



Lab 8 Azure Cognitive Services

By: Maarten Struys

IoT Solution Architect

December, 2017

If you have any issues or concerns, please email: virtualbootcamphelp@microsoft.com.

Execution Time: 60 minutes.

Required Hardware:

- Windows 10 PC
- IoT Hardware kit: <https://www.adafruit.com/product/3605> or similar hardware.
- Access to a WiFi network (without a captive portal aka web page login)

Required Operating System:

- Windows 10

Other Requirements:

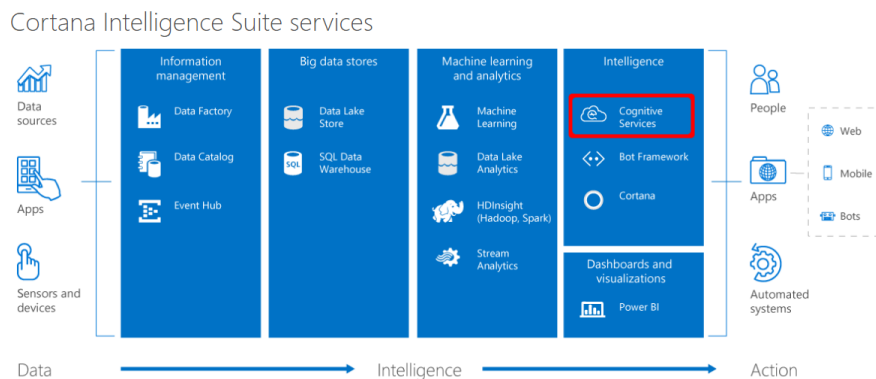
- Azure Subscription

Required Software:

Software	Size	Installation URL
Visual Studio 2017	150 MB	https://aka.ms/vs/15/release/vs_community.exe

Introduction

This lab explores image recognition using Azure 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, you 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 for your application here: [CognitiveServicesLab-InitialSolution](#). This project contains all the XAML for the UI and several empty event handlers to act on button clicks. You will add code to control the camera and to call the Face API cognitive service to create an access controller system.

All source code that you are going to need is provided in the appendices of this document to save you typing. If you find it easier to have all source code needed in separate text files, you can download those here:

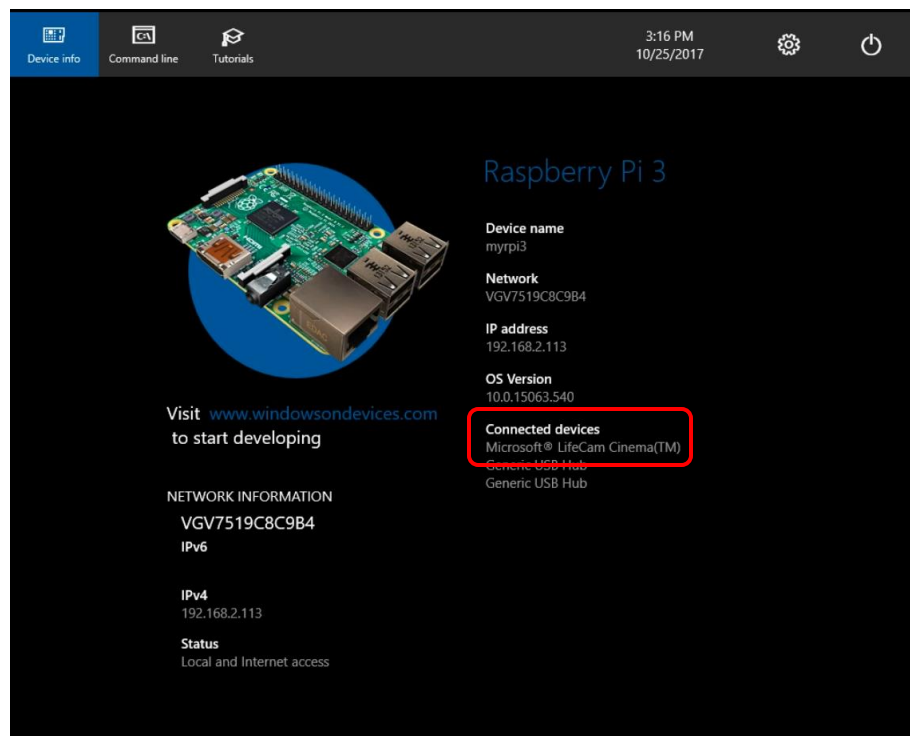
<https://1drv.ms/f/s!AmHxRmFJqwIzpgZszKlyjSPCCJ6oig>.

Controlling a USB camera on the Raspberry Pi

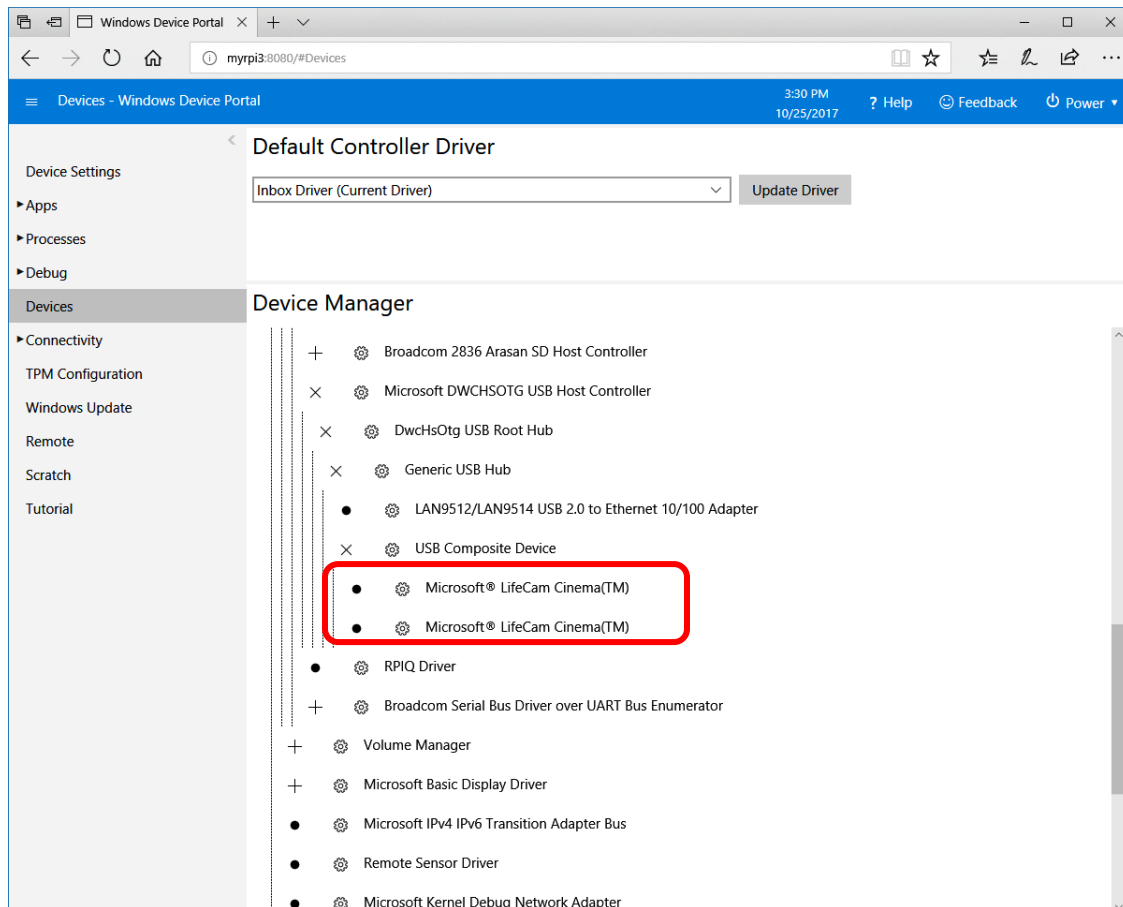
For this lab, you will not write a complete video application, but provide just enough code to initialize the camera, to display a video stream and to take individual pictures from the video stream. Because Windows 10 IoT Core has no GPU support on Raspberry Pi, the video quality is not optimal but good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

1.1 Verify if the camera connects to the Raspberry Pi

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



Step 3. Alternatively, you can also open the [Device Portal](#) in a browser and look for connected devices:

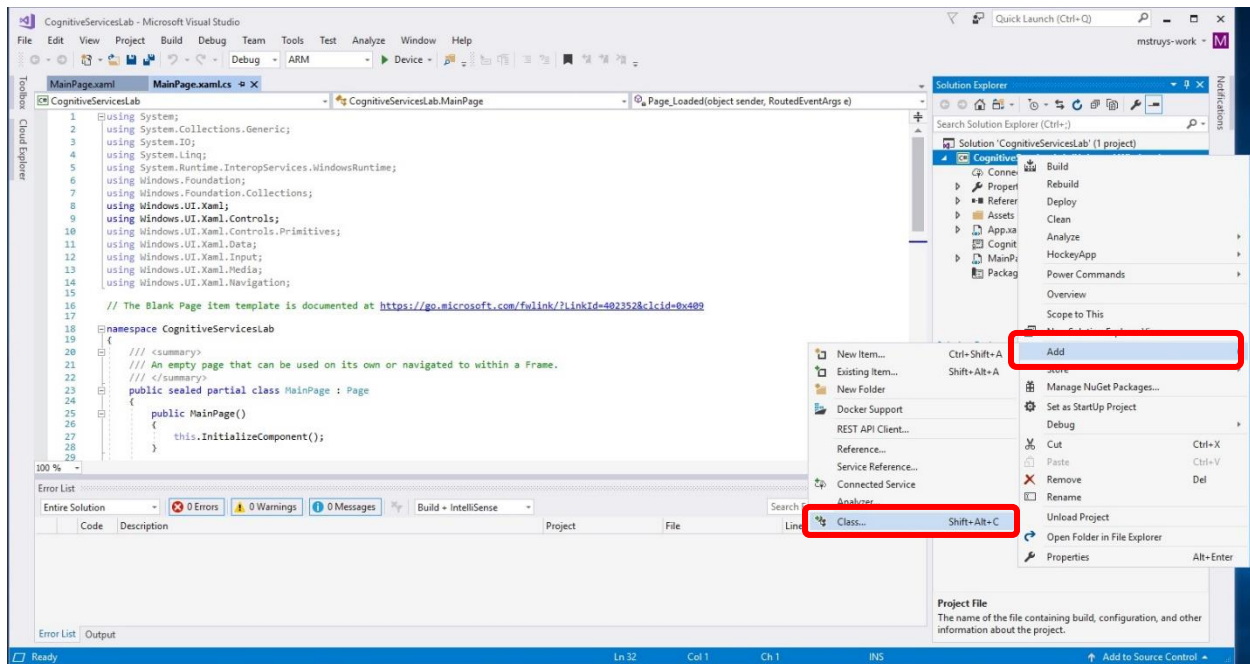


1.2 Adding a new class in your project to control the camera

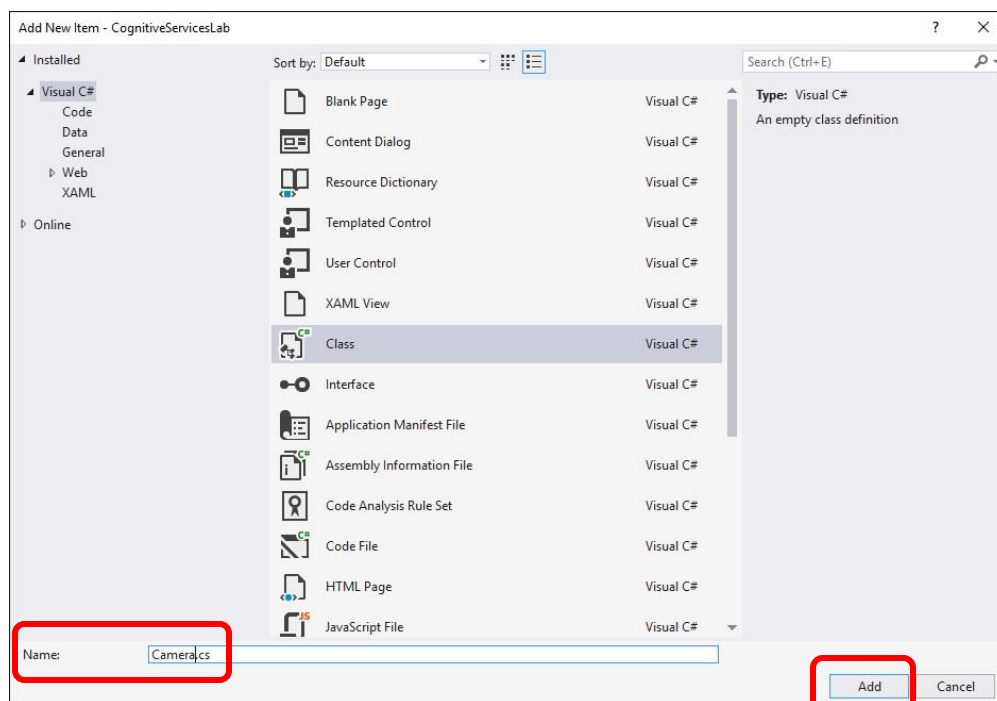
In this section we will add code to control the camera from inside a Universal Windows Platform application. In today's lab we will make use of an existing solution that already contains a user interface, defined in XAML and several empty event handlers.

