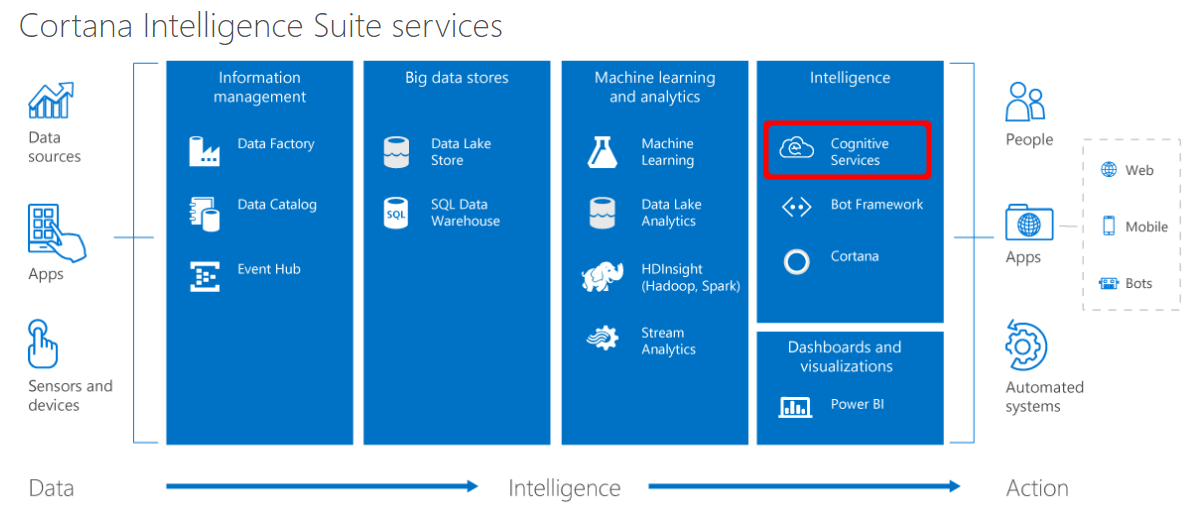
Using Cognitive Services

Access Control LAB using Image Recognition in combination with Cognitive Services

# Summary

This lab explores image recognition using Cognitive Services.



The lab will walk you through accessing a Web Camera locally on your Raspberry Pi. It will also walk you through signing up and using a few Cognitive Services to validate the identify of a person through face recognition.

## Business Case

Who has access to this location?

For this lab, you will be using Cognitive Services in combination with a live video stream, available locally on a Raspberry Pi 3 device running Windows 10 IoT Core. In the lab we take pictures from the video stream to show some of the capabilities and uses of the Cognitive Services Face API. More information about the Face API can be found here: <https://azure.microsoft.com/en-us/services/cognitive-services/face/>.

We will not use the REST API, but the **Windows SDK for the Microsoft Face API**. This is a wrapper around the REST API, making it easier to use the Face Service.

To limit the amount of time entering code, we already provide the UI for the application. You can download the initial solution that we are going to use here: [CognitiveServicesLab-InitialSolution](https://github.com/Maarten-Work/Bootcamp-2017/releases). This project contains all the XAML for the UI and a number of empty event handlers to take action on button clicks. We will add code to control the camera and to call the Face API cognitive service to create an access controller system.

# Controlling a USB camera on the Raspberry Pi

For this lab, we will not write a full-blown video application, but provide just enough code to initialize the camera, to display a video stream and to take individual pictures form the video stream. Because Windows 10 IoT Core has no GPU support on Raspberry Pi, the video quality is not optimal but good enough for this particular Lab.

