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

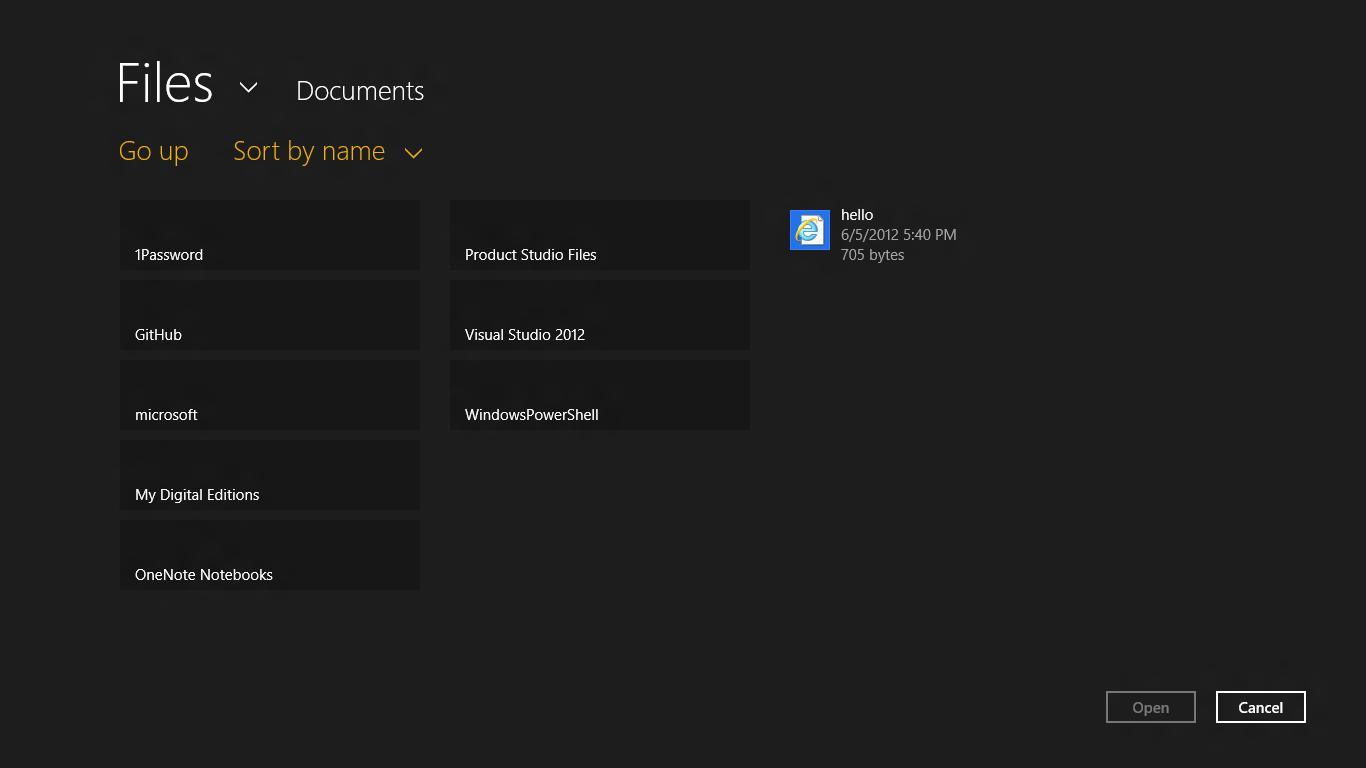
We will create a simple HTML editor in this document. Along the way, I will introduce you to a number of principles that underlie WinRT:

* Easily Create Apps
* Rich Apps that use all of Windows
* High Quality Apps

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



Before we begin, let’s take a tour of the app that we will build in this talk. When we run it, you will see that it is a split screen HTML editor. The HTML that you will edit is on the left hand side. Notice how the HTML is syntax colored. This happens using a 3rd party library called CodeMirror. On the right hand side we render the HTML whenever the user presses CTRL-ENTER. Taken together, this app **uses what you know** (HTML/CSS/JS) and **uses what you have**, CodeMirror, and packages it inside of a Metro style app.



Our app is integrated with the Metro style user experience and the user’s local files. Let’s open a file from the user’s file system. Notice how we can bring up a Metro style AppBar by swiping up from the bottom of the screen, or by pressing Windows-Z for folks who don’t yet have a touch screen. When we click on the Open button, you’ll see how we have what we call a File Open Picker dialog that lets the user browse for files in their file system. Both of these are examples of how we can **light up your app on the OS**. Users want apps to behave consistently across the OS, and the AppBar is one example of this. Users also want to use files that they already have on their computer, and the File Picker is a great way to enable this in your app.

Notice how this dialog is full screen; remember that Metro style is all about a touch-first experience, so the touch targets are intentionally large to make it easier for your customers to press them using their fingers. Once you select the file, notice how we can load it up directly in the HTML editor. If we hit CTRL-ENTER you can see that the file now renders.

Not only can our app open regular HTML files, it can also open .zip files via the .NET [ZipArchive](http://msdn.microsoft.com/en-us/library/system.io.compression.ziparchive%28v=VS.110%29.aspx) API. For simplicity, I’ve omitted a UI for picking files out of a .zip archive, instead choosing to open the first file that we find in the .zip archive and assuming that it is a HTML file. This introduces another key principle: **use the right language/runtime for the job**. Since we’re building an HTML editor, it makes sense that we use JS/HTML/CSS as our rendering runtime. However, since there is a tested and supported API for manipulating .zip files in the .NET libraries, it makes sense to use C# to grab the data.

Our app also uses the Search contract in Windows 8 to search for HTML pages on Wikipedia and provide as-you-type-it search suggestions. As it turns out, it is incredibly easy to take code from the [Search Contract Sample](http://code.msdn.microsoft.com/windowsapps/Search-app-contract-sample-118a92f5) the [Windows Developer Center for Metro style apps](http://msdn.microsoft.com/en-us/windows/apps/br229512.aspx) and copy-and-paste a key piece of code into your app to get it to work. This is the principle of **use what others know** in action.

All apps have bugs. It’s just a matter of whether you discover them or your users discover them. If we use the File Picker in this app and click on the cancel button, the app will crash (in this case by design). In this talk we’ll see how the platform can **help you when bad things happen** by seeing how Visual Studio can be used to do both live and post-mortem debugging of your app.

Windows 8 enables a broad and diverse ecosystem of devices. Some of those devices will run on CPUs with the ARM architecture. You should test your apps on those devices as well to ensure that you have a great user experience on machines that are not as powerful as your development machine. We’ll see how you can use platform features to **help you test your app** and to diagnose problems if your app crashes on those devices.

Finally the Store lets you distribute your app to hundreds of millions of users. The Windows Metro style SDK helps you rapidly onboard your app to the store by letting you run the same set of automated tests that the Store runs on your app when you submit it. This **helps you ship your app** faster since running tests ahead of time will let you catch and fix problems that could delay getting your app out to your customers.

Let’s start by building our app using what we know.

## An HTML page as an “app”

Let’s begin by creating an HTML page 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](http://codemirror.net/), a library for a syntax coloring text editor) to rapidly assemble the core of an application.

1. Create file called default.html that contains:

<html>

<body>

<textarea id="editor" cols="80" rows="10">&lt;h1&gt;hello, world&lt;/h1&gt;</textarea>

<div id="output"></div>

<script>

window.onload = function () {

editor.onkeyup = function (args) {

if (args.ctrlKey && args.key == "Enter") {

output.innerHTML = editor.innerText;

}

};

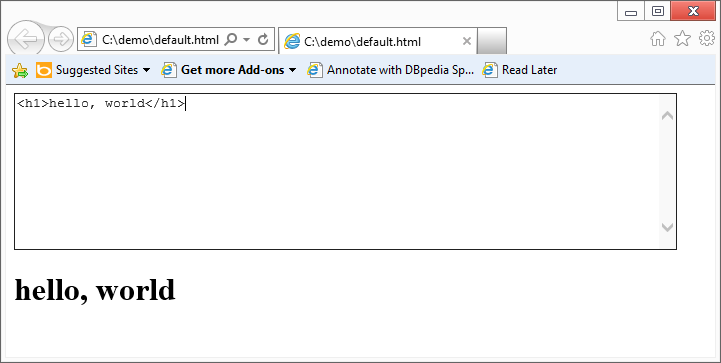
};

</script>

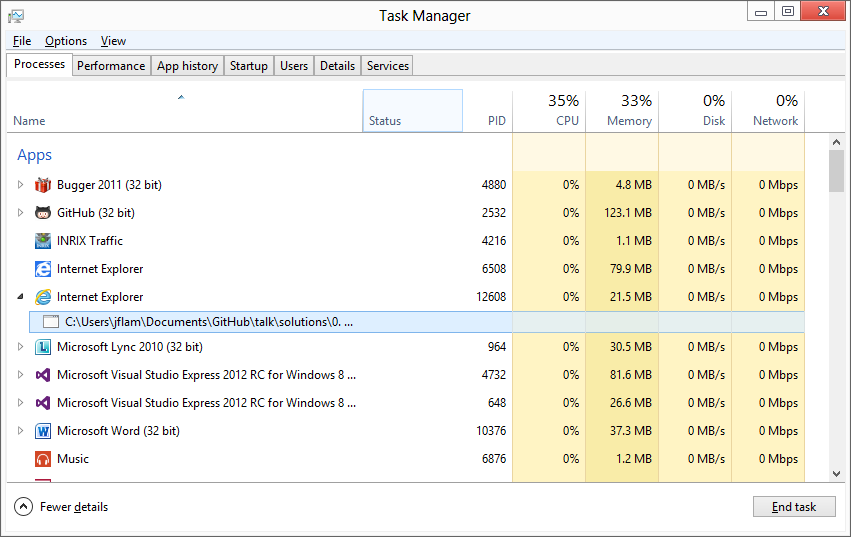
</body>

</html>

1. Note that we have an event handler bound to the CTRL-ENTER key that will take the text that the user entered in the <textarea> element and inject it into the **output** <div> element as HTML.
2. Open the file in Internet Explorer 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:



1. A small bit of CSS goes a long way; insert this <head> element into your code:

<head>

<style>

body {

font-family: "Segoe UI";

font-size: 18px;

background-color: black;

color: white;

}

#editor {

height: 90vh;

width: 50vw;

border: 0px;

position: absolute;

background-color: black;

color: white;

font-size: 18px;

}

#output {

left: 50vw;

width: 50vw;

top: 0px;

position: absolute;

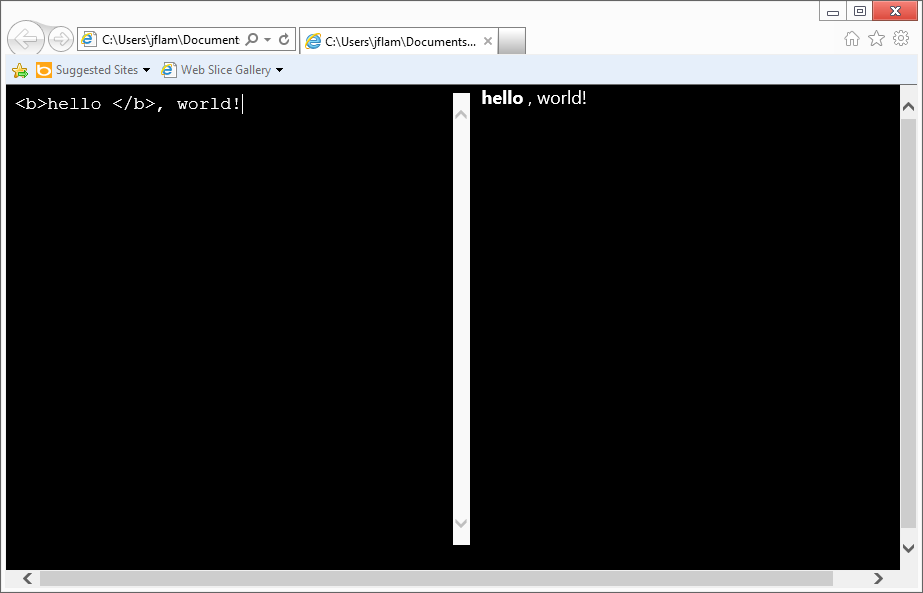
margin-left: 20px;

}

</style>

</head>

1. This is what you’ll see:



1. Now let’s go beyond <textarea>; let’s search the web to find a library that will do syntax coloring for a browser-based editor. If you search for “browser editor javascript” you’ll find [CodeMirror](http://codemirror.net/).
2. The library contains a number of files. Let’s integrate a few key files into your app. If you’re following along at home, copy these files into a directory called **js**:
   * codemirror.js
   * css.js
   * htmlmixed.js
   * javascript.js
   * xml.js
3. Copy these files into a directory called **css**:
   * codemirror.css
   * lesser-dark.css
4. Edit your default.html file and add some references to the JS files:

<!-- CodeMirror JS References -->

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

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

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

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

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

1. Add some references to the CSS files:

<!-- CSS References -->

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

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

1. Add some <style>:

<style>

body {

color: white;

background-color: black;

font-family: "Segoe UI";

font-size: 18px;

}

.CodeMirror {

font-size: 18px;

width: 50`%;

}

.CodeMirror-scroll {

height: 100vh;

}

#output {

position: absolute;

top: 0px;

width: 50vw;

left: 50vw;

height: 100vh;

margin-left: 20px;

}

</style>

1. Change your <textarea> to a <div>:

<div id="editor">&lt;h1&gt;hello, world&lt;/h1&gt;</div>

1. Replace your existing <script>:

<script>

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

lineNumbers: true,

indentUnit: 2,

theme: "lesser-dark",

keyMap: "html\_editor"

});

CodeMirror.keyMap.html\_editor = {

'Ctrl-Enter': function (cm) {

var html = editor.getValue();

output.innerHTML = html;

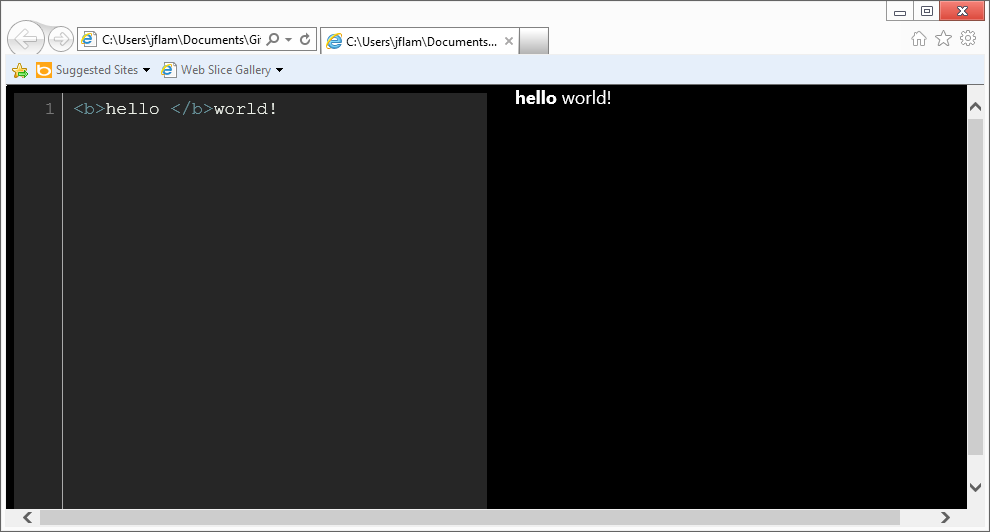
},

fallthrough: ["default"]

};

</script>

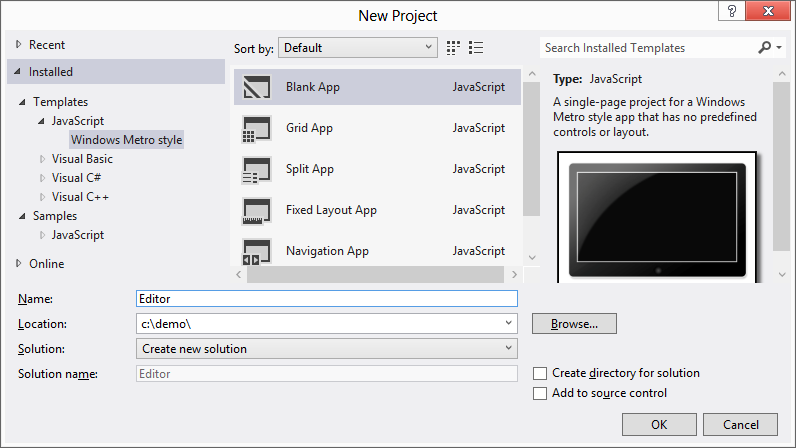
1. When you run this new version of the app, you’ll see that you have syntax coloring and a full-featured text editor:



## Turning our HTML page into a Metro style app

Now that you’ve seen how to create a simple web page “as an app”, let’s turn it into a full-fledged Metro style app that you can deploy to the app store and get paid (or get recognition, or drive recognition of your brand … it really is up to you!).

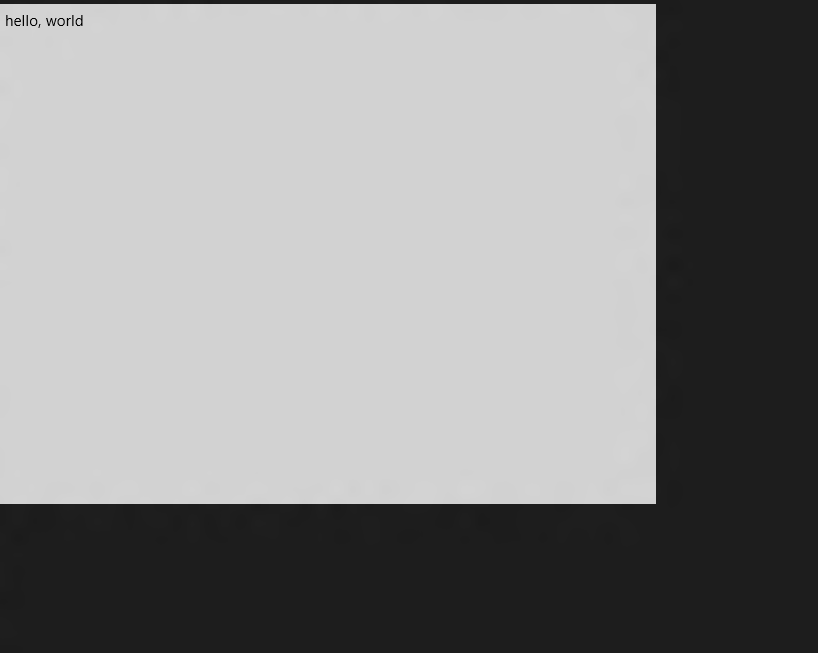
1. Startup Visual Studio and select File … New … Project. Name your project **Editor**.



