# Using Windows Runtime and SDK to Build Metro style Apps (DEV350)

In this document, we will create a simple HTML based editor. Throughout this talk, we will be introduced to a number of powerful features in WinRT:

* Using what you know: HTML/CSS, C#, C++
* Using what you have: an existing JS library, CodeMirror
* Lighting up on the OS: integrating with your user’s documents
* Mixing and matching your languages: using the .NET ZipArchive library from your JS app
* When bad things happen: live and post mortem debugging of JS and C++ code
* Use what others know: adding a Search contract that uses Bing search suggestions
* Getting it right: testing on other devices and post mortem debugging from that device (C++/C#)
* Ship it: using the Windows App Certification Kit before you ship

This document will describe in detail all of the steps that I did in my Tech Ed talk. You can use this document to follow along.

## Create an HTML page

In this section, we’ll start by creating a pure HTML page (not a WinRT app) that implements a simple HTML editor. We’ll see how you can use what you know already (HTML/JS) and what you have already (CodeMirror, a library for a syntax coloring text editor) to rapidly assemble the core of an application.

TODO: clean up the code in this section so that it gives a prettier looking app.

1. Create step1 directory.
2. Create file called editor.html that contains:

<html>

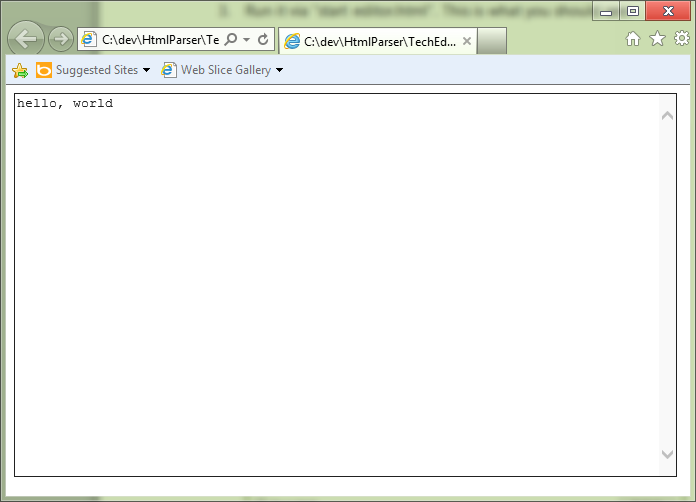
<body>

<textarea cols="80" rows="25">hello, world</textarea>

</body>

</html>

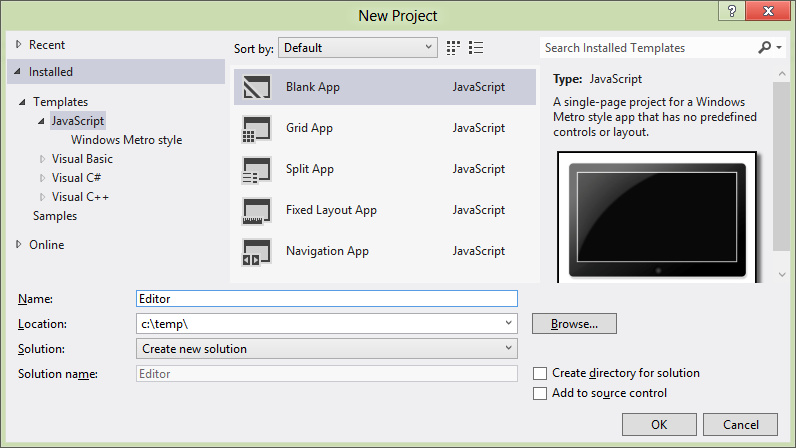
1. Run it via "start editor.html". This is what you should see:



1. Note that it starts up in **iexplore.exe**. You can see it via task manager:

Machine generated alternative text: Apps (21)
L’  GitHub (32 bit)
Internet Explorer
i Internet Explorer
r— C:\dev\HtmlParser\TechEd\stepl\editor.html -
E Mail
L’ ‘ij] Microsoft OneNote (32 bit)
L’ Øj Microsoft Outlook (32 bit) (2)
Microsoft Visual Studio 2010 (32 bit)
L’ 4 Microsoft Visual Studio Express 2012 RC for Windows 8 ...
4 Microsoft Visual Studio Express 2012 RC for Windows 8 ...
j Notepad2x&4
ELI OneNoteMX
iý Product Studio Client UI (32 bit)
<
(Z) Fewer details
0% 46.2 MB
0% 45.6 MB
0% 14.9 MB
0% 78.4MB
0.1% 30.6 MB
0.2% 88.4 MB
0% 97.5 MB
0% 132.0 MB
0% 159.7 MB
0% 1.5 MB
0% 345MB
0% 31.1 MB
C
>
End task

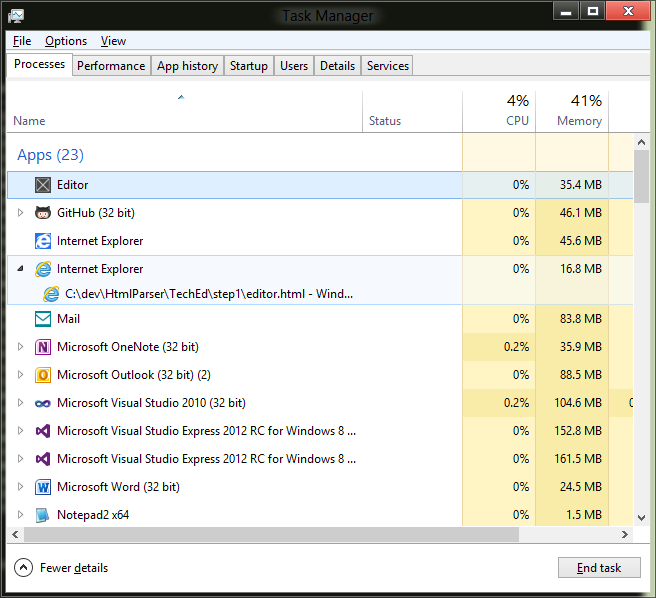
1. Let’s go beyond just a <textarea>. Let’s search the web to find a library that will do syntax coloring for a browser-based editor. Search for “browser editor javascript” – we find [CodeMirror](http://codemirror.net/).
2. We’ll do the Julia Childs approach here and we’ll integrate a set of CodeMirror files into our app. We’ll need exactly 5 files from CodeMirror. If you’re following from home, you’ll need to grab these files from CodeMirror and copy them into our app:
3. This is a traditional web app. Now let's turn it into a Metro style app.
4. Startup Visual Studio and select File … New … Project. Name your project **Editor**.



