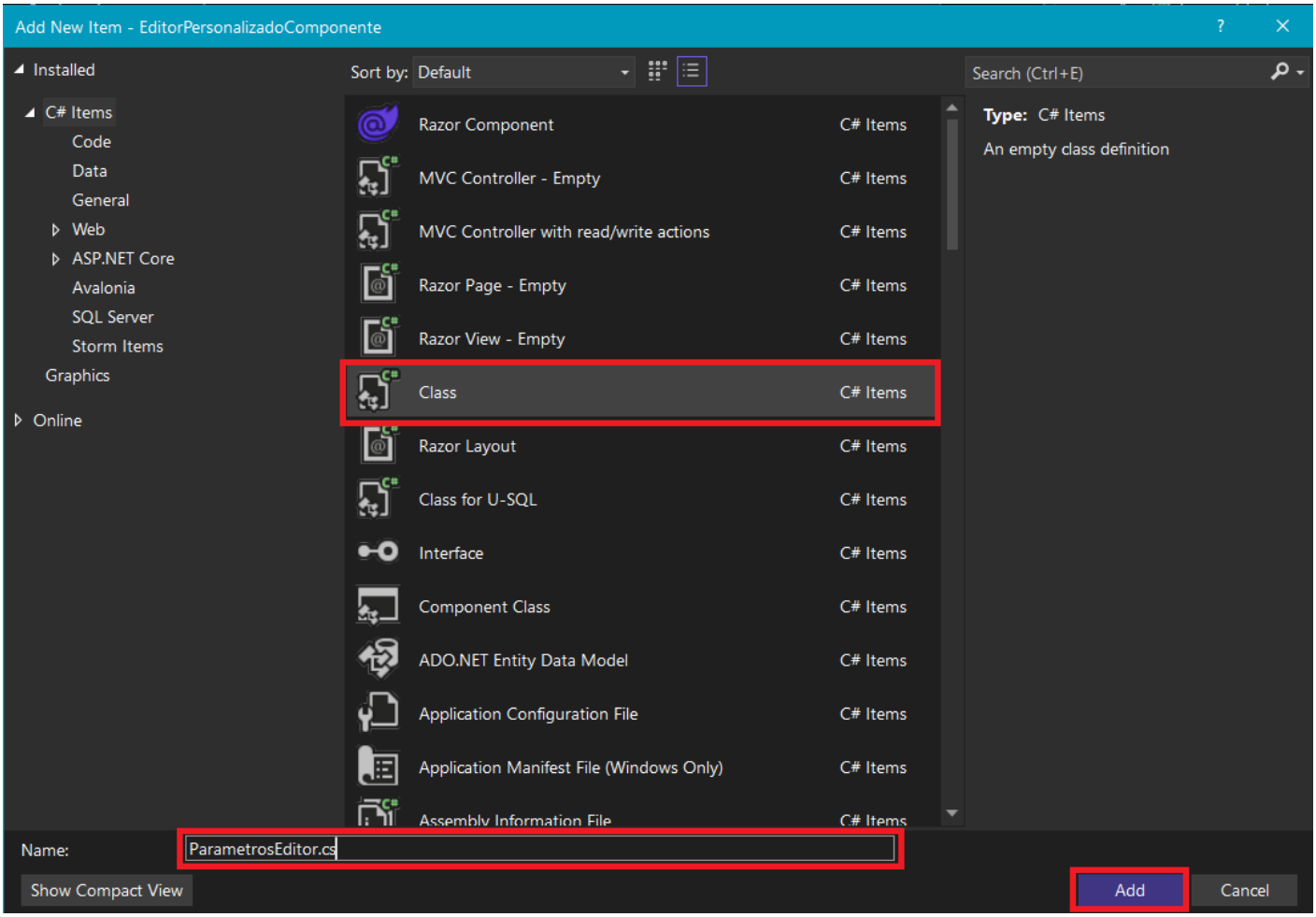
Uso de un componente Blazor en una ventana de diálogo TinyMCE

En esta segunda parte del tutorial vamos a crear un componente de Blazor que usaremos en TinyMCE como diálogo personalizado. También veremos un método para gestionar la configuración de TinyMCE desde el exterior del componente.

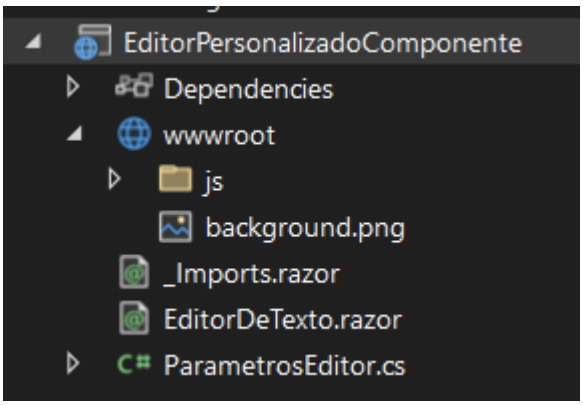
Configuración del componente TinyMCE

TinyMCE es un editor de texto complejo. Este tiene muchas opciones configurables, que deben ser provistas durante la creación del control. Para ello se debe pasar al constructor de TinyMCE un objeto con las propiedades y sus valores. Todo ello ocurre al nivel de javascript. Nosotros estamos usando TinyMCE incrustado en un componente Blazor, con lo que tendremos que configurar TinyMCE a través de Blazor y C#.

