


Customizing a WebView

03/31/2020 • 14 minutes to read • 

In this article

[Create the HybridWebView](#)

[Consume the HybridWebView](#)

[Create the custom renderer on each platform](#)

[Related links](#)



[Download the sample](#)

A `Xamarin.Forms WebView` is a view that displays web and HTML content in your app. This article explains how to create a custom renderer that extends the `WebView` to allow C# code to be invoked from JavaScript.

Every `Xamarin.Forms` view has an accompanying renderer for each platform that creates an instance of a native control. When a `WebView` is rendered by a `Xamarin.Forms` application on iOS, the `WkWebViewRenderer` class is instantiated, which in turn instantiates a native `WkWebView` control. On the Android platform, the `WebViewRenderer` class instantiates a native `WebView` control. On the Universal Windows Platform (UWP), the `WebViewRenderer` class instantiates a native `WebView` control. For more information about the renderer and native control classes that `Xamarin.Forms` controls map to, see [Renderer Base Classes and Native Controls](#).

The following diagram illustrates the relationship between the `View` and the corresponding native controls that implement it:



WebView



The rendering process can be used to implement platform customizations by creating a custom renderer for a [WebView](#) on each platform. The process for doing this is as follows:

1. [Create](#) the `HybridWebView` custom control.
2. [Consume](#) the `HybridWebView` from `Xamarin.Forms`.
3. [Create](#) the custom renderer for the `HybridWebView` on each platform.

Each item will now be discussed in turn to implement a `HybridWebView` renderer that enhances the `Xamarin.Forms WebView` to allow C# code to be invoked from JavaScript. The `HybridWebView` instance will be used to display an HTML page that asks the user to enter their name. Then, when the user clicks an HTML button, a JavaScript function will invoke a C# Action that displays a pop-up containing the users name.

For more information about the process for invoking C# from JavaScript, see [Invoke C# from JavaScript](#). For more information about the HTML page, see [Create the Web Page](#).

ⓘ Note

A [WebView](#) can invoke a JavaScript function from C#, and return any result to the calling C# code. For more information, see [Invoking JavaScript](#).

Create the HybridWebView

The HybridWebView custom control can be created by subclassing the [WebView](#) class:

C#

 Copy

```
public class HybridWebView : WebView
{
    Action<string> action;

    public static readonly BindableProperty UriProperty = BindableProperty.Create(
        propertyName: "Uri",
        returnType: typeof(string),
        declaringType: typeof(HybridWebView),
        defaultValue: default(string));

    public string Uri
    {
        get { return (string)GetValue(UriProperty); }
        set { SetValue(UriProperty, value); }
    }

    public void RegisterAction(Action<string> callback)
    {
        action = callback;
    }

    public void Cleanup()
    {
        action = null;
    }

    public void InvokeAction(string data)
    {
        if (action == null || data == null)
        {

```

```
        return;  
    }  
    action.Invoke(data);  
}  
}
```

The `HybridWebView` custom control is created in the .NET Standard library project and defines the following API for the control:

- A `Uri` property that specifies the address of the web page to be loaded.
- A `RegisterAction` method that registers an `Action` with the control. The registered action will be invoked from JavaScript contained in the HTML file referenced through the `Uri` property.
- A `CleanUp` method that removes the reference to the registered `Action`.
- An `InvokeAction` method that invokes the registered `Action`. This method will be called from a custom renderer in each platform project.

Consume the HybridWebView

The `HybridWebView` custom control can be referenced in XAML in the .NET Standard library project by declaring a namespace for its location and using the namespace prefix on the custom control. The following code example shows how the `HybridWebView` custom control can be consumed by a XAML page:

XAML	 Copy
<pre><ContentPage ... xmlns:local="clr-namespace:CustomRenderer;assembly=CustomRenderer" x:Class="CustomRenderer.HybridWebViewPage" Padding="0,40,0,0"> <local:HybridWebView x:Name="hybridWebView" Uri="index.html" /> </ContentPage></pre>	

The `local` namespace prefix can be named anything. However, the `clr-namespace` and `assembly` values must match the details of the custom control. Once the namespace is declared, the prefix is used to reference the custom control.

