Contents

What is the Azure Face API?

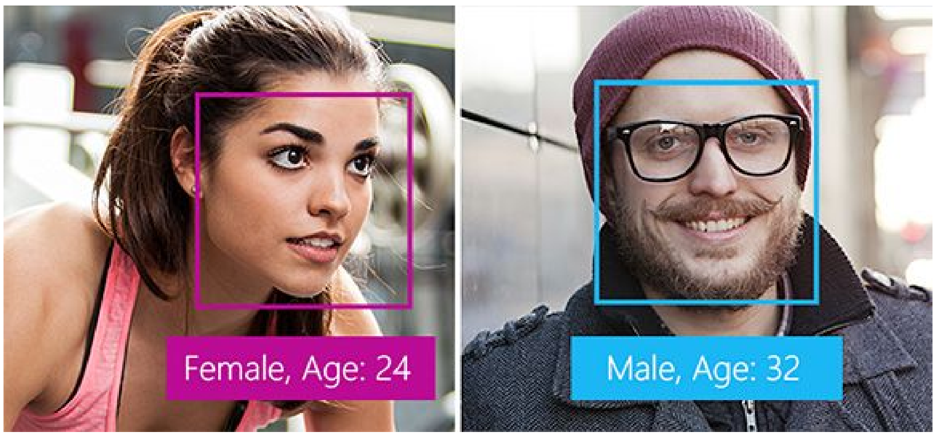
11/1/2018 • 3 minutes to read • Edit Online

The Azure Face API is a cognitive service that provides algorithms for detecting, recognizing, and analyzing human faces in images. The ability to process human face information is important in many different software scenarios, such as security, natural user interface, image content analysis and management, mobile apps, and robotics.

The Face API provides several different functions, each outlined in the following sections. Read on to learn more about each one and determine if it suits your needs.

Face detection

The Face API can detect human faces in an image and return the rectangle coordinates of their locations. Optionally, face detection can extract a series of face-related attributes such as pose, gender, age, head pose, facial hair, and glasses.



The face detection feature is also available through the Computer Vision API, but if you wish to do further operations with face data, you should use the Face API (this service). For more information on face detection, see the Detect API.

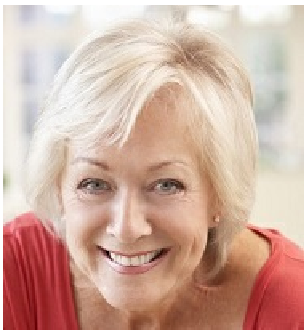
Face verification

The Verify API performs an authentication against two detected faces or from one detected face to one person object. Practically, it evaluates whether two faces belong to the same person. This is potentially useful in security scenarios. For more information, see the Verify API.

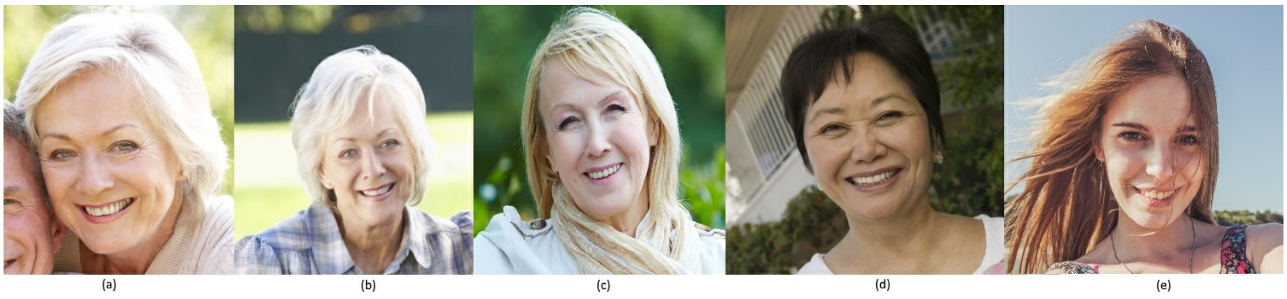
Find similar faces

The Find Similar API takes a target face and a set of candidate faces and finds a smaller set of faces that look most similar to the target face. Two working modes, **matchPerson** and **matchFace** are supported. **matchPerson** mode returns similar faces after filtering for the same person (using the Verify API). **matchFace** mode ignores the same-person filter and returns a list of similar candidate faces that may or may not belong to the same person.

In the following example, this is the target face:



And these are the candidate faces:



To find four similar faces, **matchPerson** mode would return (a) and (b), which depict the same person as the target face. **matchFace** mode returns (a), (b), (c) and (d)—exactly four candidates, even if some are not the same person as the target or have low similarity. For more information, see the Find Similar API.

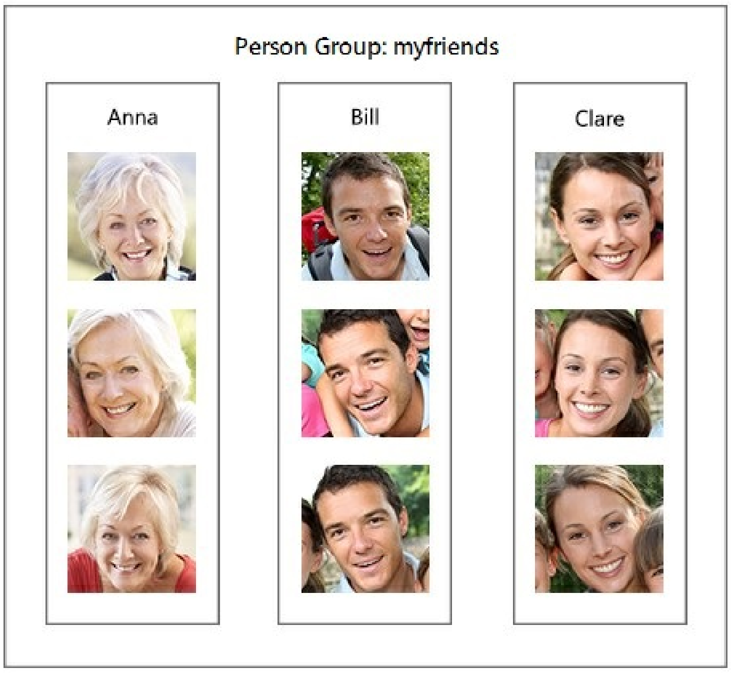
Face grouping

The Group API divides a set of unknown faces into several groups based on similarity. Each group is a disjoint proper subset of the original set of faces. All of the faces in a group are likely to belong to the same person, but there can be several different groups for a single person (differentiated by another factor, such as expression for example). For more information, see the Group API.

Person identification

The Identify API can be used to identify a detected face against a database of people. This may be useful for automatic image tagging in photo management software. You create the database in advance, and it can be edited over time.

The following image depicts an example of a database named "myfriends." Each group may contain up to 1,000,000 different person objects, and each person object can have up to 248 faces registered.



After a database has been created and trained, you can perform identification against the group with a new detected face. If the face is identified as a person in the group, the person object is returned.

For more information about person identification, see the Identify API.

Sample apps

The following sample applications showcase a few of the ways the Face API can be used.

- Microsoft Face API: Windows Client Library & Sample - a WPF app that demonstrates several scenarios of Face detection, analysis and identification.
- FamilyNotes UWP app - a Universal Windows Platform (UWP) app that uses face identification along with speech, Cortana, ink, and camera in a family note-sharing scenario.

Next steps

Follow a quickstart to implement a simple face detection scenario in code.

- Quickstart: Detect faces in an image using the .NET SDK with C# (other languages available)

Quickstart: Detect faces in an image using the .NET SDK with C#

10/23/2018 • 3 minutes to read • Edit Online

In this quickstart, you detect human faces in an image using the Face Windows client library.

Prerequisites

- You need a subscription key to run the sample. You can get free trial subscription keys from Try Cognitive Services.
- Any edition of Visual Studio 2017.
- The Microsoft.Azure.CognitiveServices.Vision.Face 2.2.0-preview client library NuGet package. It isn't necessary to download the package. Installation instructions are provided below.

DetectWithUrlAsync method

TIP
Get the latest code as a Visual Studio solution from Github.

The `DetectWithUrlAsync` and `DetectWithStreamAsync` methods wrap the Face - Detect API for remote and local images, respectively. You can use these methods to detect faces in an image and return face attributes including:

- Face ID: Unique ID used in several Face API scenarios.
- Face Rectangle: The left, top, width, and height indicating the location of the face in the image.
- Landmarks: An array of 27-point face landmarks pointing to the important positions of face components.
- Facial attributes including age, gender, smile intensity, head pose, and facial hair.

To run the sample, do the following steps:

1. Create a new Visual C# Console App in Visual Studio.
2. Install the Face client library NuGet package.
   a. On the top menu, click **Tools**, select **NuGet Package Manager**, then **Manage NuGet Packages for Solution**.
   b. Click the **Browse** tab and then select **Include prerelease**.
   c. In the **Search** box type "Microsoft.Azure.CognitiveServices.Vision.Face".
   d. Select **Microsoft.Azure.CognitiveServices.Vision.Face** when it displays, then click the checkbox next to your project name, and **Install**.
3. Replace *Program.cs* with the following code.
4. Replace `<Subscription Key>` with your valid subscription key.
5. Change `faceEndpoint` to the Azure region associated with your subscription keys, if necessary.
6. Optionally, replace < `LocalImage>` with the path and file name of a local image (will be ignored if not set).
7. Optionally, set `remoteImageUrl` to a different image.
8. Run the program.

```
using Microsoft.Azure.CognitiveServices.Vision.Face;
using Microsoft.Azure.CognitiveServices.Vision.Face.Models;
using System;
using System.Collections.Generic;
using System.IO;
using System.Threading.Tasks;

namespace DetectFace
{
    class Program
    {
        // subscriptionKey = "0123456789abcdef0123456789ABCDEF"
        private const string subscriptionKey = "<SubscriptionKey>";

        // You must use the same region as you used to get your subscription
        // keys. For example, if you got your subscription keys from westus,
        // replace "westcentralus" with "westus".
        //
        // Free trial subscription keys are generated in the westcentralus
        // region. If you use a free trial subscription key, you shouldn't
        // need to change the region.
        // Specify the Azure region
        private const string faceEndpoint =
            "https://westcentralus.api.cognitive.microsoft.com";

        // localImagePath = @"C:\Documents\LocalImage.jpg"
        private const string localImagePath = @"<LocalImage>";

        private const string remoteImageUrl =
            "https://upload.wikimedia.org/wikipedia/commons/3/37/Dagestani_man_and_woman.jpg";

        private static readonly FaceAttributeType[] faceAttributes =
            { FaceAttributeType.Age, FaceAttributeType.Gender };

        static void Main(string[] args)
        {
            FaceClient faceClient = new FaceClient(
                new ApiKeyServiceClientCredentials(subscriptionKey),
                new System.Net.Http.DelegatingHandler[] { });
            faceClient.Endpoint = faceEndpoint;

            Console.WriteLine("Faces being detected ...");
            var t1 = DetectRemoteAsync(faceClient, remoteImageUrl);
            var t2 = DetectLocalAsync(faceClient, localImagePath);

            Task.WhenAll(t1, t2).Wait(5000);
            Console.WriteLine("Press any key to exit");
            Console.ReadLine();
        }

        // Detect faces in a remote image
        private static async Task DetectRemoteAsync(
            FaceClient faceClient, string imageUrl)
        {
            if (!Uri.IsWellFormedUriString(imageUrl, UriKind.Absolute))
            {
                Console.WriteLine("\nInvalid remoteImageUrl:\n{0} \n", imageUrl);
                return;
            }

            try
            {
                IList<DetectedFace> faceList =
                    await faceClient.Face.DetectWithUrlAsync(
                        imageUrl, true, false, faceAttributes);

                DisplayAttributes(GetFaceAttributes(faceList, imageUrl), imageUrl);
            }
            catch (APIErrorException e)
            {
                Console.WriteLine(imageUrl + ": " + e.Message);
            }
        }

        // Detect faces in a local image
        private static async Task DetectLocalAsync(FaceClient faceClient, string imagePath)
        {
            if (!File.Exists(imagePath))
            {
                Console.WriteLine(
                    "\nUnable to open or read localImagePath:\n{0} \n", imagePath);
                return;
            }

            try
            {
                using (Stream imageStream = File.OpenRead(imagePath))
                {
                    IList<DetectedFace> faceList =
                            await faceClient.Face.DetectWithStreamAsync(
                                imageStream, true, false, faceAttributes);
                    DisplayAttributes(
                        GetFaceAttributes(faceList, imagePath), imagePath);
                }
            }
            catch (APIErrorException e)
            {
                Console.WriteLine(imagePath + ": " + e.Message);
            }
        }

        private static string GetFaceAttributes(
            IList<DetectedFace> faceList, string imagePath)
        {
            string attributes = string.Empty;

            foreach (DetectedFace face in faceList)
            {
                double? age = face.FaceAttributes.Age;
                string gender = face.FaceAttributes.Gender.ToString();
                attributes += gender + " " + age + "   ";
            }

            return attributes;
        }

        // Display the face attributes
        private static void DisplayAttributes(string attributes, string imageUri)
        {
            Console.WriteLine(imageUri);
            Console.WriteLine(attributes + "\n");
        }
    }
}
```

DetectWithUrlAsync response

A successful response displays the gender and age for each face in the image.

See API Quickstarts: Detect faces in an image using C# for an example of raw JSON output.

```
https://upload.wikimedia.org/wikipedia/commons/3/37/Dagestani_man_and_woman.jpg
Male 37   Female 56
```

Next steps

Learn how to create a WPF Windows application that uses the Face service to detect faces in an image. The application draws a frame around each face and displays a description of the face on the status bar.

Tutorial: Create a WPF app to detect and frame faces in an image

Quickstart: Detect faces in an image using the REST API and C#

10/23/2018 • 4 minutes to read • Edit Online

In this quickstart, you detect human faces in an image using the Face API.

Prerequisites

You need a subscription key to run the sample. You can get free trial subscription keys from Try Cognitive Services.

Detect faces in an image

Use the Face - Detect method to detect faces in an image and return face attributes including:

- Face ID: Unique ID used in several Face API scenarios.
- Face Rectangle: The left, top, width, and height indicating the location of the face in the image.
- Landmarks: An array of 27-point face landmarks pointing to the important positions of face components.
- Facial attributes including age, gender, smile intensity, head pose, and facial hair.

To run the sample, do the following steps:

1. Create a new Visual C# Console App in Visual Studio.
2. Replace Program.cs with the following code.
3. Replace `<Subscription Key>` with your valid subscription key.
4. Change the `uriBase` value to use the location where you obtained your subscription keys, if necessary.
5. Run the program.
6. At the prompt, enter the path to an image.

Face - Detect request

```
using System;
using System.IO;
using System.Net.Http;
using System.Net.Http.Headers;
using System.Text;

namespace CSHttpClientSample
{
    static class Program
    {
        // Replace <Subscription Key> with your valid subscription key.
        const string subscriptionKey = "<Subscription Key>";

        // NOTE: You must use the same region in your REST call as you used to
        // obtain your subscription keys. For example, if you obtained your
        // subscription keys from westus, replace "westcentralus" in the URL
        // below with "westus".
        //
        // Free trial subscription keys are generated in the westcentralus region.
        // If you use a free trial subscription key, you shouldn't need to change
        // this region.
        const string uriBase =
            "https://westcentralus.api.cognitive.microsoft.com/face/v1.0/detect";

        static void Main()
        {
            // Get the path and filename to process from the user.
            Console.WriteLine("Detect faces:");
            Console.Write(
                "Enter the path to an image with faces that you wish to analyze: ");
            string imageFilePath = Console.ReadLine();

            if (File.Exists(imageFilePath))
            {
                // Execute the REST API call.
                try
                {
                    MakeAnalysisRequest(imageFilePath);
                    Console.WriteLine("\nWait a moment for the results to appear.\n");
                }
                catch (Exception e)
                {
                    Console.WriteLine("\n" + e.Message + "\nPress Enter to exit...\n");
                }
            }
            else
            {
                Console.WriteLine("\nInvalid file path.\nPress Enter to exit...\n");
            }
            Console.ReadLine();
        }


        /// <summary>
        /// Gets the analysis of the specified image by using the Face REST API.
        /// </summary>
        /// <param name="imageFilePath">The image file.</param>
        static async void MakeAnalysisRequest(string imageFilePath)
        {
            HttpClient client = new HttpClient();

            // Request headers.
            client.DefaultRequestHeaders.Add(
                "Ocp-Apim-Subscription-Key", subscriptionKey);

            // Request parameters. A third optional parameter is "details".
            string requestParameters = "returnFaceId=true&returnFaceLandmarks=false" +
                "&returnFaceAttributes=age,gender,headPose,smile,facialHair,glasses," +
                "emotion,hair,makeup,occlusion,accessories,blur,exposure,noise";

            // Assemble the URI for the REST API Call.
            string uri = uriBase + "?" + requestParameters;

            HttpResponseMessage response;

            // Request body. Posts a locally stored JPEG image.
            byte[] byteData = GetImageAsByteArray(imageFilePath);

            using (ByteArrayContent content = new ByteArrayContent(byteData))
            {
                // This example uses content type "application/octet-stream".
                // The other content types you can use are "application/json"
                // and "multipart/form-data".
                content.Headers.ContentType =
                    new MediaTypeHeaderValue("application/octet-stream");

                // Execute the REST API call.
                response = await client.PostAsync(uri, content);

                // Get the JSON response.
                string contentString = await response.Content.ReadAsStringAsync();

                // Display the JSON response.
                Console.WriteLine("\nResponse:\n");
                Console.WriteLine("\nResponse:\n");
                Console.WriteLine(JsonPrettyPrint(contentString));
                Console.WriteLine("\nPress Enter to exit...");
            }
        }


        /// <summary>
        /// Returns the contents of the specified file as a byte array.
        /// </summary>
        /// <param name="imageFilePath">The image file to read.</param>
        /// <returns>The byte array of the image data.</returns>
        static byte[] GetImageAsByteArray(string imageFilePath)
        {
            using (FileStream fileStream =
                new FileStream(imageFilePath, FileMode.Open, FileAccess.Read))
            {
                BinaryReader binaryReader = new BinaryReader(fileStream);
                return binaryReader.ReadBytes((int)fileStream.Length);
            }
        }


        /// <summary>
        /// Formats the given JSON string by adding line breaks and indents.
        /// </summary>
        /// <param name="json">The raw JSON string to format.</param>
        /// <returns>The formatted JSON string.</returns>
        static string JsonPrettyPrint(string json)
        {
            if (string.IsNullOrEmpty(json))
                return string.Empty;

            json = json.Replace(Environment.NewLine, "").Replace("\t", "");

            StringBuilder sb = new StringBuilder();
            bool quote = false;
            bool ignore = false;
            int offset = 0;
            int indentLength = 3;

            foreach (char ch in json)
            {
                switch (ch)
                {
                    case '"':
                        if (!ignore) quote = !quote;
                        break;
                    case '\'':
                        if (quote) ignore = !ignore;
                        break;
                }

                if (quote)
                    sb.Append(ch);
                else
                {
                    switch (ch)
                    {
                        case '{':
                        case '[':
                            sb.Append(ch);
                            sb.Append(Environment.NewLine);
                            sb.Append(new string(' ', ++offset * indentLength));
                            break;
                        case '}':
                        case ']':
                            sb.Append(Environment.NewLine);
                            sb.Append(new string(' ', --offset * indentLength));
                            sb.Append(ch);
```

```
                            sb.Append(ch);
                            break;
                        case ',':
                            sb.Append(ch);
                            sb.Append(Environment.NewLine);
                            sb.Append(new string(' ', offset * indentLength));
                            break;
                        case ':':
                            sb.Append(ch);
                            sb.Append(' ');
                            break;
                        default:
                            if (ch != ' ') sb.Append(ch);
                            break;
                    }
                }
            }

            return sb.ToString().Trim();
        }
    }
}
```

Face - Detect response

A successful response is returned in JSON, for example:

```
[
   {
      "faceId": "f7eda569-4603-44b4-8add-cd73c6dec644",
      "faceRectangle": {
         "top": 131,
         "left": 177,
         "width": 162,
         "height": 162
      },
      "faceAttributes": {
         "smile": 0.0,
         "headPose": {
            "pitch": 0.0,
            "roll": 0.1,
            "yaw": -32.9
         },
         "gender": "female",
         "age": 22.9,
         "facialHair": {
            "moustache": 0.0,
            "beard": 0.0,
            "sideburns": 0.0
         },
         "glasses": "NoGlasses",
         "emotion": {
            "anger": 0.0,
            "contempt": 0.0,
            "disgust": 0.0,
            "fear": 0.0,
            "happiness": 0.0,
            "neutral": 0.986,
            "sadness": 0.009,
            "surprise": 0.005
         },
         "blur": {
            "blurLevel": "low",
            "value": 0.06
         },
         "exposure": {
            "exposureLevel": "goodExposure",
            "value": 0.67
```

```
          },
          "noise": {
             "noiseLevel": "low",
             "value": 0.0
          },
          "makeup": {
             "eyeMakeup": true,
             "lipMakeup": true
          },
          "accessories": [

          ],
          "occlusion": {
             "foreheadOccluded": false,
             "eyeOccluded": false,
             "mouthOccluded": false
          },
          "hair": {
             "bald": 0.0,
             "invisible": false,
             "hairColor": [
                {
                   "color": "brown",
                   "confidence": 1.0
                },
                {
                   "color": "black",
                   "confidence": 0.87
                },
                {
                   "color": "other",
                   "confidence": 0.51
                },
                {
                   "color": "blond",
                   "confidence": 0.08
                },
                {
                   "color": "red",
                   "confidence": 0.08
                },
                {
                   "color": "gray",
                   "confidence": 0.02
                }
             ]
          }
       }
    }
]
```

Next steps

Learn how to create a WPF Windows application that uses the Face service to detect faces in an image. The application draws a frame around each face and displays a description of the face on the status bar.

Tutorial: Getting Started with Face API in C#

Quickstart: Detect faces in an image using the REST API and cURL

10/23/2018 • 2 minutes to read • Edit Online

In this quickstart, you detect faces in an image using Face API.

Prerequisites

You need a subscription key to run the sample. You can get free trial subscription keys from Try Cognitive Services.

Detect faces in an image

Use the Face - Detect method to detect faces in an image and return face attributes including:

- Face ID: Unique ID used in several Face API scenarios.
- Face Rectangle: The left, top, width, and height indicating the location of the face in the image.
- Landmarks: An array of 27-point face landmarks pointing to the important positions of face components.
- Facial attributes including age, gender, smile intensity, head pose, and facial hair.

To run the sample, do the following steps:

1. Open a Command Prompt.
2. Replace `<Subscription Key>` with your valid subscription key.
3. Change the URL (`https://westcentralus.api.cognitive.microsoft.com/face/v1.0/detect`) to use the location where you obtained your subscription keys, if necessary.
4. Optionally, change the image (`"{\"url\":...`) to analyze.
5. Paste the code in the command window.
6. Run the command.

Face - Detect request

> **NOTE**
>
> You must use the same location in your REST call as you used to obtain your subscription keys. For example, if you obtained your subscription keys from westus, replace "westcentralus" in the following URL with "westus".

```
curl -H "Ocp-Apim-Subscription-Key: <Subscription Key>"
"https://westcentralus.api.cognitive.microsoft.com/face/v1.0/detect?
returnFaceId=true&returnFaceLandmarks=false&returnFaceAttributes=age,gender,headPose,smile,facialHair,glasses,e
motion,hair,makeup,occlusion,accessories,blur,exposure,noise" -H "Content-Type: application/json" --data-ascii
"{\"url\":\"https://upload.wikimedia.org/wikipedia/commons/c/c3/RH_Louise_Lillian_Gish.jpg\"}"
```

Face - Detect response

A successful response is returned in JSON.

```
[
  {
    "faceId": "49d55c17-e018-4a42-ba7b-8cbbdfae7c6f",
    "faceRectangle": {
      "top": 131,
        "left": 177,
        "width": 162,
        "height": 162
      },
      "faceAttributes": {
        "smile": 0,
        "headPose": {
          "pitch": 0,
          "roll": 0.1,
          "yaw": -32.9
        },
        "gender": "female",
        "age": 22.9,
        "facialHair": {
          "moustache": 0,
          "beard": 0,
          "sideburns": 0
        },
        "glasses": "NoGlasses",
        "emotion": {
          "anger": 0,
          "contempt": 0,
          "disgust": 0,
          "fear": 0,
          "happiness": 0,
          "neutral": 0.986,
          "sadness": 0.009,
          "surprise": 0.005
        },
        "blur": {
          "blurLevel": "low",
          "value": 0.06
        },
        "exposure": {
          "exposureLevel": "goodExposure",
          "value": 0.67
        },
        "noise": {
          "noiseLevel": "low",
          "value": 0
        },
        "makeup": {
          "eyeMakeup": true,
          "lipMakeup": true
        },
        "accessories": [],
        "occlusion": {
          "foreheadOccluded": false,
          "eyeOccluded": false,
          "mouthOccluded": false
        },
        "hair": {
          "bald": 0,
          "invisible": false,
          "hairColor": [
            {
              "color": "brown",
              "confidence": 1
            },
            {
              "color": "black",
              "confidence": 0.87
            },
            {
              "color": "other",
              "confidence": 0.51
            },
            {
              "color": "blond",
            "confidence": 0.08
          },
          {
            "color": "red",
            "confidence": 0.08
          },
          {
            "color": "gray",
            "confidence": 0.02
          }
        ]
      }
    }
  }
]
```

Next steps

Explore the Face APIs used to detect human faces in an image, demarcate the faces with rectangles, and return attributes such as age and gender.

Face APIs

Quickstart: Detect faces in an image using the REST API and Go

10/23/2018 • 3 minutes to read • Edit Online

In this quickstart, you detect human faces in an image using the Face API.

Prerequisites

You need a subscription key to run the sample. You can get free trial subscription keys from Try Cognitive Services.

Face - Detect request

Use the Face - Detect method to detect faces in an image and return face attributes including:

- Face ID: Unique ID used in several Face API scenarios.
- Face Rectangle: The left, top, width, and height indicating the location of the face in the image.
- Landmarks: An array of 27-point face landmarks pointing to the important positions of face components.
- Facial attributes including age, gender, smile intensity, head pose, and facial hair.

To run the sample, do the following steps:

1. Copy the following code into an editor.
2. Replace `<Subscription Key>` with your valid subscription key.
3. Change the `uriBase` value to the location where you got your subscription keys, if necessary.
4. Optionally, change the `imageUrl` value to the image you want to analyze.
5. Save the file with a `.go` extension.
6. Open a command prompt on a computer with Go installed.
7. Build the file, for example: `go build detect-face.go`.
8. Run the file, for example: `detect-face`.

```go
package main

import (
    "encoding/json"
    "fmt"
    "io/ioutil"
    "net/http"
    "strings"
    "time"
)

func main() {
    const subscriptionKey = "<Subscription Key>"

    // You must use the same location in your REST call as you used to get your
    // subscription keys. For example, if you got your subscription keys from
    // westus, replace "westcentralus" in the URL below with "westus".
    const uriBase =
      "https://westcentralus.api.cognitive.microsoft.com/face/v1.0/detect"
    const imageUrl =
      "https://upload.wikimedia.org/wikipedia/commons/3/37/Dagestani_man_and_woman.jpg"

    const params = "?returnFaceAttributes=age,gender,headPose,smile,facialHair," +
        "glasses,emotion,hair,makeup,occlusion,accessories,blur,exposure,noise"
    const uri = uriBase + params
    const imageUrlEnc = "{\"url\":\"" + imageUrl + "\"}"

    reader := strings.NewReader(imageUrlEnc)

    // Create the Http client
    client := &http.Client{
        Timeout: time.Second * 2,
    }

    // Create the Post request, passing the image URL in the request body
    req, err := http.NewRequest("POST", uri, reader)
    if err != nil {
        panic(err)
    }

    // Add headers
    req.Header.Add("Content-Type", "application/json")
    req.Header.Add("Ocp-Apim-Subscription-Key", subscriptionKey)

    // Send the request and retrieve the response
    resp, err := client.Do(req)
    if err != nil {
        panic(err)
    }

    defer resp.Body.Close()

    // Read the response body.
    // Note, data is a byte array
    data, err := ioutil.ReadAll(resp.Body)
    if err != nil {
        panic(err)
    }

    // Parse the Json data
    var f interface{}
    json.Unmarshal(data, &f)

    // Format and display the Json result
    jsonFormatted, _ := json.MarshalIndent(f, "", "  ")
    fmt.Println(string(jsonFormatted))
}
```

Face - Detect response

A successful response is returned in JSON, for example:

```
[
  {
    "faceId": "ae8952c1-7b5e-4a5a-a330-a6aa351262c9",
    "faceRectangle": {
      "top": 621,
      "left": 616,
      "width": 195,
      "height": 195
    },
    "faceAttributes": {
      "smile": 0,
      "headPose": {
        "pitch": 0,
        "roll": 6.8,
        "yaw": 3.7
      },
      "gender": "male",
        "age": 37,
        "facialHair": {
          "moustache": 0.4,
          "beard": 0.4,
          "sideburns": 0.1
        },
        "glasses": "NoGlasses",
        "emotion": {
          "anger": 0,
          "contempt": 0,
          "disgust": 0,
          "fear": 0,
          "happiness": 0,
          "neutral": 0.999,
          "sadness": 0.001,
          "surprise": 0
        },
        "blur": {
          "blurLevel": "high",
          "value": 0.89
        },
        "exposure": {
          "exposureLevel": "goodExposure",
          "value": 0.51
        },
        "noise": {
          "noiseLevel": "medium",
          "value": 0.59
        },
        "makeup": {
          "eyeMakeup": true,
          "lipMakeup": false
        },
        "accessories": [],
        "occlusion": {
          "foreheadOccluded": false,
          "eyeOccluded": false,
          "mouthOccluded": false
        },
        "hair": {
          "bald": 0.04,
          "invisible": false,
          "hairColor": [
            {
              "color": "black",
              "confidence": 0.98
            },
            {
              "color": "brown",
              "confidence": 0.87
            },
            {
              "color": "gray",
              "confidence": 0.85
            },
            {
              "color": "other",
              "confidence": 0.25
            },
            {
              "color": "blond",
              "confidence": 0.07
            },
            {
              "color": "red",
              "confidence": 0.02
            }
          ]
        }
```

```
    }
  },
  {
    "faceId": "b1bb3cbe-5a73-4f8d-96c8-836a5aca9415",
    "faceRectangle": {
      "top": 693,
      "left": 1503,
      "width": 180,
      "height": 180
    },
    "faceAttributes": {
      "smile": 0.003,
      "headPose": {
        "pitch": 0,
        "roll": 2,
        "yaw": -2.2
      },
      "gender": "female",
      "age": 56,
      "facialHair": {
        "moustache": 0,
        "beard": 0,
        "sideburns": 0
      },
      "glasses": "NoGlasses",
      "emotion": {
        "anger": 0,
        "contempt": 0.001,
        "disgust": 0,
        "fear": 0,
        "happiness": 0.003,
        "neutral": 0.984,
        "sadness": 0.011,
        "surprise": 0
      },
      "blur": {
        "blurLevel": "high",
        "value": 0.83
      },
      "exposure": {
        "exposureLevel": "goodExposure",
        "value": 0.41
      },
      "noise": {
        "noiseLevel": "high",
        "value": 0.76
      },
      "makeup": {
        "eyeMakeup": false,
        "lipMakeup": false
      },
      "accessories": [],
      "occlusion": {
        "foreheadOccluded": false,
        "eyeOccluded": false,
        "mouthOccluded": false
      },
      "hair": {
        "bald": 0.06,
        "invisible": false,
        "hairColor": [
          {
            "color": "black",
            "confidence": 0.99
          },
          {
            "color": "gray",
            "confidence": 0.89
          },
          {
            "color": "other",
            "confidence": 0.64
          },
          {
            "color": "brown",
            "confidence": 0.34
          },
          {
            "color": "blond",
            "confidence": 0.07
          },
          {
            "color": "red",
            "confidence": 0.03
          }
        ]
      }
    }
  }
]
```

Next steps

Explore the Face APIs used to detect human faces in an image, demarcate the faces with rectangles, and return attributes such as age and gender.

