTask = new(() => _jsruntime.InvokeAsync<IJSObjectReference>("import",
"./js/editorDeTexto.js").AsTask());

        var module = await moduleTask.Value;
        await module.InvokeVoidAsync("initEditor", "#editor", "/js/tinymce");
    }
}
```

En el código anterior lo que hacemos es definir un área de texto (textarea) que servirá para renderizar el editor TinyMCE. Dentro del método `OnAfterRenderedAsync`, realizamos dos importaciones: una directa, que es la importación de la librería TinyMCE como módulo (lin. 10). La segunda es la carga del módulo javascript de nuestro editor (lin 11), pero esta vez guardamos una referencia para poder acceder a sus métodos desde C# en todo momento (lin. 13 - 15). Finalmente, en la linea 15, llamamos al método `initEditor` y le pasamos dos parámetros. Como veremos a continuación, esta función se ocupa de inicializar el editor TinyMCE.

Pasemos ahora al archivo `editorDeTexto.js`. Escribe el siguiente código:

```
export function initEditor(selector, base_url) {

    tinymce.init({
        selector: selector,
        base_url: base_url,
        suffix: '.min'
    });
}
```

En el código anterior estamos realizando una configuración básica del editor TinyMCE. El `selector`, es el elemento dentro del HTML que se usará para renderizar el editor. El base_url se usa para definir la url de base donde se encuentran el resto de archivos de TinyMCE. Como estamos haciendo una carga dinámica de la librería, es necesario darle la url de base para que TinyMCE sea capaz de cargar el resto de archivos. El último parámetro, *suffix*, guarda relación con la carga de archivos también, si nuestra distribución de TinyMCE es una versión minimizada (minimized), debemos definir el sufijo a *min*, para que TinyMCE pueda construir correctamente las rutas a los archivos. Para saber si la distribución que descargaste está minimizada, observa los nombres de los archivos. Si éstos contienen `xxxxx.min.xxx` es una version minimizada, si no, es una versión normal. Un ejemplo de archivo minimizado es `tinymce.min.js` y uno normal `tinymce.js`.

Por último, en el archivo `_Imports.razor` del proyecto *EditorPersonalizadoComponente,* añade la siguiente línea de código:

```
@using Microsoft.JSInterop
```

El archivo quedará así:

```
@using Microsoft.AspNetCore.Components.Web
@using Microsoft.JSInterop
```

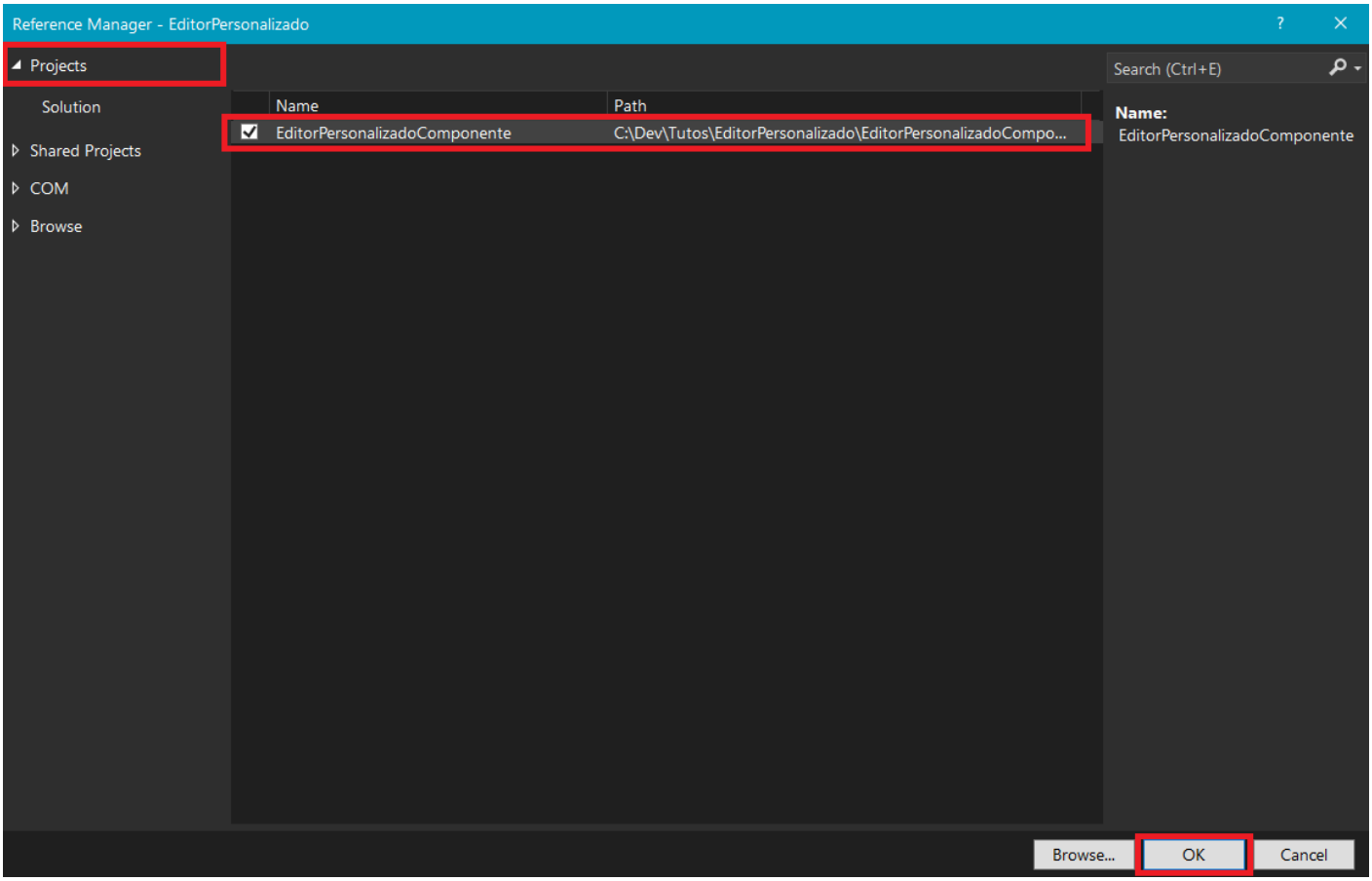
El archivo `_Imports.razor` es un archivo de declaración de importaciones de librería global. Este se encuentra en el directorio raíz del proyecto, y todas las importaciones declaradas aquí son válidas para todos los componentes definidos en la raíz y todos los subdirectorios del proyecto.

Con todo esto ya tenemos la base para nuestro componente personalizado.

Usar nuestro componente personalizado

Una vez hemos creado el componente base, ya estamos listos para mostrarlo. Procedamos ahora a importar la librería en el proyecto *EditorPersonalizado*. Selecciona el nodo `Dependencias` del proyecto *EditorPersonalizado,* haz clic derecho y selecciona la opción `Añadir referencia de proyecto`. En la nueva ventana asegúrate que la pestaña `Proyectos` esté visible y marca la casilla

correspondiente al proyecto `EditorPersonalizadoComponente`. Para finalizar pulsa el botón OK.



Esto añadirá una referencia al proyecto *EditorPersonalizadoComponente* de tal manera que podremos acceder a todas sus clases y componentes desde el proyecto *EditorPersonalizado*.

Modifica el archivo `Pages/Index.razor` para que quede así:

```
@page "/"