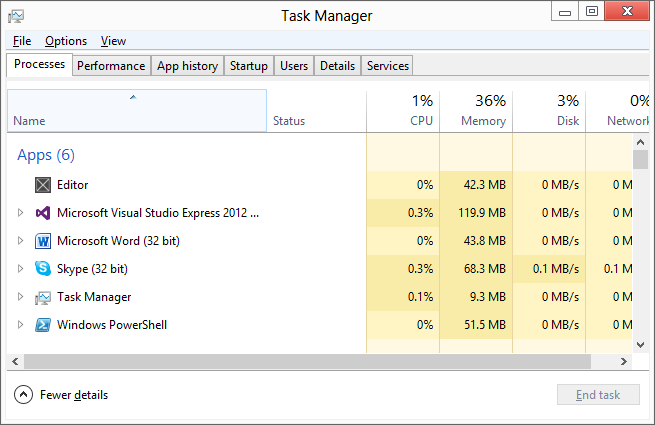
1. Navigate to **default.html** in the solution explorer, and replace the <p>Content goes here</p> HTML with:

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

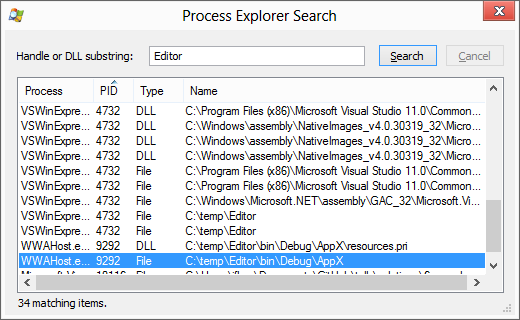
1. If you run the app, you should see a full screen Metro style app that does much like what you saw before:



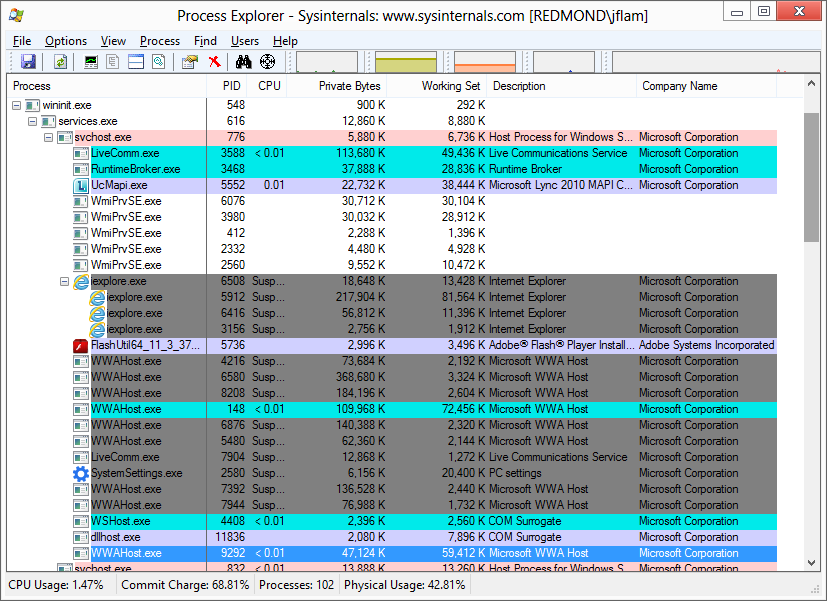
1. However, if we look under task manager, you’ll see a different view of the world than you saw before. You’ll see that Editor is now a top-level entry under Apps vs. being a category under Internet Explorer.



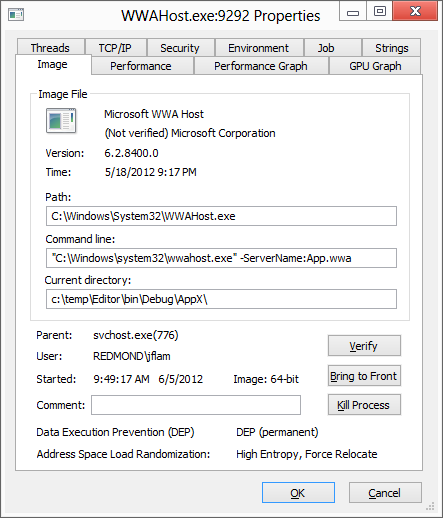
1. Let’s get some more information by running [procexp](http://technet.microsoft.com/en-us/sysinternals/bb896653.aspx) (Process Explorer). Start procexp from the command line and hit CTRL-F to search for our app. Search for a string called “Editor”. On my machine, it found a process called WWAHost that references my Editor directory. On my machine, its PID is 9292:



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

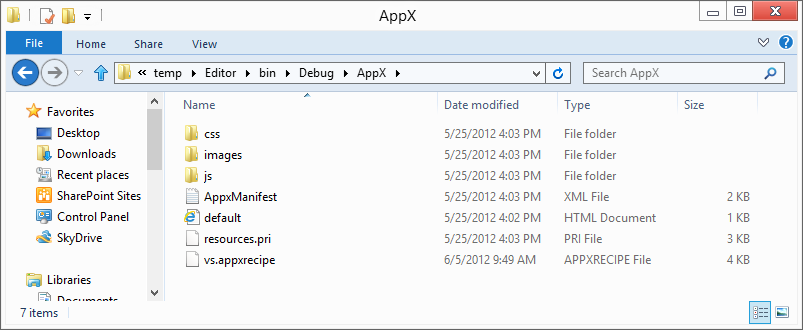


If we right click on WWAHost.exe and click Properties … we’ll see additional information about the process, including the command line used to start it and its current directory:

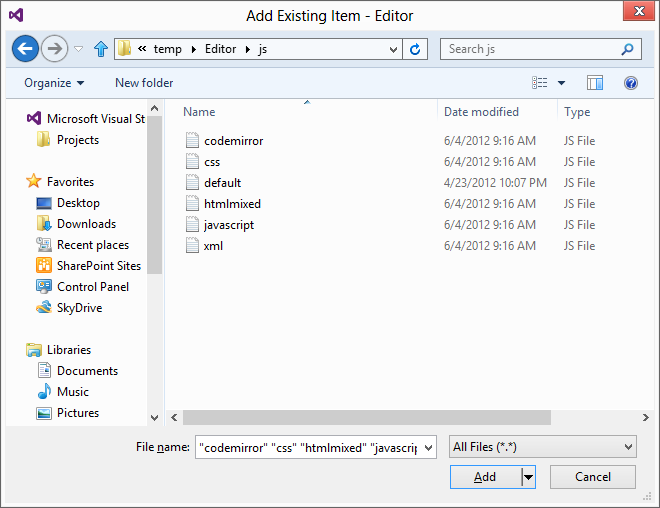


There are many more details here, but the key points are that we are launching your “application” now using a different host application: **WWAHost.exe** vs. **iexplore.exe**.

1. Another important distinction is that your app is running in a “current directory” – c:\temp\Editor\bin\Debug\AppX\ on my machine. If you look inside this directory you’ll see a copy of all of the project files from your app:



1. Because we are running under a debugger, you’ll see that VS will simply point Wwahost.exe at all of the files in your project to run it. However, if you package up your application for deployment or submission to the store, you’ll see that everything will wind up inside of an **.appx** file, which is really just a **.zip** file.
2. An important note here is that all of the files **must be in your project** if you want VS to copy them to this directory. Let’s add files from our existing app to this directory.
3. Copy the **codemirror.js, htmlmixed.js, javascript.js, css.js, and xml.js** files from the directory you used to create the app from the previous step to the **js** directory of your new project. Also the **codemirror.css, lesser-dark.css** files from that directory to the **css** directory in your new project.
4. 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>

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

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

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

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

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

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.html**:

<script>

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

lineNumbers: true,

indentUnit: 2,

theme: "lesser-dark",

keyMap: "html\_editor"

});

CodeMirror.keyMap.html\_editor = {

'Ctrl-Enter': function (cm) {

var html = WinJS.Application.editor.getValue();

output.innerHTML = window.toStaticHTML(html);

},

fallthrough: ["default"]

};

</script>

This is the first time that we’ve seen a reference to “WinJS” anywhere. In this case, you can see that we are creating a CodeMirror editor object and assigning it to the **editor** attribute of an object called **WinJS.Application**. Later in the event handler for the Ctrl-Enter keystroke, we retrieve this object. You should store attributes that have application-wide scope in the **WinJS.Application** object.

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

<style>

body {

color: white;

background-color: black;

font-family: "Segoe UI";

font-size: 18px;

}

.CodeMirror {

font-size: 18px;

width: 50%;

}

.CodeMirror-scroll {

height: 100vh;

}

#output {

position: absolute;

top: 0px;

width: 50vw;

left: 50vw;

height: 100vh;

margin-left: 20px;