Step 4. Inside Visual Studio 2017, open the solution **CognitiveServicesLab** that you have downloaded as prerequisite for this lab.

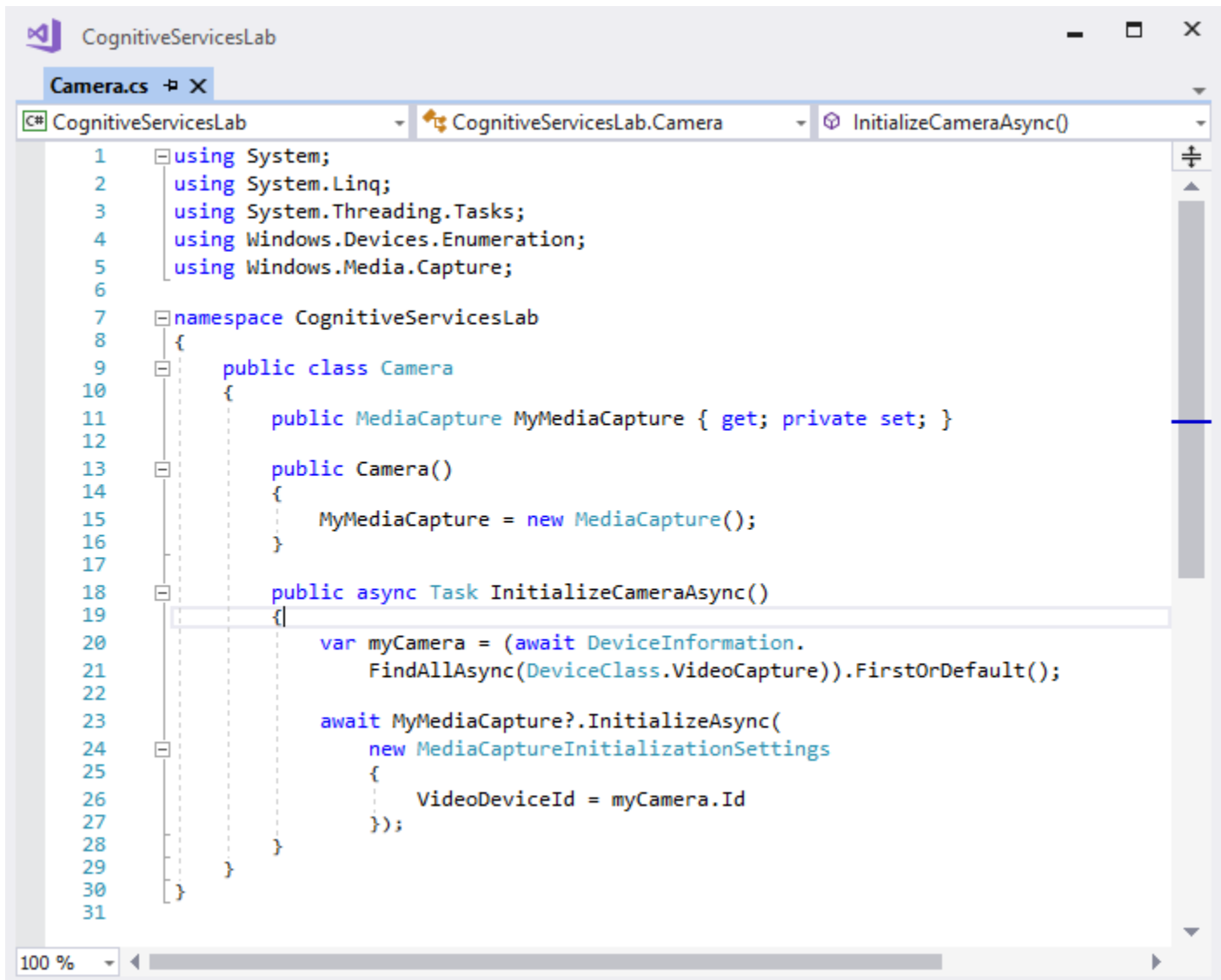
- Step 5. Right click on the project **CognitiveServicesLab** and select **Add Class** in the popup menu to create a new empty class.



- Step 6. Give the class the name **Camera.cs** and click on **Add**.



- Step 7. Replace the code for the newly created camera class with the following code. To save time, you can copy and paste the code for this class from [Appendix 1](#) of this document. Just double click on the source code in Appendix 1, copy all the contents and replace the contents of Camera.cs with the copied source code.



```
1  using System;
2  using System.Linq;
3  using System.Threading.Tasks;
4  using Windows.Devices.Enumeration;
5  using Windows.Media.Capture;
6
7  namespace CognitiveServicesLab
8  {
9      public class Camera
10     {
11         public MediaCapture MyMediaCapture { get; private set; }
12
13         public Camera()
14         {
15             MyMediaCapture = new MediaCapture();
16         }
17
18         public async Task InitializeCameraAsync()
19         {
20             var myCamera = (await DeviceInformation.
21                 FindAllAsync(DeviceClass.VideoCapture)).FirstOrDefault();
22
23             await MyMediaCapture?.InitializeAsync(
24                 new MediaCaptureInitializationSettings
25                 {
26                     VideoDeviceId = myCamera.Id
27                 });
28         }
29     }
30 }
31
```

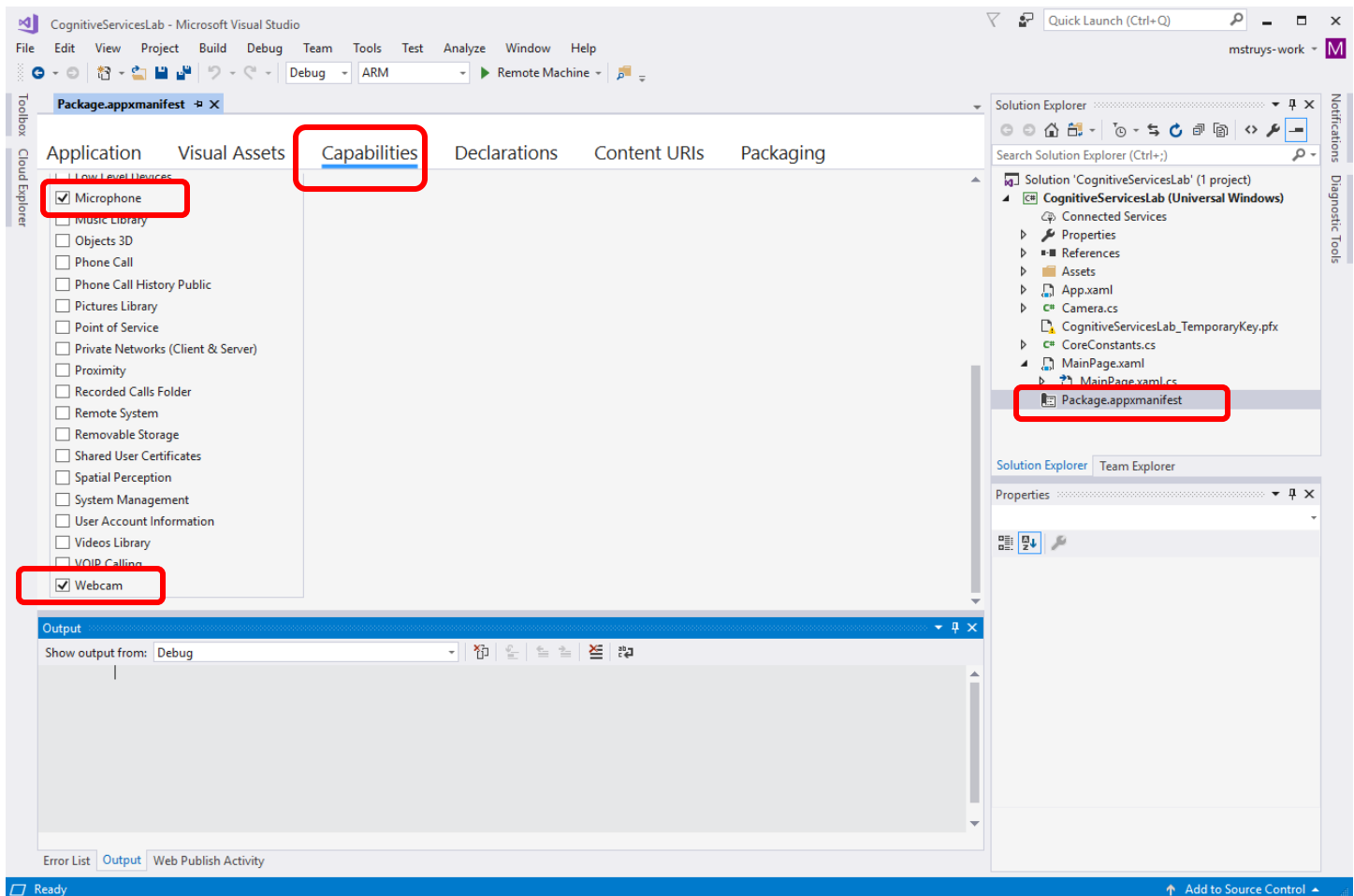
The code you just entered uses a MediaCapture object to capture video and photos from the connected webcam. To do so, we initialize a MediaCapture object with the first USB webcam found that is connected to the Raspberry Pi.

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 for this Lab, we omit that.

1.3 Add capability declarations to the app manifest

For your app to access the camera, it is not enough to initialize a MediaCapture object. You also need to declare that your app uses the webcam device capabilities. For that, you need to edit the app manifest.