The following code example shows how the `HybridWebView` custom control can be consumed by a C# page:

C#

 Copy

```
public HybridWebViewPageCS()
{
    var hybridWebView = new HybridWebView
    {
        Uri = "index.html"
    };
    // ...
    Padding = new Thickness(0, 40, 0, 0);
    Content = hybridWebView;
}
```

The `HybridWebView` instance will be used to display a native web control on each platform. It's `Uri` property is set to an HTML file that is stored in each platform project, and which will be displayed by the native web control. The rendered HTML asks the user to enter their name, with a JavaScript function invoking a C# Action in response to an HTML button click.

The `HybridWebViewPage` registers the action to be invoked from JavaScript, as shown in the following code example:

C#

 Copy

```
public partial class HybridWebViewPage : ContentPage
{
    public HybridWebViewPage()
    {
        // ...
        hybridWebView.RegisterAction(data => DisplayAlert("Alert", "Hello " + data, "OK"));
    }
}
```

This action calls the [DisplayAlert](#) method to display a modal pop-up that presents the name entered in the HTML page displayed by the `HybridWebView` instance.

A custom renderer can now be added to each application project to enhance the platform web controls by allowing C# code to be invoked from JavaScript.

Create the custom renderer on each platform

The process for creating the custom renderer class is as follows:

1. Create a subclass of the `WkWebViewRenderer` class on iOS, and the `WebViewRenderer` class on Android and UWP, that renders the custom control.
2. Override the `OnElementChanged` method that renders the [WebView](#) and write logic to customize it. This method is called when a `HybridWebView` object is created.
3. Add an `ExportRenderer` attribute to the custom renderer class or *AssemblyInfo.cs*, to specify that it will be used to render the `Xamarin.Forms` custom control. This attribute is used to register the custom renderer with `Xamarin.Forms`.

ⓘ Note

For most `Xamarin.Forms` elements, it is optional to provide a custom renderer in each platform project. If a custom renderer isn't registered, then the default renderer for the control's base class will be used. However, custom renderers are required in each platform project when rendering a [View](#) element.

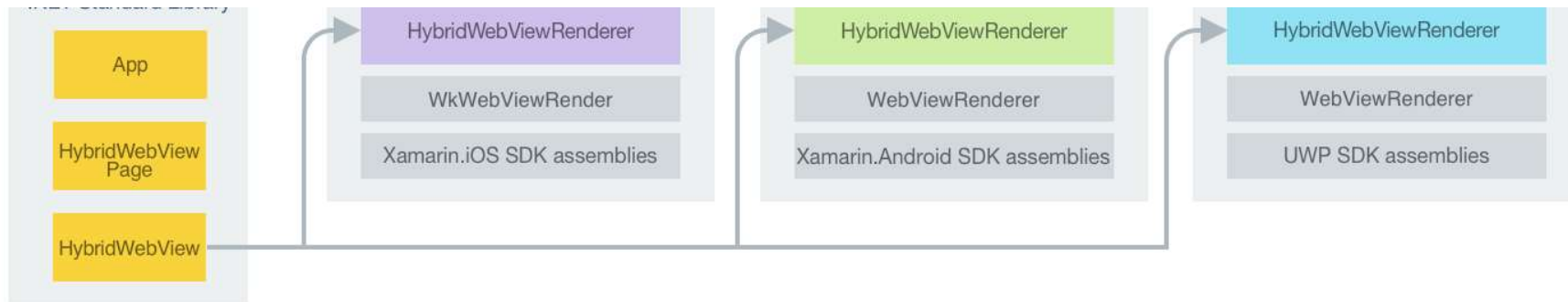
The following diagram illustrates the responsibilities of each project in the sample application, along with the relationships between them:

CustomRenderer
.NET Standard Library

CustomRenderer.iOS

CustomRenderer.Droid

CustomRenderer.UWP



The `HybridWebView` custom control is rendered by platform renderer classes, which derive from the `WkWebViewRenderer` class on iOS, and from the `WebViewRenderer` class on Android and UWP. This results in each `HybridWebView` custom control being rendered with native web controls, as shown in the following screenshots:



The `WkWebViewRenderer` and `WebViewRenderer` classes expose the `onElementChanged` method, which is called when the Xamarin.Forms custom control is created to render the corresponding native web control. This method takes a

`VisualElementChangedEventArgs` parameter that contains `OldElement` and `NewElement` properties. These properties represent the `Xamarin.Forms` element that the renderer *was* attached to, and the `Xamarin.Forms` element that the renderer *is* attached to, respectively. In the sample application the `OldElement` property will be `null` and the `NewElement` property will contain a reference to the `HybridWebView` instance.

An overridden version of the `OnElementChanged` method, in each platform renderer class, is the place to perform the native web control customization. A reference to the `Xamarin.Forms` control that's being rendered can be obtained through the `Element` property.

Each custom renderer class is decorated with an `ExportRenderer` attribute that registers the renderer with `Xamarin.Forms`. The attribute takes two parameters – the type name of the `Xamarin.Forms` custom control being rendered, and the type name of the custom renderer. The `assembly` prefix to the attribute specifies that the attribute applies to the entire assembly.

The following sections discuss the structure of the web page loaded by each native web control, the process for invoking C# from JavaScript, and the implementation of this in each platform custom renderer class.

Create the web page

The following code example shows the web page that will be displayed by the `HybridWebView` custom control:

HTML	 Copy
<pre><html> <body> <script src="http://code.jquery.com/jquery-2.1.4.min.js"></script> <h1>HybridWebView Test</h1>
 Enter name: <input type="text" id="name">

 <button type="button" onclick="javascript: invokeCSCode(\$('#name').val());">Invoke C# Code</button>
 <p id="result">Result:</p></pre>	


```
<script type="text/javascript">function log(str) {
    $('#result').text($('#result').text() + " " + str);
}

function invokeCSCode(data) {
    try {
        log("Sending Data:" + data);
        invokeCSharpAction(data);
    }
    catch (err) {
        log(err);
    }
}
</script>
</body>
</html>
```

The web page allows a user to enter their name in an `input` element, and provides a `button` element that will invoke C# code when clicked. The process for achieving this is as follows:

- When the user clicks on the `button` element, the `invokeCSCode` JavaScript function is called, with the value of the `input` element being passed to the function.
- The `invokeCSCode` function calls the `log` function to display the data it is sending to the C# `Action`. It then calls the `invokeCSharpAction` method to invoke the C# `Action`, passing the parameter received from the `input` element.

The `invokeCSharpAction` JavaScript function is not defined in the web page, and will be injected into it by each custom renderer.

On iOS, this HTML file resides in the `Content` folder of the platform project, with a build action of **BundleResource**. On Android, this HTML file resides in the `Assets/Content` folder of the platform project, with a build action of **AndroidAsset**.

Invoke C# from JavaScript

The process for invoking C# from JavaScript is identical on each platform:

- The custom renderer creates a native web control and loads the HTML file specified by the `HybridWebView.Uri` property.

- The custom renderer creates a native web control and loads the HTML file specified by the `HybridWebView.Url` property.
- Once the web page is loaded, the custom renderer injects the `invokeCSharpAction` JavaScript function into the web page.
- When the user enters their name and clicks on the HTML `button` element, the `invokeCSCode` function is invoked, which in turn invokes the `invokeCSharpAction` function.
- The `invokeCSharpAction` function invokes a method in the custom renderer, which in turn invokes the `HybridWebView.InvokeAction` method.
- The `HybridWebView.InvokeAction` method invokes the registered `Action`.

The following sections will discuss how this process is implemented on each platform.