}

</style>

1. When you run your app now, you should see:

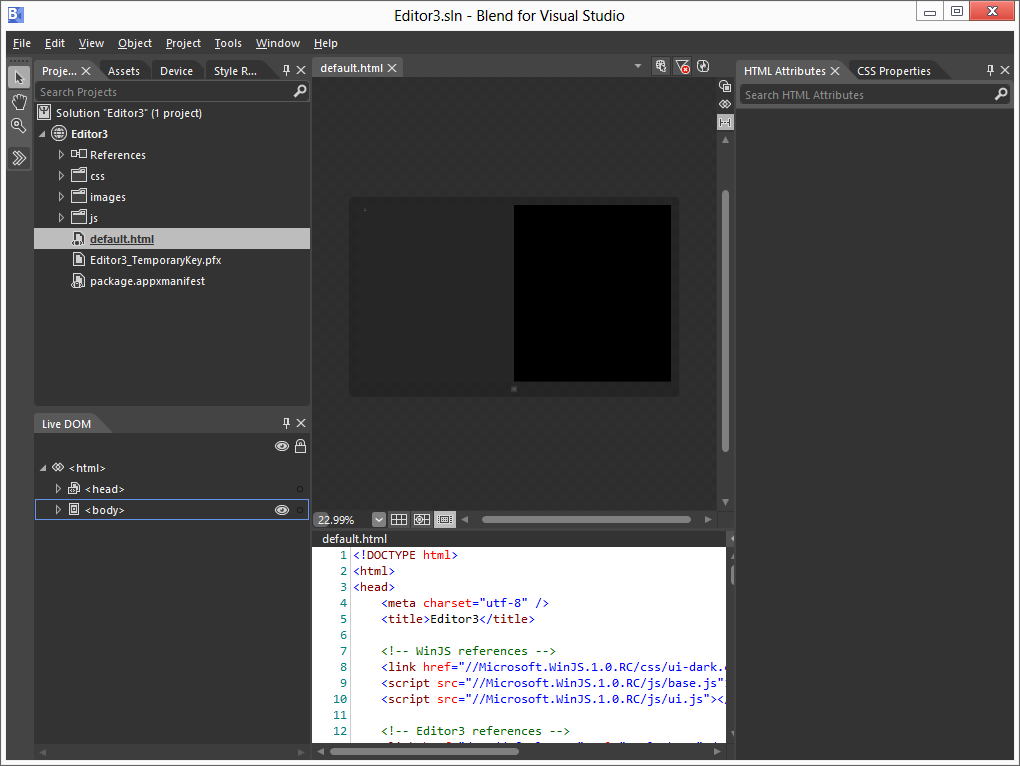


## Lighting up our app on Windows

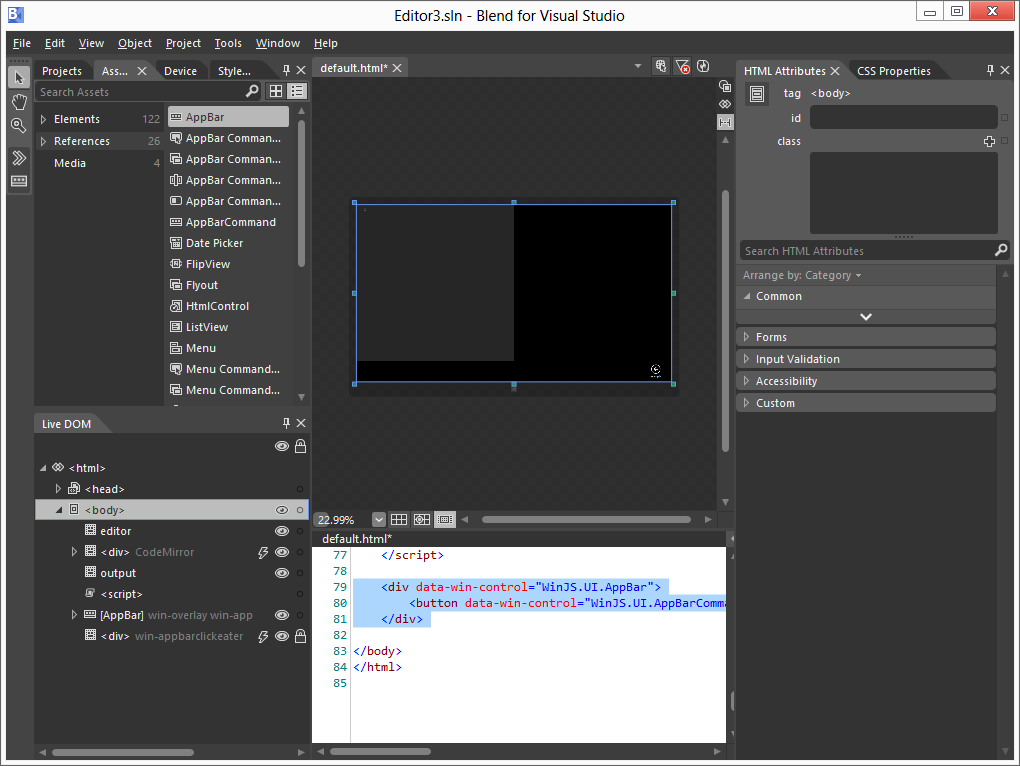
Up until now, we’ve implemented a local HTML editor that lives in a self-contained package. Now let’s do some work to integrate it into both the look and feel of the Windows 8 Metro style UI as well as adding support for interacting with files on the user’s computer.

In the first part, let’s add an App Bar with a couple of buttons using Blend.

1. Right click on **default.html** in VS and select Open in Blend:



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 Open button:

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

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

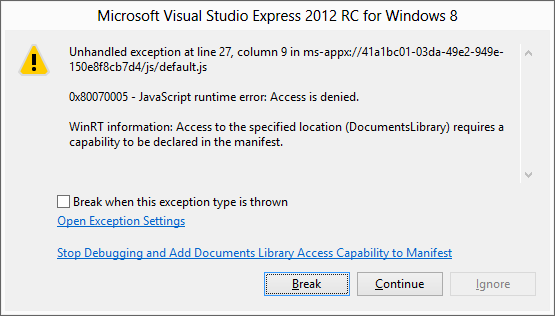
var file = Windows.Storage.KnownFolders.documentsLibrary.getFileAsync("hello.html")

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

};

Note that this code is referencing APIs in the **Windows.Storage** namespace. You should experiment with IntelliSense to examine other APIs in this namespace.

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. We won’t be doing this for this app, but for apps that need this kind of programmatic access (imagine a music player that enumerates all of the **.mp3** files in your music collection) they will need to do this explicitly in the manifest.
2. Instead, let’s write some code to open a file picker that lets the user choose the file to open.

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", ".zip"]);

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

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

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

app.editor.setValue(clean\_html);

});

});

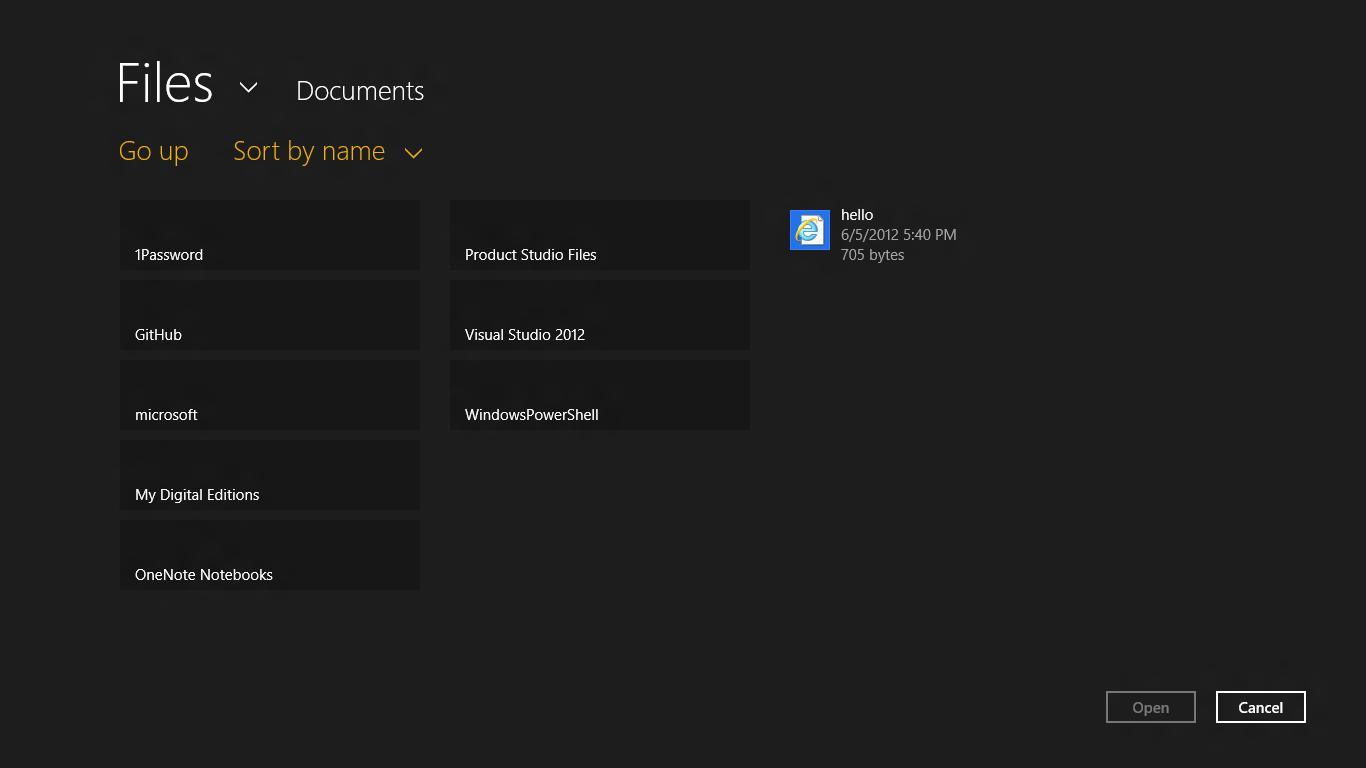
};

This snippet creates a FileOpenPicker object, configures it to browse for documents that end in a “.htm” extension in the user’s documents directory. What’s interesting in this code snippet is how you tell the FileOpenPicker to go and pick some files. Notice that we that call the **pickSingleFileAsync** method on the object. This is our first exposure to the asynchronous programming model in WinRT. To help users create fast and fluid applications, we deliberately made virtually all of the APIs in WinRT async. This means that the call to **pickSingleFileAsync** returns immediately. But what does it return?

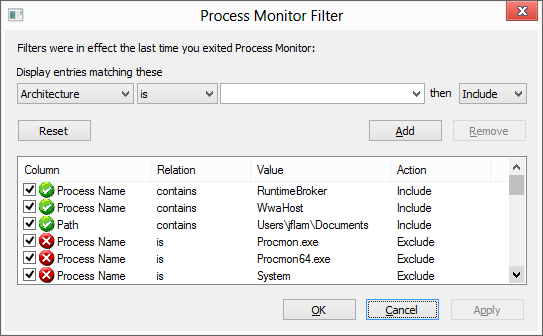
It returns a WinJS **Promise** object. The **done** method of the Promise object accepts (at a minimum) a completion function that is called once the user has picked a file. This function accepts a **file** parameter that can be used in subsequent code to manipulate the file that the user selected.

Notice that we can call another asynchronous function – the **readTextAsync** method on the **FileIO** object using the **file** object. The code here follows exactly the same pattern as the earlier call to **pickSingleFileAsync** – we pass a completion function to the Promise returned by the call to **readTextAsync**. In WinRT we have very strong naming conventions. You can expect that any method whose name ends in Async will return a WinJS Promise object.

