## Preparation

1. Open ExtensibilitySample in Visual Studio
2. Open Extensions\GreyscaleImageExtension in Visual Studio
3. Uninstall #AppExtensibility Greyscale, AppExtensibility Rotate and AppExtensibility Scale apps

# Demo 1 – LaunchUriForResults

* Run ExtensibilitySample
* Explain it is a broad sample used in the Build App Extensibility talk, and also shows Cropping by calling out to the Photos app using LaunchUriForResultsAsync
* Click the Crop button – it opens a File Picker so pick a photo.
* Show the code behind the Crop Button Click – **EditorTab.xaml.cs** method **Crop\_Click** and **CropImageAsync**

# Demo 2 – App Services

* Run the ExtensibilitySample app, click the Open button to open an image, then click the Greyscale button. The dialog comes up to tell you you need another app.   
  Show the code in **EditorTab.xaml.cs** method **Greyscale\_Click** to show how that happens
* Point out that the App Service it is trying to connect to is defined by:

connection.AppServiceName = "com.microsoft.grayscaleservice";

connection.PackageFamilyName = "AppExtensibility.Extension.Grayscale\_byq669axdz8jy";

* Switch to VS with the GreyscaleImageExtension. Show the package manifest (GUI editor) – show the Packaging tab where the PFN is shown – matches what the clientside code specifies. Show the Declarations tab where the App Service is defined.
* Show code in GreyscaleService.Service which actually does the greyscale processing. Talk through the App Service operation:
  + **Run** method executes when the App Service is activated when the incoming connection comes in. RequestReceived event is wired up in there to handle incoming messages
  + **RequestReceived** handles incoming messages which are just ValueSet objects – like a Dictionary. In this case it contains the Command which can be Greyscale or Load, both of which do the same thing. It also contains the image pixels as a byte array in bgra8 format – the GreyscaleByteArray method takes the incoming byte array and processes it to greyscale each of the pixels.
  + Last line of RequestReceived sends the data back in another valueset
* Deploy (don’t run) the Greyscale app. Then run the original app and show that it works. Show the code in **EditorTab.xaml.cs** method **Greyscale\_Click** to show the client side of the app service communication

# Demo 3 – App Extensions

* Leave the app running
  1. Run the ExtensibilitySample app again, but choose the Extensions tab.   
     It Lists an extension for the greyscale app. We will explain why that is there in a moment, but before we do that…
* keeping the running app visible – from another copy of VS, deploy (don’t run) the **Invert** extension app – Immediately, a new entry turns up on the Extensions tab. Then install the **Rotate** extension app (use the packages this time, just for variety). Immediately, two more entries turn up on the Extensions tab. Enable the new extensions and show that they show up as buttons on the main UI. Use them to invert and rotate an image.
* Show the code:
  1. First show appxmanifest of **ExtensibilitySample** as code – point out the windows.appExtensionHost name

<uap3:Extension Category="windows.appExtensionHost" xmlns:uap3="http://schemas.microsoft.com/appx/manifest/uap/windows10/3">

<uap3:AppExtensionHost>

<uap3:Name>build2016.appextensibility.demo</uap3:Name>

</uap3:AppExtensionHost>

</uap3:Extension>

* 1. Show appxmanifest of rotate extension – declares two app extensions using the same name
  2. Show that the Public and Public2 folders in this case contain html files containing javascript
  3. In host, open **ExtensionManager.cs**
     1. Show how the constructor is called from App.xaml.cs and how it opens the AppExtensionCatalog (a machine wide catalog) filtering on the contract name, which is "build2016.appextensibility.demo" in this case.
     2. Show the Initialize method (called from MainPage.xaml.cs) that hooks up event handlers so gets invoked when new app extensions are installed or uninstalled on this machine.
     3. FindAllExtensions()
     4. Catalog\_PackageInstalled()
  4. Find **Extension** class (same file) – explain that this encapsulates all data associated with a App Extension that we discover. In particular, the InvokeLoad method is responsible for executing the extension code
     1. Extension.InvokeLoad() line 353 The host executes the javascript in the extension
     2. Point out that this is less secure than App Services, because the store cannot check for malicious code in javascript in the same way it can for compiled code.
* Show the Greyscale extension
  1. Now back to the **Greyscale** sample, why was that listed on the Extensions page?
  2. That’s because that app not only supports ‘traditional’ direct app service invocation that we walked through in the previous demo, but also demonstrates an alternative way for a client app to discover the availability of an app service, and to allow it to be enabled/disabled from the client app – through App Extensions.
  3. Show the appxmanifest as code: Point out that it has an AppExtension declared that uses the same name as the host app. It also has custom properties declaring the AppService name Note: doesn’t have to declare the PFN here because the host can find that out when it loads the AppExtensions data

<uap3:Extension Category="windows.appExtension">

<uap3:AppExtension Name="build2016.appextensibility.demo" Id="base" PublicFolder="Public" DisplayName="Grayscale" Description="Grayscale Image">

<uap3:Properties>

<Service>com.microsoft.grayscaleservice</Service>

</uap3:Properties>

</uap3:AppExtension>

</uap3:Extension>

* 1. – line 361 (expand) calls app extension App Service
  2. **Tip**: A UWP can’t do LoadLibrary to dynamically execute loaded code - but this technique with app services allows pseudo dynamic execution of code
  3. In EditorTab.xaml.cs, set visibility of the Greyscale button used for direct app service invocation to Hidden. Run the app again and enable the app extensions – now we have the same functionality as before, but with the additional capability of getting notified when the app extension is installed/uninstalled and the additional flexibility that the app extensions framework gives us.
* Show uninstallation
  1. While the app is running, Uninstall Rotate from Apps menu – it disappears automatically from the app because our app learned about the uninstallation through events we registered in ExtensionManager\Initialize method
  2. Uninstall Greyscale from the Remove button on the extensions tab in the app – show that you can do this in-app as well

# Demo 4 – Project Rome

Warning: this demo can fail when run over conference wifi. You should enable Bluetooth connections on your machines so that proximal messaging was used, then it should work OK.

There’s a slight ‘gotcha’ right now for proximal connectivity – you have to go to Privacy Windows Settings and check the ‘Allow Bluetooth Connections’ box which is off by default – there is no API for a dev to query current state of this setting unfortunately.

To run the demo, you need two machines, both running 14393+, and both associated with the same MSA.

Install the Roman test app on both machines: <http://aka.ms/romeapp> . Run the app on one machine, connect to the other machine from the app, then simply draw with a pen in the box bottom right of the app UI. The same app should launch on the other machione and the ink strikes appear there too.

The interesting teaching point about this demo is that you can ink on the source machine and on the target, the app launches and then echos the ink strokes. How does this work? There would have to be some kind of messaging going on between the foreground app on one system and the foreground app on the other system – there aren’t any UWP APIs that allow this.

Answer is that the source app lauches the app on the other system when it detects two or more ink strokes entered on the source, and then it launches the App Service for the same app on the target and sends the ink stroikes data in a ValueSet to the target app service. Clever bit here, is that the Roman Test App uses the new ‘in-proc’ background tasks capability in 1607, so the App Service is running in the same process as the foreground app, so the app service can easily communicate the received ink strokes to the foreground app functionality through a simple InkStrokesReceived event that it implements.