A Modern Web Browser Control

Christof Wollenhaupt
foxpert GmbH
Ulzburger Straße 352
22846 Norderstedt, Germany
Voice: +49-40-6053373-70
Website: http://www.foxpert.com
Email: christof.wollenhaupt@foxpert.com

For many years there was only one way to view web content in a Windows 32-bit desktop application: the IE based web browser control. The web browser control is difficult to tame and handle (unless you are Rick Strahl), particularly if you have no control over the content that should be displayed.

The situation has changed, though. Microsoft announced an embeddable version of Edge Chromium in 2018 which we will briefly cover in this session. Most of the session will focus on another control though. We will look at the Chromium Embedded Framework, an open source project for embedding a Chromium browser in an application. Chromium is the open source foundation on which Google Chrome, Microsoft Edge, and Opera are built on.

The CefSharp project makes CEF available as a .NET WinForm and WPF control which we can use in our Visual FoxPro applications. This session looks at how you embed CefSharp, how to view remote and local files, how to extend the control, how to interact with the web page from your VFP app, how to run code in VFP from your web control, how to exchange data with VFP, how to debug JavaScript code in your application and, of course, how to distribute CefSharp with your application.

Introduction

When I consider embedding a web browser control in an application, my perspective is that of a business application developer. I see the web browser control as a way of displaying more complex reports, to offer a nicer UI for displaying data, to interact with other web applications like catalog systems, status pages, appointment systems, etc.

Microsoft uses web browser controls in a similar way. Some of the UI elements in Outlook, such the sidebar, are web view elements instead of WPF controls.

But there are use cases that were not obvious, at least to me. It appears that the gaming industry is relying on HTML content rendered into the game. Unreal Engine, one of the leading gaming engines from Epic Software, includes the CefSharp browsers that renders HTML content into a texture that can be used within the game. Most launchers and chat tools for games are also browser based.

Many applications come as a web product and a desktop application. The desktop application is often just a repackaged web application. If you run Slack, Trello, Skype, Spotify, Amazon Music, Evernote, Signal, BaseCamp, WhatsApp, Github Desktop, Microsoft Teams, Yammer, even Visual Studio Code or the XBox app, you are running Chromium either as part of Electron or with the Chromium Embedded Framework (CEF) which is what we will mostly use in this session.

You probably have heard about the Chrome browser even though you might prefer FireFox, Edge, Safari, or Opera, or any other browser not associated with one of the big players. How is Chromium different from Chrome?

Chrome is not open source, Chromium is. Chrome is similar to the Edge browser. Both base on the Chromium browser, but add proprietary features that are not open sourced.

Chrome supports codecs that require license fees such as AAC, H.264 and MP3. This means that Chromium doesn't support some websites that rely on these modern streaming codecs for video and music. The same would be true for your application, because CefSharp also bases on Chromium, but doesn't add any codecs. If you wonder when you would need these codecs, let me say one word: Netflix.

Adobe Flash is sandboxed in Chrome, but not installed by default in Chromium. We will discuss Adobe Flash in a later section to get around the limitations of Chromium.

Chrome is woven into the Google infrastructure. It has automatic updates, a sophisticated UI, an app store that is the only available source for Chrome plugins and an error reporting that is most noticeable by its energy consumption that forces the fans to rotate at maximum speed. Most unique Chrome features focus on the UI experience, such as input prediction, synchronization, anti-phishing support.