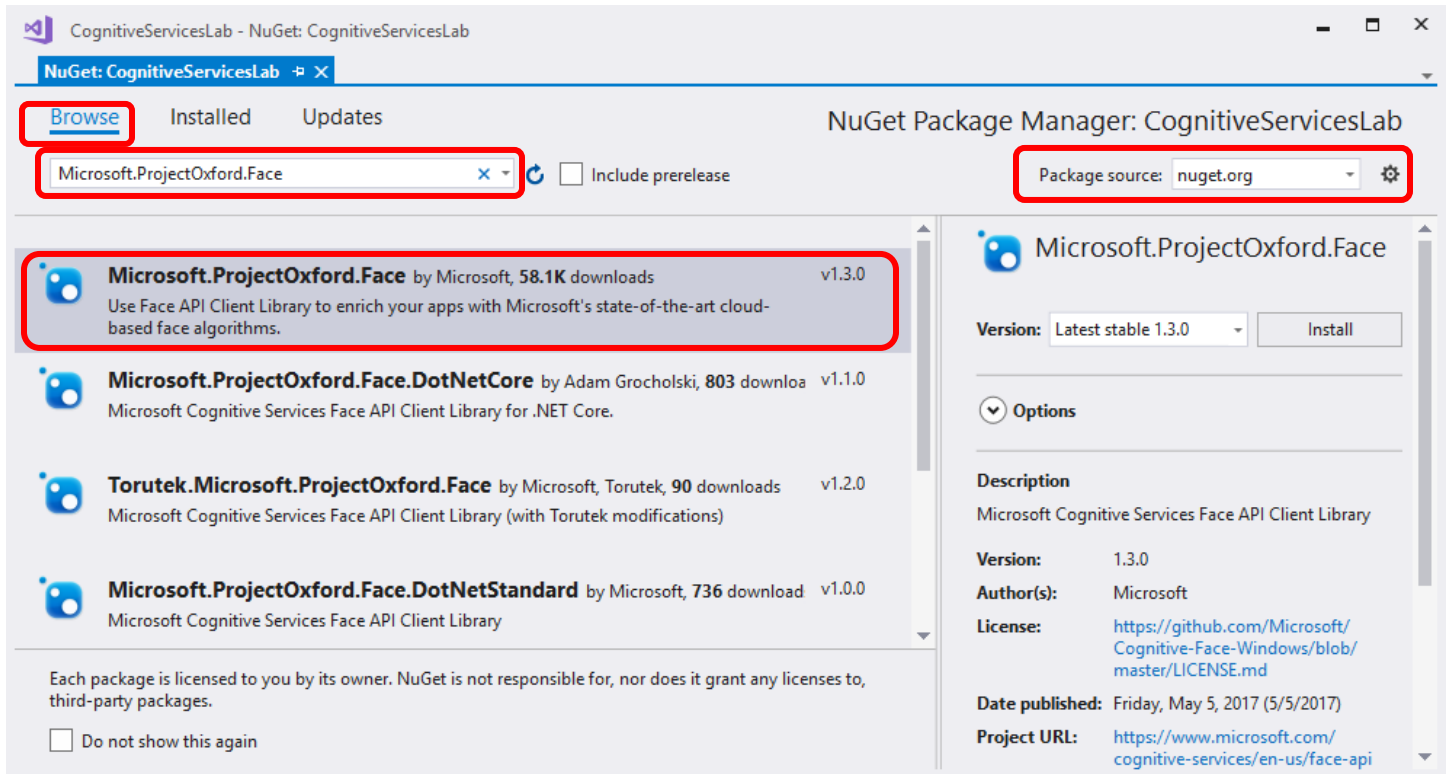
- Step 8. In Microsoft Visual Studio, in **Solution Explorer**, open the designer for the application manifest by double-clicking the **package.appxmanifest** item.
- Step 9. Select the **Capabilities** tab.
- Step 10. Check the box for **Webcam** and the **Microphone**
- Step 11. To make sure that your newly added code does not contain syntax errors, it is a good idea to compile the solution at this moment. You can do so by selecting **Build – Build Solution** from the Visual Studio 2017 Menu (or use the **Control – Shift – B** key combination).



Configuring 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. Execute the next steps to configure the client library:

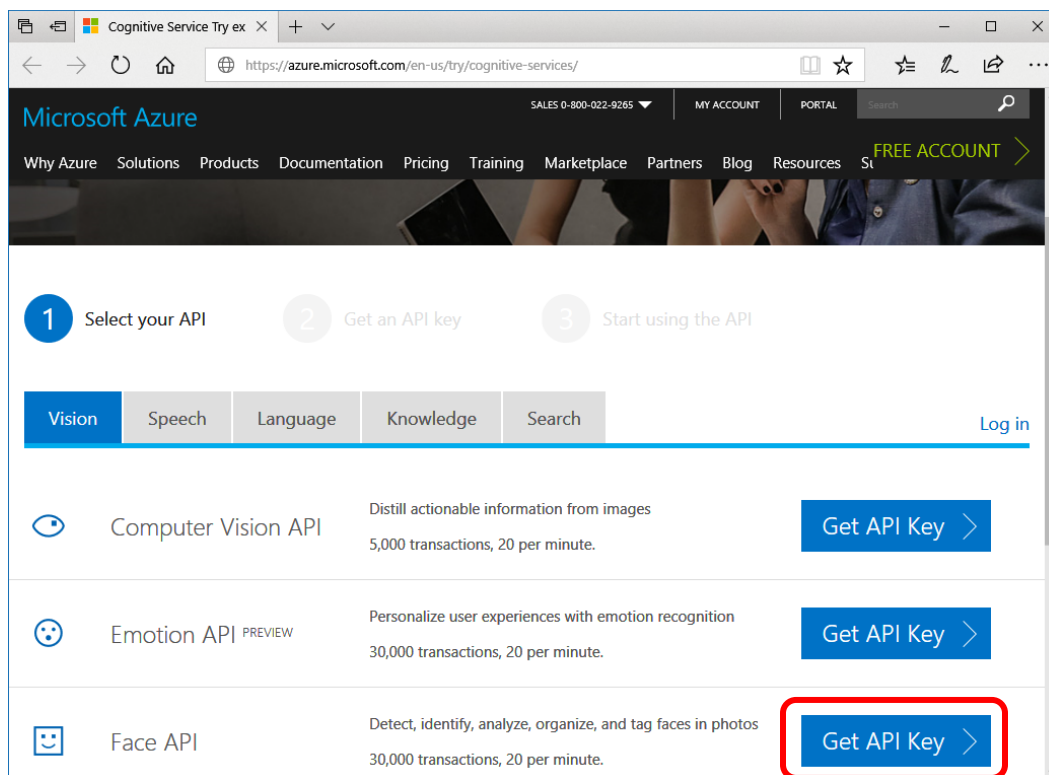
- Step 12. In Solution Explorer, right-click your project and then click **Manage NuGet Packages**.
- Step 13. In the **NuGet Package Manager** window, select **nuget.org** as your Package source.
- Step 14. Search for **Microsoft.ProjectOxford.Face** and **Install** (in Visual Studio 2017, first click the **Browse** tab, then **Search**).



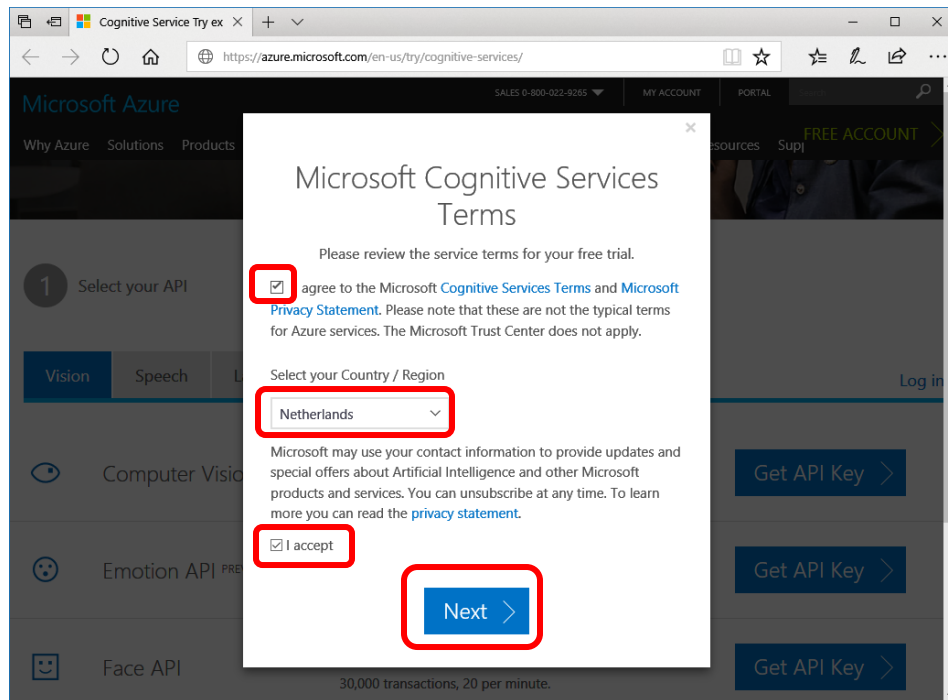
Subscribe to the Face API and get your subscription key

Before you can use the Face API, you must sign up to use it.

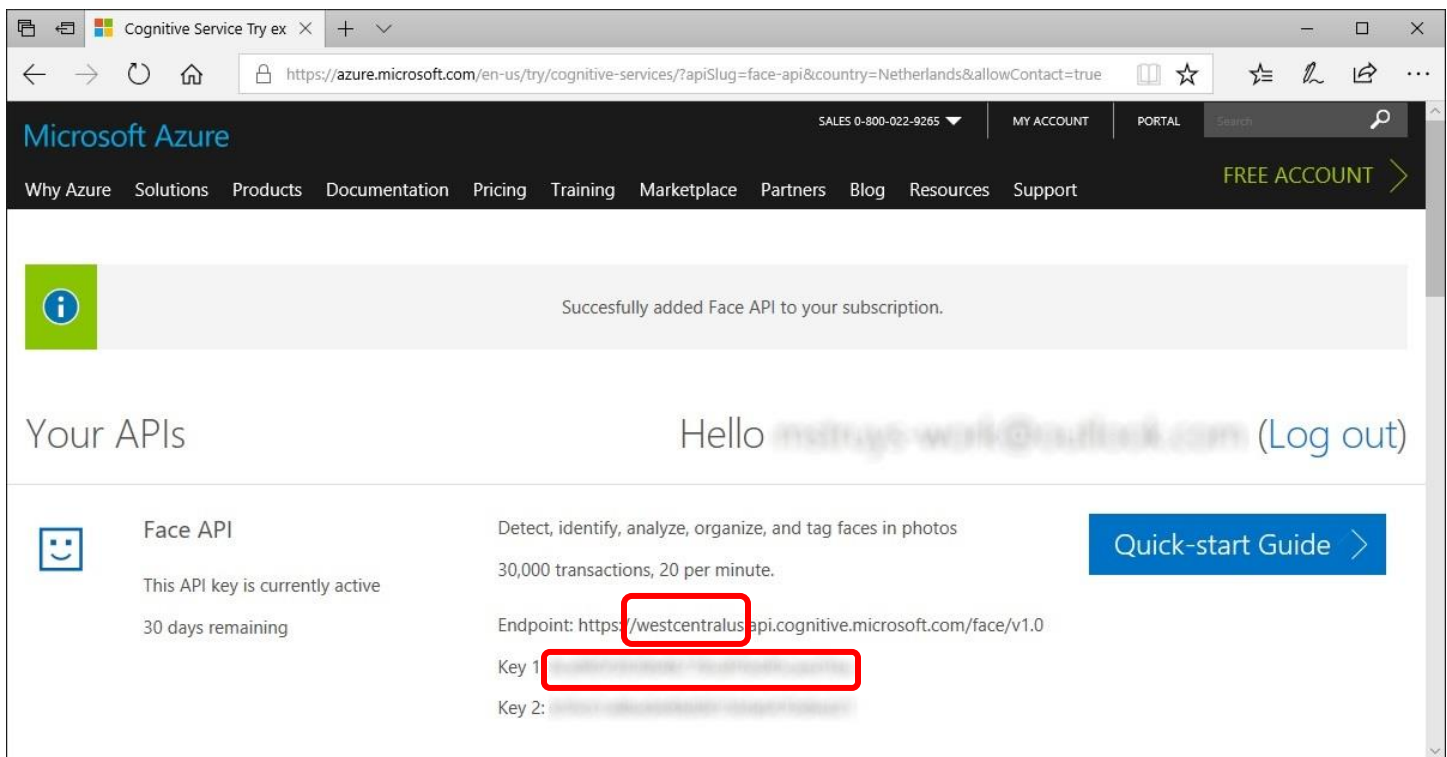
- Step 15. Browse to <https://azure.microsoft.com/en-us/try/cognitive-services/>
- Step 16. Click on the Get API Key button for the Face API



Step 17. Accept the service terms and select your country / region and click **Next**.



Step 18. Sign in with your MSA account to get your free 30 day trial period:



Step 19. Record the region that is part of the Endpoint and the value of Key 1. Tip: Store them temporary in Notepad for later use.

Step 20. Inside Visual Studio, right click on the project **CognitiveServicesLab** and select **Add Class** in the popup menu to create a new empty class.

Step 21. Give the class the name **CoreConstants.cs** and click on **Add**.

- Step 22. Replace the code for the newly created CoreConstants class with the following code. To save time, you can copy and paste the code for this class from [Appendix 2](#) of this document. Just double click on the source code in Appendix 1, copy all the contents and replace the contents of CoreConstants.cs with the copied source code.



