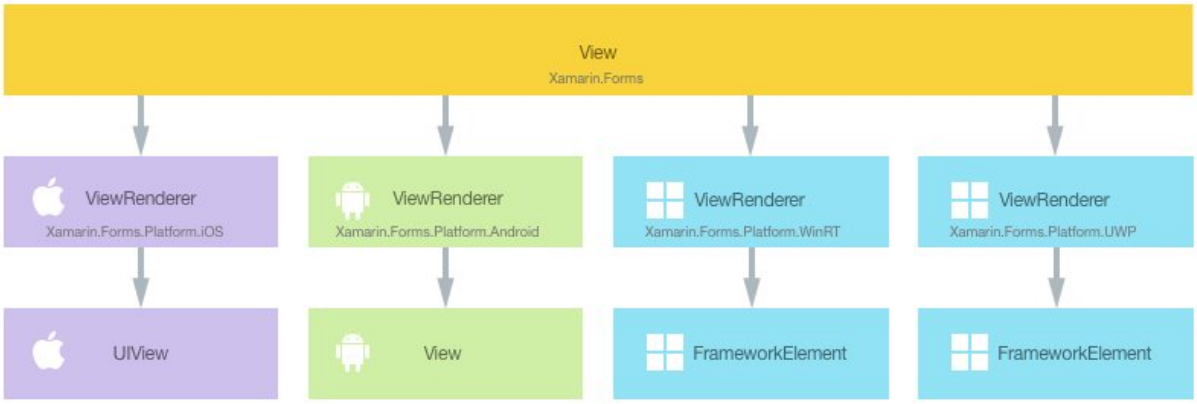
# Implementing a HybridWebView

## Rendering a platform-specific view

Every Xamarin.Forms view has an accompanying renderer for each platform that creates an instance of a native control. When a `View` is rendered by a Xamarin.Forms application in iOS, the `ViewRenderer` class is instantiated, which in turn instantiates a native `UIView` control. On the Android platform, the `ViewRenderer` class instantiates a `View` control. On Windows Phone and the Universal Windows Platform (UWP), the `ViewRenderer` class instantiates a native `FrameworkElement` 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:




The rendering process can be used to implement platform-specific customizations by creating a custom renderer for a `View` 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 in order to implement a `HybridWebView` renderer that enhances the platform-specific web controls 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 Invoking C# from JavaScript. For more information about the HTML page, see Creating the Web Page.

# Creating the HybridWebView

The `HybridWebView` custom control can be created by subclassing the `View` class, as shown in the following code example:

```
public class HybridWebView : View
{
  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 portable class library (PCL) 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-specific project.

# Consuming the HybridWebView

The `HybridWebView` custom control can be referenced in XAML in the PCL 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:

```
<ContentPage ...
             xmlns:local="clr-namespace:CustomRenderer;assembly=CustomRenderer"
             x:Class="CustomRenderer.HybridWebViewPage"
             Padding="0,20,0,0">
    <ContentPage.Content>
        <local:HybridWebView x:Name="hybridWebView" Uri="index.html"
          HorizontalOptions="FillAndExpand" VerticalOptions="FillAndExpand" />
    </ContentPage.Content>
</ContentPage>
```

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:

```
public class HybridWebViewPageCS : ContentPage
{
  public HybridWebViewPageCS ()
  {
    var hybridWebView = new HybridWebView {
      Uri = "index.html",
      HorizontalOptions = LayoutOptions.FillAndExpand,
      VerticalOptions = LayoutOptions.FillAndExpand
    };
    ...
    Padding = new Thickness (0, 20, 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-specific 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:

```
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 in order to enhance the platform-specific web controls by allowing C# code to be invoked from JavaScript.

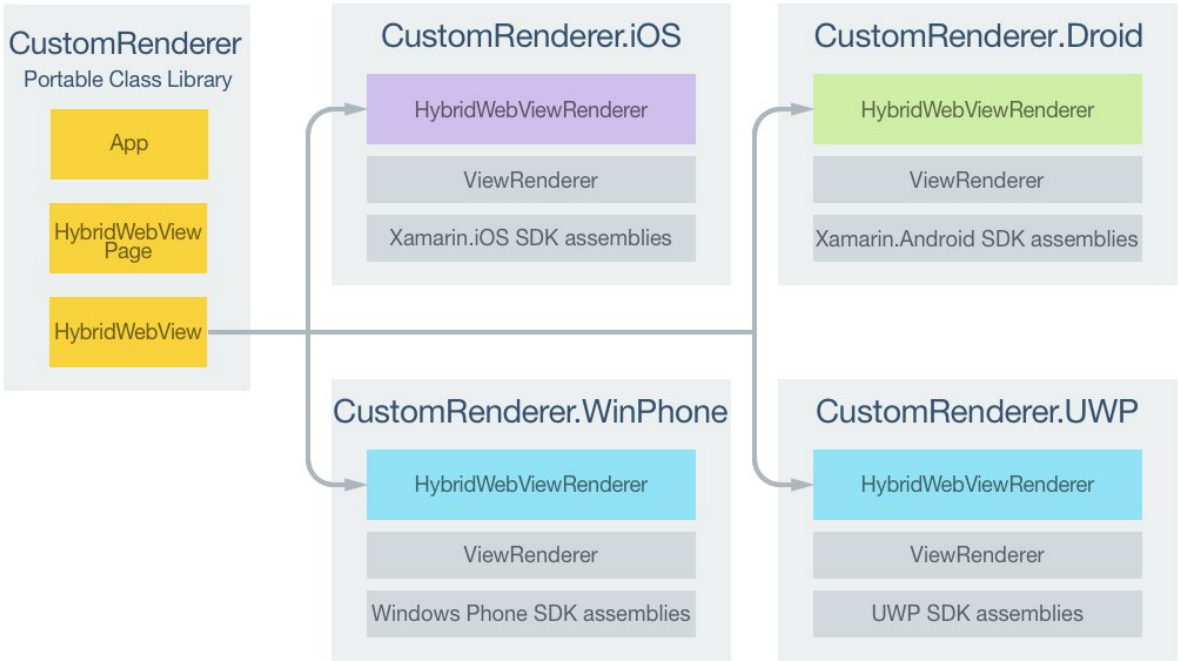
# Creating the Custom Renderer on each Platform

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

1. Create a subclass of the `ViewRenderer<T1,T2>` class that renders the custom control. The first type argument should be the custom control the renderer is for, in this case `HybridWebView`. The second type argument should be the native control that will implement the custom view.
2. Override the `OnElementChanged` method that renders the custom control and write logic to customize it. This method is called when the corresponding Xamarin.Forms custom control is created.
3. Add an `ExportRenderer` attribute to the custom renderer class 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.

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:




The `HybridWebView` custom control is rendered by platform-specific renderer classes, which all derive

from the `ViewRenderer` class for each platform. This results in each `HybridWebView` custom control being rendered with platform-specific web controls, as shown in the following screenshots:


The `ViewRenderer` class exposes the `OnElementChanged` method, which is called when the Xamarin.Forms custom control is created in order to render the corresponding native web control. This method takes an `ElementChangedEventArgs` 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-specific renderer class, is the place to perform the native web control instantiation and customization. The `SetNativeControl` method should be used to instantiate the native web control, and this method will also assign the control reference to the `Control` property. In addition, a reference to the Xamarin.Forms control that's being rendered can be obtained through the `Element` property.

In some circumstances the `OnElementChanged` method can be called multiple times, and so care must be taken when instantiating a new native control in order to prevent memory leaks. The approach to use when instantiating a new native control in a custom renderer is shown in the following code example:

```
protected override void OnElementChanged
(ElementChangedEventArgs<NativeListView> e)
```