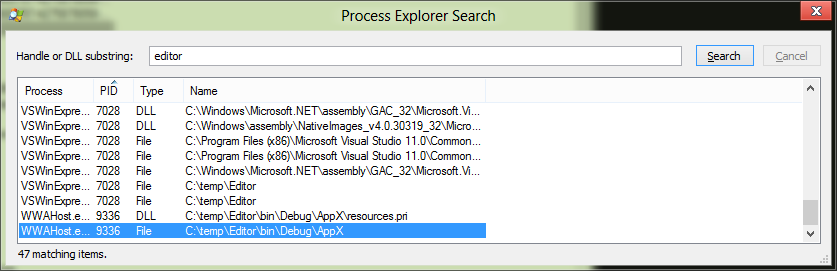
1. Navigate to **default.html** in the solution explorer, and replace the <p>Content goes here</p> HTML with:

<textarea cols="80" rows="25">hello, world</textarea>

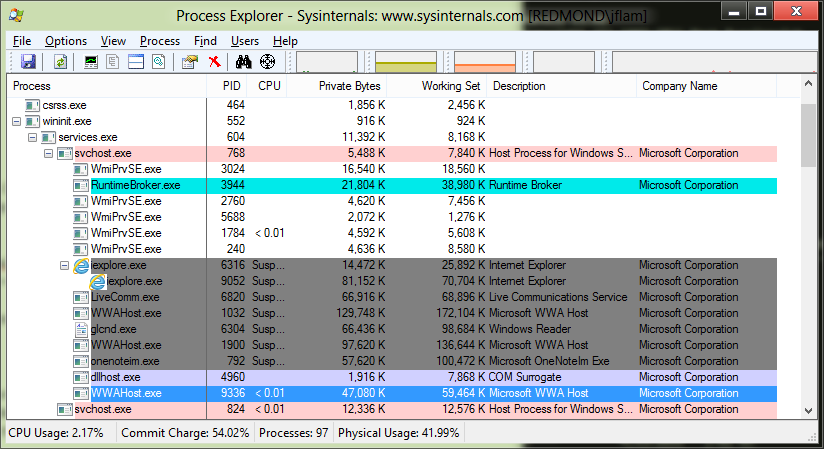
1. If you run the app, you should see a full screen Metro style app that does much like what you saw before. However, if we look under task manager, you’ll see a different view of the world than you saw before. You’ll see that Editor is now a top-level entry under Apps vs. being a category under Internet Explorer.



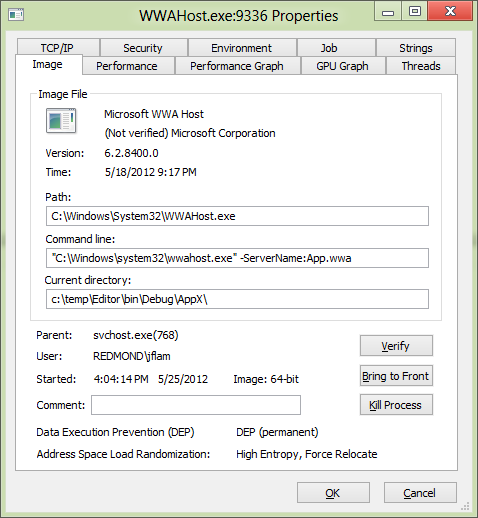
1. Let’s get some more information by running procexp by Mark Russinovich. Start procexp from the command line and hit CTRL-F to search for our app. Search for a string called “Editor”. On my machine, it found a process called WWAHost that references my Editor directory. It’s PID is 9336:



In the rest of procexp we can see additional interesting details, including the relationship of its parent process tree:

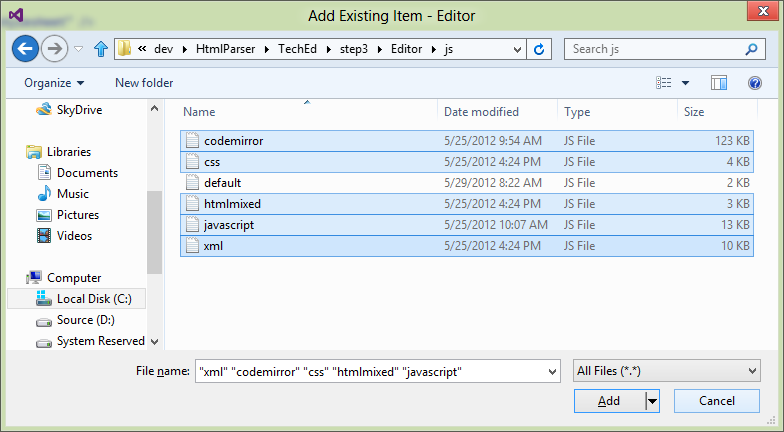


If we right click on WWAHost.exe and click Properties … we’ll see additional information about the process, including the command line used to start it and its current directory:



There are many more details here, but the key points are that we are launching your “application” now using a different host application: WWAHost.exe vs. iexplore.exe.

1. Now, let’s make the app a bit more interesting by injecting some pre-existing code into the app. We’ll make our editor better by adding syntax coloring to it, along with a dynamic preview pane. This will turn it into a simple HTML editor.
2. Copy the **codemirror.js, htmlmixed.js, javascript.js, css.js, and xml.js** files from the **zzz** directory to the **js** directory. Copy the **codemirror.css, lesser-dark.css** files from the zzz directory to the **css** directory.
3. Add them to your project by right-clicking on the **js** directory in solution explorer and selecting Add Existing Item ... Repeat this action for the files from the **css** directory.



1. Now add some references to those files in **default.html**:

<!-- Codemirror references -->

<script src="/js/codemirror.js"></script>

<script src="/js/htmlmixed.js"></script>

<link rel="stylesheet" href="/js/codemirror.css" />

<link rel="stylesheet" href="/js/lesser-dark.css" />

You may want to move the CSS directories under the **css** folder, but that’s up to you.