Face APIs

Quickstart: Detect faces in an image using the REST API and Java

10/23/2018 • 3 minutes to read • Edit Online

In this quickstart, you detect human faces in an image using the Face API.

Prerequisites

You need a subscription key to run the sample. You can get free trial subscription keys from Try Cognitive Services.

Detect faces in an image

Use the Face - Detect method to detect faces in an image and return face attributes including:

- Face ID: Unique ID used in several Face API scenarios.
- Face Rectangle: The left, top, width, and height indicating the location of the face in the image.
- Landmarks: An array of 27-point face landmarks pointing to the important positions of face components.
- Facial attributes including age, gender, smile intensity, head pose, and facial hair.

To run the sample, do the following steps:

1. Create a new command-line app in your favorite Java IDE.
2. Replace the Main class with the following code (keep any `package` statements).
3. Replace `<Subscription Key>` with your valid subscription key.
4. Change the `uriBase` value to use the location where you obtained your subscription keys, if necessary.
5. Download these global libraries from the Maven Repository to the `lib` directory in your project:
    - `org.apache.httpcomponents:httpclient:4.2.4`
    - `org.json:json:20170516`
6. Run 'Main'.

Face - Detect request

```
// This sample uses Apache HttpComponents:
// http://hc.apache.org/httpcomponents-core-ga/httpcore/apidocs/
// https://hc.apache.org/httpcomponents-client-ga/httpclient/apidocs/

import java.net.URI;
import org.apache.http.HttpEntity;
import org.apache.http.HttpResponse;
import org.apache.http.client.HttpClient;
import org.apache.http.client.methods.HttpPost;
import org.apache.http.entity.StringEntity;
import org.apache.http.client.utils.URIBuilder;
import org.apache.http.impl.client.DefaultHttpClient;
import org.apache.http.util.EntityUtils;
import org.json.JSONArray;
import org.json.JSONObject;

public class Main
{
    // Replace <Subscription Key> with your valid subscription key.
    private static final String subscriptionKey = "<Subscription Key>";

    // NOTE: You must use the same region in your REST call as you used to
    // obtain your subscription keys. For example, if you obtained your
    // subscription keys from westus, replace "westcentralus" in the URL
    // below with "westus".
    //
    // Free trial subscription keys are generated in the westcentralus region. If you
    // use a free trial subscription key, you shouldn't need to change this region.
    private static final String uriBase =
        "https://westcentralus.api.cognitive.microsoft.com/face/v1.0/detect";

    private static final String imageWithFaces =
        "{\"url\":\"https://upload.wikimedia.org/wikipedia/commons/c/c3/RH_Louise_Lillian_Gish.jpg\"}";

    private static final String faceAttributes =

"age,gender,headPose,smile,facialHair,glasses,emotion,hair,makeup,occlusion,accessories,blur,exposure,noise";

    public static void main(String[] args)
    {
        HttpClient httpclient = new DefaultHttpClient();

        try
        {
            URIBuilder builder = new URIBuilder(uriBase);

            // Request parameters. All of them are optional.
            builder.setParameter("returnFaceId", "true");
            builder.setParameter("returnFaceLandmarks", "false");
            builder.setParameter("returnFaceAttributes", faceAttributes);

            // Prepare the URI for the REST API call.
            URI uri = builder.build();
            HttpPost request = new HttpPost(uri);

            // Request headers.
            request.setHeader("Content-Type", "application/json");
            request.setHeader("Ocp-Apim-Subscription-Key", subscriptionKey);

            // Request body.
            StringEntity reqEntity = new StringEntity(imageWithFaces);
            request.setEntity(reqEntity);

            // Execute the REST API call and get the response entity.
            HttpResponse response = httpclient.execute(request);
            HttpEntity entity = response.getEntity();

            if (entity != null)
            {
                // Format and display the JSON response.
                System.out.println("REST Response:\n");

                String jsonString = EntityUtils.toString(entity).trim();
                if (jsonString.charAt(0) == '[') {
                    JSONArray jsonArray = new JSONArray(jsonString);
                    System.out.println(jsonArray.toString(2));
                }
                else if (jsonString.charAt(0) == '{') {
                    JSONObject jsonObject = new JSONObject(jsonString);
                    System.out.println(jsonObject.toString(2));
                } else {
                    System.out.println(jsonString);
                }
            }
        }
        catch (Exception e)
        {
            // Display error message.
            System.out.println(e.getMessage());
        }
    }
    }
}
```

Face - Detect response

A successful response is returned in JSON.

```
[{
  "faceRectangle": {
    "top": 131,
    "left": 177,
    "width": 162,
    "height": 162
  },
  "faceAttributes": {
    "makeup": {
      "eyeMakeup": true,
      "lipMakeup": true
    },
    "facialHair": {
      "sideburns": 0,
      "beard": 0,
      "moustache": 0
    },
    "gender": "female",
    "accessories": [],
    "blur": {
      "blurLevel": "low",
      "value": 0.06
    },
    "headPose": {
      "roll": 0.1,
      "pitch": 0,
      "yaw": -32.9
    },
    "smile": 0,
    "glasses": "NoGlasses",
    "hair": {
      "bald": 0,
      "invisible": false,
      "hairColor": [
        {
          "color": "brown",
          "confidence": 1
        },
        {
          "color": "black",
          "confidence": 0.87
        },
        {
          "color": "other",
          "confidence": 0.51
        },
        {
          "color": "blond",
          "confidence": 0.08
        },
        {
          "color": "red",
          "confidence": 0.08
        },
        {
          "color": "gray",
          "confidence": 0.02
        }
      ]
    },
    "emotion": {
      "contempt": 0,
      "surprise": 0.005,
      "happiness": 0,
      "neutral": 0.986,
      "sadness": 0.009,
      "disgust": 0,
      "anger": 0,
      "fear": 0
    },
    "exposure": {
      "value": 0.67,
      "exposureLevel": "goodExposure"
    },
    "occlusion": {
      "eyeOccluded": false,
      "mouthOccluded": false,
      "foreheadOccluded": false
    },
    "noise": {
      "noiseLevel": "low",
      "value": 0
    },
    "age": 22.9
  },
  "faceId": "49d55c17-e018-4a42-ba7b-8cbbdfae7c6f"
}]
```

Next steps

Learn how to create an Android application that uses the Face service to detect faces in an image. The application displays the image with a frame drawn around each face.

Tutorial: Getting Started with Face API in Android

Quickstart: Detect faces in an image using the REST API and JavaScript

10/23/2018 • 3 minutes to read • Edit Online

In this quickstart, you detect faces in an image using the Face API.

Prerequisites

You need a subscription key to run the sample. You can get free trial subscription keys from Try Cognitive Services.

Detect faces in an image

Use the Face - Detect method to detect faces in an image and return face attributes including:

- Face ID: Unique ID used in several Face API scenarios.
- Face Rectangle: The left, top, width, and height indicating the location of the face in the image.
- Landmarks: An array of 27-point face landmarks pointing to the important positions of face components.
- Facial attributes including age, gender, smile intensity, head pose, and facial hair.

To run the sample, do the following steps:

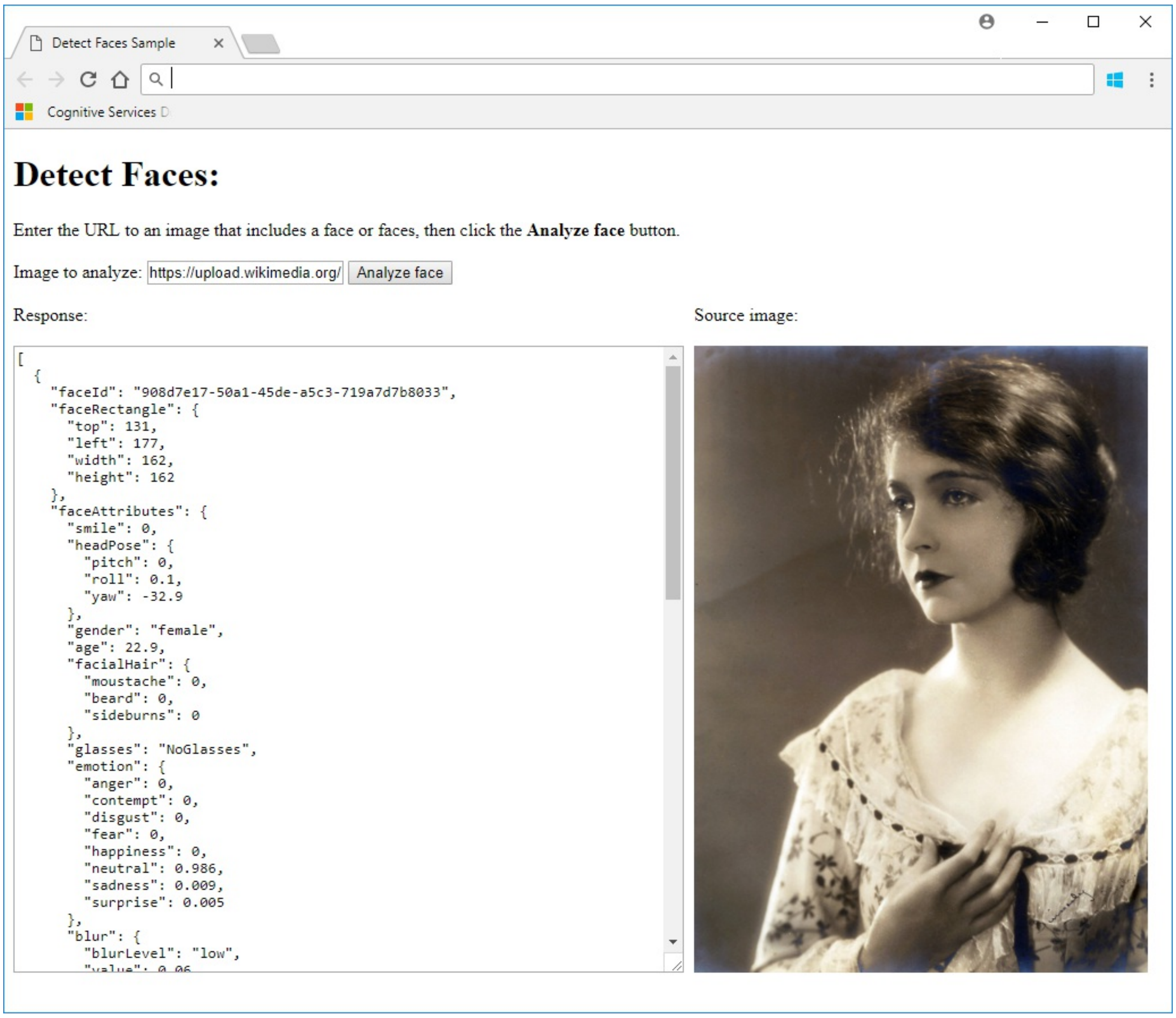
1. Copy the following and save it to a file such as `detectFaces.html`.
2. Replace `<Subscription Key>` with your valid subscription key.
3. Change the `uriBase` value to use the location where you obtained your subscription keys, if necessary.
4. Drag-and-drop the file into your browser.
5. Click the `Analyze faces` button.

Face - Detect request

```
<!DOCTYPE html>
<html>
<head>
    <title>Detect Faces Sample</title>
    <script src="http://ajax.googleapis.com/ajax/libs/jquery/1.9.0/jquery.min.js"></script>
</head>
<body>

<script type="text/javascript">
    function processImage() {
        // Replace <Subscription Key> with your valid subscription key.
        var subscriptionKey = "<Subscription Key>";

        // NOTE: You must use the same region in your REST call as you used to
        // obtain your subscription keys. For example, if you obtained your
        // subscription keys from westus, replace "westcentralus" in the URL
        // below with "westus".
        //
        // Free trial subscription keys are generated in the westcentralus region.
        // If you use a free trial subscription key, you shouldn't need to change
        // this region.
        var uriBase =
            "https://westcentralus.api.cognitive.microsoft.com/face/v1.0/detect";

        // Request parameters.
        var params = {
            "returnFaceId": "true",
            "returnFaceLandmarks": "false",
            "returnFaceAttributes":
                "age,gender,headPose,smile,facialHair,glasses,emotion," +
                "hair,makeup,occlusion,accessories,blur,exposure,noise"
        };

        // Display the image.
        var sourceImageUrl = document.getElementById("inputImage").value;
        document.querySelector("#sourceImage").src = sourceImageUrl;

        // Perform the REST API call.
        $.ajax({
            url: uriBase + "?" + $.param(params),

            // Request headers.
            beforeSend: function(xhrObj){
                xhrObj.setRequestHeader("Content-Type","application/json");
                xhrObj.setRequestHeader("Ocp-Apim-Subscription-Key", subscriptionKey);
            },

            type: "POST",

            // Request body.
            data: '{"url": ' + '"' + sourceImageUrl + '"}',
        })

        .done(function(data) {
            // Show formatted JSON on webpage.
            $("#responseTextArea").val(JSON.stringify(data, null, 2));
        })

        .fail(function(jqXHR, textStatus, errorThrown) {
            // Display error message.
            var errorString = (errorThrown === "") ?
                "Error. " : errorThrown + " (" + jqXHR.status + "): ";
            errorString += (jqXHR.responseText === "") ?
                "" : (jQuery.parseJSON(jqXHR.responseText).message) ?
                    jQuery.parseJSON(jqXHR.responseText).message :
                        jQuery.parseJSON(jqXHR.responseText).error.message;
            alert(errorString);
        });
    };
</script>

<h1>Detect Faces:</h1>
Enter the URL to an image that includes a face or faces, then click
the <strong>Analyze face</strong> button.<br><br>

Image to analyze: <input type="text" name="inputImage" id="inputImage"
value="https://upload.wikimedia.org/wikipedia/commons/c/c3/RH_Louise_Lillian_Gish.jpg" />

<button onclick="processImage()">Analyze face</button><br><br>

<div id="wrapper" style="width:1020px; display:table;">
    <div id="jsonOutput" style="width:600px; display:table-cell;">
        Response:<br><br>

        <textarea id="responseTextArea" class="UIInput"
                  style="width:580px; height:400px;"></textarea>
    </div>
    <div id="imageDiv" style="width:420px; display:table-cell;">
        Source image:<br><br>

        <img id="sourceImage" width="400" />
    </div>
</div>
</body>
</html>
```

Face - Detect response

A successful response is returned in JSON.



Detect Faces:

Enter the URL to an image that includes a face or faces, then click the **Analyze face** button.

Image to analyze: https://upload.wikimedia.org/ Analyze face

Response:

```
[
  {
    "faceId": "908d7e17-50a1-45de-a5c3-719a7d7b8033",
    "faceRectangle": {
      "top": 131,
      "left": 177,
      "width": 162,
      "height": 162
    },
    "faceAttributes": {
      "smile": 0,
      "headPose": {
        "pitch": 0,
        "roll": 0.1,
        "yaw": -32.9
      },
      "gender": "female",
      "age": 22.9,
      "facialHair": {
        "moustache": 0,
        "beard": 0,
        "sideburns": 0
      },
      "glasses": "NoGlasses",
      "emotion": {
        "anger": 0,
        "contempt": 0,
        "disgust": 0,
        "fear": 0,
        "happiness": 0,
        "neutral": 0.986,
        "sadness": 0.009,
        "surprise": 0.005
      },
      "blur": {
        "blurLevel": "low",
        "value": 0.06
```

Source image:

Following is an example of a successful response:

```
[
  {
    "faceId": "49d55c17-e018-4a42-ba7b-8cbbdfae7c6f",
    "faceRectangle": {
      "top": 131,
      "left": 177,
      "width": 162,
      "height": 162
    },
    "faceAttributes": {
      "smile": 0,
      "headPose": {
        "pitch": 0,
        "roll": 0.1,
        "yaw": -32.9
      },
      "gender": "female",
      "age": 22.9,
      "facialHair": {
        "moustache": 0,
        "beard": 0,
        "sideburns": 0
      },
      "glasses": "NoGlasses",
      "emotion": {
        "anger": 0,
        "contempt": 0,
        "disgust": 0,
        "fear": 0,
        "happiness": 0,
        "neutral": 0.986,
        "sadness": 0.009,
        "surprise": 0.005
      },
      "blur": {
        "blurLevel": "low",
        "value": 0.06
      },
      "exposure": {
        "exposureLevel": "goodExposure",
        "value": 0.67
      },
      "noise": {
        "noiseLevel": "low",
        "value": 0
      },
      "makeup": {
        "eyeMakeup": true,
        "lipMakeup": true
      },
      "accessories": [],
      "occlusion": {
        "foreheadOccluded": false,
        "eyeOccluded": false,
        "mouthOccluded": false
      },
      "hair": {
        "bald": 0,
        "invisible": false,
        "hairColor": [
          {
            "color": "brown",
            "confidence": 1
          },
          {
            "color": "black",
            "confidence": 0.87
          },
          {
            "color": "other",
            "confidence": 0.51
          },
          {
            "color": "blond",
            "confidence": 0.08
          },
          {
            "color": "red",
            "confidence": 0.08
          },
          {
            "color": "gray",
            "confidence": 0.02
          }
        ]
      }
    }
  }
]
```

Next steps

Explore the Face APIs used to detect human faces in an image, demarcate the faces with rectangles, and return attributes such as age and gender.

Face APIs

Quickstart: Detect faces in an image using the REST API and Node.js

10/23/2018 • 3 minutes to read • Edit Online

In this quickstart, you detect human faces in an image using the Face API.

Prerequisites

You need a subscription key to run the sample. You can get free trial subscription keys from Try Cognitive Services.

Face - Detect request

Use the Face - Detect method to detect faces in an image and return face attributes including:

- Face ID: Unique ID used in several Face API scenarios.
- Face Rectangle: The left, top, width, and height indicating the location of the face in the image.
- Landmarks: An array of 27-point face landmarks pointing to the important positions of face components.
- Facial attributes including age, gender, smile intensity, head pose, and facial hair.

To run the sample, do the following steps:

1. Copy the following code into an editor.
2. Replace `<Subscription Key>` with your valid subscription key.
3. Change the `uriBase` value to the location where you obtained your subscription keys, if necessary.
4. Optionally, set `imageUri` to the image you want to analyze.
5. Save the file with an `.js` extension.
6. Open the Node.js command prompt and run the file, for example: `node myfile.js`.

```
'use strict';

const request = require('request');

// Replace <Subscription Key> with your valid subscription key.
const subscriptionKey = '<Subscription Key>';

// You must use the same location in your REST call as you used to get your
// subscription keys. For example, if you got your subscription keys from
// westus, replace "westcentralus" in the URL below with "westus".
const uriBase = 'https://westcentralus.api.cognitive.microsoft.com/face/v1.0/detect';

const imageUrl =
    'https://upload.wikimedia.org/wikipedia/commons/3/37/Dagestani_man_and_woman.jpg';

// Request parameters.
const params = {
    'returnFaceId': 'true',
    'returnFaceLandmarks': 'false',
    'returnFaceAttributes': 'age,gender,headPose,smile,facialHair,glasses,' +
        'emotion,hair,makeup,occlusion,accessories,blur,exposure,noise'
};

const options = {
    uri: uriBase,
    qs: params,
    body: '{"url": ' + '"' + imageUrl + '"}',
    headers: {
        'Content-Type': 'application/json',
        'Ocp-Apim-Subscription-Key' : subscriptionKey
    }
};

request.post(options, (error, response, body) => {
  if (error) {
    console.log('Error: ', error);
    return;
  }
  let jsonResponse = JSON.stringify(JSON.parse(body), null, '  ');
  console.log('JSON Response\n');
  console.log(jsonResponse);
});
```

Face - Detect response

A successful response is returned in JSON, for example:

```
[
  {
    "faceId": "ae8952c1-7b5e-4a5a-a330-a6aa351262c9",
    "faceRectangle": {
      "top": 621,
      "left": 616,
      "width": 195,
      "height": 195
    },
    "faceAttributes": {
      "smile": 0,
      "headPose": {
        "pitch": 0,
        "roll": 6.8,
        "yaw": 3.7
      },
      "gender": "male",
      "age": 37,
      "facialHair": {
        "moustache": 0.4,
        "beard": 0.4,
        "sideburns": 0.1
      },
      "glasses": "NoGlasses",
      "emotion": {
        "anger": 0,
        "contempt": 0,
        "disgust": 0,
        "fear": 0,
        "happiness": 0,
        "neutral": 0.999,
        "sadness": 0.001,
        "surprise": 0
      },
      "blur": {
        "blurLevel": "high",
        "value": 0.89
      },
      "exposure": {
        "exposureLevel": "goodExposure",
        "value": 0.51
      },
      "noise": {
        "noiseLevel": "medium",
        "value": 0.59
      },
      "makeup": {
        "eyeMakeup": true,
        "lipMakeup": false
      },
      "accessories": [],
      "occlusion": {
        "foreheadOccluded": false,
        "eyeOccluded": false,
        "mouthOccluded": false
      },
      "hair": {
        "bald": 0.04,
        "invisible": false,
        "hairColor": [
          {
            "color": "black",
            "confidence": 0.98
          },
          {
            "color": "brown",
            "confidence": 0.87
          },
          {
            "color": "gray",
            "confidence": 0.85
          },
          {
            "color": "other",
            "confidence": 0.25
          },
          {
            "color": "blond",
            "confidence": 0.07
          },
          {
            "color": "red",
            "confidence": 0.02
          }
        ]
      }
    }
    },
    {
      "faceId": "b1bb3cbe-5a73-4f8d-96c8-836a5aca9415",
      "faceRectangle": {
        "top": 693,
        "left": 1503,
        "width": 180,
        "height": 180
      },
      "faceAttributes": {
        "smile": 0.003,
        "headPose": {
          "pitch": 0,
          "roll": 2,
          "yaw": -2.2
        },
        "gender": "female",
        "age": 56,
        "facialHair": {
          "moustache": 0,
          "beard": 0,
          "sideburns": 0
        },
        "glasses": "NoGlasses",
        "emotion": {
          "anger": 0,
          "contempt": 0.001,
          "disgust": 0,
          "fear": 0,
          "happiness": 0.003,
          "neutral": 0.984,
          "sadness": 0.011,
          "surprise": 0
        },
        "blur": {
          "blurLevel": "high",
          "value": 0.83
        },
        "exposure": {
          "exposureLevel": "goodExposure",
          "value": 0.41
        },
        "noise": {
          "noiseLevel": "high",
          "value": 0.76
        },
        "makeup": {
          "eyeMakeup": false,
          "lipMakeup": false
        },
        "accessories": [],
        "occlusion": {
          "foreheadOccluded": false,
          "eyeOccluded": false,
          "mouthOccluded": false
        },
        "hair": {
          "bald": 0.06,
          "invisible": false,
          "hairColor": [
            {
              "color": "black",
              "confidence": 0.99
            },
            {
              "color": "gray",
              "confidence": 0.89
            },
            {
```

```
          {
            "color": "other",
            "confidence": 0.64
          },
          {
            "color": "brown",
            "confidence": 0.34
          },
          {
            "color": "blond",
            "confidence": 0.07
          },
          {
            "color": "red",
            "confidence": 0.03
          }
        ]
      }
    }
  }
]
```

Next steps

Explore the Face APIs used to detect human faces in an image, demarcate the faces with rectangles, and return attributes such as age and gender.

Face APIs

Quickstart: Detect faces in an image using the REST API and PHP

10/23/2018 • 3 minutes to read • Edit Online

In this quickstart, you detect human faces in an image using the Face API.

Prerequisites

You need a subscription key to run the sample. You can get free trial subscription keys from Try Cognitive Services.

Face - Detect request

Use the Face - Detect method to detect faces in an image and return face attributes including:

- Face ID: Unique ID used in several Face API scenarios.
- Face Rectangle: The left, top, width, and height indicating the location of the face in the image.
- Landmarks: An array of 27-point face landmarks pointing to the important positions of face components.
- Facial attributes including age, gender, smile intensity, head pose, and facial hair.

To run the sample, do the following steps:

1. Copy the following code into an editor.
2. Replace `<Subscription Key>` with your valid subscription key.
3. Change `uriBase` to use the location where you obtained your subscription keys, if necessary.
4. Optionally, set `imageUrl` to the image you want to analyze.
5. Save the file with a `.php` extension.
6. Open the file in a browser window with PHP support.

```
<html>
<head>
    <title>Face Detect Sample</title>
</head>
<body>
<?php
// Replace <Subscription Key> with a valid subscription key.
$ocpApimSubscriptionKey = '<Subscription Key>';

// You must use the same location in your REST call as you used to obtain
// your subscription keys. For example, if you obtained your subscription keys
// from westus, replace "westcentralus" in the URL below with "westus".
$uriBase = 'https://westcentralus.api.cognitive.microsoft.com/face/v1.0/';

$imageUrl =
    'https://upload.wikimedia.org/wikipedia/commons/3/37/Dagestani_man_and_woman.jpg';

// This sample uses the PHP5 HTTP_Request2 package
// (http://pear.php.net/package/HTTP_Request2).
require_once 'HTTP/Request2.php';

$request = new Http_Request2($uriBase . '/detect');
$url = $request->getUrl();

$headers = array(
    // Request headers
    'Content-Type' => 'application/json',
    'Ocp-Apim-Subscription-Key' => $ocpApimSubscriptionKey
);
$request->setHeader($headers);

$parameters = array(
    // Request parameters
    'returnFaceId' => 'true',
    'returnFaceLandmarks' => 'false',
    'returnFaceAttributes' => 'age,gender,headPose,smile,facialHair,glasses,' .
        'emotion,hair,makeup,occlusion,accessories,blur,exposure,noise');
$url->setQueryVariables($parameters);

$request->setMethod(HTTP_Request2::METHOD_POST);

// Request body parameters
$body = json_encode(array('url' => $imageUrl));

// Request body
$request->setBody($body);

try
{
    $response = $request->send();
    echo "<pre>" .
        json_encode(json_decode($response->getBody()), JSON_PRETTY_PRINT) . "</pre>";
}
catch (HttpException $ex)
{
    echo "<pre>" . $ex . "</pre>";
}
?>
</body>
</html>
```

Face - Detect response

A successful response is returned in JSON, for example:

```
[
    {
        "faceId": "e93e0db1-036e-4819-b5b6-4f39e0f73509",
        "faceRectangle": {
            "top": 621,
            "left": 616,
            "width": 195,
            "height": 195
        },
        "faceAttributes": {
            "smile": 0,
            "headPose": {
                "pitch": 0,
                "roll": 6.8,
                "yaw": 3.7
            },
            "gender": "male",
            "age": 37,
            "facialHair": {
                "moustache": 0.4,
                "beard": 0.4,
                "sideburns": 0.1
            },
            "glasses": "NoGlasses",
            "emotion": {
                "anger": 0,
                "contempt": 0,
                "disgust": 0,
                "fear": 0,
                "happiness": 0,
                "neutral": 0.999,
                "sadness": 0.001,
                "surprise": 0
            },
            "blur": {
                "blurLevel": "high",
                "value": 0.89
            },
            "exposure": {
                "exposureLevel": "goodExposure",
                "value": 0.51
            },
            "noise": {
                "noiseLevel": "medium",
                "value": 0.59
            },
            "makeup": {
                "eyeMakeup": true,
                "lipMakeup": false
            },
            "accessories": [],
            "occlusion": {
                "foreheadOccluded": false,
                "eyeOccluded": false,
                "mouthOccluded": false
            },
            "hair": {
                "bald": 0.04,
                "invisible": false,
                "hairColor": [
                    {
                        "color": "black",
                        "confidence": 0.98
                    },
                    {
                        "color": "brown",
                        "confidence": 0.87
                    },
                    {
```

```
                    "color": "gray",
                    "confidence": 0.85
                },
                {
                    "color": "other",
                    "confidence": 0.25
                },
                {
                    "color": "blond",
                    "confidence": 0.07
                },
                {
                    "color": "red",
                    "confidence": 0.02
                }
            ]
        }
    }
},
{
    "faceId": "37c7c4bc-fda3-4d8d-94e8-b85b8deaf878",
    "faceRectangle": {
        "top": 693,
        "left": 1503,
        "width": 180,
        "height": 180
    },
    "faceAttributes": {
        "smile": 0.003,
        "headPose": {
            "pitch": 0,
            "roll": 2,
            "yaw": -2.2
        },
        "gender": "female",
        "age": 56,
        "facialHair": {
            "moustache": 0,
            "beard": 0,
            "sideburns": 0
        },
        "glasses": "NoGlasses",
        "emotion": {
            "anger": 0,
            "contempt": 0.001,
            "disgust": 0,
            "fear": 0,
            "happiness": 0.003,
            "neutral": 0.984,
            "sadness": 0.011,
            "surprise": 0
        },
        "blur": {
            "blurLevel": "high",
            "value": 0.83
        },
        "exposure": {
            "exposureLevel": "goodExposure",
            "value": 0.41
        },
        "noise": {
            "noiseLevel": "high",
            "value": 0.76
        },
        "makeup": {
            "eyeMakeup": false,
            "lipMakeup": false
        },
        "accessories": [],
            "accessories": [],
            "occlusion": {
                "foreheadOccluded": false,
                "eyeOccluded": false,
                "mouthOccluded": false
            },
            "hair": {
                "bald": 0.06,
                "invisible": false,
                "hairColor": [
                    {
                        "color": "black",
                        "confidence": 0.99
                    },
                    {
                        "color": "gray",
                        "confidence": 0.89
                    },
                    {
                        "color": "other",
                        "confidence": 0.64
                    },
                    {
                        "color": "brown",
                        "confidence": 0.34
                    },
                    {
                        "color": "blond",
                        "confidence": 0.07
                    },
                    {
                        "color": "red",
                        "confidence": 0.03
                    }
                ]
            }
        }
    }
]
```

Next steps

Explore the Face APIs used to detect human faces in an image, demarcate the faces with rectangles, and return attributes such as age and gender.