## Step 1 - Verify if the camera connects correctly to the Raspberry Pi

1. Connect a USB camera to one of the available USB ports of the Raspberry Pi
2. The camera is connected and recognized if you see it appear in the list of **Connected devices** inside the default application

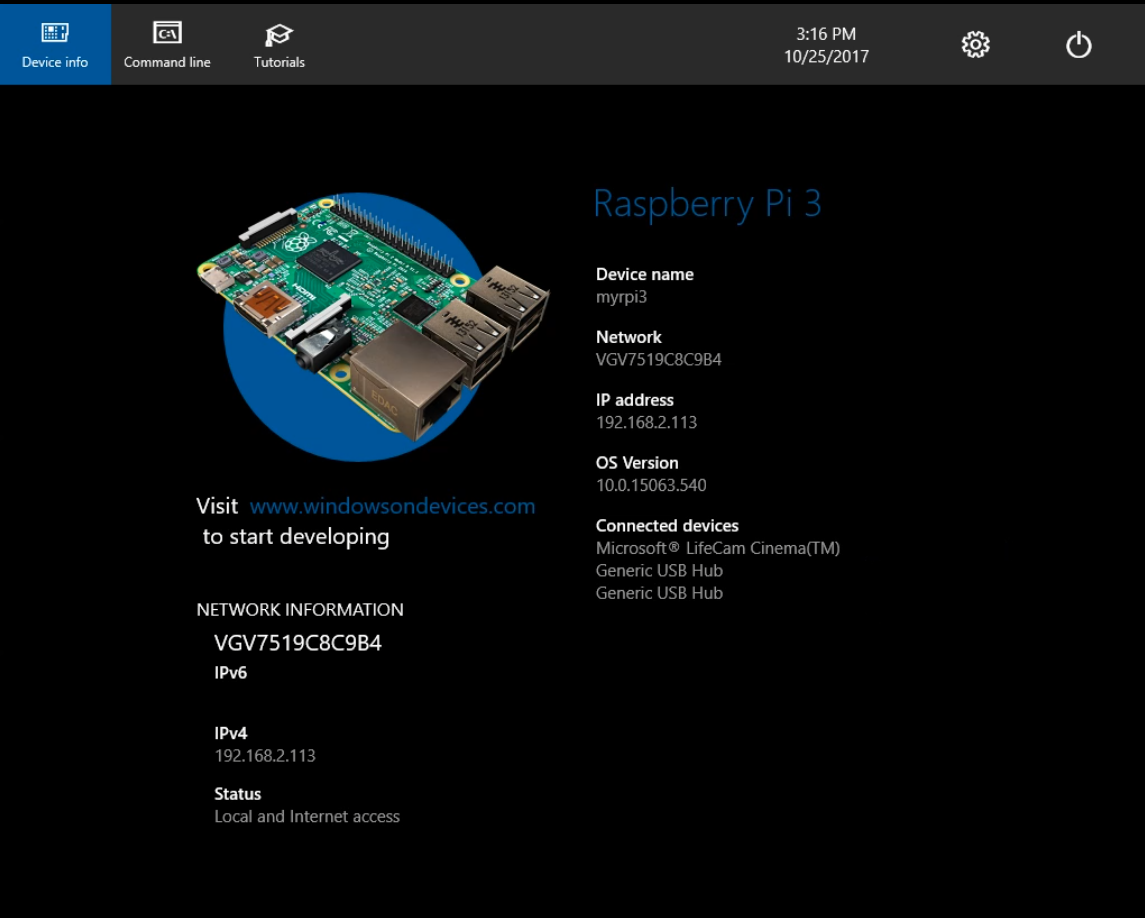


Figure - Windows 10 IoT Core default application showing a camera as Connected device.

1. Alternatively, you can also open the [Device Portal](https://docs.microsoft.com/en-us/windows/iot-core/manage-your-device/DevicePortal) in a browser and look for connected devices