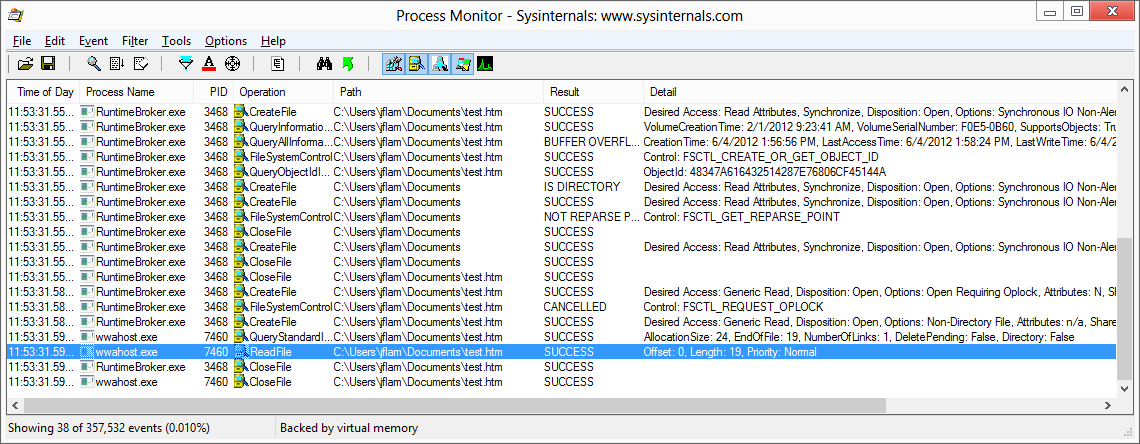
1. When you now run the application, and click on the Open command bar button, you’ll see the File picker experience:



1. Now let’s see if we can understand what caused that dialog box to pop up in VS when we tried to access the file programmatically. Let’s use [procmon](http://technet.microsoft.com/en-us/sysinternals/bb896645.aspx) to take a look at what’s manipulating the files that we are interested in. 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. Note that procmon shows us that there is a special process, **RuntimeBroker.exe** that first opens the file, and now we see that **WWAHost.exe** now reads the file, but without opening it. RuntimeBroker is responsible for opening the file and then marshaling the file handle back to WwaHost.exe who can now manipulate it.
2. Let’s complete our scenario by adding the ability to save our files. Let’s add a save button. 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 another handler. Put this after the other **addEventListener** call in **app.onactivated**:

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

1. Now let’s add the **save\_file** function, making sure that we use the FileSavePicker to open the file 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) {

// Squirrel away a token for future access

app.file\_token = Windows.Storage.AccessCache.StorageApplicationPermissions.futureAccessList.add(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 let’s add code to our **open\_file** API above to ensure that we squirrel away a token for future access when we open the file as well. Do this before we call **readTextAsync**:

// Squirrel away a token for future access

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

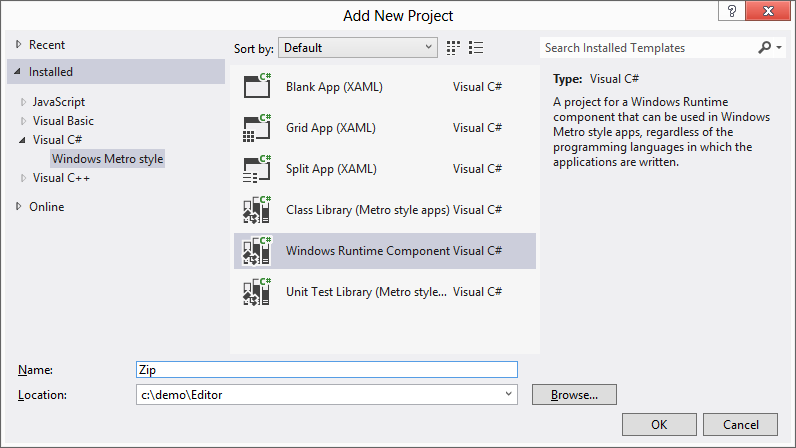
1. We now have a fully functional app that supports loading and saving to the file system. Note that we don’t need to add any additional capabilities to the app – programmatic access to the file system is not allowed.

## Mixing and matching code and libraries written with different languages

Let’s add an unusual, but not unreasonable capability to our app. 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.

If we look around, we’ll find a few JS libraries that can manipulate **.zip** files. However, if we look a bit harder we’ll see that there was a new ZipArchive class that was added in .NET 4.5. While that ZipArchive class is not a WinRT component, we’ll see that it is really easy to create a .NET WinRT component that calls ZipArchive on behalf of our app. Let’s do this now.

1. Add a New Project to the solution by right-clicking on the Solution node in Solution Explorer and selecting Add New Project from the popup menu. It must be a C# Windows Runtime Component called **Zip**:



1. Replace the existing code in **Class1.cs** with the following and rename the file 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 Zip

{

public sealed class Zip

{

// Inefficient copy

private static string CopyStreamToString(Stream stream)

{

using (var reader = new StreamReader(stream))

{

return reader.ReadToEnd();

}

}

private async static Task<MemoryStream> CopyToMemoryStream(StorageFile file) {

var memoryStream = new MemoryStream();

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

{

stream.CopyTo(memoryStream);

memoryStream.Seek(0, SeekOrigin.Begin);

}

return memoryStream;

}

// Private awaitable async function

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

{

using (var memoryStream = await CopyToMemoryStream(file))

{

using (var archive = new ZipArchive(memoryStream))

{

// Return the first opened file

if (archive.Entries.Count > 0)

{

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

}

else

{

return String.Empty; // default case where there aren't any files

}

}

}

}

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

public static IAsyncOperation<string> ReadFirstFile(StorageFile file)

{

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

}

}

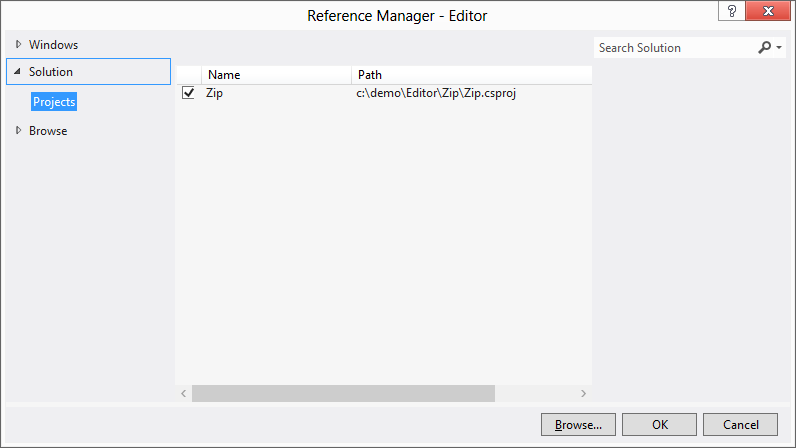
}

Let’s focus on the **ReadFirstFileInternal** function. If you have any familiarity with the asynchronous programming features recently introduced in C#, you’ll immediately recognize the use of both the **async** and the **await** keywords in this code sample. Since this method does a potentially lengthy operation, we want to make it asynchronous to conform to the fast and fluid principle of Metro style apps. Notice that we took some shortcuts in this function – we only open the first file in the .zip archive and we assume that it is a HTML file. A robust implementation with a UI to select the right file from the .zip archive is an exercise left to the reader ☺

Notice that the **ReadFirstFileInternal** method is private. It is called from the **ReadFirstFile** function, which is marked as a public method. By default, all public methods in a .NET WinMD assembly are callable using the WinRT calling convention. One additional thing to point out is that the WinRT calling convention mandates that asynchronous operations return an interface type of **IAsyncOperation<T>**. .NET asynchronous operations return a **Task<T>**. The **ReadFirstFile** method uses the **.AsAsyncOperation** extension method from the **System.Runtime.InteropServices.WindowsRuntime** namespace to adapt the .NET asynchronous calling convention to the WinRT asynchronous calling convention.

One additional point: if we look in the file system, we’ll see that we actually compile the file into a Zip.winmd file (see c:\demo\Zip\bin\debug).

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



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

if (file.fileType == ".zip") {

Zip.Zip.readFirstFile(file).done(function (html) {

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

app.editor.setValue(clean\_html);

});

} else {

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

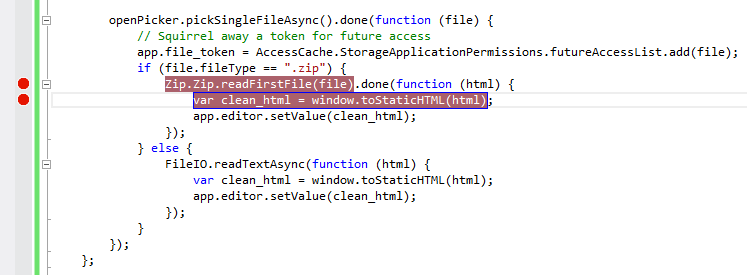
var clean\_html = window.toStaticHTML(html);

app.editor.setValue(clean\_html);

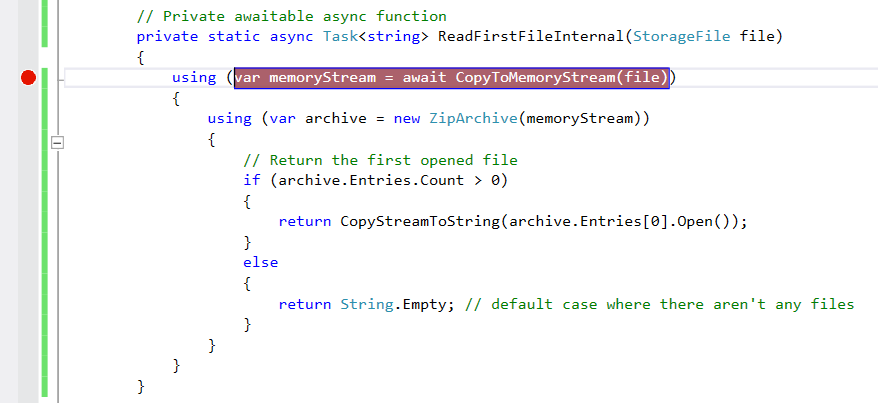
});

}

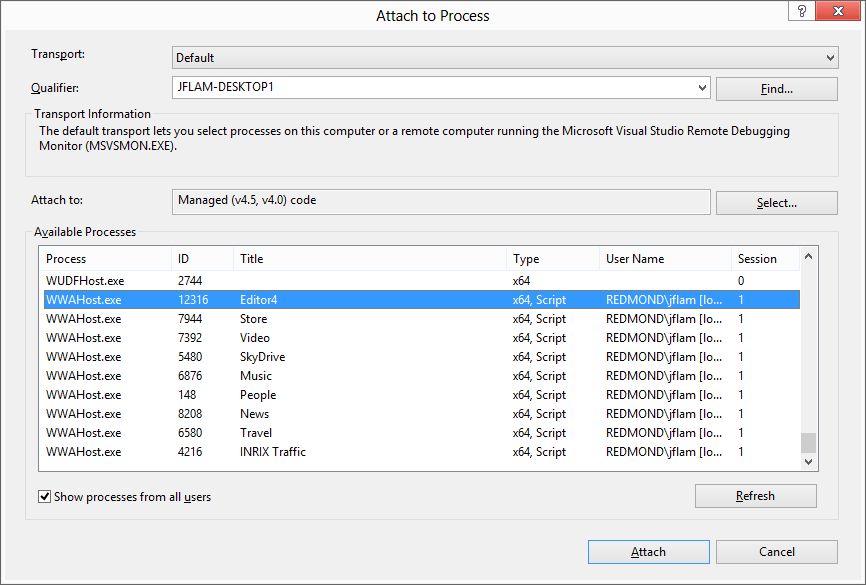
1. To see the WinRT calling convention, we can attach two debuggers to the process. Set two breakpoints, one on the call to Zip.Zip.readFirstFile() and one on the line inside of the callback function:



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 readFirstFileInternal 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.