Face APIs

Quickstart: Detect faces in an image using the REST API and Python

10/23/2018 • 4 minutes to read • Edit Online

In this quickstart, you detect human faces in a remote image using the Face service. The detected faces are demarcated with rectangles and superimposed with the gender and age of each person. To use a local image, see the syntax in Computer Vision: Analyze a local image with Python.

You can run this quickstart as a Jupyter notebook on MyBinder. To launch Binder, select the following button:

launch binder

Prerequisites

You need a subscription key to run the sample. You can get free trial subscription keys from Try Cognitive Services.

Detect faces in an image

Use the Face - Detect method to detect faces in an image and return face attributes including:

- Face ID: Unique ID used in several Face API scenarios.
- Face Rectangle: The left, top, width, and height indicating the location of the face in the image.
- Landmarks: An array of 27-point face landmarks pointing to the important positions of face components.
- Facial attributes including age, gender, smile intensity, head pose, and facial hair.

To run the sample, do the following steps:

1. Copy the following code to a new Python script file.
2. Replace `<Subscription Key>` with your valid subscription key.
3. Change the `face_api_url` value to the location where you obtained your subscription keys, if necessary.
4. Optionally, change the `image_url` value to another image.
5. Run the script.

Face - Detect request

The following code uses the Python `requests` library to call the Face Detect API. It returns the results as a JSON object. The API key is passed in via the `headers` dictionary. The types of features to recognize is passed in via the `params` dictionary.

```python
import requests
# If you are using a Jupyter notebook, uncomment the following line.
#%matplotlib inline
import matplotlib.pyplot as plt
from PIL import Image
from matplotlib import patches
from io import BytesIO

# Replace <Subscription Key> with your valid subscription key.
subscription_key = "<Subscription Key>"
assert subscription_key

# You must use the same region in your REST call as you used to get your
# subscription keys. For example, if you got your subscription keys from
# westus, replace "westcentralus" in the URI below with "westus".
#
# Free trial subscription keys are generated in the westcentralus region.
# If you use a free trial subscription key, you shouldn't need to change
# this region.
face_api_url = 'https://westcentralus.api.cognitive.microsoft.com/face/v1.0/detect'

# Set image_url to the URL of an image that you want to analyze.
image_url = 'https://how-old.net/Images/faces2/main007.jpg'

headers = {'Ocp-Apim-Subscription-Key': subscription_key}
params = {
    'returnFaceId': 'true',
    'returnFaceLandmarks': 'false',
    'returnFaceAttributes': 'age,gender,headPose,smile,facialHair,glasses,' +
    'emotion,hair,makeup,occlusion,accessories,blur,exposure,noise'
}
data = {'url': image_url}
response = requests.post(face_api_url, params=params, headers=headers, json=data)
faces = response.json()

# Display the original image and overlay it with the face information.
image = Image.open(BytesIO(requests.get(image_url).content))
plt.figure(figsize=(8, 8))
ax = plt.imshow(image, alpha=0.6)
for face in faces:
    fr = face["faceRectangle"]
    fa = face["faceAttributes"]
    origin = (fr["left"], fr["top"])
    p = patches.Rectangle(
        origin, fr["width"], fr["height"], fill=False, linewidth=2, color='b')
    ax.axes.add_patch(p)
    plt.text(origin[0], origin[1], "%s, %d"%(fa["gender"].capitalize(), fa["age"]),
             fontsize=20, weight="bold", va="bottom")
_ = plt.axis("off")
```

Face - Detect response

A successful response is returned in JSON, for example:

```json
[
  {
    "faceId": "35102aa8-4263-4139-bfd6-185bb0f52d88",
    "faceRectangle": {
      "top": 208,
      "left": 228,
      "width": 91,
      "height": 91
    },
    "faceAttributes": {
      "smile": 1,
      "headPose": {
```

```
      "pitch": 0,
      "roll": 4.3,
      "yaw": -0.3
    },
    "gender": "female",
    "age": 27,
    "facialHair": {
      "moustache": 0,
      "beard": 0,
      "sideburns": 0
    },
    "glasses": "NoGlasses",
    "emotion": {
      "anger": 0,
      "contempt": 0,
      "disgust": 0,
      "fear": 0,
      "happiness": 1,
      "neutral": 0,
      "sadness": 0,
      "surprise": 0
    },
    "blur": {
      "blurLevel": "low",
      "value": 0
    },
    "exposure": {
      "exposureLevel": "goodExposure",
      "value": 0.65
    },
    "noise": {
      "noiseLevel": "low",
      "value": 0
    },
    "makeup": {
      "eyeMakeup": true,
      "lipMakeup": true
    },
    "accessories": [],
    "occlusion": {
      "foreheadOccluded": false,
      "eyeOccluded": false,
      "mouthOccluded": false
    },
    "hair": {
      "bald": 0.06,
      "invisible": false,
      "hairColor": [
        {
          "color": "brown",
          "confidence": 1
        },
        {
          "color": "blond",
          "confidence": 0.5
        },
        {
          "color": "black",
          "confidence": 0.34
        },
        {
          "color": "red",
          "confidence": 0.32
        },
        {
          "color": "gray",
          "confidence": 0.14
        },
        {
          "color": "other",
          "confidence": 0.03
        }
      ]
    }
  }
},
{
  "faceId": "42502166-31bb-4ac8-81c0-a7adcb3b3e70",
  "faceRectangle": {
    "top": 109,
    "left": 125,
    "width": 79,
    "height": 79
  },
  "faceAttributes": {
    "smile": 1,
    "headPose": {
      "pitch": 0,
      "roll": 1.7,
      "yaw": 2.1
    },
    "gender": "male",
    "age": 32,
    "facialHair": {
      "moustache": 0.4,
      "beard": 0.4,
      "sideburns": 0.4
    },
    "glasses": "NoGlasses",
    "emotion": {
      "anger": 0,
      "contempt": 0,
      "disgust": 0,
      "fear": 0,
      "happiness": 1,
      "neutral": 0,
      "sadness": 0,
      "surprise": 0
    },
    "blur": {
      "blurLevel": "low",
      "value": 0.11
    },
    "exposure": {
      "exposureLevel": "goodExposure",
      "value": 0.74
    },
    "noise": {
      "noiseLevel": "low",
      "value": 0
    },
    "makeup": {
      "eyeMakeup": false,
      "lipMakeup": true
    },
    "accessories": [],
    "occlusion": {
      "foreheadOccluded": false,
      "eyeOccluded": false,
      "mouthOccluded": false
    },
    "hair": {
      "bald": 0.02,
      "invisible": false,
      "hairColor": [
        {
          "color": "brown",
          "confidence": 1
          },
          {
            "color": "blond",
            "confidence": 0.94
          },
          {
            "color": "red",
            "confidence": 0.76
          },
          {
            "color": "gray",
            "confidence": 0.2
          },
          {
            "color": "other",
            "confidence": 0.03
          },
          {
            "color": "black",
            "confidence": 0.01
          }
        ]
      }
    }
  }
]
```

Next steps

Explore the Face APIs used to detect human faces in an image, demarcate the faces with rectangles, and return attributes such as age and gender.

Face APIs

Quickstart: Detect faces in an image using the REST API and Ruby

10/23/2018 • 3 minutes to read • Edit Online

In this quickstart, you detect human faces in an image using the Face API.

Prerequisites

You need a subscription key to run the sample. You can get free trial subscription keys from Try Cognitive Services.

Face - Detect request

Use the Face - Detect method to detect faces in an image and return face attributes including:

- Face ID: Unique ID used in several Face API scenarios.
- Face Rectangle: The left, top, width, and height indicating the location of the face in the image.
- Landmarks: An array of 27-point face landmarks pointing to the important positions of face components.
- Facial attributes including age, gender, smile intensity, head pose, and facial hair.

To run the sample, do the following steps:

1. Copy the following code into an editor.
2. Replace `<Subscription Key>` with your valid subscription key.
3. Change the `uri` value to the location where you obtained your subscription keys, if necessary.
4. Optionally, set `imageUri` to the image you want to analyze.
5. Save the file with an `.rb` extension.
6. Open the Ruby Command Prompt and run the file, for example: `ruby myfile.rb`.

```
require 'net/http'

# You must use the same location in your REST call as you used to get your
# subscription keys. For example, if you got your subscription keys from  westus,
# replace "westcentralus" in the URL below with "westus".
uri = URI('https://westcentralus.api.cognitive.microsoft.com/face/v1.0/detect')
uri.query = URI.encode_www_form({
    # Request parameters
    'returnFaceId' => 'true',
    'returnFaceLandmarks' => 'false',
    'returnFaceAttributes' => 'age,gender,headPose,smile,facialHair,glasses,' +
        'emotion,hair,makeup,occlusion,accessories,blur,exposure,noise'
})

request = Net::HTTP::Post.new(uri.request_uri)

# Request headers
# Replace <Subscription Key> with your valid subscription key.
request['Ocp-Apim-Subscription-Key'] = '<Subscription Key>'
request['Content-Type'] = 'application/json'

imageUri = "https://upload.wikimedia.org/wikipedia/commons/3/37/Dagestani_man_and_woman.jpg"
request.body = "{\"url\": \"" + imageUri + "\"}"

response = Net::HTTP.start(uri.host, uri.port, :use_ssl => uri.scheme == 'https') do |http|
    http.request(request)
end

puts response.body
```

Face - Detect response

A successful response is returned in JSON, for example:

```
[
  {
    "faceId": "e93e0db1-036e-4819-b5b6-4f39e0f73509",
    "faceRectangle": {
      "top": 621,
      "left": 616,
      "width": 195,
      "height": 195
    },
    "faceAttributes": {
      "smile": 0,
      "headPose": {
        "pitch": 0,
        "roll": 6.8,
        "yaw": 3.7
      },
      "gender": "male",
      "age": 37,
      "facialHair": {
        "moustache": 0.4,
        "beard": 0.4,
        "sideburns": 0.1
      },
      "glasses": "NoGlasses",
      "emotion": {
        "anger": 0,
        "contempt": 0,
        "disgust": 0,
        "fear": 0,
        "happiness": 0,
        "neutral": 0.999,
        "sadness": 0.001,
        "surprise": 0
      },
      "blur": {
        "blurLevel": "high",
        "value": 0.89
      },
      "exposure": {
        "exposureLevel": "goodExposure",
        "value": 0.51
      },
      "noise": {
        "noiseLevel": "medium",
        "value": 0.59
      },
      "makeup": {
        "eyeMakeup": true,
        "lipMakeup": false
      },
      "accessories": [],
      "occlusion": {
        "foreheadOccluded": false,
        "eyeOccluded": false,
        "mouthOccluded": false
      },
      "hair": {
        "bald": 0.04,
        "invisible": false,
        "hairColor": [
          {
            "color": "black",
            "confidence": 0.98
          },
          {
            "color": "brown",
            "confidence": 0.87
          },
          {
            "color": "gray",
            "confidence": 0.85
          },
          {
            "color": "other",
            "confidence": 0.25
          },
          {
            "color": "blond",
            "confidence": 0.07
          },
          {
            "color": "red",
            "confidence": 0.02
          }
        ]
      }
    }
  },
  {
    "faceId": "37c7c4bc-fda3-4d8d-94e8-b85b8deaf878",
    "faceRectangle": {
      "top": 693,
      "left": 1503,
      "width": 180,
      "height": 180
    },
    "faceAttributes": {
      "smile": 0.003,
      "headPose": {
        "pitch": 0,
          "roll": 2,
          "yaw": -2.2
        },
        "gender": "female",
        "age": 56,
        "facialHair": {
          "moustache": 0,
          "beard": 0,
          "sideburns": 0
        },
        "glasses": "NoGlasses",
        "emotion": {
          "anger": 0,
          "contempt": 0.001,
          "disgust": 0,
          "fear": 0,
          "happiness": 0.003,
          "neutral": 0.984,
          "sadness": 0.011,
          "surprise": 0
        },
        "blur": {
          "blurLevel": "high",
          "value": 0.83
        },
        "exposure": {
          "exposureLevel": "goodExposure",
          "value": 0.41
        },
        "noise": {
          "noiseLevel": "high",
          "value": 0.76
        },
        "makeup": {
          "eyeMakeup": false,
          "lipMakeup": false
        },
        "accessories": [],
        "occlusion": {
          "foreheadOccluded": false,
          "eyeOccluded": false,
          "mouthOccluded": false
        },
        "hair": {
          "bald": 0.06,
          "invisible": false,
          "hairColor": [
            {
              "color": "black",
              "confidence": 0.99
            },
            {
              "color": "gray",
              "confidence": 0.89
            },
            {
              "color": "other",
              "confidence": 0.64
            },
            {
              "color": "brown",
              "confidence": 0.34
            },
            {
              "color": "blond",
              "confidence": 0.07
            },
            {
              "color": "red",
              "color": "red",
              "confidence": 0.03
            }
          ]
        }
      }
    }
  ]
```

Next steps

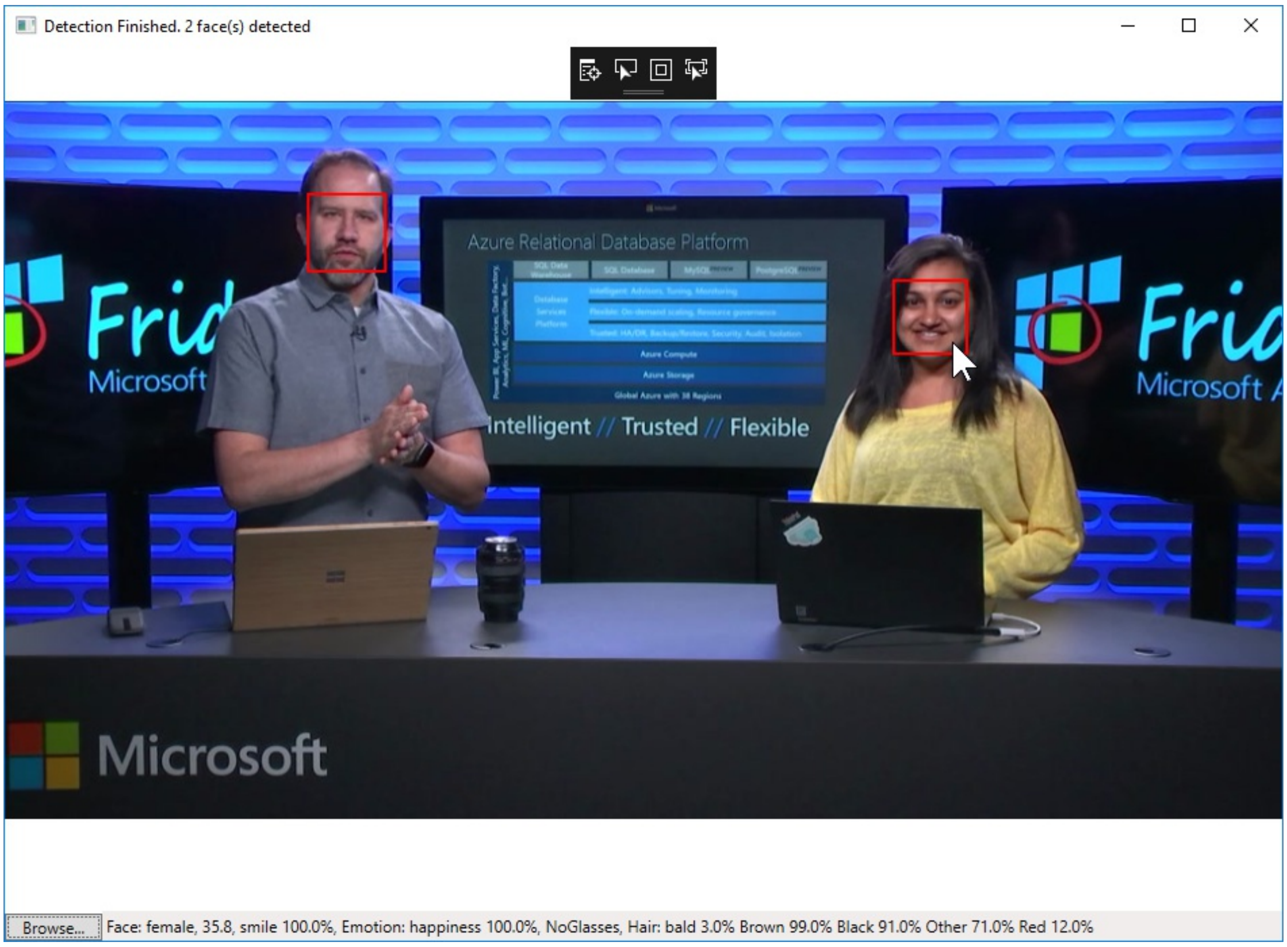
Explore the Face APIs used to detect human faces in an image, demarcate the faces with rectangles, and return attributes such as age and gender.

Face APIs

Tutorial: Create a WPF app to detect and frame faces in an image using the .NET SDK

10/23/2018 • 9 minutes to read • Edit Online

In this tutorial, you create a Windows Presentation Framework (WPF) application that uses the Face service through its .NET client library. The app detects faces in an image, draws a frame around each face, and displays a description of the face on the status bar. The complete sample code is available on GitHub at Detect and frame faces in an image on Windows.



Face: female, 35.8, smile 100.0%, Emotion: happiness 100.0%, NoGlasses, Hair: bald 3.0% Brown 99.0% Black 91.0% Other 71.0% Red 12.0%

This tutorial shows you how to:

- Create a WPF application
- Install the Face service client library
- Use the client library to detect faces in an image
- Draw a frame around each detected face
- Display a description of the face on the status bar

Prerequisites

- You need a subscription key to run the sample. You can get free trial subscription keys from Try Cognitive Services.
- Any edition of Visual Studio 2015 or 2017. For Visual Studio 2017, the .NET Desktop application development workload is required. This tutorial uses Visual Studio 2017 Community Edition.
- The Microsoft.Azure.CognitiveServices.Vision.Face 2.2.0-preview client library NuGet package. It isn't necessary

to download the package. Installation instructions are provided below.

Create the Visual Studio solution

Follow these steps to create a Windows WPF application project.

1. Open Visual Studio and from the **File** menu, click **New**, then **Project**.
   - In Visual Studio 2017, expand **Installed**, then **Other Languages**. Select **Visual C#**, then **WPF App (.NET Framework)**.
   - In Visual Studio 2015, expand **Installed**, then **Templates**. Select **Visual C#**, then **WPF Application**.
2. Name the application **FaceTutorial**, then click **OK**.

Install the Face service client library

Follow these instructions to install the client library.

1. From the **Tools** menu, select **NuGet Package Manager**, then **Package Manager Console**.
2. In the **Package Manager Console**, paste the following, then press **Enter**.

```
Install-Package Microsoft.Azure.CognitiveServices.Vision.Face -Version 2.2.0-preview
```

Add the initial code

MainWindow.xaml

Open *MainWindow.xaml* (tip: swap panes using the **up/down arrow icon**) and replace the contents with the following code. This xaml code is used to create the UI window. Note the event handlers, `FacePhoto_MouseMove` and `BrowseButton_Click`.

```
<Window x:Class="FaceTutorial.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="700" Width="960">
    <Grid x:Name="BackPanel">
        <Image x:Name="FacePhoto" Stretch="Uniform" Margin="0,0,0,50" MouseMove="FacePhoto_MouseMove" />
        <DockPanel DockPanel.Dock="Bottom">
            <Button x:Name="BrowseButton" Width="72" Height="20" VerticalAlignment="Bottom"
HorizontalAlignment="Left"
                    Content="Browse..."
                    Click="BrowseButton_Click" />
            <StatusBar VerticalAlignment="Bottom">
                <StatusBarItem>
                    <TextBlock Name="faceDescriptionStatusBar" />
                </StatusBarItem>
            </StatusBar>
        </DockPanel>
    </Grid>
</Window>
```

MainWindow.xaml.cs

Expand *MainWindow.xaml*, then open *MainWindow.xaml.cs*, and replace the contents with the following code. Ignore the squiggly red underlines; they'll disappear after the first build.

The first two lines import the client library namespaces. Next, the `FaceClient` is created, passing in the subscription key, while the Azure region is set in the `MainWindow` constructor. The two methods, `BrowseButton_Click` and `FacePhoto_MouseMove`, correspond to the event handlers declared in *MainWindow.xaml*.

`BrowseButton_Click` creates an `OpenFileDialog`, which allows the user to select a jpg image. The image is read and displayed in the main window. The remaining code for `BrowseButton_Click` and the code for `FacePhoto_MouseMove` are inserted in subsequent steps.

```
using Microsoft.Azure.CognitiveServices.Vision.Face;
using Microsoft.Azure.CognitiveServices.Vision.Face.Models;

using System;
using System.Collections.Generic;
using System.IO;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;

namespace FaceTutorial
{
    public partial class MainWindow : Window
    {
        // Replace <SubscriptionKey> with your valid subscription key.
        // For example, subscriptionKey = "0123456789abcdef0123456789ABCDEF"
        private const string subscriptionKey = "<SubscriptionKey>";

        // Replace or verify the region.
        //
        // You must use the same region as you used to obtain your subscription
        // keys. For example, if you obtained your subscription keys from the
        // westus region, replace "westcentralus" with "westus".
        //
        // NOTE: Free trial subscription keys are generated in the westcentralus
        // region, so if you are using a free trial subscription key, you should
        // not need to change this region.
        private const string faceEndpoint =
            "https://westcentralus.api.cognitive.microsoft.com";

        private readonly IFaceClient faceClient = new FaceClient(
            new ApiKeyServiceClientCredentials(subscriptionKey),
            new System.Net.Http.DelegatingHandler[] { });

        IList<DetectedFace> faceList;   // The list of detected faces.
        String[] faceDescriptions;      // The list of descriptions for the detected faces.
        double resizeFactor;            // The resize factor for the displayed image.

        public MainWindow()
        {
            InitializeComponent();

            if (Uri.IsWellFormedUriString(faceEndpoint, UriKind.Absolute))
            {
                faceClient.Endpoint = faceEndpoint;
            }
            else
            {
                MessageBox.Show(faceEndpoint,
                    "Invalid URI", MessageBoxButton.OK, MessageBoxImage.Error);
                Environment.Exit(0);
            }
        }

        // Displays the image and calls UploadAndDetectFaces.
        private async void BrowseButton_Click(object sender, RoutedEventArgs e)
        {
            // Get the image file to scan from the user.
            var openDlg = new Microsoft.Win32.OpenFileDialog();

            openDlg.Filter = "JPEG Image(*.jpg)|*.jpg";
            bool? result = openDlg.ShowDialog(this);
            bool? result = openDlg.ShowDialog(this);

            // Return if canceled.
            if (!(bool)result)
            {
                return;
            }

            // Display the image file.
            string filePath = openDlg.FileName;

            Uri fileUri = new Uri(filePath);
            BitmapImage bitmapSource = new BitmapImage();

            bitmapSource.BeginInit();
            bitmapSource.CacheOption = BitmapCacheOption.None;
            bitmapSource.UriSource = fileUri;
            bitmapSource.EndInit();

            FacePhoto.Source = bitmapSource;
        }

        // Displays the face description when the mouse is over a face rectangle.
        private void FacePhoto_MouseMove(object sender, MouseEventArgs e)
        {
        }
    }
}
```

Insert your subscription key and verify or change the region

- Find the following line in *MainWindow.xaml.cs* and replace `<Subscription Key>` with your Face API subscription key:

  ```
  private const string subscriptionKey = "<SubscriptionKey>";
  ```

- Find the following line in *MainWindow.xaml.cs* and replace or verify the Azure region associated with your subscription key:

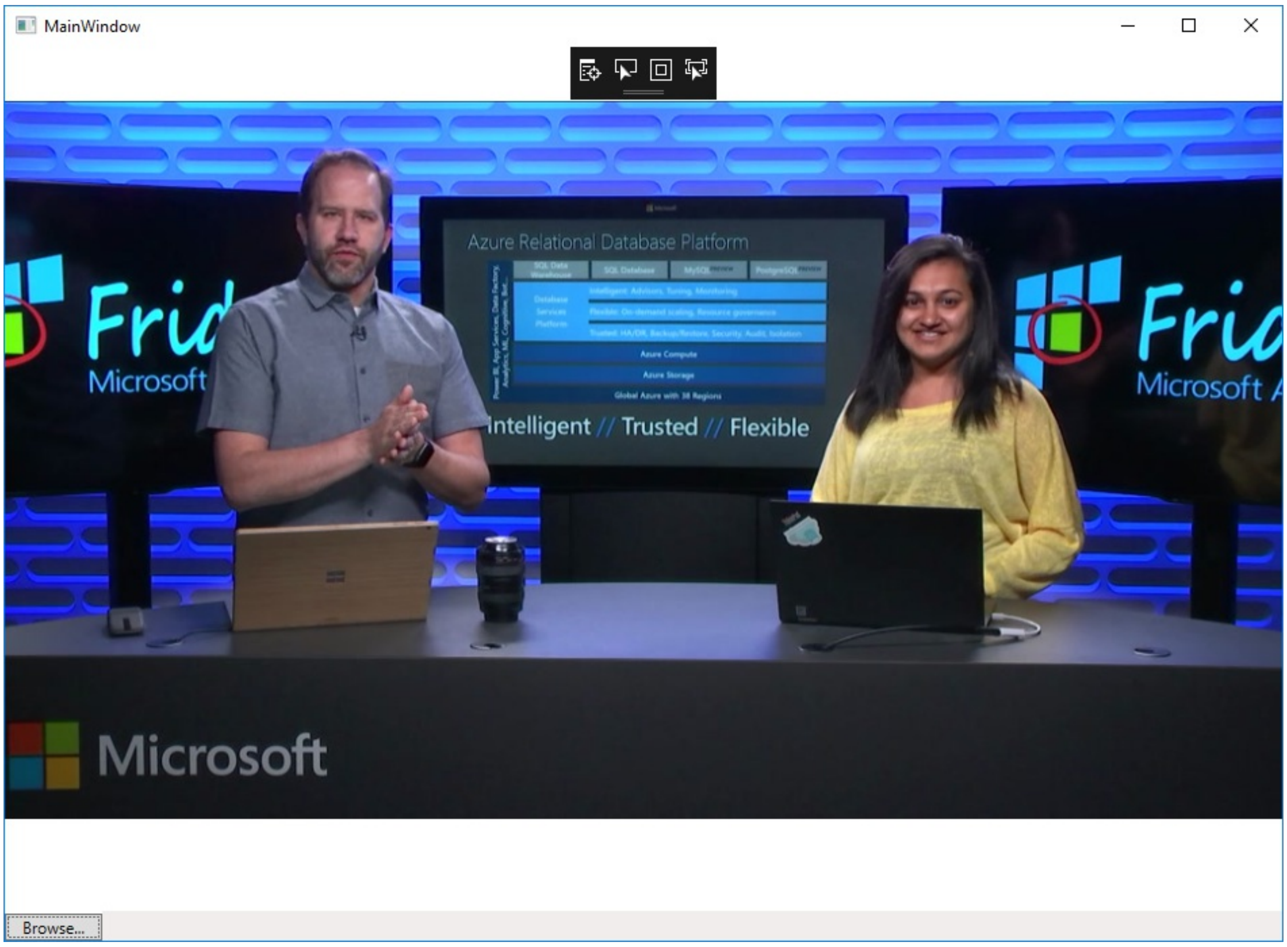
  ```
  private const string Endpoint =
      "https://westcentralus.api.cognitive.microsoft.com";
  ```

  Make sure the location is the same as where you obtained your subscription keys. If you obtained your subscription keys from the **westus** region, for example, replace `Westcentralus` with `Westus`.

  If you received your subscription keys by using the free trial, the region for your keys is **westcentralus**, so no change is required.

Test the app

Press **Start** on the menu to test your app. When the window opens, click **Browse** in the lower left corner. A **File Open** dialog appears where you can browse and select a photo, which is then displayed in the window.



Upload an image to detect faces

The most straightforward way to detect faces is by calling the `FaceClient.Face.DetectWithStreamAsync` method, which wraps the Detect API method for uploading the local image.

Insert the following method in the `MainWindow` class, below the `FacePhoto_MouseMove` method.