Create the custom renderer on iOS

The following code example shows the custom renderer for the iOS platform:

C#

 Copy

```
[assembly: ExportRenderer(typeof(HybridWebView), typeof(HybridWebViewRenderer))]  
namespace CustomRenderer.iOS  
{  
    public class HybridWebViewRenderer : WKWebViewRenderer, IWKScriptMessageHandler  
    {  
        const string JavaScriptFunction = "function invokeCSharpAction(data)  
{window.webkit.messageHandlers.invokeAction.postMessage(data);}";  
        WKUserContentController userController;  
  
        public HybridWebViewRenderer() : this(new WKWebViewConfiguration())  
        {  
        }  
  
        public HybridWebViewRenderer(WKWebViewConfiguration config) : base(config)  
        {  
            userController = config.UserContentController;  
            var script = new WKUserScript(new NSString(JavaScriptFunction),  
            WKUserScriptInjectionTime.AtDocumentEnd, false);
```

```
        userController.AddUserScript(script);
        userController.AddScriptMessageHandler(this, "invokeAction");
    }

    protected override void OnElementChanged(VisualElementChangedEventArgs e)
    {
        base.OnElementChanged(e);

        if (e.OldElement != null)
        {
            userController.RemoveAllUserScripts();
            userController.RemoveScriptMessageHandler("invokeAction");
            HybridWebView hybridWebView = e.OldElement as HybridWebView;
            hybridWebView.Cleanup();
        }

        if (e.NewElement != null)
        {
            string filename = Path.Combine(NSBundle.MainBundle.BundlePath,
$"Content/{((HybridWebView)Element).Uri}");
            LoadRequest(new URLRequest(new NSURL(filename, false)));
        }
    }

    public void DidReceiveScriptMessage(WKUserContentController userContentController, WKScriptMessage
message)
    {
        ((HybridWebView)Element).InvokeAction(message.Body.ToString());
    }

    protected override void Dispose(bool disposing)
    {
        if (disposing)
        {
            ((HybridWebView)Element).Cleanup();
        }
        base.Dispose(disposing);
    }
}
```

The `HybridWebViewRenderer` class loads the web page specified in the `HybridWebView.Uri` property into a native [WKWebView](#) control, and the `invokeCSharpAction` JavaScript function is injected into the web page. Once the user enters their name and clicks the HTML button element, the `invokeCSharpAction` JavaScript function is executed, with the `DidReceiveScriptMessage` method being called after a message is received from the web page. In turn, this method invokes the `HybridWebView.InvokeAction` method, which will invoke the registered action to display the pop-up.

This functionality is achieved as follows:

- The renderer constructor creates a `WkWebViewConfiguration` object, and retrieves its [WKUserContentController](#) object. The `WkUserContentController` object allows posting messages and injecting user scripts into a web page.
- The renderer constructor creates a [WKUserScript](#) object, which injects the `invokeCSharpAction` JavaScript function into the web page after the web page is loaded.
- The renderer constructor calls the [WKUserContentController.AddUserScript](#) method to add the [WKUserScript](#) object to the content controller.
- The renderer constructor calls the [WKUserContentController.AddScriptMessageHandler](#) method to add a script message handler named `invokeAction` to the [WKUserContentController](#) object, which will cause the JavaScript function `window.webkit.messageHandlers.invokeAction.postMessage(data)` to be defined in all frames in all `webView` instances that use the `WkUserContentController` object.
- Provided that the custom renderer is attached to a new `Xamarin.Forms` element:
 - The [WKWebView.LoadRequest](#) method loads the HTML file that's specified by the `HybridWebView.Uri` property. The code specifies that the file is stored in the `Content` folder of the project. Once the web page is displayed, the `invokeCSharpAction` JavaScript function will be injected into the web page.
- Resources are released when the element the renderer is attached to changes.
- The `Xamarin.Forms` element is cleaned up when the renderer is disposed of.

ⓘ Note

The `WKWebView` class is only supported in iOS 8 and later.