1. Now, change the <textarea> element into a <div> element that has the id of **editor**:  
     
   <div id="editor">hello, world</div>
2. Add some code to initialize the editor in **default.js**:

app.onloaded = function (args) {

app.editor = CodeMirror.fromTextArea(document.getElementById("editor"), {

lineNumbers: true,

indentUnit: 4,

theme: "lesser-dark"

});

};

1. Now add some code to **default.css** to ensure that the editor occupies the left half of the app:

.CodeMirror {

background-color: #000;

width: 50%;

}

.CodeMirror-scroll {

height: 100vh;

}

We’ve now got essentially the same app, but now running as a Windows 8 Metro style HTML application.

*TODO: consider moving the CSS hackery to the pure HTML part of the app and then copying everything as-is into the WWA. That’s probably a better demo.*

*~~TODO: add the keystroke bindings so that we update the <div> output on the right hand side for a nice WYSIWYG editing experience.~~*

1. Now, add some code to **default.js** so that we can map Ctrl-Enter as a key binding to render the HTML in the editor in the <div> next to it.

CodeMirror.keyMap.html\_editor = {

'Ctrl-Enter': function (cm) {

// Eval the HTML here and drop it into the <div>

var html = app.editor.getValue();

preview.innerHTML = html;

},

fallthrough: ["default"],

};

1. This goes in **default.css**:

#preview {

position: absolute;

top: 0px;

width: 50%;

height: 100vh;

left: 50vw;

}

1. This goes in the **default.html**:

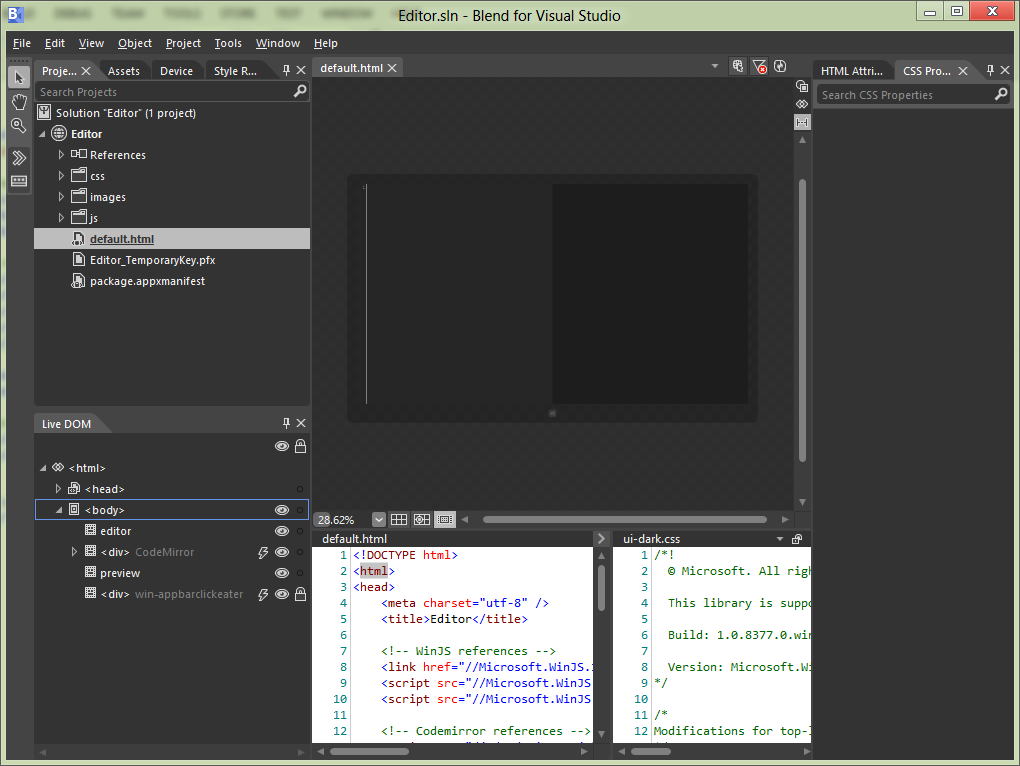
<div id="preview"></div>

# Integrating the app with the OS

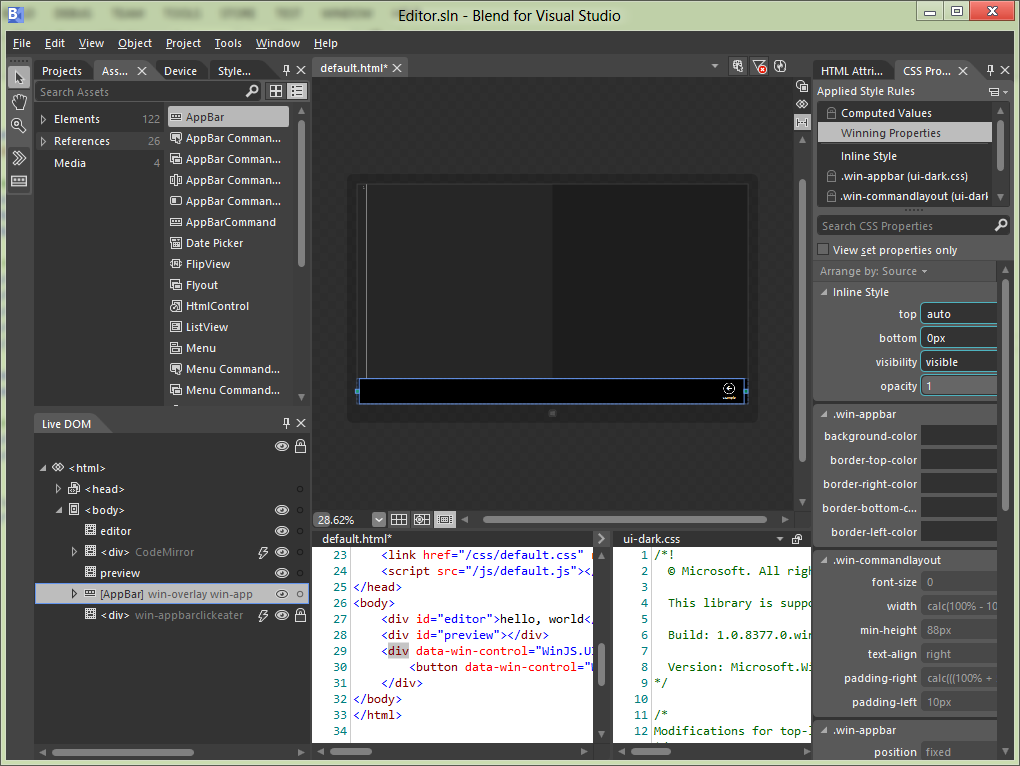
Now we need to do the work to integrate the app into the OS. Let’s add the ability to save and open a local file.