A list of face attributes to analyze is created and the submitted image file is read into a `Stream`. Both are passed to the `DetectWithStreamAsync` call.

```
// Uploads the image file and calls DetectWithStreamAsync.
private async Task<IList<DetectedFace>> UploadAndDetectFaces(string imageFilePath)
{
    // The list of Face attributes to return.
    IList<FaceAttributeType> faceAttributes =
        new FaceAttributeType[]
        {
            FaceAttributeType.Gender, FaceAttributeType.Age,
            FaceAttributeType.Smile, FaceAttributeType.Emotion,
            FaceAttributeType.Glasses, FaceAttributeType.Hair
        };

    // Call the Face API.
    try
    {
        using (Stream imageFileStream = File.OpenRead(imageFilePath))
        {
            // The second argument specifies to return the faceId, while
            // the third argument specifies not to return face landmarks.
            IList<DetectedFace> faceList =
                await faceClient.Face.DetectWithStreamAsync(
                    imageFileStream, true, false, faceAttributes);
            return faceList;
        }
    }
    // Catch and display Face API errors.
    catch (APIErrorException f)
    {
        MessageBox.Show(f.Message);
        return new List<DetectedFace>();
    }
    // Catch and display all other errors.
    catch (Exception e)
    {
        MessageBox.Show(e.Message, "Error");
        return new List<DetectedFace>();
    }
}
```

Draw rectangles around each face

Add the code to draw a rectangle around each detected face in the image.

In *MainWindow.xaml.cs*, add the `async` modifier to the `BrowseButton_Click` method.

```
private async void BrowseButton_Click(object sender, RoutedEventArgs e)
```

Insert the following code at the end of the `BrowseButton_Click` method, after the `FacePhoto.Source = bitmapSource` line.

The list of detected faces is populated by the call to `UploadAndDetectFaces`. Next, a rectangle is drawn around each face, and the modified image is displayed in the main window.

```
// Detect any faces in the image.
Title = "Detecting...";
faceList = await UploadAndDetectFaces(filePath);
Title = String.Format(
    "Detection Finished. {0} face(s) detected", faceList.Count);

if (faceList.Count > 0)
{
    // Prepare to draw rectangles around the faces.
    DrawingVisual visual = new DrawingVisual();
    DrawingContext drawingContext = visual.RenderOpen();
    drawingContext.DrawImage(bitmapSource,
        new Rect(0, 0, bitmapSource.Width, bitmapSource.Height));
    double dpi = bitmapSource.DpiX;
    resizeFactor = (dpi > 0) ? 96 / dpi : 1;
    faceDescriptions = new String[faceList.Count];

    for (int i = 0; i < faceList.Count; ++i)
    {
        DetectedFace face = faceList[i];

        // Draw a rectangle on the face.
        drawingContext.DrawRectangle(
            Brushes.Transparent,
            new Pen(Brushes.Red, 2),
            new Rect(
                face.FaceRectangle.Left * resizeFactor,
                face.FaceRectangle.Top * resizeFactor,
                face.FaceRectangle.Width * resizeFactor,
                face.FaceRectangle.Height * resizeFactor
                )
        );

        // Store the face description.
        faceDescriptions[i] = FaceDescription(face);
    }

    drawingContext.Close();

    // Display the image with the rectangle around the face.
    RenderTargetBitmap faceWithRectBitmap = new RenderTargetBitmap(
        (int)(bitmapSource.PixelWidth * resizeFactor),
        (int)(bitmapSource.PixelHeight * resizeFactor),
        96,
        96,
        PixelFormats.Pbgra32);

    faceWithRectBitmap.Render(visual);
    FacePhoto.Source = faceWithRectBitmap;

    // Set the status bar text.
    faceDescriptionStatusBar.Text =
        "Place the mouse pointer over a face to see the face description.";
}
```

Describe the faces in the image

Append the following method to the `MainWindow` class, below the `UploadAndDetectFaces` method.

The method converts the face attributes into a string describing the face. The string is displayed when the mouse pointer hovers over the face rectangle.

```
// Creates a string out of the attributes describing the face.
private string FaceDescription(DetectedFace face)
{
    StringBuilder sb = new StringBuilder();

    sb.Append("Face: ");

    // Add the gender, age, and smile.
    sb.Append(face.FaceAttributes.Gender);
    sb.Append(", ");
    sb.Append(face.FaceAttributes.Age);
    sb.Append(", ");
    sb.Append(String.Format("smile {0:F1}%, ", face.FaceAttributes.Smile * 100));

    // Add the emotions. Display all emotions over 10%.
    sb.Append("Emotion: ");
    Emotion emotionScores = face.FaceAttributes.Emotion;
    if (emotionScores.Anger >= 0.1f)
        sb.Append(String.Format("anger {0:F1}%, ", emotionScores.Anger * 100));
    if (emotionScores.Contempt >= 0.1f)
        sb.Append(String.Format("contempt {0:F1}%, ", emotionScores.Contempt * 100));
    if (emotionScores.Disgust >= 0.1f)
        sb.Append(String.Format("disgust {0:F1}%, ", emotionScores.Disgust * 100));
    if (emotionScores.Fear >= 0.1f)
        sb.Append(String.Format("fear {0:F1}%, ", emotionScores.Fear * 100));
    if (emotionScores.Happiness >= 0.1f)
        sb.Append(String.Format("happiness {0:F1}%, ", emotionScores.Happiness * 100));
    if (emotionScores.Neutral >= 0.1f)
        sb.Append(String.Format("neutral {0:F1}%, ", emotionScores.Neutral * 100));
    if (emotionScores.Sadness >= 0.1f)
        sb.Append(String.Format("sadness {0:F1}%, ", emotionScores.Sadness * 100));
    if (emotionScores.Surprise >= 0.1f)
        sb.Append(String.Format("surprise {0:F1}%, ", emotionScores.Surprise * 100));

    // Add glasses.
    sb.Append(face.FaceAttributes.Glasses);
    sb.Append(", ");

    // Add hair.
    sb.Append("Hair: ");

    // Display baldness confidence if over 1%.
    if (face.FaceAttributes.Hair.Bald >= 0.01f)
        sb.Append(String.Format("bald {0:F1}% ", face.FaceAttributes.Hair.Bald * 100));

    // Display all hair color attributes over 10%.
    IList<HairColor> hairColors = face.FaceAttributes.Hair.HairColor;
    foreach (HairColor hairColor in hairColors)
    {
        if (hairColor.Confidence >= 0.1f)
        {
            sb.Append(hairColor.Color.ToString());
            sb.Append(String.Format(" {0:F1}% ", hairColor.Confidence * 100));
        }
    }

    // Return the built string.
    return sb.ToString();
}
```

Display the face description

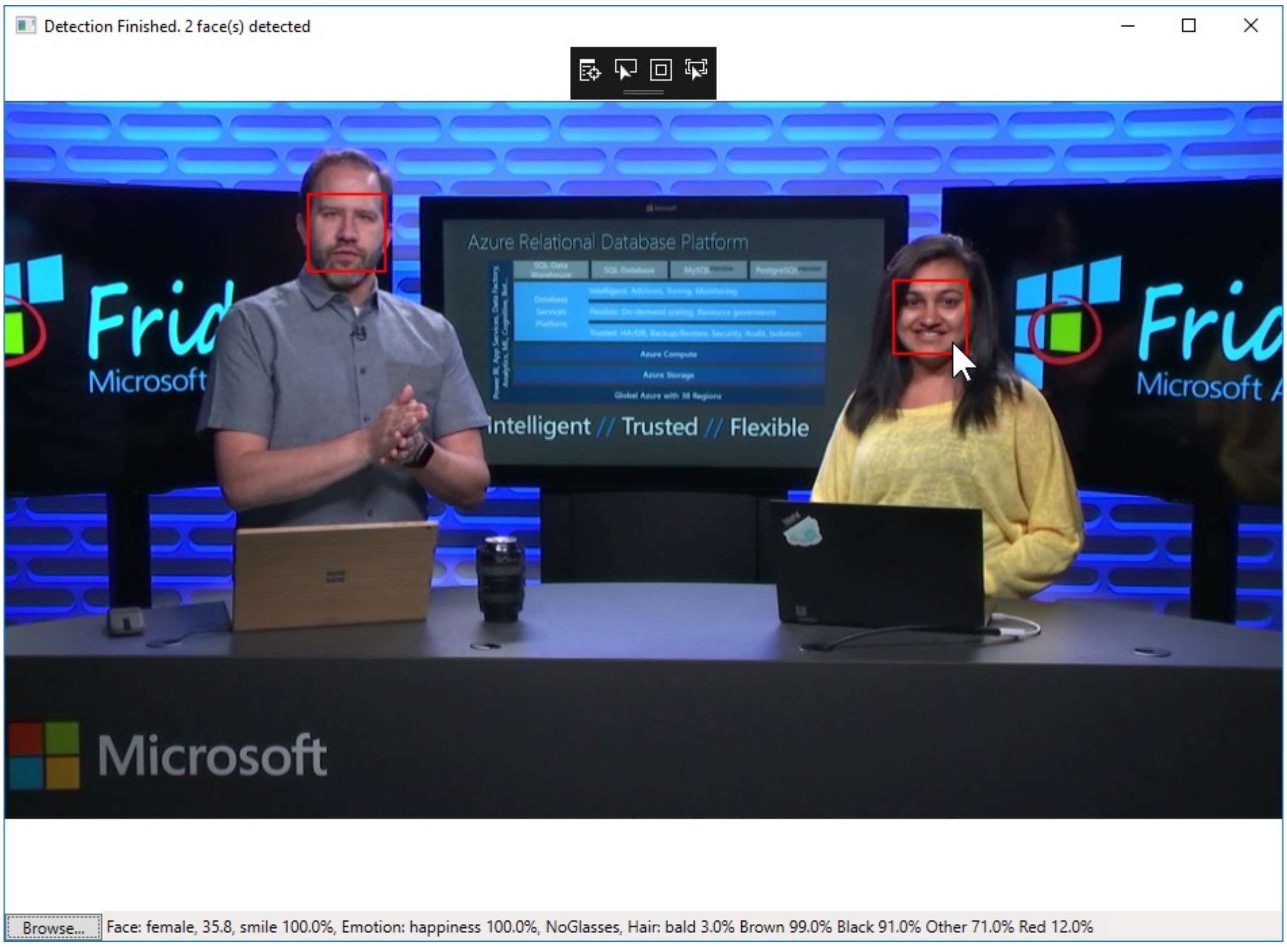
Replace the `FacePhoto_MouseMove` method with the following code.

This event handler displays the face description string when the mouse pointer hovers over the face rectangle.

```
private void FacePhoto_MouseMove(object sender, MouseEventArgs e)
{
    // If the REST call has not completed, return.
    if (faceList == null)
        return;

    // Find the mouse position relative to the image.
    Point mouseXY = e.GetPosition(FacePhoto);

    ImageSource imageSource = FacePhoto.Source;
    BitmapSource bitmapSource = (BitmapSource)imageSource;

    // Scale adjustment between the actual size and displayed size.
    var scale = FacePhoto.ActualWidth / (bitmapSource.PixelWidth / resizeFactor);

    // Check if this mouse position is over a face rectangle.
    bool mouseOverFace = false;

    for (int i = 0; i < faceList.Count; ++i)
    {
        FaceRectangle fr = faceList[i].FaceRectangle;
        double left = fr.Left * scale;
        double top = fr.Top * scale;
        double width = fr.Width * scale;
        double height = fr.Height * scale;

        // Display the face description if the mouse is over this face rectangle.
        if (mouseXY.X >= left && mouseXY.X <= left + width &&
            mouseXY.Y >= top  && mouseXY.Y <= top + height)
        {
            faceDescriptionStatusBar.Text = faceDescriptions[i];
            mouseOverFace = true;
            break;
        }
    }

    // String to display when the mouse is not over a face rectangle.
    if (!mouseOverFace)
        faceDescriptionStatusBar.Text =
            "Place the mouse pointer over a face to see the face description.";
}
```

Run the app

Run the application and browse for an image containing a face. Wait for a few seconds to allow the Face service to respond. After that, you'll see a red rectangle on the faces in the image. By moving the mouse over a face rectangle, the description of that face appears on the status bar.


Detection Finished. 2 face(s) detected

Face: female, 35.8, smile 100.0%, Emotion: happiness 100.0%, NoGlasses, Hair: bald 3.0% Brown 99.0% Black 91.0% Other 71.0% Red 12.0%

Summary

In this tutorial, you learned the basic process for using the Face service client library, and created an application to display and frame faces in an image.

Next steps

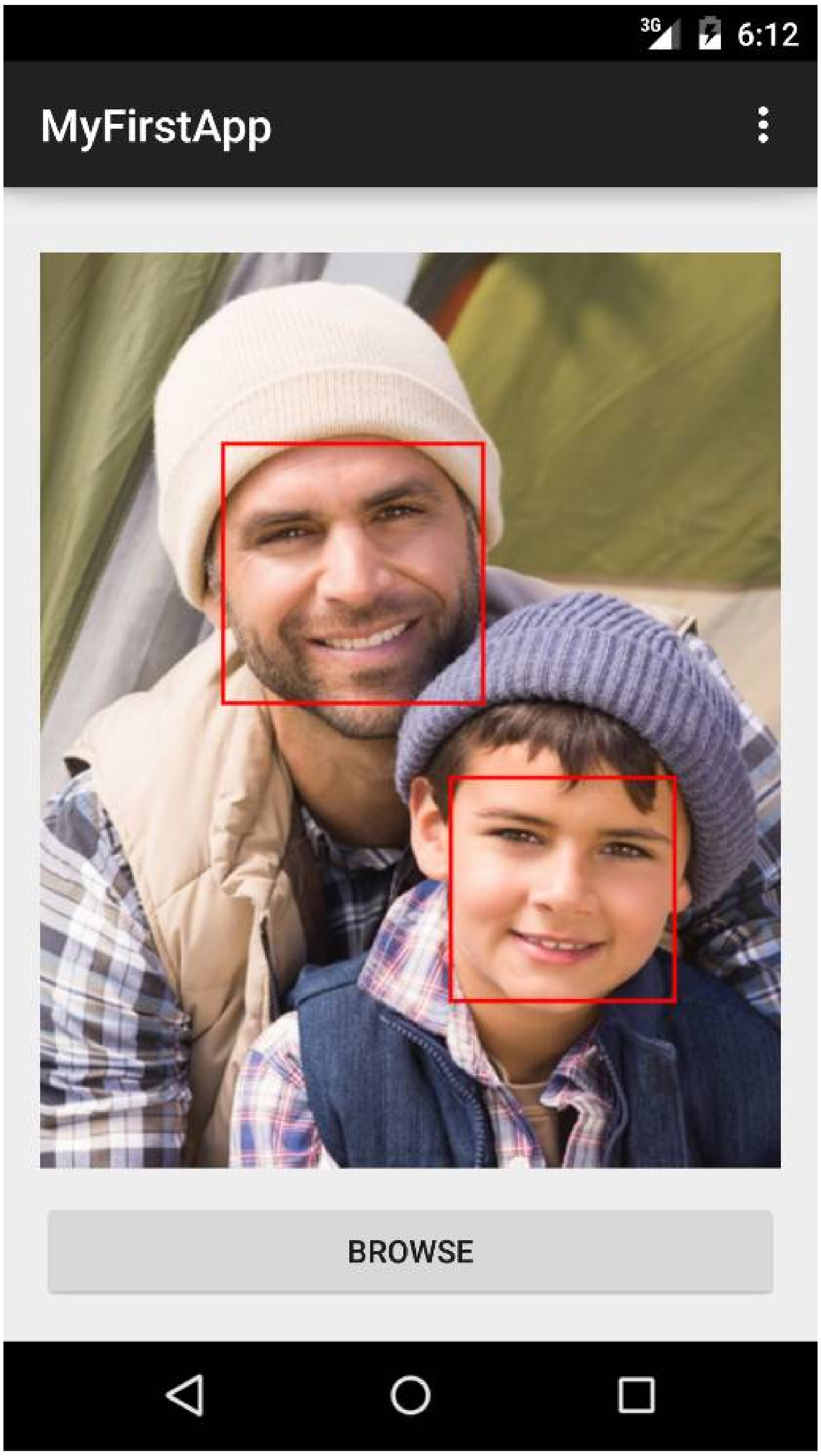
Learn about detecting and using face landmarks.

How to Detect Faces in an Image

Tutorial: Create an Android app to detect and frame faces in an image

10/23/2018 • 6 minutes to read • Edit Online

In this tutorial, you create a simple Android application that uses the Face service Java class library to detect human faces in an image. The application shows a selected image with each detected face framed by a rectangle. The complete sample code is available on GitHub at Detect and frame faces in an image on Android.



This tutorial shows you how to:

- Create an Android application

- Install the Face service client library
- Use the client library to detect faces in an image
- Draw a frame around each detected face

Prerequisites

- You need a subscription key to run the sample. You can get free trial subscription keys from Try Cognitive Services.
- Android Studio with minimum SDK 22 (required by the Face client library).
- The com.microsoft.projectoxford:face:1.4.3 Face client library from Maven. It isn't necessary to download the package. Installation instructions are provided below.

Create the project

Create your Android application project by following these steps:

1. Open Android Studio. This tutorial uses Android Studio 3.1.
2. Select **Start a new Android Studio project**.
3. On the **Create Android Project** screen, modify the default fields, if necessary, then click **Next**.
4. On the **Target Android Devices** screen, use the dropdown selector to choose **API 22** or higher, then click **Next**.
5. Select **Empty Activity**, then click **Next**.
6. Uncheck **Backwards Compatibility**, then click **Finish**.

Create the UI for selecting and displaying the image

Open *activity_main.xml*; you should see the Layout Editor. Select the **Text** tab, then replace the contents with the following code.

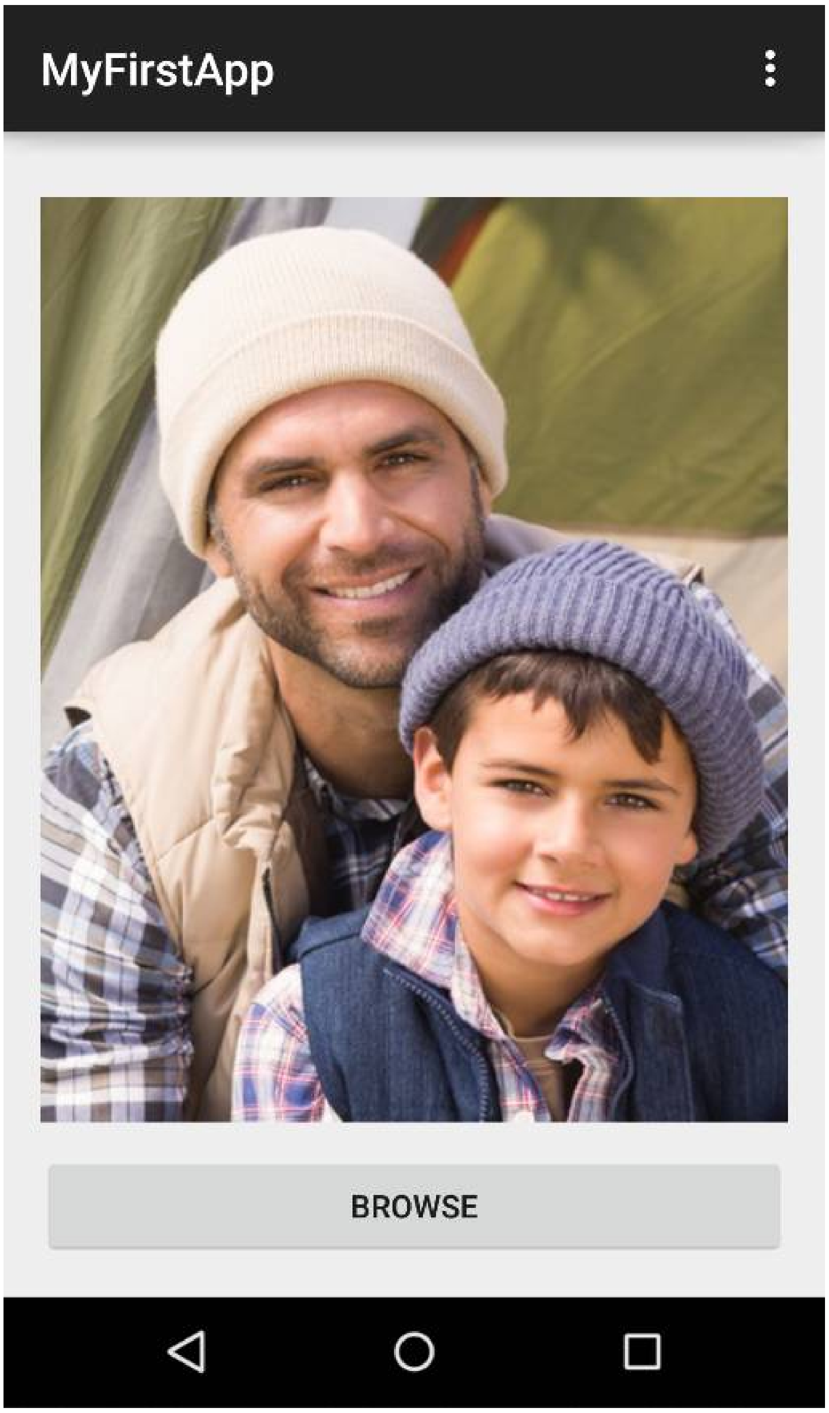
```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <ImageView
        android:layout_width="match_parent"
        android:layout_height="fill_parent"
        android:id="@+id/imageView1"
        android:layout_above="@+id/button1"
        android:contentDescription="Image with faces to analyze"/>

    <Button
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Browse for face image"
        android:id="@+id/button1"
        android:layout_alignParentBottom="true"/>
</RelativeLayout>
```

Open *MainActivity.java*, then replace everything but the first `package` statement with the following code.

The code sets an event handler on the `Button` that starts a new activity to allow the user to select a picture. Once selected, the picture is displayed in the `ImageView`.

```
import java.io.*;
import android.app.*;
import android.content.*;
import android.net.*;
import android.os.*;
import android.view.*;
import android.graphics.*;
import android.widget.*;
import android.provider.*;

public class MainActivity extends Activity {
    private final int PICK_IMAGE = 1;
    private ProgressDialog detectionProgressDialog;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
            setContentView(R.layout.activity_main);
            Button button1 = (Button)findViewById(R.id.button1);
            button1.setOnClickListener(new View.OnClickListener() {
                @Override
                public void onClick(View v) {
                Intent intent = new Intent(Intent.ACTION_GET_CONTENT);
                intent.setType("image/*");
                startActivityForResult(Intent.createChooser(
                        intent, "Select Picture"), PICK_IMAGE);
            }
        });

        detectionProgressDialog = new ProgressDialog(this);
    }

    @Override
    protected void onActivityResult(int requestCode, int resultCode, Intent data) {
        super.onActivityResult(requestCode, resultCode, data);
        if (requestCode == PICK_IMAGE && resultCode == RESULT_OK &&
                data != null && data.getData() != null) {
            Uri uri = data.getData();
            try {
                Bitmap bitmap = MediaStore.Images.Media.getBitmap(
                        getContentResolver(), uri);
                ImageView imageView = (ImageView) findViewById(R.id.imageView1);
                imageView.setImageBitmap(bitmap);

                // Uncomment
                //detectAndFrame(bitmap);
                } catch (IOException e) {
                    e.printStackTrace();
                }
        }
    }
}
```

Now your app can browse for a photo and display it in the window, similar to the image below.

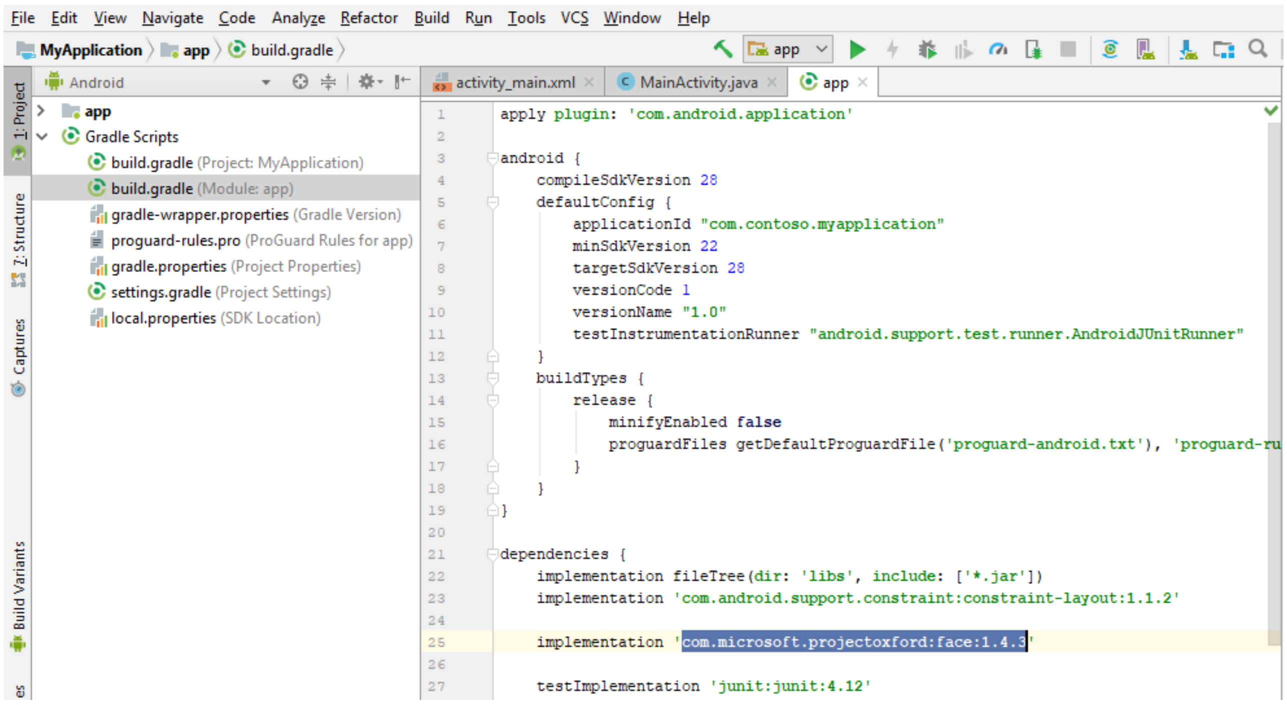


Configure the Face client library

The Face API is a cloud API, which you can call using HTTPS requests. This tutorial uses the Face client library, which encapsulates these web requests, to simplify your work.

In the **Project** pane, use the dropdown selector to select **Android**. Expand **Gradle Scripts**, then open *build.gradle (Module: app)*.

Add a dependency for the Face client library, `com.microsoft.projectoxford:face:1.4.3`, as shown in the screenshot below, then click **Sync Now**.


File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
MyApplication
app
build.gradle
Android
1: Project
7: Structure
Captures
Build Variants
app
Gradle Scripts
build.gradle (Project: MyApplication)
build.gradle (Module: app)
gradle-wrapper.properties (Gradle Version)
proguard-rules.pro (ProGuard Rules for app)
gradle.properties (Project Properties)
settings.gradle (Project Settings)
local.properties (SDK Location)
activity_main.xml
MainActivity.java
app

```
apply plugin: 'com.android.application'

android {
    compileSdkVersion 28
    defaultConfig {
        applicationId "com.contoso.myapplication"
        minSdkVersion 22
        targetSdkVersion 28
        versionCode 1
        versionName "1.0"
        testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
    }
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-ru
        }
    }
}

dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
    implementation 'com.android.support.constraint:constraint-layout:1.1.2'

    implementation 'com.microsoft.projectoxford:face:1.4.3'

    testImplementation 'junit:junit:4.12'
```

Open **MainActivity.java** and append the following import directives:

```
import com.microsoft.projectoxford.face.*;
import com.microsoft.projectoxford.face.contract.*;
```

Add the Face client library code

Insert the following code in the `MainActivity` class, above the `onCreate` method:

```
private final String apiEndpoint = "<API endpoint>";
private final String subscriptionKey = "<Subscription Key>";

private final FaceServiceClient faceServiceClient =
        new FaceServiceRestClient(apiEndpoint, subscriptionKey);
```

Replace `<API endpoint>` with the API endpoint that was assigned to your key. Free trial subscription keys are generated in the **westcentralus** region. So if you're using a free trial subscription key, the statement would be:

```
apiEndpoint = "https://westcentralus.api.cognitive.microsoft.com/face/v1.0";
```

Replace `<Subscription Key>` with your subscription key. For example:

```
subscriptionKey = "0123456789abcdef0123456789ABCDEF"
```

In the **Project** pane, expand **app**, then **manifests**, and open *AndroidManifest.xml*.

Insert the following element as a direct child of the `manifest` element:

```
<uses-permission android:name="android.permission.INTERNET" />
```

Build your project to check for errors. Now you're ready to call the Face service.

Upload an image to detect faces

The most straightforward way to detect faces is to call the `FaceServiceClient.detect` method. This method wraps the Detect API method and returns an array of `Face` 's.

Each returned `Face` includes a rectangle to indicate its location, combined with a series of optional face attributes. In this example, only the face locations are required.

If an error occurs, an `AlertDialog` displays the underlying reason.