In addition, **Info.plist** must be updated to include the following values:

XML

 Copy

```
<key>NSAppTransportSecurity</key>
<dict>
  <key>NSAllowsArbitraryLoads</key>
  <true/>
</dict>
```

Create the custom renderer on android

The following code example shows the custom renderer for the Android platform:

C#

 Copy

```
[assembly: ExportRenderer(typeof(HybridWebView), typeof(HybridWebViewRenderer))]
namespace CustomRenderer.Droid
{
    public class HybridWebViewRenderer : WebViewRenderer
    {
        const string JavascriptFunction = "function invokeCSharpAction(data){jsBridge.invokeAction(data);}";
        Context _context;

        public HybridWebViewRenderer(Context context) : base(context)
        {
            _context = context;
        }

        protected override void OnElementChanged(ElementChangedEventArgs<WebView> e)
        {
            base.OnElementChanged(e);

            if (e.OldElement != null)
            {
                Control.RemoveJavascriptInterface("jsBridge");
            }
        }
    }
}
```

```

        ((HybridWebView)Element).Cleanup();
    }
    if (e.NewElement != null)
    {
        Control.SetWebViewClient(new JavascriptWebViewClient(this, $"javascript:
{JavascriptFunction}"));
        Control.AddJavascriptInterface(new JSBridge(this), "jsBridge");
        Control.LoadUrl($"file:///android_asset/Content/{((HybridWebView)Element).Uri}");
    }
}

protected override void Dispose(bool disposing)
{
    if (disposing)
    {
        ((HybridWebView)Element).Cleanup();
    }
    base.Dispose(disposing);
}
}
}

```

The `HybridWebViewRenderer` class loads the web page specified in the `HybridWebView.Uri` property into a native [WebView](#) control, and the `invokeCSharpAction` JavaScript function is injected into the web page, after the web page has finished loading, with the `OnPageFinished` override in the `JavascriptWebViewClient` class:

C#



```

public class JavascriptWebViewClient : FormsWebViewClient
{
    string _javascript;

    public JavascriptWebViewClient(HybridWebViewRenderer renderer, string javascript) : base(renderer)
    {
        _javascript = javascript;
    }

    public override void OnPageFinished(WebView view, string url)

```

```
{  
    base.OnPageFinished(view, url);  
    view.EvaluateJavascript(_javascript, null);  
}  
}
```

Once the user enters their name and clicks the HTML button element, the `invokeCSharpAction` JavaScript function is executed. This functionality is achieved as follows:

- Provided that the custom renderer is attached to a new `Xamarin.Forms` element:
 - The `SetWebViewClient` method sets a new `JavascriptWebViewClient` object as the implementation of `WebViewClient`.
 - The `WebView.AddJavascriptInterface` method injects a new `JSBridge` instance into the main frame of the `WebView`'s JavaScript context, naming it `jsBridge`. This allows methods in the `JSBridge` class to be accessed from JavaScript.
 - The `WebView.LoadUrl` method loads the HTML file that's specified by the `HybridWebView.Uri` property. The code specifies that the file is stored in the `content` folder of the project.
 - In the `JavascriptWebViewClient` class, the `invokeCSharpAction` JavaScript function is injected into the web page once the page has finished loading.
- Resources are released when the element the renderer is attached to changes.
- The `Xamarin.Forms` element is cleaned up when the renderer is disposed of.

When the `invokeCSharpAction` JavaScript function is executed, it in turn invokes the `JSBridge.InvokeAction` method, which is shown in the following code example:

```
C#  
  
public class JSBridge : Java.Lang.Object  
{  
    readonly WeakReference<HybridWebViewRenderer> hybridWebViewRenderer;  
  
    public JSBridge(HybridWebViewRenderer hybridRenderer)  
    {  
        hybridWebViewRenderer = new WeakReference<HybridWebViewRenderer>(hybridRenderer);  
    }  
}
```



```
[JavascriptInterface]
[Export("invokeAction")]
public void InvokeAction(string data)
{
    HybridWebViewRenderer hybridRenderer;

    if (hybridWebViewRenderer != null && hybridWebViewRenderer.TryGetTarget(out hybridRenderer))
    {
        ((HybridWebView)hybridRenderer.Element).InvokeAction(data);
    }
}
```

The class must derive from `Java.Lang.Object`, and methods that are exposed to JavaScript must be decorated with the `[JavascriptInterface]` and `[Export]` attributes. Therefore, when the `invokeCSharpAction` JavaScript function is injected into the web page and is executed, it will call the `JSBridge.InvokeAction` method due to being decorated with the `[JavascriptInterface]` and `[Export("invokeAction")]` attributes. In turn, the `InvokeAction` method invokes the `HybridWebView.InvokeAction` method, which will invoke the registered action to display the pop-up.