Let’s do this by adding an App Bar with a couple of buttons using Blend

1. Right click in VS and select Blend
2. Opens up in Blend … looks like this:



1. Now let’s insert an AppBar by clicking on Assets and dragging and dropping an AppBar onto the surface:



1. Now let’s tweak the attributes of the Example button on the AppBar. Let’s zoom in on the HTML Attributes of the sample button, and change the **icon** attribute to **openfile** and the **label** attribute to Open, and the **id** attribute to **cmdOpen**:



1. Now let’s bind the event handler to the click event handler of the AppBar button:

document.getElementById('cmdOpen').addEventListener('click', app.open\_file);

*TODO: add a section here on trying to directly manipulate a file, watch it fail. And then add some code to use procmon to monitor runtimebroker.exe to see how it brokers access to the underlying file on behalf of the app.*

*Need to ensure that we have the right filters here so that we can see both runtimebroker access to the file and what the app does to access the file. I don’t really understand how this works …*

1. Let’s try to programmatically open a file on the user’s computer. Add this code:

app.open\_file = function (args) {

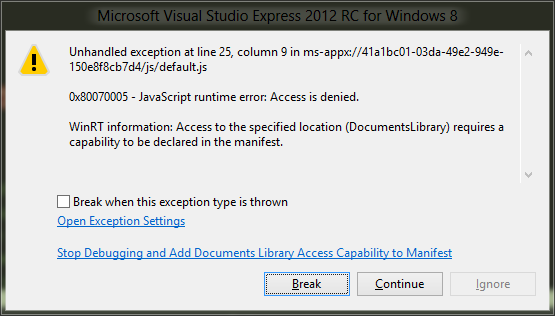
// This won't work

var file = Windows.Storage.KnownFolders.documentsLibrary.getFileAsync("hello.html")

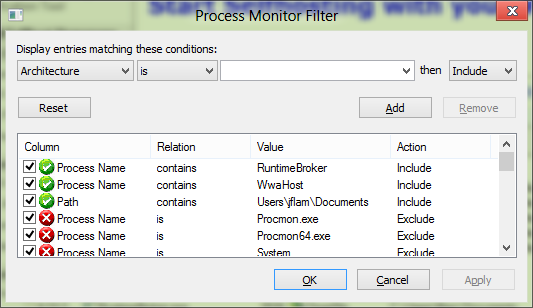
var text = Windows.Storage.FileIO.readTextAsync(file);

};

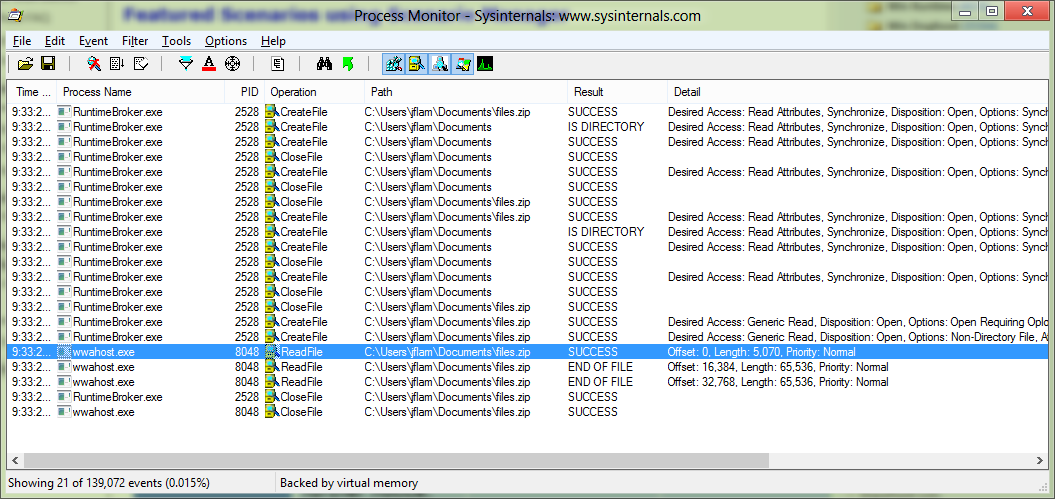
1. When we run it this is what you’ll see:



1. Note that we are not manifesting the Documents Library Access capability which enables programmatic access to files in the user’s Documents directory which map the file type associations declared in the manifest.
2. We can setup some filters in **procmon** to filter out events of interest. Note that we’re looking for activity in either the RuntimeBroker.exe process or in a WwaHost.exe process where the Path contains my Documents directory.



1. This is what procmon shows when we look for all access to files under my Documents directory.



1. Let’s write some code to handle the event:

app.open\_file = function (args) {

var openPicker = new Windows.Storage.Pickers.FileOpenPicker();

openPicker.viewMode = Windows.Storage.Pickers.PickerViewMode.list;

openPicker.suggestedStartLocation = Windows.Storage.Pickers.PickerLocationId.documentsLibrary;

openPicker.fileTypeFilter.replaceAll([".htm"]);

openPicker.pickSingleFileAsync().done(function (file) {

Windows.Storage.FileIO.readTextAsync(file).done(function (html) {

var clean\_html = window.toStaticHTML(html);

app.editor.setValue(clean\_html);

});

});

};

1. We need a bit of helper code to clean up the HTML a bit more for Wikipedia.

var regex = /\"\/\/(.\*?)\"/ig;

var result = html.replace(regex, "\"http://$1\"");

var clean\_html = window.toStaticHTML(result);

app.editor.setValue(clean\_html);

1. Insert this HTML in the App Bar section of **default.html**:

<button data-win-control="WinJS.UI.AppBarCommand" data-win-options="{icon:'save', id:'cmdSave', label:'Save', section:'global', type:'button'}"></button>