Insert the following methods into the `MainActivity` class.

```
// Detect faces by uploading a face image.
// Frame faces after detection.
private void detectAndFrame(final Bitmap imageBitmap) {
    ByteArrayOutputStream outputStream = new ByteArrayOutputStream();
    imageBitmap.compress(Bitmap.CompressFormat.JPEG, 100, outputStream);
    ByteArrayInputStream inputStream =
            new ByteArrayInputStream(outputStream.toByteArray());

    AsyncTask<InputStream, String, Face[]> detectTask =
            new AsyncTask<InputStream, String, Face[]>() {
                String exceptionMessage = "";

                @Override
                protected Face[] doInBackground(InputStream... params) {
                    try {
                        publishProgress("Detecting...");
                        Face[] result = faceServiceClient.detect(
                                params[0],
                                true,         // returnFaceId
                                false,        // returnFaceLandmarks
                                null          // returnFaceAttributes:
                                /* new FaceServiceClient.FaceAttributeType[] {
                                    FaceServiceClient.FaceAttributeType.Age,
                                    FaceServiceClient.FaceAttributeType.Gender }
                                */
                        );
                        if (result == null){
                            publishProgress(
                                    "Detection Finished. Nothing detected");
                            return null;
                        }
                        publishProgress(String.format(
                                "Detection Finished. %d face(s) detected",
                                result.length));
                        return result;
                    } catch (Exception e) {
                        exceptionMessage = String.format(
                                "Detection failed: %s", e.getMessage());
                        return null;
                    }
                }

                @Override
                protected void onPreExecute() {
                    //TODO: show progress dialog
                }
                @Override
                protected void onProgressUpdate(String... progress) {
                    //TODO: update progress
                }
                @Override
                protected void onPostExecute(Face[] result) {
                    //TODO: update face frames
                }
            };

    detectTask.execute(inputStream);
}

private void showError(String message) {
    new AlertDialog.Builder(this)
    .setTitle("Error")
    .setMessage(message)
    .setPositiveButton("OK", new DialogInterface.OnClickListener() {
            public void onClick(DialogInterface dialog, int id) {
        }})
    .create().show();
}
```

Frame faces in the image

Insert the following helper method into the `MainActivity` class. This method draws a rectangle around each detected face.

```
private static Bitmap drawFaceRectanglesOnBitmap(
        Bitmap originalBitmap, Face[] faces) {
    Bitmap bitmap = originalBitmap.copy(Bitmap.Config.ARGB_8888, true);
    Canvas canvas = new Canvas(bitmap);
    Paint paint = new Paint();
    paint.setAntiAlias(true);
    paint.setStyle(Paint.Style.STROKE);
    paint.setColor(Color.RED);
    paint.setStrokeWidth(10);
    if (faces != null) {
        for (Face face : faces) {
            FaceRectangle faceRectangle = face.faceRectangle;
            canvas.drawRect(
                    faceRectangle.left,
                    faceRectangle.top,
                    faceRectangle.left + faceRectangle.width,
                    faceRectangle.top + faceRectangle.height,
                    paint);
        }
    }
    return bitmap;
}
```

Complete the `AsyncTask` methods, indicated by the `TODO` comments, in the `detectAndFrame` method. On success, the selected image is displayed with framed faces in the `ImageView`.

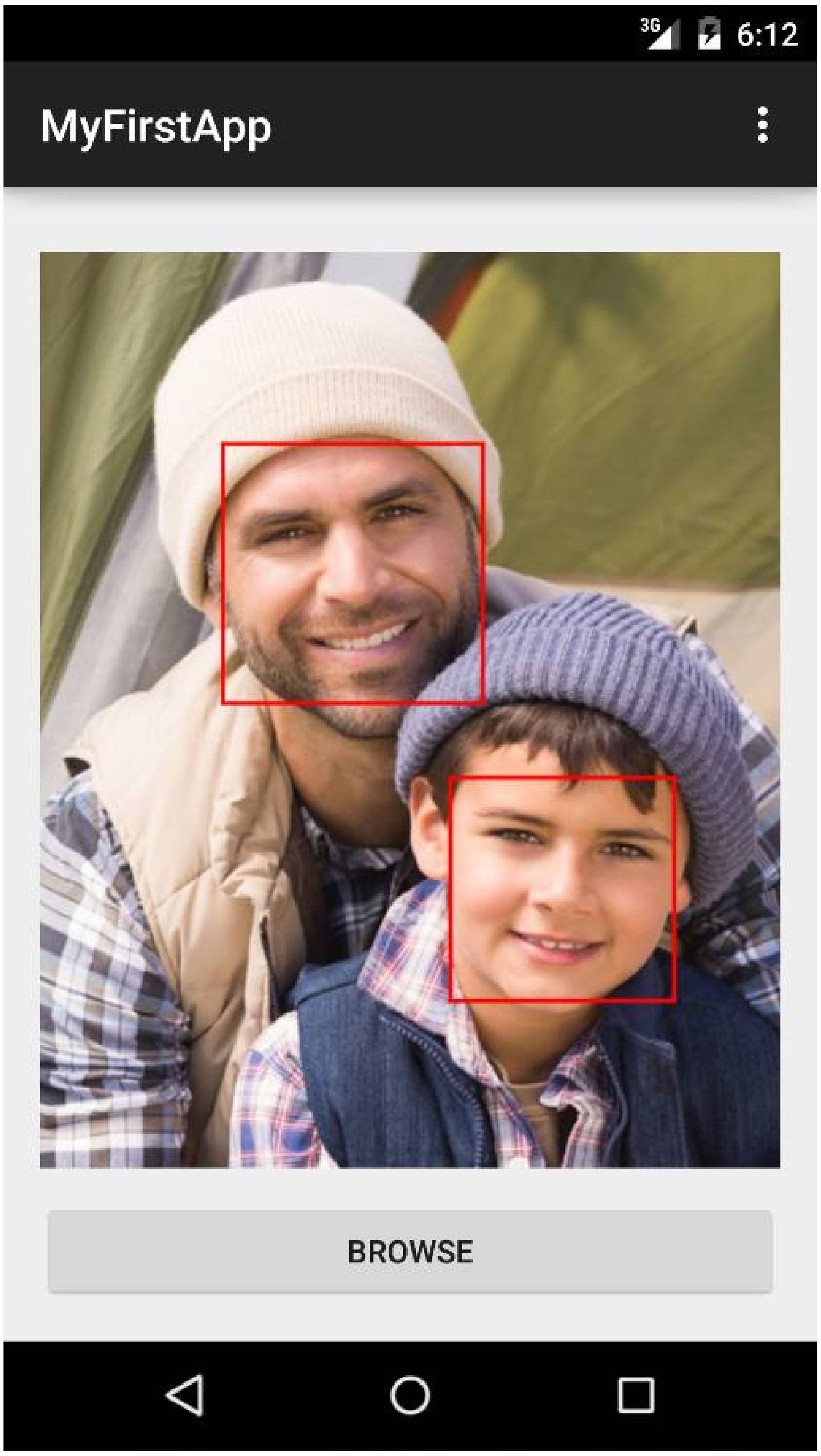
```
@Override
protected void onPreExecute() {
    detectionProgressDialog.show();
}
@Override
protected void onProgressUpdate(String... progress) {
    detectionProgressDialog.setMessage(progress[0]);
}
@Override
protected void onPostExecute(Face[] result) {
    detectionProgressDialog.dismiss();
    if(!exceptionMessage.equals("")){
        showError(exceptionMessage);
    }
    if (result == null) return;
    ImageView imageView = findViewById(R.id.imageView1);
    imageView.setImageBitmap(
            drawFaceRectanglesOnBitmap(imageBitmap, result));
    imageBitmap.recycle();
}
```

Finally, in the `onActivityResult` method, uncomment the call to the `detectAndFrame` method, as shown below.

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    super.onActivityResult(requestCode, resultCode, data);

    if (requestCode == PICK_IMAGE && resultCode == RESULT_OK &&
                data != null && data.getData() != null) {
        Uri uri = data.getData();
        try {
            Bitmap bitmap = MediaStore.Images.Media.getBitmap(
                    getContentResolver(), uri);
            ImageView imageView = findViewById(R.id.imageView1);
            imageView.setImageBitmap(bitmap);

            // Uncomment
            detectAndFrame(bitmap);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Run the app

Run the application and browse for an image with a face. Wait a few seconds to allow the Face service to respond. After that, you'll get a result similar to the image below:



Summary

In this tutorial, you learned the basic process for using the Face service and created an application to display framed faces in an image.

Next steps

Learn about detecting and using face landmarks.

How to Detect Faces in an Image

Explore the Face APIs used to detect faces and their attributes such as pose, gender, age, head pose, facial hair, and glasses.

Face API Reference.

Tutorial: Detect and frame faces with the Face API and Python

9/18/2018 • 2 minutes to read • Edit Online

In this tutorial, you will learn to invoke the Face API via the Python SDK to detect human faces in an image.

Prerequisites

To use the tutorial, you will need to do the following:

- Install either Python 2.7+ or Python 3.5+.
- Install pip.
- Install the Python SDK for the Face API as follows:

```
pip install cognitive_face
```

- Obtain a subscription key for Microsoft Cognitive Services. You can use either your primary or your secondary key in this tutorial. (Note that to use any Face API, you must have a valid subscription key.)

Detect a Face in an Image

```
import cognitive_face as CF

KEY = '<Subscription Key>'  # Replace with a valid subscription key (keeping the quotes in place).
CF.Key.set(KEY)

BASE_URL = 'https://westus.api.cognitive.microsoft.com/face/v1.0/'  # Replace with your regional Base URL
CF.BaseUrl.set(BASE_URL)

# You can use this example JPG or replace the URL below with your own URL to a JPEG image.
img_url = 'https://raw.githubusercontent.com/Microsoft/Cognitive-Face-Windows/master/Data/detection1.jpg'
faces = CF.face.detect(img_url)
print(faces)
```

Below is an example result. It's a `list` of detected faces. Each item in the list is a `dict` instance where `faceId` is a unique ID for the detected face and `faceRectangle` describes the position of the detected face. A face ID expires in 24 hours.

```
[{u'faceId': u'68a0f8cf-9dba-4a25-afb3-f9cdf57cca51', u'faceRectangle': {u'width': 89, u'top': 66, u'height':
89, u'left': 446}}]
```

Draw rectangles around the faces

Using the json coordinates that you received from the previous command, you can draw rectangles on the image to visually represent each face. You will need to `pip install Pillow` to use the `PIL` imaging module. At the top of the file, add the following:

```
import requests
from io import BytesIO
from PIL import Image, ImageDraw
```

Then, after `print(faces)`, include the following in your script:

```
#Convert width height to a point in a rectangle
def getRectangle(faceDictionary):
    rect = faceDictionary['faceRectangle']
    left = rect['left']
    top = rect['top']
    bottom = left + rect['height']
    right = top + rect['width']
    return ((left, top), (bottom, right))

#Download the image from the url
response = requests.get(img_url)
img = Image.open(BytesIO(response.content))

#For each face returned use the face rectangle and draw a red box.
draw = ImageDraw.Draw(img)
for face in faces:
    draw.rectangle(getRectangle(face), outline='red')

#Display the image in the users default image browser.
img.show()
```

Further Exploration

To help you further explore the Face API, this tutorial provides a GUI sample. To run it, first install wxPython then run the commands below.

```
git clone https://github.com/Microsoft/Cognitive-Face-Python.git
cd Cognitive-Face-Python
python sample
```

Summary

In this tutorial, you have learned the basic process for using the Face API via invoking the Python SDK. For more information on API details, please refer to the How-To and API Reference.

Related Topics

- Getting Started with Face API in CSharp
- Getting Started with Face API in Java for Android

Example: How to Detect Faces in Image

9/18/2018 • 5 minutes to read • Edit Online

This guide will demonstrate how to detect faces from an image, with face attributes like gender, age, or pose extracted. The samples are written in C# using the Face API client library.

Concepts

If you are not familiar with any of the following concepts in this guide, please refer to the definitions in our Glossary at any time:

- Face detection
- Face landmarks
- Head pose
- Face attributes

Preparation

In this sample, we will demonstrate the following features:

- Detecting faces from an image, and marking them using rectangular framing
- Analyzing the locations of pupils, the nose or mouth, and then marking them in the image
- Analyzing the head pose, gender and age of the face

In order to execute these features, you will need to prepare an image with at least one clear face.

Step 1: Authorize the API call

Every call to the Face API requires a subscription key. This key needs to be either passed through a query string parameter, or specified in the request header. To pass the subscription key through query string, please refer to the request URL for the Face - Detect as an example:

```
https://westus.api.cognitive.microsoft.com/face/v1.0/detect[?returnFaceId][&returnFaceLandmarks]
[&returnFaceAttributes]
&subscription-key=<Subscription Key>
```

As an alternative, the subscription key can also be specified in the HTTP request header: **ocp-apim-subscription-key: <Subscription Key>** When using a client library, the subscription key is passed in through the constructor of the FaceServiceClient class. For example:

```
faceServiceClient = new FaceServiceClient("<Subscription Key>");
```

Step 2: Upload an image to the service and execute face detection

The most basic way to perform face detection is by uploading an image directly. This is done by sending a "POST" request with application/octet-stream content type, with the data read from a JPEG image. The maximum size of the image is 4 MB.

Using the client library, face detection by means of uploading is done by passing in a Stream object. See the example below:

```
using (Stream s = File.OpenRead(@"D:\MyPictures\image1.jpg"))
{
    var faces = await faceServiceClient.DetectAsync(s, true, true);

    foreach (var face in faces)
    {
        var rect = face.FaceRectangle;
        var landmarks = face.FaceLandmarks;
    }
}
```

Note that the DetectAsync method of FaceServiceClient is async. The calling method should be marked as async as well, in order to use the await clause. If the image is already on the web and has a URL, face detection can be executed by also providing the URL. In this example, the request body will be a JSON string, which contains the URL. Using the client library, face detection by means of a URL can be executed easily using another overload of the DetectAsync method.

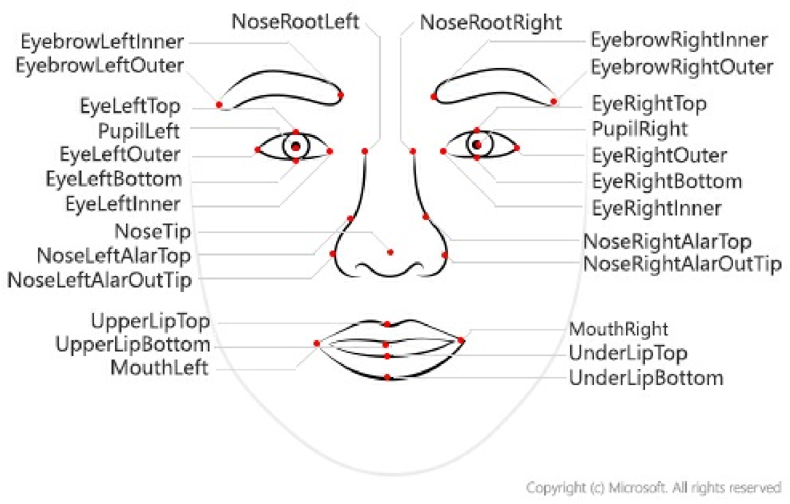
```
string imageUrl = "http://news.microsoft.com/ceo/assets/photos/06_web.jpg";
var faces = await faceServiceClient.DetectAsync(imageUrl, true, true);

foreach (var face in faces)
{
    var rect = face.FaceRectangle;
    var landmarks = face.FaceLandmarks;
}
```

The FaceRectangle property that is returned with detected faces is essentially locations on the face in pixels. Usually, this rectangle contains the eyes, eyebrows, the nose, and the mouth –the top of head, ears, and the chin are not included. If you crop a complete head or mid-shot portrait (a photo ID type image), you may want to expand the area of the rectangular face frame because the area of the face may be too small for some applications. To locate a face more precisely, using face landmarks (locate face features or face direction mechanisms) described in the next section will prove to be useful.

Step 3: Understanding and using face landmarks

Face landmarks are a series of detailed points on a face; typically points of face components like the pupils, canthus, or nose. Face landmarks are optional attributes that can be analyzed during face detection. You can either pass 'true' as a Boolean value to the returnFaceLandmarks query parameter when calling the Face - Detect, or use the returnFaceLandmarks optional parameter for the FaceServiceClient class DetectAsync method in order to include the face landmarks in the detection results.

By default, there are 27 predefined landmark points. The following figure shows how all 27 points are defined:



The points returned are in units of pixels, just like the face rectangular frame. Therefore making it easier to mark specific points of interest in the image. The following code demonstrates retrieving the locations of the nose and pupils:

```
var faces = await faceServiceClient.DetectAsync(imageUrl, returnFaceLandmarks:true);

foreach (var face in faces)
{
    var rect = face.FaceRectangle;
    var landmarks = face.FaceLandmarks;

    double noseX = landmarks.NoseTip.X;
    double noseY = landmarks.NoseTip.Y;

    double leftPupilX = landmarks.PupilLeft.X;
    double leftPupilY = landmarks.PupilLeft.Y;

    double rightPupilX = landmarks.PupilRight.X;
    double rightPupilY = landmarks.PupilRight.Y;
}
```

In addition to marking face features in an image, face landmarks can also be used to accurately calculate the direction of the face. For example, we can define the direction of the face as a vector from the center of the mouth to the center of the eyes. The code below explains this in detail:

```
var landmarks = face.FaceLandmarks;

var upperLipBottom = landmarks.UpperLipBottom;
var underLipTop = landmarks.UnderLipTop;

var centerOfMouth = new Point(
    (upperLipBottom.X + underLipTop.X) / 2,
    (upperLipBottom.Y + underLipTop.Y) / 2);

var eyeLeftInner = landmarks.EyeLeftInner;
var eyeRightInner = landmarks.EyeRightInner;

var centerOfTwoEyes = new Point(
    (eyeLeftInner.X + eyeRightInner.X) / 2,
    (eyeLeftInner.Y + eyeRightInner.Y) / 2);

Vector faceDirection = new Vector(
    centerOfTwoEyes.X - centerOfMouth.X,
    centerOfTwoEyes.Y - centerOfMouth.Y);
```

By knowing the direction that the face is in, you can then rotate the rectangular face frame to align it with the face. It is clear that using face landmarks can provide more detail and utility.

Step 4: Using other face attributes

Besides face landmarks, Face – Detect API can also analyze several other attributes on a face. These attributes include:

- Age
- Gender
- Smile intensity
- Facial hair
- A 3D head pose

These attributes are predicted by using statistical algorithms and may not always be 100% precise. However, they are still helpful when you want to classify faces by these attributes. For more information about each of the attributes, please refer to the Glossary.

Below is a simple example of extracting face attributes during face detection:

```
var requiredFaceAttributes = new FaceAttributeType[] {
                FaceAttributeType.Age,
                FaceAttributeType.Gender,
                FaceAttributeType.Smile,
                FaceAttributeType.FacialHair,
                FaceAttributeType.HeadPose,
                FaceAttributeType.Glasses
            };
var faces = await faceServiceClient.DetectAsync(imageUrl,
    returnFaceLandmarks: true,
    returnFaceAttributes: requiredFaceAttributes);

foreach (var face in faces)
{
    var id = face.FaceId;
    var attributes = face.FaceAttributes;
    var age = attributes.Age;
    var gender = attributes.Gender;
    var smile = attributes.Smile;
    var facialHair = attributes.FacialHair;
    var headPose = attributes.HeadPose;
    var glasses = attributes.Glasses;
}
```

Summary

In this guide you have learned the functionalities of Face - Detect API, how it can detect faces for local uploaded images or image URLs on the web; how it can detect faces by returning rectangular face frames; and how it can also analyze face landmarks, 3D head poses and other face attributes as well.

For more information about API details, please refer to the API reference guide for Face - Detect.

Related Topics

How to Identify Faces in Image

Example: How to identify faces in images

9/18/2018 • 6 minutes to read • Edit Online

This guide demonstrates how to identify unknown faces using PersonGroups, which are created from known people in advance. The samples are written in C# using the Face API client library.

Concepts

If you are not familiar with the following concepts in this guide, please search for the definitions in our glossary at any time:

- Face - Detect
- Face - Identify
- PersonGroup

Preparation

In this sample, we demonstrate the following:

- How to create a PersonGroup - This PersonGroup contains a list of known people.
- How to assign faces to each person - These faces are used as a baseline to identify people. It is recommended to use clear front faces, just like your photo ID. A good set of photos should contain faces of the same person in different poses, clothes' colors or hair styles.

To carry out the demonstration of this sample, you need to prepare a bunch of pictures:

- A few photos with the person's face. Click here to download sample photos for Anna, Bill, and Clare.
- A series of test photos, which may or may not contain the faces of Anna, Bill or Clare used to test identification. You can also select some sample images from the preceding link.

Step 1: Authorize the API call

Every call to the Face API requires a subscription key. This key can be either passed through a query string parameter, or specified in the request header. To pass the subscription key through query string, please refer to the request URL for the Face - Detect as an example:

```
https://westus.api.cognitive.microsoft.com/face/v1.0/detect[?returnFaceId][&returnFaceLandmarks]
[&returnFaceAttributes]
&subscription-key=<Subscription key>
```

As an alternative, the subscription key can also be specified in the HTTP request header: **ocp-apim-subscription-key: <Subscription Key>** When using a client library, the subscription key is passed in through the constructor of the FaceServiceClient class. For example:

```
faceServiceClient = new FaceServiceClient("<Subscription Key>");
```

The subscription key can be obtained from the Marketplace page of your Azure portal. See Subscriptions.

Step 2: Create the PersonGroup

In this step, we created a PersonGroup named "MyFriends" that contains three people: Anna, Bill, and Clare. Each person has several faces registered. The faces need to be detected from the images. After all of these steps, you have a PersonGroup like the following image:

2.1 Define people for the PersonGroup

A person is a basic unit of identify. A person can have one or more known faces registered. However, a PersonGroup is a collection of people, and each person is defined within a particular PersonGroup. The identification is done against a PersonGroup. So, the task is to create a PersonGroup, and then create people in it, such as Anna, Bill, and Clare.

First, you need to create a new PersonGroup. This is executed by using the PersonGroup - Create API. The corresponding client library API is the CreatePersonGroupAsync method for the FaceServiceClient class. The group ID specified to create the group is unique for each subscription –you can also get, update, or delete PersonGroups using other PersonGroup APIs. Once a group is defined, people can then be defined within it by using the PersonGroup Person - Create API. The client library method is CreatePersonAsync. You can add face to each person after they're created.

```
// Create an empty PersonGroup
string personGroupId = "myfriends";
await faceServiceClient.CreatePersonGroupAsync(personGroupId, "My Friends");

// Define Anna
CreatePersonResult friend1 = await faceServiceClient.CreatePersonAsync(
    // Id of the PersonGroup that the person belonged to
    personGroupId,
    // Name of the person
    "Anna"
);

// Define Bill and Clare in the same way
```

2.2 Detect faces and register them to correct person

Detection is done by sending a "POST" web request to the Face - Detect API with the image file in the HTTP request body. When using the client library, face detection is executed through the DetectAsync method for the FaceServiceClient class.

For each face detected you can call PersonGroup Person – Add Face to add it to the correct person.

The following code demonstrates the process of how to detect a face from an image and add it to a person:

```
// Directory contains image files of Anna
const string friend1ImageDir = @"D:\Pictures\MyFriends\Anna\";

foreach (string imagePath in Directory.GetFiles(friend1ImageDir, "*.jpg"))
{
    using (Stream s = File.OpenRead(imagePath))
    {
        // Detect faces in the image and add to Anna
        await faceServiceClient.AddPersonFaceAsync(
            personGroupId, friend1.PersonId, s);
    }
}
// Do the same for Bill and Clare
```

Notice that if the image contains more than one face, only the largest face is added. You can add other faces to the person by passing a string in the format of "targetFace = left, top, width, height" to PersonGroup Person - Add Face API's targetFace query parameter, or using the targetFace optional parameter for the AddPersonFaceAsync method to add other faces. Each face added to the person will be given a unique persisted face ID, which can be used in PersonGroup Person – Delete Face and Face – Identify.

Step 3: Train the PersonGroup

The PersonGroup must be trained before an identification can be performed using it. Moreover, it has to be retrained after adding or removing any person, or if any person has their registered face edited. The training is done by the PersonGroup – Train API. When using the client library, it is simply a call to the TrainPersonGroupAsync method:

```
await faceServiceClient.TrainPersonGroupAsync(personGroupId);
```

The training is an asynchronous process. It may not be finished even after the TrainPersonGroupAsync method returned. You may need to query the training status by PersonGroup - Get Training Status API or GetPersonGroupTrainingStatusAsync method of the client library. The following code demonstrates a simple logic of waiting PersonGroup training to finish:

```
TrainingStatus trainingStatus = null;
while(true)
{
    trainingStatus = await faceServiceClient.GetPersonGroupTrainingStatusAsync(personGroupId);

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

    await Task.Delay(1000);
}
```

Step 4: Identify a face against a defined PersonGroup

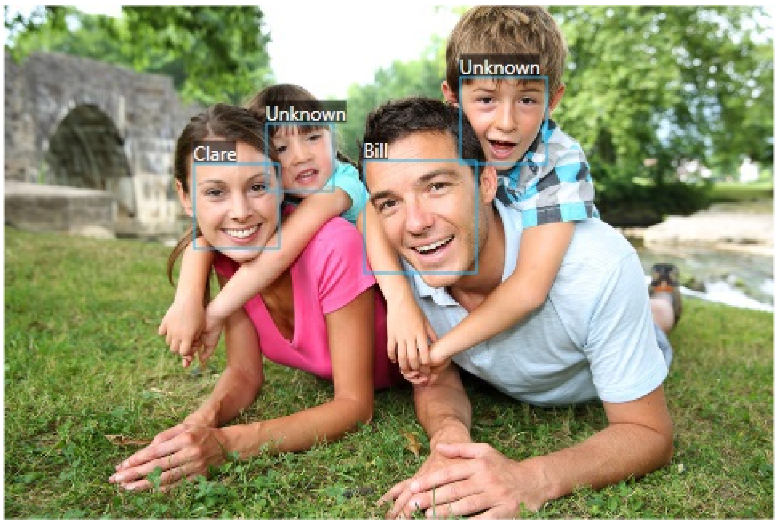
When performing identifications, the Face API can compute the similarity of a test face among all the faces within a group, and returns the most comparable person(s) for that testing face. This is done through the Face - Identify API or the IdentifyAsync method of the client library.

The testing face needs to be detected using the previous steps, and then the face ID is passed to the identify API as a second argument. Multiple face IDs can be identified at once, and the result will contain all the identify results. By default, the identify returns only one person that matches the test face best. If you prefer, you can specify the optional parameter maxNumOfCandidatesReturned to let the identify return more candidates.

The following code demonstrates the process of identify:

```
string testImageFile = @"D:\Pictures\test_img1.jpg";

using (Stream s = File.OpenRead(testImageFile))
{
    var faces = await faceServiceClient.DetectAsync(s);
    var faceIds = faces.Select(face => face.FaceId).ToArray();

    var results = await faceServiceClient.IdentifyAsync(personGroupId, faceIds);
    foreach (var identifyResult in results)
    {
        Console.WriteLine("Result of face: {0}", identifyResult.FaceId);
        if (identifyResult.Candidates.Length == 0)
        {
            Console.WriteLine("No one identified");
        }
        else
        {
            // Get top 1 among all candidates returned
            var candidateId = identifyResult.Candidates[0].PersonId;
            var person = await faceServiceClient.GetPersonAsync(personGroupId, candidateId);
            Console.WriteLine("Identified as {0}", person.Name);
        }
    }
}
```

When you have finished the steps, you can try to identify different faces and see if the faces of Anna, Bill or Clare can be correctly identified, according to the image(s) uploaded for face detection. See the following examples:



Step 5: Request for large-scale

As is known, a PersonGroup can hold up to 10,000 persons due to the limitation of previous design. For more information about up to million-scale scenarios, see How to use the large-scale feature.

Summary

In this guide, you have learned the process of creating a PersonGroup and identifying a person. The following are a quick reminder of the features previously explained and demonstrated:

- Detecting faces using the Face - Detect API

- Creating PersonGroups using the PersonGroup - Create API
- Creating persons using the PersonGroup Person - Create API
- Train a PersonGroup using the PersonGroup – Train API
- Identifying unknown faces against the PersonGroup using the Face - Identify API

Related Topics

- How to Detect Faces in Image
- How to Add Faces
- How to use the large-scale feature

Example: How to add faces

9/18/2018 • 2 minutes to read • Edit Online

This guide demonstrates the best practice to add massive number of persons and faces to a PersonGroup. The same strategy also applies to FaceList and LargePersonGroup. The samples are written in C# using the Face API client library.

Step 1: Initialization

Several variables are declared and a helper function is implemented to schedule the requests.

- `PersonCount` is the total number of persons.
- `CallLimitPerSecond` is the maximum calls per second according to the subscription tier.
- `_timeStampQueue` is a Queue to record the request timestamps.
- `await WaitCallLimitPerSecondAsync()` will wait until it is valid to send next request.

```
const int PersonCount = 10000;
const int CallLimitPerSecond = 10;
static Queue<DateTime> _timeStampQueue = new Queue<DateTime>(CallLimitPerSecond);

static async Task WaitCallLimitPerSecondAsync()
{
    Monitor.Enter(_timeStampQueue);
    try
    {
        if (_timeStampQueue.Count >= CallLimitPerSecond)
        {
            TimeSpan timeInterval = DateTime.UtcNow - _timeStampQueue.Peek();
            if (timeInterval < TimeSpan.FromSeconds(1))
            {
                await Task.Delay(TimeSpan.FromSeconds(1) - timeInterval);
            }
            _timeStampQueue.Dequeue();
        }
        _timeStampQueue.Enqueue(DateTime.UtcNow);
    }
    finally
    {
        Monitor.Exit(_timeStampQueue);
    }
}
```

Step 2: Authorize the API call

When using a client library, the subscription key is passed in through the constructor of the FaceServiceClient class. For example:

```
FaceServiceClient faceServiceClient = new FaceServiceClient("<Subscription Key>");
```

The subscription key can be obtained from the Marketplace page of your Azure portal. See Subscriptions.

Step 3: Create the PersonGroup

A PersonGroup named "MyPersonGroup" is created to save the persons. The request time is enqueued to `_timeStampQueue` to ensure the overall validation.

```
const string personGroupId = "mypersongroupid";
const string personGroupName = "MyPersonGroup";
_timeStampQueue.Enqueue(DateTime.UtcNow);
await faceServiceClient.CreatePersonGroupAsync(personGroupId, personGroupName);
```

Step 4: Create the persons to the PersonGroup

Persons are created concurrently and `await WaitCallLimitPerSecondAsync()` is also applied to avoid exceeding the call limit.

```
CreatePersonResult[] persons = new CreatePersonResult[PersonCount];
Parallel.For(0, PersonCount, async i =>
{
    await WaitCallLimitPerSecondAsync();

    string personName = $"PersonName#{i}";
    persons[i] = await faceServiceClient.CreatePersonAsync(personGroupId, personName);
});
```

Step 5: Add faces to the persons

Adding faces to different persons are processed concurrently, while for one specific person is sequential. Again, `await WaitCallLimitPerSecondAsync()` is invoked to ensure the request frequency is within the scope of limitation.

```
Parallel.For(0, PersonCount, async i =>
{
    Guid personId = persons[i].PersonId;
    string personImageDir = @"/path/to/person/i/images";

    foreach (string imagePath in Directory.GetFiles(personImageDir, "*.jpg"))
    {
        await WaitCallLimitPerSecondAsync();

        using (Stream stream = File.OpenRead(imagePath))
        {
            await faceServiceClient.AddPersonFaceAsync(personGroupId, personId, stream);
        }
    }
});
```

Summary

In this guide, you have learned the process of creating a PersonGroup with massive number of persons and faces. Several reminders:

- This strategy also applies to FaceList and LargePersonGroup.
- Adding/Deleting faces to different FaceLists or Persons in LargePersonGroup can be processed concurrently.
- Same operations to one specific FaceList or Person in LargePersonGroup should be done sequentially.
- To keep the simplicity, the handling of potential exception is omitted in this guide. If you want to enhance more robustness, proper retry policy should be applied.