<PageTitle>Index</PageTitle>

<h1>Editor de texto personalizado</h1>

<EditorDeTexto />
```

Como ves, gracias a la gestión automática de los módulos javascript no ha sido necesaria ninguna referencia a ninguna librería javascript en el proyecto Blazor. Una simple llamada al componente <EditorDeTexto /> es suficiente.

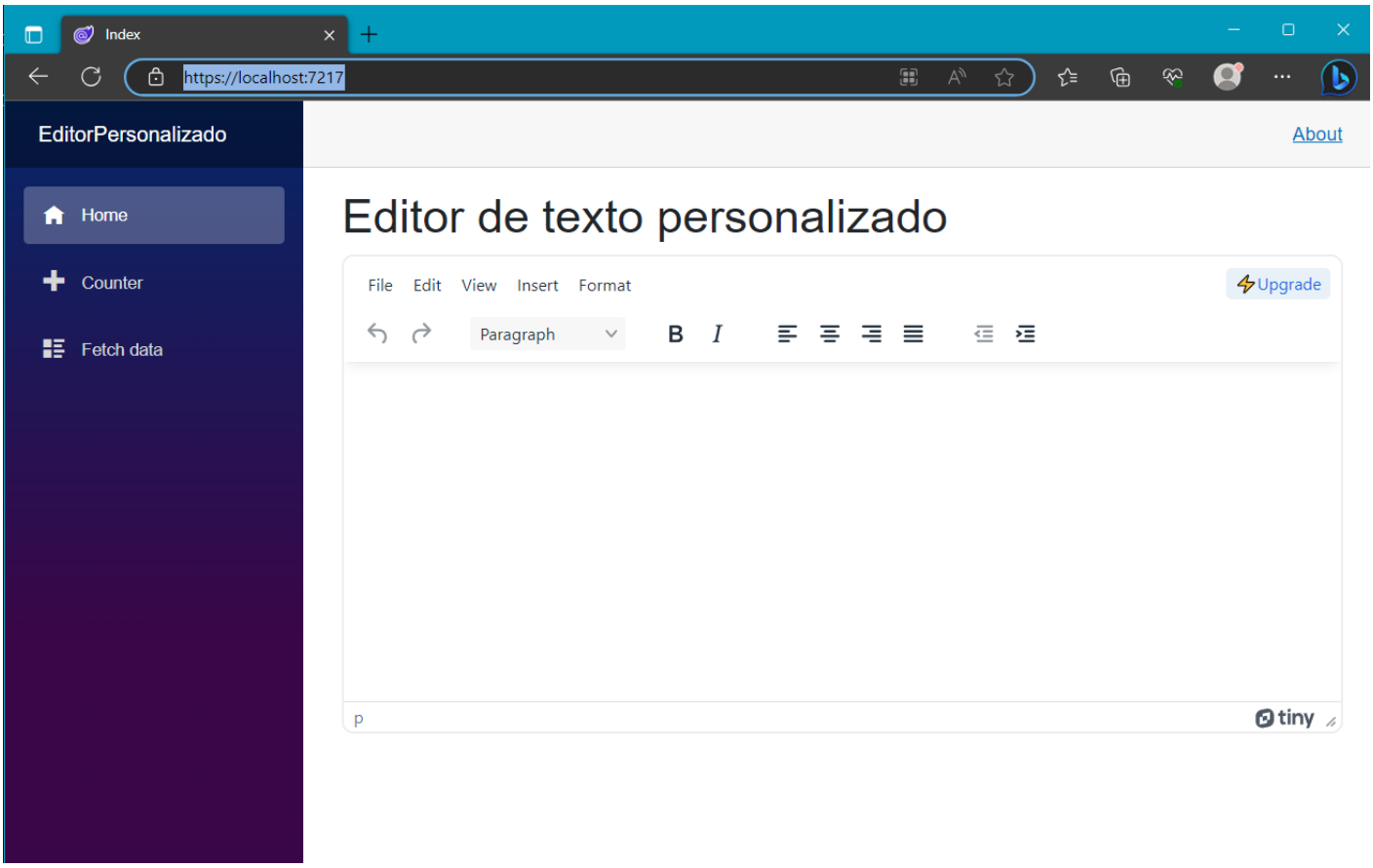
Modifica el archivo `_Imports.razor` y añade la siguiente línea para hacer referencia al nuevo componente:

```
@using EditorPersonalizadoComponente
```

El archivo quedará así:

```
@using System.Net.Http
@using Microsoft.AspNetCore.Authorization
@using Microsoft.AspNetCore.Components.Authorization
@using Microsoft.AspNetCore.Components.Forms
@using Microsoft.AspNetCore.Components.Routing
@using Microsoft.AspNetCore.Components.Web
@using Microsoft.AspNetCore.Components.Web.Virtualization
@using Microsoft.JSInterop
@using EditorPersonalizado
@using EditorPersonalizado.Shared
@using EditorPersonalizadoComponente
```

Ya solo nos queda probar. Pulsa F5 para ejecutar el proyecto. Deberías ver algo similar a esto:



Resumen

En esta primera parte, hemos aprendido a crear un proyecto para albergar una librería de clases y componenes de Razor. Hemos creado un componente que sirve para envolver un editor TinyMCE y hemos visto una técnica para la carga de las dependencias javascript de manera transparente y

automática.

También hemos visto cómo podemos incrustar los archivos necesarios dentro de la librería para que estén disponibles de una forma sencilla para el proyecto consumidor. Como has visto ninguna referencia a la librería TinyMCE ha sido necesaria.

En el siguiente apartado crearemos un componente que usaremos como diálogo personalizado en TinyMCE. También desarrollaremos una técnica para configurar TinyMCE desde Blazor.

Revisión #4
Creado 3 julio 2023 06:47:07 por Guillermo
Actualizado 31 julio 2023 11:34:51 por Guillermo