1. We will bind the save button to this other handler. Put this after the other **addEventListener** call in **app.onactivated**:

document.getElementById('cmdSave').addEventListener('click', app.save\_file);

1. Now let’s add code to our **open\_file** API above to ensure that we squirrel away a token for future access. Do this before we call **readTextAsync**:

// Squirrel away a token for future access

app.file\_token = Windows.Storage.AccessCache.StorageApplicationPermissions.futureAccessList.add(file);

1. Now let’s add a function to save the file, making sure that we use the FileSavePicker to and the futureAccessList APIs to cache it so that we can do CTRL-S correctly.

app.save\_file = function (args) {

var text = app.editor.getValue();

if (app.file\_token == null) {

var savePicker = new Windows.Storage.Pickers.FileSavePicker();

savePicker.suggestedStartLocation = Windows.Storage.Pickers.PickerLocationId.documentsLibrary;

savePicker.defaultFileExtension = ".htm";

savePicker.suggestedFileName = "my\_html";

savePicker.fileTypeChoices.insert("HTML", [".htm"]);

savePicker.pickSaveFileAsync().done(function (file) {

if (file) {

Windows.Storage.FileIO.writeTextAsync(file, text);

}

});

} else {

Windows.Storage.AccessCache.StorageApplicationPermissions.futureAccessList.getFileAsync(app.file\_token).done(function (file) {

if (file) {

Windows.Storage.FileIO.writeTextAsync(file, text);

}

});

}

};

1. Now write some code to bind CTRL-S to **app.save\_file**. Insert this code in the **CodeMirror.keyMap.html\_editor** object definition:

'Ctrl-S': function (cm) {

app.save\_file(null);

},

Now look, we have a fully functional app that supports loading and saving to the file system.

Note that we don’t need to add any additional capabilities to the app – as long as the user consents we can keep the app open.

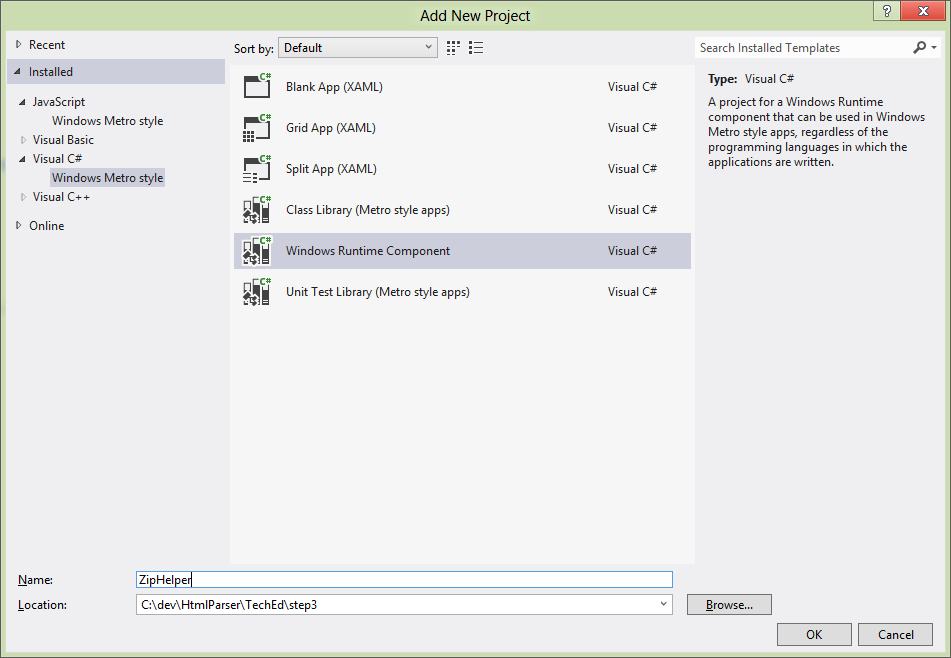
# Mixing and matching code and libraries written with different languages

Let’s open a zipped file that contains some HTML that we would like to edit. For simplicity we will assume that the file contains a single HTML file and decompress and open.

How do we make this work? Use C#

Create WinRT component

1. Add a New Project to the solution. It must be a C# Windows Runtime Component called **ZipHelper**:



1. Paste this code into **Class1.cs** and rename it to **Zip.cs**:

using System;

using System.IO;

using System.IO.Compression;

using System.Runtime.InteropServices.WindowsRuntime;

using System.Threading.Tasks;

using Windows.Foundation;

using Windows.Storage;

namespace ZipHelper

{

public sealed class Zip

{

// Inefficient copy

private static string CopyStreamToString(Stream stream)

{

using (var reader = new StreamReader(stream))

{

return reader.ReadToEnd();

}

}

private static async Task<string> OpenInternal(StorageFile file)

{

var extension = Path.GetExtension(file.Name);

if (extension == ".gz")

{

// handle gzip files specially by dealing with them directdly

var gzipStream = new GZipStream(await file.OpenStreamForReadAsync(), CompressionMode.Decompress);

return CopyStreamToString(gzipStream);

}

else if (extension == ".htm")

{

// An uncompressed file, so we just open and return the contents

return CopyStreamToString(await file.OpenStreamForReadAsync());

}

else if (extension == ".zip")

{

// This is an archive file ... need to do the right thing here

// Remember that in unix you first create a .tar and then gzip it to .gz

// This means that the archive format ris the .tar and the compression is the .gz

// This also means that there is a bunch of ambiguity about what a .zip is

// A .zip is a package of compressed files. So it is the INVERSE of a .tar .gz

// We use ZipArchive to manage .zips

// How do we deal with the compression of the individual stream/files in the .zip?