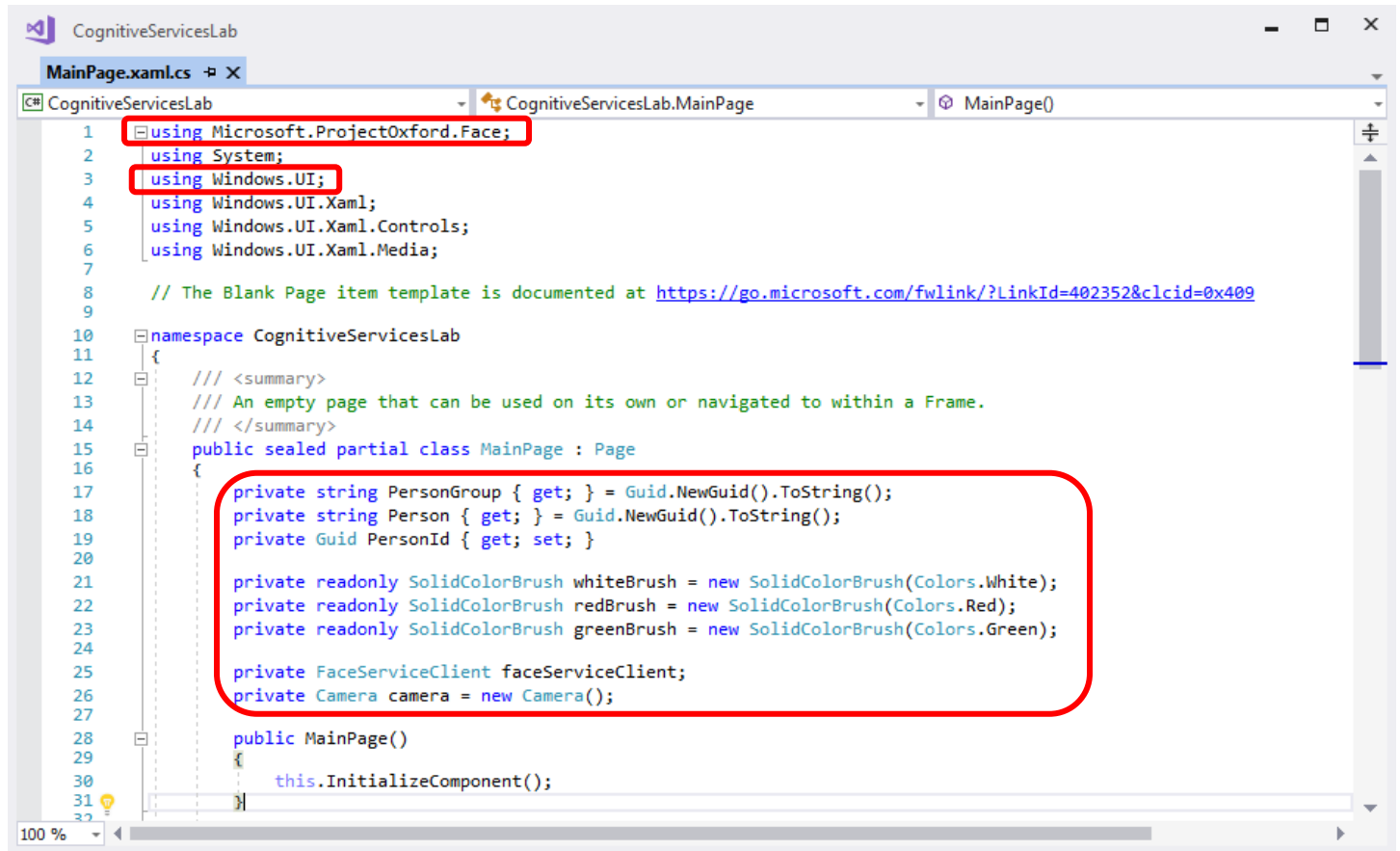
```
1 namespace CognitiveServicesLab
2 {
3     public class CoreConstants
4     {
5         private static string CognitiveServicesRegion = "<your region>";
6         public static string CognitiveServicesBaseUrl =
7             $"https://{CognitiveServicesRegion}.api.cognitive.microsoft.com/face/v1.0";
8         public static string FaceApiSubscriptionKey = "<your subscription key>";
9     }
10 }
11
```

- Step 23. On line 5, enter the region that you stored temporarily in Notepad (e.g. westcentralus).
- Step 24. On line 8, enter the value of Key 1 that you stored temporarily in Notepad.

Add variables to your application to store data needed

Now you will add code to the **MainPage.xaml.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.

Step 25. In MainPage.xaml.cs, add the following declarations (you will also add using directives to Windows.UI and to Microsoft.ProjectOxford.Face). To save time, you can copy and paste the variable declarations from [Appendix 3](#). Note that you still manually must add the using directives.

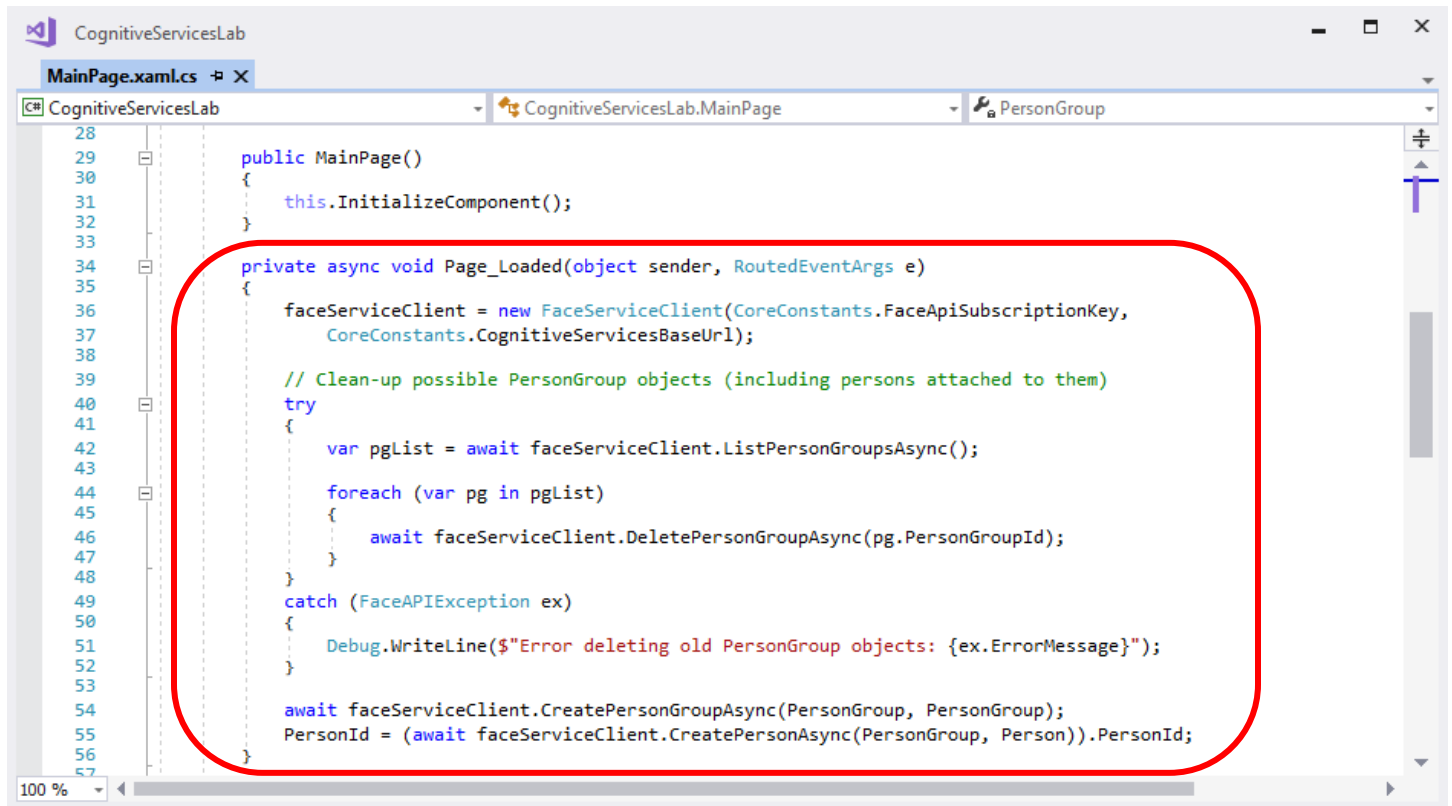


```
1 using Microsoft.ProjectOxford.Face;
2 using System;
3 using Windows.UI;
4 using Windows.UI.Xaml;
5 using Windows.UI.Xaml.Controls;
6 using Windows.UI.Xaml.Media;
7
8 // The Blank Page item template is documented at https://go.microsoft.com/fwlink/?LinkId=402352&clcid=0x409
9
10 namespace CognitiveServicesLab
11 {
12     /// <summary>
13     /// An empty page that can be used on its own or navigated to within a Frame.
14     /// </summary>
15     public sealed partial class MainPage : Page
16     {
17         private string PersonGroup { get; } = Guid.NewGuid().ToString();
18         private string Person { get; } = Guid.NewGuid().ToString();
19         private Guid PersonId { get; set; }
20
21         private readonly SolidColorBrush whiteBrush = new SolidColorBrush(Colors.White);
22         private readonly SolidColorBrush redBrush = new SolidColorBrush(Colors.Red);
23         private readonly SolidColorBrush greenBrush = new SolidColorBrush(Colors.Green);
24
25         private FaceServiceClient faceServiceClient;
26         private Camera camera = new Camera();
27
28         public MainPage()
29         {
30             this.InitializeComponent();
31         }
32     }
33 }
```

Initialize the Microsoft Face API and cleanup resources from previous runs

Now you will 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.

- Step 26. Find the **Page_Loaded** method inside **MainPage.xaml.cs** and replace it entirely with the following code. To save time, you can copy and paste the code from [Appendix 4](#). You also need to ask a using directive to **System.Diagnostics** at the top of file to be able to use the **Debug** class.

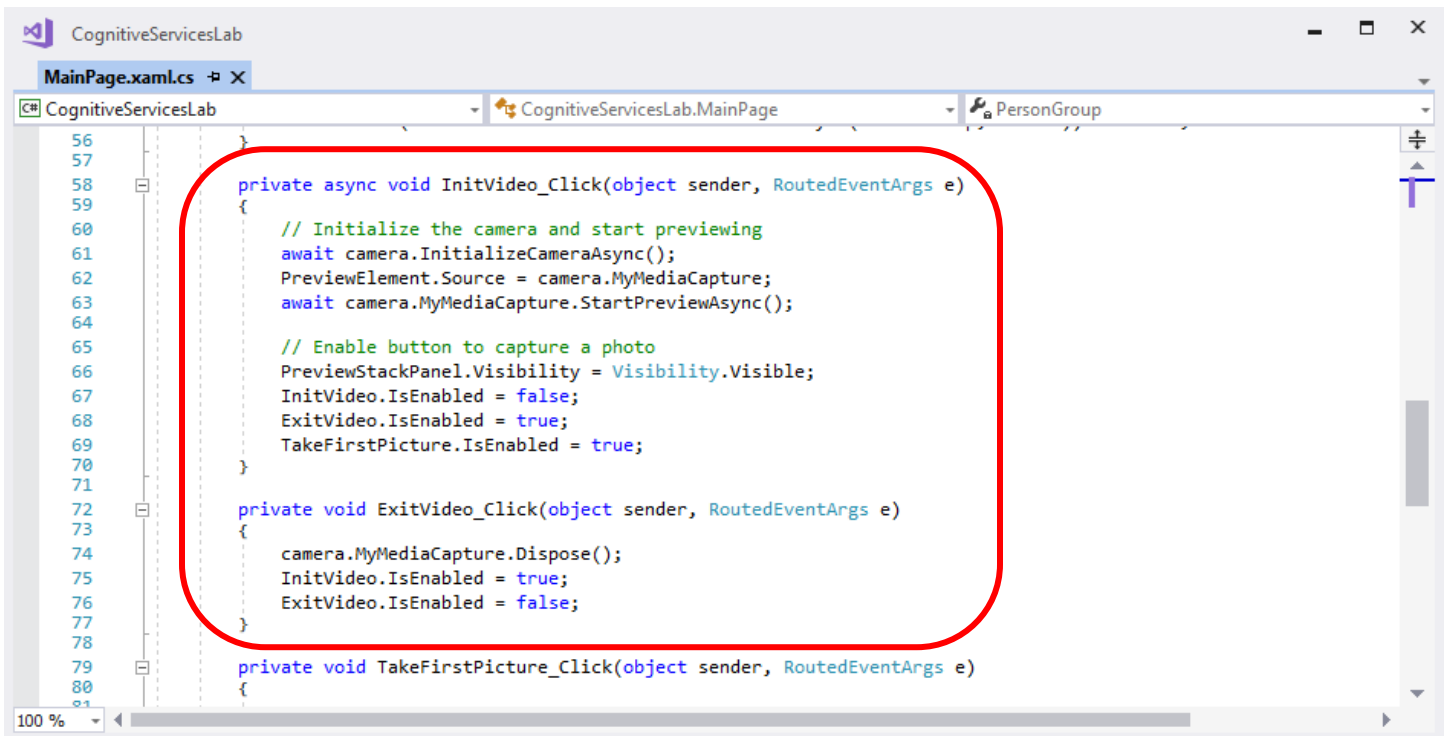


```
28
29 public MainPage()
30 {
31     this.InitializeComponent();
32 }
33
34 private async void Page_Loaded(object sender, RoutedEventArgs e)
35 {
36     faceServiceClient = new FaceServiceClient(CoreConstants.FaceApiSubscriptionKey,
37         CoreConstants.CognitiveServicesBaseUrl);
38
39     // Clean-up possible PersonGroup objects (including persons attached to them)
40     try
41     {
42         var pgList = await faceServiceClient.ListPersonGroupsAsync();
43
44         foreach (var pg in pgList)
45         {
46             await faceServiceClient.DeletePersonGroupAsync(pg.PersonGroupId);
47         }
48     }
49     catch (FaceAPIException ex)
50     {
51         Debug.WriteLine($"Error deleting old PersonGroup objects: {ex.ErrorMessage}");
52     }
53
54     await faceServiceClient.CreatePersonGroupAsync(PersonGroup, PersonGroup);
55     PersonId = (await faceServiceClient.CreatePersonAsync(PersonGroup, Person)).PersonId;
56 }
57
```