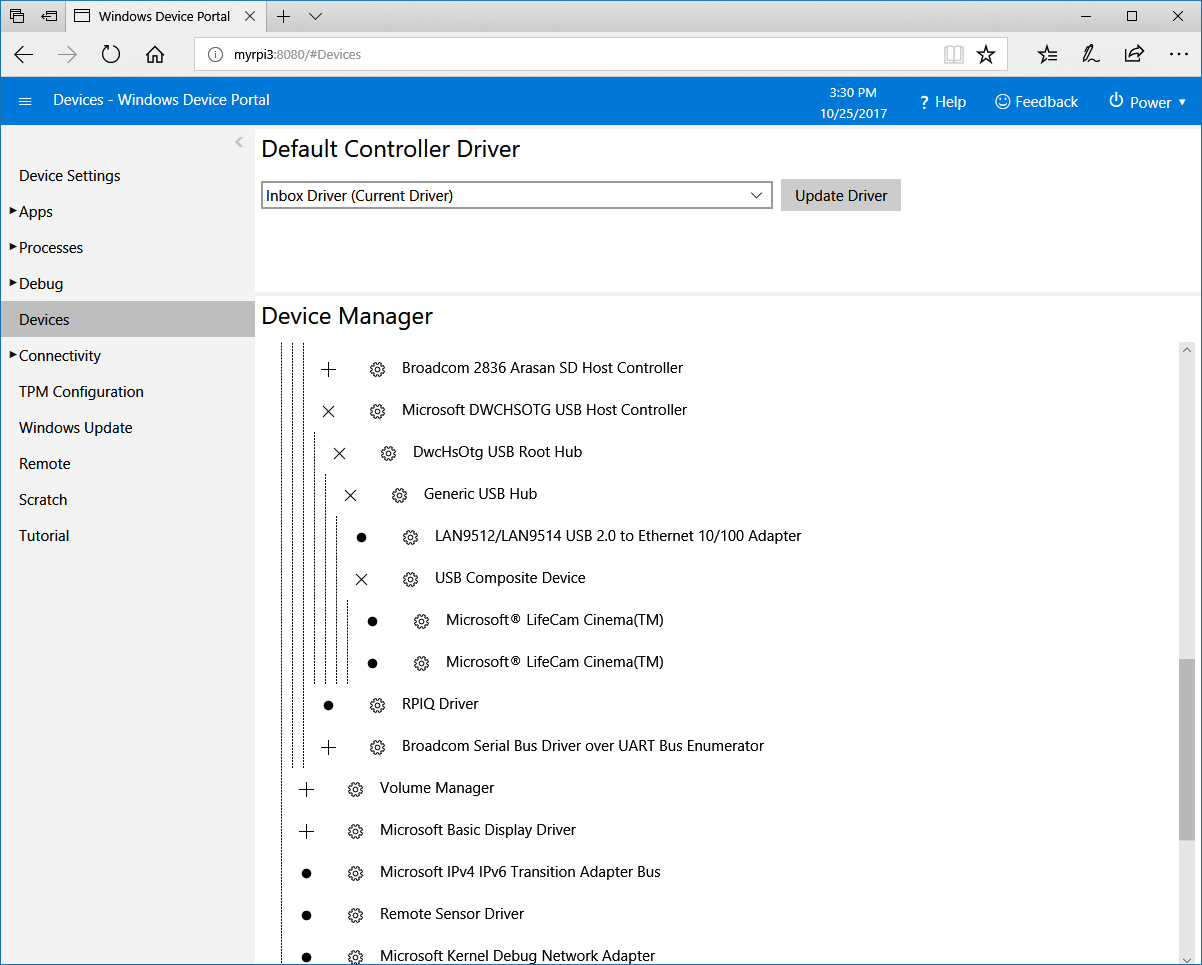


Figure - Camera showing up in the Device tab of the Device Portal

## Step 2 – Adding a new class in your project to control the camera

In this section we will add code to control the camera from inside a Universal Windows Platform application. In today’s lab we will make use of an existing solution that already contains a user interface, defined in XAML and several empty event handlers.

1. Inside Visual Studio 2017, open the solution **CognitiveServicesLab** that you have downloaded as prerequisite for this lab.
2. Right click on the project **CognitiveServicesLab** and select **Add Class** in the popup menu to create a new empty class.