The following are a quick reminder of the features previously explained and demonstrated:

- Creating PersonGroups using the PersonGroup - Create API
- Creating persons using the PersonGroup Person - Create API
- Adding faces to persons using the PersonGroup Person - Add Face API

Related Topics

- How to Identify Faces in Image
- How to Detect Faces in Image
- How to use the large-scale feature

Example: How to use the large-scale feature

9/18/2018 • 8 minutes to read • Edit Online

This guide is an advanced article on code migration to scale up from existing PersonGroup and FaceList to LargePersonGroup and LargeFaceList respectively. This guide demonstrates the migration process with assumption of knowing basic usage of PersonGroup and FaceList. For getting familiar with basic operations, please see other tutorials such as How to identify faces in images,

The Face API recently released two features to enable large-scale scenarios, LargePersonGroup and LargeFaceList, collectively referred to as Large-scale operations. LargePersonGroup can contain up to 1,000,000 persons each with a maximum of 248 faces, and LargeFaceList can hold up to 1,000,000 faces.

The large-scale operations are similar to the conventional PersonGroup and FaceList, but have some notable differences due to the new architecture. This guide demonstrates the migration process with assumption of knowing basic usage of PersonGroup and FaceList. The samples are written in C# using the Face API client library.

To enable Face search performance for Identification and FindSimilar in the large scale, you need to introduce a Train operation to pre-process the LargeFaceList and LargePersonGroup. The training time varies from seconds to about half an hour depending on the actual capacity. During the training period, it is still possible to perform Identification and FindSimilar if a successful training is done before. However, the drawback is that the new added persons/faces will not appear in the result until a new post migration to large-scale training is completed.

Concepts

If you are not familiar with the following concepts in this guide, the definitions can be found in the glossary:

- LargePersonGroup: A collection of Persons with capacity up to 1,000,000.
- LargeFaceList: A collection of Faces with capacity up to 1,000,000.
- Train: A pre-process to ensure Identification/FindSimilar performance.
- Identification: Identify one or more faces from a PersonGroup or LargePersonGroup.
- FindSimilar: Search similar faces from a FaceList or LargeFaceList.

Step 1: Authorize the API call

When using the Face API client library, the subscription key and subscription endpoint are passed in through the constructor of the FaceServiceClient class. For example:

```
string SubscriptionKey = "<Subscription Key>";
// Use your own subscription endpoint corresponding to the subscription key.
string SubscriptionRegion = "https://westcentralus.api.cognitive.microsoft.com/face/v1.0/";
FaceServiceClient FaceServiceClient = new FaceServiceClient(SubscriptionKey, SubscriptionRegion);
```

The subscription key with corresponding endpoint can be obtained from the Marketplace page of your Azure portal. See Subscriptions.

Step 2: Code Migration in action

This section only focuses on migrating PersonGroup/FaceList implementation to LargePersonGroup/LargeFaceList. Although LargePersonGroup/LargeFaceList differs from PersonGroup/FaceList in design and internal implementation, the API interfaces are similar for back-compatibility.

Data migration is not supported, you have to recreate the LargePersonGroup/LargeFaceList instead.

Step 2.1: Migrate PersonGroup to LargePersonGroup

The migration from PersonGroup to LargePersonGroup is smooth as they share exactly the same group-level operations.

For PersonGroup/Person related implementation, it is only necessary to change the API paths or SDK class/module to LargePersonGroup and LargePersonGroup Person.

In terms of data migration, see How to Add Faces for reference.

Step 2.2: Migrate FaceList to LargeFaceList

FACELIST APIS	LARGEFACELIST APIS
Create	Create
Delete	Delete
Get	Get
List	List
Update	Update
-	Train
-	Get Training Status

The preceding table is a comparison of list-level operations between FaceList and LargeFaceList. As is shown, LargeFaceList comes with new operations, Train, and Get Training Status, when compared with FaceList. Getting the LargeFaceList trained is a precondition of FindSimilar operation while there is no Train required for FaceList. The following snippet is a helper function to wait for the training of a LargeFaceList.

```
/// <summary>
/// Helper function to train LargeFaceList and wait for finish.
/// </summary>
/// <remarks>
/// The time interval can be adjusted considering the following factors:
/// - The training time which depends on the capacity of the LargeFaceList.
/// - The acceptable latency for getting the training status.
/// - The call frequency and cost.
///
/// Estimated training time for LargeFaceList in different scale:
/// -     1,000 faces cost about  1 to  2 seconds.
/// -    10,000 faces cost about  5 to 10 seconds.
/// -   100,000 faces cost about  1 to  2 minutes.
/// - 1,000,000 faces cost about 10 to 30 minutes.
/// </remarks>
/// <param name="largeFaceListId">The Id of the LargeFaceList for training.</param>
/// <param name="timeIntervalInMilliseconds">The time interval for getting training status in milliseconds.
</param>
/// <returns>A task of waiting for LargeFaceList training finish.</returns>
private static async Task TrainLargeFaceList(
    string largeFaceListId,
    int timeIntervalInMilliseconds = 1000)
{
    // Trigger a train call.
    await FaceServiceClient.TrainLargeFaceListAsync(largeFaceListId);

    // Wait for training finish.
    while (true)
    {
        Task.Delay(timeIntervalInMilliseconds).Wait();
        var status = await FaceServiceClient.GetLargeFaceListTrainingStatusAsync(largeFaceListId);

        if (status.Status == Status.Running)
        {
            continue;
        }
        else if (status.Status == Status.Succeeded)
        {
            break;
        }
        else
        {
            throw new Exception("The train operation is failed!");
        }
    }
}
```

Previously, a typical usage of FaceList with adding faces and FindSimilar would be

```
// Create a FaceList.
const string FaceListId = "myfacelistid_001";
const string FaceListName = "MyFaceListDisplayName";
const string ImageDir = @"/path/to/FaceList/images";
FaceServiceClient.CreateFaceListAsync(FaceListId, FaceListName).Wait();

// Add Faces to the FaceList.
Parallel.ForEach(
    Directory.GetFiles(ImageDir, "*.jpg"),
    async imagePath =>
        {
            using (Stream stream = File.OpenRead(imagePath))
            {
                await FaceServiceClient.AddFaceToFaceListAsync(FaceListId, stream);
            }
        });

// Perform FindSimilar.
const string QueryImagePath = @"/path/to/query/image";
var results = new List<SimilarPersistedFace[]>();
using (Stream stream = File.OpenRead(QueryImagePath))
{
    var faces = FaceServiceClient.DetectAsync(stream).Result;
    foreach (var face in faces)
    {
        results.Add(await FaceServiceClient.FindSimilarAsync(face.FaceId, FaceListId, 20));
    }
}
```

When migrating it to LargeFaceList, it should become

```
// Create a LargeFaceList.
const string LargeFaceListId = "mylargefacelistid_001";
const string LargeFaceListName = "MyLargeFaceListDisplayName";
const string ImageDir = @"/path/to/FaceList/images";
FaceServiceClient.CreateLargeFaceListAsync(LargeFaceListId, LargeFaceListName).Wait();

// Add Faces to the LargeFaceList.
Parallel.ForEach(
    Directory.GetFiles(ImageDir, "*.jpg"),
    async imagePath =>
        {
            using (Stream stream = File.OpenRead(imagePath))
            {
                await FaceServiceClient.AddFaceToLargeFaceListAsync(LargeFaceListId, stream);
            }
        });

// Train() is newly added operation for LargeFaceList.
// Must call it before FindSimilarAsync() to ensure the newly added faces searchable.
await TrainLargeFaceList(LargeFaceListId);

// Perform FindSimilar.
const string QueryImagePath = @"/path/to/query/image";
var results = new List<SimilarPersistedFace[]>();
using (Stream stream = File.OpenRead(QueryImagePath))
{
    var faces = FaceServiceClient.DetectAsync(stream).Result;
    foreach (var face in faces)
    {
        results.Add(await FaceServiceClient.FindSimilarAsync(face.FaceId, largeFaceListId: LargeFaceListId));
    }
}
```

As is shown above, the data management and the FindSimilar part are almost the same. The only exception is that a fresh pre-processing Train operation must complete in the LargeFaceList before FindSimilar works.

Step 3: Train Suggestions

Although the Train operation speeds up the FindSimilar and Identification, the training time suffers especially when coming to large scale. The estimated training time in different scales is listed in the following table:

SCALE (FACES OR PERSONS)	ESTIMATED TRAINING TIME
1,000	1-2 s
10,000	5-10 s
100,000	1 - 2 min
1,000,000	10 - 30 min

To better utilize the large-scale feature, some strategies are recommended to take into consideration.

Step 3.1: Customize Time Interval

As is shown in the `TrainLargeFaceList()`, there is a `timeIntervalInMilliseconds` to delay the infinite training status checking process. For LargeFaceList with more faces, using a larger interval reduces the call counts and cost. The time interval should be customized according to the expected capacity of the LargeFaceList.

Same strategy also applies to LargePersonGroup. For example, when training a LargePersonGroup with 1,000,000 persons, the `timeIntervalInMilliseconds` could be 60,000 (a.k.a. 1-minute interval).

Step 3.2 Small-scale buffer

Persons/Faces in LargePersonGroup/LargeFaceList are searchable only after being trained. In a dynamic scenario, new persons/faces are constantly added and need to be immediately searchable, yet training could take longer than desired. To mitigate this problem, you can use an extra small-scale LargePersonGroup/LargeFaceList as a buffer only for the newly added entries. This buffer takes shorter time to train because of much smaller size and the immediate search on this temporary buffer should work. Use this buffer in combination with training on the master LargePersonGroup/LargeFaceList by executing the master training on a more sparse interval, for example, in the mid-night, and daily.

An example workflow:

1. Create a master LargePersonGroup/LargeFaceList (master collection) and a buffer LargePersonGroup/LargeFaceList (buffer collection). The buffer collection is only for newly added Persons/Faces.
2. Add new Persons/Faces to both the master collection and the buffer collection.
3. Only train the buffer collection with a short time interval to ensure the newly added entries taking effect.
4. Call Identification/FindSimilar against both the master collection and the buffer collection, and merge the results.
5. When buffer collection size increases to a threshold or at a system idle time, create a new buffer collection and trigger the train on master collection.
6. Delete the old buffer collection after the finish of training on the master collection.

Step 3.3 Standalone Training

If a relatively long latency is acceptable, it is not necessary to trigger the Train operation right after adding new data. Instead, the Train operation can be split from the main logic and triggered regularly. This strategy is suitable for dynamic scenarios with acceptable latency, and can be applied to static scenarios to further reduce the Train frequency.

Suppose there is a `TrainLargePersonGroup` function similar to the `TrainLargeFaceList`. A typical implementation of the standalone Training on LargePersonGroup by invoking the `Timer` class in `System.Timers` would be:

```
private static void Main()
{
    // Create a LargePersonGroup.
    const string LargePersonGroupId = "mylargepersongroupid_001";
    const string LargePersonGroupName = "MyLargePersonGroupDisplayName";
    FaceServiceClient.CreateLargePersonGroupAsync(LargePersonGroupId, LargePersonGroupName).Wait();

    // Setup a standalone training at regular intervals.
    const int TimeIntervalForStatus = 1000 * 60; // 1 minute interval for getting training status.
    const double TimeIntervalForTrain = 1000 * 60 * 60; // 1 hour interval for training.
    var trainTimer = new Timer(TimeIntervalForTrain);
    trainTimer.Elapsed += (sender, args) => TrainTimerOnElapsed(LargePersonGroupId, TimeIntervalForStatus);
    trainTimer.AutoReset = true;
    trainTimer.Enabled = true;

    // Other operations like creating persons, adding faces and Identification except for Train.
    // ...
}

private static void TrainTimerOnElapsed(string largePersonGroupId, int timeIntervalInMilliseconds)
{
    TrainLargePersonGroup(largePersonGroupId, timeIntervalInMilliseconds).Wait();
}
```

More information about data management and identification-related implementations, see How to Add Faces and How to Identify Faces in Image.

Summary

In this guide, you have learned how to migrate the existing PersonGroup/FaceList code (not data) to the LargePersonGroup/LargeFaceList:

- LargePersonGroup and LargeFaceList works similar to the PersonGroup/FaceList, except Train operation is required by LargeFaceList.
- Take proper train strategy to dynamic data update for large-scale dataset.

Related Topics

- How to Identify Faces in Image
- How to Add Faces

Example: How to Analyze Videos in Real-time

9/18/2018 • 7 minutes to read • Edit Online

This guide will demonstrate how to perform near-real-time analysis on frames taken from a live video stream. The basic components in such a system are:

- Acquire frames from a video source
- Select which frames to analyze
- Submit these frames to the API
- Consume each analysis result that is returned from the API call

These samples are written in C# and the code can be found on GitHub here: https://github.com/Microsoft/Cognitive-Samples-VideoFrameAnalysis.

The Approach

There are multiple ways to solve the problem of running near-real-time analysis on video streams. We will start by outlining three approaches in increasing levels of sophistication.

A Simple Approach

The simplest design for a near-real-time analysis system is an infinite loop, where in each iteration we grab a frame, analyze it, and then consume the result:

```
while (true)
{
    Frame f = GrabFrame();
    if (ShouldAnalyze(f))
    {
        AnalysisResult r = await Analyze(f);
        ConsumeResult(r);
    }
}
```

If our analysis consisted of a lightweight client-side algorithm, this approach would be suitable. However, when our analysis is happening in the cloud, the latency involved means that an API call might take several seconds, during which time we are not capturing images, and our thread is essentially doing nothing. Our maximum frame-rate is limited by the latency of the API calls.

Parallelizing API Calls

While a simple single-threaded loop makes sense for a lightweight client-side algorithm, it doesn't fit well with the latency involved in cloud API calls. The solution to this problem is to allow the long-running API calls to execute in parallel with the frame-grabbing. In C#, we could achieve this using Task-based parallelism, for example:

```
while (true)
{
    Frame f = GrabFrame();
    if (ShouldAnalyze(f))
    {
        var t = Task.Run(async () =>
        {
            AnalysisResult r = await Analyze(f);
            ConsumeResult(r);
        }
    }
}
```

This launches each analysis in a separate Task, which can run in the background while we continue grabbing new frames. This avoids blocking the main thread while waiting for an API call to return, however we have lost some of the guarantees that the simple version provided -- multiple API calls might occur in parallel, and the results might get returned in the wrong order. This could also cause multiple threads to enter the ConsumeResult() function simultaneously, which could be dangerous, if the function is not thread-safe. Finally, this simple code does not keep track of the Tasks that get created, so exceptions will silently disappear. Thus, the final ingredient for us to add is a "consumer" thread that will track the analysis tasks, raise exceptions, kill long-running tasks, and ensure the results get consumed in the correct order, one at a time.

A Producer-Consumer Design

In our final "producer-consumer" system, we have a producer thread that looks very similar to our previous infinite loop. However, instead of consuming analysis results as soon as they are available, the producer simply puts the tasks into a queue to keep track of them.

```
// Queue that will contain the API call tasks.
var taskQueue = new BlockingCollection<Task<ResultWrapper>>();

// Producer thread.
while (true)
{
    // Grab a frame.
    Frame f = GrabFrame();

    // Decide whether to analyze the frame.
    if (ShouldAnalyze(f))
    {
        // Start a task that will run in parallel with this thread.
        var analysisTask = Task.Run(async () =>
        {
            // Put the frame, and the result/exception into a wrapper object.
            var output = new ResultWrapper(f);
            try
            {
                output.Analysis = await Analyze(f);
            }
            catch (Exception e)
            {
                output.Exception = e;
            }
            return output;
        }

        // Push the task onto the queue.
        taskQueue.Add(analysisTask);
    }
}
```

We also have a consumer thread, that is taking tasks off the queue, waiting for them to finish, and either displaying the result or raising the exception that was thrown. By using the queue, we can guarantee that results get consumed one at a time, in the correct order, without limiting the maximum frame-rate of the system.

```
// Consumer thread.
while (true)
{
    // Get the oldest task.
    Task<ResultWrapper> analysisTask = taskQueue.Take();

    // Await until the task is completed.
    var output = await analysisTask;

    // Consume the exception or result.
    if (output.Exception != null)
    {
        throw output.Exception;
    }
    else
    {
        ConsumeResult(output.Analysis);
    }
}
```

Implementing the Solution

Getting Started

To get your app up and running as quickly as possible, we have implemented the system described above, intending it to be flexible enough to implement many scenarios, while being easy to use. To access the code, go to https://github.com/Microsoft/Cognitive-Samples-VideoFrameAnalysis.

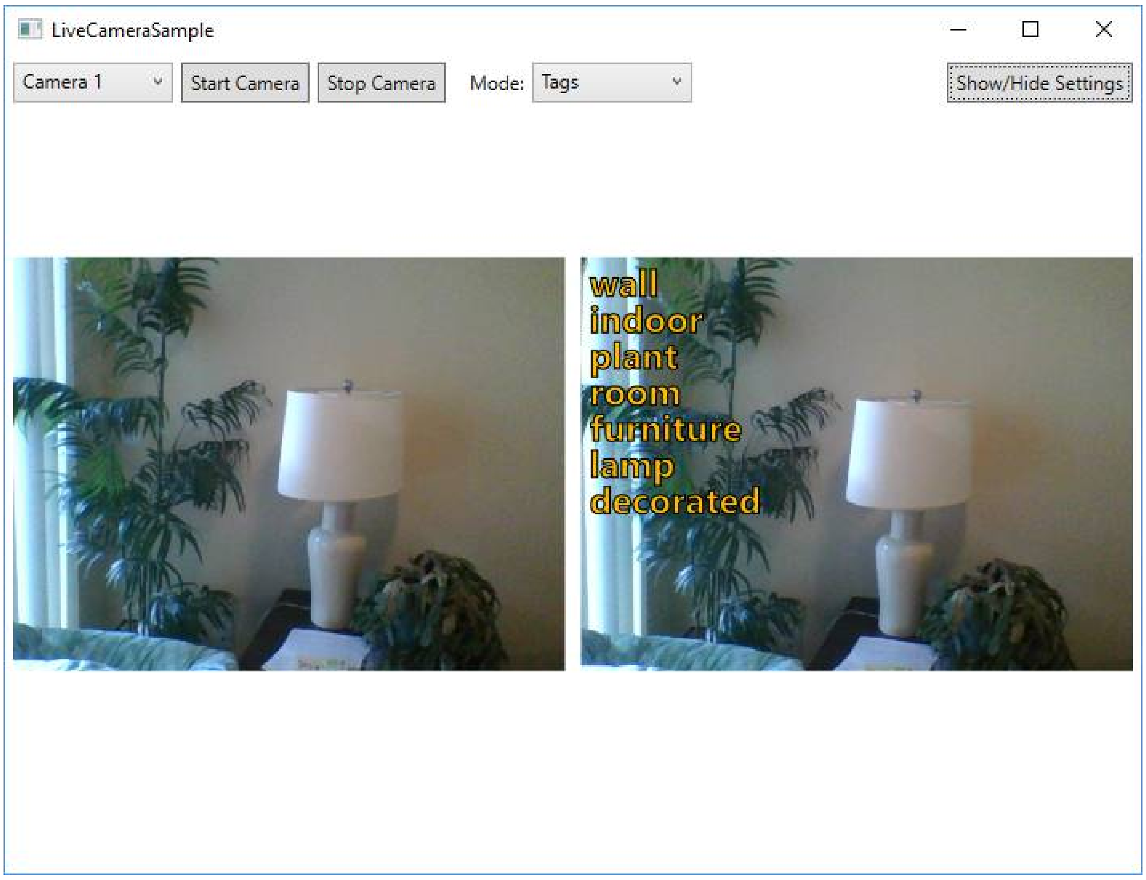
The library contains the class FrameGrabber, which implements the producer-consumer system discussed above to process video frames from a webcam. The user can specify the exact form of the API call, and the class uses events to let the calling code know when a new frame is acquired, or a new analysis result is available.

To illustrate some of the possibilities, there are two sample apps that uses the library. The first is a simple console app, and a simplified version of this is reproduced below. It grabs frames from the default webcam, and submits them to the Face API for face detection.

```
using System;
using VideoFrameAnalyzer;
using Microsoft.ProjectOxford.Face;
using Microsoft.ProjectOxford.Face.Contract;

namespace VideoFrameConsoleApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            // Create grabber, with analysis type Face[].
            FrameGrabber<Face[]> grabber = new FrameGrabber<Face[]>();

            // Create Face API Client. Insert your Face API key here.
            FaceServiceClient faceClient = new FaceServiceClient("<Subscription Key>");

            // Set up our Face API call.
            grabber.AnalysisFunction = async frame => return await
faceClient.DetectAsync(frame.Image.ToMemoryStream(".jpg"));

            // Set up a listener for when we receive a new result from an API call.
            grabber.NewResultAvailable += (s, e) =>
            {
                if (e.Analysis != null)
                    Console.WriteLine("New result received for frame acquired at {0}. {1} faces detected",
e.Frame.Metadata.Timestamp, e.Analysis.Length);
            };

            // Tell grabber to call the Face API every 3 seconds.
            grabber.TriggerAnalysisOnInterval(TimeSpan.FromMilliseconds(3000));

            // Start running.
            grabber.StartProcessingCameraAsync().Wait();

            // Wait for keypress to stop
            Console.WriteLine("Press any key to stop...");
            Console.ReadKey();

            // Stop, blocking until done.
            grabber.StopProcessingAsync().Wait();
        }
    }
}
```

The second sample app is a bit more interesting, and allows you to choose which API to call on the video frames. On the left hand side, the app shows a preview of the live video, on the right hand side it shows the most recent API result overlaid on the corresponding frame.

In most modes, there will be a visible delay between the live video on the left, and the visualized analysis on the right. This delay is the time taken to make the API call. The exception to this is in the "EmotionsWithClientFaceDetect" mode, which performs face detection locally on the client computer using OpenCV, before submitting any images to Cognitive Services. By doing this, we can visualize the detected face immediately, and then update the emotions later once the API call returns. This demonstrates the possibility of a "hybrid" approach, where some simple processing can be performed on the client, and then Cognitive Services APIs can be used to augment this with more advanced analysis when necessary.



Integrating into your codebase

To get started with this sample, follow these steps:

1. Get API keys for the Vision APIs from Subscriptions. For video frame analysis, the applicable APIs are:
   - Computer Vision API
   - Emotion API
   - Face API
2. Clone the Cognitive-Samples-VideoFrameAnalysis GitHub repo
3. Open the sample in Visual Studio 2015, build and run the sample applications:
   - For BasicConsoleSample, the Face API key is hard-coded directly in BasicConsoleSample/Program.cs.
   - For LiveCameraSample, the keys should be entered into the Settings pane of the app. They will be persisted across sessions as user data.

When you're ready to integrate, **simply reference the VideoFrameAnalyzer library from your own projects.**

Developer Code of Conduct

As with all the Cognitive Services, Developers developing with our APIs and samples are required to follow the "Developer Code of Conduct for Microsoft Cognitive Services."

The image, voice, video or text understanding capabilities of VideoFrameAnalyzer uses Microsoft Cognitive Services. Microsoft will receive the images, audio, video, and other data that you upload (via this app) and may use them for service improvement purposes. We ask for your help in protecting the people whose data your app sends to Microsoft Cognitive Services.

Summary

In this guide, you learned how to run near-real-time analysis on live video streams using the Face, Computer Vision, and Emotion APIs, and how you can use our sample code to get started. You can get started building your app with free API keys at the Microsoft Cognitive Services sign-up page.

Please feel free to provide feedback and suggestions in the GitHub repository, or for more broad API feedback, on our UserVoice site.

Related Topics

- How to Identify Faces in Image
- How to Detect Faces in Image

Connecting to Cognitive Services Face API by using Connected Services in Visual Studio

9/18/2018 • 5 minutes to read • Edit Online

By using the Cognitive Services Face API, you can detect, analyze, organize, and tag faces in photos.

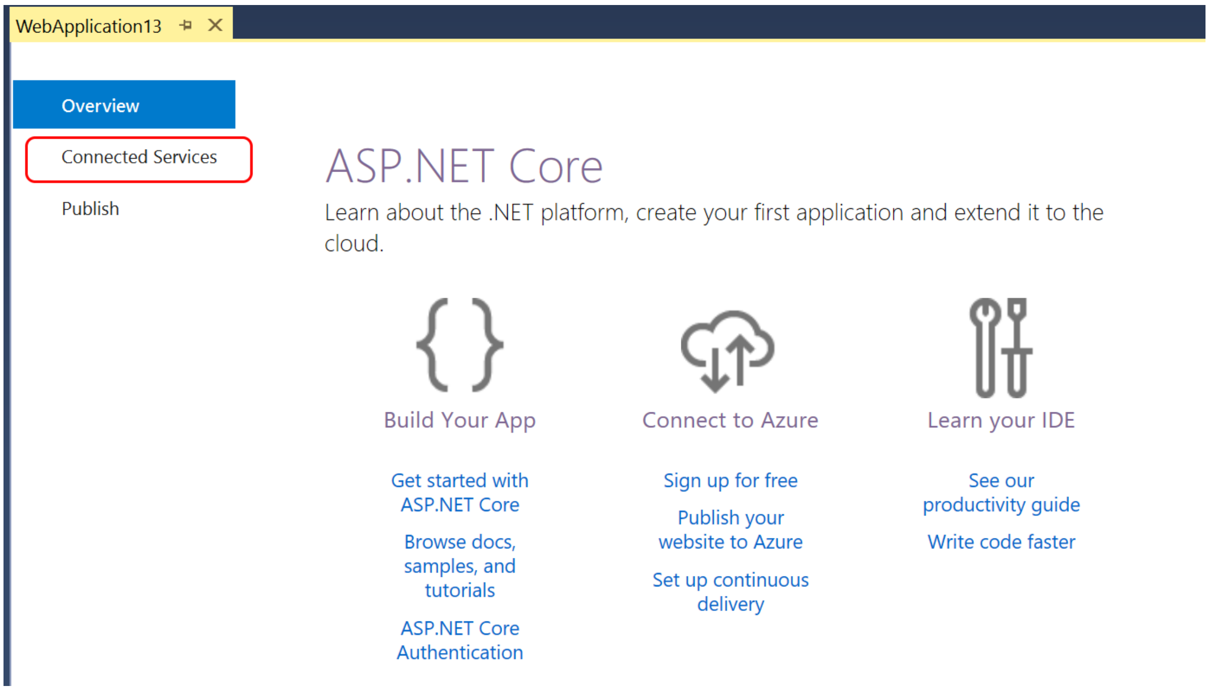
This article and its companion articles provide details for using the Visual Studio Connected Service feature for Cognitive Services Face API. The capability is available in both Visual Studio 2017 15.7 or later, with the Cognitive Services extension installed.

Prerequisites

- **An Azure subscription**. If you do not have one, you can sign up for a free account.
- **Visual Studio 2017 version 15.7** with the **Web Development** workload installed. Download it now.

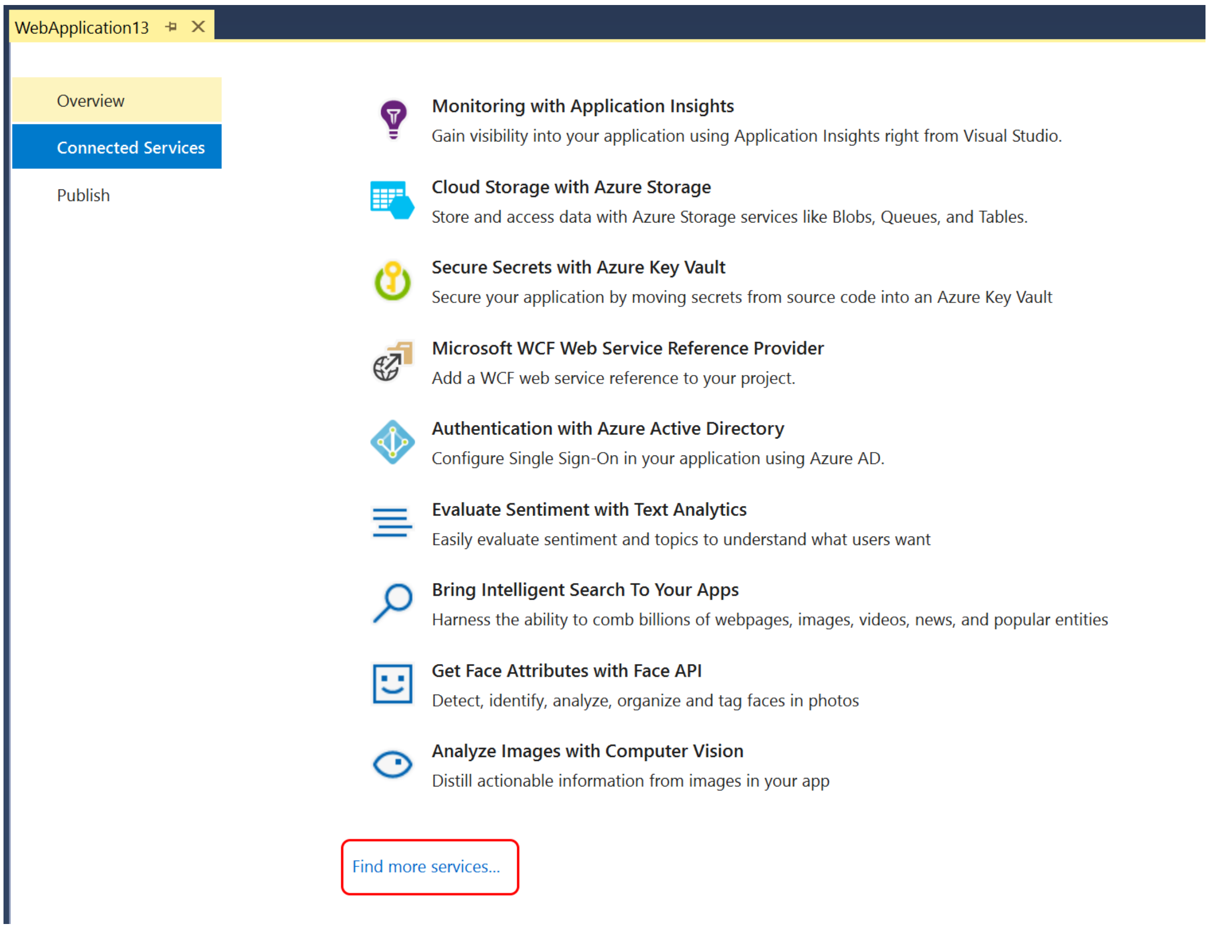
Install the Cognitive Services VSIX Extension

1. With your web project open in Visual Studio, choose the **Connected Services** tab. The tab is available on the welcome page that appears when you open a new project. If you don't see the tab, select **Connected Services** in your project in Solution Explorer.