Initializing and Exiting Video

You will now add code to initialize the camera, start a preview window in which you display video and enable UI elements to take a first reference picture that you will use for image recognition later in this Lab. You will also write code to properly terminate the video camera. A big part of the following code also controls the User Interface.

Step 27. Find the **InitVideo_Click** method and the **ExitVideo_Click** method inside **MainPage.xaml.cs** and replace them with the following code. To save time, you can copy and paste the code from [Appendix 5](#).



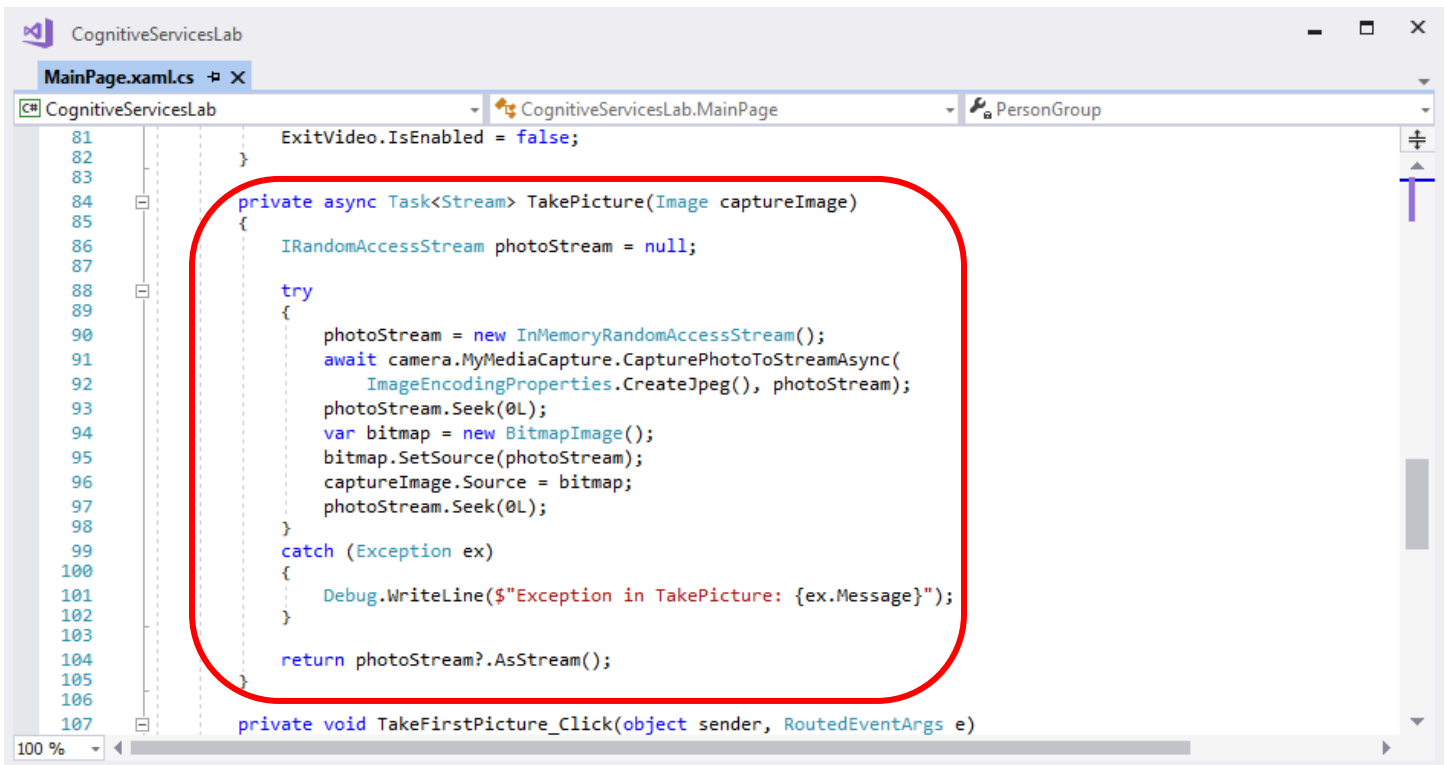
```
56 }
57
58 private async void InitVideo_Click(object sender, RoutedEventArgs e)
59 {
60     // Initialize the camera and start previewing
61     await camera.InitializeCameraAsync();
62     PreviewElement.Source = camera.MyMediaCapture;
63     await camera.MyMediaCapture.StartPreviewAsync();
64
65     // Enable button to capture a photo
66     PreviewStackPanel.Visibility = Visibility.Visible;
67     InitVideo.IsEnabled = false;
68     ExitVideo.IsEnabled = true;
69     TakeFirstPicture.IsEnabled = true;
70 }
71
72 private void ExitVideo_Click(object sender, RoutedEventArgs e)
73 {
74     camera.MyMediaCapture.Dispose();
75     InitVideo.IsEnabled = true;
76     ExitVideo.IsEnabled = false;
77 }
78
79 private void TakeFirstPicture_Click(object sender, RoutedEventArgs e)
80 {
81 }
```

Take a picture and store it in a memory stream

You will now add a method to **MainPage.xaml.cs** to take a picture. The picture will be used as a reference picture for our access controller. The picture will be stored in a memory stream, but it will not be preserved in a file.

Step 28. Add the following using directives to the beginning of the **MainPage.xaml.cs** source file:
System.Threading.Task, **Windows.Storage.Streams**, **Windows.Media.MediaProperties** and **Windows.Xaml.Media.Imaging**.

Step 29. Under the **ExitVideo_Click** method inside **MainPage.xaml.cs**, add a new method called **TakePicture** with the following code. To save time, you can copy and paste the code from [Appendix 6](#).

A screenshot of the Visual Studio IDE showing the `MainPage.xaml.cs` file. The `TakePicture` method is highlighted with a red rounded rectangle. The code is as follows:

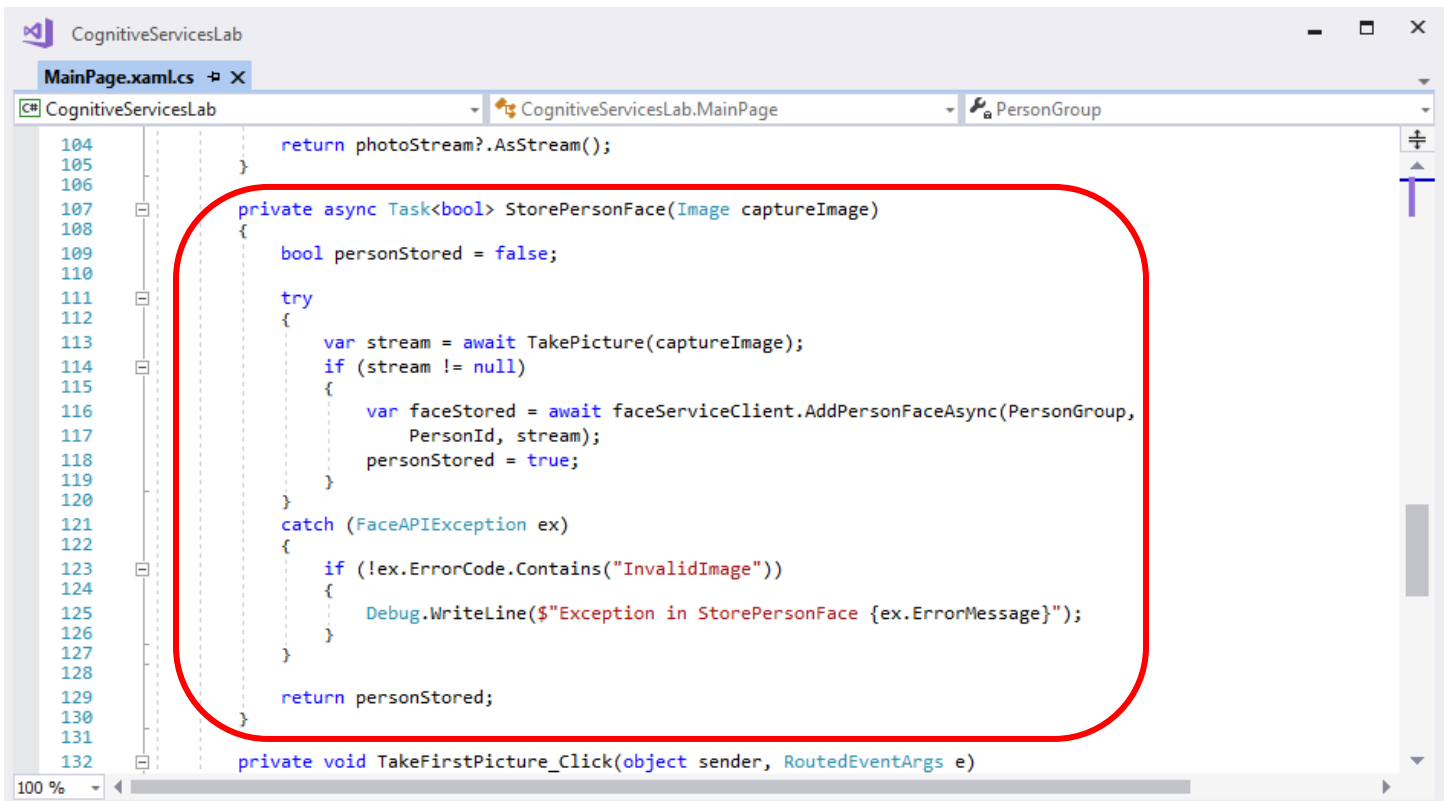
```
81     }
82     ExitVideo.IsEnabled = false;
83 }
84 private async Task<Stream> TakePicture(Image captureImage)
85 {
86     IRandomAccessStream photoStream = null;
87
88     try
89     {
90         photoStream = new InMemoryRandomAccessStream();
91         await camera.MyMediaCapture.CapturePhotoToStreamAsync(
92             ImageEncodingProperties.CreateJpeg(), photoStream);
93         photoStream.Seek(0L);
94         var bitmap = new BitmapImage();
95         bitmap.SetSource(photoStream);
96         captureImage.Source = bitmap;
97         photoStream.Seek(0L);
98     }
99     catch (Exception ex)
100     {
101         Debug.WriteLine($"Exception in TakePicture: {ex.Message}");
102     }
103
104     return photoStream?.AsStream();
105 }
106
107 private void TakeFirstPicture_Click(object sender, RoutedEventArgs e)
```

Detect a face in a picture and store it in a Person's PersonGroup

When you are 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 you 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".