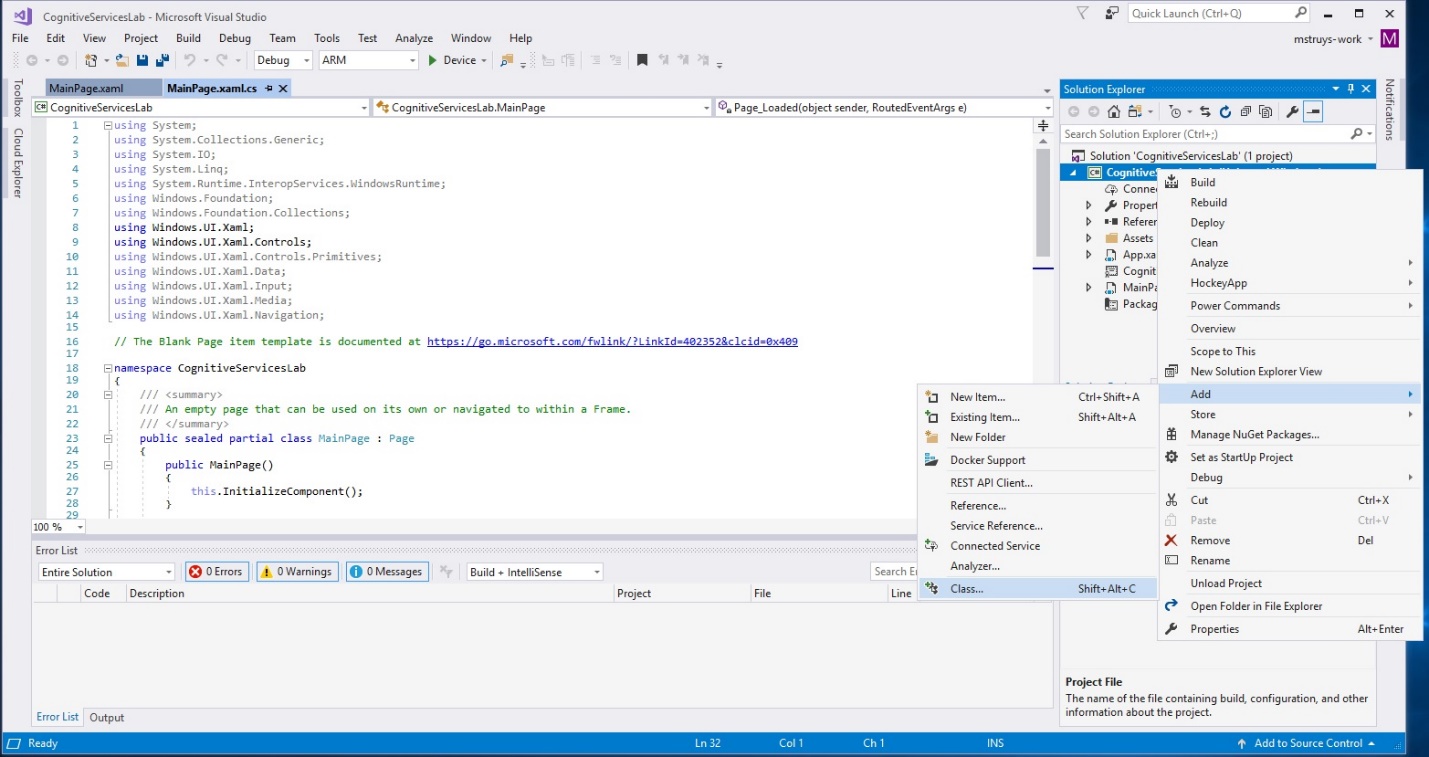


Figure - Adding a new Camera class to the solution

1. Give the class the name **Camera.cs** and click on **Add**

The empty class called Camera.cs will now be added to your project.