WebApplication13
Overview
Connected Services
Publish
ASP.NET Core
Build Your App
Get started with ASP.NET Core
Browse docs, samples, and tutorials
ASP.NET Core Authentication
Connect to Azure
Sign up for free
Publish your website to Azure
Set up continuous delivery
Learn your IDE
See our productivity guide
Write code faster

Learn about the .NET platform, create your first application and extend it to the cloud.

2. Scroll down to the bottom of the list of services, and select **Find more services**.


WebApplication13

Overview

Connected Services

Publish

Monitoring with Application Insights
Gain visibility into your application using Application Insights right from Visual Studio.

Cloud Storage with Azure Storage
Store and access data with Azure Storage services like Blobs, Queues, and Tables.

Secure Secrets with Azure Key Vault
Secure your application by moving secrets from source code into an Azure Key Vault

Microsoft WCF Web Service Reference Provider
Add a WCF web service reference to your project.

Authentication with Azure Active Directory
Configure Single Sign-On in your application using Azure AD.

Evaluate Sentiment with Text Analytics
Easily evaluate sentiment and topics to understand what users want

Bring Intelligent Search To Your Apps
Harness the ability to comb billions of webpages, images, videos, news, and popular entities

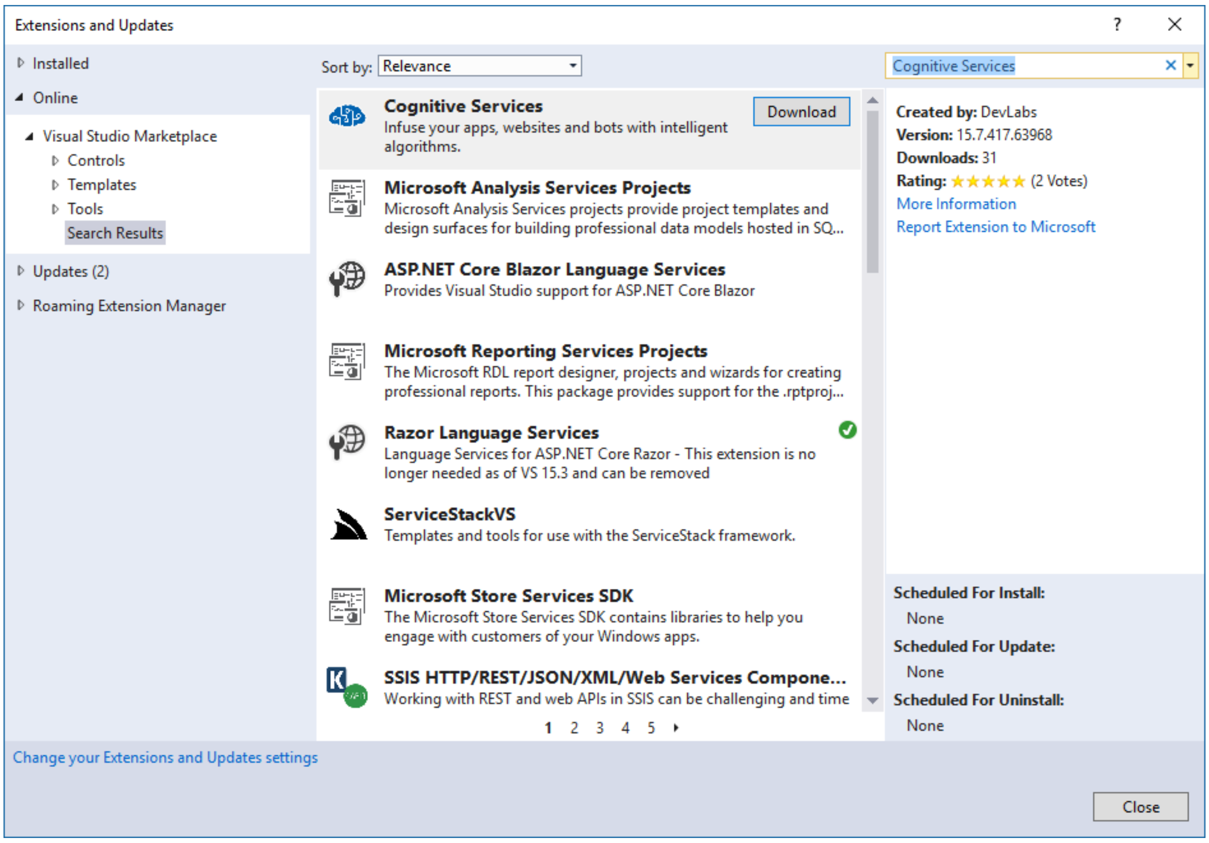
Get Face Attributes with Face API
Detect, identify, analyze, organize and tag faces in photos

Analyze Images with Computer Vision
Distill actionable information from images in your app

Find more services...

The **Extensions and Updates** dialog box appears.

3. In the **Extensions and Updates** dialog box, search for **Cognitive Services**, and then download and install the Cognitive Services VSIX package.


Extensions and Updates

Installed
Online
Visual Studio Marketplace
Controls
Templates
Tools
Search Results
Updates (2)
Roaming Extension Manager

Sort by: Relevance

Cognitive Services

Cognitive Services — Download
Infuse your apps, websites and bots with intelligent algorithms.

Microsoft Analysis Services Projects
Microsoft Analysis Services projects provide project templates and design surfaces for building professional data models hosted in SQ...

ASP.NET Core Blazor Language Services
Provides Visual Studio support for ASP.NET Core Blazor

Microsoft Reporting Services Projects
The Microsoft RDL report designer, projects and wizards for creating professional reports. This package provides support for the .rptproj...

Razor Language Services
Language Services for ASP.NET Core Razor - This extension is no longer needed as of VS 15.3 and can be removed

ServiceStackVS
Templates and tools for use with the ServiceStack framework.

Microsoft Store Services SDK
The Microsoft Store Services SDK contains libraries to help you engage with customers of your Windows apps.

SSIS HTTP/REST/JSON/XML/Web Services Compone...
Working with REST and web APIs in SSIS can be challenging and time

1 2 3 4 5

Created by: DevLabs
Version: 15.7.417.63968
Downloads: 31
Rating: ★★★★★ (2 Votes)
More Information
Report Extension to Microsoft

Scheduled For Install:
None
Scheduled For Update:
None
Scheduled For Uninstall:
None

Change your Extensions and Updates settings

Close

Installing an extension requires a restart of the integrated development environment (IDE).

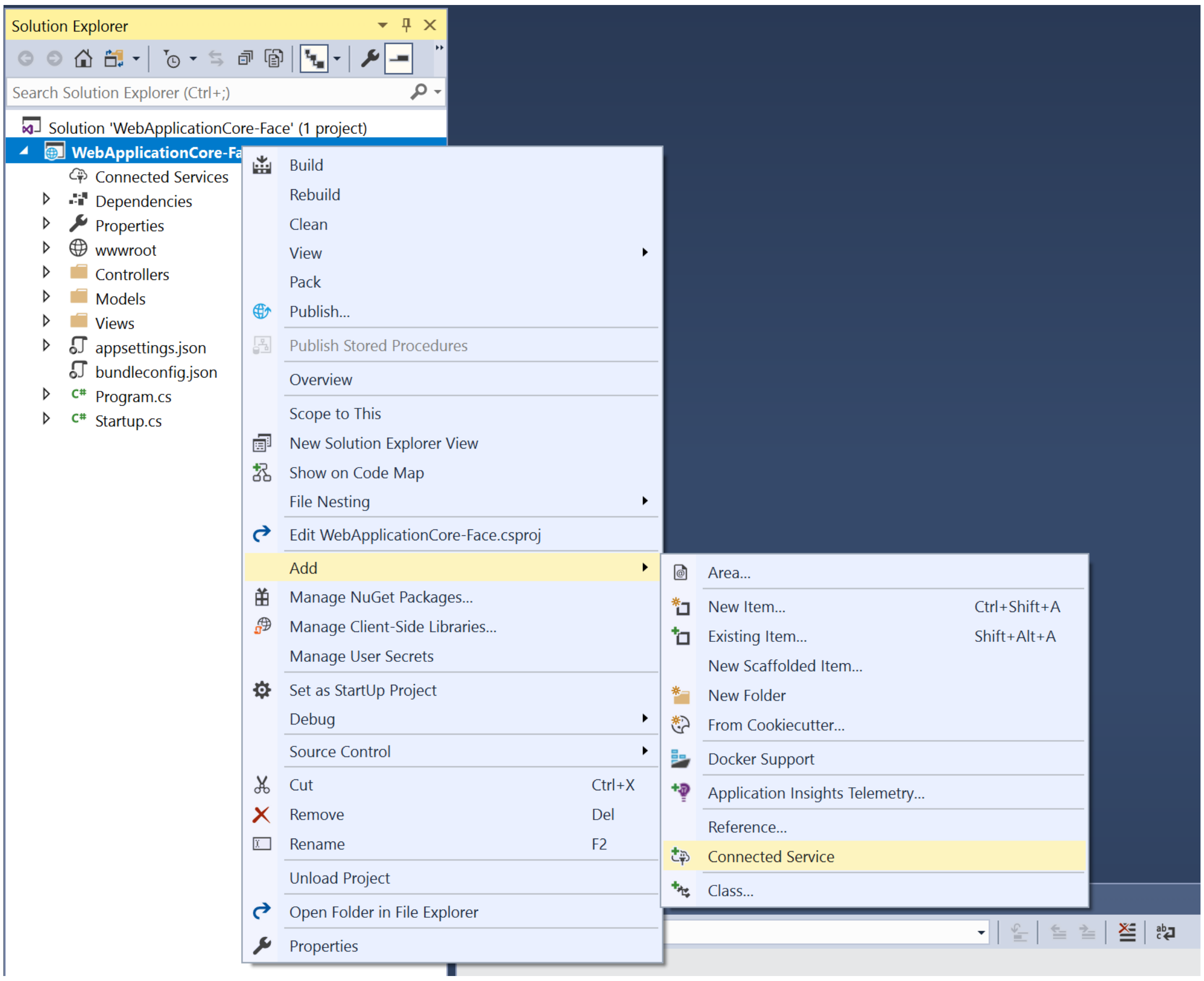
4. Restart Visual Studio. The extension installs when you close Visual Studio, and is available next time you

launch the IDE.

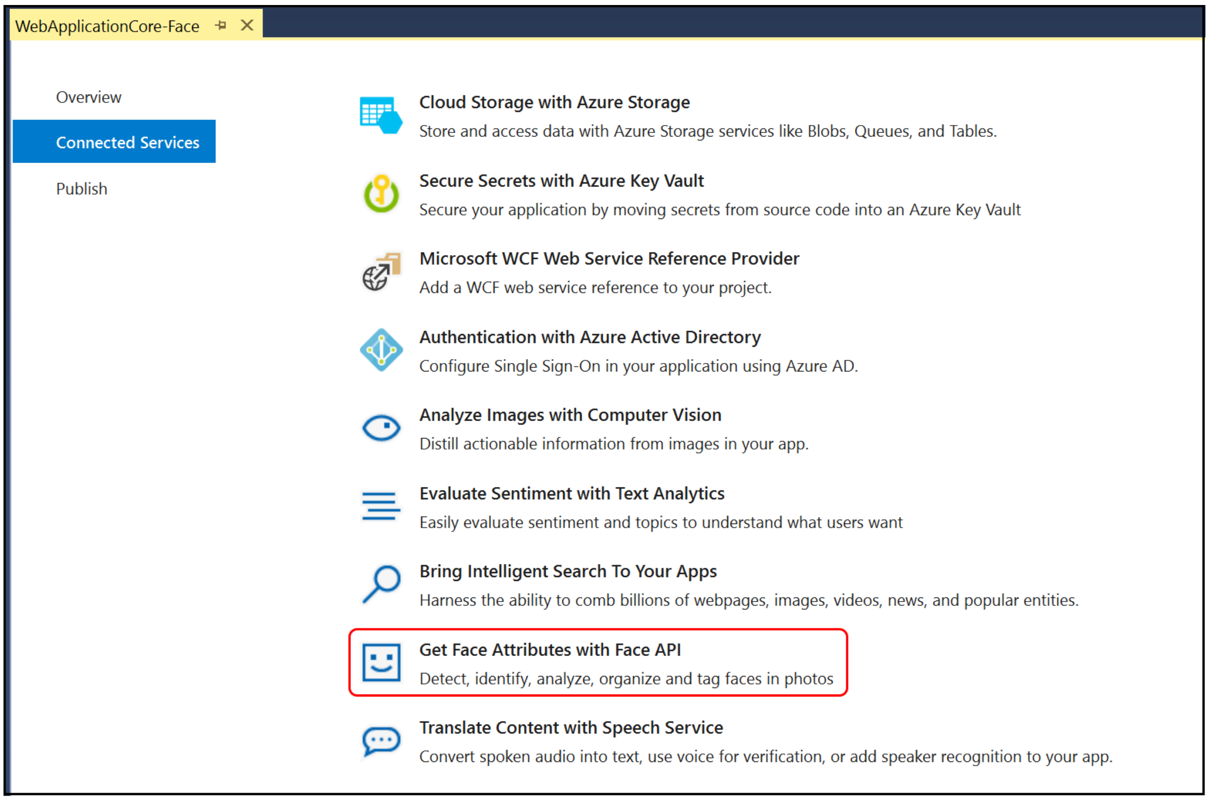
Create a project and add support for Cognitive Services Face API

1. Create a new ASP.NET Core web project. Use the Empty project template.

2. In **Solution Explorer**, choose **Add** > **Connected Service**. The Connected Service page appears with services you can add to your project.



3. In the menu of available services, choose **Cognitive Services Face API**.



Cloud Storage with Azure Storage
Store and access data with Azure Storage services like Blobs, Queues, and Tables.

Secure Secrets with Azure Key Vault
Secure your application by moving secrets from source code into an Azure Key Vault

Microsoft WCF Web Service Reference Provider
Add a WCF web service reference to your project.

Authentication with Azure Active Directory
Configure Single Sign-On in your application using Azure AD.

Analyze Images with Computer Vision
Distill actionable information from images in your app.

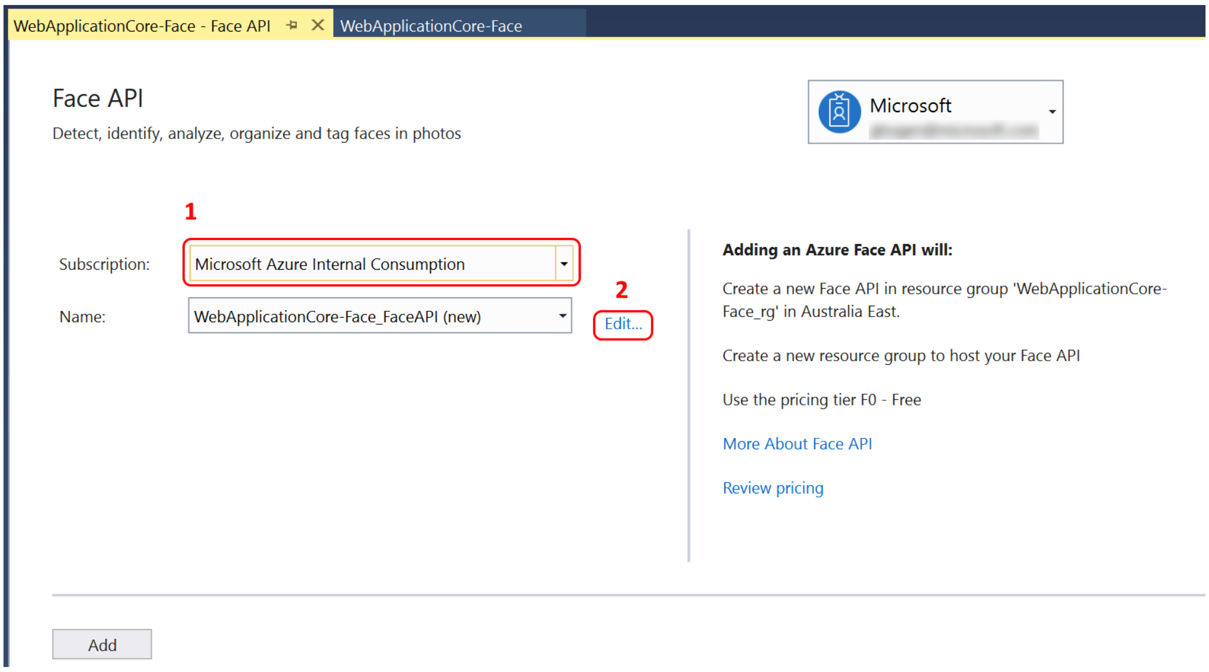
Evaluate Sentiment with Text Analytics
Easily evaluate sentiment and topics to understand what users want

Bring Intelligent Search To Your Apps
Harness the ability to comb billions of webpages, images, videos, news, and popular entities.

Get Face Attributes with Face API
Detect, identify, analyze, organize and tag faces in photos

Translate Content with Speech Service
Convert spoken audio into text, use voice for verification, or add speaker recognition to your app.

If you've signed into Visual Studio, and have an Azure subscription associated with your account, a page appears with a dropdown list with your subscriptions.



Face API
Detect, identify, analyze, organize and tag faces in photos

Subscription: Microsoft Azure Internal Consumption
Name: WebApplicationCore-Face_FaceAPI (new) Edit...

Adding an Azure Face API will:

Create a new Face API in resource group 'WebApplicationCore-Face_rg' in Australia East.

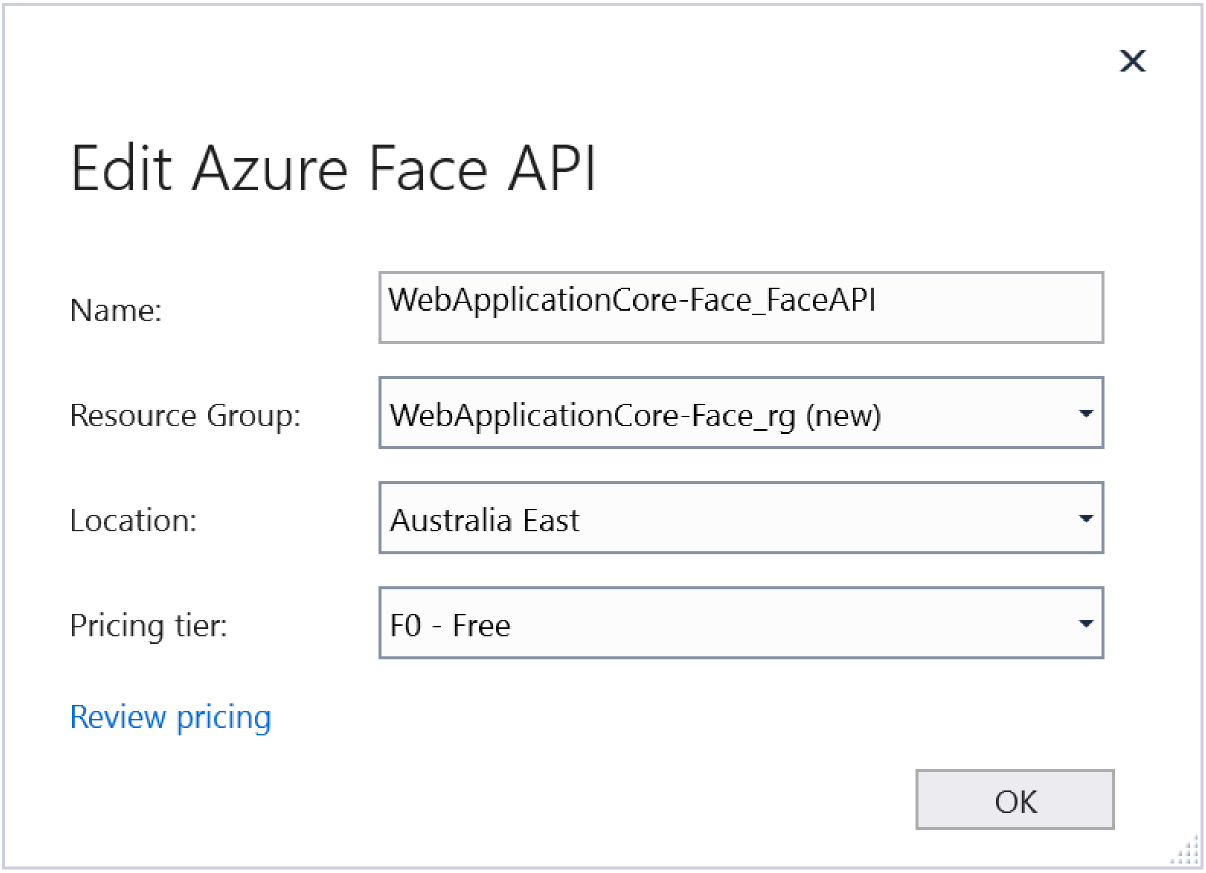
Create a new resource group to host your Face API

Use the pricing tier F0 - Free

More About Face API

Review pricing

4. Select the subscription you want to use, and then choose a name for the Face API, or choose the Edit link to modify the automatically generated name, choose the resource group, and the Pricing Tier.



Edit Azure Face API

Name: WebApplicationCore-Face_FaceAPI

Resource Group: WebApplicationCore-Face_rg (new)

Location: Australia East

Pricing tier: F0 - Free

Follow the link for details on the pricing tiers.

5. Choose Add to add supported for the Connected Service. Visual Studio modifies your project to add the NuGet packages, configuration file entries, and other changes to support a connection the Face API.

Use the Face API to detect attributes of faces in an image

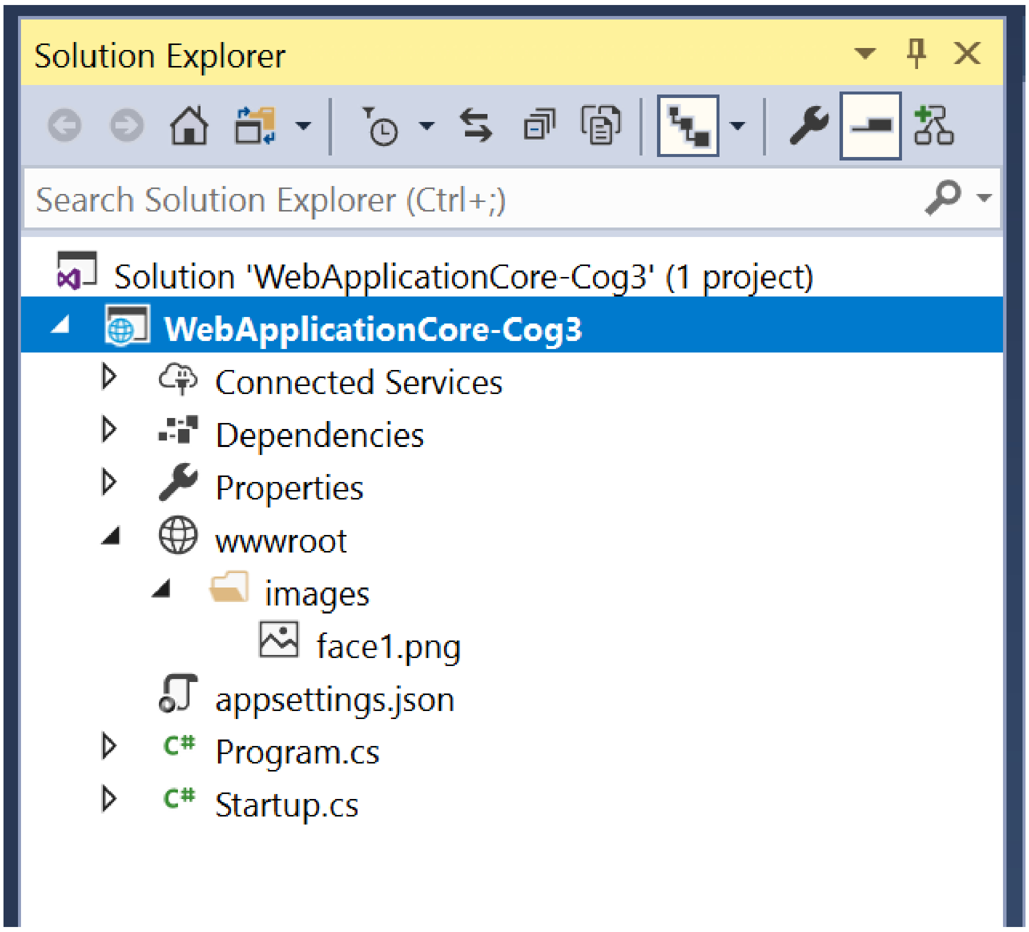
1. Add the following using statements in Startup.cs.

```
using System.IO;
using System.Text;
using Microsoft.Extensions.Configuration;
using System.Net.Http;
using System.Net.Http.Headers;
```

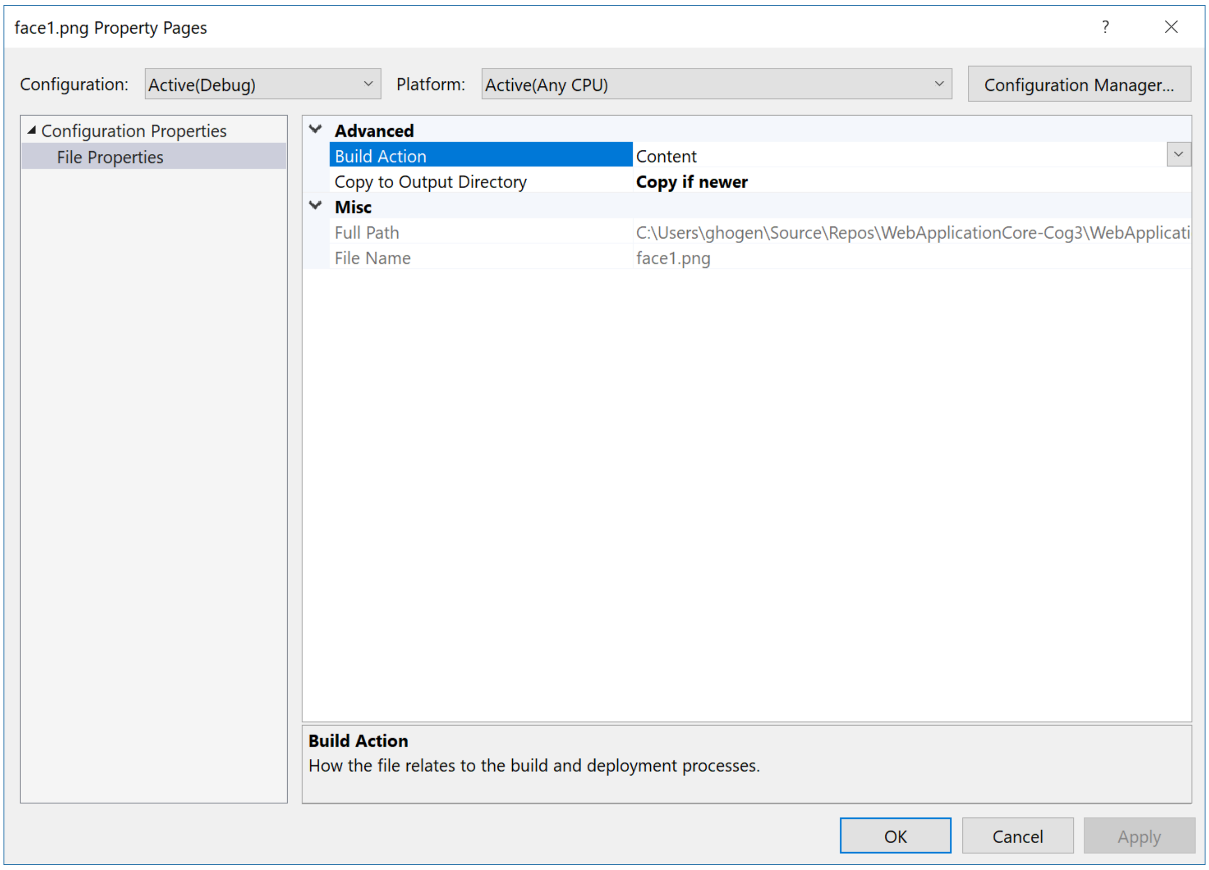
2. Add a configuration field, and add a constructor that initializes the configuration field in Startup class to enable Configuration in your program.

```
    private IConfiguration configuration;

    public Startup(IConfiguration configuration)
    {
        this.configuration = configuration;
    }
```

3. In the wwwroot folder in your project, add an images folder, and add an image file to your wwwroot folder. As an example, you can use one of the images on this Face API page. Right click on one of the images, save to your local hard drive, then in Solution Explorer, right-click on the images folder, and choosee **Add** > **Existing Item** to add it to your project. Your project should look something like this in Solution Explorer:



4. Right-click on the image file, choose Properties, and then choose **Copy if newer**.


face1.png Property Pages

Configuration: Active(Debug) Platform: Active(Any CPU) Configuration Manager...

Configuration Properties
File Properties

Advanced	
Build Action	Content
Copy to Output Directory	**Copy if newer**
Misc	
Full Path	C:\Users\ghogen\Source\Repos\WebApplicationCore-Cog3\WebApplicati
File Name	face1.png

Build Action
How the file relates to the build and deployment processes.