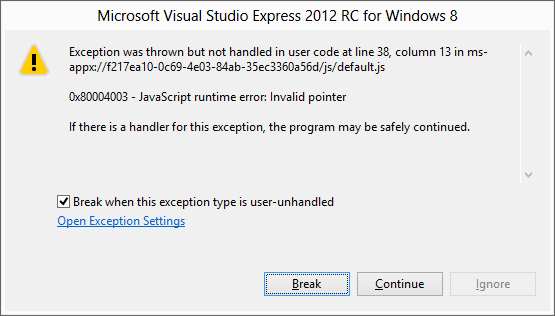


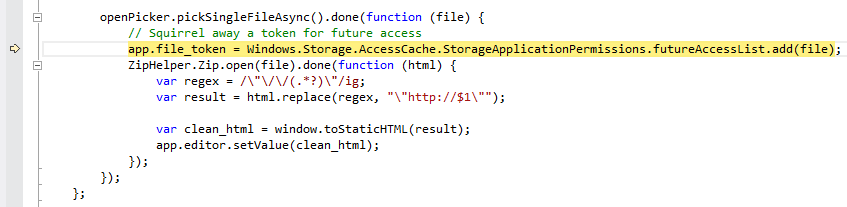
1. Now step into the app first function by opening a file. You should see that you are stopped at the breakpoint on Zip.open line. Now hit F5 to continue. You should now see that you are stopped in the other instance of Visual Studio in the C# breakpoint. Now hit F5 again and you should re-emerge on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

## Error handling

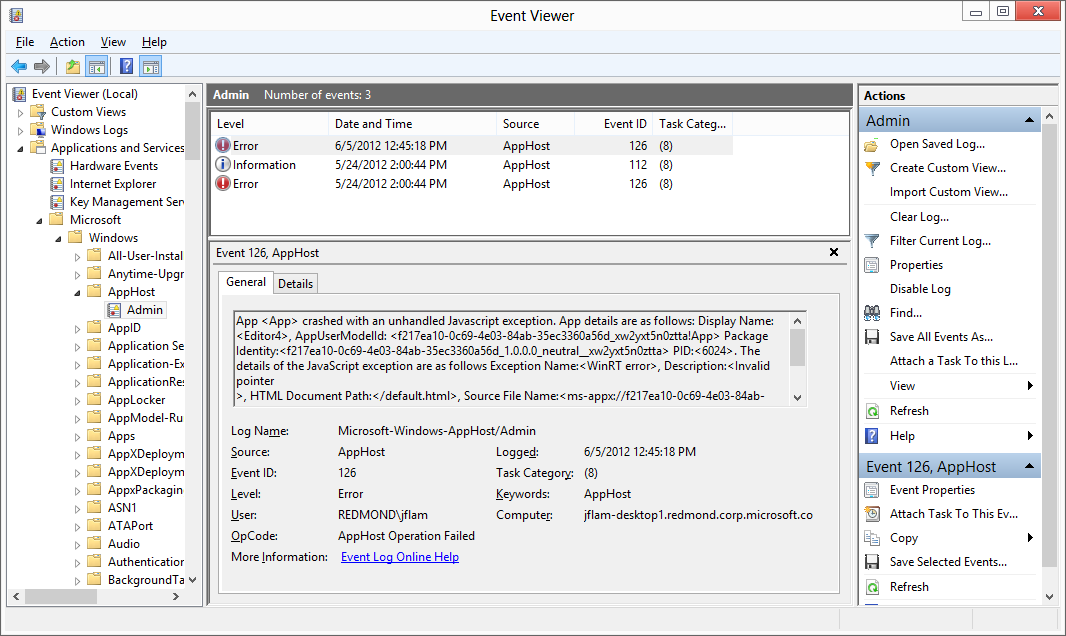
Let’s see what happens when our app does something that we don’t expect. There is a bug in the app – it will crash with a null reference exception if we click cancel instead of clicking on the open button in the File Open Picker.

1. Run the app using F5 in Visual Studio.
2. Open a file open dialog box and click on the cancel button. You should see this appear in VS with the line of code highlighted in the editor:





1. Now let’s run the app without the debugger attached using CTRL-F5 in Visual Studio. When you click Cancel did you notice that the app just “disappeared” and you were left looking at the Start Screen? This is the crash experience that a user experiences.
2. If we were testing this on a machine that didn’t have Visual Studio running, how would we know what happened? Fortunately, when an app crashes there is useful information deposited on your machine. Since we are building a Metro style HTML app, we can use Event Viewer to see what happened.
3. Click on the Windows key and type **eventvwr**. This will launch Event Viewer.
4. Expand the Applications and Services event group and drill into **Microsoft/Windows/AppHost/Admin**



1. 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:<Editor4>,

AppUserModelId: <f217ea10-0c69-4e03-84ab-35ec3360a56d\_xw2yxt5n0ztta!App>

Package Identity:<f217ea10-0c69-4e03-84ab-35ec3360a56d\_1.0.0.0\_neutral\_\_xw2yxt5n0ztta>

PID:<6024>.

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://f217ea10-0c69-4e03-84ab-35ec3360a56d/default.html>,

Source Line Number:<157>,

Source Column Number:<28>,

and Stack Trace: ms-appx://f217ea10-0c69-4e03-84ab-35ec3360a56d/default.html:

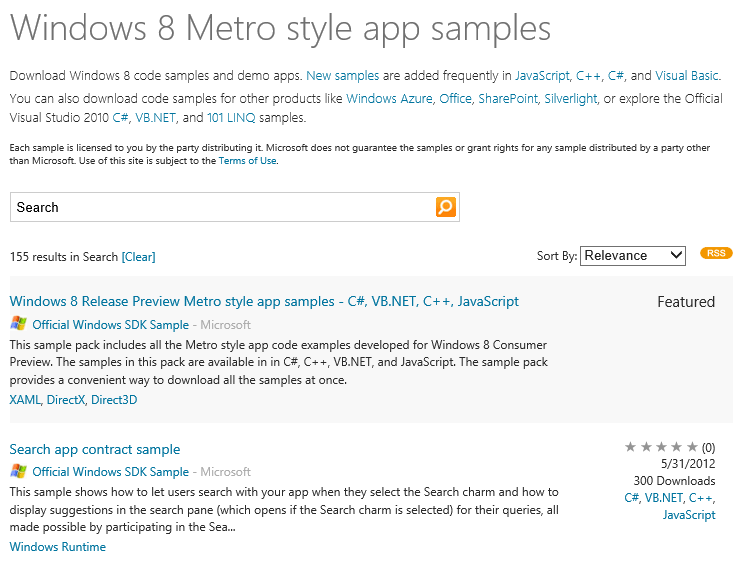
157:28 Anonymous function().

## Integrating the Windows Search experience

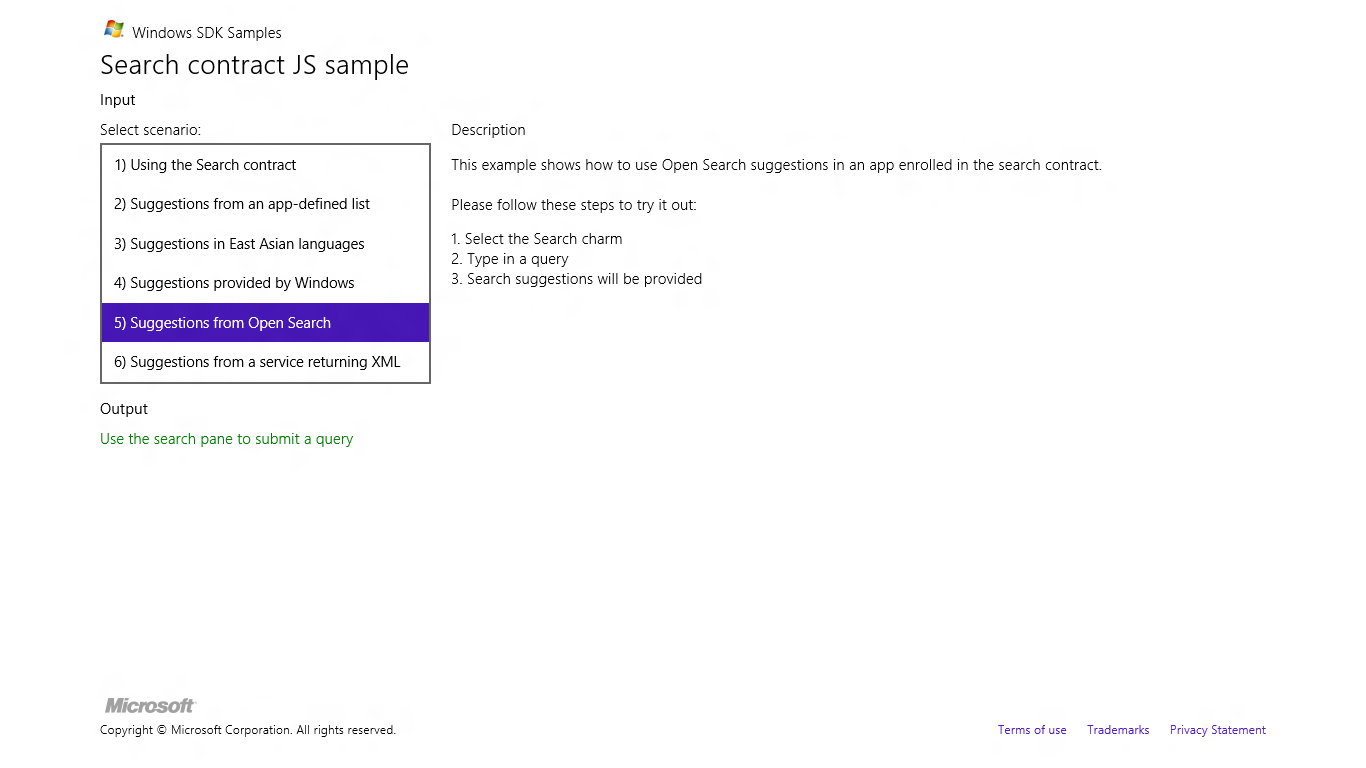
Integrating into Windows often involves adding support for one or more contracts. One of the most prominent new features of Windows 8 is the Search contract. It gives customers a consistent experience searching for content via the Search charm. Not only can you handle what the user types into the search box, you can also provide hints for what to search for as well. In this part of the app, we’ll see how we can integrate this into our app.