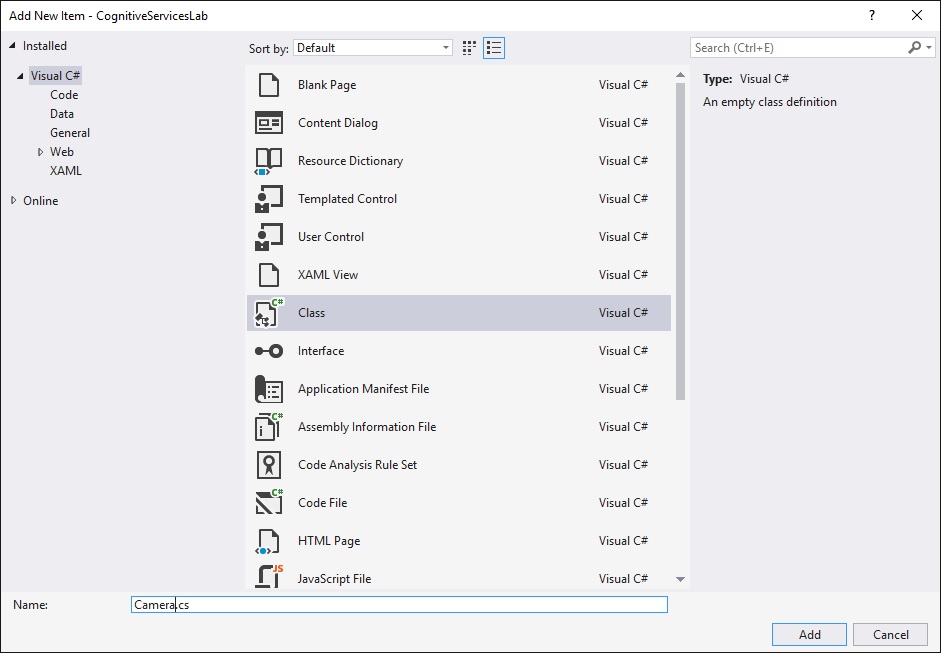


Figure - Add the Camera class to your project

1. Make your newly created class public



## Step 3 – Adding code to initialize the camera

All of the camera capture methods used in the Lab require the first step of initializing a [**MediaCapture**](https://msdn.microsoft.com/library/windows/apps/Windows.Media.Capture.MediaCapture) object by calling the constructor and then calling [**InitializeAsync**](https://msdn.microsoft.com/library/windows/apps/Windows.Media.Capture.MediaCapture.InitializeAsync). Since the **MediaCapture** object will be accessed from multiple places in your app, declare a property to hold the object.

Inside your newly created Camera class, you are now going to add some code to initialize the camera.

1. Add a public property of type **MediaCapture** to the Camera class and initialize it in the **Camera** constructor to make sure we have an object of type **MediaCapture** that will be used to control the camera.



You might need to add a using directive to **Windows.Media.Capture** to be able to use the **MediaCapture** type.

1. Add the following method inside the Camera class to find the USB camera, to initialize the MediaCapture object and to associate it to the camera.



You might need to add a using directive to **Windows.Devices.Enumeration** to be able to use the **DeviceInformation** type.

NOTE: For production code, you also need to make sure to dispose of the MediaCapture object after using it. To limit the amount of code you have to write in this Lab, we omit that.

## Step 3 – Add capability declarations to the app manifest

For your app to access a device's camera, you must declare that your app uses the *webcam* and *microphone* device capabilities.