Important

Android projects that use the `[Export]` attribute must include a reference to `Mono.Android.Export`, or a compiler error will result.

Note that the `JSBridge` class maintains a `WeakReference` to the `HybridWebViewRenderer` class. This is to avoid creating a circular reference between the two classes. For more information see [Weak References](#).

Create the custom renderer on UWP

The following code example shows the custom renderer for UWP:


```
[assembly: ExportRenderer(typeof(HybridWebView), typeof(HybridWebViewRenderer))]  
namespace CustomRenderer.UWP  
{  
    public class HybridWebViewRenderer : WebViewRenderer  
    {  
        const string JavaScriptFunction = "function invokeCSharpAction(data){window.external.notify(data);}";  
  
        protected override void OnElementChanged(ElementChangedEventArgs<Xamarin.Forms.WebView> e)  
        {  
            base.OnElementChanged(e);  
  
            if (e.OldElement != null)  
            {  
                Control.NavigationCompleted -= OnWebViewNavigationCompleted;  
                Control.ScriptNotify -= OnWebViewScriptNotify;  
            }  
            if (e.NewElement != null)  
            {  
                Control.NavigationCompleted += OnWebViewNavigationCompleted;  
                Control.ScriptNotify += OnWebViewScriptNotify;  
                Control.Source = new Uri($"ms-appx-web:///Content/{((HybridWebView)Element).Uri}");  
            }  
        }  
  
        async void OnWebViewNavigationCompleted(Windows.UI.Xaml.Controls.WebView sender,  
        WebViewNavigationCompletedEventArgs args)  
        {  
            if (args.IsSuccess)  
            {  
                // Inject JS script  
                await Control.InvokeScriptAsync("eval", new[] { JavaScriptFunction });  
            }  
        }  
  
        void OnWebViewScriptNotify(object sender, NotifyEventArgs e)  
        {  
            ((HybridWebView)Element).InvokeAction(e.Value);  
        }  
    }  
}
```

```
protected override void Dispose(bool disposing)
{
    if (disposing)
    {
        ((HybridWebView)Element).Cleanup();
    }
    base.Dispose(disposing);
}
}
```

The `HybridWebViewRenderer` class loads the web page specified in the `HybridWebView.Uri` property into a native `WebView` control, and the `invokeCSharpAction` JavaScript function is injected into the web page, after the web page has loaded, with the `WebView.InvokeScriptAsync` method. Once the user enters their name and clicks the HTML button element, the `invokeCSharpAction` JavaScript function is executed, with the `onWebViewScriptNotify` method being called after a notification is received from the web page. In turn, this method invokes the `HybridWebView.InvokeAction` method, which will invoke the registered action to display the pop-up.

This functionality is achieved as follows:

- Provided that the custom renderer is attached to a new `Xamarin.Forms` element:
 - Event handlers for the `NavigationCompleted` and `ScriptNotify` events are registered. The `NavigationCompleted` event fires when either the native `WebView` control has finished loading the current content or if navigation has failed. The `ScriptNotify` event fires when the content in the native `WebView` control uses JavaScript to pass a string to the application. The web page fires the `ScriptNotify` event by calling `window.external.notify` while passing a string parameter.
 - The `WebView.Source` property is set to the URI of the HTML file that's specified by the `HybridWebView.Uri` property. The code assumes that the file is stored in the `Content` folder of the project. Once the web page is displayed, the `NavigationCompleted` event will fire and the `onWebViewNavigationCompleted` method will be invoked. The

`invokeCSharpAction` JavaScript function will then be injected into the web page with the `WebView.InvokeScriptAsync` method, provided that the navigation completed successfully.