Para empezar vamos a crear una nueva clase dentro del proyecto *EditorPersonalizadoComponente*. Haz clic derecho en la raíz del proyecto y selecciona `Añadir > Nuevo elemento`. En la ventana que se abre selecciona el tipo de elemento `Clase`, como nombre, la llamaremos *ParametrosEditor.cs* y pulsa `Añadir` para finalizar.



Después de crear la nueva clase el proyecto EditorPersonalizadoComponente debería quedar así:



Escribe el siguiente código en *ParametrosEditor.cs*:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace EditorPersonalizadoComponente
```

```
{
    public class ParametrosEditor
    {
      public ParametrosEditor(string selector)
        {
            Selector = selector;
        }

        public string Selector { get; set; }

        [JsonPropertyName("external_plugins")]
        public Dictionary<string,string> ExternalPlugins { get; set; } = new
Dictionary<string, string>();
    }
}
```

El código anterior define una serie de propiedades en C#. Estas propiedades las usaremos para configurar el editor TinyMCE. Lo siguiente es modificar el archivo `wwwroot/editorDeTexto.js` para adaptarlo de forma que pueda gestionar los parámetros. Escribe el siguiente código:

```
export function initEditor(configuration) {

    tinymce.init({
        selector: configuration.selector,
        base_url: "./_content/EditorPersonalizadoComponente/js/tinymce",
        suffix: ".min"
    });
}
```

Ahora adaptaremos el componente *EditorDeTexto.razor* para que acepte un parámetro definiendo el objeto de configuración. Modifica el código existente del siguiente modo:

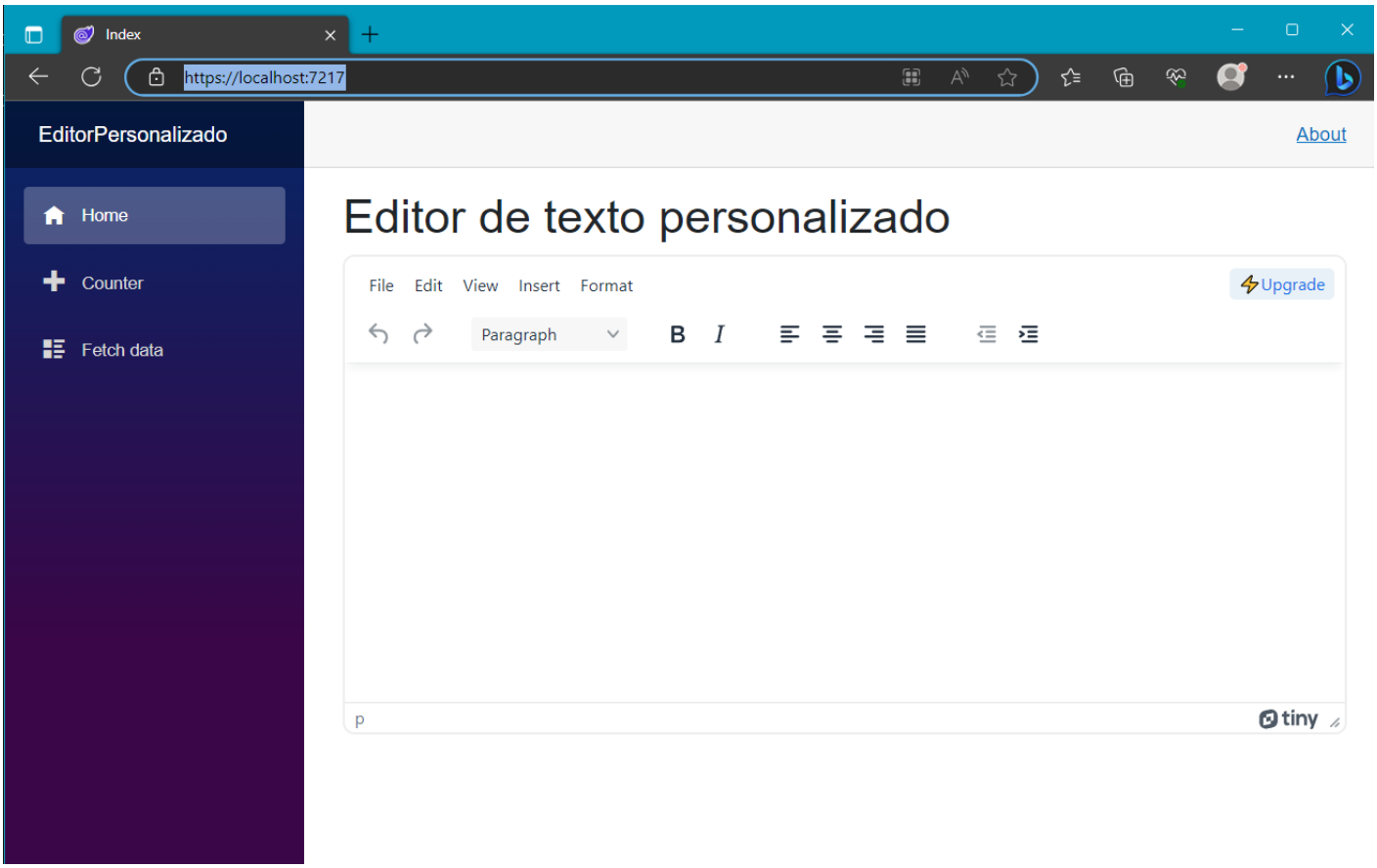
```
@inject IJSRuntime _jsruntime