// Open the local storage and call CreateFolderAsync to create a temporary folder with the name of the archive

var folder = await ApplicationData.Current.LocalFolder.CreateFolderAsync(file.Name, CreationCollisionOption.OpenIfExists);

try

{

// Copy the input stream to a MemoryStream to workaround bug 669923

using (var memoryStream = new MemoryStream())

{

using (var stream = await file.OpenStreamForReadAsync())

{

stream.CopyTo(memoryStream);

memoryStream.Seek(0, SeekOrigin.Begin);

using (var archive = new ZipArchive(memoryStream))

{

// Return the first opened file

if (archive.Entries.Count > 0)

{

return CopyStreamToString(archive.Entries[0].Open());

}

else

{

return "nothing in the archive";

}

}

}

}

}

catch (Exception e)

{

return e.Message;

}

}

return String.Empty;

}

// Public exposed function that maps an IAsyncOperation<string> to Task<string>

public static IAsyncOperation<string> Open(StorageFile file)

{

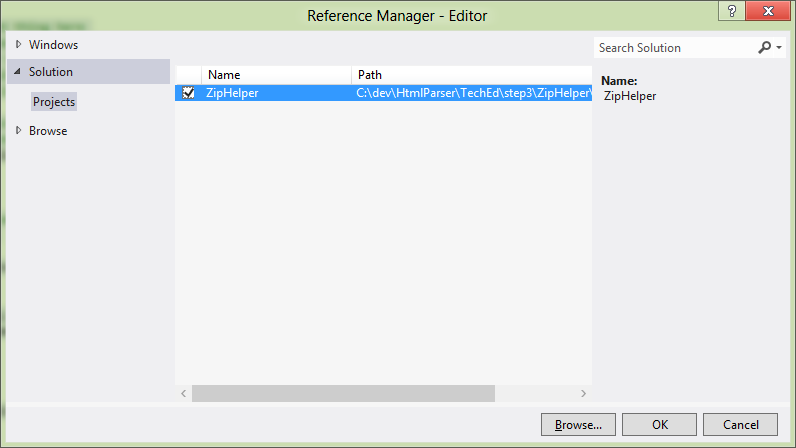
return AsyncInfo.Run(token => Zip.OpenInternal(file));

}

}

}

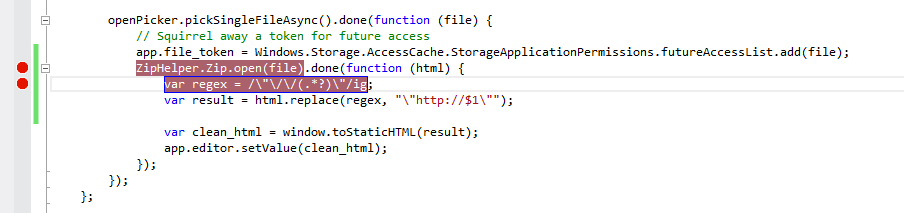
1. Add a reference from **Editor** to **ZipHelper**:



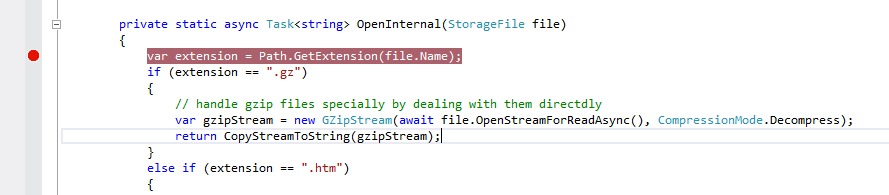
1. Replace the call to Windows.Storage.FileIO.readTextAsync(file) in **app.open\_file** with ZipHelper.Zip.open(file) to open a zip file:

ZipHelper.Zip.open(file).done(function (html) { …

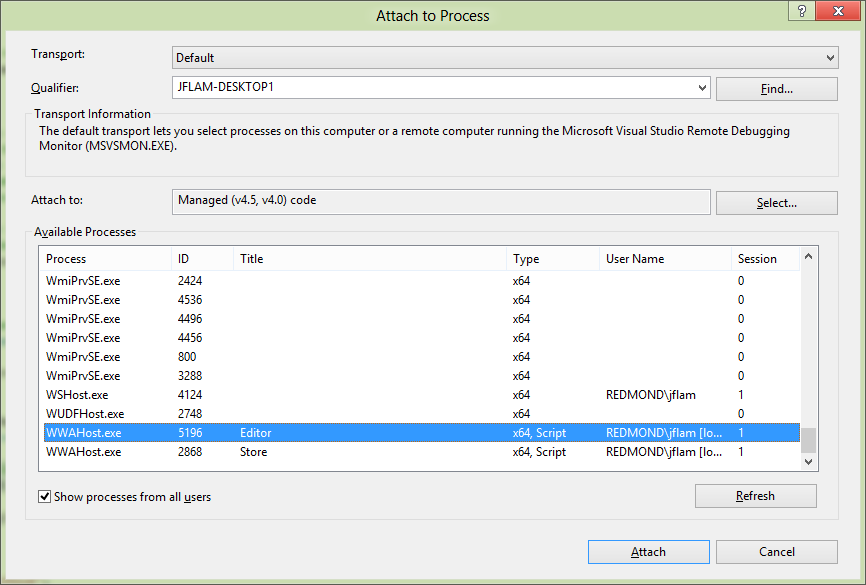
1. We can even debug this cleanly. Set two breakpoints, one on the call to ZipHelper.Zip.open() and one on the line inside of the lambda:



1. Open a new instance of Visual Studio and open your Editor.sln project in it. Open the Zip.cs file and set a breakpoint on the first line of code in the OpenInternal function:



1. Hit F5 in the first instance of VS. Make sure that the script debugger is selected.
2. Now select the Debug … Attach to process menu option. Make sure that the Managed (v4.5, v4.0) code debugger option is selected in the Attach to drop-down and that your app (hosted inside of WWAHost.exe – Editor app is selected.

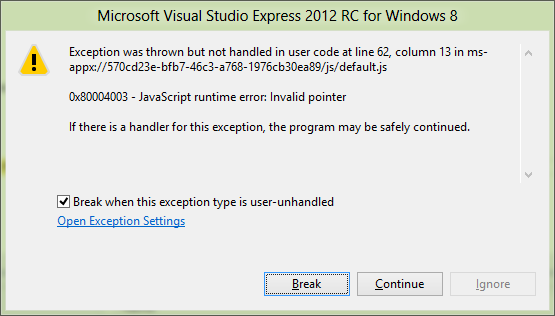


Now step into the app first function by opening a file. You should see that you are stopped at the breakpoing on the line of Zip.open. Now hit F5 to continue. You should now see that you are stopped in the other instance of Visual Studio in the C# breakpoint. Now hit F5 again and you should re-emerge on the other side in the JS delegate.

# Error handling

Let’s try and open a file that doesn’t exist? What happens?

1. Let’s force a null pointer exception by just dismissing the dialog box (we don’t handle the Cancel button case for return).
2. Open a file open dialog box
3. Select a file
4. Switch to Explorer
5. Delete the file that was selected
6. Switch back to the app and click on Open
7. Look at the dialog that we see:



1. Let’s run app by clicking on its tile in start menu and repeat the operation with the missing file. Notice how the app just returned to the start screen? This is what a user sees when your app crashes.
2. Let’s see if there’s any useful information there.
3. Run event viewer by hitting Windows key and **eventvwr**
4. Expand the Applications and Services event group and drill into **Microsoft/Windows/AppHost/Admin**
5. Copy and paste the General section and you’ll get some details of the JS exception that was thrown (lightly formatted):

App <App> crashed with an unhandled Javascript exception.   
App details are as follows: Display Name:<Editor>,   
AppUserModelId: <570cd23e-bfb7-46c3-a768-1976cb30ea89\_xw2yxt5n0ztta!App>   
Package Identity:<570cd23e-bfb7-46c3-a768-1976cb30ea89\_1.0.0.0\_neutral\_\_xw2yxt5n0ztta>  
PID:<3296>.   
The details of the JavaScript exception are as follows Exception Name:<WinRT error>, Description:<Invalid pointer>,   
HTML Document Path:</default.html>,   
Source File Name:<ms-appx://570cd23e-bfb7-46c3-a768-1976cb30ea89/default.html>,   
Source Line Number:<157>,   
Source Column Number:<28>,   
and Stack Trace:   
ms-appx://570cd23e-bfb7-46c3-a768-1976cb30ea89/default.html:157:28 Anonymous function()

1. **TODO: this is only applicable for C# apps. Not for WWA apps.** Now let’s turn on the local dumps feature that lets you collect local crash information.
2. Let’s turn on dumps. Create this reg key on the ARM device: **HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\Windows\Windows Error Reporting\LocalDumps**
3. Create a string value called DumpFolder and set it to c:\temp.
4. Create a c:\temp directory
5. Run and crash the app again. Now look in the directory.

Let’s post mortem debug that using WinDBG (and talk about how you can open up an x86/x64 dump directly using VS).

*TODO: how do I uninstall an app manually or force uninstall?*

1. Now let’s install onto ARM device for. First we need to install Remote Tools for Visual Studio 2012 RC. This will be pre-staged on the device since this tool isn’t readily available now.
2. Start up the debugger on the device and keep an eye on the IP address and/or machine name. Depending on local network configuration I may be able to connect using machine name or IP
3. First run experience on device will pop up Firewall configuration. Accept defaults and continue.



1. To deploy app from VS select Remote Machine from the toolbar:



1. Now run the app on the device by hitting F5. It will prompt you for the name of the device. Either get the IP address of the device or connect by name.



1. Type in your local account creds for the ARM device. If you are using a Microsoft account, use your Microsoft Account ID and password (it works!)
2. First run it will kick off a Get a developer license for Windows RT dialog on your machine. You will see this prompt:



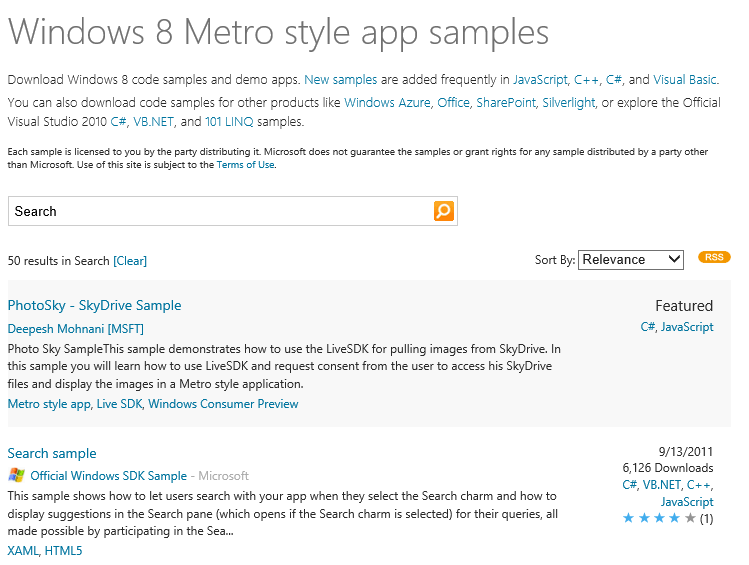
1. Enter your Microsoft account credentials to complete getting a developer license.
2. Switch back to your dev box and dismiss the dialog from before. You’ll now see your app starting up on your ARM device.

# Editing files directly from Wikipedia for fun and profit

This is more integration with the OS – using the Search contract

Let’s look up a sample

1. Got to <http://dev.windows.com>
2. Click on Metro style apps
3. Search for the Search sample



1. Download the Search sample and open: C:\samples\SearchContract\js
2. Run the sample and select scenario 5: Suggestions from Open Search
   1. Open search charm using Windows C
   2. Type something into the search box and see the suggestions appear
   3. 
3. Open up the code for Scenario 5 – it’s under js\scenario5.js
4. Copy the code that implements the **onsuggestionsrequested** event into your **app.onload** method.

// Provide suggestions using an URL that supports the Open Search suggestion format.

// Scenarios 2-6 introduce different methods of providing suggestions. The registration for the onsuggestionsrequested

// event is added in a local scope for this sample's purpose, but in the common case, you should place this code in the

// global scope, e.g. default.js, to run as soon as your app is launched. This way your app can provide suggestions

// anytime the user brings up the search pane.

Windows.ApplicationModel.Search.SearchPane.getForCurrentView().onsuggestionsrequested = function (eventObject) {

var queryText = eventObject.queryText, language = eventObject.language, suggestionRequest = eventObject.request;

// The deferral object is used to supply suggestions asynchronously for example when fetching suggestions from a web service.

// Indicate that we'll obtain suggestions asynchronously:

var deferral = suggestionRequest.getDeferral();

// This refers to a local package file that contains a sample JSON response.