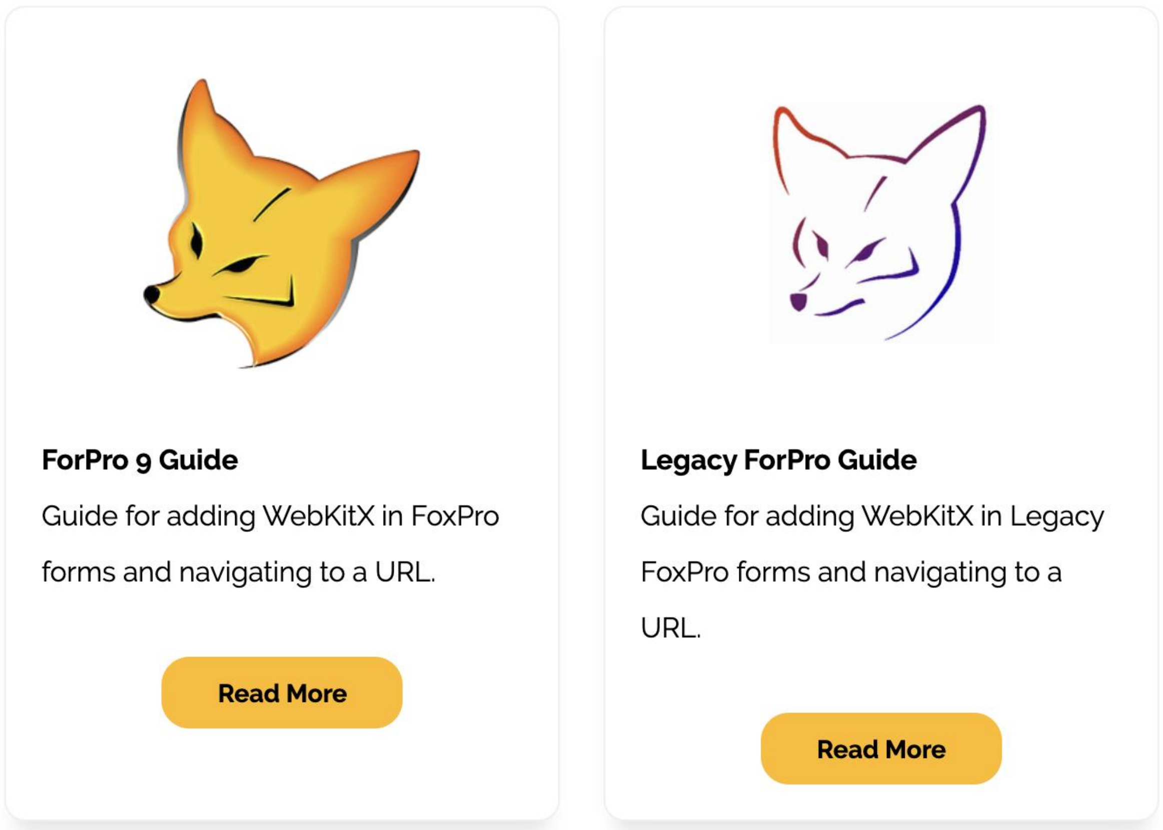
Chromium is the foundation for Chrome, Opera, Edge and Amazon Silk. It's based on webkit which is the basis for Safari. Webkit is based on the KDE browser engine which incorporated Gecko, the rendering engine that drives FireFox. Dozens of browsers are based on variations of these engines. Just one engine is finally gone for good: Internet Explorer.

The Chromium Embedded Framework (CEF) is based on Chromium. It's a library that lets us embed a Chromium browser into our own application. I mentioned various applications that are based on CEF. One company is more important than others here. The Swedish company Spotify is maintaining versions for Linux, Mac and Windows.

CEF is a C/C++ product and easy to use in these languages (if you consider C++ to be easy). However, most of the world, at least in the business world, isn't using C++. They rely on tools like FoxPro, .NET, Java and JavaScript (yes, the Wikipedia article for CEF mentions FoxPro).

There are several ways to use CEF in Visual FoxPro. One of the more recent additions is WebKit ActiveX which is the reason Wikipedia mentions Visual FoxPro. The control officially supports all VFP versions starting with VFP 6 and has samples on their website to get you started in VFP. Funnily, these samples start with a fresh installed copy of VFP:

https://www.webkitx.com/doc/light/Getting%20Started%20with%20FoxPro%209.html



ForPro 9 Guide

Guide for adding WebKitX in FoxPro forms and navigating to a URL.

Legacy ForPro Guide

Guide for adding WebKitX in Legacy FoxPro forms and navigating to a URL.

Do not get confused that FoxPro is sometimes called ForPro, we all know that real Pros use FoxPro.

WebKitX is a commercial product that requires a license. A royalty free license for desktop applications is £599, this is British pounds. At the time of writing this is between $700 and $800. Extra licenses are required to run the control on servers or non-desktop devices.

If you are looking for a commercially supported product that is as easy to integrate into your application as any other ActiveX control, then this might be an option for you.

There are two reasons why I did not take this route and used CefSharp which I present in this session and whitepaper. First, there is a timing issue. I researched CefSharp for the first time around 2015 and implemented it in a commercial application in 2018. That year was just when WebKitX started as a new product with no track record.

The other reason is that after switching to wwDotNetBridge, I really want to minimize the number of ActiveX controls I must deal with. This is especially true for controls where I do not have control over the source. My solution does require an ActiveX control which is just a few lines of code. But it is a generic one, it's available in source code and it doesn't limit what I could do in CefSharp.

The drawback, of course, is that CefSharp is harder to use and has a steep learning curve, much like a lot of Open Source software in general has.

Using the browser control 101

Executive Summary aka TL;DR

If you are only interested in replacing the Microsoft web browser control to show HTML documents or remote content, then this is the chapter you want to read.