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

@code {
    [Parameter] public ParametrosEditor Parametros { get; set; }

    private Lazy<Task<IJSObjectReference>> moduleTask;
```

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

        var module = await moduleTask.Value;
        await module.InvokeVoidAsync("initEditor", Parametros);
    }
}
```

Por último solo nos queda modificar la página *Index.razor* en el proyecto *EditorPersonalizado*, en el que vamos a adaptar la llamada al componente TinyMCE para proporcionar los parámetros. Modifica el código como sigue:

```
@page "/"

<PageTitle>Index</PageTitle>

<h1>Editor de texto personalizado</h1>

<EditorDeTexto Parametros="_parametrosEditor" />

@code {
    private ParametrosEditor _parametrosEditor;

    protected override void OnInitialized()
    {
        _parametrosEditor = new ParametrosEditor("#editor");
    }
}
```

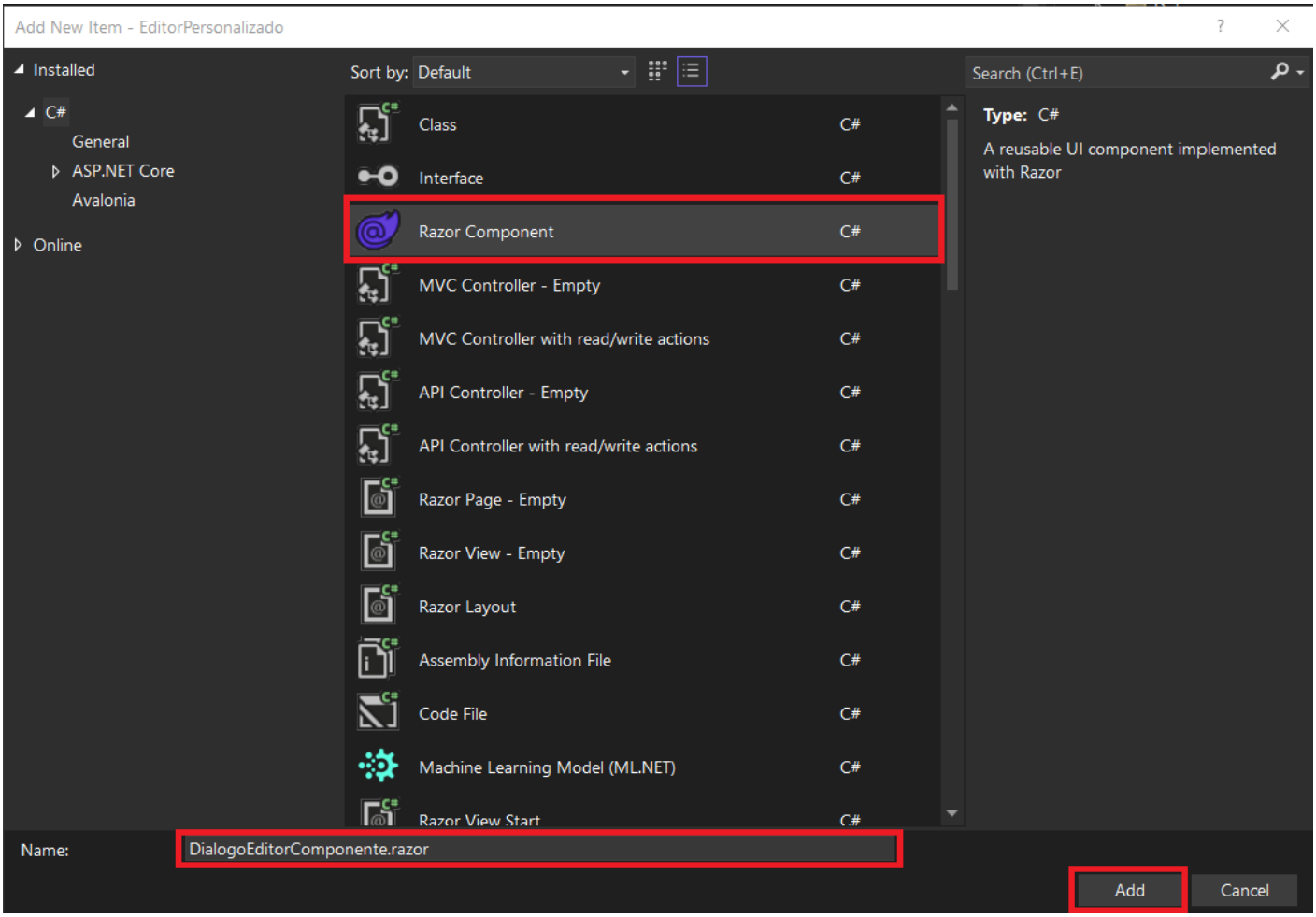
Pulsa F5 para ejecutar el proyecto. De hecho visualmente nada ha cambiado, pues sigues viendo lo mismo que en la sección anterior, pero sí, hemos cambiado el modo en el que configuramos el editor TinyMCE. Este es el resultado:



Uso de un componente de Blazor para diálogo en TinyMCE

La API de TinyMCE ofrece un ámplio abanico de posibilidades de personalización y extensión. Una de estas es la creación de ventanas de diálogo personalizadas. TinyMCE ofrece dos tipos: ventanas de diálogo definidas en TinyMCE o ventanas de diálogo que renderizan una URL. Dado que nosotros trabajamos con Blazor, según qué operaciones queramos realizar, las que renderizan una URL son más prácticas. Imagina por ejemplo un navegador de recursos (una galería de imágenes) es más sencilla de desarrollar desde Blazor que usando funciones para crear un formulario, eventos para leer y escribir en el servidor a través de una API, etc. Por ello, en este tutorial nos vamos a focalizar en la API para crear ventanas de diálogo que renderizan una URL en TinyMCE.