```
{
  base.OnElementChanged (e);

  if (Control == null) {
    // Instantiate the native control and assign it to the Control property
with
    // the SetNativeControl method
  }

  if (e.OldElement != null) {
    // Unsubscribe from event handlers and cleanup any resources
  }

  if (e.NewElement != null) {
    // Configure the control and subscribe to event handlers
  }
}
```

A new native control should only be instantiated once, when the `Control` property is `null`. The control should only be configured and event handlers subscribed to when the custom renderer is attached to a new Xamarin.Forms element. Similarly, any event handlers that were subscribed to should only be unsubscribed from when the element the renderer is attached to changes. Adopting this approach will help to create a performant custom renderer that doesn't suffer from memory leaks.

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-specific custom renderer class.

## Creating the Web Page

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

```
<html>
```

```
<body>
<script src="http://code.jquery.com/jquery-1.11.0.min.js"></script>
<h1>HybridWebView Test</h1>
<br/>
Enter name: <input type="text" id="name">
<br/>
<br/>
<button type="button"
onclick="javascript:invokeCSCode($('#name').val());">Invoke C# Code</button>
<br/>
<p id="result">Result:</p>
<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 in order 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.

## Invoking 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.
- 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.

## Creating the Custom Renderer on iOS

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

```
[assembly: ExportRenderer (typeof(HybridWebView),
typeof(HybridWebViewRenderer))]
namespace CustomRenderer.iOS
{
    public class HybridWebViewRenderer : ViewRenderer<HybridWebView,
WKWebView>, IWKScriptMessageHandler
    {
        const string JavaScriptFunction = "function invokeCSharpAction(data)
{window.webkit.messageHandlers.invokeAction.postMessage(data);}";
        WKUserContentController userController;

        protected override void OnElementChanged
(ElementChangedEventArgs<HybridWebView> e)
        {
```

```
            base.OnElementChanged (e);

            if (Control == null) {
                userController = new WKUserContentController ();
                var script = new WKUserScript (new NSString
(JavaScriptFunction), WKUserScriptInjectionTime.AtDocumentEnd, false);
                userController.AddUserScript (script);
                userController.AddScriptMessageHandler (this, "invokeAction");

                var config = new WKWebViewConfiguration { UserContentController
= userController };
                var webView = new WKWebView (Frame, config);
                SetNativeControl (webView);
            }
            if (e.OldElement != null) {
                userController.RemoveAllUserScripts ();
                userController.RemoveScriptMessageHandler ("invokeAction");
                var hybridWebView = e.OldElement as HybridWebView;
                hybridWebView.Cleanup ();
            }
            if (e.NewElement != null) {
                string fileName = Path.Combine (NSBundle.MainBundle.BundlePath,
string.Format ("Content/{0}", Element.Uri));
                Control.LoadRequest (new NSUrlRequest (new NSUrl (fileName,
false)));
            }
        }

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

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 in order to display the pop-up.

This functionality is achieved as follows:

- Provided that the `Control` property is `null`, the following operations are carried out:
  - A `WKUserContentController` instance is created, which allows posting messages and injecting user scripts into a web page.
  - A `WKUserScript` instance is created in order to inject the `invokeCSharpAction` JavaScript function into the web page after the web page is loaded.
  - The `WKUserContentController.AddScript` method adds the `WKUserScript` instance to the content controller.
  - The `WKUserContentController.AddScriptMessageHandler` method adds a script message handler named `invokeAction` to the `WKUserContentController` instance, which will cause the JavaScript function `window.webkit.messageHandlers.invokeAction.postMessage(data)` to be defined in all frames in all web views that will use the `WKUserContentController` instance.
  - A `WKWebViewConfiguration` instance is created, with the `WKUserContentController` instance being set as the content controller.
  - A `WKWebView` control is instantiated, and the `SetNativeControl` method is called to assign a reference to the `WKWebView` control to the `Control` property.
- 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.
- When the element the renderer is attached to changes:
  - Resources are released.

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

## Creating the Custom Renderer on Android

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