Step 30. Under the **TakePicture** method inside **MainPage.xaml.cs**, add a new method called **StorePersonFace** with the following code. To save time, you can copy and paste the code from [Appendix 7](#).



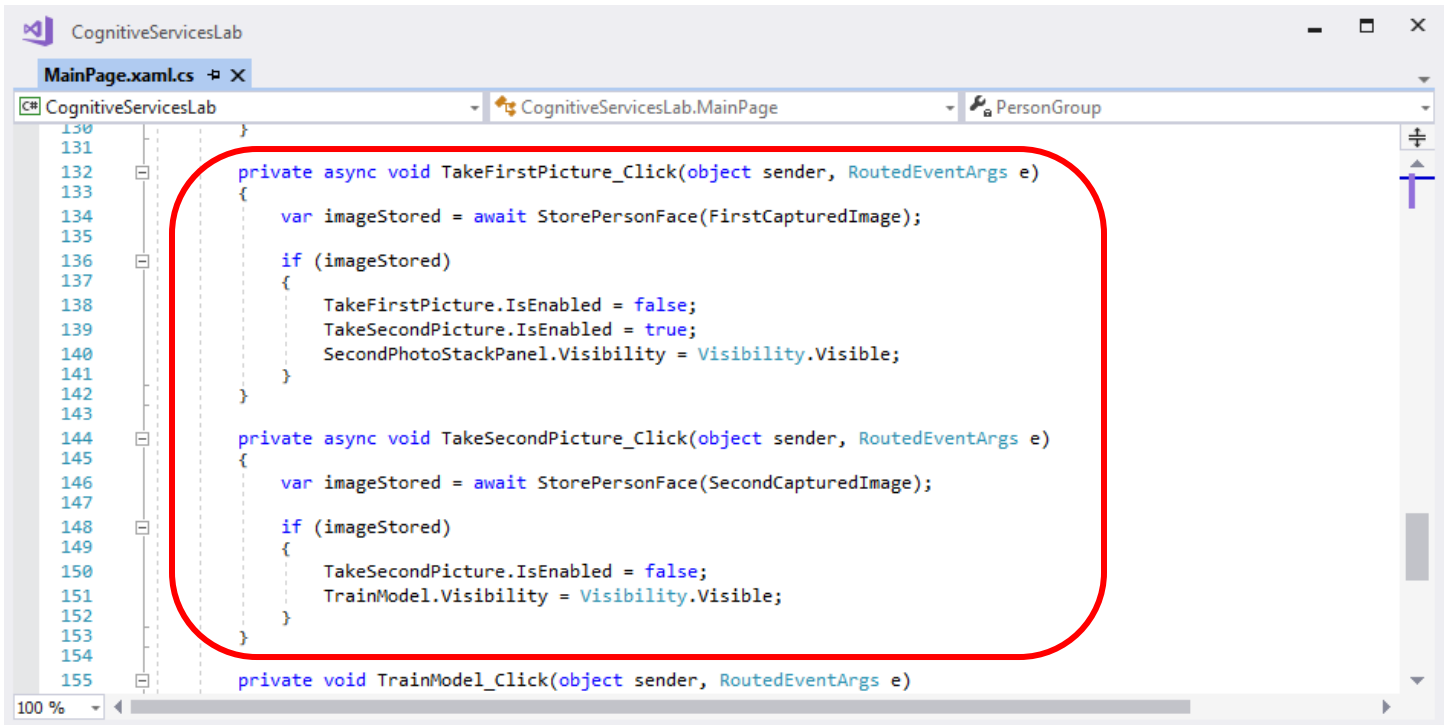
The screenshot shows the Visual Studio IDE with the file **MainPage.xaml.cs** open. The code editor displays the implementation of the **StorePersonFace** method, which is highlighted with a red rounded rectangle. The method is a private asynchronous function that takes an **Image** parameter and returns a **Task<bool>**. It attempts to take a picture using **TakePicture**, checks if the stream is not null, and then uses **faceServiceClient.AddPersonFaceAsync** to store the face. If an exception occurs and it contains the text "InvalidImage", a debug message is written. The method returns **personStored**.

```
104         return photoStream?.AsStream();
105     }
106
107     private async Task<bool> StorePersonFace(Image captureImage)
108     {
109         bool personStored = false;
110
111         try
112         {
113             var stream = await TakePicture(captureImage);
114             if (stream != null)
115             {
116                 var faceStored = await faceServiceClient.AddPersonFaceAsync(PersonGroup,
117                                     PersonId, stream);
118                 personStored = true;
119             }
120         }
121         catch (FaceAPIException ex)
122         {
123             if (!ex.ErrorCode.Contains("InvalidImage"))
124             {
125                 Debug.WriteLine($"Exception in StorePersonFace {ex.ErrorMessage}");
126             }
127         }
128
129         return personStored;
130     }
131
132     private void TakeFirstPicture_Click(object sender, RoutedEventArgs e)
```

Add code to the UI to take a picture, detect a face and persist it

You 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 you will create some code to access this functionality from the user interface of the application. The application can take two different pictures. This is the minimum to be able to find the identity of a person through the Face API cognitive service.

- Step 31. Find the **TakeFirstPicture_Click** method and the **TakeSecondPicture_Click** method inside **MainPage.xaml.cs** and replace them with the following code. To save time, you can copy and paste the code from [Appendix 8](#).



The screenshot shows the Visual Studio IDE with the file **MainPage.xaml.cs** open. The code is for the **CognitiveServicesLab.MainPage** class. A red rounded rectangle highlights the two methods to be replaced: **TakeFirstPicture_Click** (lines 132-143) and **TakeSecondPicture_Click** (lines 144-153). The **TrainModel_Click** method (line 155) is visible below the highlighted area. The code inside the highlighted area is as follows:

```
130 }
131
132 private async void TakeFirstPicture_Click(object sender, RoutedEventArgs e)
133 {
134     var imageStored = await StorePersonFace(FirstCapturedImage);
135
136     if (imageStored)
137     {
138         TakeFirstPicture.IsEnabled = false;
139         TakeSecondPicture.IsEnabled = true;
140         SecondPhotoStackPanel.Visibility = Visibility.Visible;
141     }
142 }
143
144 private async void TakeSecondPicture_Click(object sender, RoutedEventArgs e)
145 {
146     var imageStored = await StorePersonFace(SecondCapturedImage);
147
148     if (imageStored)
149     {
150         TakeSecondPicture.IsEnabled = false;
151         TrainModel.Visibility = Visibility.Visible;
152     }
153 }
154
155 private void TrainModel_Click(object sender, RoutedEventArgs e)
```

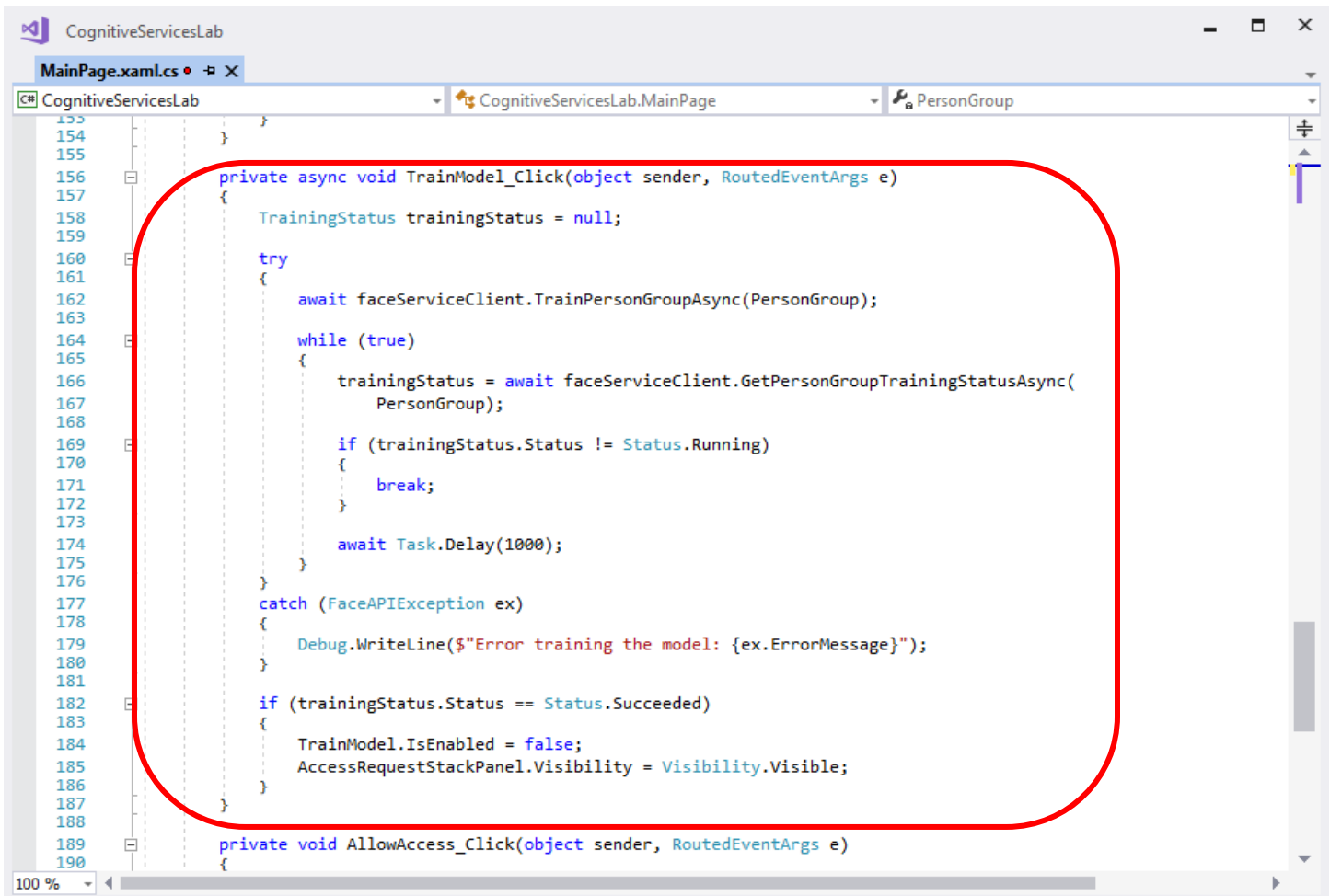
Train the person group with the faces just recognized

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** API. When using the client library, it is simply a call to the **TrainPersonGroupAsync** method.

Please note that 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** API or **GetPersonGroupTrainingStatusAsync** method of the client library.

- Step 32. Add the following using directive to the beginning of the **MainPage.xaml.cs** source file:
Microsoft.ProjectOxford.Face.Contract.

Step 33. Find the **TrainModel_Click** method and replace it with the following code. To save time, you can copy and paste the code from [Appendix 9](#).