We’ve gotten consistent feedback from developers that one of the first places that they turn to for help are the Windows Metro style samples on the [Windows Developer Center](http://dev.windows.com/).

1. Got to <http://dev.windows.com> and click on Metro style apps. You can search for the Search sample:



1. Download the Javascript code for the Search app contract sample and open it up in Visual Studio and run via F5:



1. The highlighted scenario (5) is the one that we are interested in: Suggestions from Open Search.
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



1. Open up the code for Scenario 5 – it’s under js\scenario5.js
2. Let’s extract the code for search suggestions and add it to our app. 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();

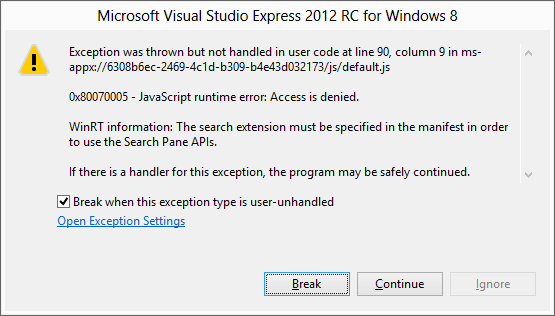
});

};

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. The final step in our app is adding the code that will retrieve a page from Wikipedia based on the search suggestion that was provided to the user. This lets us retrieve an article quickly without having to implement a full search results UI (that exercise is left to the reader ☺). We’ll replace the existing code in **app.activate** with code that uses the **WinJS.xhr** object to retrieve the HTML from Wikipedia. When your app is activated, Windows tells your app why it was activated. In this case we’re going to add some code to handle the case where we were activated for **ActivationKind.search**.

app.onactivated = function (args) {

if (args.detail.kind === activation.ActivationKind.launch) {

if (args.detail.previousExecutionState !== activation.ApplicationExecutionState.terminated) {

// TODO: This application has been newly launched. Initialize

// your application here.

} else {

// TODO: This application has been reactivated from suspension.

// Restore application state here.

}

args.setPromise(WinJS.UI.processAll());

} else if (args.detail.kind === activation.ActivationKind.search) {

var term = args.detail.queryText;

var url = "http://en.wikipedia.org/wiki/" + encodeURIComponent(term.replace(" ", "\_"));

WinJS.xhr({ url: url }).done(function (response) {

if (response.responseText) {

app.editor.setValue(response.responseText);

}

});

}

};

## 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

r

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)

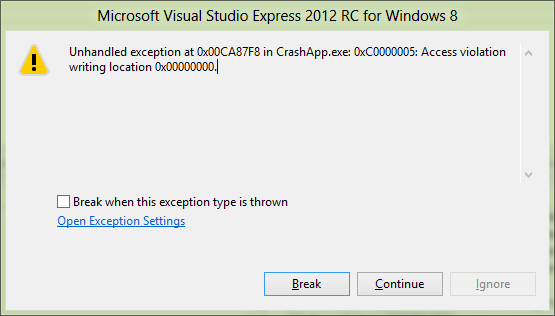
{

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). However, since this is a native application, we’ll need to turn on the local dumps feature of Windows Error Reporting (WER) so that we can collect a local crash dump.
2. You can create a local dump using this .reg file:

[HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\Windows\Windows Error Reporting\LocalDumps]

"DumpFolder"="c:\\temp"

"DumpType"=dword:00000001

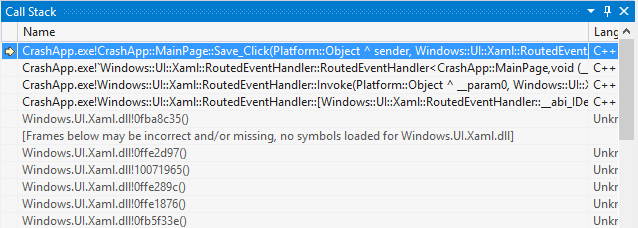
1. The **DumpFolder** string value identifies the directory where you want the dumps to be created in. The **DumpType** DWORD value tells WER that we want to collect a mini dump. For more details see [the MSDN doc page for activating Local Dumps](http://msdn.microsoft.com/en-us/library/windows/desktop/bb787181(v=vs.85).aspx).
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:



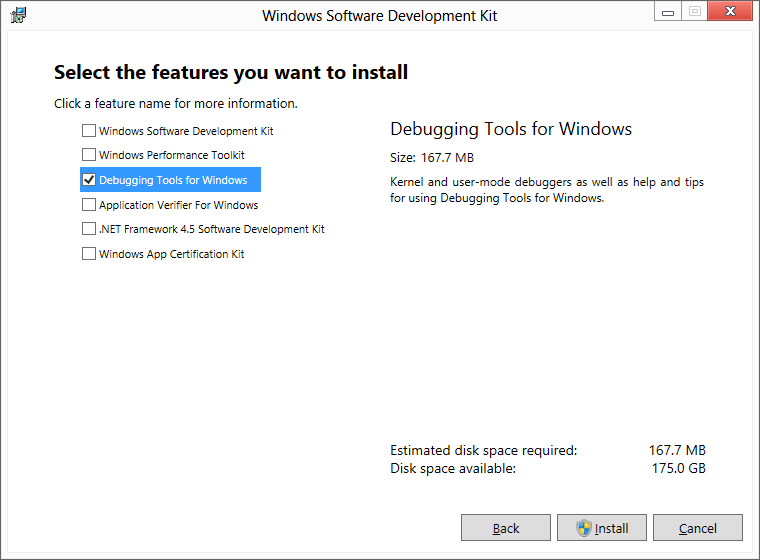
1. This is the same as if the app had crashed locally under the debugger. Notice how the call stack is the same as in the live debugging session:



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.

## Crash dumps on an ARM device

1. To debug crash dumps that come from an ARM device, you’ll need to install the Debugging Tools for Windows from the Windows Developer Center.



1. This is not intended to be a guide to how to use WinDBG – the online documentation that ships with the tool is excellent. However, we will walk through how to do a quick dump analysis using the built-in features of WinDBG.
2. You’ll need to setup the ARM device for remote debugging and deployment. 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 since ARM devices are not readily available now.
3. 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
4. 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:



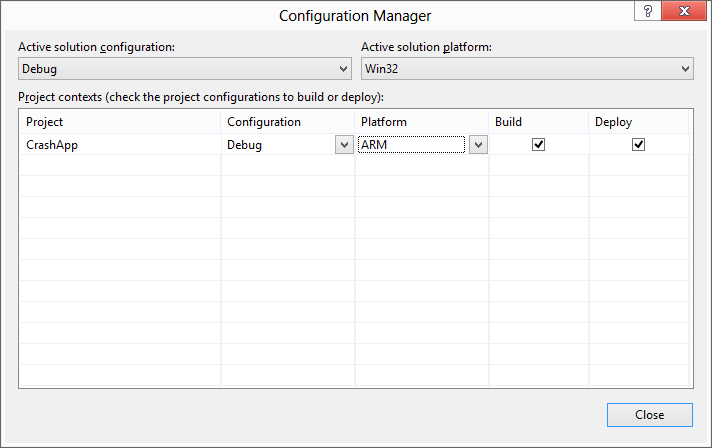
1. Now run the app on the device by hitting F5. It will prompt you for the name of the device. Either get the IP address of the device or connect by name.