The session material contains version 84 of CefSharp, the current version in September 2020. Copy the various file into your project folder:

cef-bin-v84.4.10 – This folder must be a subfolder of your project root. The entire content needs to be shipped to your client as a subfolder of your EXE. Yes, it’s large at 152 MB. But wait until you see how much memory CefSharp consumes. It’s Chrome, after all.

fpDotNet – The content of this folder can go anywhere you want. However, as this is an ActiveX Control, you must register the DLL not only on your machine, but also on your client’s machines. This requires administrative privileges. This is a .NET 4.0 component which means you use regasm rather then regsvr32

```
C:\Windows\Microsoft.NET\Framework\v4.0.30319\regasm fpDotNet.dll /tlb fpDotNet.tlb
/codebase
```

Execute this line in a CMD window that runs as an administrator. If you changed your Windows installation folder, modify the command line accordingly. Do not replace wwDotNetBridge.dll with your own copy. This is a modified version.

ClrHost.dll, wwDotNetBridge.dll – These files must be in the path so that VFP can load them. Do not replace wwDotNetBridge.dll with your own copy. This is a modified version.

CefSharpBrowser.prg, various .h files – These can go anywhere in your project if you include the prg file into your project.

All other files are demo files that you can modify to suit your own needs. Showing HTML content requires two steps.

The visual control in VFP is an ActiveX control. Open whatever class or form you want to add the browser to. Then add an OLEControl instance and pick fpDotNot.DotNetContainer from the list of available classes. If you can't see this, you have skipped the REGASM call. Please read this section again.

Next you bind an instance of CefSharpBrowser to the UI control. These are two easy lines of code. I keep the object reference in a property named oCefSharpBrowser:

```
This.oCefSharpBrowser = NewObject ("CefSharpBrowser", "CefSharpBrowser.fxp")
This.oCefSharpBrowser.BindToHost (This.oleBrowser, m.lcUrl)
```

`lcUrl` in this sample is a variable with the full web address including http, https or file. Run this code and after a brief delay you should see the HTML document in the browser.

If you encountered any problems or want to know what else you can do, keep reading.

Printing content

Your browser control finally displays the content that you want, running modern JavaScript code and providing all the features a modern browser should support.

No matter how nice you make it look on the screen, there will always be users that need to print the content. That does not always mean that trees are harmed, and physical paper is produced. Windows has been shipping with PDF printer drivers for some time now. If you display documents, it is likely they need to be sent to customers, filed with a document management system, or added to another workflow.

In fact, generating HTML can be a lot more flexible than using a VFP report, because you get all the flow layout and formatting out of the box. With custom CSS you can even let users style their report. Wouldn't it be nice if you could automatically produce a PDF file from HTML content?

You might remember how non-obvious it was to print with the Microsoft web browser control. Basically, anything that was possible from the UI happened in a small method named ExecWB. You would pass, for instance, IDM_EXECPRINT to this method, and it would execute the same code that choosing File > Print in Internet Explorer would execute.

If you hoped for a nicer interface this time around, you hope will unfortunately be disappointed. But it's a different interface for a change.

We have already used the ChromiumWebBrowser class to show the content. This class implements IWebBrowser (not to be confused with IBrowser). A quick peek at the documentation reveals that the browser control has a Print method. That seems easy enough to call, except if you tried, you would get an error message stating that the method does not exist.

As often the clue is in the documentation. Print is what C# calls an extension method. Why does it have to be so complicated?

CefSharp is a browser implementation that is independent of the UI technology that is used. There is an implementation for WinForms (which is the one we are using), for WPF, even one that renders into bitmaps. The interface to interact with the browser window in a neutral way is IWebBrowser. The "I" tells us that this is an interface that does not have any implementation attached.

However, quite a bit of implementation is identical for different technologies. The implementation is in a helper class named cefsharp.WebBrowserExtension. The name of the class derives from a shortcut you can do in C#. When you have a static class with static methods, which means there is no object involved, and use a special syntax, you can call these extension methods as if they are native methods of the class they extend.

The compiler rewrites this method call behind the scenes. Instead of calling a method in the object such as our web browser control, the compiler calls the static method passing the reference to the web browser control as the first parameter.

What happens transparently in C#, is something we need to handle explicitly from VFP.

Interacting with the browser control

So far, we have loaded HTML content in the browser control and let the user interact with the document much in the same way they would interact with it in the browser.

This can be quite useful, for HTML reports, embedding independent tools or displaying documents. For example, in one application we scan pictures and documents. One option we offer is to create a multi-page HTML document with all pictures combined into one document. We use a browser control to display this document in our application, but we do not need to interact with it in any way.

The web browser control has more power when we can respond to events in the document. This requires two steps that we look at individually. The first step is being able to react to events that happen in the browser, that is, being notified when the user performs an action. The second step is being able to respond to that reaction, like updating content.

Creating your own protocol

Most developers used the BeforeNavigate2 event in the web browser control to intercept actions from the browser. You would receive the URL, headers, extra post data from the HTML form; basically, the same information that are in the HTTP request. You could then act on the data and decide to either cancel the request or let the browser control send it to the actual recipient.