A screenshot of the Visual Studio IDE showing the code for the `TrainModel_Click` method in `MainPage.xaml.cs`. The code is enclosed in a red rounded rectangle. The method is an asynchronous void function that takes an `object sender` and `RoutedEventArgs e` as parameters. It initializes a `TrainingStatus` variable to `null` and enters a `try` block. Inside the `try` block, it calls `await faceServiceClient.TrainPersonGroupAsync(PersonGroup);`. Then, it enters a `while (true)` loop. Inside the loop, it calls `await faceServiceClient.GetPersonGroupTrainingStatusAsync(PersonGroup);`. It then checks if `trainingStatus.Status != Status.Running`; if true, it breaks the loop. Otherwise, it calls `await Task.Delay(1000);`. After the `while` loop, it has a `catch (FaceAPIException ex)` block that writes an error message to the debug console: `Debug.WriteLine($"Error training the model: {ex.ErrorMessage}");`. Finally, it has an `if (trainingStatus.Status == Status.Succeeded)` block that sets `TrainModel.IsEnabled = false;` and `AccessRequestStackPanel.Visibility = Visibility.Visible;`. Below the `try` block, there is another method `private void AllowAccess_Click(object sender, RoutedEventArgs e)` which is partially visible.

```
155     }
156
157     private async void TrainModel_Click(object sender, RoutedEventArgs e)
158     {
159         TrainingStatus trainingStatus = null;
160
161         try
162         {
163             await faceServiceClient.TrainPersonGroupAsync(PersonGroup);
164
165             while (true)
166             {
167                 trainingStatus = await faceServiceClient.GetPersonGroupTrainingStatusAsync(
168                     PersonGroup);
169
170                 if (trainingStatus.Status != Status.Running)
171                 {
172                     break;
173                 }
174
175                 await Task.Delay(1000);
176             }
177         }
178         catch (FaceAPIException ex)
179         {
180             Debug.WriteLine($"Error training the model: {ex.ErrorMessage}");
181         }
182
183         if (trainingStatus.Status == Status.Succeeded)
184         {
185             TrainModel.IsEnabled = false;
186             AccessRequestStackPanel.Visibility = Visibility.Visible;
187         }
188     }
189
190     private void AllowAccess_Click(object sender, RoutedEventArgs e)
191     {
```

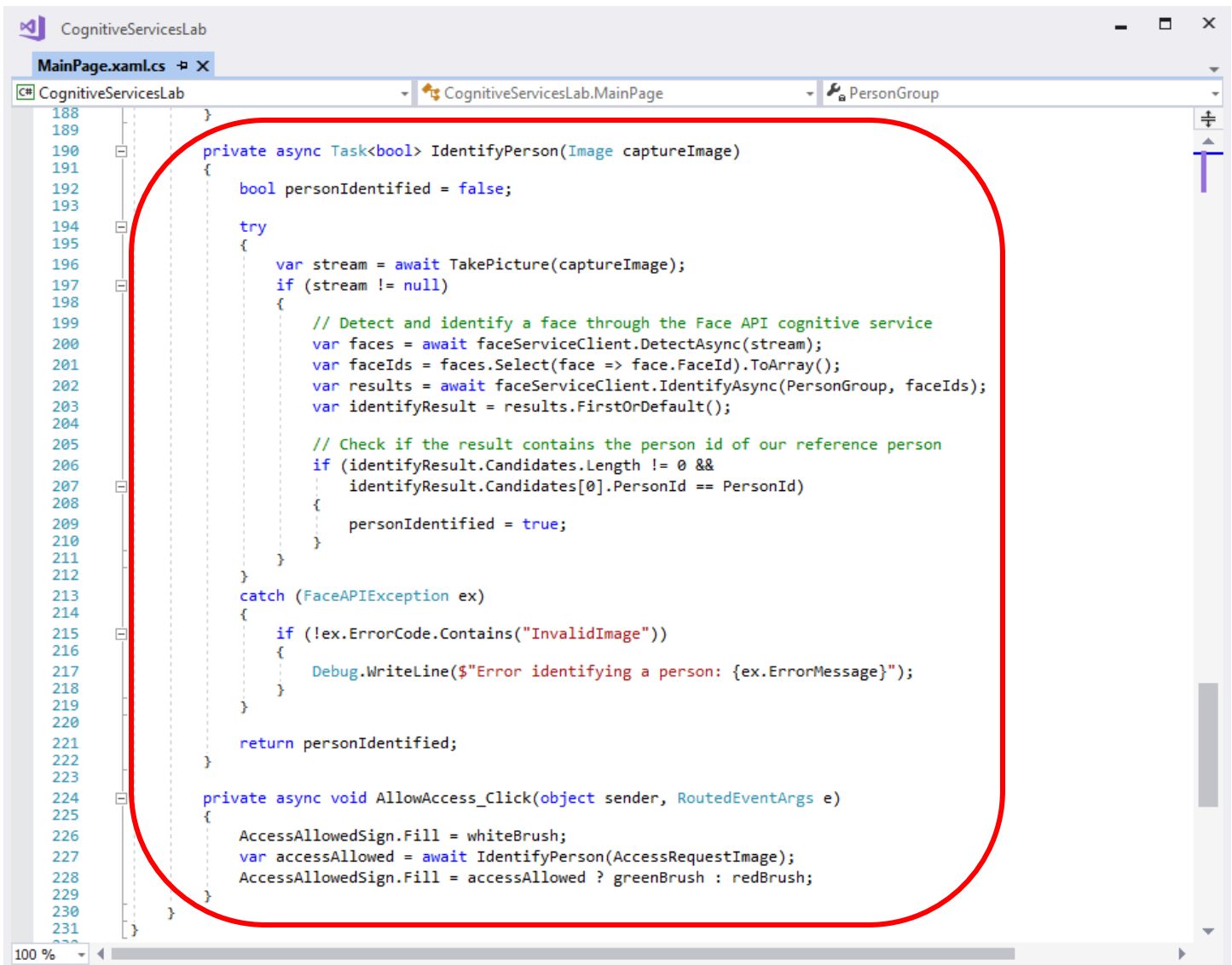
Identify a face to grant access to a person

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** 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.

- Step 34. Find the **AllowedAccess_Click** method and replace it with the following code. To save time, you can copy and paste the code from [Appendix 10](#).



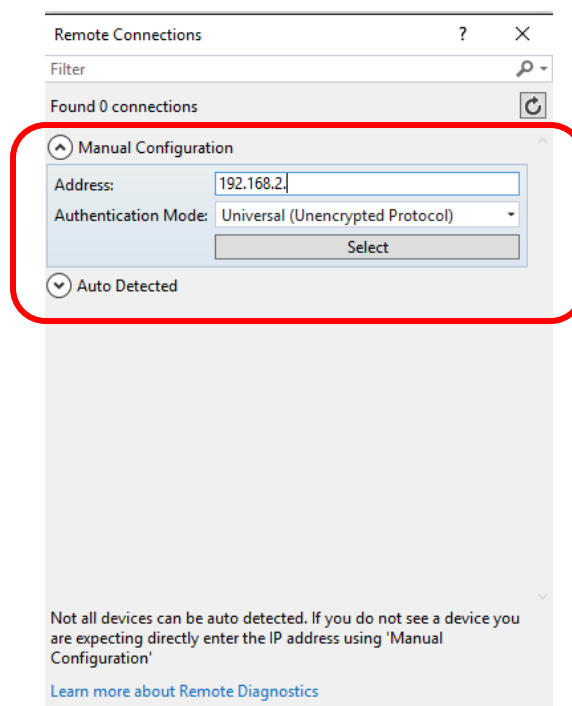
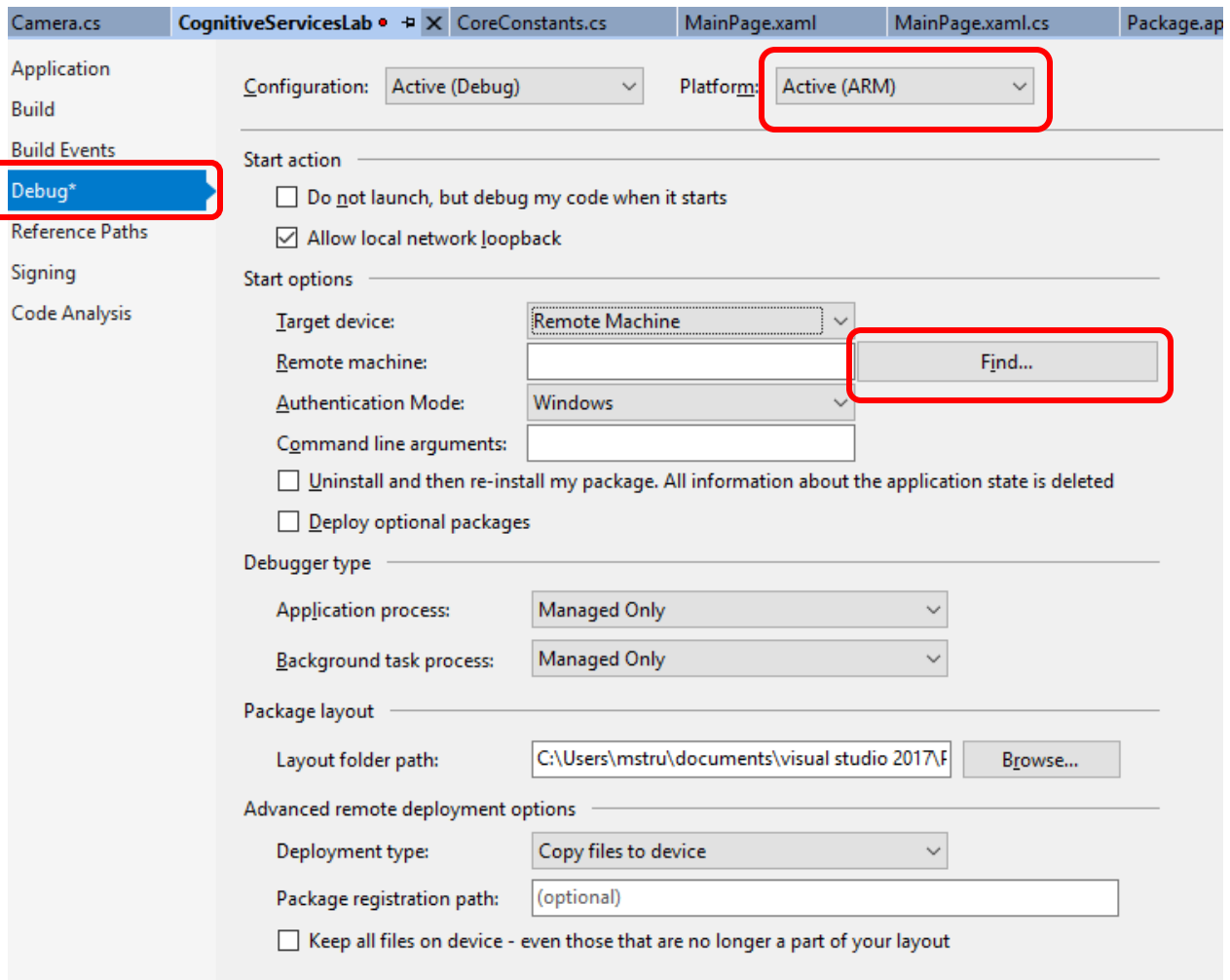
The screenshot shows the Visual Studio IDE with the file **MainPage.xaml.cs** open. The **PersonGroup** class is selected in the Solution Explorer. The **IdentifyPerson** method is highlighted with a red rounded rectangle. The code for **IdentifyPerson** is as follows:

```
188 }
189
190 private async Task<bool> IdentifyPerson(Image captureImage)
191 {
192     bool personIdentified = false;
193
194     try
195     {
196         var stream = await TakePicture(captureImage);
197         if (stream != null)
198         {
199             // Detect and identify a face through the Face API cognitive service
200             var faces = await faceServiceClient.DetectAsync(stream);
201             var faceIds = faces.Select(face => face.FaceId).ToArray();
202             var results = await faceServiceClient.IdentifyAsync(PersonGroup, faceIds);
203             var identifyResult = results.FirstOrDefault();
204
205             // Check if the result contains the person id of our reference person
206             if (identifyResult.Candidates.Length != 0 &&
207                 identifyResult.Candidates[0].PersonId == PersonId)
208             {
209                 personIdentified = true;
210             }
211         }
212     }
213     catch (FaceAPIException ex)
214     {
215         if (!ex.ErrorCode.Contains("InvalidImage"))
216         {
217             Debug.WriteLine($"Error identifying a person: {ex.ErrorMessage}");
218         }
219     }
220
221     return personIdentified;
222 }
223
224 private async void AllowedAccess_Click(object sender, RoutedEventArgs e)
225 {
226     AccessAllowedSign.Fill = whiteBrush;
227     var accessAllowed = await IdentifyPerson(AccessRequestImage);
228     AccessAllowedSign.Fill = accessAllowed ? greenBrush : redBrush;
229 }
230
231 }
```