Lo primero de todo es añadir un nuevo componente al proyecto *EditorPersonalizado*. Haz clic derecho sobre la carpeta `Shared`. En el menu contextual selecciona `Añadir > Componente de Razor`. En la ventana de opciones, pondremos el nombre *DialogoEditorComponente.razor* y pulsamos `Añadir` para terminar.



Escribe el siguiente código dentro de *DialogoEditorComponente.razor*:

```
<h3>Dialogo personalizado en Blazor para TinyMCE</h3>

@code {

}
```

Otro elemento necesario para renderizar nuestro nuevo componente correctamente, es la plantilla de base. Toda página de Blazor hereda una plantilla que usa para renderizar el contenido. En nuestro caso, como deseamos renderizar un componente sin ningún otro contenido, nos conviene crear una plantilla vacía para renderizar solamente nuestro componente. Haz clic derecho en la carpeta `Shared`, añade un nuevo componente de Razor y llámalo *LightLayout.razor*. Escribe el siguiente código:

```
@inherits LayoutComponentBase

@Body
```

Si comparas la nueva plantilla con la plantilla por defecto *MainLayout.razor*, podrás observar la diferencia. Nuestra plantilla solo define el marcador `@Body`, que sirve para insertar el contenido del

componente que la usa. En *MainLayout.razor* hay más código que sirve para maquetar el contenido de una forma más precisa.

Ahora vamos a crear una página para renderizar el componente diálogo, de modo que sea accesible a través de una URL. Dentro de la carpeta `Pages`, crea un nuevo Componente de Razor y llámalo *DialogoEditorPagina.razor*. Escribe el siguiente código:

```
@page "/editor-dialog"
@layout LightLayout

<DialogoEditorComponente />
```

En el código anterior hemos definido una nueva página con la URL /editor-dialog que renderiza el componente que creamos anteriormente. Además le indicamos al compilador que debe usar la nueva plantilla que creamos, LightLayout.razor. De este modo nos aseguramos que en la página no habrá más que el componente, sin ningún otro contenido superfluo.

El siguiente paso consiste en crear un archivo javascript con el que definiremos un plugin para TinyMCE. En este plugin, definiremos las opciones que permitan renderizar nuestro componente diálogo en una ventana diálogo de TinyMCE. En el proyecto *EditorPersonalizado*, dentro de la carpeta `wwwroot/js` crea un archivo javascript llamado *editorDialogoPlugin.js* y escribe el siguiente código:

```
tinymce.PluginManager.add('editorDialogo', (editor, url) => {
    const openDialog = () => editor.windowManager.openUrl({
        title: 'Dialogo personalizado',
        url: '/editor-dialog'
    });

    editor.ui.registry.addButton('editorDialogo', {
        icon: 'sharpen',
        tooltip: "Este boton abre un componente de Blazor dentro de una ventana de dialogo de
TinyMCE",
        onAction: () => {
            openDialog();
        }
    });

    return {
        getMetadata: () => ({
            name: 'Ventana de dialogo personalizada',
            url: 'https://ejemplo.ej'
```

```
        })
    };
});
```

En la definición del plugin usamos la API que TinyMCE pode a disposición para registrar un nuevo plugin. Del mismo modo definimos un botón para la barra de herramientas. La función `getMetadata()` puede ser usada por el plugin `help` para mostrar información acerca de los plugins en uso.

Para que TinyMCE pueda usar nuestro nuevo plugin, debemos modificar los parámetros de TinyMCE, la función de inicialización del editor y la página donde usamos el componente de TinyMCE. Empecemos por modificar el archivo ParametrosEditor.cs del proyecto *EditorPersonalizadoComponente* y añade una nueva propiedad:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Text.Json.Serialization;
using System.Threading.Tasks;

namespace EditorPersonalizadoComponente
{
    public class ParametrosEditor
    {
        public ParametrosEditor(string selector)
        {
            Selector = selector;
        }

        public string Selector { get; set; }

        public string Toolbar { get; set; } = "undo redo | blocks | bold italic | alignleft
aligncenter alignright alignjustify | outdent indent";

        [JsonPropertyName("external_plugins")]
        public Dictionary<string, string> ExternalPlugins { get; set; } = new
Dictionary<string, string>();
    }
}
```

La propiedad que hemos añadido sirve para configurar la barra de herramientas del editor TinyMCE. El valor por defecto corresponde con la configuración por defecto de TinyMCE. Esta configuración nos sirve de base para configurar la barra de herramientas.

Continuemos ahora con el archivo *editorDeTexto.js* del proyecto *EditorPersonalizadoComponente*, el código debe quedar así:

```
export function initEditor(configuration) {

    var defaultOpts = tinymce.defaultOptions;

    defaultOpts.selector = configuration.selector;
    defaultOpts.base_url = "./_content/EditorPersonalizadoComponente/js/tinymce";
    defaultOpts.suffix = ".min";
    defaultOpts.toolbar = configuration.toolbar;

    var extPlugCnt = Object.keys(configuration.external_plugins).length;

    if (extPlugCnt > 0) {
        var plugins = {};

        for (let property in configuration.external_plugins) {
            plugins[property] = configuration.external_plugins[property];
        }

        defaultOpts.external_plugins = plugins;
    }

    tinymce.init(defaultOpts);
}
```

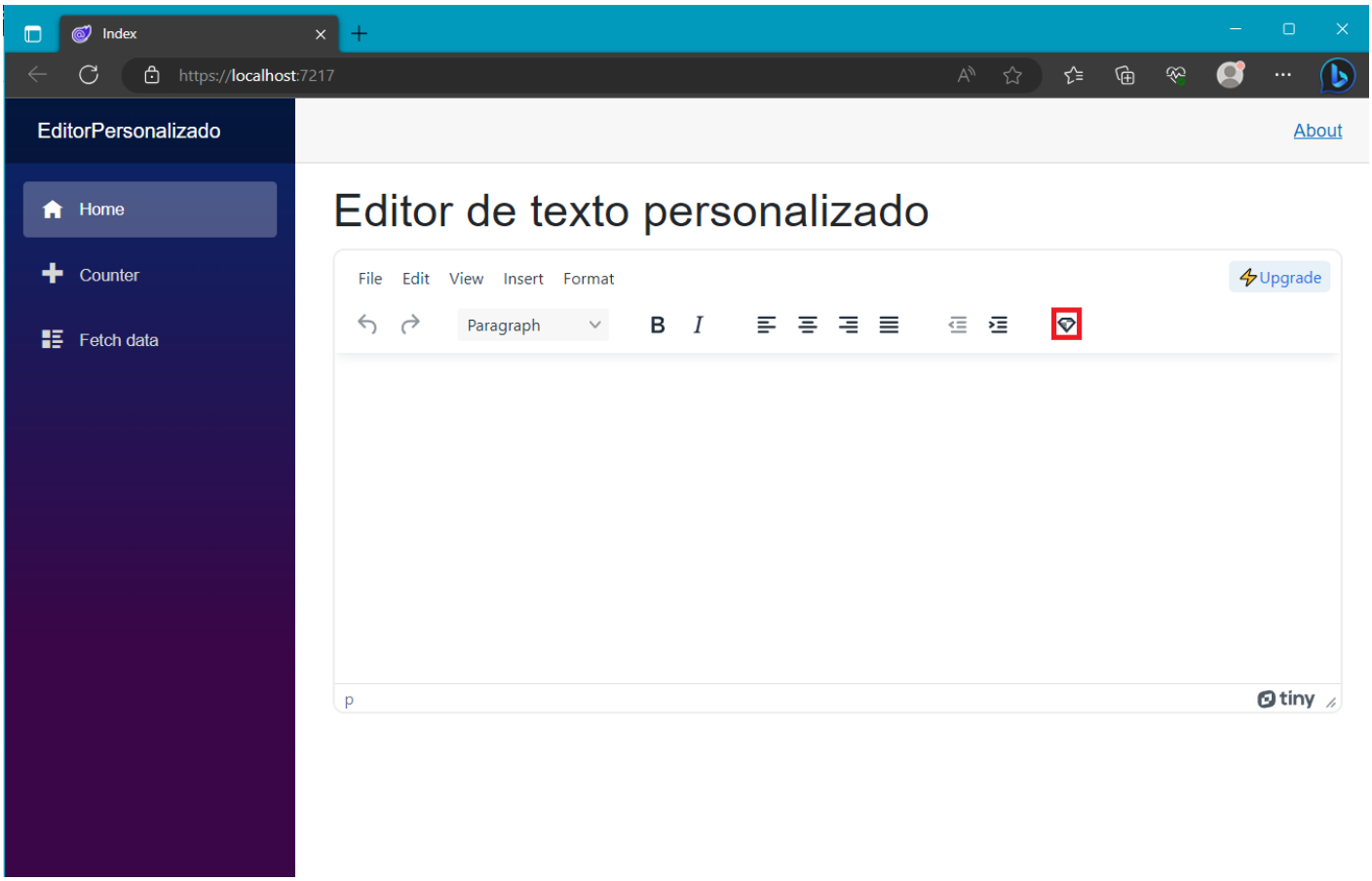
En esta nueva versión del inicializador de TinyMCE hemos cambiado la estrategia para definir los parámetros. Ahora recuperamos primero los valores por defecto de TinyMCE y modificamos los parámetros necesarios. De este modo podemos añadir una cantidad arbitraria de plugins externos. Una vez configuradas las opciones, las usamos para inicializar TinyMCE.

Por último solo nos queda modificar la página *Index.razor* en el proyecto *EditorPersonalizado*, en el que vamos a adaptar la llamada al componente TinyMCE para proporcionar los nuevos parámetros. Modifica el código como sigue:

```
@page "/"
```

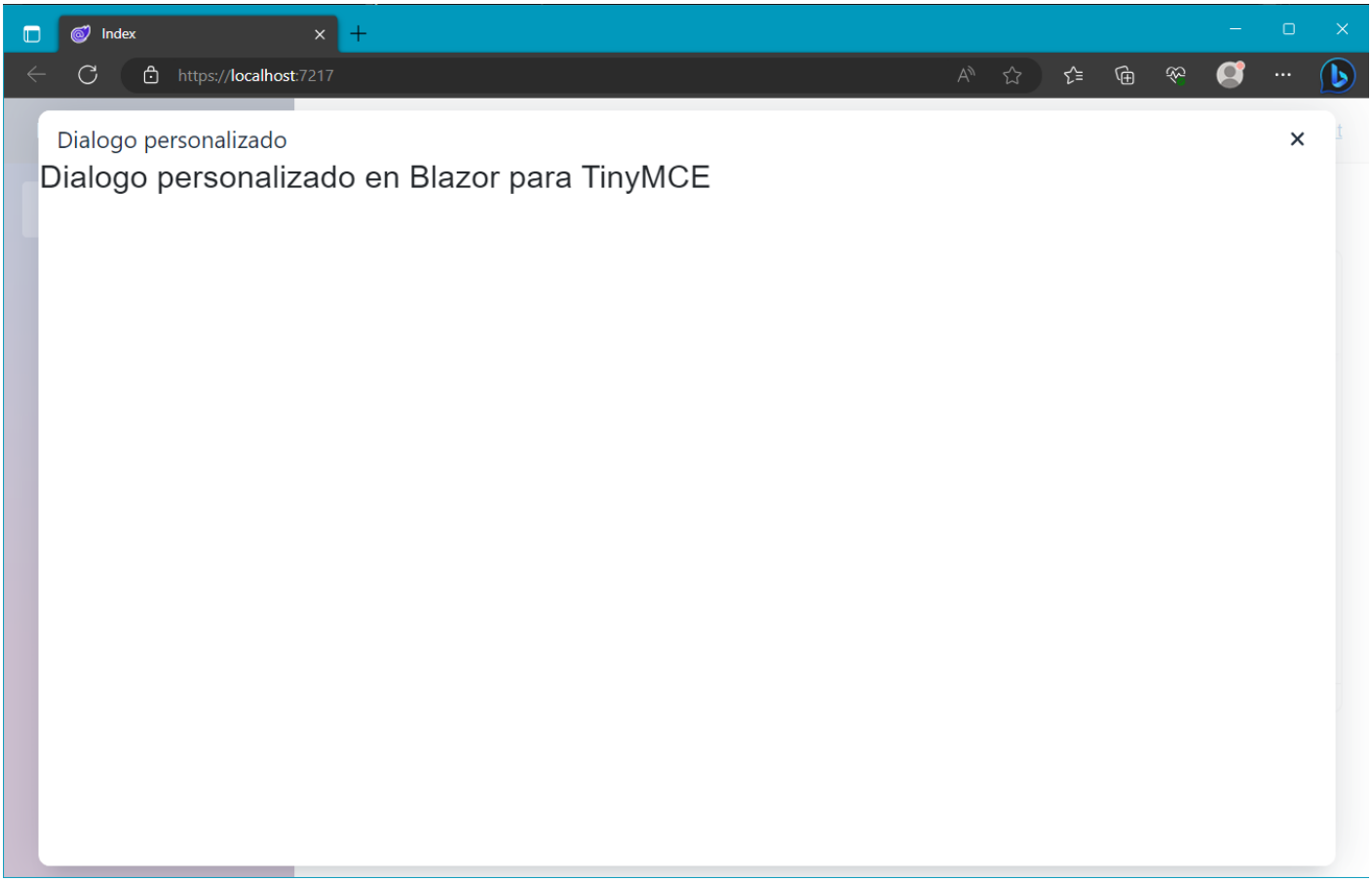
```
<PageTitle>Index</PageTitle>

<h1>Editor de texto personalizado</h1>

<EditorDeTexto Parametros="_parametrosEditor" />

@code {
    private ParametrosEditor _parametrosEditor;

    protected override void OnInitialized()
    {
        _parametrosEditor = new ParametrosEditor("#editor");
        _parametrosEditor.Toolbar += " | editorDialogo";
        _parametrosEditor.ExternalPlugins.Add("editorDialogo", "/js/editorDialogoPlugin.js");
    }
}
```

Ya estamos listos. Pulsa F5 para ejecutar el proyecto. Ahora deberías ver algo así:



Si observas, en la barra de herramientas hay un nuevo botón. En la definición del plugin, cuando definimos el botón de la barra de herramientas, definimos un icono llamado sharpen, este, corresponde con un diamante. Nuestro nuevo botón tiene un icono de diamante, haz clic y podrás

ver tu componente Blazor como contenido de una ventana de diálogo de TinyMCE.



Comunicación entre el componente Blazor y TinyMCE

Ahora que podemos usar un componente de Blazor en nuestro editor TinyMCE, necesitamos un mecanismo de comunicación entre ambos. Para ello haremos uso de la función `window.parent.postMessage()`. Esta función permite enviar mensajes dentro de la misma ventana. Estos mensajes pueden ser leídos por cualquier elemento dentro de la misma página. TinyMCE implementa un mecanismo que lee los mensajes y busca aquellos que tengan un formato convenido, que permite que podamos ejecutar acciones en el editor desde la ventana de diálogo. Esto es así porque, desde la ventana de diálogo no tenemos acceso a la instancia de TinyMCE, ya que se trata de una página externa y, el único elemento común es el elemento `window` de javascript.

Vamos a implementar el mecanismo de comunicación entre el componente Blazor y TinyMCE. Crea un nuevo archivo dentro del proyecto *EditorPersonalizado*, en la carpeta `wwwroot/js` llamado *dialogoEditorComponente.js* y escribe el siguiente código:

```
export function sendMessage(message) {
    window.parent.postMessage({
```

```
        mceAction: 'insertContent',
        content: message
    }, '*');

    window.parent.postMessage({
        mceAction: 'close'
    }, '*');
}
```

En el código anterior definimos una función que envía un mensaje por el bus interno de la ventana de javascript. El valor del mensaje es un objeto, definido por TinyMCE y que debe ser respetado para que sea tenido en cuenta. Las acciones están definidas en la API de TinyMCE. Por ejemplo `insertContent` añade el contenido pasado en `content` donde se encuentra el cursor en el editor. La acción `close`, cierra la ventana de diálogo.

> Para que conozcas más acerca la API para ventanas de diálogo puede visitar la documentación oficial.

El siguiente paso consiste en modificar el componente en sí mismo para añadir un poco más de funcionalidad que permita probar el envío de mensajes. Modifica el código de DialogoEditorComponente.razor como sigue:

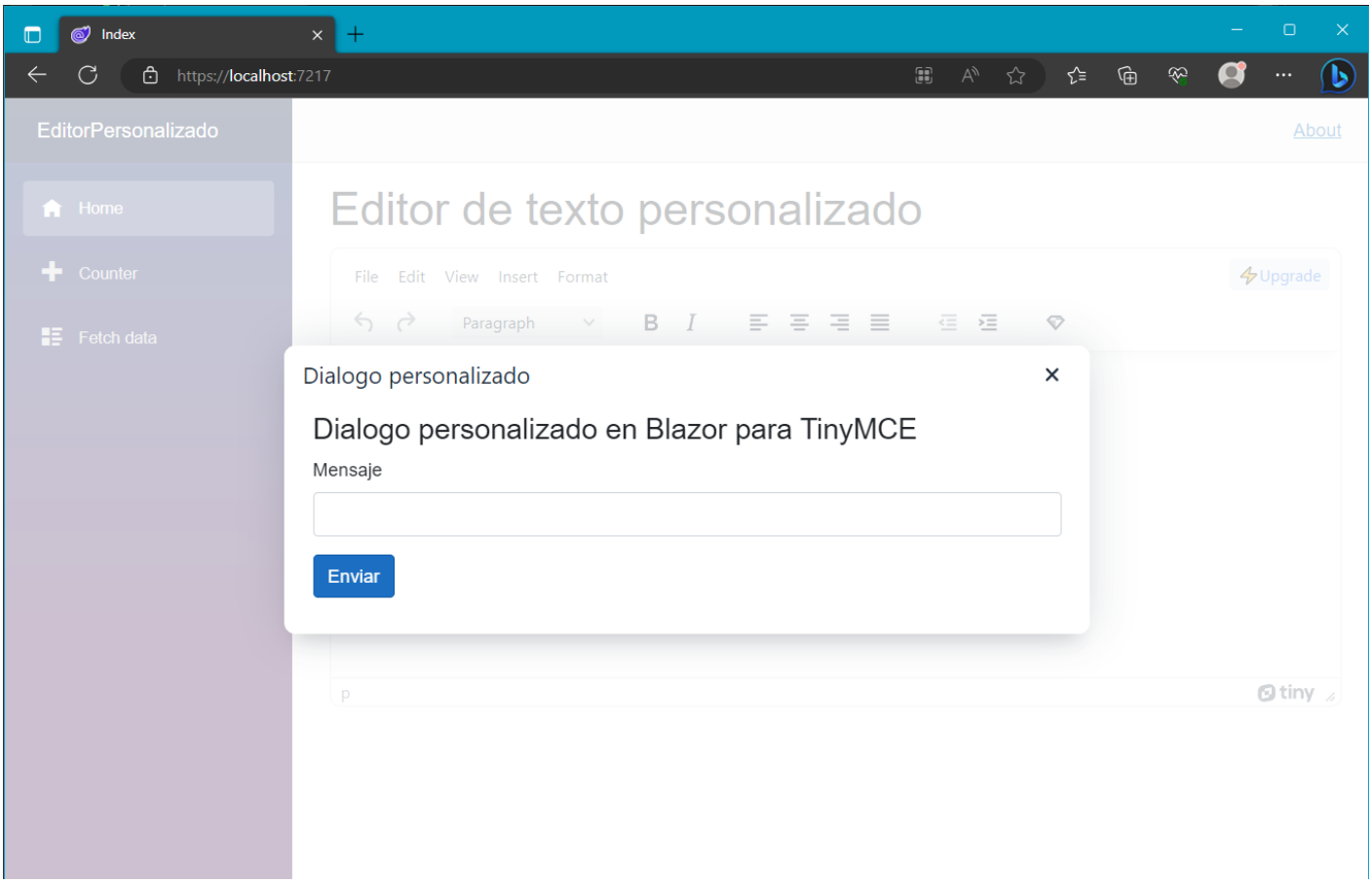
```
@inject IJSRuntime _jsruntime

<div class="mt-3 px-4">
    <h3>Dialogo personalizado en Blazor para TinyMCE</h3>

    <label for="message" class="form-label">Mensaje</label>
    <input type="text" id="message" class="form-control" @bind-value="_messageValue" />
    <button type="button" class="btn btn-primary mt-3" @onclick="OnSend">Enviar</button>
</div>

@code {
    private Lazy<Task<IJSObjectReference>> moduleTask;
    private string _messageValue = "";

    protected override void OnAfterRender(bool firstRender)
    {
        moduleTask = new(() => _jsruntime.InvokeAsync<IJSObjectReference>("import",
"./js/dialogoEditorComponente.js").AsTask());
```