The most common approach to encode URLs for the purpose of intercepting them was to create a fake protocol. For example, the button that navigates to the next record would navigate to "app:next". This would not interfere with any actual links and make it easy to check for a special link. Just check whether the first four characters are "app:".

Because this was a familiar approach for me, it was the first one I tried with CefSharp. Where the web browser control would present me with a simple interface with limited choices, CefSharp offers an abundance of objects, methods, controls, namespaces, you name it. So far I have found two approaches to create your own protocol.

The more common one is to create a handler for your new protocol. Protocols are called schemes in CefSharp.

```
*======================================================================================
* Creating your own protocol...
*======================================================================================

   #include Acodey.h

   *------------------------------------------------------------------------------
   * Create a factory instance. cefBrowser calls the factory to create a new handler
   * instance. This is a special implementation that calls the OnProcessRequest
   * method.
   *------------------------------------------------------------------------------
   Local loFactory
   loFactory = toBridge.CreateInstance ;
      ("fpDotNet.fpCefSharp.fpSchemeHandlerFactory")
   #IF __DEBUGLEVEL >= __DEBUG_REGULAR
      Assert Vartype (m.loFactory) == T_OBJECT
   #ENDIF

   *------------------------------------------------------------------------------
   * Create a new protocol (scheme)
   *------------------------------------------------------------------------------
   Local loScheme
   loScheme = toBridge.CreateInstance ("CefSharp.CefCustomScheme")
   loScheme.SchemeName = "app"
   toBridge.SetProperty (m.loScheme, "SchemeHandlerFactory", m.loFactory)
   toBridge.InvokeMethod (m.toSettings, "RegisterScheme", m.loScheme)
```

The code is almost straight forward. When you configure CefSharp in your application, you register a SchemeHandlerFactory. This must be a class that implements ISchemeHandlerFactory. CefSharp calls this object whenever a request is processed. However, as the name implies, the factory object doesn't process the request directly.

Instead it creates a new instance of a SchemeHandler object that handles the request. This requires a few extra classes written in C# that we cannot easily implement with wwDotNetBridge. We look at these classes more in detail later on.

You will remember that you can initialize CefSharp only once per process. This means you must register all protocols you need in your application all at once. You cannot break your application into independent pieces that register and configure a browser window as they need at the time.

Registering my scheme worked just fine for my simple test pages. I would display data and a button that would call my own scheme. Whenever I click the button, the OnProcessRequest method is called in VFP and I can do whatever I want.

But then I needed to interact with another web application. Our application would log our users into a remote catalog system and display it in a form within our application. Users would be able to select items from the catalog according to various criteria that the catalog system handled. In the end we would need to retrieve this list of items and import it into our system.

The system provides a call back URL and a way for the user to initiate the data transfer. We pass a custom URL. They use JavaScript to create a new HTTP request, combine all items into an XML string and sends the POST-request to the URL that we specified.

Except that it did not work.

Because with JavaScript requests there is something named CORS, Cross-Origin Resource Sharing. JavaScript and browsers have strict rules which servers you can connect to from code. Code that has been downloaded from one server cannot suddenly send data to another server using a different protocol.

file:// vs http:// vs https://

It turned out that CefSharp treats my own schemes more like the file: scheme. A URL starting with "file://" refers to a local file on your computer. Because it can be mis-used to send data from your computer to some malicious server, the file scheme is very limited in what it can do with regard to other servers. The reverse is true, too. Web pages loaded from a server are prevented from accessing local file schemes, such as the one we just created.

While there are several options to configure your own protocol, there is still the issue that going from an https-protocol based web site to an "app://import" URL is a change of the protocol with various cross-protocol implications. For instance, CefSharp (actually CEF which is behind CefSharp) runs one render process for each combination of scheme and domain.

Some features only work with http:// and https://, but not with any custom scheme or any file:// scheme. This includes posting data cross-domain, this includes access to privacy-related services such as the camera, the microphone, location services, etc.

When even the documentation states you should rather use either http or https rather than creating your own scheme, you should take the hint.

Creating your own server

In fact, CefSharp provides a comparable convenient way to hook into http and https requests, basically letting you create a virtual web server just for your application. There is another big advantage of this approach.

When you use the web browser control you must load HTML into the control. When you are displaying a remote website that you integrate into your application this all quite easy. Just navigate to the URL. Often you want to show application specific and application generated content within the web browser control.

There are two common approaches to loading an arbitrary HTML document into the web browser control. You could create a temporary HTML file and navigate to it using the file:// protocol. Or, you would load a page such as "about:blank" and then manipulate the DOM to load the string.

Both work apparently, but both end up not being in the Internet zone and applying restrictions to what the code can do. For the Internet Explorer based control that might not be such a big issue, because it doesn't support a lot of the modern APIs anyway.

However with cefSharp the same approach would seriously restrict HTML5 features and interoperability with web services.