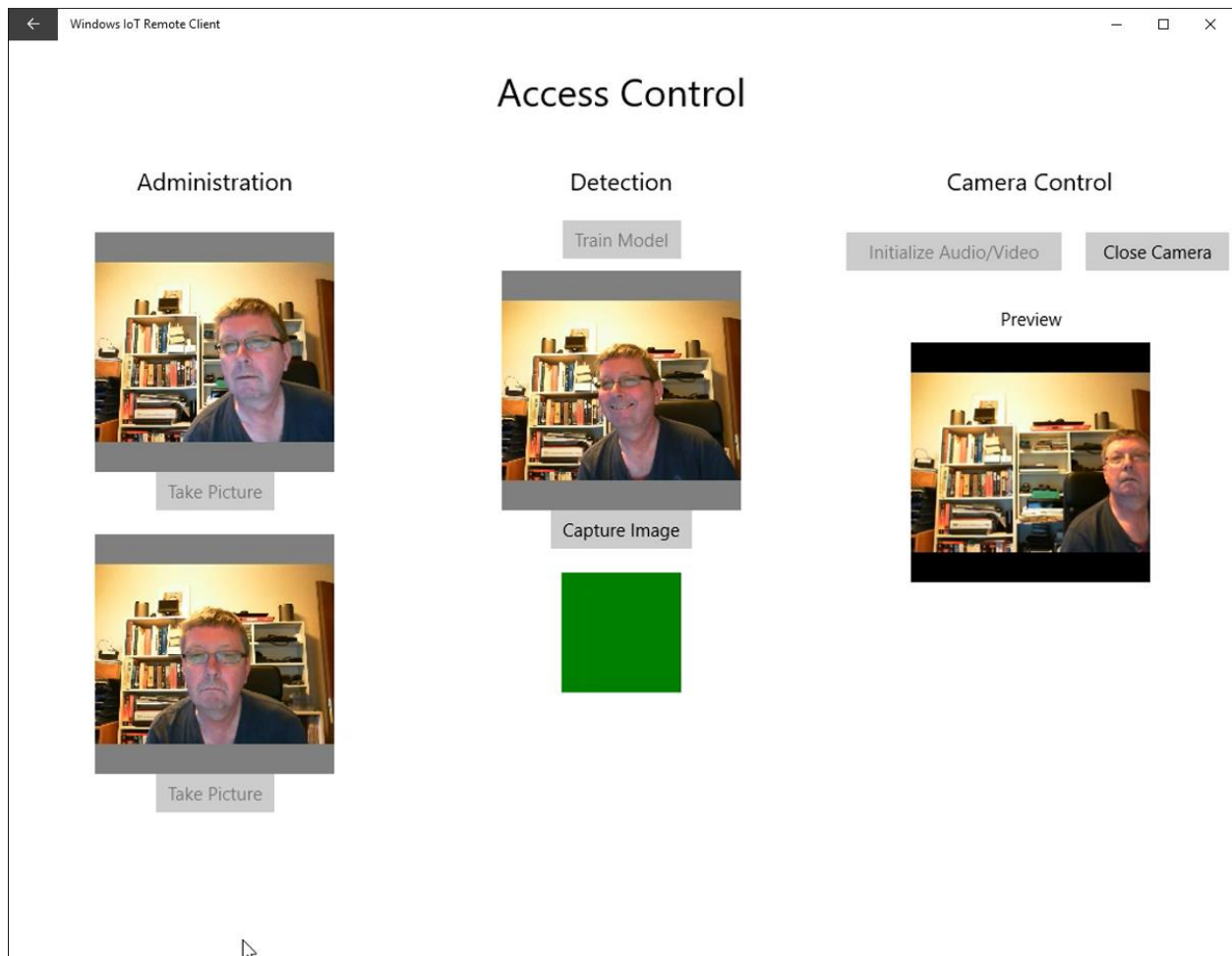
Test your application on a Raspberry Pi

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

- Step 35. Make sure that compile your application for ARM.
- Step 36. 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.
- Step 37. Specify a remote machine on which to deploy the application using the ip address or the name of your Raspberry Pi. 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.
- Step 38. In the Debug Tab, select **Remote Machine** as **Target device** and click on **Find** to specify the connection details of the target device.



- Step 39. After entering a device name or the device IP address, click on **Select**.
- Step 40. Start debugging by pressing F5 or by clicking the run button on the Visual Studio menu.
- Step 41. After a while your application will automatically start on the Raspberry Pi.
- Step 42. Click on Initialize Video to start previewing through the connected video camera.
- Step 43. Take two different pictures of the same person (yourself)
- Step 44. In the detection part of the UI, click the Train Model button
- Step 45. Finally, take another picture using the Capture Image button and see if you get access which is simulated by the green rectangle.
- Step 46. Take a few picture of other people to see that they don't get access (because their faces are not recognized).



This completes this lab.

Appendix 1 – Code for class Camera.cs

```
using System;
using System.Linq;
using System.Threading.Tasks;
using Windows.Devices.Enumeration;
using Windows.Media.Capture;

namespace CognitiveServicesLab
{
    public class Camera
    {
        public MediaCapture MyMediaCapture { get; private set; }

        public Camera()
        {
            MyMediaCapture = new MediaCapture();
        }

        public async Task InitializeCameraAsync()
        {
            var myCamera = (await DeviceInformation.
                FindAllAsync(DeviceClass.VideoCapture)).FirstOrDefault();

            await MyMediaCapture?.InitializeAsync(
                new MediaCaptureInitializationSettings
                {
                    VideoDeviceId = myCamera.Id
                });
        }
    }
}
```

Appendix 2 – Code for class CoreConstants.cs

```
namespace CognitiveServicesLab
{
    public class CoreConstants
    {
        private static string CognitiveServicesRegion = "<your region>";
        public static string CognitiveServicesBaseUrl =
            $"https://{CognitiveServicesRegion}.api.cognitive.microsoft.com/face/v1.0";
        public static string FaceApiSubscriptionKey = "<your subscription key>";
    }
}
```

Appendix 3 – Variable declarations for MainForm.cs

```
private string PersonGroup { get; } = Guid.NewGuid().ToString();
private string Person { get; } = Guid.NewGuid().ToString();
private Guid PersonId { get; set; }

private readonly SolidColorBrush whiteBrush = new SolidColorBrush(Colors.White);
private readonly SolidColorBrush redBrush = new SolidColorBrush(Colors.Red);
private readonly SolidColorBrush greenBrush = new SolidColorBrush(Colors.Green);

private FaceServiceClient faceServiceClient;
private Camera camera = new Camera();
```

Appendix 4 – Code for the Page_Loaded method

```
private async void Page_Loaded(object sender, RoutedEventArgs e)
{
    faceServiceClient = new FaceServiceClient(CoreConstants.FaceApiSubscriptionKey,
        CoreConstants.CognitiveServicesBaseUrl);

    // Clean-up possible PersonGroup objects (including persons attached to them)
    try
    {
        var pgList = await faceServiceClient.ListPersonGroupsAsync();

        foreach (var pg in pgList)
        {
            await faceServiceClient.DeletePersonGroupAsync(pg.PersonGroupId);
        }
    }
    catch (FaceAPIException ex)
    {
        Debug.WriteLine($"Error deleting old PersonGroup objects: {ex.ErrorMessage}");
    }

    await faceServiceClient.CreatePersonGroupAsync(PersonGroup, PersonGroup);
    PersonId = (await faceServiceClient.CreatePersonAsync(PersonGroup, Person)).PersonId;
}
```

Appendix 5 – Code for Initializing and Exiting Video

```
private async void InitVideo_Click(object sender, RoutedEventArgs e)
{
    // Initialize the camera and start previewing
    await camera.InitializeCameraAsync();
    PreviewElement.Source = camera.MyMediaCapture;
    await camera.MyMediaCapture.StartPreviewAsync();

    // Enable button to capture a photo
    PreviewStackPanel.Visibility = Visibility.Visible;
    InitVideo.IsEnabled = false;
    ExitVideo.IsEnabled = true;
    TakeFirstPicture.IsEnabled = true;
}

private void ExitVideo_Click(object sender, RoutedEventArgs e)
{
    camera.MyMediaCapture.Dispose();
    InitVideo.IsEnabled = true;
    ExitVideo.IsEnabled = false;
}
```

Appendix 6 – Code to take a picture and store it in a memory stream

```
private async Task<Stream> TakePicture(Image captureImage)
{
    IRandomAccessStream photoStream = null;

    try
    {
        photoStream = new InMemoryRandomAccessStream();
        await camera.MyMediaCapture.CapturePhotoToStreamAsync(
            ImageEncodingProperties.CreateJpeg(), photoStream);
        photoStream.Seek(0L);
        var bitmap = new BitmapImage();
        bitmap.SetSource(photoStream);
        captureImage.Source = bitmap;
        photoStream.Seek(0L);
    }
    catch (Exception ex)
    {
        Debug.WriteLine($"Exception in TakePicture: {ex.Message}");
    }

    return photoStream?.AsStream();
}
```


Appendix 7 – Code to detect a face and store it in a Person's PersonGroup

```
private async Task<bool> StorePersonFace(Image captureImage)
{
    bool personStored = false;

    try
    {
        var stream = await TakePicture(captureImage);
        if (stream != null)
        {
            var faceStored = await faceServiceClient.AddPersonFaceAsync(PersonGroup,
                PersonId, stream);
            personStored = true;
        }
    }
    catch (FaceAPIException ex)
    {
        if (!ex.ErrorCode.Contains("InvalidImage"))
        {
            Debug.WriteLine($"Exception in StorePersonFace {ex.ErrorMessage}");
        }
    }

    return personStored;
}
```

Appendix 8 - Code to take a picture, detect a face and persist it

```
private async void TakeFirstPicture_Click(object sender, RoutedEventArgs e)
{
    var imageStored = await StorePersonFace(FirstCapturedImage);

    if (imageStored)
    {
        TakeFirstPicture.IsEnabled = false;
        TakeSecondPicture.IsEnabled = true;
        SecondPhotoStackPanel.Visibility = Visibility.Visible;
    }
}

private async void TakeSecondPicture_Click(object sender, RoutedEventArgs e)
{
    var imageStored = await StorePersonFace(SecondCapturedImage);

    if (imageStored)
    {
        TakeSecondPicture.IsEnabled = false;
        TrainModel.Visibility = Visibility.Visible;
    }
}
```

Appendix 9 – Code to train the person group with the faces just recognized

```
private async void TrainModel_Click(object sender, RoutedEventArgs e)
{
    TrainingStatus trainingStatus = null;

    try
    {
        await faceServiceClient.TrainPersonGroupAsync(PersonGroup);

        while (true)
        {
            trainingStatus = await faceServiceClient.GetPersonGroupTrainingStatusAsync(
                PersonGroup);

            if (trainingStatus.Status != Status.Running)
            {
                break;
            }

            await Task.Delay(1000);
        }
    }
    catch (FaceAPIException ex)
    {
        Debug.WriteLine($"Error training the model: {ex.ErrorMessage}");
    }

    if (trainingStatus.Status == Status.Succeeded)
    {
        TrainModel.IsEnabled = false;
        AccessRequestStackPanel.Visibility = Visibility.Visible;
    }
}
```

Appendix 10 – Code to identify a face to grant access to a person

```
private async Task<bool> IdentifyPerson(Image captureImage)
{
    bool personIdentified = false;

    try
    {
        var stream = await TakePicture(captureImage);
        if (stream != null)
        {
            // Detect and identify a face through the Face API cognitive service
            var faces = await faceServiceClient.DetectAsync(stream);
            var faceIds = faces.Select(face => face.FaceId).ToArray();
            var results = await faceServiceClient.IdentifyAsync(PersonGroup, faceIds);
            var identifyResult = results.FirstOrDefault();

            // Check if the result contains the person id of our reference person
            if (identifyResult.Candidates.Length != 0 &&
                identifyResult.Candidates[0].PersonId == PersonId)
            {
                personIdentified = true;
            }
        }
    }
    catch (FaceAPIException ex)
    {
        if (!ex.ErrorCode.Contains("InvalidImage"))
        {
            Debug.WriteLine($"Error identifying a person: {ex.ErrorMessage}");
        }
    }

    return personIdentified;
}

private async void AllowAccess_Click(object sender, RoutedEventArgs e)
{
    AccessAllowedSign.Fill = whiteBrush;
    var accessAllowed = await IdentifyPerson(AccessRequestImage);
    AccessAllowedSign.Fill = accessAllowed ? greenBrush : redBrush;
}
```