**To add capabilities to the app manifest**

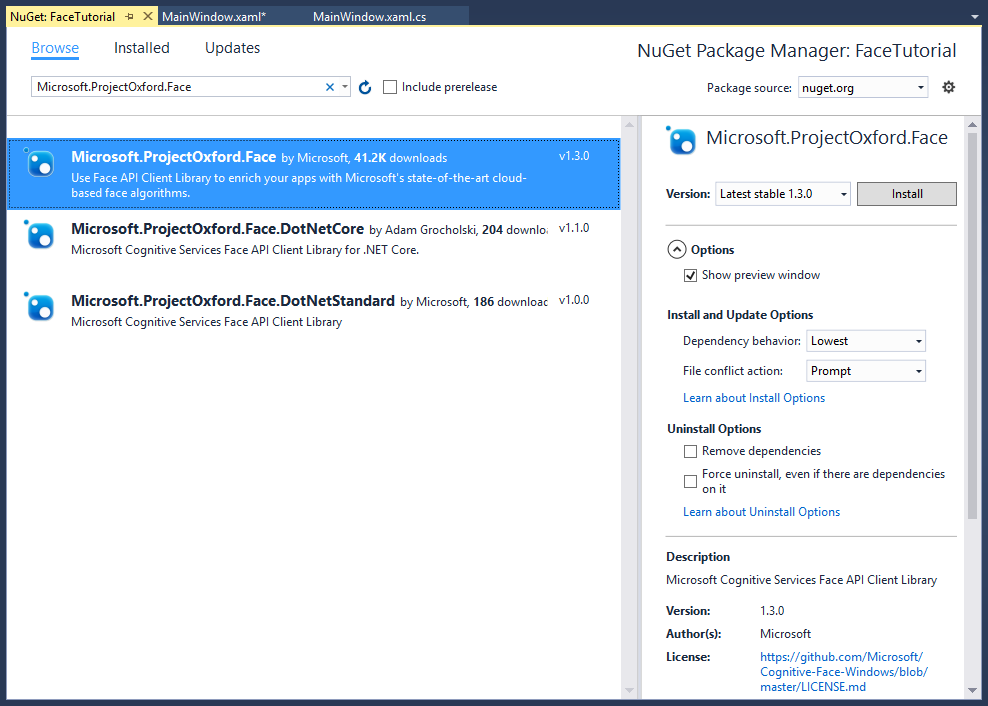
1. In Microsoft Visual Studio, in **Solution Explorer**, open the designer for the application manifest by double-clicking the **package.appxmanifest** item.
2. Select the **Capabilities** tab.
3. Check the box for **Webcam** and the box for **Microphone**.
4. To make sure that your newly added code does not contain syntax errors, it is a good idea to compile the solution at this moment. Do so by selecting **Build – Build Solution** from the Visual Studio 2017 menu.

## Step 4 – Configure the Face API Client Library

Face API is a cloud API that you can invoke through HTTPS REST requests. For ease-of-use in .NET applications, a .NET client library encapsulates the Face API REST requests. In this example, we use the client library to simplify our work.

Follow these instructions to configure the client library:

1. In the **Solution Explorer**, right-click your project and then click **Manage NuGet Packages**.
2. In the **NuGet Package Manager** window, select **nuget.org** as your Package source.
3. Search for **Microsoft.ProjectOxford.Face**, then **Install**. (In Visual Studio 2017, first click the **Browse** tab, then **Search**).



## Step 5 – Adding instance variables to the MainPage class

Now you will add code to the MainPage.cs source file. We are going to use several instance variables to store information that we need throughout the application’s life time. We will also make use of the *Microsoft Face API Windows SDK* to be able to use the Microsoft Face API, a cloud-based API that provides the most advanced algorithms for face detection and recognition.

1. Add the **Microsoft.ProjectOxford.Face** NuGet package to your project
2. In MainPage.xaml.cs, add the following declarations inside the class, just before the constructor:



You might need to add using directives to **Windows.UI** and to **Microsoft.ProjectOxford.Face**.

## Step 6 – Subscribe to the Face API and get your subscription key

Before using the Face API, you must sign up to subscribe to Face API in the Microsoft Cognitive Services portal. See [subscriptions](https://azure.microsoft.com/en-us/try/cognitive-services/). Either the primary or secondary subscription key can be used in this tutorial.

1. Add a new class called **CoreConstants.cs** to your project and make this class public
2. Add the following declarations to the class:



Replace **<your subscription key>** with the primary or secondary subscription key that just obtained.

If you sign-up for the Microsoft Face API with your Microsoft Azure account through the <https://azure.microsoft.com/> website, you specify the region for your trial from the preceding list of regions.

For example, if you sign up for Computer Vision with your Microsoft Azure account and you specify westus for your region, you must use the westus region for your REST API calls (https://westus.api.cognitive.microsoft.com/vision/v1.0/).

Make sure to have the right region in CoreConstants.cs.

## Step 7 – Initializing the Microsoft Face API and cleaning up resources from previous runs

In this step we delete possible person groups from our Face API subscription and create a new Person Group and a new Person to be used for our access control application.

1. Find the **Page\_Loaded** method inside **MainPage.xaml.cs** and mark the method as **private async void**
2. Add the following code to the **Page\_Loaded** method



You might need to add using directives to **System.Diagnostics** to be able to use the Debug class.