// You can update the Uri to a service that supports this standard in order to see suggestions come from a web service.

var suggestionUri = "jsonSuggestionService/exampleJsonResponse.json";

// If you are using a webservice,the query string should be encoded into the URI. See example below:

//// suggestionUri += encodeURIComponent(queryText);

// Cancel the previous suggestion request if it is not finished.

if (xhrRequest && xhrRequest.cancel) {

xhrRequest.cancel();

}

// Create request to obtain suggestions from service and supply them to the Search Pane.

xhrRequest = WinJS.xhr({ url: suggestionUri });

xhrRequest.done(

function (request) {

if (request.responseText) {

var parsedResponse = JSON.parse(request.responseText);

if (parsedResponse && parsedResponse instanceof Array) {

var suggestions = parsedResponse[1];

if (suggestions) {

suggestionRequest.searchSuggestionCollection.appendQuerySuggestions(suggestions);

WinJS.log && WinJS.log("Suggestions provided for query: " + queryText, "sample", "status");

} else {

WinJS.log && WinJS.log("No suggestions provided for query: " + queryText, "sample", "status");

}

}

}

deferral.complete(); // Indicate we're done supplying suggestions.

},

function (error) {

WinJS.log && WinJS.log("Error retrieving suggestions for query: " + queryText, "sample", "status");

// Call complete on the deferral when there is an error.

deferral.complete();

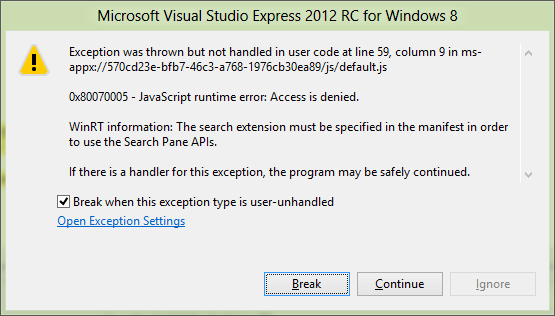
});

};

1. Add this line to the method as well:

var xhrRequest = null;

1. Modify **suggestionUri** to point to Wikipedia endpoint for Open Search: <http://en.wikipedia.org/w/api.php?action=opensearch&search>=
2. Uncomment the line of code that is below suggestionUri that will append a URI encoded query string to the URI.
3. Run the app. See this error:



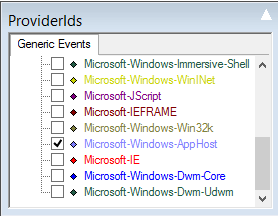
1. We need to remember to manifest the Search contract declaration. Double click on **package.appxmanifest** in the Solution Explorer. Click on the Declarations tab. Click on the Add button and select Search as a declaration



1. See the suggestions:



1. How does this thing run? Do we create another apartment for search activation? How do I see this?
2. Get the Windows Performance Analysis Tools from the [Performance Analysis Developer Center](http://msdn.microsoft.com/en-us/performance/cc825801)
3. Cool. I can see the app being activated from VS and I can see it being activated for Search as well. Navigate to Generic Events. Disable all ProviderIds and enable only the ProviderIds for **Microsoft-Windows-AppHost**.



1. Use flyout on left to filter everything down to just Generic events
2. You can get a list of providers installed for xperf using: xperf –providers i. interesting providers are Microsoft-Windows-AppHost, and
3. List all of the providers using Powershell via:
4. $p = xperf –providers i
5. Now generate a list and sort it using:

($p | select-string "Microsoft-Windows-.\*") | ForEach-Object { $\_.Matches[0].Groups[0].Value.Split(":") } | Sort

1. Now let’s get COM and AppHost events.
2. Not seeing them. Need to figure this out.

# Using the WACK

Now let’s get ready to submit our app to the store. Let’s run the packaging experience wizard from VS.

Let’s run the WACK

Let’s talk about what the WACK does while it is running

Complete … we can now submit …

# Post mortem debugging

We saw earlier how you can post mortem debug a Javascript app. We can do the same thing for a native C++ application as well. In this case we actually have more information available, since for native applications we’ll get a stack trace that contains locals and parameters as well.

1. Create a new Visual C++ XAML project



1. Add a button to the app bar:

<Page.BottomAppBar>

<AppBar >

<Button x:Name="Save" Style="{StaticResource SaveAppBarButtonStyle}" Click="Save\_Click" />

</AppBar>

</Page.BottomAppBar>

1. Add some code to the event handler:

void CrashApp::MainPage::Save\_Click(Platform::Object^ sender,  
 Windows::UI::Xaml::RoutedEventArgs^ e)

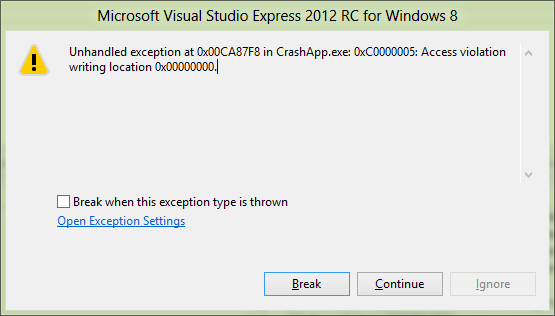
{

int \*j = 0;

\*j = 42;

}

1. Run the app and click on the Save AppBar button. You should see this:



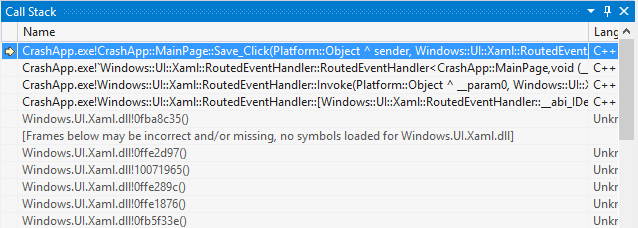
1. Now, let’s run the app outside of Visual Studio (you can do so using CTRL-F5). Recall that we have the local dumps feature enabled from our earlier session debugging a Javascript app.
2. You’ll find the dump in your c:\temp directory. It will be named CrashApp.exe.NNNN.dmp. Open the dump using Visual Studio:



1. When you click on the Debug with Native Only button, you’ll see this dialog appear:



1. This is the same as if the app had crashed locally under the debugger. Notice how the call stack is the same as in the live debugging session:



1. However, it is not the same as live debugging since the dump does not contain heap data, only stack data. However, in many cases this is sufficient for diagnosing the problem and preparing a fix.