```
    }

    private async void OnSend()
    {
        var module = await moduleTask.Value;
        await module.InvokeVoidAsync("sendMessage", _messageValue);
    }
}
```

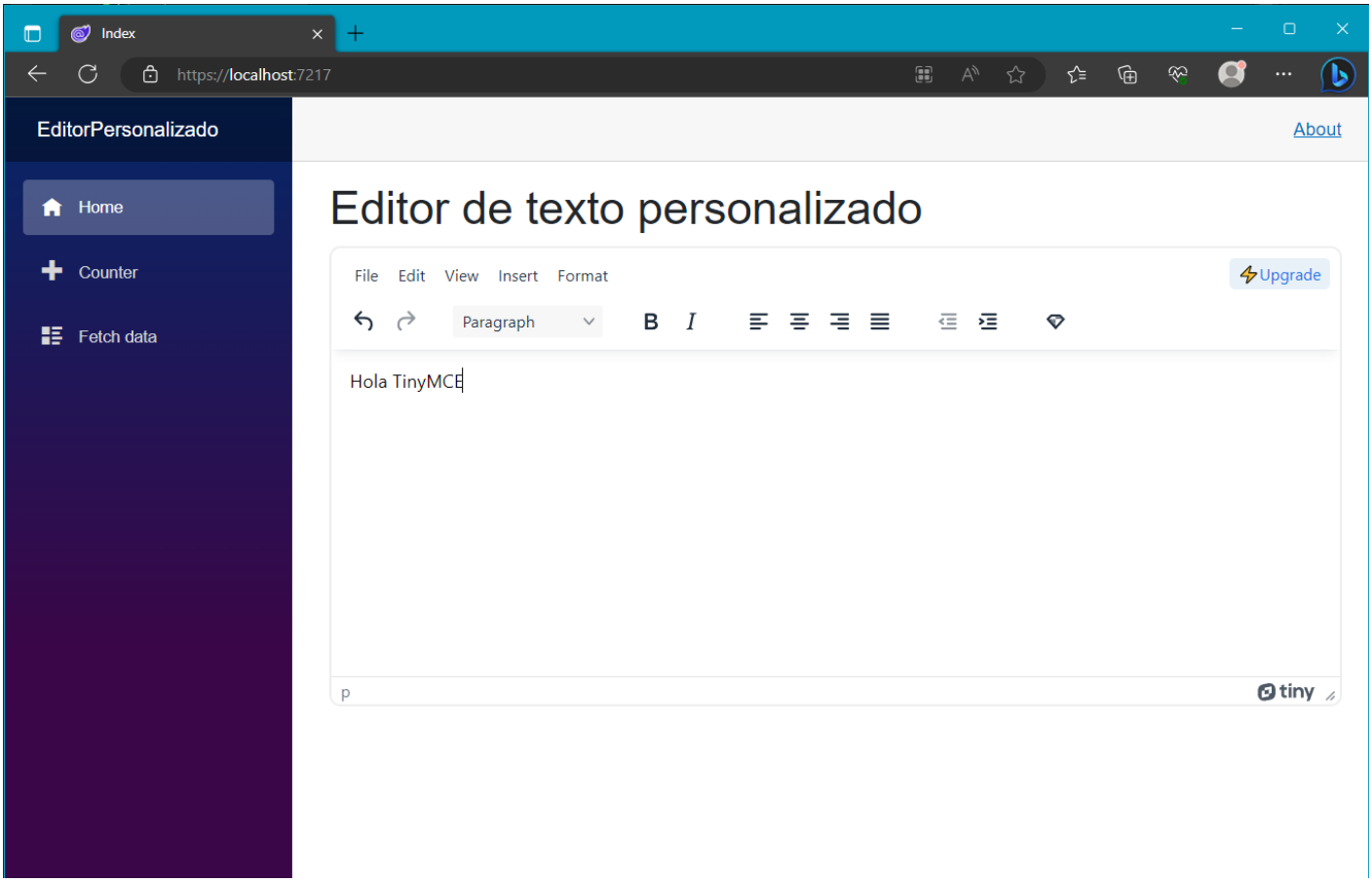
Por último, vamos a ajustar un poco el tamaño de la ventana del plugin ya que, se ve un poco grande. Modifica el archivo *editorDialogoPlugin.js* y añade las propiedades para el ancho y el largo:

```
tinymce.PluginManager.add('editorDialogo', (editor, url) => {
    const openDialog = () => editor.windowManager.openUrl({
        title: 'Dialogo personalizado',
        url: '/editor-dialog',
        width: 700,
        height: 250
    });

    editor.ui.registry.addButton('editorDialogo', {
        icon: 'sharpen',
        tooltip: "Este boton abre un componente de Blazor dentro de una ventana de dialogo de
TinyMCE",
        onAction: () => {
            openDialog();
        }
    });

    return {
        getMetadata: () => ({
            name: 'Ventana de dialogo personalizada',
            url: 'https://ejemplo.ej'
        })
    };
});
```

Todo listo. Pulsa F5 para arrancar el proyecto y abre la ventana de diálogo. Ahora tiene un nuevo aspecto:



Prueba a escribir un mensaje, por ejemplo *Hola TinyMCE* y pulsa el botón enviar. La ventana de diálogo se cerrará automáticamente y el mensaje aparecerá en el editor:



Resumen

En esta segunda parte hemos aprendido a crear un componente de Blazor y a usarlo como base para construir una ventana de diálogo personalizada de TinyMCE. También hemos aprendido a gestionar de un modo diferente los parámetros de inicialización de TinyMCE desde Blazor. Finalmente hemos visto el mecanismo de comunicación entre el componente Blazor y el editor TinyMCE que permite ejecutar acciones en el editor desde el componente Blazor.

Revisión #2
Creado 4 julio 2023 08:06:58 por Guillermo
Actualizado 31 julio 2023 11:35:54 por Guillermo