## Step 8 – Initializing the Camera and enable UI buttons to take a picture

In this step we will initialize the camera, start a preview window in which we display video and enable UI elements to take a first reference picture that we will use for image recognition later in this Lab.

1. Find the **InitVideo\_Click** method inside **MainPage.xaml.cs** and mark the method as **private async void**
2. Add the following code to **InitVideo\_Click** method



## Step 9 – Stop previewing video

In this step we stop the video camera.

1. Find the **ExitVideo\_Click** method inside **MainPage.xaml.cs** and add the following code to it



## Step 10 – Method to take a picture and store it in a memory stream for later use

In this step we will add a method to **MainPage.xaml.cs** to take a picture. The picture will be used as reference picture for our access controller. The picture will be stored in a memory stream but will not be stored in a file.

1. Add the following method to **MainPage.xaml.cs**



You might need to add using directives to **System.Threading.Task**, **Windows.Storage.Streams**, **Windows.Media.MediaProperties** and **Windows.Xaml.Media.Imaging**.

## Step 11 – Method to detect an image in a picture and store it for a Person in the PersonGroup

When using the client library, face detection is executed through the DetectAsync method for the FaceServiceClient class. This functionality is also available in the AddPersonFaceAsync method, that we will be using here. This method detects one or more persons in a stream and adds the image to the specified person. Notice that if the image contains more than one face, only the largest face will be added. Each face added to the person will be given a unique persisted face ID.

When no face is detected in an image, passed to the AddPersonFaceAsync method, it will throw an exception with error code “InvalidImage”.

1. Add the following method to **MainPage.xaml.cs**



## Step 12 – Add code to the UI to take a picture, detect a face and persist the face

We how have all code available to take a picture and to detect a face in it. If a face is detected, it will be added to the list of faces of a person. Now we will create some code to access this functionality from the user interface of the application. We will allow to take two different pictures. This is the minimum to be able to find the identity of a person through the Face API cognitive service.

1. Find the **TakeFirstPicture\_Click** method inside **MainPage.xaml.cs** and mark the method as **private async void**
2. Add the following code to **TakeFirstPicture\_Click**



1. Find the **TakeSecondPicture\_Click** method inside **MainPage.xaml.cs** and mark the method as **private async void**
2. Add the following code to **TakeSecondPicture\_Click**



## Step 13 – Add code to train the person group with the faces recognized in the images just taken

The person group must be trained before an identification can be performed using it. Moreover, it must be re-trained after adding or removing any person, or if any person has their registered face edited. The training is done by the [**Person Group – Train Person Group**](https://westus.dev.cognitive.microsoft.com/docs/services/563879b61984550e40cbbe8d/operations/563879b61984550f30395249) API. When using the client library, it is simply a call to the **TrainPersonGroupAsync** method.

Please note that the training is an asynchronous process. It may not be finished even after the the TrainPersonGroupAsync method returned. You may need to query the training status by [**Person Group - Get Person Group Training Status**](https://westus.dev.cognitive.microsoft.com/docs/services/563879b61984550e40cbbe8d/operations/563879b61984550f30395247) API or **GetPersonGroupTrainingStatusAsync** method of the client library. The following code demonstrates a simple logic of waiting person group training to finish.

1. Find the **TrainModel\_Click** method inside **MainPage.xaml.cs** and mark the method as **private async void**
2. Add the following code to **TrainModel\_Click**



You might need to add a using directive to **Microsoft.ProjectOxford.Face.Contract** to be able to use the TrainingStatus class.

## Step 14 – Add code to identify a face and to indicate that this person is granted access

Now you are going to add some code to recognize a person against the trained model. The code will also give a visual indication when a person is recognized (green rectangle) or not recognized (red rectangle).

When performing identify, the Face API can compute the similarity of a test face among all the faces within a group, and returns the most comparable person(s) for that testing face. This is done through the [**Face - Identify**](https://westus.dev.cognitive.microsoft.com/docs/services/563879b61984550e40cbbe8d/operations/563879b61984550f30395239) API or the IdentifyAsync method of the client library.

The testing face needs to be detected and passed to the Identity API. By default, the identify returns only one person which matches the test face best.

1. Add the following method to **MainPage.xaml.cs**



## Step 15 – Add code to the UI to identify a face

We how have all code available to identify a face and to check if it belongs to a person. Now we will create some code to access this functionality from the user interface of the application. We will allow to take one pictures that will be compared against the trained model.

1. Find the **AllowAccess\_Click** method inside **MainPage.xaml.cs** and mark the method as **private async void**
2. Add the following code to **AllowAccess\_Click**



## Step 16 – Test your application on a Raspberry Pi

Finally we will compile the application, deploy it to a Raspberry Pi that has a USB camera connected and start the application using the Visual Studio remote debugger.

1. Make sure that compile your application for ARM.
2. Compile the project and fix possible syntax errors. Note, if you see a warning about conflicts between different versions of the same dependent assembly, you can ignore that for now. This has to do with the fact that the ProjectOxford NuGet package refers to an older version of the NewtonSoft.Json library internally.
3. Specify a remote machine on which to deploy the application using the ip address or the name of your Raspberry Pi 2. To do this, go to **Debug** – **Project Properties** from the Visual Studio menu or right click the project file and select **Properties** in the popup menu.
4. In the Debug Tab, select **Remote Machine** as **Target device** and click on **Find** to specify the connection details of the target device.

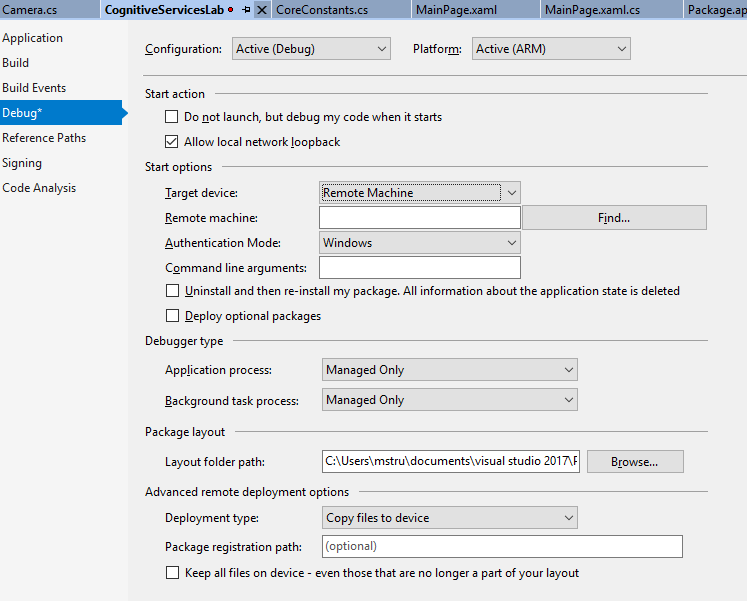


Figure - Setting up remote debugging

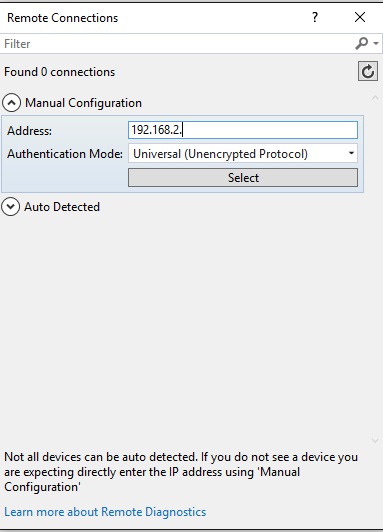


Figure - Setting the IP address for the Raspberry Pi

1. After entering a device name or the device IP address, click on **Select**.
2. Start debugging by pressing F5 or by clicking the run button on the Visual Studio menu
3. After a while your application will automatically start on the Raspberry Pi
4. Click on Initialize Video to start previewing through the connected video camera
5. Take two different pictures (for instance of yourself)
6. In the detection part of the UI, click the Train Model button
7. Finally take another picture using the Capture Image button and see if you get access which is simulated by the green rectangle
8. Take a few pictures of other people to see that they don’t get access (because their faces are not recognized).