5. Replace the Configure method with the following code to access the Face API and test an image. Change the imagePath string to the correct path and filename for your face image.

```
 // This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
        // TODO: Change this to your image's path on your site.
        string imagePath = @"images/face1.png";

        // Enable static files such as image files.
        app.UseStaticFiles();

        string faceApiKey = this.configuration["FaceAPI_ServiceKey"];
        string faceApiEndPoint = this.configuration["FaceAPI_ServiceEndPoint"];

        HttpClient client = new HttpClient();

        // Request headers.
        client.DefaultRequestHeaders.Add("Ocp-Apim-Subscription-Key", faceApiKey);

        // Request parameters. A third optional parameter is "details".
        string requestParameters =
"returnFaceId=true&returnFaceLandmarks=false&returnFaceAttributes=age,gender,headPose,smile,facialHair,g
lasses,emotion,hair,makeup,occlusion,accessories,blur,exposure,noise";

        // Assemble the URI for the REST API Call.
        string uri = faceApiEndPoint + "/detect?" + requestParameters;

        // Request body. Posts an image you've added to your site's images folder.
        var fileInfo = env.WebRootFileProvider.GetFileInfo(imagePath);
        var byteData = GetImageAsByteArray(fileInfo.PhysicalPath);

        string contentStringFace = string.Empty;
        using (ByteArrayContent content = new ByteArrayContent(byteData))
        {
            // This example uses content type "application/octet-stream".
            // The other content types you can use are "application/json" and "multipart/form-data".
            content.Headers.ContentType = new MediaTypeHeaderValue("application/octet-stream");

            // Execute the REST API call.
            var response = client.PostAsync(uri, content).Result;

            // Get the JSON response.
            contentStringFace = response.Content.ReadAsStringAsync().Result;
        }

        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }

        app.Run(async (context) =>
        {
            await context.Response.WriteAsync($"<p><b>Face Image:</b></p>");
            await context.Response.WriteAsync($"<div><img src=\"" + imagePath + "\" /></div>");
            await context.Response.WriteAsync($"<p><b>Face detection API results:</b></p>");
            await context.Response.WriteAsync("<p>");
            await context.Response.WriteAsync(JsonPrettyPrint(contentStringFace));
            await context.Response.WriteAsync("<p>");
        });
    }
```

The code in this step constructs a HTTP request with a call to the Face REST API, using the key you added when you added the connected service.

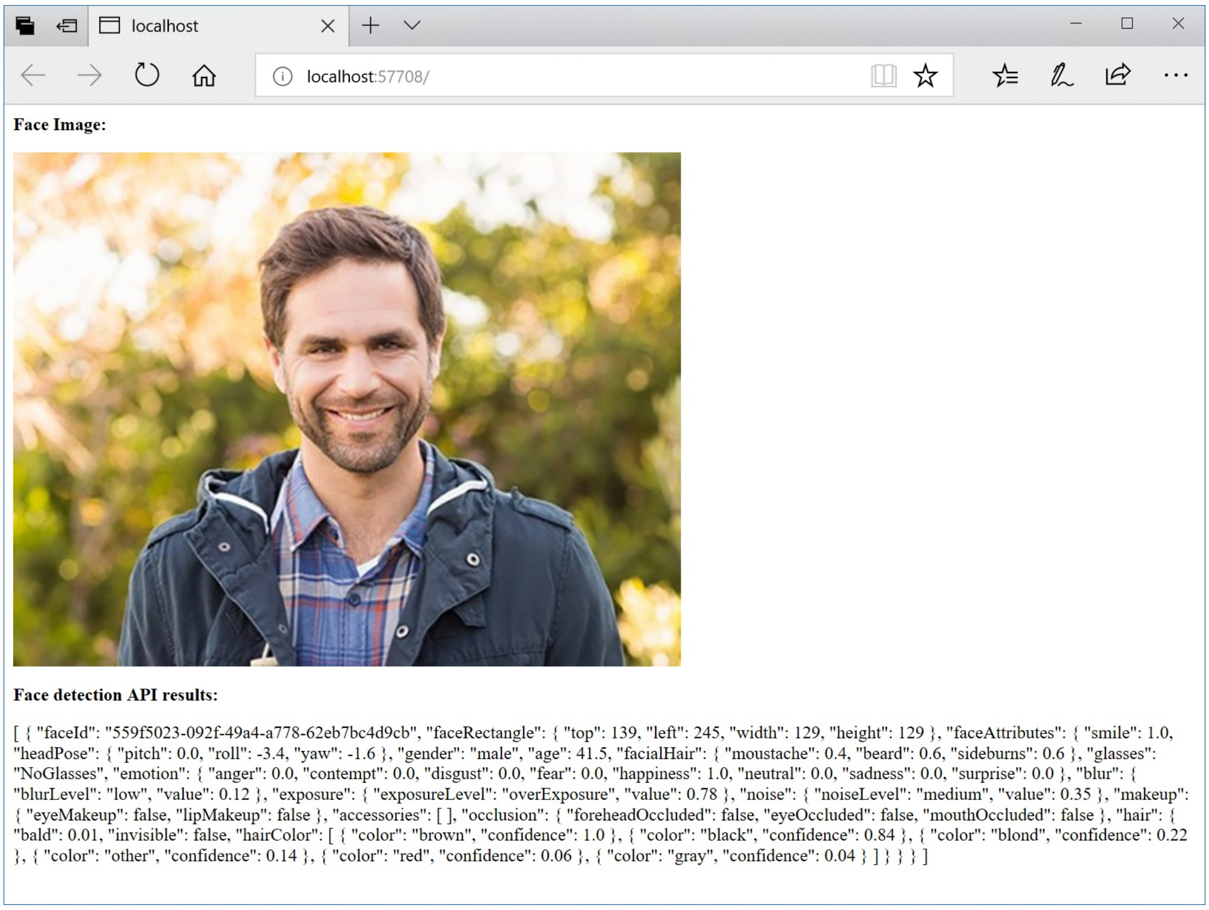
6. Add the helper functions GetImageAsByteArray and JsonPrettyPrint.

```
    /// <summary>
    /// Returns the contents of the specified file as a byte array.
```

```
        /// Returns the contents of the specified file as a byte array.
        /// </summary>
        /// <param name="imageFilePath">The image file to read.</param>
        /// <returns>The byte array of the image data.</returns>
        static byte[] GetImageAsByteArray(string imageFilePath)
        {
            FileStream fileStream = new FileStream(imageFilePath, FileMode.Open, FileAccess.Read);
            BinaryReader binaryReader = new BinaryReader(fileStream);
            return binaryReader.ReadBytes((int)fileStream.Length);
        }

        /// <summary>
        /// Formats the given JSON string by adding line breaks and indents.
        /// </summary>
        /// <param name="json">The raw JSON string to format.</param>
        /// <returns>The formatted JSON string.</returns>
        static string JsonPrettyPrint(string json)
        {
            if (string.IsNullOrEmpty(json))
                return string.Empty;

            json = json.Replace(Environment.NewLine, "").Replace("\t", "");

            string INDENT_STRING = "    ";
            var indent = 0;
            var quoted = false;
            var sb = new StringBuilder();
            for (var i = 0; i < json.Length; i++)
            {
                var ch = json[i];
                switch (ch)
                {
                    case '{':
                    case '[':
                        sb.Append(ch);
                        if (!quoted)
                        {
                            sb.AppendLine();
                        }
                        break;
                    case '}':
                    case ']':
                        if (!quoted)
                        {
                            sb.AppendLine();
                        }
                        sb.Append(ch);
                        break;
                    case '"':
                        sb.Append(ch);
                        bool escaped = false;
                        var index = i;
                        while (index > 0 && json[--index] == '\\')
                            escaped = !escaped;
                        if (!escaped)
                            quoted = !quoted;
                        break;
                    case ',':
                        sb.Append(ch);
                        if (!quoted)
                        {
                            sb.AppendLine();
                        }
                        break;
                    case ':':
                        sb.Append(ch);
                        if (!quoted)
                            sb.Append(" ");
                        break;
                    default:
```

```
                default:
                    sb.Append(ch);
                    break;
                }
            }
            return sb.ToString();
        }
```

7. Run the web application and see what Face API found in the image.



Face Image:

Face detection API results:

```
[ { "faceId": "559f5023-092f-49a4-a778-62eb7bc4d9cb", "faceRectangle": { "top": 139, "left": 245, "width": 129, "height": 129 }, "faceAttributes": { "smile": 1.0,
"headPose": { "pitch": 0.0, "roll": -3.4, "yaw": -1.6 }, "gender": "male", "age": 41.5, "facialHair": { "moustache": 0.4, "beard": 0.6, "sideburns": 0.6 }, "glasses":
"NoGlasses", "emotion": { "anger": 0.0, "contempt": 0.0, "disgust": 0.0, "fear": 0.0, "happiness": 1.0, "neutral": 0.0, "sadness": 0.0, "surprise": 0.0 }, "blur": {
"blurLevel": "low", "value": 0.12 }, "exposure": { "exposureLevel": "overExposure", "value": 0.78 }, "noise": { "noiseLevel": "medium", "value": 0.35 }, "makeup":
{ "eyeMakeup": false, "lipMakeup": false }, "accessories": [ ], "occlusion": { "foreheadOccluded": false, "eyeOccluded": false, "mouthOccluded": false }, "hair": {
"bald": 0.01, "invisible": false, "hairColor": [ { "color": "brown", "confidence": 1.0 }, { "color": "black", "confidence": 0.84 }, { "color": "blond", "confidence": 0.22
}, { "color": "other", "confidence": 0.14 }, { "color": "red", "confidence": 0.06 }, { "color": "gray", "confidence": 0.04 } ] } } } ]
```

Clean up resources

When no longer needed, delete the resource group. This deletes the cognitive service and related resources. To delete the resource group through the portal:

1. Enter the name of your resource group in the Search box at the top of the portal. When you see the resource group used in this QuickStart in the search results, select it.
2. Select **Delete resource group**.
3. In the **TYPE THE RESOURCE GROUP NAME:** box type in the name of the resource group and select **Delete**.

Next steps

Learn more about the Face API by reading the Face API Documentation.

10/19/2018 • 2 minutes to read • Edit Online

title: API Reference - Face API titleSuffix: Azure Cognitive Services description: API reference provides information about the Person Management, LargePersonGroup/PersonGroup Management, LargeFaceList/FaceList Management, and Face Algorithms APIs. services: cognitive-services author: SteveMSFT manager: cgronlun

ms.service: cognitive-services ms.component: face-api ms.topic: reference ms.date: 03/01/2018

ms.author: sbowles

API Reference

The Microsoft Face API is a cloud-based API that provides the most advanced algorithms for face detection and recognition.

Face APIs cover the following categories:

- Face Algorithm APIs: Covers core functions such as Detection, Find Similar, Verification, Identification, and Group.
- FaceList Management APIs: Used to manage a FaceList for Find Similar.
- LargePersonGroup Person Management APIs: Used to manage LargePersonGroup Person Faces for Identification.
- LargePersonGroup Management APIs: Used to manage a LargePersonGroup dataset for Identification.
- LargeFaceList Management APIs: Used to manage a LargeFaceList for Find Similar.
- PersonGroup Person Management APIs: Used to manage PersonGroup Person Faces for Identification.
- PersonGroup Management APIs: Used to manage a PersonGroup dataset for Identification.

Glossary

9/18/2018 • 11 minutes to read • Edit Online

A

Attributes

Attributes are optional in the detection results, such as age, gender, head pose, facial hair, smiling. They can be obtained from the detection API by specifying the query parameters: returnFaceAttributes. Attributes give extra information regarding selected faces; in addition to the face ID and the rectangle.

For more details, please refer to the guide Face - Detect.

Age (Attribute)

Age is one of the attributes that describes the age of a particular face. The age attribute is optional in the detection results, and can be controlled with a detection request by specifying the returnFaceAttributes parameter.

For more details, please refer to the guide Face - Detect.

B

C

Candidate

Candidates are essentially identification results (e.g. identified persons and level of confidence in detections). A candidate is represented by the PersonID and confidence, indicating that the person is identified with a high level of confidence.

For more details, please refer to the guide Face - Identify.

Confidence

Confidence is a measurement revealing the similarity between faces or Person in numerical values –which is used in identification, and verification to indicate the similarities of searched, identified and verified results.

For more details, please refer to the following guides: Face - Find Similar, Face - Identify, Face - Verify.

D

Detection/Face Detection

Face detection is the action of locating faces in images. Users can upload an image or specify an image URL in the request. The detected faces are returned with face IDs indicating a unique identity in Face API. The rectangles indicate the face locations in the image in pixels, as well as the optional attributes for each face such as age, gender, head pose, facial hair and smiling.

For more details, please refer to the guide Face - Detect.

E

F

Face

Face is a unified term for the results derived from Face API related with detected faces. Ultimately, face is represented by a unified identity (Face ID), a specified region in images (Face Rectangle), and extra face related attributes, such as age, gender, landmarks and head pose. Additionally, faces can be returned from detection.

For more details, please refer to the guide Face - Detect.

Face API

Face API is a cloud-based API that provides the most advanced algorithms for face detection and recognition. The main functionality of Face API can be divided into two categories: face detection with attributes, and face recognition.

For more details, please refer to the following guides: Face API Overview, Face - Detect, Face - Find Similar, Face - Group, Face - Identify, Face - Verify.

Face Attributes/Facial Attributes

Please see Attributes.

Face ID

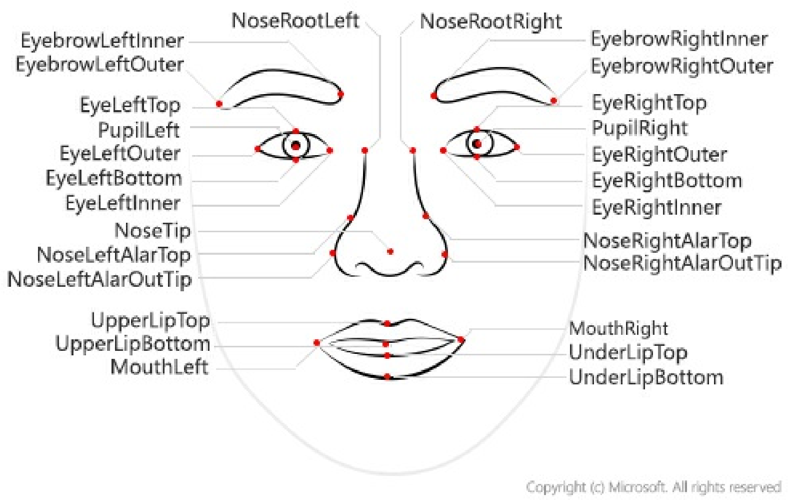
Face ID is derived from the detection results, in which a string represents a face in Face API.

For more details, please refer to the guide Face - Detect.

Face Landmarks/Facial Landmarks

Landmarks are optional in the detection results; which are semantic facial points, such as the eyes, nose and mouth (illustrated in following figure). Landmarks can be controlled with a detection request by the Boolean number returnFaceLandmarks. If returnFaceLandmarks is set as true, the returned faces will have landmark attributes.

For more details, please refer to the guide Face - Detect.



Face Rectangle

Face rectangle is derived from the detection results, which is an upright rectangle (left, top, width, height) in images in pixels. The top-left corner of a face (left, top), besides the width and height, indicates face sizes in x and y axes respectively.

For more details, please refer to the guide Face - Detect.

Facial Hair (Attribute)

Facial hair is one of the attributes used to describe the facial hair length of the available faces. The facial hair attribute is optional in the detection results, and can be controlled with a detection request by returnFaceAttributes. If returnFaceAttributes contains 'facialHair', the returned faces will have facial hair attributes.

For more details, please refer to the guide Face - Detect.

FaceList

FaceList is a collection of PersistedFace and is the unit of Find Similar. A FaceList comes with a FaceList ID, as well as other attributes such as Name and User Data.

For more details, please refer to the following guides: FaceList - Create, FaceList - Get.

FaceList ID

FaceList ID is a user provided string used as an identifier of a FaceList. The FaceList ID must be unique within the subscription.

For more details, please refer to the following guides: FaceList - Create, FaceList - Get.

Find Similar

This API is used to search/query similar faces based on a collection of faces. Query faces and face collections are represented as face IDs or FceList ID/LargeFaceList ID in the request. Returned results are searched similar faces, represented by face IDs or PersistedFace IDs.

For more details, please refer to the following guides: Face - Find Similar, LargeFaceList - Create, FaceList - Create.

G

Gender (Attribute)

Gender is one of the attributes used to describe the genders of the available faces. The gender attribute is optional in the detection results, and can be controlled with a detection request by returnFaceAttributes. If returnfaceAttributes contains 'gender', the returned faces will have gender attributes.

For more details, please refer to the guide Face - Detect.

Grouping

Face grouping is the grouping of a collection of faces according to facial similarities. Face collections are indicated by the face ID collections in the request. As a result of grouping, similar faces are grouped together as groups, and faces that are not similar to any other face are merged together as a messy group. There is at the most, one messy group in the grouping result.

For more details, please refer to the guide Face - Group.

Groups

Groups are derived from the grouping results. Each group contains a collection of similar faces, where faces are indicated by face IDs.

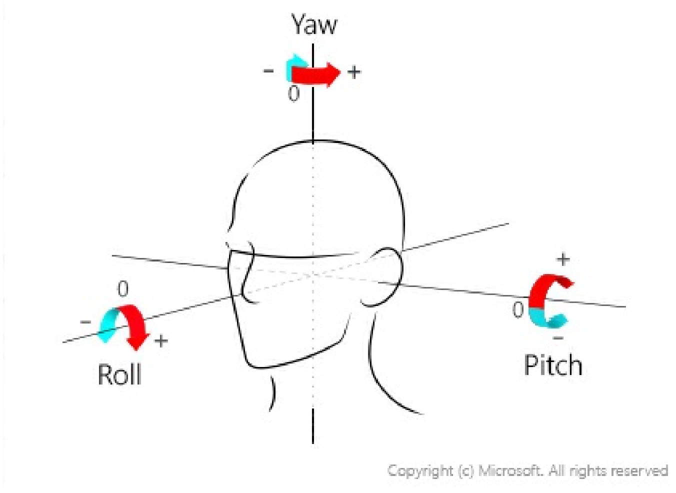
For more details, please refer to the guide Face - Group.

H

Head Pose (Attribute)

Head pose is one of the attributes that represents face orientation in 3D space according to roll, pitch and yaw angles, as shown in following figure. The value ranges of roll and yaw are [-180, 180] and [-90, 90] in degrees. In the current version, the pitch value returned from detection is always 0. The head pose attribute is optional in the detection results, and can be controlled with a detection request by the returnFaceAttributes parameter. If returnFaceAttributes parameter contains 'headPose', the returned faces will have head pose attributes.

For more details, please refer to the guide Face - Detect.



I

Identification

Identification is to identify one or more faces from a LargePersonGroup/PersonGroup. A PersonGroup/LargePersonGroup is a collection of Persons. Faces and the LargePersonGroup/PersonGroup are represented by face IDs and LargePersonGroup IDs/PersonGroup IDs respectively in the request. Identified results are candidates, represented by Persons with confidence. Multiple faces in the input are considered separately, and each face will have its own identified result.

> **NOTE**
>
> The LargePersonGroup/PersonGroup should be trained successfully before identification. If the LargePersonGroup/PersonGroup is not trained, or the training status is not shown as 'succeeded' (i.e. 'running', 'failed', or 'timeout'), the request response is 400.

For more details, please refer to the following guides: Face - Identify, LargePersonGroup Person - Create, LargePersonGroup - Create, LargePersonGroup - Train, PersonGroup Person - Create, PersonGroup - Create, PersonGroup - Train.

IsIdentical

IsIdentical is a Boolean field of verification results indicating whether two faces belong to the same person.

For more details, please refer to the guide Face - Verify.

J

K

L

Landmarks

Please see face landmarks.

LargeFaceList

LargeFaceList is a collection of PersistedFace and is the unit of Find Similar. A LargeFaceList comes with a LargeFaceList ID, as well as other attributes such as Name and User Data.

For more details, please refer to the following guides: LargeFaceList - Create, LargeFaceList - Get, LargeFaceList - List Face.

LargeFaceList ID

LargeFaceList ID is a user provided string used as an identifier of a LargeFaceList. The LargeFaceList ID must be unique within the subscription.

For more details, please refer to the following guides: LargeFaceList - Create, LargeFaceList - Get.

LargePersonGroup

LargePersonGroup is a collection of Persons and is the unit of Identification. A LargePersonGroup comes with a LargePersonGroup ID, as well as other attributes such as Name and User Data.

For more details, please refer to the following guides: LargePersonGroup - Create, LargePersonGroup - Get, LargePersonGroup Person - List.

LargePersonGroup ID

LargePersonGroup ID is a user provided string used as an identifier of a LargePersonGroup. The LargePersonGroup ID must be unique within the subscription.

For more details, please refer to the following guides: LargePersonGroup - Create, LargePersonGroup - Get.

M

Messy group

Messy group is derived from the grouping results; which contains faces not similar to any other face. Each face in a messy group is indicated by the face ID.

For more details, please refer to the guide Face - Group.

N

Name (Person)

Name is a user friendly descriptive string for Person. Unlike the Person ID, the name of people can be duplicated within a group.

For more details, please refer to the following guides: LargePersonGroup Person - Create, LargePersonGroup Person - Get, PersonGroup Person - Create, PersonGroup Person - Get.

Name (LargePersonGroup/PersonGroup)

Name is also a user friendly descriptive string for LargePersonGroup/PersonGroup. Unlike the LargePersonGroup ID/PersonGroup ID, the name of LargePersonGroups/PersonGroups can be duplicated within a subscription.

For more details, please refer to the following guides: LargePersonGroup - Create, LargePersonGroup - Get, PersonGroup - Create, PersonGroup - Get.

O

P

PersistedFace

PersistedFace is a data structure in Face API. PersistedFace comes with a PersistedFace ID, as well as other attributes such as Name, and User Data.

For more details, please refer to the following guides: LargeFaceList - Add Face, FaceList - Add Face, LargePersonGroup Person - Add Face, PersonGroup Person - Add Face.

Person ID

Person ID is generated when a PersistedFace is created successfully. A string is created to represent this Face in Face API.

For more details, please refer to the following guides: LargeFaceList - Add Face, FaceList - Add Face, LargePersonGroup Person - Add Face, PersonGroup Person - Add Face.

Person

Person is a data structure managed in Face API. Person comes with a Person ID, as well as other attributes such as Name, a collection of PersistedFace, and User Data.

For more details, please refer to the following guides: LargePersonGroup Person - Create, LargePersonGroup Person - Get, PersonGroup Person - Create, PersonGroup Person - Get.

Person ID

Person ID is generated when a Person is created successfully. A string is created to represent this person in Face API.

For more details, please refer to the following guides: LargePersonGroup Person - Create, LargePersonGroup Person - Get, PersonGroup Person - Create, PersonGroup Person - Get.

PersonGroup

PersonGroup is a collection of Persons and is the unit of Identification. A PersonGroup comes with a PersonGroup ID, as well as other attributes such as Name and User Data.

For more details, please refer to the following guides: PersonGroup - Create, PersonGroup - Get, PersonGroup Person - List.

PersonGroup ID

PersonGroup ID is a user provided string used as an identifier of a PersonGroup. The group ID must be unique within the subscription.

For more details, please refer to the following guides: PersonGroup - Create, PersonGroup - Get.

Pose (Attribute)

Please see Head Pose.

Q

R

Recognition

Recognition is a popular application area for face technologies, such as Find Similar, Grouping, Identify,verifying two faces same or not.

For more details, please refer to the following guides: Face - Find Similar, Face - Group, Face - Identify, Face - Verify.

Rectangle (Face)

Please see face rectangle.

S

Smile (Attribute)

Smile is one of the attributes used to describe the smile expression of the available faces. The smile attribute is optional in the detection results, and can be controlled with a detection request by returnFaceAttributes. If returnFaceAttributes contains 'smile', the returned faces will have smile attributes.

For more details, please refer to the guide Face - Detect.

Similar Face Searching

Please see Find Similar.

Status (Train)

Status is a string used to describe the procedure for Training LargeFaceList/LargePersonGroups/PersonGroups, including 'notstarted', 'running', 'succeeded', 'failed'.

For more details, please refer to the guide LargeFaceList - Train, LargePersonGroup - Train, PersonGroup - Train.

Subscription key

Subscription key is a string that you need to specify as a query string parameter in order to invoke any Face API. The subscription key can be found in My Subscriptions page after you sign in to Microsoft Cognitive Services portal. There will be two keys associated with each subscription: one primary key and one secondary key. Both can be used to invoke API identically. You need to keep the subscription keys secure, and you can regenerate subscription keys at any time from My Subscriptions page as well.

T

Train (LargeFaceList/LargePersonGroup/PersonGroup)

This API is used to pre-process the LargeFaceList/LargePersonGroup/PersonGroup to ensure the Find Similar/Identification performance. If the training is not operated, or the Training Status is not shown as succeeded, the identification for this PersonGroup will result in failure.

For more details, please refer to the following guides: LargeFaceList - Train, LargePersonGroup - Train, PersonGroup - Train, Face - Identify.

U

UserData/User Data

User data is extra information associated with Person and PersonGroup/LargePersonGroup. User data is set by users to make data easier to use, understand and remember.

For more details, please refer to the following guides: LargePersonGroup - Create, LargePersonGroup - Update, LargePersonGroup Person - Create, LargePersonGroup Person - Update, PersonGroup - Create, PersonGroup - Update, PersonGroup Person - Create, PersonGroup Person - Update.

V

Verification

This API is used to verify whether two faces are the same or not. Both faces are represented as face IDs in the request. Verified results contain a Boolean field (isIdentical) indicating same if true and a number field (confidence) indicating the level of confidence.

For more details, please refer to the guide Face - Verify.

W

X

Y

Z

Face API Frequently Asked Questions

9/18/2018 • 2 minutes to read • Edit Online

If you can't find answers to your questions in this FAQ, try asking the Face API community on StackOverflow or contact Help and Support on UserVoice.

Question: What factors can reduce Face API's accuracy for Recognition, Verification, or Find Similar?

Answer: Generally it is the same cases where humans have difficulty identifying someone including;

- Obstructions blocking one or both eyes
- Harsh lighting, e.g. severe backlighting
- Changes to hair style or facial hair
- Changes due to age
- Extreme facial expressions (e.g. screaming)

Face API is often successful in challenging cases like these, but accuracy can be reduced. To make recognition more robust and address these challenges, train your Persons with photos that include a diversity of angles and lighting.

Question: I am passing the binary image data in but I get an "Invalid face image" error.

Answer: This implies that the algorithm had an issue parsing the image. Causes include:

- The supported input image formats includes JPEG, PNG, GIF(the first frame), BMP.
- Image file size should be no larger than 4MB
- The detectable face size range is 36x36 to 4096x4096 pixels. Faces out of this range will not be detected
- Some faces may not be detected due to technical challenges, e.g. very large face angles (head-pose), large occlusion. Frontal and near-frontal faces have the best results

---

Face API Release Notes

9/18/2018 • 2 minutes to read • Edit Online

This article pertains to Face API Service version 1.0.

Release changes in May 2018

- Improved `gender` attribute significantly and also improved `age`, `glasses`, `facialHair`, `hair`, `makeup` attributes. Use them through Face - Detect `returnFaceAttributes` parameter.
- Increased input image file size limit from 4 MB to 6 MB in Face - Detect, FaceList - Add Face, LargeFaceList - Add Face, PersonGroup Person - Add Face and LargePersonGroup Person - Add Face.

Release changes in March 2018

- Added Million-Scale Container: LargeFaceList and LargePersonGroup. More details in How to use the large-scale feature.
- Increased Face - Identify `maxNumOfCandidatesReturned` parameter from [1, 5] to [1, 100] and default to 10.

Release changes in May 2017

- Added `hair`, `makeup`, `accessory`, `occlusion`, `blur`, `exposure`, and `noise` attributes in Face - Detect `returnFaceAttributes` parameter.
- Supported 10K persons in a PersonGroup and Face - Identify.
- Supported pagination in PersonGroup Person - List with optional parameters: `start` and `top`.
- Supported concurrency in adding/deleting faces against different FaceLists and different persons in PersonGroup.

Release changes in March 2017

- Added `emotion` attribute in Face - Detect `returnFaceAttributes` parameter.
- Fixed the face could not be redetected with rectangle returned from Face - Detect as `targetFace` in FaceList - Add Face and PersonGroup Person - Add Face.
- Fixed the detectable face size to make sure it is strictly between 36x36 to 4096x4096 pixels.

Release changes in November 2016

- Added Face Storage Standard subscription to store additional persisted faces when using PersonGroup Person - Add Face or FaceList - Add Face for identification or similarity matching. The stored images are charged at $0.5 per 1000 faces and this rate is prorated on a daily basis. Free tier subscriptions continue to be limited to 1,000 total persons.

Release changes in October 2016

- Changed the error message of more than one face in the targetFace from 'There are more than one face in the image' to 'There is more than one face in the image' in FaceList - Add Face and PersonGroup Person - Add Face.

Release changes in July 2016

- Supported Face to Person object authentication in Face - Verify.
- Added optional `mode` parameter enabling selection of two working modes: `matchPerson` and `matchFace` in Face - Find Similar and default is `matchPerson`.
- Added optional `confidenceThreshold` parameter for user to set the threshold of whether one face belongs to

a Person object in Face - Identify.

- Added optional `start` and `top` parameters in PersonGroup - List to enable user to specify the start point and the total PersonGroups number to list.

V1.0 changes from V0

- Updated service root endpoint from `https://westus.api.cognitive.microsoft.com/face/v0/` to `https://westus.api.cognitive.microsoft.com/face/v1.0/`. Changes applied to: Face - Detect, Face - Identify, Face - Find Similar and Face - Group.

- Updated the minimal detectable face size to 36x36 pixels. Faces smaller than 36x36 pixels will not be detected.

- Deprecated the PersonGroup and Person data in Face V0. Those data cannot be accessed with the Face V1.0 service.

- Deprecated the V0 endpoint of Face API on June 30, 2016.