```
[assembly: ExportRenderer (typeof(HybridWebView),
typeof(HybridWebViewRenderer))]
namespace CustomRenderer.Droid
{
    public class HybridWebViewRenderer : ViewRenderer<HybridWebView,
Android.Webkit.WebView>
    {
        const string JavaScriptFunction = "function invokeCSharpAction(data)
{jsBridge.invokeAction(data);}";

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

            if (Control == null) {
                var webView = new Android.Webkit.WebView (Forms.Context);
                webView.Settings.JavaScriptEnabled = true;
                SetNativeControl (webView);
            }
            if (e.OldElement != null) {
                Control.RemoveJavascriptInterface ("jsBridge");
                var hybridWebView = e.OldElement as HybridWebView;
                hybridWebView.Cleanup ();
            }
            if (e.NewElement != null) {
                Control.AddJavascriptInterface (new JSBridge (this),
"jsBridge");
                Control.LoadUrl (string.Format
("file:///android_asset/Content/{0}", Element.Uri));
                InjectJS (JavaScriptFunction);
            }
        }

        void InjectJS (string script)
        {
```

```
            if (Control != null) {
                Control.LoadUrl (string.Format ("javascript: {0}", script));
            }
        }
    }
}
```

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 `InjectJS` method. 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 `Control` property is `null`, the following operations are carried out:
  - A native `WebView` instance is created, and JavaScript is enabled in the control.
  - The `SetNativeControl` method is called to assign a reference to the native `WebView` control to the `Control` property.
- Provided that the custom renderer is attached to a new Xamarin.Forms element:
  - 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.
  - The `InjectJS` method is invoked in order to inject the `invokeCSharpAction` JavaScript function into the web page.
- When the element the renderer is attached to changes:
  - Resources are released.

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

```
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)) {
      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 invoked the registered action in order to display the pop-up.

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 in order to avoid creating a circular reference between the two classes. For more information see Weak References on MSDN.

## Creating the Custom Renderer on Windows Phone and UWP

The following code example shows the custom renderer for Windows Phone and UWP:

```
[assembly: ExportRenderer(typeof(HybridWebView),
typeof(HybridWebViewRenderer))]
```

```
namespace CustomRenderer.WinPhone81
{
    public class HybridWebViewRenderer : ViewRenderer<HybridWebView,
Windows.UI.Xaml.Controls.WebView>
    {
        const string JavaScriptFunction = "function invokeCSharpAction(data)
{window.external.notify(data);}";

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

            if (Control == null)
            {
                SetNativeControl(new Windows.UI.Xaml.Controls.WebView());
            }
            if (e.OldElement != null)
            {
                Control.NavigationCompleted -= OnWebViewNavigationCompleted;
                Control.ScriptNotify -= OnWebViewScriptNotify;
            }
            if (e.NewElement != null)
            {
                Control.NavigationCompleted += OnWebViewNavigationCompleted;
                Control.ScriptNotify += OnWebViewScriptNotify;
                Control.Source = new Uri(string.Format("ms-appx-
web:///Content//{0}", Element.Uri));
            }
        }

        async void OnWebViewNavigationCompleted(WebView sender,
WebViewNavigationCompletedEventArgs args)
        {
            if (args.IsSuccess)
            {
                // Inject JS script
```

```
                await Control.InvokeScriptAsync("eval", new[] {
JavaScriptFunction });
            }
        }

        void OnWebViewScriptNotify(object sender, NotifyEventArgs e)
        {
            Element.InvokeAction(e.Value);
        }
    }
}
```

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 in order to display the pop-up.

This functionality is achieved as follows:

- Provided that the `Control` property is `null`, the following operations are carried out:
  - The `SetNativeControl` method is called to instantiate a new native `WebView` control and assign a reference to it to the `Control` property.
- 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.

- When the element the renderer is attached to changes:
  - Events are unsubscribed from.

# Summary

This article has demonstrated how to create a custom renderer for a `HybridWebView` custom control, that demonstrates how to enhance the platform-specific web controls to allow C# code to be invoked from JavaScript.