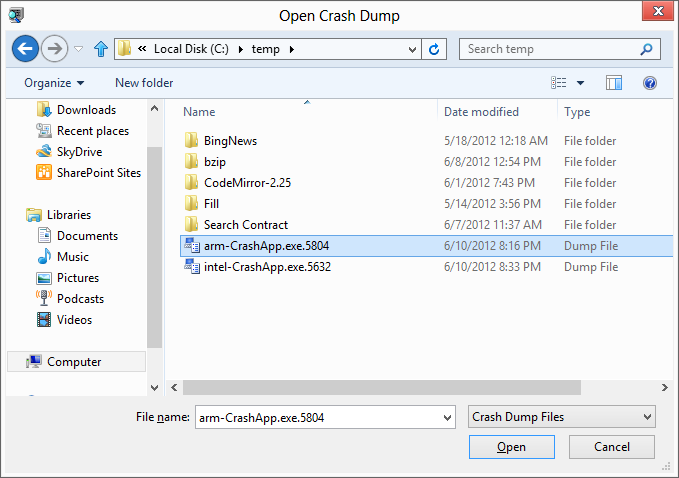
1. Type in your local account creds for the ARM device. If you are using a Microsoft account, use your Microsoft Account ID and password (it works!)
2. First run it will kick off a Get a developer license for Windows RT dialog on your machine. You will see this prompt:



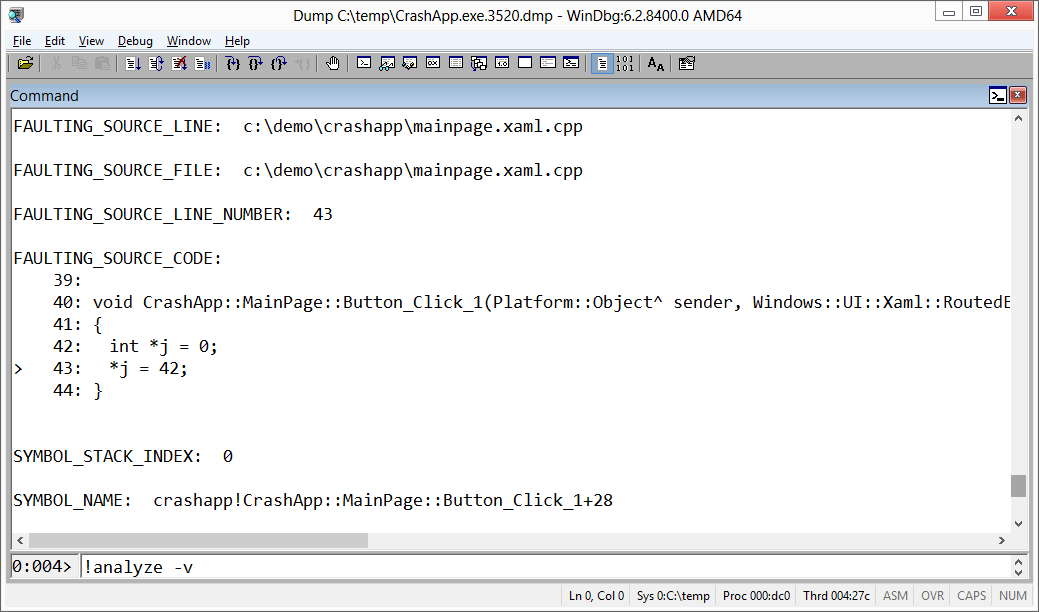
1. Enter your Microsoft account credentials to complete getting a developer license.
2. Switch back to your dev box and dismiss the dialog from before. You’ll now see your app starting up on your ARM device.
3. If you click on the button you’ll see the same crash experience inside of Visual Studio as you had before.
4. Now let’s do a post mortem debug. Recompile using the ARM target and deploy your app to your ARM device. You’ll need to switch your build configuration first:



1. Switch to the ARM device and run the app by clicking on the tile from the Start screen. The ARM device must be configured to collect local dumps using the same registry settings as we used earlier.
2. Open up the dump file using WinDBG:



1. Type !analyze –verbose. This should give you a nice source-based dump of where the crash occurred:

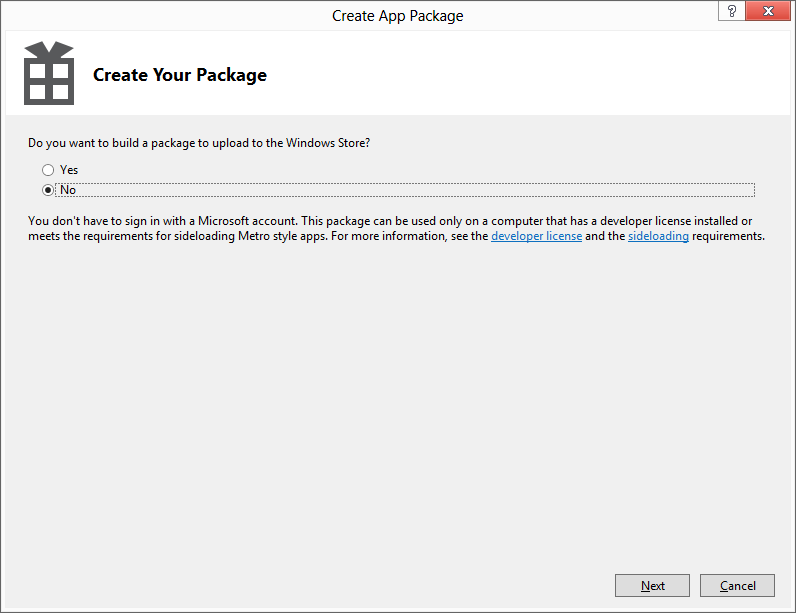


## Using the WACK

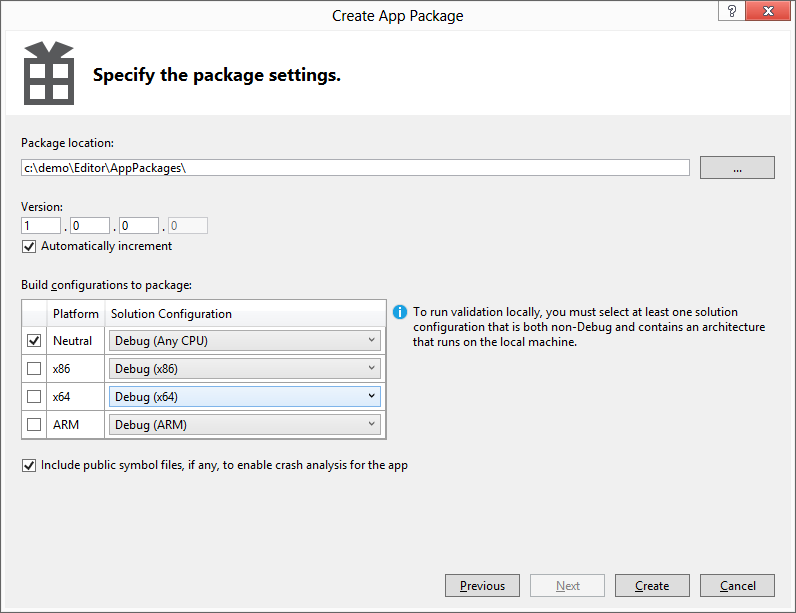
Now let’s get ready to submit our app to the store. Let’s run the packaging experience wizard from VS.

* + 1. To prepare your app for upload to the store you’ll first need to create an app package. Do this by selecting Create App Package from the STORE menu in VS.

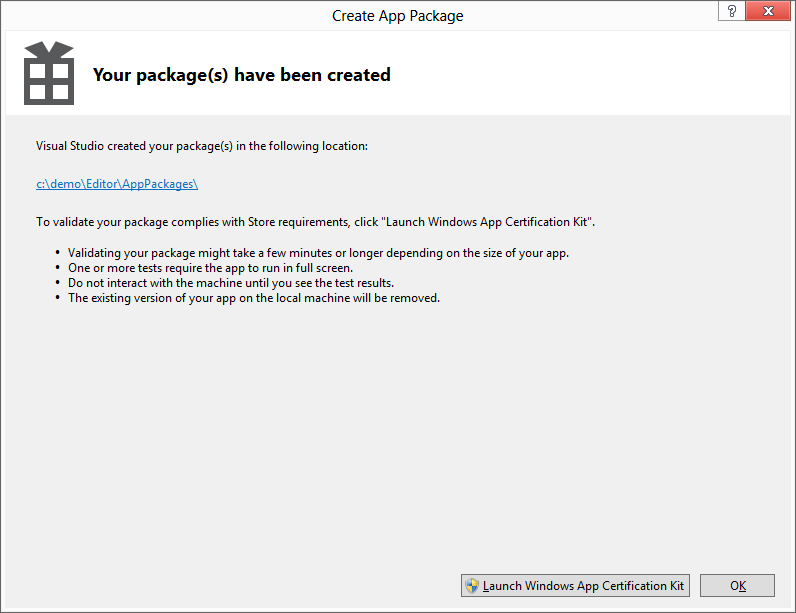
1. Make sure to compile as Release mode
2. Say No to the first dialog – you want to create a local package as opposed to a package that can be uploaded to the Store. For that to work you’ll need a Store developer account, and as of the time of this writing the Store is only open by invitation only.



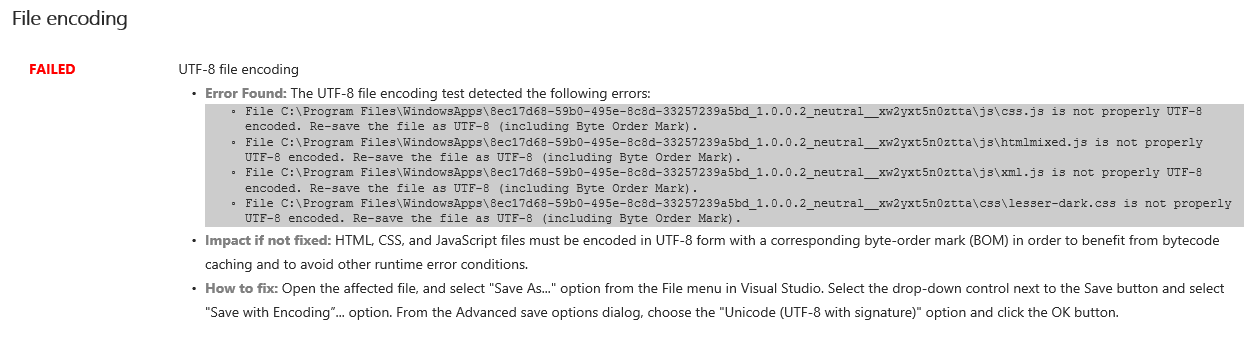
1. Select the defaults on the next page:



1. Click on the Create button to complete creating your package.
2. Click on the Launch Windows App Certification Kit button in the next dialog:



1. Now sit back and wait for the results. Note that encoding errors are also common if you use 3rd party libraries – you will need to open and re-save the files in Visual Studio using UTF-8 with a byte-order mark.



1. This is what a SUCCESS result looks like:

