Contents

What is the Azure Face API?

7/5/2019 • 3 minutes to read • Edit Online

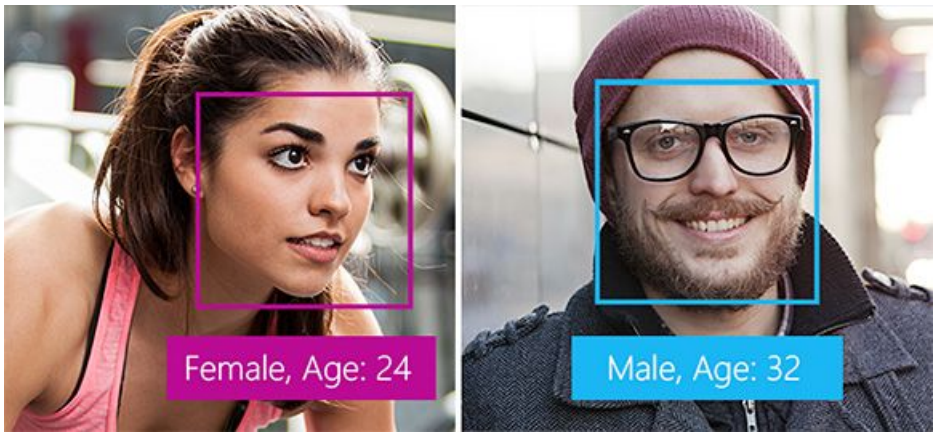
The Azure Cognitive Services Face API provides algorithms that are used to detect, recognize, and analyze human faces in images. The ability to process human face information is important in many different software scenarios. Example scenarios are security, natural user interface, image content analysis and management, mobile apps, and robotics.

The Face API provides several different functions. Each function is outlined in the following sections. Read on to learn more about them.

Face detection

The Face API detects human faces in an image and returns the rectangle coordinates of their locations. Optionally, face detection can extract a series of face-related attributes. Examples are head pose, gender, age, emotion, facial hair, and glasses.

> **NOTE**
> The face detection feature is also available through the Computer Vision API. If you want to do further operations with face data, use the Face API, which is the service discussed in this article.



For more information on face detection, see the Face detection concepts article. Also see the Detect API reference documentation.

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 capability is potentially useful in security scenarios. For more information, see the Face recognition concepts guide or the Verify API reference documentation.

Find similar faces

The Find Similar API compares a target face with a set of candidate faces to find a smaller set of faces that look similar to the target face. Two working modes, matchPerson and matchFace, are supported. The matchPerson mode returns similar faces after it filters for the same person by using the Verify API. The matchFace mode ignores the same-person filter. It returns a list of similar candidate faces that might or might not belong to the same person.

The following example shows the target face:

And these are the candidate faces:

To find four similar faces, the matchPerson mode returns a and b, which show the same person as the target face. The matchFace mode returns a, b, c, and d, exactly four candidates, even if some aren't the same person as the target or have low similarity. For more information, see the Face recognition concepts guide or the Find Similar API reference documentation.

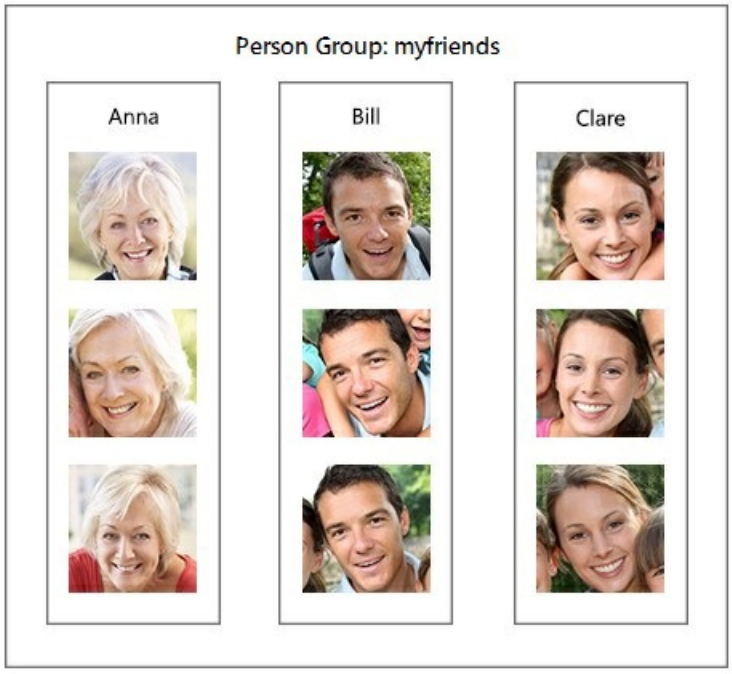
Face grouping

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. There can be several different groups for a single person. The groups are differentiated by another factor, such as expression, for example. For more information, see the Face recognition concepts guide or the Group API reference documentation.

Person identification

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

The following image shows an example of a database named `"myfriends"`. Each group can contain up to 1 million different person objects. Each person object can have up to 248 faces registered.



After you create and train a database, 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 Face recognition concepts guide or the Identify API reference documentation.

Use containers

Use the Face container to detect, recognize, and identify faces by installing a standardized Docker container closer to your data.

Sample apps

The following sample applications show a few ways to use the Face API:

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

Data privacy and security

As with all of the Cognitive Services resources, developers who use the Face service must be aware of Microsoft's policies on customer data. For more information, see the Cognitive Services page on the Microsoft Trust Center.

Next steps

Follow a quickstart to implement a face-detection scenario in code:

- Quickstart: Detect faces in an image by using the .NET SDK with C#. Other languages are available.

Quickstart: Face client library for .NET

8/29/2019 • 16 minutes to read • Edit Online

Get started with the Face client library for .NET. Follow these steps to install the package and try out the example code for basic tasks. The Face API service provides you with access to advanced algorithms for detecting and recognizing human faces in images.

Use the Face client library for .NET to:

- Authenticate the client
- Detect faces in an image
- Find similar faces
- Create and train a person group
- Identify a face
- Take a snapshot for data migration

Reference documentation | Library source code | Package (NuGet) | Samples

Prerequisites

- Azure subscription - Create one for free
- The current version of .NET Core.

Setting up

Create a Face Azure resource

Azure Cognitive Services are represented by Azure resources that you subscribe to. Create a resource for Face using the Azure portal or Azure CLI on your local machine. You can also:

- Get a trial key valid for seven days for free. After you sign up, it will be available on the Azure website.
- View your resource on the Azure portal.

After you get a key from your trial subscription or resource, create an environment variable for the key and endpoint URL, named `FACE_SUBSCRIPTION_KEY` and `FACE_ENDPOINT`, respectively.

Create a new C# application

Create a new .NET Core application in your preferred editor or IDE.

In a console window (such as cmd, PowerShell, or Bash), use the `dotnet new` command to create a new console app with the name `face-quickstart`. This command creates a simple "Hello World" C# project with a single source file: *Program.cs*.

```
dotnet new console -n face-quickstart
```

Change your directory to the newly created app folder. You can build the application with:

```
dotnet build
```

The build output should contain no warnings or errors.

```
...
Build succeeded.
 0 Warning(s)
 0 Error(s)
...
```

From the project directory, open the *Program.cs* file in your preferred editor or IDE. Add the following `using` directives:

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;

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

In the application's `Main` method, create variables for your resource's Azure endpoint and key.

```
// From your Face subscription in the Azure portal, get your subscription key and endpoint.
// Set your environment variables using the names below. Close and reopen your project for changes to take
effect.
string SUBSCRIPTION_KEY = Environment.GetEnvironmentVariable("FACE_SUBSCRIPTION_KEY");
string ENDPOINT = Environment.GetEnvironmentVariable("FACE_ENDPOINT");
```

Install the client library

Within the application directory, install the Face client library for .NET with the following command:

```
dotnet add package Microsoft.Azure.CognitiveServices.Vision.Face --version 2.5.0-preview.1
```

If you're using the Visual Studio IDE, the client library is available as a downloadable NuGet package.

Object model

The following classes and interfaces handle some of the major features of the Face .NET SDK:

NAME	DESCRIPTION
FaceClient	This class represents your authorization to use the Face service, and you need it for all Face functionality. You instantiate it with your subscription information, and you use it to produce instances of other classes.
FaceOperations	This class handles the basic detection and recognition tasks that you can do with human faces.
DetectedFace	This class represents all of the data that was detected from a single face in an image. You can use it to retrieve detailed information about the face.
FaceListOperations	This class manages the cloud-stored **FaceList** constructs, which store an assorted set of faces.

NAME	DESCRIPTION
PersonGroupPersonExtensions	This class manages the cloud-stored **Person** constructs, which store a set of faces that belong to a single person.
PersonGroupOperations	This class manages the cloud-stored **PersonGroup** constructs, which store a set of assorted **Person** objects.
ShapshotOperations	This class manages the Snapshot functionality. You can use it to temporarily save all of your cloud-based Face data and migrate that data to a new Azure subscription.

Code examples

The code snippets below show you how to do the following tasks with the Face client library for .NET:

- Authenticate the client
- Detect faces in an image
- Find similar faces
- Create and train a person group
- Identify a face
- Take a snapshot for data migration

Authenticate the client

> **NOTE**
>
> This quickstart assumes you've created environment variables for your Face key and endpoint, named `FACE_SUBSCRIPTION_KEY` and `FACE_ENDPOINT`.

In a new method, instantiate a client with your endpoint and key. Create a CognitiveServicesCredentials object with your key, and use it with your endpoint to create a FaceClient object.

```
/*
 * AUTHENTICATE
 * Uses subscription key and region to create a client.
 */
public static IFaceClient Authenticate(string endpoint, string key)
{
    return new FaceClient(new ApiKeyServiceClientCredentials(key)) { Endpoint = endpoint };
}
```

You'll likely want to call this method in the `Main` method.

```
// Authenticate.
IFaceClient client = Authenticate(ENDPOINT, SUBSCRIPTION_KEY);
```

Detect faces in an image

At the root of your class, define the following URL string. This URL points to a set of sample images.

```
// Used for all examples.
// URL for the images.
const string IMAGE_BASE_URL = "https://csdx.blob.core.windows.net/resources/Face/Images/";
```

Optionally, you can choose which AI model to use to extract data from the detected face(s). See Specify a recognition model for information on these options.

```
// Used in the Detect Faces and Verify examples.
// Recognition model 2 is used for feature extraction, use 1 to simply recognize/detect a face.
// However, the API calls to Detection that are used with Verify, Find Similar, or Identify must share the same
recognition model.
const string RECOGNITION_MODEL2 = RecognitionModel.Recognition02;
const string RECOGNITION_MODEL1 = RecognitionModel.Recognition01;
```

The final Detect operation will take a FaceClient object, an image URL, and a recognition model.

```
// Detect - get features from faces.
DetectFaceExtract(client, IMAGE_BASE_URL, RECOGNITION_MODEL2).Wait();
```

In this case, the `DetectFaceExtract` method detects faces in three of the images at the given URL and creates a list of DetectedFace objects in program memory. The list of FaceAttributeType values specifies which features to extract. See the sample code on GitHub for helper code that can print this data in an intuitive way.

```
/*
 * DETECT FACES
 * Detects features from faces and IDs them.
 */
public static async Task DetectFaceExtract(IFaceClient client, string url, string recognitionModel)
{
    Console.WriteLine("========DETECT FACES========");
    Console.WriteLine();

    // Create a list of images
    List<string> imageFileNames = new List<string>
                    {
                        "detection1.jpg",    // single female with glasses
                        // "detection2.jpg", // (optional: single man)
                        // "detection3.jpg", // (optional: single male construction worker)
                        // "detection4.jpg", // (optional: 3 people at cafe, 1 is blurred)
                        "detection5.jpg",    // family, woman child man
                        "detection6.jpg"     // elderly couple, male female
                    };

    foreach (var imageFileName in imageFileNames)
    {
        IList<DetectedFace> detectedFaces;

        // Detect faces with all attributes from image url.
        detectedFaces = await client.Face.DetectWithUrlAsync($"{url}{imageFileName}",
                returnFaceAttributes: new List<FaceAttributeType> { FaceAttributeType.Accessories,
FaceAttributeType.Age,
                FaceAttributeType.Blur, FaceAttributeType.Emotion, FaceAttributeType.Exposure,
FaceAttributeType.FacialHair,
                FaceAttributeType.Gender, FaceAttributeType.Glasses, FaceAttributeType.Hair,
FaceAttributeType.HeadPose,
                FaceAttributeType.Makeup, FaceAttributeType.Noise, FaceAttributeType.Occlusion,
FaceAttributeType.Smile },
                recognitionModel: recognitionModel);

        Console.WriteLine($"{detectedFaces.Count} face(s) detected from image `{imageFileName}`.");
```

Find similar faces

The following code takes a single detected face (source) and searches a set of other faces (target) to find matches. When it finds a match, it prints the ID of the matched face to the console.

Detect faces for comparison

First, define a second face detection method. You need to detect faces in images before you can compare them, and this detection method is optimized for comparison operations. It doesn't extract detailed face attributes like in the section above, and it uses a different recognition model.

```
private static async Task<List<DetectedFace>> DetectFaceRecognize(IFaceClient faceClient, string url, string
RECOGNITION_MODEL1)
{
    // Detect faces from image URL. Since only recognizing, use the recognition model 1.
    IList<DetectedFace> detectedFaces = await faceClient.Face.DetectWithUrlAsync(url, recognitionModel:
RECOGNITION_MODEL1);
    Console.WriteLine($"{detectedFaces.Count} face(s) detected from image `{Path.GetFileName(url)}`");
    return detectedFaces.ToList();
}
```

Find matches

The following method detects faces in a set of target images and in a single source image. Then, it compares them and finds all the target images that are similar to the source image.

```
/*
 * FIND SIMILAR
 * This example will take an image and find a similar one to it in another image.
 */
public static async Task FindSimilar(IFaceClient client, string url, string RECOGNITION_MODEL1)
{
    Console.WriteLine("========FIND SIMILAR========");
    Console.WriteLine();

    List<string> targetImageFileNames = new List<string>
                        {
                            "Family1-Dad1.jpg",
                            "Family1-Daughter1.jpg",
                            "Family1-Mom1.jpg",
                            "Family1-Son1.jpg",
                            "Family2-Lady1.jpg",
                            "Family2-Man1.jpg",
                            "Family3-Lady1.jpg",
                            "Family3-Man1.jpg"
                        };

    string sourceImageFileName = "findsimilar.jpg";
    IList<Guid?> targetFaceIds = new List<Guid?>();
    foreach (var targetImageFileName in targetImageFileNames)
    {
        // Detect faces from target image url.
        var faces = await DetectFaceRecognize(client, $"{url}{targetImageFileName}", RECOGNITION_MODEL1);
        // Add detected faceId to list of GUIDs.
        targetFaceIds.Add(faces[0].FaceId.Value);
    }

    // Detect faces from source image url.
    IList<DetectedFace> detectedFaces = await DetectFaceRecognize(client, $"{url}{sourceImageFileName}",
RECOGNITION_MODEL1);
    Console.WriteLine();

    // Find a similar face(s) in the list of IDs. Comapring only the first in list for testing purposes.
    IList<SimilarFace> similarResults = await client.Face.FindSimilarAsync(detectedFaces[0].FaceId.Value, null,
null, targetFaceIds);
```

Print matches

The following code prints the match details to the console.

```
foreach (var similarResult in similarResults)
{
    Console.WriteLine($"Faces from {sourceImageFileName} & ID:{similarResult.FaceId} are similar with
confidence: {similarResult.Confidence}.");
}
Console.WriteLine();
```

Create and train a person group

The following code creates a **PersonGroup** with six different **Person** objects. It associates each **Person** with a set of example images, and then it trains to recognize each person by their facial characteristics. **Person** and **PersonGroup** objects are used in the Verify, Identify, and Group operations.

If you haven't done so already, define the following URL string at the root of your class. This points to a set of sample images.

```
// Used for all examples.
// URL for the images.
const string IMAGE_BASE_URL = "https://csdx.blob.core.windows.net/resources/Face/Images/";
```

The code later in this section will specify a recognition model to extract data from faces, and the following snippet creates references to the available models. See Specify a recognition model for information on recognition models.

```
// Used in the Detect Faces and Verify examples.
// Recognition model 2 is used for feature extraction, use 1 to simply recognize/detect a face.
// However, the API calls to Detection that are used with Verify, Find Similar, or Identify must share the same
recognition model.
const string RECOGNITION_MODEL2 = RecognitionModel.Recognition02;
const string RECOGNITION_MODEL1 = RecognitionModel.Recognition01;
```

Create PersonGroup

Declare a string variable at the root of your class to represent the ID of the **PersonGroup** you'll create.

```
static string sourcePersonGroup = null;
```

In a new method, add the following code. This code associates the names of persons with their example images.

```
// Create a dictionary for all your images, grouping similar ones under the same key.
Dictionary<string, string[]> personDictionary =
    new Dictionary<string, string[]>
        { { "Family1-Dad", new[] { "Family1-Dad1.jpg", "Family1-Dad2.jpg" } },
          { "Family1-Mom", new[] { "Family1-Mom1.jpg", "Family1-Mom2.jpg" } },
          { "Family1-Son", new[] { "Family1-Son1.jpg", "Family1-Son2.jpg" } },
          { "Family1-Daughter", new[] { "Family1-Daughter1.jpg", "Family1-Daughter2.jpg" } },
          { "Family2-Lady", new[] { "Family2-Lady1.jpg", "Family2-Lady2.jpg" } },
          { "Family2-Man", new[] { "Family2-Man1.jpg", "Family2-Man2.jpg" } }
        };
// A group photo that includes some of the persons you seek to identify from your dictionary.
string sourceImageFileName = "identification1.jpg";
```

Next, add the following code to create a **Person** object for each person in the Dictionary and add the face data from the appropriate images. Each **Person** object is associated with the same **PersonGroup** through its unique ID string. Remember to pass the variables `client`, `url`, and `RECOGNITION_MODEL1` into this method.

```
// Create a person group.
string personGroupId = Guid.NewGuid().ToString();
sourcePersonGroup = personGroupId; // This is solely for the snapshot operations example
Console.WriteLine($"Create a person group ({personGroupId}).");
await client.PersonGroup.CreateAsync(personGroupId, personGroupId, recognitionModel: recognitionModel);
// The similar faces will be grouped into a single person group person.
foreach (var groupedFace in personDictionary.Keys)
{
    // Limit TPS
    await Task.Delay(250);
    Person person = await client.PersonGroupPerson.CreateAsync(personGroupId: personGroupId, name:
groupedFace);
    Console.WriteLine($"Create a person group person '{groupedFace}'.");

    // Add face to the person group person.
    foreach (var similarImage in personDictionary[groupedFace])
    {
        Console.WriteLine($"Add face to the person group person({groupedFace}) from image `{similarImage}`");
        PersistedFace face = await client.PersonGroupPerson.AddFaceFromUrlAsync(personGroupId, person.PersonId,
            $"{url}{similarImage}", similarImage);
    }
}
```

Train PersonGroup

Once you've extracted face data from your images and sorted it into different **Person** objects, you must train the **PersonGroup** to identify the visual features associated with each of its **Person** objects. The following code calls the asynchronous **train** method and polls the results, printing the status to the console.

```
// Start to train the person group.
Console.WriteLine();
Console.WriteLine($"Train person group {personGroupId}.");
await client.PersonGroup.TrainAsync(personGroupId);

// Wait until the training is completed.
while (true)
{
    await Task.Delay(1000);
    var trainingStatus = await client.PersonGroup.GetTrainingStatusAsync(personGroupId);
    Console.WriteLine($"Training status: {trainingStatus.Status}.");
    if (trainingStatus.Status == TrainingStatusType.Succeeded) { break; }
}
```

This **Person** group and its associated **Person** objects are now ready to be used in the Verify, Identify, or Group operations.

Identify a face

The Identify operation takes an image of a person (or multiple people) and looks to find the identity of each face in the image. It compares each detected face to a **PersonGroup**, a database of different **Person** objects whose facial features are known.

> **IMPORTANT**
>
> In order to run this example, you must first run the code in Create and train a person group. The variables used in that section—`client`, `url`, and `RECOGNITION_MODEL1`—must also be available here.

Get a test image

Notice that the code for Create and train a person group defines a variable `sourceImageFileName`. This variable

corresponds to the source image—the image that contains people to identify.

Identify faces

The following code takes the source image and creates a list of all the faces detected in the image. These are the faces that will be identified against the **PersonGroup**.

```
List<Guid> sourceFaceIds = new List<Guid>();
// Detect faces from source image url.
List<DetectedFace> detectedFaces = await DetectFaceRecognize(client, $"{url}{sourceImageFileName}",
recognitionModel);

// Add detected faceId to sourceFaceIds.
foreach (var detectedFace in detectedFaces) { sourceFaceIds.Add(detectedFace.FaceId.Value); }
```

The next code snippet calls the Identify operation and prints the results to the console. Here, the service attempts to match each face from the source image to a **Person** in the given **PersonGroup**.

```
// Identify the faces in a person group.
var identifyResults = await client.Face.IdentifyAsync(sourceFaceIds, personGroupId);

foreach (var identifyResult in identifyResults)
{
    Person person = await client.PersonGroupPerson.GetAsync(personGroupId,
identifyResult.Candidates[0].PersonId);
    Console.WriteLine($"Person '{person.Name}' is identified for face in: {sourceImageFileName} -
{identifyResult.FaceId}," +
        $" confidence: {identifyResult.Candidates[0].Confidence}.");
}
Console.WriteLine();
```

Take a snapshot for data migration

The Snapshots feature lets you move your saved Face data, such as a trained **PersonGroup**, to a different Azure Cognitive Services Face subscription. You may want to use this feature if, for example, you've created a **PersonGroup** object using a free trial subscription and want to migrate it to a paid subscription. See Migrate your face data for an overview of the Snapshots feature.

In this example, you will migrate the **PersonGroup** you created in Create and train a person group. You can either complete that section first, or create your own Face data construct(s) to migrate.

Set up target subscription

First, you must have a second Azure subscription with a Face resource; you can do this by following the steps in the Setting up section.

Then, define the following variables in the `Main` method of your program. You'll need to create new environment variables for the subscription ID of your Azure account, as well as the key, endpoint, and subscription ID of your new (target) account.

```
// The Snapshot example needs its own 2nd client, since it uses two different regions.
string TARGET_SUBSCRIPTION_KEY = Environment.GetEnvironmentVariable("FACE_SUBSCRIPTION_KEY2");
string TARGET_ENDPOINT = Environment.GetEnvironmentVariable("FACE_ENDPOINT2");
// Grab your subscription ID, from any resource in Azure, from the Overview page (all resources have the same
subscription ID).
Guid AZURE_SUBSCRIPTION_ID = new Guid(Environment.GetEnvironmentVariable("AZURE_SUBSCRIPTION_ID"));
// Target subscription ID. It will be the same as the source ID if created Face resources from the same
// subscription (but moving from region to region). If they are different subscriptions, add the other
// target ID here.
Guid TARGET_AZURE_SUBSCRIPTION_ID = new Guid(Environment.GetEnvironmentVariable("AZURE_SUBSCRIPTION_ID"));
```

For this example, declare a variable for the ID of the target **PersonGroup**—the object that belongs to the new subscription, which you will copy your data to.


```
// The Snapshot example needs its own 2nd client, since it uses two different regions.
string TARGET_SUBSCRIPTION_KEY = Environment.GetEnvironmentVariable("FACE_SUBSCRIPTION_KEY2");
string TARGET_ENDPOINT = Environment.GetEnvironmentVariable("FACE_ENDPOINT2");
// Grab your subscription ID, from any resource in Azure, from the Overview page (all resources have the same
subscription ID).
Guid AZURE_SUBSCRIPTION_ID = new Guid(Environment.GetEnvironmentVariable("AZURE_SUBSCRIPTION_ID"));
// Target subscription ID. It will be the same as the source ID if created Face resources from the same
// subscription (but moving from region to region). If they are different subscriptions, add the other
// target ID here.
Guid TARGET_AZURE_SUBSCRIPTION_ID = new Guid(Environment.GetEnvironmentVariable("AZURE_SUBSCRIPTION_ID"));
```


Authenticate target client

Next, add the code to authenticate your secondary Face subscription.

```
// Authenticate for another region or subscription (used in Snapshot only).
IFaceClient clientTarget = Authenticate(TARGET_ENDPOINT, TARGET_SUBSCRIPTION_KEY);
```

Use a snapshot

The rest of the snapshot operations must take place within an asynchronous method.

1. The first step is to **take** the snapshot, which saves your original subscription's face data to a temporary cloud location. This method returns an ID that you use to query the status of the operation.

```
/*
 * SNAPSHOT OPERATIONS
 * Copies a person group from one Azure region (or subscription) to another. For example: from the
EastUS region to the WestUS.
 * The same process can be used for face lists.
 * NOTE: the person group in the target region has a new person group ID, so it no longer associates
with the source person group.
 */
public static async Task Snapshot(IFaceClient clientSource, IFaceClient clientTarget, string
personGroupId, Guid azureId, Guid targetAzureId)
{
    Console.WriteLine("========SNAPSHOT OPERATIONS========");
    Console.WriteLine();

    // Take a snapshot for the person group that was previously created in your source region.
    var takeSnapshotResult = await clientSource.Snapshot.TakeAsync(SnapshotObjectType.PersonGroup,
personGroupId, new[] { azureId }); // add targetAzureId to this array if your target ID is different
from your source ID.

    // Get operation id from response for tracking the progress of snapshot taking.
    var operationId = Guid.Parse(takeSnapshotResult.OperationLocation.Split('/')[2]);
    Console.WriteLine($"Taking snapshot(operation ID: {operationId})... Started");
```

2. Next, query the ID until the operation has completed.

```
// Wait for taking the snapshot to complete.
OperationStatus operationStatus = null;
do
{
    Thread.Sleep(TimeSpan.FromMilliseconds(1000));
    // Get the status of the operation.
    operationStatus = await clientSource.Snapshot.GetOperationStatusAsync(operationId);
    Console.WriteLine($"Operation Status: {operationStatus.Status}");
}
while (operationStatus.Status != OperationStatusType.Succeeded && operationStatus.Status !=
OperationStatusType.Failed);
// Confirm the location of the resource where the snapshot is taken and its snapshot ID
var snapshotId = Guid.Parse(operationStatus.ResourceLocation.Split('/')[2]);
Console.WriteLine($"Source region snapshot ID: {snapshotId}");
Console.WriteLine($"Taking snapshot of person group: {personGroupId}... Done\n");
```

3. Then use the **apply** operation to write your face data to your target subscription. This method also returns an ID value.

```
// Apply the snapshot in target region, with a new ID.
var newPersonGroupId = Guid.NewGuid().ToString();
targetPersonGroup = newPersonGroupId;

try
{
    var applySnapshotResult = await clientTarget.Snapshot.ApplyAsync(snapshotId, newPersonGroupId);

    // Get operation id from response for tracking the progress of snapshot applying.
    var applyOperationId = Guid.Parse(applySnapshotResult.OperationLocation.Split('/')[2]);
    Console.WriteLine($"Applying snapshot(operation ID: {applyOperationId})... Started");
```

4. Again, query the new ID until the operation has completed.

```
// Apply the snapshot in target region, with a new ID.
var newPersonGroupId = Guid.NewGuid().ToString();
targetPersonGroup = newPersonGroupId;

try
{
    var applySnapshotResult = await clientTarget.Snapshot.ApplyAsync(snapshotId, newPersonGroupId);

    // Get operation id from response for tracking the progress of snapshot applying.
    var applyOperationId = Guid.Parse(applySnapshotResult.OperationLocation.Split('/')[2]);
    Console.WriteLine($"Applying snapshot(operation ID: {applyOperationId})... Started");
```

5. Finally, complete the try/catch block and finish the method.

```
    catch (Exception e)
    {
        throw new ApplicationException("Do you have a second Face resource in Azure? " +
            "It's needed to transfer the person group to it for the Snapshot example.", e);
    }
}
```

At this point, your new **PersonGroup** object should have the same data as the original one and should be accessible from your new (target) Azure Face subscription.

Run the application

Run the application from your application directory with the `dotnet run` command.

```
dotnet run
```

Clean up resources

If you want to clean up and remove a Cognitive Services subscription, you can delete the resource or resource group. Deleting the resource group also deletes any other resources associated with it.

- Portal
- Azure CLI

If you created a **PersonGroup** in this quickstart and you want to delete it, run the following code in your program:

```
// At end, delete person groups in both regions (since testing only)
Console.WriteLine("========DELETE PERSON GROUP========");
Console.WriteLine();
DeletePersonGroup(client, sourcePersonGroup).Wait();
```

Define the deletion method with the following code:

```
/*
 * DELETE PERSON GROUP
 * After this entire example is executed, delete the person group in your Azure account,
 * otherwise you cannot recreate one with the same name (if running example repeatedly).
 */
public static async Task DeletePersonGroup(IFaceClient client, String personGroupId)
{
    await client.PersonGroup.DeleteAsync(personGroupId);
    Console.WriteLine($"Deleted the person group {personGroupId}.");
}
```

Additionally, if you migrated data using the Snapshot feature in this quickstart, you'll also need to delete the **PersonGroup** saved to the target subscription.

```
DeletePersonGroup(clientTarget, targetPersonGroup).Wait();
Console.WriteLine();
```

Next steps

In this quickstart, you learned how to use the Face library for .NET to do basis tasks. Next, explore the reference documentation to learn more about the library.

Face API reference (.NET)

- What is the Face API?
- The source code for this sample can be found on GitHub.

Quickstart: Face client library for Python

10/9/2019 • 17 minutes to read • Edit Online

Get started with the Face client library for Python. Follow these steps to install the package and try out the example code for basic tasks. The Face API service provides you with access to advanced algorithms for detecting and recognizing human faces in images.

Use the Face client library for Python to:

- Detect faces in an image
- Find similar faces
- Create and train a person group
- Identify a face
- Verify faces
- Take a snapshot for data migration

Reference documentation | Library source code | Package (PiPy) | Samples

Prerequisites

- Azure subscription - Create one for free
- Python 3.x

Setting up

Create a Face Azure resource

Azure Cognitive Services are represented by Azure resources that you subscribe to. Create a resource for Face using the Azure portal or Azure CLI on your local machine. You can also:

- Get a trial key valid for seven days for free. After you sign up, it will be available on the Azure website.
- View your resource on the Azure portal

After you get a key from your trial subscription or resource, create an environment variable for the key, named `FACE_SUBSCRIPTION_KEY`.

Create a new Python application

Create a new Python script—*quickstart-file.py*, for example. Then open it in your preferred editor or IDE and import the following libraries.

```
import asyncio, io, glob, os, sys, time, uuid, requests
from urllib.parse import urlparse
from io import BytesIO
from PIL import Image, ImageDraw
from azure.cognitiveservices.vision.face import FaceClient
from msrest.authentication import CognitiveServicesCredentials
from azure.cognitiveservices.vision.face.models import TrainingStatusType, Person, SnapshotObjectType,
OperationStatusType
```

Then, create variables for your resource's Azure endpoint and key. You may need to change the first part of the endpoint (`westus`) to match your subscription.

```
# Set the FACE_SUBSCRIPTION_KEY environment variable with your key as the value.
# This key will serve all examples in this document.
KEY = os.environ['FACE_SUBSCRIPTION_KEY']

# Set the FACE_ENDPOINT environment variable with the endpoint from your Face service in Azure.
# This endpoint will be used in all examples in this quickstart.
ENDPOINT = os.environ['FACE_ENDPOINT']
```

> **NOTE**
>
> If you created the environment variable after you launched the application, you will need to close and reopen the editor, IDE, or shell running it to access the variable.

Install the client library

You can install the client library with:

```
pip install --upgrade azure-cognitiveservices-vision-face
```

Object model

The following classes and interfaces handle some of the major features of the Face Python SDK.

NAME	DESCRIPTION
FaceClient	This class represents your authorization to use the Face service, and you need it for all Face functionality. You instantiate it with your subscription information, and you use it to produce instances of other classes.
FaceOperations	This class handles the basic detection and recognition tasks that you can do with human faces.
DetectedFace	This class represents all of the data that was detected from a single face in an image. You can use it to retrieve detailed information about the face.
FaceListOperations	This class manages the cloud-stored **FaceList** constructs, which store an assorted set of faces.
PersonGroupPersonOperations	This class manages the cloud-stored **Person** constructs, which store a set of faces that belong to a single person.
PersonGroupOperations	This class manages the cloud-stored **PersonGroup** constructs, which store a set of assorted **Person** objects.
ShapshotOperations	This class manages the Snapshot functionality; you can use it to temporarily save all of your cloud-based face data and migrate that data to a new Azure subscription.

Code examples

These code snippets show you how to do the following tasks with the Face client library for Python:

- Authenticate the client

- Detect faces in an image
- Find similar faces
- Create and train a person group
- Identify a face
- Verify faces
- Take a snapshot for data migration

Authenticate the client

> **NOTE**
> This quickstart assumes you've created an environment variable for your Face key, named `FACE_SUBSCRIPTION_KEY`.

Instantiate a client with your endpoint and key. Create a CognitiveServicesCredentials object with your key, and use it with your endpoint to create a FaceClient object.

```
# Create an authenticated FaceClient.
face_client = FaceClient(ENDPOINT, CognitiveServicesCredentials(KEY))
```

Detect faces in an image

The following code detects a face in a remote image. It prints the detected face's ID to the console and also stores it in program memory. Then, it detects the faces in an image with multiple people and prints their IDs to the console as well. By changing the parameters in the detect_with_url method, you can return different information with each DetectedFace object.

```
# Detect a face in an image that contains a single face
single_face_image_url = 'https://www.biography.com/.image/t_share/MTQ1MzAyNzYzOTgxNTE0NTEz/john-f-kennedy---
mini-biography.jpg'
single_image_name = os.path.basename(single_face_image_url)
detected_faces = face_client.face.detect_with_url(url=single_face_image_url)
if not detected_faces:
    raise Exception('No face detected from image {}'.format(single_image_name))

# Display the detected face ID in the first single-face image.
# Face IDs are used for comparison to faces (their IDs) detected in other images.
print('Detected face ID from', single_image_name, ':')
for face in detected_faces: print (face.face_id)
print()

# Save this ID for use in Find Similar
first_image_face_ID = detected_faces[0].face_id
```

See the sample code on GitHub for more detection scenarios.

Display and frame faces

The following code outputs the given image to the display and draws rectangles around the faces, using the DetectedFace.faceRectangle property.

```
# Detect a face in an image that contains a single face
single_face_image_url = 'https://raw.githubusercontent.com/Microsoft/Cognitive-Face-
Windows/master/Data/detection1.jpg'
single_image_name = os.path.basename(single_face_image_url)
detected_faces = face_client.face.detect_with_url(url=single_face_image_url)
if not detected_faces:
    raise Exception('No face detected from image {}'.format(single_image_name))

# Convert width height to a point in a rectangle
def getRectangle(faceDictionary):
    rect = faceDictionary['face_rectangle']
    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 detected_faces:
    draw.rectangle(getRectangle(face), outline='red')

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

Find similar faces

The following code takes a single detected face and searches a set of other faces to find matches. When it finds a match, it prints the rectangle coordinates of the matched face to the console.

Find matches

First, run the code in the above section (Detect faces in an image) to save a reference to a single face. Then run the

following code to get references to several faces in a group image.

```
# Detect the faces in an image that contains multiple faces
# Each detected face gets assigned a new ID
multi_face_image_url = "http://www.historyplace.com/kennedy/president-family-portrait-closeup.jpg"
multi_image_name = os.path.basename(multi_face_image_url)
detected_faces2 = face_client.face.detect_with_url(url=multi_face_image_url)
```

Then add the following code block to find instances of the first face in the group. See the find_similar method to learn how to modify this behavior.

```
# Search through faces detected in group image for the single face from first image.
# First, create a list of the face IDs found in the second image.
second_image_face_IDs = list(map(lambda x: x.face_id, detected_faces2))
# Next, find similar face IDs like the one detected in the first image.
similar_faces = face_client.face.find_similar(face_id=first_image_face_ID, face_ids=second_image_face_IDs)
if not similar_faces[0]:
    print('No similar faces found in', multi_image_name, '.')
```

Print matches

Use the following code to print the match details to the console.

```
# Print the details of the similar faces detected
print('Similar faces found in', multi_image_name + ':')
for face in similar_faces:
    first_image_face_ID = face.face_id
    # The similar face IDs of the single face image and the group image do not need to match,
    # they are only used for identification purposes in each image.
    # The similar faces are matched using the Cognitive Services algorithm in find_similar().
    face_info = next(x for x in detected_faces2 if x.face_id == first_image_face_ID)
    if face_info:
        print('  Face ID: ', first_image_face_ID)
        print('  Face rectangle:')
        print('    Left: ', str(face_info.face_rectangle.left))
        print('    Top: ', str(face_info.face_rectangle.top))
        print('    Width: ', str(face_info.face_rectangle.width))
        print('    Height: ', str(face_info.face_rectangle.height))
```

Create and train a person group

The following code creates a **PersonGroup** with three different **Person** objects. It associates each **Person** with a set of example images, and then it trains to be able to recognize each person.

Create PersonGroup

To step through this scenario, you need to save the following images to the root directory of your project: https://github.com/Azure-Samples/cognitive-services-sample-data-files/tree/master/Face/images.

This group of images contains three sets of face images corresponding to three different people. The code will define three **Person** objects and associate them with image files that start with `woman`, `man`, and `child`.

Once you've set up your images, define a label at the top of your script for the **PersonGroup** object you'll create.

```
# Used in the Person Group Operations,  Snapshot Operations, and Delete Person Group examples.
# You can call list_person_groups to print a list of preexisting PersonGroups.
# SOURCE_PERSON_GROUP_ID should be all lowercase and alphanumeric. For example, 'mygroupname' (dashes are OK).
PERSON_GROUP_ID = 'my-unique-person-group'
# Used for the Snapshot and Delete Person Group examples.
TARGET_PERSON_GROUP_ID = str(uuid.uuid4()) # assign a random ID (or name it anything)
```

Then add the following code to the bottom of your script. This code creates a **PersongGroup** and three **Person** objects.

```
'''
Create the PersonGroup
'''
# Create empty Person Group. Person Group ID must be lower case, alphanumeric, and/or with '-', '_'.
print('Person group:', PERSON_GROUP_ID)
face_client.person_group.create(person_group_id=PERSON_GROUP_ID, name=PERSON_GROUP_ID)

# Define woman friend
woman = face_client.person_group_person.create(PERSON_GROUP_ID, "Woman")
# Define man friend
man = face_client.person_group_person.create(PERSON_GROUP_ID, "Man")
# Define child friend
child = face_client.person_group_person.create(PERSON_GROUP_ID, "Child")
```

Assign faces to Persons

The following code sorts your images by their prefix, detects faces, and assigns the faces to each **Person** object.

```
'''
Detect faces and register to correct person
'''
# Find all jpeg images of friends in working directory
woman_images = [file for file in glob.glob('*.jpg') if file.startswith("woman")]
man_images = [file for file in glob.glob('*.jpg') if file.startswith("man")]
child_images = [file for file in glob.glob('*.jpg') if file.startswith("child")]

# Add to a woman person
for image in woman_images:
    w = open(image, 'r+b')
    face_client.person_group_person.add_face_from_stream(PERSON_GROUP_ID, woman.person_id, w)

# Add to a man person
for image in man_images:
    m = open(image, 'r+b')
    face_client.person_group_person.add_face_from_stream(PERSON_GROUP_ID, man.person_id, m)

# Add to a child person
for image in child_images:
    ch = open(image, 'r+b')
    face_client.person_group_person.add_face_from_stream(PERSON_GROUP_ID, child.person_id, ch)
```

Train PersonGroup

Once you've assigned faces, you must train the **PersonGroup** so that it can identify the visual features associated with each of its **Person** objects. The following code calls the asynchronous **train** method and polls the result, printing the status to the console.

```
'''
Train PersonGroup
'''
print()
print('Training the person group...')
# Train the person group
face_client.person_group.train(PERSON_GROUP_ID)

while (True):
    training_status = face_client.person_group.get_training_status(PERSON_GROUP_ID)
    print("Training status: {}.".format(training_status.status))
    print()
    if (training_status.status is TrainingStatusType.succeeded):
        break
    elif (training_status.status is TrainingStatusType.failed):
        sys.exit('Training the person group has failed.')
    time.sleep(5)
```

Identify a face

The following code takes an image with multiple faces and looks to find the identity of each person in the image. It compares each detected face to a **PersonGroup**, a database of different **Person** objects whose facial features are known.

> **IMPORTANT**
>
> In order to run this example, you must first run the code in Create and train a person group.

Get a test image

The following code looks in the root of your project for an image *test-image-person-group.jpg* and detects the faces in the image. You can find this image with the images used for **PersonGroup** management: https://github.com/Azure-Samples/cognitive-services-sample-data-files/tree/master/Face/images.

```
'''
Identify a face against a defined PersonGroup
'''
# Reference image for testing against
group_photo = 'test-image-person-group.jpg'
IMAGES_FOLDER = os.path.join(os.path.dirname(os.path.realpath(__file__)))

# Get test image
test_image_array = glob.glob(os.path.join(IMAGES_FOLDER, group_photo))
image = open(test_image_array[0], 'r+b')

# Detect faces
face_ids = []
faces = face_client.face.detect_with_stream(image)
for face in faces:
    face_ids.append(face.face_id)
```

Identify faces

The **identify** method takes an array of detected faces and compares them to a **PersonGroup**. If it can match a detected face to a **Person**, it saves the result. This code prints detailed match results to the console.

```
# Identify faces
results = face_client.face.identify(face_ids, PERSON_GROUP_ID)
print('Identifying faces in {}')
if not results:
    print('No person identified in the person group for faces from the
{}.'.format(os.path.basename(image.name)))
for person in results:
    print('Person for face ID {} is identified in {} with a confidence of {}.'.format(person.face_id,
os.path.basename(image.name), person.candidates[0].confidence)) # Get topmost confidence score
```

Verify faces

The Verify operation takes a face ID and either another face ID or a **Person** object and determines whether they belong to the same person.

The following code detects faces in two source images and then verifies them against a face detected from a target image.

Get test images

The following code blocks declare variables that will point to the source and target images for the verification operation.

```
# Base url for the Verify and Facelist/Large Facelist operations
IMAGE_BASE_URL = 'https://csdx.blob.core.windows.net/resources/Face/Images/'
```

```
# Create a list to hold the target photos of the same person
target_image_file_names = ['Family1-Dad1.jpg', 'Family1-Dad2.jpg']
# The source photos contain this person
source_image_file_name1 = 'Family1-Dad3.jpg'
source_image_file_name2 = 'Family1-Son1.jpg'
```

Detect faces for verification

The following code detects faces in the source and target images and saves them to variables.

```
# Detect face(s) from source image 1, returns a list[DetectedFaces]
detected_faces1 = face_client.face.detect_with_url(IMAGE_BASE_URL + source_image_file_name1)
# Add the returned face's face ID
source_image1_id = detected_faces1[0].face_id
print('{} face(s) detected from image {}.'.format(len(detected_faces1), source_image_file_name1))

# Detect face(s) from source image 2, returns a list[DetectedFaces]
detected_faces2 = face_client.face.detect_with_url(IMAGE_BASE_URL + source_image_file_name2)
# Add the returned face's face ID
source_image2_id = detected_faces2[0].face_id
print('{} face(s) detected from image {}.'.format(len(detected_faces2), source_image_file_name2))

# List for the target face IDs (uuids)
detected_faces_ids = []
# Detect faces from target image url list, returns a list[DetectedFaces]
for image_file_name in target_image_file_names:
    detected_faces = face_client.face.detect_with_url(IMAGE_BASE_URL + image_file_name)
    # Add the returned face's face ID
    detected_faces_ids.append(detected_faces[0].face_id)
    print('{} face(s) detected from image {}.'.format(len(detected_faces), image_file_name))
```

Get verification results

The following code compares each of the source images to the target image and prints a message indicating

whether they belong to the same person.

```
# Verification example for faces of the same person. The higher the confidence, the more identical the faces in
the images are.
# Since target faces are the same person, in this example, we can use the 1st ID in the detected_faces_ids list
to compare.
verify_result_same = face_client.face.verify_face_to_face(source_image1_id, detected_faces_ids[0])
print('Faces from {} & {} are of the same person, with confidence: {}'
    .format(source_image_file_name1, target_image_file_names[0], verify_result_same.confidence)
    if verify_result_same.is_identical
    else 'Faces from {} & {} are of a different person, with confidence: {}'
        .format(source_image_file_name1, target_image_file_names[0], verify_result_same.confidence))

# Verification example for faces of different persons.
# Since target faces are same person, in this example, we can use the 1st ID in the detected_faces_ids list to
compare.
verify_result_diff = face_client.face.verify_face_to_face(source_image2_id, detected_faces_ids[0])
print('Faces from {} & {} are of the same person, with confidence: {}'
    .format(source_image_file_name2, target_image_file_names[0], verify_result_diff.confidence)
    if verify_result_diff.is_identical
    else 'Faces from {} & {} are of a different person, with confidence: {}'
        .format(source_image_file_name2, target_image_file_names[0], verify_result_diff.confidence))
```

Take a snapshot for data migration

The Snapshots feature lets you move your saved face data, such as a trained **PersonGroup**, to a different Azure Cognitive Services Face subscription. You may want to use this feature if, for example, you've created a **PersonGroup** object using a free trial subscription and now want to migrate it to a paid subscription. See the Migrate your face data for a broad overview of the Snapshots feature.

In this example, you will migrate the **PersonGroup** you created in Create and train a person group. You can either complete that section first, or use your own Face data construct(s).

Set up target subscription

First, you must have a second Azure subscription with a Face resource; you can do this by following the steps in the Setting up section.

Then, create the following variables near the top of your script. You'll also need to create new environment variables for the subscription ID of your Azure account, as well as the key, endpoint, and subscription ID of your new (target) account.

```
'''
Snapshot operations variables
These are only used for the snapshot example. Set your environment variables accordingly.
'''
# Source endpoint, the location/subscription where the original person group is located.
SOURCE_ENDPOINT = ENDPOINT
# Source subscription key. Must match the source endpoint region.
SOURCE_KEY = KEY
# Source subscription ID (different than  key). From the Azure portal.
SOURCE_ID = os.environ['AZURE_SUBSCRIPTION_ID']
# Person group name that will get created in this quickstart's Person Group Operations example.
SOURCE_PERSON_GROUP_ID = PERSON_GROUP_ID
# Target endpoint. A separate Face resource in a different region (or a different subscription with same
region).
TARGET_ENDPOINT = os.environ["FACE_ENDPOINT2"]
# Target subscription key. Must match the target endpoint region/subscription.
TARGET_KEY = os.environ['FACE_SUBSCRIPTION_KEY2']
# Target subscription ID. It will be the same as the source ID if created Face resources from the
# same subscription (but moving from region to region). If they are different subscriptions, add the other
target ID here.
TARGET_ID = os.environ['AZURE_SUBSCRIPTION_ID']
# NOTE: We do not need to specify the target PersonGroup ID here because we generate it with this example.
# Each new location you transfer a person group to will have a generated, new person group ID for that region.
```

Authenticate target client

Later in your script, save your current client object as the source client, and then authenticate a new client object for your target subscription.

```
'''
Authenticate
'''
# Use your source client already created (it has the person group ID you need in it).
face_client_source = face_client
# Create a new FaceClient instance for your target with authentication.
face_client_target = FaceClient(TARGET_ENDPOINT, CognitiveServicesCredentials(TARGET_KEY))
```

Use a snapshot

The rest of the snapshot operations take place within an asynchronous function.

1. The first step is to **take** the snapshot, which saves your original subscription's face data to a temporary cloud location. This method returns an ID that you use to query the status of the operation.

```
'''
Snapshot operations in 4 steps
'''
async def run():
    # STEP 1, take a snapshot of your person group, then track status.
    # This list must include all subscription IDs from which you want to access the snapshot.
    source_list = [SOURCE_ID, TARGET_ID] # You may have many sources, if transferring from many regions
    # remove any duplicates from the list. Passing the same subscription ID more than once causes
    # the Snapshot.take operation to fail.
    source_list = list(dict.fromkeys(source_list))

    # Note Snapshot.take is not asynchronous.
    take_snapshot_result = face_client_source.snapshot.take(
        type=SnapshotObjectType.person_group,
        object_id=PERSON_GROUP_ID,
        apply_scope=source_list,
        # Set this to tell Snapshot.take to return the response; otherwise it returns None.
        raw=True
        )
    # Get operation ID from response for tracking
    # Snapshot.type return value is of type msrest.pipeline.ClientRawResponse.
    take_operation_id = take_snapshot_result.response.headers['Operation-
Location'].replace('/operations/', '')

    print('Taking snapshot( operation ID:', take_operation_id, ')...')
```

2. Next, query the ID until the operation has completed.

```
# STEP 2, Wait for snapshot taking to complete.
take_status = await wait_for_operation(face_client_source, take_operation_id)

# Get snapshot id from response.
snapshot_id = take_status.resource_location.replace ('/snapshots/', '')

print('Snapshot ID:', snapshot_id)
print('Taking snapshot... Done\n')
```

This code makes use of the `wait_for_operation` function, which you should define separately:

```
# Helper function that waits and checks status of API call processing.
async def wait_for_operation(client, operation_id):
    # Track progress of taking the snapshot.
    # Note Snapshot.get_operation_status is not asynchronous.
    result = client.snapshot.get_operation_status(operation_id=operation_id)

    status = result.status.lower()
    print('Operation status:', status)
    if ('notstarted' == status or 'running' == status):
        print("Waiting 10 seconds...")
        await asyncio.sleep(10)
        result = await wait_for_operation(client, operation_id)
    elif ('failed' == status):
        raise Exception("Operation failed. Reason:" + result.message)
    return result
```

3. Go back to your asynchronous function. Use the **apply** operation to write your face data to your target subscription. This method also returns an ID.

```
# STEP 3, apply the snapshot to target region(s)
# Snapshot.apply is not asynchronous.
apply_snapshot_result = face_client_target.snapshot.apply(
    snapshot_id=snapshot_id,
    # Generate a new UUID for the target person group ID.
    object_id=TARGET_PERSON_GROUP_ID,
    # Set this to tell Snapshot.apply to return the response; otherwise it returns None.
    raw=True
    )
apply_operation_id = apply_snapshot_result.response.headers['Operation-
Location'].replace('/operations/', '')
print('Applying snapshot( operation ID:', apply_operation_id, ')...')
```

4. Again, use the `wait_for_operation` function to query the ID until the operation has completed.

```
# STEP 4, wait for applying snapshot process to complete.
await wait_for_operation(face_client_target, apply_operation_id)
print('Applying snapshot... Done\n')
print('End of transfer.')
print()
```

Once you've completed these steps, you'll be able to access your face data constructs from your new (target) subscription.

Run the application

Run the application with the `python` command on your quickstart file.

```
python quickstart-file.py
```

Clean up resources

If you want to clean up and remove a Cognitive Services subscription, you can delete the resource or resource group. Deleting the resource group also deletes any other resources associated with it.

- Portal
- Azure CLI

If you created a **PersonGroup** in this quickstart and you want to delete it, run the following code in your script:

```
# Delete the main person group.
face_client.person_group.delete(person_group_id=PERSON_GROUP_ID)
print("Deleted the person group {} from the source location.".format(PERSON_GROUP_ID))
print()
```

If you migrated data using the Snapshot feature in this quickstart, you'll also need to delete the **PersonGroup** saved to the target subscription.

```
# Delete the person group in the target region.
face_client_target.person_group.delete(TARGET_PERSON_GROUP_ID)
print("Deleted the person group {} from the target location.".format(TARGET_PERSON_GROUP_ID))
```

Next steps

In this quickstart, you learned how to use the Face library for Python to do basis tasks. Next, explore the reference documentation to learn more about the library.

Face API reference (Python)

- What is the Face API?
- The source code for this sample can be found on GitHub.

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

9/10/2019 • 5 minutes to read • Edit Online

In this quickstart, you will use the Azure Face REST API with C# to detect human faces in an image.

If you don't have an Azure subscription, create a free account before you begin.

Prerequisites

- A Face API subscription key. You can get a free trial subscription key from Try Cognitive Services. Or, follow the instructions in Create a Cognitive Services account to subscribe to the Face API service and get your key.
- Any edition of Visual Studio 2015 or 2017.

Create the Visual Studio project

1. In Visual Studio, create a new **Console app (.NET Framework)** project and name it **FaceDetection**.
2. If there are other projects in your solution, select this one as the single startup project.

Add face detection code

Open the new project's *Program.cs* file. Here, you will add the code needed to load images and detect faces.

Include namespaces

Add the following `using` statements to the top of your *Program.cs* file.

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

Add essential fields

Add the **Program** class containing the following fields. This data specifies how to connect to the Face service and where to get the input data. You'll need to update the `subscriptionKey` field with the value of your subscription key, and you may need to change the `uriBase` string so that it contains your resource endpoint string.

> **NOTE**
>
> New resources created after July 1, 2019, will use custom subdomain names. For more information and a complete list of regional endpoints, see Custom subdomain names for Cognitive Services.

```
namespace DetectFace
{
    class Program
    {

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

        // replace <myresourcename> with the string found in your endpoint URL
        const string uriBase =
            "https://<myresourcename>.cognitive.microsoft.com/face/v1.0/detect";
```

Receive image input

Add the following code to the **Main** method of the **Program** class. This code writes a prompt to the console asking the user to enter an image URL. Then it calls another method, **MakeAnalysisRequest**, to process the image at that location.

```
        static void Main(string[] args)
        {

            // 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))
            {
                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();
        }
```

Call the face detection REST API

Add the following method to the **Program** class. It constructs a REST call to the Face API to detect face information in the remote image (the `requestParameters` string specifies which face attributes to retrieve). Then it writes the output data to a JSON string.

You will define the helper methods in the following steps.

```
        // Gets the analysis of the specified image by using the Face REST API.
        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(JsonPrettyPrint(contentString));
                Console.WriteLine("\nPress Enter to exit...");
            }
        }
```

Process the input image data

Add the following method to the **Program** class. This method converts the image at the specified URL into a byte array.

```
        // Returns the contents of the specified file as a byte array.
        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);
            }
        }
```

Parse the JSON response

Add the following method to the **Program** class. This method formats the JSON input to be more easily readable. Your app will write this string data to the console. You can then close the class and namespace.

```
        // Formats the given JSON string by adding line breaks and indents.
        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);
                            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();
        }
    }
}
```

Run the app

A successful response will display Face data in easily readable JSON format. 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

In this quickstart, you created a simple .NET console application that uses REST calls with the Azure Face API to detect faces in an image and return their attributes. Next, explore the Face API reference documentation to learn more about the supported scenarios.

Face API

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

9/10/2019 • 2 minutes to read • Edit Online

In this quickstart, you will use the Azure Face REST API with cURL to detect human faces in an image.

If you don't have an Azure subscription, create a free account before you begin.

Prerequisites

- A Face API subscription key. You can get a free trial subscription key from Try Cognitive Services. Or, follow the instructions in Create a Cognitive Services account to subscribe to the Face API service and get your key.

Write the command

You'll use a command like the following to call the Face API and get face attribute data from an image. First, copy the code into a text editor—you'll need to make changes to certain parts of the command before you can run it.

```
curl -H "Ocp-Apim-Subscription-Key: <Subscription Key>" "https://<My Endpoint String>.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\"}"
```

Subscription key

Replace `<Subscription Key>` with your valid Face subscription key.

Face endpoint URL

The URL `https://<My Endpoint String>.com/face/v1.0/detect` indicates the Azure Face endpoint to query. You may need to change the first part of this URL to match the endpoint that corresponds to your subscription key.

> **NOTE**
> New resources created after July 1, 2019, will use custom subdomain names. For more information and a complete list of regional endpoints, see Custom subdomain names for Cognitive Services.

URL query string

The query string of the Face endpoint URL specifies which face attributes to retrieve. You may wish to change this string depending on your intended use.

```
?
returnFaceId=true&returnFaceLandmarks=false&returnFaceAttributes=age,gender,headPose,smile,facialHair,glasses,e
motion,hair,makeup,occlusion,accessories,blur,exposure,noise
```

Image source URL

The source URL indicates the image to use as input. You can change this to point to any image you want to analyze.

```
https://upload.wikimedia.org/wikipedia/commons/c/c3/RH_Louise_Lillian_Gish.jpg
```

Run the command

Once you've made your changes, open a command prompt and enter the new command. You should see the face information displayed as JSON data in the console window. For example:


```
[
  {
    "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

In this quickstart, you wrote a cURL command that calls the Azure Face API to detect faces in an image and return their attributes. Next, explore the Face API reference documentation to learn more.

Face API

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

9/10/2019 • 3 minutes to read • Edit Online

In this quickstart, you will use the Azure Face REST API with Go to detect human faces in an image.

Prerequisites

- A Face API subscription key. You can get a free trial subscription key from Try Cognitive Services. Or, follow the instructions in Create a Cognitive Services account to subscribe to the Face API service and get your key.
- A code editor such as Visual Studio Code

Write the script

Create a new file, *faceDetection.go*, and add the following code. This calls the Face API for a given image URL.

```
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://<My Endpoint String>.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)

    //Configure TLS, etc.
    tr := &http.Transport{
        TLSClientConfig: &tls.Config{
            InsecureSkipVerify: true,
        },
    }

    // Create the Http client
    client := &http.Client{
        Transport: tr,
        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))
}
```

You'll need to update the `subscriptionKey` value with your subscription key, and change the `uriBase` string so that it contains the correct endpoint string.

> **NOTE**
> New resources created after July 1, 2019, will use custom subdomain names. For more information and a complete list of regional endpoints, see Custom subdomain names for Cognitive Services.

You may also wish to change the `imageUrl` field to point to your own input image. You might also wish to change the `returnFaceAttributes` field that specifies which face attributes to retrieve.

Run the script

Open a command prompt and build the program with the following command:

```
go build faceDetection.go
```

Then run the program:

```
detect-face
```

You should see a JSON string of detected face data printed to the console. The following is an example of a successful JSON response.

```
[
```

```
{
  "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

In this quickstart, you wrote a Ruby script that calls the Azure Face API to detect faces in an image and return their attributes. Next, explore the Face API reference documentation to learn more.

Face API

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

9/10/2019 • 4 minutes to read • Edit Online

In this quickstart, you will use the Azure Face REST API with Java to detect human faces in an image.

If you don't have an Azure subscription, create a free account before you begin.

Prerequisites

- A Face API subscription key. You can get a free trial subscription key from Try Cognitive Services. Or, follow the instructions in Create a Cognitive Services account to subscribe to the Face API service and get your key.
- Any Java IDE of your choice.

Create the Java project

1. Create a new command-line Java app in your IDE and add a **Main** class with a **main** method.
2. Import the following libraries into your Java project. If you're using Maven, the Maven coordinates are provided for each library.
   - Apache HTTP client (org.apache.httpcomponents:httpclient:4.5.6)
   - Apache HTTP core (org.apache.httpcomponents:httpcore:4.4.10)
   - JSON library (org.json:json:20180130)
   - Apache Commons logging (commons-logging:commons-logging:1.1.2)

Add face detection code

Open the main class of your project. Here, you will add the code needed to load images and detect faces.

Import packages

Add the following `import` statements to the top of the file.

```
// 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.HttpClientBuilder;
import org.apache.http.util.EntityUtils;
import org.json.JSONArray;
import org.json.JSONObject;
```

Add essential fields

Replace the **Main** class with the following code. This data specifies how to connect to the Face service and where to get the input data. You'll need to update the `subscriptionKey` field with the value of your subscription key, and change the `uriBase` string so that it contains the correct endpoint string. You may also wish to set the

`imageWithFaces` value to a path that points to a different image file.

> **NOTE**
> New resources created after July 1, 2019, will use custom subdomain names. For more information and a complete list of regional endpoints, see Custom subdomain names for Cognitive Services.

The `faceAttributes` field is simply a list of certain types of attributes. It will specify which information to retrieve about the detected faces.

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

    private static final String uriBase =
        "https://<My Endpoint String>.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";
```

Call the face detection REST API

Add the **main** method with the following code. It constructs a REST call to the Face API to detect face information in the remote image (the `faceAttributes` string specifies which face attributes to retrieve). Then it writes the output data to a JSON string.

```
    public static void main(String[] args) {
        HttpClient httpclient = HttpClientBuilder.create().build();

        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();
```

Parse the JSON response

Directly below the previous code, add the following block, which converts the returned JSON data into a more easily readable format before printing it to the console. Finally, close out the try-catch block, the **main** method, and

the **Main** class.

```
            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());
        }
    }
}
```

Run the app

Compile the code and run it. A successful response will display Face data in easily readable JSON format in the console window. For example:

```
[{
  "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
```

```
      "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

In this quickstart, you created a simple Java console application that uses REST calls with the Azure Face API to detect faces in an image and return their attributes. Next, learn how to do more with this functionality in an Android application.

Tutorial: Create an Android app to detect and frame faces

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

9/10/2019 • 3 minutes to read • Edit Online

In this quickstart, you will use the Azure Face REST API with JavaScript to detect human faces in an image.

Prerequisites

- A Face API subscription key. You can get a free trial subscription key from Try Cognitive Services. Or, follow the instructions in Create a Cognitive Services account to subscribe to the Face API service and get your key.
- A code editor such as Visual Studio Code

Initialize the HTML file

Create a new HTML file, *detectFaces.html*, and add the following code.

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

Then add the following code inside the `body` element of the document. This code sets up a basic user interface with a URL field, an **Analyze face** button, a response pane, and an image display pane.

```
<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>
```

Write the JavaScript script

Add the following code immediately above the `h1` element in your document. This code sets up the JavaScript code that calls the Face API.

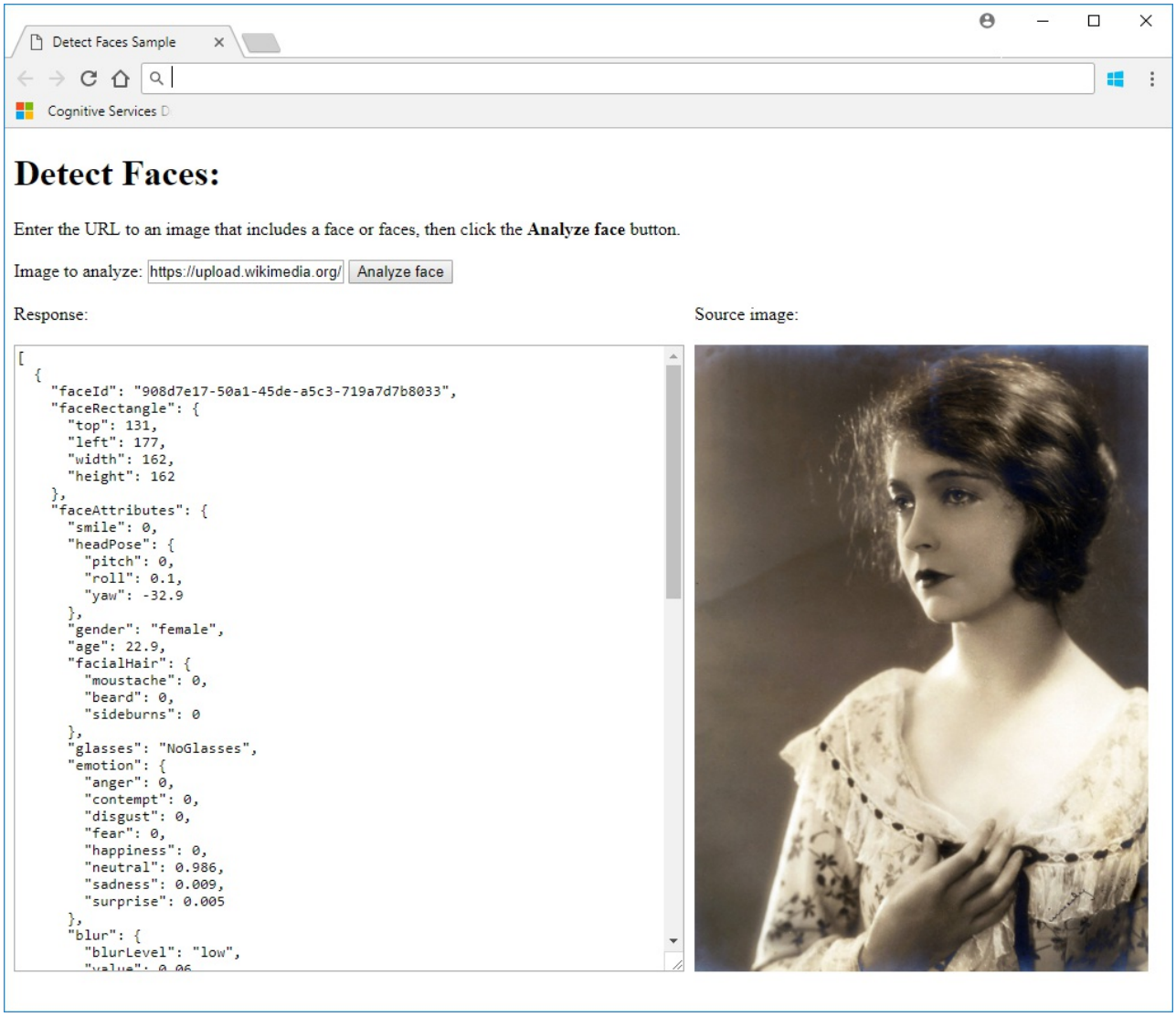
```
<script type="text/javascript">
    function processImage() {
        // Replace <Subscription Key> with your valid subscription key.
        var subscriptionKey = "<Subscription Key>";

        var uriBase =
            "https://<My Endpoint String>.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>
```

You'll need to update the `subscriptionKey` field with the value of your subscription key, and you need to change the `uriBase` string so that it contains the correct endpoint string. The `returnFaceAttributes` field specifies which face attributes to retrieve; you may wish to change this string depending on your intended use.

> **NOTE**
>
> New resources created after July 1, 2019, will use custom subdomain names. For more information and a complete list of regional endpoints, see Custom subdomain names for Cognitive Services.

Run the script

Open *detectFaces.html* in your browser. When you click the **Analyze face** button, the app should display the image from the given URL and print out a JSON string of face data.



The following text is an example of a successful JSON 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

In this quickstart, you wrote a JavaScript script that calls the Azure Face API to detect faces in an image and return their attributes. Next, explore the Face API reference documentation to learn more.

Face API

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

9/10/2019 • 3 minutes to read • Edit Online

In this quickstart, you will use the Azure Face REST API with Node.js to detect human faces in an image.

If you don't have an Azure subscription, create a free account before you begin.

Prerequisites

- A Face API subscription key. You can get a free trial subscription key from Try Cognitive Services. Or, follow the instructions in Create a Cognitive Services account to subscribe to the Face API service and get your key.
- A code editor such as Visual Studio Code

Set up the Node environment

Go to the folder where you'd like to create your project and create a new file, *facedetection.js*. Then install the `requests` module to this project. This allows your scripts to make HTTP requests.

```
npm install request --save
```

Write the Node.js script

Paste the following code into *facedetection.js*. These fields specify how to connect to the Face service and where to get the input data. You'll need to update the `subscriptionKey` field with the value of your subscription key, and you need to change the `uriBase` string so that it contains the correct endpoint string. You may wish to change the `imageUrl` field to point to your own input image.

> **NOTE**
> New resources created after July 1, 2019, will use custom subdomain names. For more information and a complete list of regional endpoints, see Custom subdomain names for Cognitive Services.

```
'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://<My Endpoint String>.com/face/v1.0/detect';

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

Then, add the following code to call the Face API and get face attribute data from the input image. The `returnFaceAttributes` field specifies which face attributes to retrieve. You may wish to change this string depending

on your intended use.

```
// 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);
});
```

Save and run the script

After you've made your changes, open a command prompt and run the file with the `node` command.

```
node facedetection.js
```

You should see the face information displayed as JSON data in the console window. 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

In this quickstart, you wrote a Node.js script that calls the Azure Face API to detect faces in an image and return their attributes. Next, explore the Face API reference documentation to learn more.

Face API

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

9/10/2019 • 3 minutes to read • Edit Online

In this quickstart, you will use the Azure Face REST API with PHP to detect human faces in an image.

Prerequisites

- A Face API subscription key. You can get a free trial subscription key from Try Cognitive Services. Or, follow the instructions in Create a Cognitive Services account to subscribe to the Face API service and get your key.
- A code editor such as Visual Studio Code.
- The PHP HTTP_Request2 package.
- A PHP-enabled web browser. If you have not set this up, you can do so by installing and setting up XAMPP on your machine.

Initialize the HTML file

Create a new HTML file, *detectFaces.html*, and add the following code.

```
<html>
    <head>
        <title>Face Detect Sample</title>
    </head>
    <body></body>
</html>
```

Write the PHP script

Add the following code inside the `body` element of the document. This code sets up a basic user interface with a URL field, an **Analyze face** button, a response pane, and an image display pane.

```
<?php
// Replace <Subscription Key> with a valid subscription key.
$ocpApimSubscriptionKey = '<Subscription Key>';

// Replace <My Endpoint String> with the string in your endpoint URL.
$uriBase = 'https:/<My Endpoint String>.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
// (https://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>";
}
?>
```

You'll need to update the `subscriptionKey` field with the value of your subscription key, and you need to change the `uriBase` string so that it contains the correct endpoint string. The `returnFaceAttributes` field specifies which face attributes to retrieve; you may wish to change this string depending on your intended use.

> **NOTE**
>
> New resources created after July 1, 2019, will use custom subdomain names. For more information and a complete list of regional endpoints, see Custom subdomain names for Cognitive Services.

Run the script

Open the file in a PHP-enabled web browser. You should get a JSON string of Face data, like the following.


```
[
    {
        "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
```

```
                "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

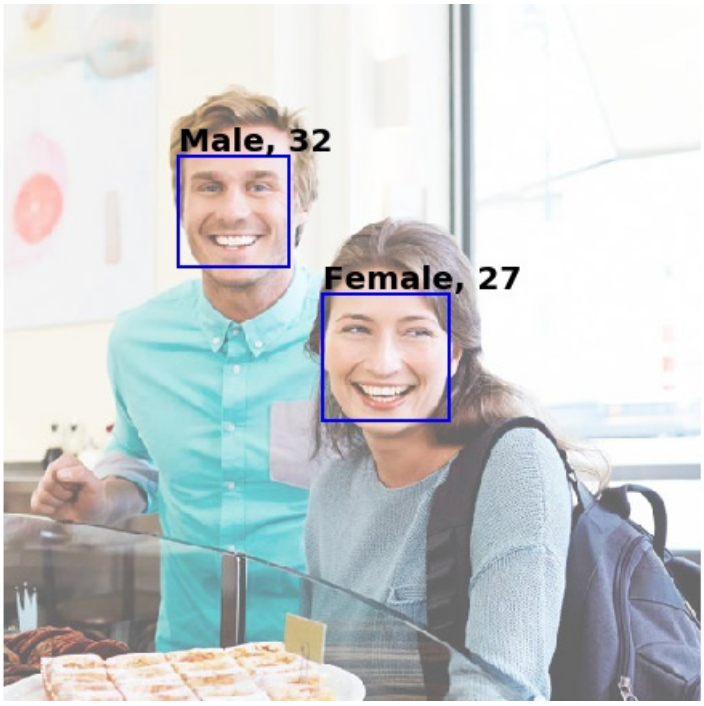
Explore the Face APIs used to detect human faces in an image, demarcate the faces with rectangles, and return attributes such as age and gender.

Face APIs

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

9/10/2019 • 3 minutes to read • Edit Online

In this quickstart, you will use the Azure Face REST API with Python to detect human faces in an image. The script will draw frames around the faces and superimpose gender and age information on the image.



If you don't have an Azure subscription, create a free account before you begin.

Prerequisites

- A Face API subscription key. You can get a free trial subscription key from Try Cognitive Services. Or, follow the instructions in Create a Cognitive Services account to subscribe to the Face API service and get your key.

Run the Jupyter notebook

You can run this quickstart as a Jupyter notebook on MyBinder. To launch Binder, select the button below. Then follow the instructions in the notebook.

launch binder

Create and run the sample

Alternately, you can run this quickstart from the command line with the following steps:

1. Copy the following code into a text editor.
2. Make the following changes in code where needed:
   a. Replace the value of `subscription_key` with your subscription key.
   b. Edit the value of `face_api_url` to include the endpoint URL for your Face API resource.
   c. Optionally, replace the value of `image_url` with the URL of a different image that you want to analyze.

3. Save the code as a file with an `.py` extension. For example, `detect-face.py`.
4. Open a command prompt window.
5. At the prompt, use the `python` command to run the sample. For example, `python detect-face.py`.

```
import requests
import json

# set to your own subscription key value
subscription_key = None
assert subscription_key

# replace <My Endpoint String> with the string from your endpoint URL
face_api_url = 'https://<My Endpoint String>.com/face/v1.0/detect'

image_url = 'https://upload.wikimedia.org/wikipedia/commons/3/37/Dagestani_man_and_woman.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',
}

response = requests.post(face_api_url, params=params,
                         headers=headers, json={"url": image_url})
print(json.dumps(response.json()))
```

Examine the response

A successful response is returned in JSON.

```
[
  {
    "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
```

```
      "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

Next, explore the Face API reference documentation to learn more about the supported scenarios.

Face API

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

9/10/2019 • 3 minutes to read • Edit Online

In this quickstart, you will use the Azure Face REST API with Ruby to detect human faces in an image.

Prerequisites

- A Face API subscription key. You can get a free trial subscription key from Try Cognitive Services. Or, follow the instructions in Create a Cognitive Services account to subscribe to the Face API service and get your key.
- A code editor such as Visual Studio Code

Write the script

Create a new file, *faceDetection.rb*, and add the following code. This code calls the Face API for a given image URL.

```
require 'net/http'

# replace <My Endpoint String> in the URL below with the string from your endpoint.
uri = URI('https://<My Endpoint String>.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
```

You'll need to update the `request['Ocp-Apim-Subscription-Key']` value with your subscription key and change the `uri` string so that it contains the correct endpoint.

> **NOTE**
>
> New resources created after July 1, 2019, will use custom subdomain names. For more information and a complete list of regional endpoints, see Custom subdomain names for Cognitive Services.

You may also wish to change the `imageUri` field to point to your own input image. You also may wish to change the

`returnFaceAttributes` field, which specifies which face attributes to retrieve.

Run the script

Run the Ruby script with the following command:

```
ruby faceDetection.rb
```

You should see a JSON string of detected face data printed to the console. The following text is an example of a successful JSON response.


```
[
  {
    "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"
```

```
        "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

In this quickstart, you wrote a Ruby script that calls the Azure Face API to detect faces in an image and return their attributes. Next, explore the Face API reference documentation to learn more.

Face API

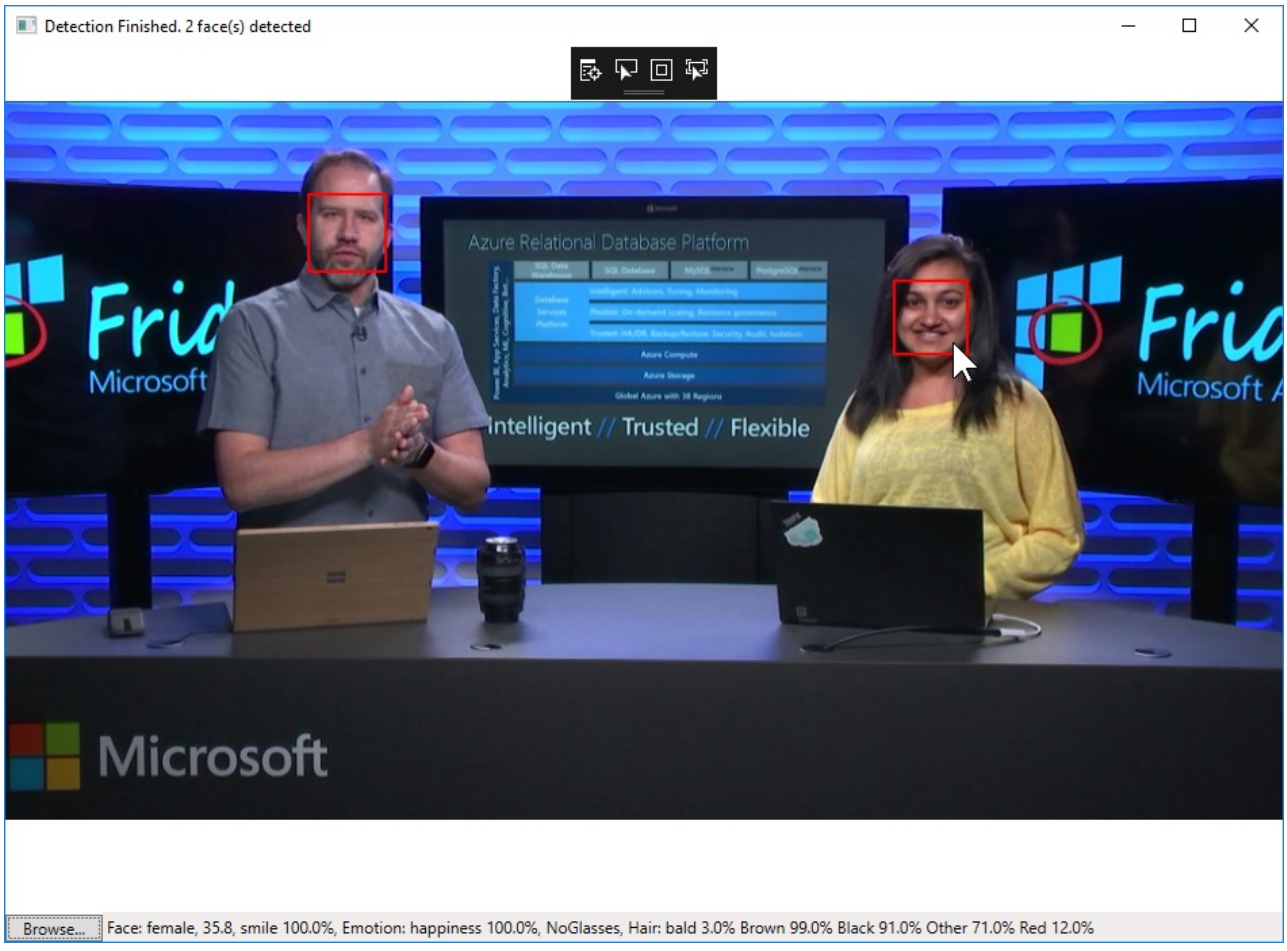
Tutorial: Create a WPF app to display face data in an image

9/10/2019 • 8 minutes to read • Edit Online

In this tutorial, you'll learn how to use the Azure Face API, through the .NET client SDK, to detect faces in an image and then present that data in the UI. You'll create a Windows Presentation Framework (WPF) application that detects faces, draws a frame around each face, and displays a description of the face in the status bar.

This tutorial shows you how to:

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



The complete sample code is available in the Cognitive Face CSharp sample repository on GitHub.

If you don't have an Azure subscription, create a free account before you begin.

Prerequisites

- A Face API subscription key. You can get a free trial subscription key from Try Cognitive Services. Or, follow the instructions in Create a Cognitive Services account to subscribe to the Face API service and get your key. Then, create environment variables for the key and service endpoint string, named `FACE_SUBSCRIPTION_KEY` and

`FACE_ENDPOINT`, respectively.

- Any edition of Visual Studio 2015 or 2017.

Create the Visual Studio project

Follow these steps to create a new WPF application project.

1. In Visual Studio, open the New Project dialog. Expand **Installed**, then **Visual C#**, then select **WPF App (.NET Framework)**.
2. Name the application **FaceTutorial**, then click **OK**.
3. Get the required NuGet packages. Right-click on your project in the Solution Explorer and select **Manage NuGet Packages**; then, find and install the following package:
   - Microsoft.Azure.CognitiveServices.Vision.Face 2.2.0-preview

Add the initial code

In this section, you will add the basic framework of the app without its face-specific features.

Create the UI

Open *MainWindow.xaml* and replace the contents with the following code—this code creates the UI window. The `FacePhoto_MouseMove` and `BrowseButton_Click` methods are event handlers that you will define later on.

```
<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>
```

Create the main class

Open *MainWindow.xaml.cs* and add the client library namespaces, along with other necessary namespaces.

```
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;
```

Next, insert the following code in the **MainWindow** class. This code creates a **FaceClient** instance using the subscription key and endpoint.

```
// Add your Face subscription key to your environment variables.
private const string subscriptionKey = Environment.GetEnvironmentVariable("FACE_SUBSCRIPTION_KEY");
// Add your Face endpoint to your environment variables.
private const string faceEndpoint = Environment.GetEnvironmentVariable("FACE_ENDPOINT");

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

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

private const string defaultStatusBarText =
    "Place the mouse pointer over a face to see the face description.";
```

Next add the **MainWindow** constructor. It checks your endpoint URL string and then associates it with the client object.

```
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);
    }
}
```

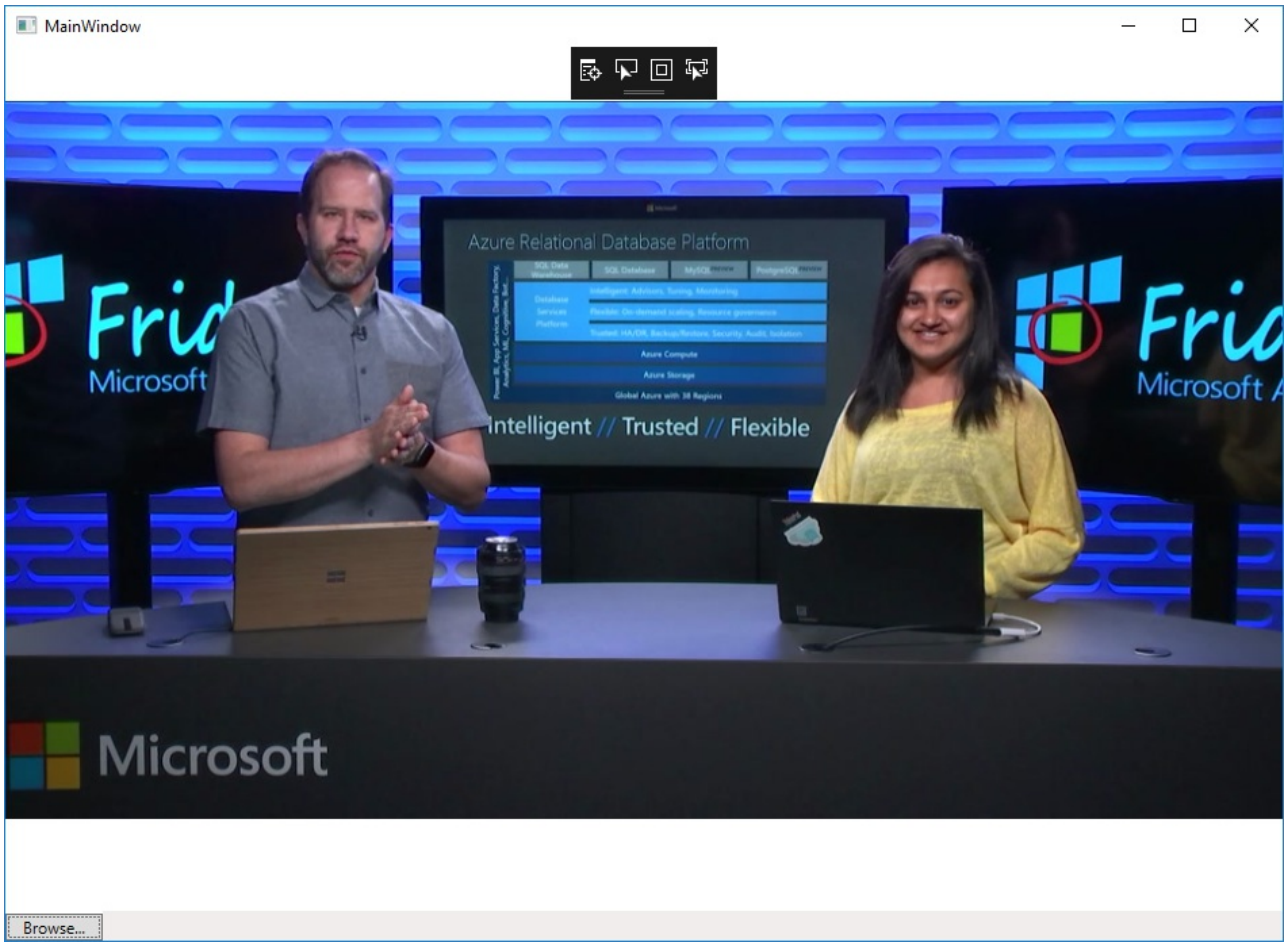
Finally, add the **BrowseButton_Click** and **FacePhoto_MouseMove** methods to the class. These methods correspond to the event handlers declared in *MainWindow.xaml*. The **BrowseButton_Click** method creates an **OpenFileDialog**, which allows the user to select a .jpg image. It then displays the image in the main window. You will insert the remaining code for **BrowseButton_Click** and **FacePhoto_MouseMove** in later steps. Also note the `faceList` reference—a list of **DetectedFace** objects. This reference is where your app will store and call the actual face data.

```
// 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);

    // 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)
{
```

Try the app

Press **Start** on the menu to test your app. When the app window opens, click **Browse** in the lower left corner. A **File Open** dialog should appear. Select an image from your filesystem and verify that it displays in the window. Then, close the app and advance to the next step.



Upload image and detect faces

Your app will detect faces by calling the **FaceClient.Face.DetectWithStreamAsync** method, which wraps the Detect REST API for uploading a local image.

Insert the following method in the **MainWindow** class, below the **FacePhoto_MouseMove** method. This method defines a list of face attributes to retrieve and reads the submitted image file into a **Stream**. Then it passes both of these objects to the **DetectWithStreamAsync** method 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 faces

Next, you will add the code to draw a rectangle around each detected face in the image. In the **MainWindow** class, insert the following code at the end of the **BrowseButton_Click** method, after the `FacePhoto.Source = bitmapSource` line. This code populates a list of detected faces from the call to **UploadAndDetectFaces**. Then it draws a rectangle around each face and displays the modified image 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;
    // Some images don't contain dpi info.
    resizeFactor = (dpi == 0) ? 1 : 96 / dpi;
    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 = defaultStatusBarText;
}
```

Describe the faces

Add the following method to the **MainWindow** class, below the **UploadAndDetectFaces** method. This method converts the retrieved face attributes into a string describing the face.

```
// 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();
}
```

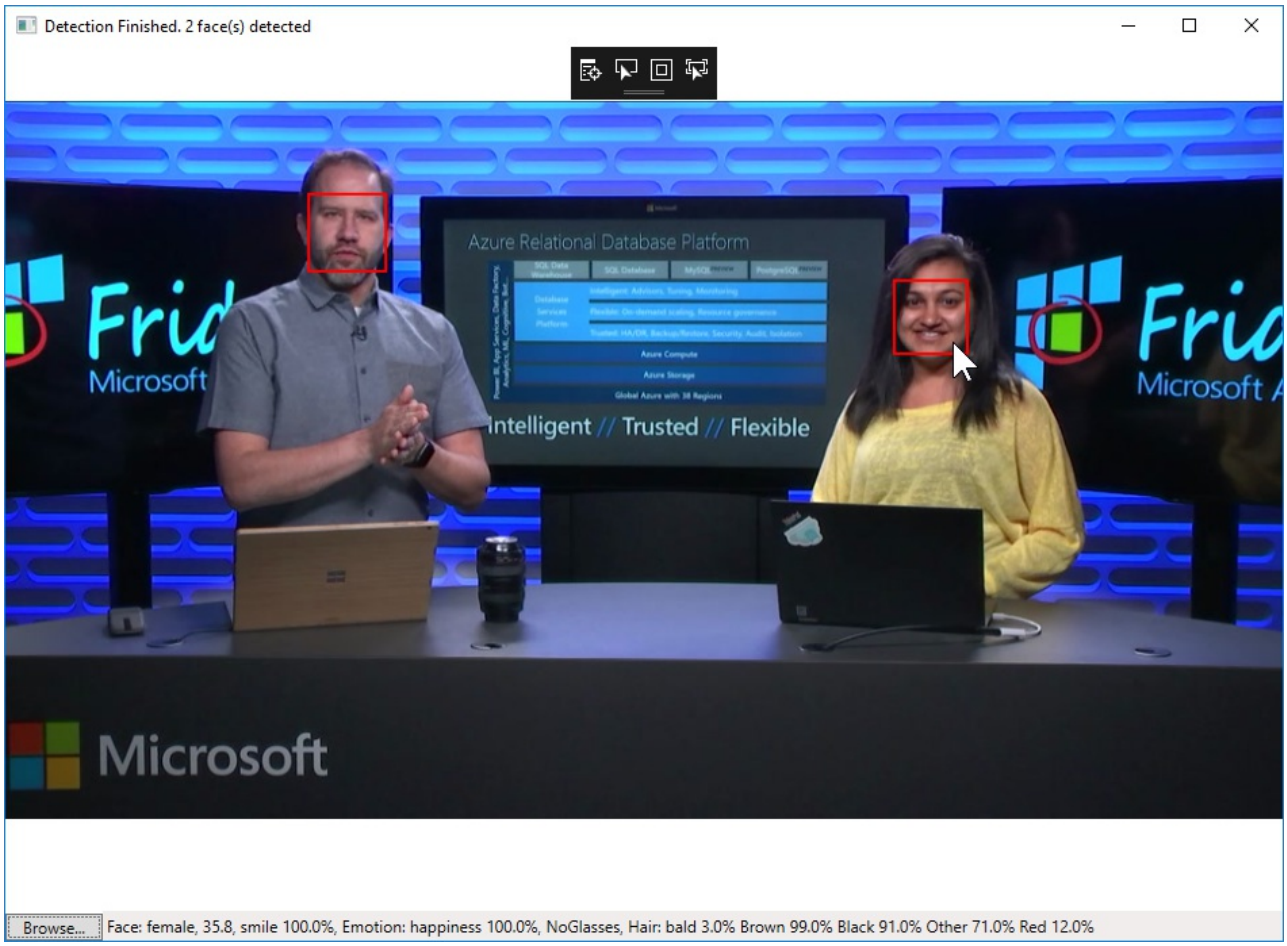
Display the face description

Add the following code to the **FacePhoto_MouseMove** method. This event handler displays the face description string in `faceDescriptionStatusBar` when the cursor hovers over a detected face rectangle.

```
// 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 = defaultStatusBarText;
```

Run the app

Run the application and browse for an image containing a face. Wait a few seconds to allow the Face service to respond. You should see a red rectangle on each of the faces in the image. If you move the mouse over a face rectangle, the description of that face should appear in the status bar.



Next steps

In this tutorial, you learned the basic process for using the Face service .NET SDK and created an application to detect and frame faces in an image. Next, learn more about the details of face detection.

How to Detect Faces in an Image

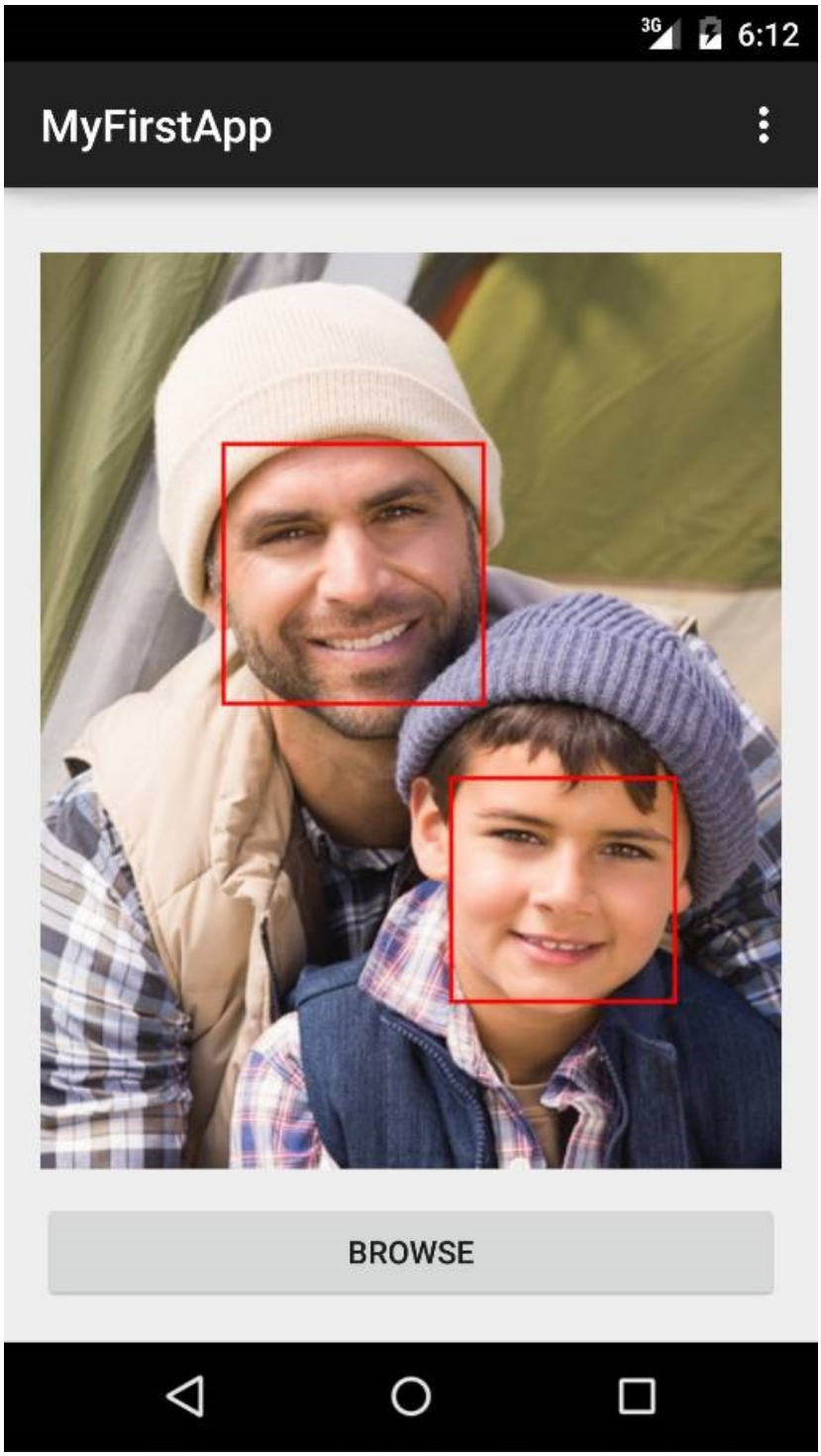
Tutorial: Create an Android app to detect and frame faces in an image

9/10/2019 • 5 minutes to read • Edit Online

In this tutorial, you will create a simple Android application that uses the Azure Face API, through the Java SDK, to detect human faces in an image. The application displays a selected image and draws a frame around each detected face.

This tutorial shows you how to:

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



The complete sample code is available in the Cognitive Services Face Android repository on GitHub.

If you don't have an Azure subscription, create a free account before you begin.

Prerequisites

- A Face API subscription key. You can get a free trial subscription key from Try Cognitive Services. Or, follow the instructions in Create a Cognitive Services account to subscribe to the Face API service and get your key. Then, create environment variables for the key and service endpoint string, named `FACE_SUBSCRIPTION_KEY` and `FACE_ENDPOINT`, respectively.
- Any edition of Visual Studio 2015 or 2017.
- Android Studio with API level 22 or later (required by the Face client library).

Create the Android Studio project

Follow these steps to create a new Android application project.

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

Add the initial code

Create the UI

Open *activity_main.xml*. In 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" tools:context=".MainActivity"
    android:layout_width="match_parent" android:layout_height="match_parent">

    <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 a face image"
        android:id="@+id/button1"
        android:layout_alignParentBottom="true"/>
</RelativeLayout>
```

Create the main class

Open *MainActivity.java* and replace the existing `import` statements with the following code.

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

Then, replace the contents of the **MainActivity** class with the following code. This creates an event handler on the **Button** that starts a new activity to allow the user to select a picture. It displays the picture in the **ImageView**.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    Button button1 = 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 = findViewById(R.id.imageView1);
            imageView.setImageBitmap(bitmap);

            // Comment out for tutorial
            detectAndFrame(bitmap);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

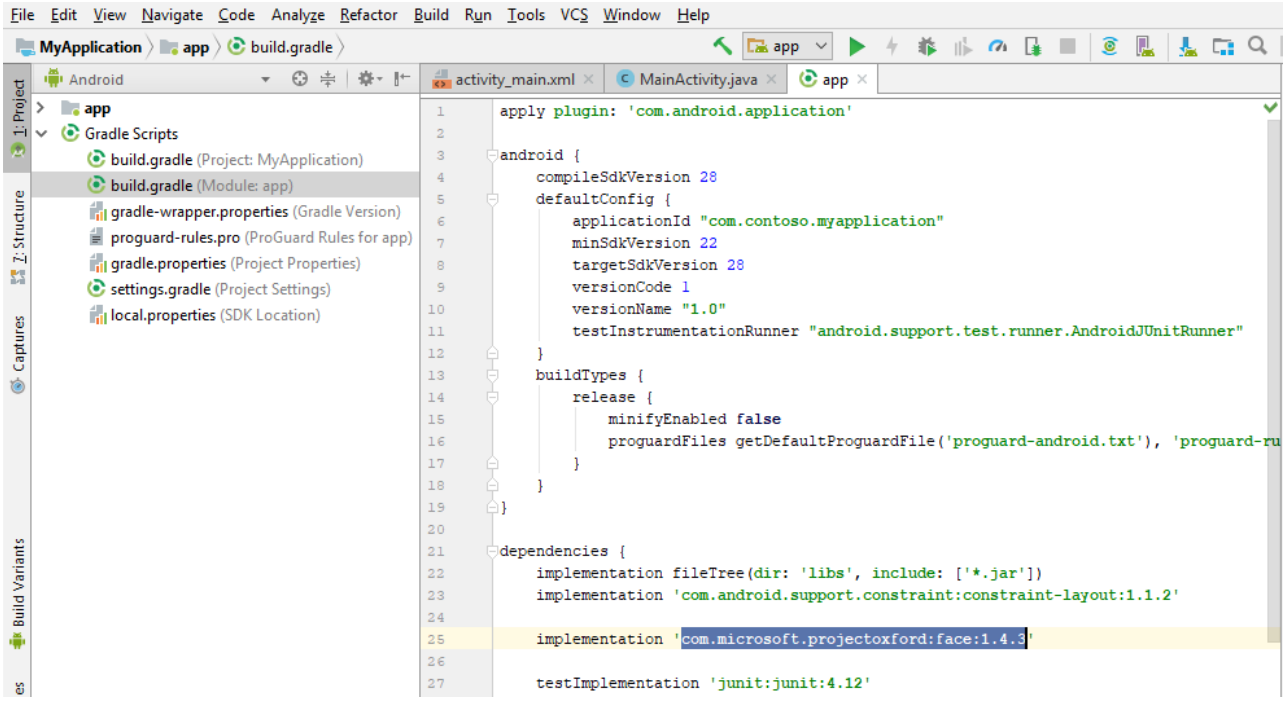
Try the app

Comment out the call to **detectAndFrame** in the **onActivityResult** method. Then press **Run** on the menu to test your app. When the app opens, either in an emulator or a connected device, click the **Browse** on the bottom. The device's file selection dialog should appear. Choose an image and verify that it displays in the window. Then, close the app and advance to the next step.

Add the Face SDK

Add the Gradle dependency

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**.



Add the Face-related project code

Go back to **MainActivity.java** and add the following `import` statements:

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

Then, insert the following code in the **MainActivity** class, above the **onCreate** method:

```
// Add your Face endpoint to your environment variables.
private final String apiEndpoint = System.getenv("FACE_ENDPOINT");
// Add your Face subscription key to your environment variables.
private final String subscriptionKey = System.getenv("FACE_SUBSCRIPTION_KEY");

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

private final int PICK_IMAGE = 1;
private ProgressDialog detectionProgressDialog;
```

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" />
```

Upload image and detect faces

Your app will detect faces by calling the **faceClient.Face.DetectWithStreamAsync** method, which wraps the Detect REST API and returns a list of **Face** instances.

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

Insert the following two methods into the **MainActivity** class. Note that when face detection completes, the app calls the **drawFaceRectanglesOnBitmap** method to modify the **ImageView**. You will define this method next.

```
// Detect faces by uploading a face image.
```

```
// 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
                    detectionProgressDialog.show();
                }
                @Override
                protected void onProgressUpdate(String... progress) {
                    //TODO: update progress
                    detectionProgressDialog.setMessage(progress[0]);
                }
                @Override
                protected void onPostExecute(Face[] result) {
                    //TODO: update face frames
                    detectionProgressDialog.dismiss();

                    if(!exceptionMessage.equals("")){
                        showError(exceptionMessage);
                    }
                    if (result == null) return;

                    ImageView imageView = findViewById(R.id.imageView1);
                    imageView.setImageBitmap(
                            drawFaceRectanglesOnBitmap(imageBitmap, result));
                    imageBitmap.recycle();
                }
            };
```

```
    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();
}
```

Draw face rectangles

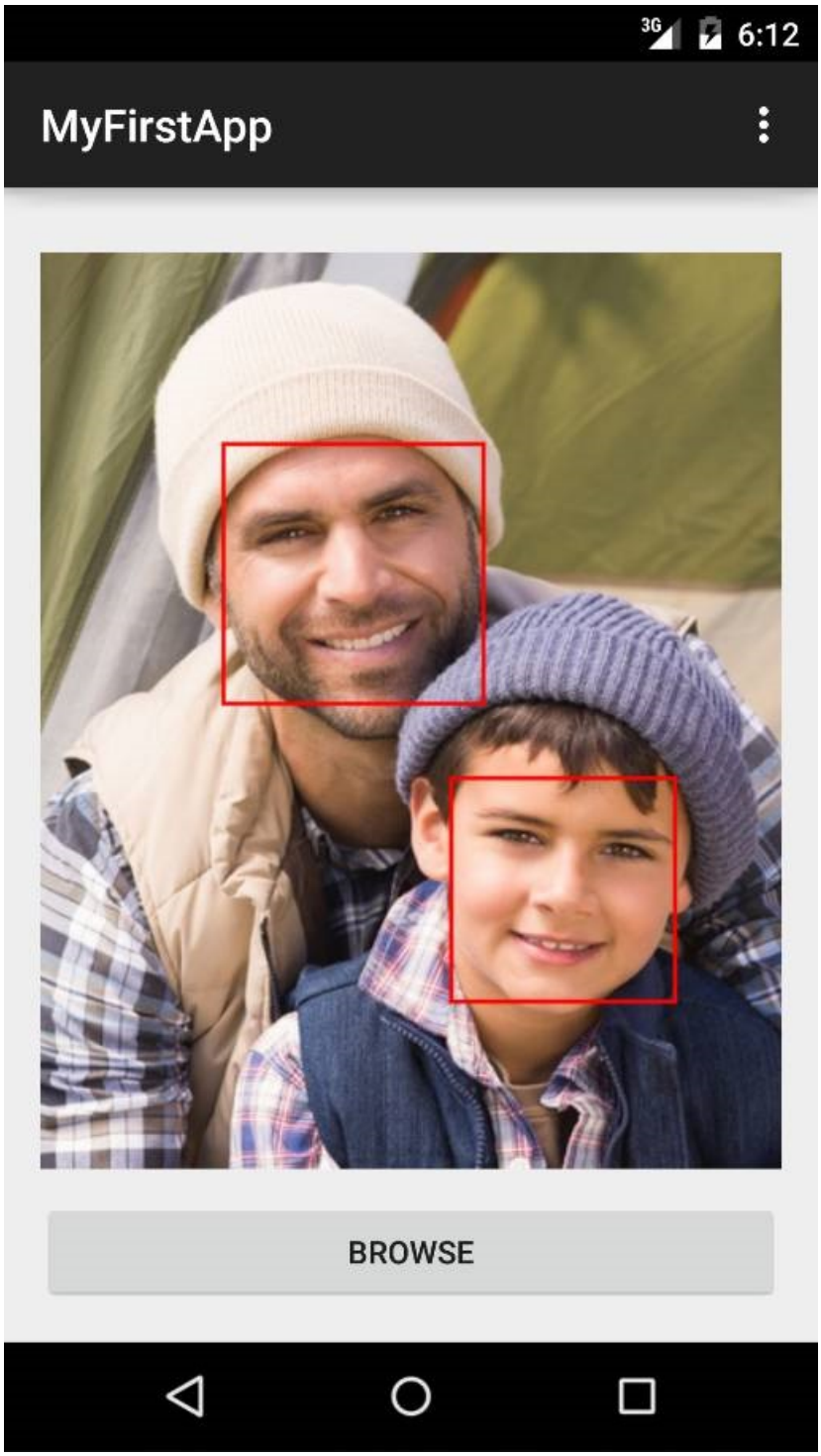
Insert the following helper method into the **MainActivity** class. This method draws a rectangle around each detected face, using the rectangle coordinates of each **Face** instance.

```
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;
}
```

Finally, uncomment the call to the **detectAndFrame** method in **onActivityResult**.

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. You should see a red rectangle on each of the faces in the image.



Next steps

In this tutorial, you learned the basic process for using the Face API Java SDK and created an application to detect and frame faces in an image. Next, learn more about the details of face detection.

How to Detect Faces in an Image

Face detection and attributes

5/20/2019 • 4 minutes to read • Edit Online

This article explains the concepts of face detection and face attribute data. Face detection is the action of locating human faces in an image and optionally returning different kinds of face-related data.

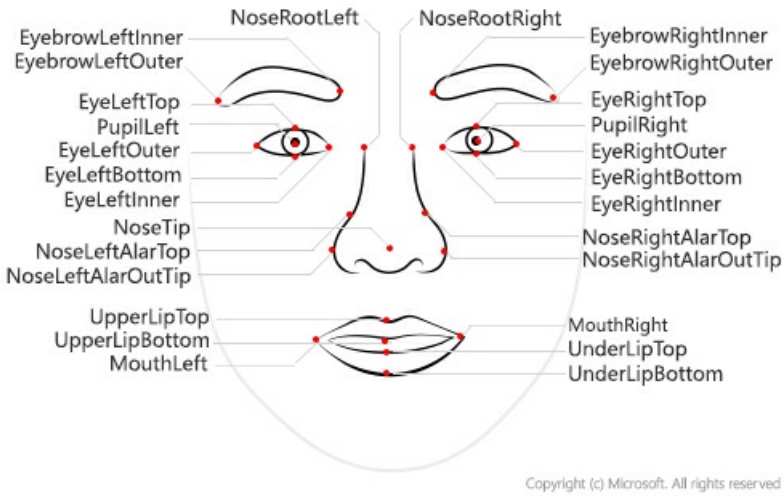
You use the Face - Detect operation to detect faces in an image. At a minimum, each detected face corresponds to a faceRectangle field in the response. This set of pixel coordinates for the left, top, width, and height mark the located face. Using these coordinates, you can get the location of the face and its size. In the API response, faces are listed in size order from largest to smallest.

Face ID

The face ID is a unique identifier string for each detected face in an image. You can request a face ID in your Face - Detect API call.

Face landmarks

Face landmarks are a set of easy-to-find points on a face, such as the pupils or the tip of the nose. By default, there are 27 predefined landmark points. The following figure shows all 27 points:



The coordinates of the points are returned in units of pixels.

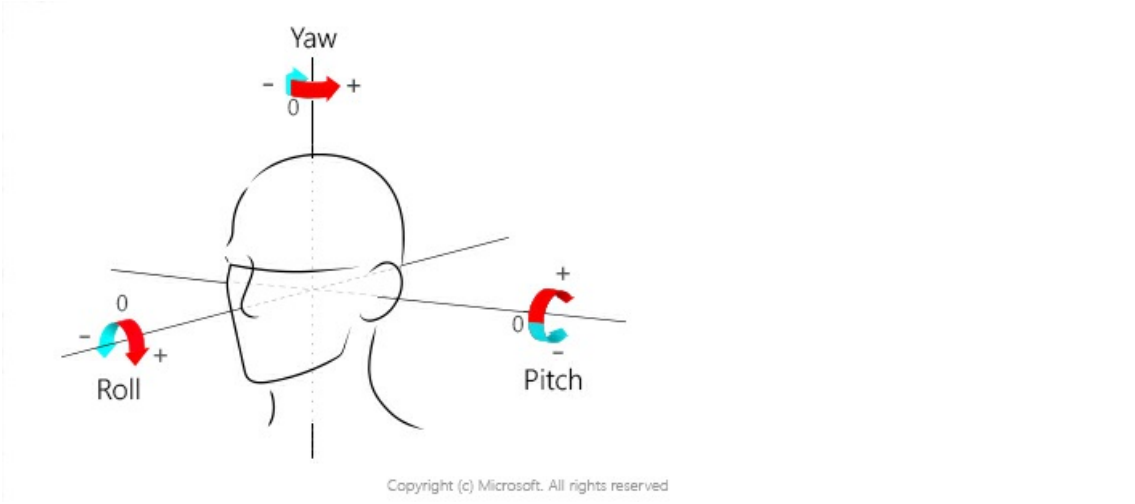
Attributes

Attributes are a set of features that can optionally be detected by the Face - Detect API. The following attributes can be detected:

- **Age**. The estimated age in years of a particular face.
- **Blur**. The blurriness of the face in the image. This attribute returns a value between zero and one and an informal rating of low, medium, or high.
- **Emotion**. A list of emotions with their detection confidence for the given face. Confidence scores are normalized, and the scores across all emotions add up to one. The emotions returned are happiness, sadness, neutral, anger, contempt, disgust, surprise, and fear.
- **Exposure**. The exposure of the face in the image. This attribute returns a value between zero and one and

an informal rating of underExposure, goodExposure, or overExposure.

- **Facial hair**. The estimated facial hair presence and the length for the given face.
- **Gender**. The estimated gender of the given face. Possible values are male, female, and genderless.
- **Glasses**. Whether the given face has eyeglasses. Possible values are NoGlasses, ReadingGlasses, Sunglasses, and Swimming Goggles.
- **Hair**. The hair type of the face. This attribute shows whether the hair is visible, whether baldness is detected, and what hair colors are detected.
- **Head pose**. The face's orientation in 3D space. This attribute is described by the pitch, roll, and yaw angles in degrees. The value ranges are -90 degrees to 90 degrees, -180 degrees to 180 degrees, and -90 degrees to 90 degrees, respectively. See the following diagram for angle mappings:



- **Makeup**. Whether the face has makeup. This attribute returns a Boolean value for eyeMakeup and lipMakeup.
- **Noise**. The visual noise detected in the face image. This attribute returns a value between zero and one and an informal rating of low, medium, or high.
- **Occlusion**. Whether there are objects blocking parts of the face. This attribute returns a Boolean value for eyeOccluded, foreheadOccluded, and mouthOccluded.
- **Smile**. The smile expression of the given face. This value is between zero for no smile and one for a clear smile.

> **IMPORTANT**
> Face attributes are predicted through the use of statistical algorithms. They might not always be accurate. Use caution when you make decisions based on attribute data.

Input data

Use the following tips to make sure that your input images give the most accurate detection results:

- The supported input image formats are JPEG, PNG, GIF for the first frame, and BMP.
- The image file size should be no larger than 4 MB.
- The detectable face size range is 36 x 36 to 4096 x 4096 pixels. Faces outside of this range won't be detected.
- Some faces might not be detected because of technical challenges. Extreme face angles (head pose) or face occlusion (objects such as sunglasses or hands that block part of the face) can affect detection. Frontal and

near-frontal faces give the best results.

If you're detecting faces from a video feed, you may be able to improve performance by adjusting certain settings on your video camera:

- **Smoothing**: Many video cameras apply a smoothing effect. You should turn this off if you can because it creates a blur between frames and reduces clarity.
- **Shutter Speed**: A faster shutter speed reduces the amount of motion between frames and makes each frame clearer. We recommend shutter speeds of 1/60 second or faster.
- **Shutter Angle**: Some cameras specify shutter angle instead of shutter speed. You should use a lower shutter angle if possible. This will result in clearer video frames.

> **NOTE**
> A camera with a lower shutter angle will receive less light in each frame, so the image will be darker. You'll need to determine the right level to use.

Next steps

Now that you're familiar with face detection concepts, learn how to write a script that detects faces in a given image.

- Detect faces in an image

Face recognition concepts

5/20/2019 • 2 minutes to read • Edit Online

This article explains the concepts of the Verify, Find Similar, Group, and Identify face recognition operations and the underlying data structures. Broadly, recognition describes the work of comparing two different faces to determine if they're similar or belong to the same person.

Recognition-related data structures

The recognition operations use mainly the following data structures. These objects are stored in the cloud and can be referenced by their ID strings. ID strings are always unique within a subscription. Name fields can be duplicated.

NAME	DESCRIPTION
DetectedFace	This single face representation is retrieved by the face detection operation. Its ID expires 24 hours after it's created.
PersistedFace	When DetectedFace objects are added to a group, such as FaceList or Person, they become PersistedFace objects. They can be retrieved at any time and don't expire.
FaceList or LargeFaceList	This data structure is an assorted list of PersistedFace objects. A FaceList has a unique ID, a name string, and optionally a user data string.
Person	This data structure is a list of PersistedFace objects that belong to the same person. It has a unique ID, a name string, and optionally a user data string.
PersonGroup or LargePersonGroup	This data structure is an assorted list of Person objects. It has a unique ID, a name string, and optionally a user data string. A PersonGroup must be trained before it can be used in recognition operations.

Recognition operations

This section details how the four recognition operations use the data structures previously described. For a broad description of each recognition operation, see Overview.

Verify

The Verify operation takes a face ID from DetectedFace or PersistedFace and either another face ID or a Person object and determines whether they belong to the same person. If you pass in a Person object, you can optionally pass in a PersonGroup to which that Person belongs to improve performance.

Find Similar

The Find Similar operation takes a face ID from DetectedFace or PersistedFace and either a FaceList or an array of other face IDs. With a FaceList, it returns a smaller FaceList of faces that are similar to the given face. With an array of face IDs, it similarly returns a smaller array.

Group

The Group operation takes an array of assorted face IDs from DetectedFace or PersistedFace and returns the same IDs grouped into several smaller arrays. Each "groups" array contains face IDs that appear similar. A single "messyGroup" array contains face IDs for which no similarities were found.

Identify

The Identify operation takes one or several face IDs from DetectedFace or PersistedFace and a PersonGroup and returns a list of Person objects that each face might belong to. Returned Person objects are wrapped as Candidate objects, which have a prediction confidence value.

Input data

Use the following tips to ensure that your input images give the most accurate recognition results:

- The supported input image formats are JPEG, PNG, GIF (the first frame), BMP.
- Image file size should be no larger than 4 MB.
- When you create Person objects, use photos that feature different kinds of angles and lighting.
- Some faces might not be recognized because of technical challenges, such as:
  - Images with extreme lighting, for example, severe backlighting.
  - Obstructions that block one or both eyes.
  - Differences in hair type or facial hair.
  - Changes in facial appearance because of age.
  - Extreme facial expressions.

Next steps

Now that you're familiar with face recognition concepts, learn how to write a script that identifies faces against a trained PersonGroup.

- Identify faces in images

Get face detection data

9/4/2019 • 3 minutes to read • Edit Online

This guide demonstrates how to use face detection to extract attributes like gender, age, or pose from a given image. The code snippets in this guide are written in C# by using the Azure Cognitive Services Face API client library. The same functionality is available through the REST API.

This guide shows you how to:

- Get the locations and dimensions of faces in an image.
- Get the locations of various face landmarks, such as pupils, nose, and mouth, in an image.
- Guess the gender, age, emotion, and other attributes of a detected face.

Setup

This guide assumes that you already constructed a FaceClient object, named `faceClient`, with a Face subscription key and endpoint URL. From here, you can use the face detection feature by calling either DetectWithUrlAsync, which is used in this guide, or DetectWithStreamAsync. For instructions on how to set up this feature, follow one of the quickstarts.

This guide focuses on the specifics of the Detect call, such as what arguments you can pass and what you can do with the returned data. We recommend that you query for only the features you need. Each operation takes additional time to complete.

Get basic face data

To find faces and get their locations in an image, call the method with the *returnFaceId* parameter set to **true**. This setting is the default.

```
IList<DetectedFace> faces = await faceClient.Face.DetectWithUrlAsync(imageUrl, true, false, null);
```

You can query the returned DetectedFace objects for their unique IDs and a rectangle that gives the pixel coordinates of the face.

```
foreach (var face in faces)
{
    string id = face.FaceId.ToString();
    FaceRectangle rect = face.FaceRectangle;
}
```

For information on how to parse the location and dimensions of the face, see FaceRectangle. Usually, this rectangle contains the eyes, eyebrows, nose, and mouth. The top of head, ears, and chin aren't necessarily included. To use the face rectangle to crop a complete head or get a mid-shot portrait, perhaps for a photo ID-type image, you can expand the rectangle in each direction.

Get face landmarks

Face landmarks are a set of easy-to-find points on a face, such as the pupils or the tip of the nose. To get face landmark data, set the *returnFaceLandmarks* parameter to **true**.

```
IList<DetectedFace> faces = await faceClient.Face.DetectWithUrlAsync(imageUrl, true, true, null);
```

The following code demonstrates how you might retrieve the locations of the nose and pupils:

```
foreach (var face in faces)
{
    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;
}
```

You also can use face landmarks data to accurately calculate the direction of the face. For example, you can define the rotation of the face as a vector from the center of the mouth to the center of the eyes. The following code calculates this vector:

```
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);
```

When you know the direction of the face, you can rotate the rectangular face frame to align it more properly. To crop faces in an image, you can programmatically rotate the image so that the faces always appear upright.

Get face attributes

Besides face rectangles and landmarks, the face detection API can analyze several conceptual attributes of a face. For a full list, see the Face attributes conceptual section.

To analyze face attributes, set the *returnFaceAttributes* parameter to a list of FaceAttributeType Enum values.

```
var requiredFaceAttributes = new FaceAttributeType[] {
    FaceAttributeType.Age,
    FaceAttributeType.Gender,
    FaceAttributeType.Smile,
    FaceAttributeType.FacialHair,
    FaceAttributeType.HeadPose,
    FaceAttributeType.Glasses,
    FaceAttributeType.Emotion
};
var faces = await faceClient.DetectWithUrlAsync(imageUrl, true, false, requiredFaceAttributes);
```

Then, get references to the returned data and do more operations according to your needs.

```
foreach (var face in faces)
{
    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;
    var emotion = attributes.Emotion;
}
```

To learn more about each of the attributes, see the Face detection and attributes conceptual guide.

Next steps

In this guide, you learned how to use the various functionalities of face detection. Next, integrate these features into your app by following an in-depth tutorial.

- Tutorial: Create a WPF app to display face data in an image
- Tutorial: Create an Android app to detect and frame faces in an image

Related topics

- Reference documentation (REST)
- Reference documentation (.NET SDK)

Example: Identify faces in images

9/25/2019 • 6 minutes to read • Edit Online

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

Preparation

This sample demonstrates:

- 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. We recommend that you use clear frontal views of faces. An example is a photo ID. A good set of photos includes faces of the same person in different poses, clothing colors, or hairstyles.

To carry out the demonstration of this sample, prepare:

- A few photos with the person's face. Download sample photos for Anna, Bill, and Clare.
- A series of test photos. The photos might or might not contain the faces of Anna, Bill, or Clare. They're used to test identification. 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 a query string, see 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>
```

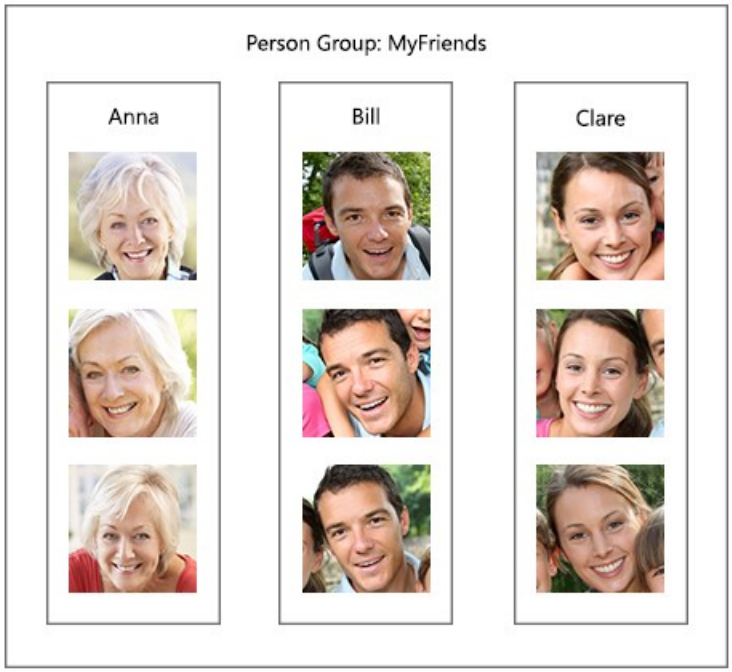
As an alternative, specify the subscription key in the HTTP request header **ocp-apim-subscription-key: <Subscription Key>**. When you use a client library, the subscription key is passed in through the constructor of the FaceClient class. For example:

```
private readonly IFaceClient faceClient = new FaceClient(
            new ApiKeyServiceClientCredentials("<subscription key>"),
            new System.Net.Http.DelegatingHandler[] { });
```

To get the subscription key, go to the Azure Marketplace from the Azure portal. For more information, see Subscriptions.

Step 2: Create the PersonGroup

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



Step 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. A PersonGroup is a collection of people. Each person is defined within a particular PersonGroup. Identification is done against a PersonGroup. The task is to create a PersonGroup, and then create the people in it, such as Anna, Bill, and Clare.

First, create a new PersonGroup by using the PersonGroup - Create API. The corresponding client library API is the CreatePersonGroupAsync method for the FaceClient class. The group ID that's specified to create the group is unique for each subscription. You also can get, update, or delete PersonGroups by using other PersonGroup APIs.

After a group is defined, you can define people within it by using the PersonGroup Person - Create API. The client library method is CreatePersonAsync. You can add a face to each person after they're created.

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

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

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

Step 2.2: Detect faces and register them to the 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 you use the client library, face detection is done through one of the Detect..Async methods of the FaceClient class.

For each face that's detected, 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 faceClient.PersonGroupPerson.AddFaceFromStreamAsync(
            personGroupId, friend1.PersonId, s);
    }
}
// Do the same for Bill and Clare
```

If the image contains more than one face, only the largest face is added. You can add other faces to the person. Pass a string in the format of "targetFace = left, top, width, height" to PersonGroup Person - Add Face API's targetFace query parameter. You also can use the targetFace optional parameter for the AddPersonFaceAsync method to add other faces. Each face added to the person is given a unique persisted face ID. You can use this ID in PersonGroup Person – Delete Face and Face – Identify.

Step 3: Train the PersonGroup

The PersonGroup must be trained before an identification can be performed by using it. The PersonGroup must be retrained after you add or remove any person or if you edit a person's registered face. The training is done by the PersonGroup – Train API. When you use the client library, it's a call to the TrainPersonGroupAsync method:

```
await faceClient.PersonGroup.TrainAsync(personGroupId);
```

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

```
TrainingStatus trainingStatus = null;
while(true)
{
    trainingStatus = await faceClient.PersonGroup.GetTrainingStatusAsync(personGroupId);

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

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

Step 4: Identify a face against a defined PersonGroup

When the Face API performs identifications, it computes the similarity of a test face among all the faces within a group. It returns the most comparable persons for the testing face. This process is done through the Face - Identify API or the IdentifyAsync method of the client library.

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

The following code demonstrates the identification process:

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

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

    var results = await faceClient.Face.IdentifyAsync(faceIds, personGroupId);
    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 faceClient.PersonGroupPerson.GetAsync(personGroupId, candidateId);
            Console.WriteLine("Identified as {0}", person.Name);
        }
    }
}
```

After you finish the steps, try to identify different faces. See if the faces of Anna, Bill, or Clare can be correctly identified according to the images uploaded for face detection. See the following examples:

Step 5: Request for large scale

A PersonGroup can hold up to 10,000 persons based on the previous design limitation. For more information about up to million-scale scenarios, see How to use the large-scale feature.

Summary

In this guide, you learned the process of creating a PersonGroup and identifying a person. The following features were explained and demonstrated:

- Detect faces by using the Face - Detect API.
- Create PersonGroups by using the PersonGroup - Create API.
- Create persons by using the PersonGroup Person - Create API.
- Train a PersonGroup by using the PersonGroup – Train API.

- Identify unknown faces against the PersonGroup by using the Face - Identify API.

Related topics

- Face recognition concepts
- Detect faces in an image
- Add faces
- Use the large-scale feature

Add faces to a PersonGroup

6/26/2019 • 2 minutes to read • Edit Online

This guide demonstrates how to add a large number of persons and faces to a PersonGroup object. The same strategy also applies to LargePersonGroup, FaceList, and LargeFaceList objects. This sample is written in C# by using the Azure Cognitive Services Face API .NET client library.

Step 1: Initialization

The following code declares several variables and implements a helper function to schedule the face add 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()` waits until it's valid to send the 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 you use a client library, you must pass your subscription key to the constructor of the **FaceClient** class. For example:

```
private readonly IFaceClient faceClient = new FaceClient(
    new ApiKeyServiceClientCredentials("<SubscriptionKey>"),
    new System.Net.Http.DelegatingHandler[] { });
```

To get the subscription key, go to the Azure Marketplace from the Azure portal. For more information, 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 faceClient.LargePersonGroup.CreateAsync(personGroupId, personGroupName);
```

Step 4: Create the persons for 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 faceClient.PersonGroupPerson.CreateAsync(personGroupId, personName);
});
```

Step 5: Add faces to the persons

Faces added to different persons are processed concurrently. Faces added for one specific person are processed sequentially. Again, `await WaitCallLimitPerSecondAsync()` is invoked to ensure that 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 faceClient.PersonGroupPerson.AddFaceFromStreamAsync(personGroupId, personId, stream);
        }
    }
});
```

Summary

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

- This strategy also applies to FaceLists and LargePersonGroups.
- Adding or deleting faces to different FaceLists or persons in LargePersonGroups are processed concurrently.
- Adding or deleting faces to one specific FaceList or person in a LargePersonGroup are done sequentially.
- For simplicity, how to handle a potential exception is omitted in this guide. If you want to enhance more robustness, apply the proper retry policy.

The following features were explained and demonstrated:

- Create PersonGroups by using the PersonGroup - Create API.
- Create persons by using the PersonGroup Person - Create API.
- Add faces to persons by using the PersonGroup Person - Add Face API.

Related topics

- Identify faces in an image
- Detect faces in an image
- Use the large-scale feature

Use the HeadPose attribute

6/11/2019 • 3 minutes to read • Edit Online

In this guide, you'll see how you can use the HeadPose attribute of a detected face to enable some key scenarios.

Rotate the face rectangle

The face rectangle, returned with every detected face, marks the location and size of the face in the image. By default, the rectangle is always aligned with the image (its sides are vertical and horizontal); this can be inefficient for framing angled faces. In situations where you want to programmatically crop faces in an image, it's better to be able to rotate the rectangle to crop.

The Cognitive Services Face WPF sample app uses the HeadPose attribute to rotate its detected face rectangles.

Explore the sample code

You can programmatically rotate the face rectangle by using the HeadPose attribute. If you specify this attribute when detecting faces (see How to detect faces), you will be able to query it later. The following method from the Cognitive Services Face WPF app takes a list of **DetectedFace** objects and returns a list of **Face** objects. **Face** here is a custom class that stores face data, including the updated rectangle coordinates. New values are calculated for **top**, **left**, **width**, and **height**, and a new field **FaceAngle** specifies the rotation.

```
/// <summary>
/// Calculate the rendering face rectangle
/// </summary>
/// <param name="faces">Detected face from service</param>
/// <param name="maxSize">Image rendering size</param>
/// <param name="imageInfo">Image width and height</param>
/// <returns>Face structure for rendering</returns>
public static IEnumerable<Face> CalculateFaceRectangleForRendering(IList<DetectedFace> faces, int maxSize,
Tuple<int, int> imageInfo)
{
    var imageWidth = imageInfo.Item1;
    var imageHeight = imageInfo.Item2;
    var ratio = (float)imageWidth / imageHeight;
    int uiWidth = 0;
    int uiHeight = 0;
    if (ratio > 1.0)
    {
        uiWidth = maxSize;
        uiHeight = (int)(maxSize / ratio);
    }
    else
    {
        uiHeight = maxSize;
        uiWidth = (int)(ratio * uiHeight);
    }

    var uiXOffset = (maxSize - uiWidth) / 2;
    var uiYOffset = (maxSize - uiHeight) / 2;
    var scale = (float)uiWidth / imageWidth;

    foreach (var face in faces)
    {
        var left = (int)(face.FaceRectangle.Left * scale + uiXOffset);
        var top = (int)(face.FaceRectangle.Top * scale + uiYOffset);

        // Angle of face rectangles, default value is 0 (not rotated).
        double faceAngle = 0;
```

```
        // If head pose attributes have been obtained, re-calculate the left & top (X & Y) positions.
        if (face.FaceAttributes?.HeadPose != null)
        {
            // Head pose's roll value acts directly as the face angle.
            faceAngle = face.FaceAttributes.HeadPose.Roll;
            var angleToPi = Math.Abs((faceAngle / 180) * Math.PI);

            // _____       | / \ |
            // |____|  =>  |/   /|
            //             | \ / |
            // Re-calculate the face rectangle's left & top (X & Y) positions.
            var newLeft = face.FaceRectangle.Left +
                face.FaceRectangle.Width / 2 -
                (face.FaceRectangle.Width * Math.Sin(angleToPi) + face.FaceRectangle.Height *
Math.Cos(angleToPi)) / 2;

            var newTop = face.FaceRectangle.Top +
                face.FaceRectangle.Height / 2 -
                (face.FaceRectangle.Height * Math.Sin(angleToPi) + face.FaceRectangle.Width *
Math.Cos(angleToPi)) / 2;

            left = (int)(newLeft * scale + uiXOffset);
            top = (int)(newTop * scale + uiYOffset);
        }

        yield return new Face()
        {
            FaceId = face.FaceId?.ToString(),
            Left = left,
            Top = top,
            OriginalLeft = (int)(face.FaceRectangle.Left * scale + uiXOffset),
            OriginalTop = (int)(face.FaceRectangle.Top * scale + uiYOffset),
            Height = (int)(face.FaceRectangle.Height * scale),
            Width = (int)(face.FaceRectangle.Width * scale),
            FaceAngle = faceAngle,
        };
    }
}
```

Display the updated rectangle

From here, you can use the returned **Face** objects in your display. The following lines from FaceDetectionPage.xaml show how the new rectangle is rendered from this data:

```
 <DataTemplate>
    <Rectangle Width="{Binding Width}" Height="{Binding Height}" Stroke="#FF26B8F4" StrokeThickness="1">
        <Rectangle.LayoutTransform>
            <RotateTransform Angle="{Binding FaceAngle}"/>
        </Rectangle.LayoutTransform>
    </Rectangle>
</DataTemplate>
```

Detect head gestures

You can detect head gestures like nodding and head shaking by tracking HeadPose changes in real time. You can use this feature as a custom liveness detector.

Liveness detection is the task of determining that a subject is a real person and not an image or video representation. A head gesture detector could serve as one way to help verify liveness, especially as opposed to an image representation of a person.

Caution

To detect head gestures in real time, you'll need to call the Face API at a high rate (more than once per second). If

you have a free-tier (f0) subscription, this will not be possible. If you have a paid-tier subscription, make sure you've calculated the costs of making rapid API calls for head gesture detection.

See the Face API HeadPose Sample on GitHub for a working example of head gesture detection.

Next steps

See the Cognitive Services Face WPF app on GitHub for a working example of rotated face rectangles. Or, see the Face API HeadPose Sample app, which tracks the HeadPose attribute in real time to detect head movements.

Specify a face recognition model

9/4/2019 • 5 minutes to read • Edit Online

This guide shows you how to specify a face recognition model for face detection, identification and similarity search using the Azure Face API.

The Face API uses machine learning models to perform operations on human faces in images. We continue to improve the accuracy of our models based on customer feedback and advances in research, and we deliver these improvements as model updates. Developers have the option to specify which version of the face recognition model they'd like to use; they can choose the model that best fits their use case.

If you are a new user, we recommend you use the latest model. Read on to learn how to specify it in different Face operations while avoiding model conflicts. If you are an advanced user and are not sure whether you should switch to the latest model, skip to the Evaluate different models section to evaluate the new model and compare results using your current data set.

Prerequisites

You should be familiar with the concepts of AI face detection and identification. If you aren't, see these how-to guides first:

- How to detect faces in an image
- How to identify faces in an image

Detect faces with specified model

Face detection identifies the visual landmarks of human faces and finds their bounding-box locations. It also extracts the face's features and stores them for use in identification. All of this information forms the representation of one face.

The recognition model is used when the face features are extracted, so you can specify a model version when performing the Detect operation.

When using the Face - Detect API, assign the model version with the `recognitionModel` parameter. The available values are:

- `recognition_01`
- `recognition_02`

Optionally, you can specify the *returnRecognitionModel* parameter (default **false**) to indicate whether *recognitionModel* should be returned in response. So, a request URL for the Face - Detect REST API will look like this:

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

If you are using the client library, you can assign the value for `recognitionModel` by passing a string representing the version. If you leave it unassigned, the default model version (*recognition_01*) will be used. See the following code example for the .NET client library.

```
string imageUrl = "https://news.microsoft.com/ceo/assets/photos/06_web.jpg";
var faces = await faceClient.Face.DetectWithUrlAsync(imageUrl, true, true, recognitionModel: "recognition_02",
returnRecognitionModel: true);
```

Identify faces with specified model

The Face API can extract face data from an image and associate it with a **Person** object (through the Add face API call, for example), and multiple **Person** objects can be stored together in a **PersonGroup**. Then, a new face can be compared against a **PersonGroup** (with the Face - Identify call), and the matching person within that group can be identified.

A **PersonGroup** should have one unique recognition model for all of the **Person**s, and you can specify this using the `recognitionModel` parameter when you create the group (PersonGroup - Create or LargePersonGroup - Create). If you do not specify this parameter, the original `recognition_01` model is used. A group will always use the recognition model it was created with, and new faces will become associated with this model when they are added to it; this cannot be changed after a group's creation. To see what model a **PersonGroup** is configured with, use the PersonGroup - Get API with the *returnRecognitionModel* parameter set as **true**.

See the following code example for the .NET client library.

```
// Create an empty PersonGroup with "recognition_02" model
string personGroupId = "mypersongroupid";
await faceClient.PersonGroup.CreateAsync(personGroupId, "My Person Group Name", recognitionModel:
"recognition_02");
```

In this code, a **PersonGroup** with ID `mypersongroupid` is created, and it is set up to use the *recognition_02* model to extract face features.

Correspondingly, you need to specify which model to use when detecting faces to compare against this **PersonGroup** (through the Face - Detect API). The model you use should always be consistent with the **PersonGroup**'s configuration; otherwise, the operation will fail due to incompatible models.

There is no change in the Face - Identify API; you only need to specify the model version in detection.

Find similar faces with specified model

You can also specify a recognition model for similarity search. You can assign the model version with `recognitionModel` when creating the face list with FaceList - Create API or LargeFaceList - Create. If you do not specify this parameter, the original `recognition_01` model is used. A face list will always use the recognition model it was created with, and new faces will become associated with this model when they are added to it; this cannot be changed after creation. To see what model a face list is configured with, use the FaceList - Get API with the *returnRecognitionModel* parameter set as **true**.

See the following code example for the .NET client library.

```
await faceClient.FaceList.CreateAsync(faceListId, "My face collection", recognitionModel: "recognition_02");
```

This code creates a face list called `My face collection`, using the *recognition_02* model for feature extraction. When you search this face list for similar faces to a new detected face, that face must have been detected (Face - Detect) using the *recognition_02* model. As in the previous section, the model needs to be consistent.

There is no change in the Face - Find Similar API; you only specify the model version in detection.

Verify faces with specified model

The Face - Verify API checks whether two faces belong to the same person. There is no change in the Verify API with regard to recognition models, but you can only compare faces that were detected with the same model. So, the two faces will both need to have been detected using `recognition_01` or `recognition_02`.

Evaluate different models

If you'd like to compare the performances of the *recognition_01* and *recognition_02* models on your data, you will need to:

1. Create two **PersonGroup**s with *recognition_01* and *recognition_02* respectively.
2. Use your image data to detect faces and register them to **Person**s for these two **PersonGroup**s, and trigger the training process with PersonGroup - Train API.
3. Test with Face - Identify on both **PersonGroup**s and compare the results.

If you normally specify a confidence threshold (a value between zero and one that determines how confident the model must be to identify a face), you may need to use different thresholds for different models. A threshold for one model is not meant to be shared to another and will not necessarily produce the same results.

Next steps

In this article, you learned how to specify the recognition model to use with different Face service APIs. Next, follow a quickstart to get started using face detection.

- Face .NET SDK
- Face Python SDK

Specify a face detection model

9/4/2019 • 4 minutes to read • Edit Online

This guide shows you how to specify a face detection model for the Azure Face API.

The Face API uses machine learning models to perform operations on human faces in images. We continue to improve the accuracy of our models based on customer feedback and advances in research, and we deliver these improvements as model updates. Developers have the option to specify which version of the face detection model they'd like to use; they can choose the model that best fits their use case.

Read on to learn how to specify the face detection model in certain face operations. The Face API uses face detection whenever it converts an image of a face into some other form of data.

If you aren't sure whether you should use the latest model, skip to the Evaluate different models section to evaluate the new model and compare results using your current data set.

Prerequisites

You should be familiar with the concept of AI face detection. If you aren't, see the face detection conceptual guide or how-to guide:

- Face detection concepts
- How to detect faces in an image

Detect faces with specified model

Face detection finds the bounding-box locations of human faces and identifies their visual landmarks. It extracts the face's features and stores them for later use in recognition operations.

When you use the Face - Detect API, you can assign the model version with the `detectionModel` parameter. The available values are:

- `detection_01`
- `detection_02`

A request URL for the Face - Detect REST API will look like this:

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

If you are using the client library, you can assign the value for `detectionModel` by passing in an appropriate string. If you leave it unassigned, the API will use the default model version (`detection_01`). See the following code example for the .NET client library.

```
string imageUrl = "https://news.microsoft.com/ceo/assets/photos/06_web.jpg";
var faces = await faceClient.Face.DetectWithUrlAsync(imageUrl, false, false, recognitionModel:
"recognition_02", detectionModel: "detection_02");
```

Add face to Person with specified model

The Face API can extract face data from an image and associate it with a **Person** object through the PersonGroup Person - Add Face API. In this API call, you can specify the detection model in the same way as in Face - Detect.

See the following code example for the .NET client library.

```
// Create a PersonGroup and add a person with face detected by "detection_02" model
string personGroupId = "mypersongroupid";
await faceClient.PersonGroup.CreateAsync(personGroupId, "My Person Group Name", recognitionModel:
"recognition_02");

string personId = (await faceClient.PersonGroupPerson.CreateAsync(personGroupId, "My Person Name")).PersonId;

string imageUrl = "https://news.microsoft.com/ceo/assets/photos/06_web.jpg";
await client.PersonGroupPerson.AddFaceFromUrlAsync(personGroupId, personId, imageUrl, detectionModel:
"detection_02");
```

This code creates a **PersonGroup** with ID `mypersongroupid` and adds a **Person** to it. Then it adds a Face to this **Person** using the `detection_02` model. If you don't specify the *detectionModel* parameter, the API will use the default model, `detection_01`.

> **NOTE**
> You don't need to use the same detection model for all faces in a **Person** object, and you don't need to use the same detection model when detecting new faces to compare with a **Person** object (in the Face - Identify API, for example).

Add face to FaceList with specified model

You can also specify a detection model when you add a face to an existing **FaceList** object. See the following code example for the .NET client library.

```
await faceClient.FaceList.CreateAsync(faceListId, "My face collection", recognitionModel: "recognition_02");

string imageUrl = "https://news.microsoft.com/ceo/assets/photos/06_web.jpg";
await client.FaceList.AddFaceFromUrlAsync(faceListId, imageUrl, detectionModel: "detection_02");
```

This code creates a **FaceList** called `My face collection` and adds a Face to it with the `detection_02` model. If you don't specify the *detectionModel* parameter, the API will use the default model, `detection_01`.

> **NOTE**
> You don't need to use the same detection model for all faces in a **FaceList** object, and you don't need to use the same detection model when detecting new faces to compare with a **FaceList** object.

Evaluate different models

The different face detection models are optimized for different tasks. See the following table for an overview of the differences.

DETECTION_01	DETECTION_02
Default choice for all face detection operations.	Released in May 2019 and available optionally in all face detection operations.
Not optimized for small, side-view, or blurry faces.	Improved accuracy on small, side-view, and blurry faces.
Returns face attributes (head pose, age, emotion, and so on) if they're specified in the detect call.	Does not return face attributes.

DETECTION_01	DETECTION_02
Returns face landmarks if they're specified in the detect call.	Does not return face landmarks.

The best way to compare the performances of the `detection_01` and `detection_02` models is to use them on a sample dataset. We recommend calling the Face - Detect API on a variety of images, especially images of many faces or of faces that are difficult to see, using each detection model. Pay attention to the number of faces that each model returns.

Next steps

In this article, you learned how to specify the detection model to use with different Face APIs. Next, follow a quickstart to get started using face detection.

- Face .NET SDK
- Face Python SDK

Example: Use the large-scale feature

8/28/2019 • 8 minutes to read • Edit Online

This guide is an advanced article on how to scale up from existing PersonGroup and FaceList objects to LargePersonGroup and LargeFaceList objects, respectively. This guide demonstrates the migration process. It assumes a basic familiarity with PersonGroup and FaceList objects, the Train operation, and the face recognition functions. To learn more about these subjects, see the Face recognition conceptual guide.

LargePersonGroup and LargeFaceList are collectively referred to as large-scale operations. LargePersonGroup can contain up to 1 million persons, each with a maximum of 248 faces. LargeFaceList can contain up to 1 million faces. The large-scale operations are similar to the conventional PersonGroup and FaceList but have some differences because of the new architecture.

The samples are written in C# by using the Azure Cognitive Services Face API client library.

> **NOTE**
> To enable Face search performance for Identification and FindSimilar in large scale, introduce a Train operation to preprocess the LargeFaceList and LargePersonGroup. The training time varies from seconds to about half an hour based on the actual capacity. During the training period, it's possible to perform Identification and FindSimilar if a successful training operating was done before. The drawback is that the new added persons and faces don't appear in the result until a new post migration to large-scale training is completed.

Step 1: Initialize the client object

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

```
string SubscriptionKey = "<Subscription Key>";
// Use your own subscription endpoint corresponding to the subscription key.
string SubscriptionEndpoint = "https://westus.api.cognitive.microsoft.com";
private readonly IFaceClient faceClient = new FaceClient(
            new ApiKeyServiceClientCredentials(subscriptionKey),
            new System.Net.Http.DelegatingHandler[] { });
faceClient.Endpoint = SubscriptionEndpoint
```

To get the subscription key with its corresponding endpoint, go to the Azure Marketplace from the Azure portal. For more information, see Subscriptions.

Step 2: Code migration

This section focuses on how to migrate PersonGroup or FaceList implementation to LargePersonGroup or LargeFaceList. Although LargePersonGroup or LargeFaceList differs from PersonGroup or FaceList in design and internal implementation, the API interfaces are similar for backward compatibility.

Data migration isn't supported. You re-create the LargePersonGroup or LargeFaceList instead.

Migrate a PersonGroup to a LargePersonGroup

Migration from a PersonGroup to a LargePersonGroup is simple. They share exactly the same group-level operations.

For PersonGroup- or person-related implementation, it's necessary to change only the API paths or SDK

class/module to LargePersonGroup and LargePersonGroup Person.

Add all of the faces and persons from the PersonGroup to the new LargePersonGroup. For more information, see Add faces.

Migrate a FaceList to a 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. Training the LargeFaceList is a precondition of the FindSimilar operation. Training isn't 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 FaceClient.LargeTrainLargeFaceListAsync(largeFaceListId);

    // Wait for training finish.
    while (true)
    {
        Task.Delay(timeIntervalInMilliseconds).Wait();
        var status = await faceClient.LargeFaceList.TrainAsync(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 use of FaceList with added faces and FindSimilar looked like the following:

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

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

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

When migrating it to LargeFaceList, it becomes the following:

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

// Add Faces to the LargeFaceList.
Parallel.ForEach(
    Directory.GetFiles(ImageDir, "*.jpg"),
    async imagePath =>
        {
            using (Stream stream = File.OpenRead(imagePath))
            {
                await faceClient.LargeFaceList.AddFaceFromStreamAsync(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 = faceClient.Face.DetectWithStreamAsync(stream).Result;
    foreach (var face in faces)
    {
        results.Add(await faceClient.Face.FindSimilarAsync(face.FaceId, largeFaceListId: LargeFaceListId));
    }
}
```

As previously shown, the data management and the FindSimilar part are almost the same. The only exception is

that a fresh preprocessing Train operation must complete in the LargeFaceList before FindSimilar works.

Step 3: Train suggestions

Although the Train operation speeds up 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 FOR FACES OR PERSONS	ESTIMATED TRAINING TIME
1,000	1-2 sec
10,000	5-10 sec
100,000	1-2 min
1,000,000	10-30 min

To better utilize the large-scale feature, we recommend the following strategies.

Step 3.1: Customize time interval

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

The same strategy also applies to LargePersonGroup. For example, when you train a LargePersonGroup with 1 million persons, `timeIntervalInMilliseconds` might be 60,000, which is a 1-minute interval.

Step 3.2: Small-scale buffer

Persons or faces in a LargePersonGroup or a LargeFaceList are searchable only after being trained. In a dynamic scenario, new persons or faces are constantly added and must be immediately searchable, yet training might take longer than desired.

To mitigate this problem, use an extra small-scale LargePersonGroup or LargeFaceList as a buffer only for the newly added entries. This buffer takes a shorter time to train because of the smaller size. The immediate search capability on this temporary buffer should work. Use this buffer in combination with training on the master LargePersonGroup or LargeFaceList by running the master training on a sparser interval. Examples are in the middle of the night and daily.

An example workflow:

1. Create a master LargePersonGroup or LargeFaceList, which is the master collection. Create a buffer LargePersonGroup or LargeFaceList, which is the buffer collection. The buffer collection is only for newly added persons or faces.
2. Add new persons or faces to both the master collection and the buffer collection.
3. Only train the buffer collection with a short time interval to ensure that the newly added entries take effect.
4. Call Identification or FindSimilar against both the master collection and the buffer collection. Merge the results.
5. When the buffer collection size increases to a threshold or at a system idle time, create a new buffer collection. Trigger the Train operation on the master collection.
6. Delete the old buffer collection after the Train operation finishes on the master collection.

Step 3.3: Standalone training

If a relatively long latency is acceptable, it isn't necessary to trigger the Train operation right after you add 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. It can be applied to static scenarios to further reduce the Train

frequency.

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

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

    // Set up 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();
}
```

For more information about data management and identification-related implementations, see Add faces and Identify faces in an image.

Summary

In this guide, you learned how to migrate the existing PersonGroup or FaceList code, not data, to the LargePersonGroup or LargeFaceList:

- LargePersonGroup and LargeFaceList work similar to PersonGroup or FaceList, except that the Train operation is required by LargeFaceList.
- Take the proper Train strategy to dynamic data update for large-scale data sets.

Next steps

Follow a how-to guide to learn how to add faces to a PersonGroup or execute the Identify operation on a PersonGroup.

- Add faces
- Identify faces in an image

Example: How to Analyze Videos in Real-time

8/28/2019 • 6 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 each iteration grabs a frame, analyzes it, and then consumes 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 analysis happens in the cloud, the latency involved means that an API call might take several seconds. During this 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 code launches each analysis in a separate Task, which can run in the background while we continue grabbing new frames. With this method we avoid blocking the main thread while waiting for an API call to return, but 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. Therefore, the final step is to add a "consumer" thread that will track the analysis tasks, raise exceptions, kill long-running tasks, and ensure that the results get consumed in the correct order.

A Producer-Consumer Design

In our final "producer-consumer" system, we have a producer thread that looks 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 takes tasks off the queue, waits for them to finish, and either displays the

result or raises 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, you will use a flexible implementation of the system described above. To access the code, go to https://github.com/Microsoft/Cognitive-Samples-VideoFrameAnalysis.

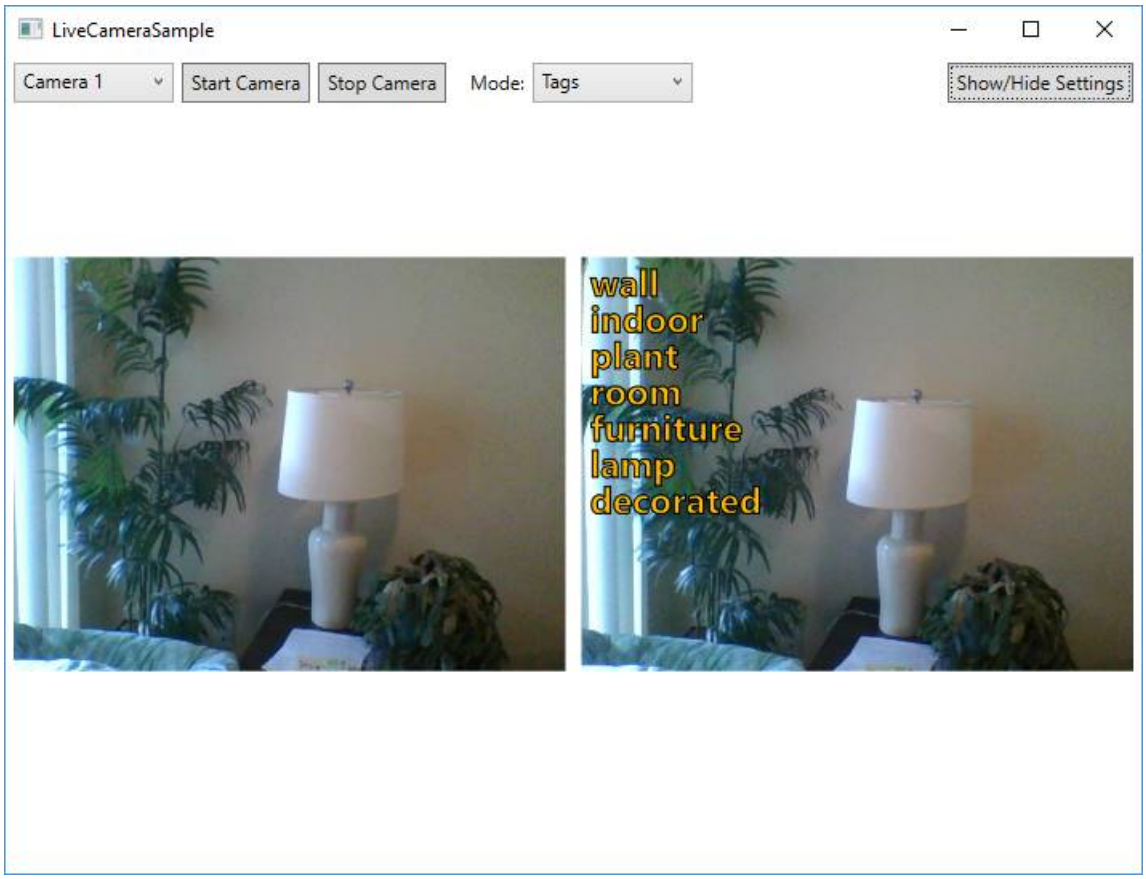
The library contains the class FrameGrabber, which implements the producer-consumer system discussed above to process video frames from a webcam. The user can specify the exact form of the API call, and the class uses events to let the calling code know when a new frame is acquired or a new analysis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

To illustrate some of the possibilities, there are two sample apps that use the library. The first is a simple console app, and a simplified version of it 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.
   private readonly IFaceClient faceClient = new FaceClient(
            new ApiKeyServiceClientCredentials("<subscription key>"),
            new System.Net.Http.DelegatingHandler[] { });

   // 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. One exception is the "EmotionsWithClientFaceDetect" mode, which performs face detection locally on the client computer using OpenCV, before submitting any images to Cognitive Services. This way, we can visualize the detected face immediately and then update the emotions once the API call returns. This is an example of a "hybrid" approach, where the client can perform some simple processing, and Cognitive Services APIs can 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, and 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, **reference the VideoFrameAnalyzer library from your own projects.**

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 to use our sample code to get started. You can start building your app with free API keys at the Azure Cognitive Services sign-up page.

Feel free to provide feedback and suggestions in the GitHub repository or, for broader API feedback, on our UserVoice site.

Related Topics

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

Migrate your face data to a different Face subscription

9/10/2019 • 6 minutes to read • Edit Online

This guide shows you how to move face data, such as a saved PersonGroup object with faces, to a different Azure Cognitive Services Face API subscription. To move the data, you use the Snapshot feature. This way you avoid having to repeatedly build and train a PersonGroup or FaceList object when you move or expand your operations. For example, perhaps you created a PersonGroup object by using a free trial subscription and now want to migrate it to your paid subscription. Or you might need to sync face data across subscriptions in different regions for a large enterprise operation.

This same migration strategy also applies to LargePersonGroup and LargeFaceList objects. If you aren't familiar with the concepts in this guide, see their definitions in the Face recognition concepts guide. This guide uses the Face API .NET client library with C#.

Prerequisites

You need the following items:

- Two Face API subscription keys, one with the existing data and one to migrate to. To subscribe to the Face API service and get your key, follow the instructions in Create a Cognitive Services account.
- The Face API subscription ID string that corresponds to the target subscription. To find it, select **Overview** in the Azure portal.
- Any edition of Visual Studio 2015 or 2017.

Create the Visual Studio project

This guide uses a simple console app to run the face data migration. For a full implementation, see the Face API snapshot sample on GitHub.

1. In Visual Studio, create a new Console app .NET Framework project. Name it **FaceApiSnapshotSample**.
2. Get the required NuGet packages. Right-click your project in the Solution Explorer, and select **Manage NuGet Packages**. Select the **Browse** tab, and select **Include prerelease**. Find and install the following package:
   - Microsoft.Azure.CognitiveServices.Vision.Face 2.3.0-preview

Create face clients

In the **Main** method in *Program.cs*, create two FaceClient instances for your source and target subscriptions. This example uses a Face subscription in the East Asia region as the source and a West US subscription as the target. This example demonstrates how to migrate data from one Azure region to another.

> **NOTE**
>
> New resources created after July 1, 2019, will use custom subdomain names. For more information and a complete list of regional endpoints, see Custom subdomain names for Cognitive Services.

```
var FaceClientEastAsia = new FaceClient(new ApiKeyServiceClientCredentials("<East Asia Subscription Key>"))
    {
        Endpoint = "https://southeastasia.api.cognitive.microsoft.com/>"
    };

var FaceClientWestUS = new FaceClient(new ApiKeyServiceClientCredentials("<West US Subscription Key>"))
    {
        Endpoint = "https://westus.api.cognitive.microsoft.com/"
    };
```

Fill in the subscription key values and endpoint URLs for your source and target subscriptions.

Prepare a PersonGroup for migration

You need the ID of the PersonGroup in your source subscription to migrate it to the target subscription. Use the PersonGroupOperationsExtensions.ListAsync method to retrieve a list of your PersonGroup objects. Then get the PersonGroup.PersonGroupId property. This process looks different based on what PersonGroup objects you have. In this guide, the source PersonGroup ID is stored in `personGroupId`.

> **NOTE**
> The sample code creates and trains a new PersonGroup to migrate. In most cases, you should already have a PersonGroup to use.

Take a snapshot of a PersonGroup

A snapshot is temporary remote storage for certain Face data types. It functions as a kind of clipboard to copy data from one subscription to another. First, you take a snapshot of the data in the source subscription. Then you apply it to a new data object in the target subscription.

Use the source subscription's FaceClient instance to take a snapshot of the PersonGroup. Use TakeAsync with the PersonGroup ID and the target subscription's ID. If you have multiple target subscriptions, add them as array entries in the third parameter.

```
var takeSnapshotResult = await FaceClientEastAsia.Snapshot.TakeAsync(
    SnapshotObjectType.PersonGroup,
    personGroupId,
    new[] { "<Azure West US Subscription ID>" /* Put other IDs here, if multiple target subscriptions wanted */
});
```

> **NOTE**
> The process of taking and applying snapshots doesn't disrupt any regular calls to the source or target PersonGroups or FaceLists. Don't make simultaneous calls that change the source object, such as FaceList management calls or the PersonGroup Train call, for example. The snapshot operation might run before or after those operations or might encounter errors.

Retrieve the snapshot ID

The method used to take snapshots is asynchronous, so you must wait for its completion. Snapshot operations can't be canceled. In this code, the `WaitForOperation` method monitors the asynchronous call. It checks the status every 100 ms. After the operation finishes, retrieve an operation ID by parsing the `OperationLocation` field.

```
var takeOperationId = Guid.Parse(takeSnapshotResult.OperationLocation.Split('/')[2]);
var operationStatus = await WaitForOperation(FaceClientEastAsia, takeOperationId);
```

A typical `OperationLocation` value looks like this:

```
"/operations/a63a3bdd-a1db-4d05-87b8-dbad6850062a"
```

The `WaitForOperation` helper method is here:

```
/// <summary>
/// Waits for the take/apply operation to complete and returns the final operation status.
/// </summary>
/// <returns>The final operation status.</returns>
private static async Task<OperationStatus> WaitForOperation(IFaceClient client, Guid operationId)
{
    OperationStatus operationStatus = null;
    do
    {
        if (operationStatus != null)
        {
            Thread.Sleep(TimeSpan.FromMilliseconds(100));
        }

        // Get the status of the operation.
        operationStatus = await client.Snapshot.GetOperationStatusAsync(operationId);

        Console.WriteLine($"Operation Status: {operationStatus.Status}");
    }
    while (operationStatus.Status != OperationStatusType.Succeeded
            && operationStatus.Status != OperationStatusType.Failed);

    return operationStatus;
}
```

After the operation status shows `Succeeded`, get the snapshot ID by parsing the `ResourceLocation` field of the returned OperationStatus instance.

```
var snapshotId = Guid.Parse(operationStatus.ResourceLocation.Split('/')[2]);
```

A typical `resourceLocation` value looks like this:

```
"/snapshots/e58b3f08-1e8b-4165-81df-aa9858f233dc"
```

Apply a snapshot to a target subscription

Next, create the new PersonGroup in the target subscription by using a randomly generated ID. Then use the target subscription's FaceClient instance to apply the snapshot to this PersonGroup. Pass in the snapshot ID and the new PersonGroup ID.

```
var newPersonGroupId = Guid.NewGuid().ToString();
var applySnapshotResult = await FaceClientWestUS.Snapshot.ApplyAsync(snapshotId, newPersonGroupId);
```

> **NOTE**
> A Snapshot object is valid for only 48 hours. Only take a snapshot if you intend to use it for data migration soon after.

A snapshot apply request returns another operation ID. To get this ID, parse the `OperationLocation` field of the returned applySnapshotResult instance.

```
var applyOperationId = Guid.Parse(applySnapshotResult.OperationLocation.Split('/')[2]);
```

The snapshot application process is also asynchronous, so again use `WaitForOperation` to wait for it to finish.

```
operationStatus = await WaitForOperation(FaceClientWestUS, applyOperationId);
```

Test the data migration

After you apply the snapshot, the new PersonGroup in the target subscription populates with the original face data. By default, training results are also copied. The new PersonGroup is ready for face identification calls without needing retraining.

To test the data migration, run the following operations and compare the results they print to the console:

```
await DisplayPersonGroup(FaceClientEastAsia, personGroupId);
await IdentifyInPersonGroup(FaceClientEastAsia, personGroupId);

await DisplayPersonGroup(FaceClientWestUS, newPersonGroupId);
// No need to retrain the person group before identification,
// training results are copied by snapshot as well.
await IdentifyInPersonGroup(FaceClientWestUS, newPersonGroupId);
```

Use the following helper methods:

```
private static async Task DisplayPersonGroup(IFaceClient client, string personGroupId)
{
    var personGroup = await client.PersonGroup.GetAsync(personGroupId);
    Console.WriteLine("Person Group:");
    Console.WriteLine(JsonConvert.SerializeObject(personGroup));

    // List persons.
    var persons = await client.PersonGroupPerson.ListAsync(personGroupId);

    foreach (var person in persons)
    {
        Console.WriteLine(JsonConvert.SerializeObject(person));
    }

    Console.WriteLine();
}
```

```
private static async Task IdentifyInPersonGroup(IFaceClient client, string personGroupId)
{
    using (var fileStream = new FileStream("data\\PersonGroup\\Daughter\\Daughter1.jpg", FileMode.Open,
FileAccess.Read))
    {
        var detectedFaces = await client.Face.DetectWithStreamAsync(fileStream);

        var result = await client.Face.IdentifyAsync(detectedFaces.Select(face => face.FaceId.Value).ToList(),
personGroupId);
        Console.WriteLine("Test identify against PersonGroup");
        Console.WriteLine(JsonConvert.SerializeObject(result));
        Console.WriteLine();
    }
}
```

Now you can use the new PersonGroup in the target subscription.

To update the target PersonGroup again in the future, create a new PersonGroup to receive the snapshot. To do this, follow the steps in this guide. A single PersonGroup object can have a snapshot applied to it only one time.

Clean up resources

After you finish migrating face data, manually delete the snapshot object.

```
await FaceClientEastAsia.Snapshot.DeleteAsync(snapshotId);
```

Next steps

Next, see the relevant API reference documentation, explore a sample app that uses the Snapshot feature, or follow a how-to guide to start using the other API operations mentioned here:

- Snapshot reference documentation (.NET SDK)
- Face API snapshot sample
- Add faces
- Detect faces in an image
- Identify faces in an image

Install and run Face containers

9/26/2019 • 10 minutes to read • Edit Online

Azure Cognitive Services Face provides a standardized Linux container for Docker that detects human faces in images. It also identifies attributes, which include face landmarks such as noses and eyes, gender, age, and other machine-predicted facial features. In addition to detection, Face can check if two faces in the same image or different images are the same by using a confidence score. Face also can compare faces against a database to see if a similar-looking or identical face already exists. It also can organize similar faces into groups by using shared visual traits.

If you don't have an Azure subscription, create a free account before you begin.

Prerequisites

You must meet the following prerequisites before you use the Face API containers.

REQUIRED	PURPOSE
Docker Engine	The Docker Engine must be installed on a host computer. Docker provides packages that configure the Docker environment on macOS, Windows, and Linux. For a primer on Docker and container basics, see the Docker overview. Docker must be configured to allow the containers to connect with and send billing data to Azure. On Windows, Docker also must be configured to support Linux containers.
Familiarity with Docker	You need a basic understanding of Docker concepts, such as registries, repositories, containers, and container images. You also need knowledge of basic `docker` commands.
Face resource	To use the container, you must have: An Azure **Face** resource and the associated API key and the endpoint URI. Both values are available on the **Overview** and **Keys** pages for the resource. They're required to start the container. **{API_KEY}**: One of the two available resource keys on the **Keys** page **{ENDPOINT_URI}**: The endpoint as provided on the **Overview** page

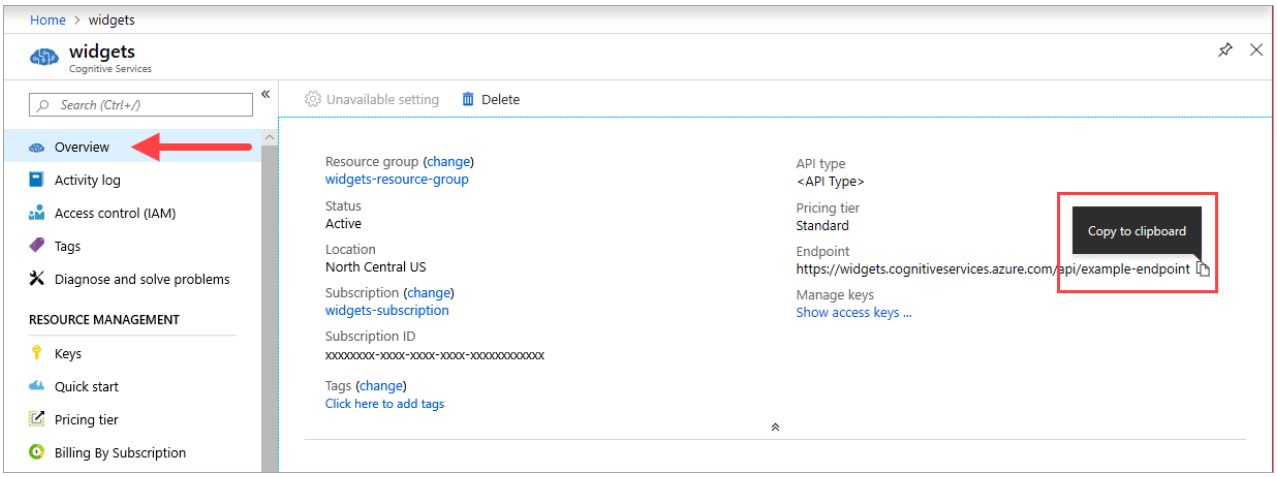
Gathering required parameters

There are three primary parameters for all Cognitive Services' containers that are required. The end-user license agreement (EULA) must be present with a value of `accept`. Additionally, both an Endpoint URL and API Key are needed.

> **NOTE**
>
> The only exception to these three required parameters is when containers are considered "Offline" containers. Offline containers do not report usage, are not metered and follow a different billing methodology.

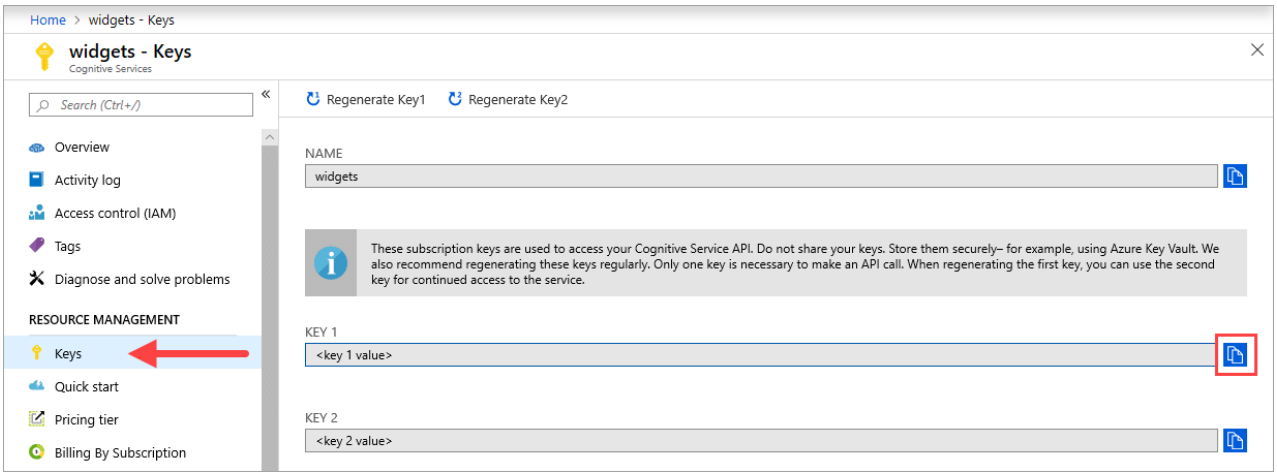
Endpoint URI `{ENDPOINT_URI}`

The **Endpoint** URI value is available on the Azure portal *Overview* page of the corresponding Cognitive Service resource. Navigate to the *Overview* page, hover over the Endpoint, and a `Copy to clipboard` icon will appear. Copy and use where needed.



Keys `{API_KEY}`

This key is used to start the container, and is available on the Azure portal's Keys page of the corresponding Cognitive Service resource. Navigate to the *Keys* page, and click on the `Copy to clipboard` icon.



> **IMPORTANT**
>
> These subscription keys are used to access your Cognitive Service API. Do not share your keys. Store them securely, for example, using Azure Key Vault. We also recommend regenerating these keys regularly. Only one key is necessary to make an API call. When regenerating the first key, you can use the second key for continued access to the service.

Request access to the private container registry

Fill out and submit the Cognitive Services Vision Containers Request form to request access to the container. The form requests information about you, your company, and the user scenario for which you'll use the container. After you submit the form, the Azure Cognitive Services team reviews it to make sure that you meet the criteria for access to the private container registry.

> **IMPORTANT**
> You must use an email address associated with either a Microsoft Account (MSA) or an Azure Active Directory (Azure AD) account in the form.

If your request is approved, you receive an email with instructions that describe how to obtain your credentials and access the private container registry.

Log in to the private container registry

There are several ways to authenticate with the private container registry for Cognitive Services containers. We recommend that you use the command-line method by using the Docker CLI.

Use the docker login command, as shown in the following example, to log in to `containerpreview.azurecr.io`, which is the private container registry for Cognitive Services containers. Replace <*username*> with the user name and <*password*> with the password provided in the credentials you received from the Azure Cognitive Services team.

```
docker login containerpreview.azurecr.io -u <username> -p <password>
```

If you secured your credentials in a text file, you can concatenate the contents of that text file to the `docker login` command. Use the `cat` command, as shown in the following example. Replace <*passwordFile*> with the path and name of the text file that contains the password. Replace <*username*> with the user name provided in your credentials.

```
cat <passwordFile> | docker login containerpreview.azurecr.io -u <username> --password-stdin
```

The host computer

The host is a x64-based computer that runs the Docker container. It can be a computer on your premises or a Docker hosting service in Azure, such as:

- Azure Kubernetes Service.
- Azure Container Instances.
- A Kubernetes cluster deployed to Azure Stack. For more information, see Deploy Kubernetes to Azure Stack.

Container requirements and recommendations

The following table describes the minimum and recommended CPU cores and memory to allocate for each Face API container.

CONTAINER	MINIMUM	RECOMMENDED	TRANSACTIONS PER SECOND (MINIMUM, MAXIMUM)
Face	1 core, 2-GB memory	1 core, 4-GB memory	10, 20

- Each core must be at least 2.6 GHz or faster.
- Transactions per second (TPS).

Core and memory correspond to the `--cpus` and `--memory` settings, which are used as part of the `docker run` command.

Get the container image with docker pull

Container images for the Face API are available.

CONTAINER	REPOSITORY
Face	`containerpreview.azurecr.io/microsoft/cognitive-services-face:latest`

TIP

You can use the docker images command to list your downloaded container images. For example, the following command lists the ID, repository, and tag of each downloaded container image, formatted as a table:

```
docker images --format "table {{.ID}}\t{{.Repository}}\t{{.Tag}}"

IMAGE ID         REPOSITORY                TAG
<image-id>       <repository-path/name>    <tag-name>
```

Docker pull for the Face container

```
docker pull containerpreview.azurecr.io/microsoft/cognitive-services-face:latest
```

Use the container

After the container is on the host computer, use the following process to work with the container.

1. Run the container with the required billing settings. More examples of the `docker run` command are available.
2. Query the container's prediction endpoint.

Run the container with docker run

Use the docker run command to run the container. Refer to gathering required parameters for details on how to get the `{ENDPOINT_URI}` and `{API_KEY}` values.

Examples of the `docker run` command are available.

```
docker run --rm -it -p 5000:5000 --memory 4g --cpus 1 \
containerpreview.azurecr.io/microsoft/cognitive-services-face \
Eula=accept \
Billing={ENDPOINT_URI} \
ApiKey={API_KEY}
```

This command:

- Runs a face container from the container image.
- Allocates one CPU core and 4 GB of memory.
- Exposes TCP port 5000 and allocates a pseudo TTY for the container.
- Automatically removes the container after it exits. The container image is still available on the host computer.

More examples of the `docker run` command are available.

> **IMPORTANT**
> The `Eula`, `Billing`, and `ApiKey` options must be specified to run the container or the container won't start. For more information, see Billing.

Run multiple containers on the same host

If you intend to run multiple containers with exposed ports, make sure to run each container with a different exposed port. For example, run the first container on port 5000 and the second container on port 5001.

You can have this container and a different Azure Cognitive Services container running on the HOST together. You also can have multiple containers of the same Cognitive Services container running.

Query the container's prediction endpoint

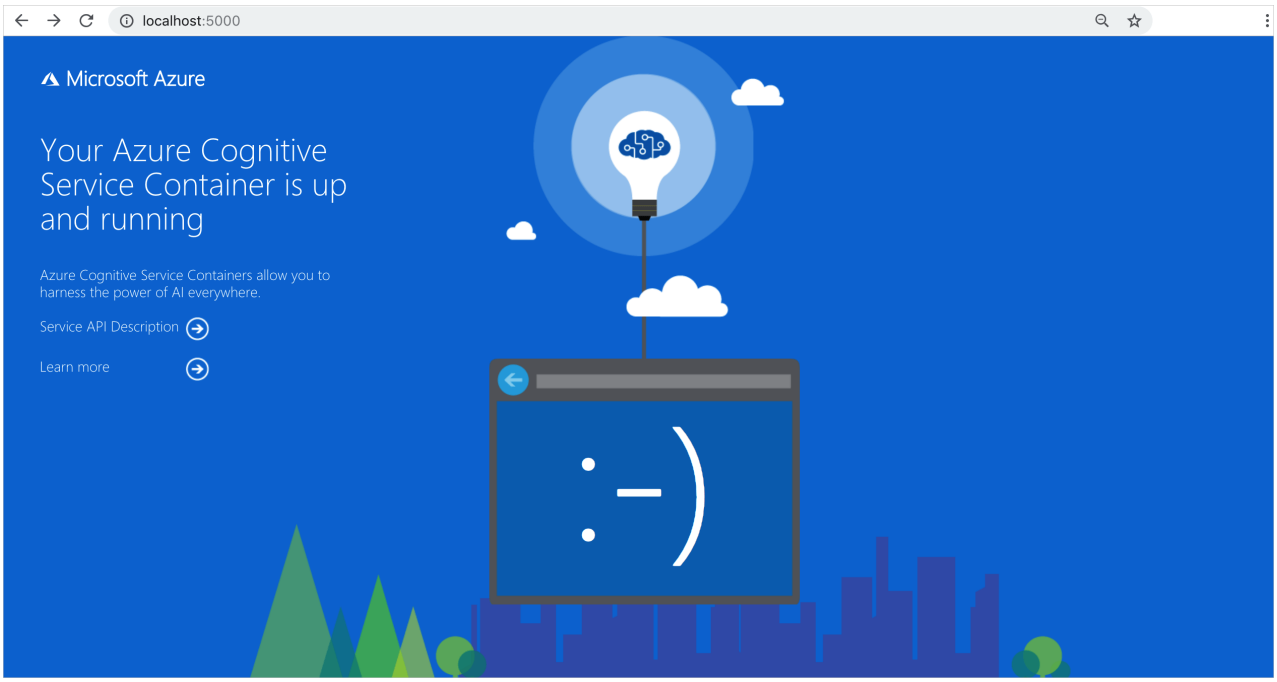
The container provides REST-based query prediction endpoint APIs.

Use the host, `http://localhost:5000`, for container APIs.

Validate that a container is running

There are several ways to validate that the container is running. Locate the *External IP* address and exposed port of the container in question, and open your favorite web browser. Use the various request URLs below to validate the container is running. The example request URLs listed below are `http://localhost:5000`, but your specific container may vary. Keep in mind that you're to rely on your container's *External IP* address and exposed port.

REQUEST URL	PURPOSE
`http://localhost:5000/`	The container provides a home page.
`http://localhost:5000/status`	Requested with an HTTP GET, to validate that the container is running without causing an endpoint query. This request can be used for Kubernetes liveness and readiness probes.
`http://localhost:5000/swagger`	The container provides a full set of documentation for the endpoints and a **Try it out** feature. With this feature, you can enter your settings into a web-based HTML form and make the query without having to write any code. After the query returns, an example CURL command is provided to demonstrate the HTTP headers and body format that's required.



Stop the container

To shut down the container, in the command-line environment where the container is running, select `Ctrl+C`.

Troubleshooting

If you run the container with an output mount and logging is enabled, the container generates log files that are helpful to troubleshoot issues that happen while you start or run the container.

> **TIP**
> For more troubleshooting information and guidance, see Cognitive Services containers frequently asked questions (FAQ).

Billing

The Face API containers send billing information to Azure by using a Face API resource on your Azure account.

Queries to the container are billed at the pricing tier of the Azure resource that's used for the `<ApiKey>`.

Azure Cognitive Services containers aren't licensed to run without being connected to the billing endpoint for metering. You must enable the containers to communicate billing information with the billing endpoint at all times. Cognitive Services containers don't send customer data, such as the image or text that's being analyzed, to Microsoft.

Connect to Azure

The container needs the billing argument values to run. These values allow the container to connect to the billing endpoint. The container reports usage about every 10 to 15 minutes. If the container doesn't connect to Azure within the allowed time window, the container continues to run but doesn't serve queries until the billing endpoint is restored. The connection is attempted 10 times at the same time interval of 10 to 15 minutes. If it can't connect to the billing endpoint within the 10 tries, the container stops running.

Billing arguments

For the `docker run` command to start the container, all three of the following options must be specified with valid values:

OPTION	DESCRIPTION
`ApiKey`	The API key of the Cognitive Services resource that's used to track billing information. The value of this option must be set to an API key for the provisioned resource that's specified in `Billing`.
`Billing`	The endpoint of the Cognitive Services resource that's used to track billing information. The value of this option must be set to the endpoint URI of a provisioned Azure resource.
`Eula`	Indicates that you accepted the license for the container. The value of this option must be set to **accept**.

For more information about these options, see Configure containers.

Blog posts

- Running Cognitive Services Containers
- Getting started with Cognitive Services Language Understanding container

Developer samples

Developer samples are available at our GitHub repository.

View webinar

Join the webinar to learn about:

- How to deploy Cognitive Services to any machine using Docker
- How to deploy Cognitive Services to AKS

Summary

In this article, you learned concepts and workflow for how to download, install, and run Face API containers. In summary:

- Container images are downloaded from the Azure Container Registry.
- Container images run in Docker.
- You can use either the REST API or the SDK to call operations in Face API containers by specifying the host URI of the container.
- You must specify billing information when you instantiate a container.

> **IMPORTANT**
> Cognitive Services containers aren't licensed to run without being connected to Azure for metering. Customers must enable the containers to communicate billing information with the metering service at all times. Cognitive Services containers don't send customer data, such as the image or text that's being analyzed, to Microsoft.

Next steps

- For configuration settings, see Configure containers.
- To learn more about how to detect and identify faces, see Face overview.

- For information about the methods supported by the container, see the Face API.
- To use more Cognitive Services containers, see Cognitive Services containers.

Configure Face Docker containers

9/18/2019 • 10 minutes to read • Edit Online

The **Face** container runtime environment is configured using the `docker run` command arguments. This container has several required settings, along with a few optional settings. Several examples of the command are available. The container-specific settings are the billing settings.

Configuration settings

The container has the following configuration settings:

REQUIRED	SETTING	PURPOSE
Yes	ApiKey	Tracks billing information.
No	ApplicationInsights	Enables adding Azure Application Insights telemetry support to your container.
Yes	Billing	Specifies the endpoint URI of the service resource on Azure.
Yes	Eula	Indicates that you've accepted the license for the container.
No	Fluentd	Writes log and, optionally, metric data to a Fluentd server.
No	Http Proxy	Configures an HTTP proxy for making outbound requests.
No	Logging	Provides ASP.NET Core logging support for your container.
No	Mounts	Reads and writes data from the host computer to the container and from the container back to the host computer.

IMPORTANT

The `ApiKey`, `Billing`, and `Eula` settings are used together, and you must provide valid values for all three of them; otherwise your container won't start. For more information about using these configuration settings to instantiate a container, see Billing.

ApiKey configuration setting

The `ApiKey` setting specifies the Azure resource key used to track billing information for the container. You must specify a value for the ApiKey and the value must be a valid key for the *Cognitive Services* resource specified for the `Billing` configuration setting.

This setting can be found in the following place:

- Azure portal: **Cognitive Services** Resource Management, under **Keys**

ApplicationInsights setting

The `ApplicationInsights` setting allows you to add Azure Application Insights telemetry support to your container. Application Insights provides in-depth monitoring of your container. You can easily monitor your container for availability, performance, and usage. You can also quickly identify and diagnose errors in your container.

The following table describes the configuration settings supported under the `ApplicationInsights` section.

REQUIRED	NAME	DATA TYPE	DESCRIPTION

REQUIRED	NAME	DATA TYPE	DESCRIPTION
No	`InstrumentationKey`	String	The instrumentation key of the Application Insights instance to which telemetry data for the container is sent. For more information, see Application Insights for ASP.NET Core. Example: `InstrumentationKey=123456789`

Billing configuration setting

The `Billing` setting specifies the endpoint URI of the *Cognitive Services* resource on Azure used to meter billing information for the container. You must specify a value for this configuration setting, and the value must be a valid endpoint URI for a *Cognitive Services* resource on Azure. The container reports usage about every 10 to 15 minutes.

This setting can be found in the following place:

- Azure portal: **Cognitive Services** Overview, labeled `Endpoint`

Remember to add the *Face* routing to the endpoint URI as shown in the example.

REQUIRED	NAME	DATA TYPE	DESCRIPTION
Yes	`Billing`	String	Billing endpoint URI Example: `Billing=https://westcentralus.api.cognitiv`

CloudAI configuration settings

The configuration settings in the `CloudAI` section provide container-specific options unique to your container. The following settings and objects are supported for the Face container in the `CloudAI` section

NAME	DATA TYPE	DESCRIPTION
`Storage`	Object	The storage scenario used by the Face container. For more information about storage scenarios and associated settings for the `Storage` object, see Storage scenario settings

Storage scenario settings

The Face container stores blob, cache, metadata, and queue data, depending on what's being stored. For example, training indexes and results for a large person group are stored as blob data. The Face container provides two different storage scenarios when interacting with and storing these types of data:

- Memory
  All four types of data are stored in memory. They're not distributed, nor are they persistent. If the Face container is stopped or removed, all of the data in storage for that container is destroyed.
  This is the default storage scenario for the Face container.
- Azure
  The Face container uses Azure Storage and Azure Cosmos DB to distribute these four types of data across persistent storage. Blob and queue data is handled by Azure Storage. Metadata and cache data is handled by Azure Cosmos DB. If the Face container is stopped or removed, all of the data in storage for that container remains stored in Azure Storage and Azure Cosmos DB.
  The resources used by the Azure storage scenario have the following additional requirements
  - The Azure Storage resource must use the StorageV2 account kind
  - The Azure Cosmos DB resource must use the Azure Cosmos DB's API for MongoDB

The storage scenarios and associated configuration settings are managed by the `Storage` object, under the `CloudAI` configuration section. The following configuration settings are available in the `Storage` object:

NAME	DATA TYPE	DESCRIPTION

NAME	DATA TYPE	DESCRIPTION
`StorageScenario`	String	The storage scenario supported by the container. The following values are available `Memory` - Default value. Container uses non-persistent, non-distributed and in-memory storage, for single-node, temporary usage. If the container is stopped or removed, the storage for that container is destroyed. `Azure` - Container uses Azure resources for storage. If the container is stopped or removed, the storage for that container is persisted.
`ConnectionStringOfAzureStorage`	String	The connection string for the Azure Storage resource used by the container. This setting applies only if `Azure` is specified for the `StorageScenario` configuration setting.
`ConnectionStringOfCosmosMongo`	String	The MongoDB connection string for the Azure Cosmos DB resource used by the container. This setting applies only if `Azure` is specified for the `StorageScenario` configuration setting.

For example, the following command specifies the Azure storage scenario and provides sample connection strings for the Azure Storage and Cosmos DB resources used to store data for the Face container.

```
docker run --rm -it -p 5000:5000 --memory 4g --cpus 1 containerpreview.azurecr.io/microsoft/cognitive-services-face Eula=accept
Billing=https://westcentralus.api.cognitive.microsoft.com/face/v1.0 ApiKey=0123456789 CloudAI:Storage:StorageScenario=Azure
CloudAI:Storage:ConnectionStringOfCosmosMongo="mongodb://samplecosmosdb:0123456789@samplecosmosdb.documents.azure.com:10255/?
ssl=true&replicaSet=globaldb"
CloudAI:Storage:ConnectionStringOfAzureStorage="DefaultEndpointsProtocol=https;AccountName=sampleazurestorage;AccountKey=0123456789;EndpointSuf
fix=core.windows.net"
```

The storage scenario is handled separately from input mounts and output mounts. You can specify a combination of those features for a single container. For example, the following command defines a Docker bind mount to the `D:\Output` folder on the host machine as the output mount, then instantiates a container from the Face container image, saving log files in JSON format to the output mount. The command also specifies the Azure storage scenario and provides sample connection strings for the Azure Storage and Cosmos DB resources used to store data for the Face container.

```
docker run --rm -it -p 5000:5000 --memory 4g --cpus 1 --mount type=bind,source=D:\Output,destination=/output
containerpreview.azurecr.io/microsoft/cognitive-services-face Eula=accept Billing=https://westcentralus.api.cognitive.microsoft.com/face/v1.0
ApiKey=0123456789 Logging:Disk:Format=json CloudAI:Storage:StorageScenario=Azure
CloudAI:Storage:ConnectionStringOfCosmosMongo="mongodb://samplecosmosdb:0123456789@samplecosmosdb.documents.azure.com:10255/?
ssl=true&replicaSet=globaldb"
CloudAI:Storage:ConnectionStringOfAzureStorage="DefaultEndpointsProtocol=https;AccountName=sampleazurestorage;AccountKey=0123456789;EndpointSuf
fix=core.windows.net"
```

Eula setting

The `Eula` setting indicates that you've accepted the license for the container. You must specify a value for this configuration setting, and the value must be set to `accept`.

REQUIRED	NAME	DATA TYPE	DESCRIPTION
Yes	`Eula`	String	License acceptance Example: `Eula=accept`

Cognitive Services containers are licensed under your agreement governing your use of Azure. If you do not have an existing agreement governing your use of Azure, you agree that your agreement governing use of Azure is the Microsoft Online Subscription Agreement, which incorporates the Online Services Terms. For previews, you also agree to the Supplemental Terms of Use for Microsoft Azure Previews. By using the container you agree to these terms.

Fluentd settings

Fluentd is an open-source data collector for unified logging. The `Fluentd` settings manage the container's connection to a Fluentd server. The container includes a Fluentd logging provider, which allows your container to write logs and, optionally, metric data to a Fluentd server.

The following table describes the configuration settings supported under the `Fluentd` section.

NAME	DATA TYPE	DESCRIPTION
`Host`	String	The IP address or DNS host name of the Fluentd server.
`Port`	Integer	The port of the Fluentd server. The default value is 24224.
`HeartbeatMs`	Integer	The heartbeat interval, in milliseconds. If no event traffic has been sent before this interval expires, a heartbeat is sent to the Fluentd server. The default value is 60000 milliseconds (1 minute).
`SendBufferSize`	Integer	The network buffer space, in bytes, allocated for send operations. The default value is 32768 bytes (32 kilobytes).
`TlsConnectionEstablishmentTimeoutMs`	Integer	The timeout, in milliseconds, to establish a SSL/TLS connection with the Fluentd server. The default value is 10000 milliseconds (10 seconds). If `UseTLS` is set to false, this value is ignored.
`UseTLS`	Boolean	Indicates whether the container should use SSL/TLS for communicating with the Fluentd server. The default value is false.

Http proxy credentials settings

If you need to configure an HTTP proxy for making outbound requests, use these two arguments:

NAME	DATA TYPE	DESCRIPTION
HTTP_PROXY	string	The proxy to use, for example, `http://proxy:8888` `<proxy-url>`
HTTP_PROXY_CREDS	string	Any credentials needed to authenticate against the proxy, for example, username:password.
`<proxy-user>`	string	The user for the proxy.
`<proxy-password>`	string	The password associated with `<proxy-user>` for the proxy.

```
docker run --rm -it -p 5000:5000 \
--memory 2g --cpus 1 \
--mount type=bind,src=/home/azureuser/output,target=/output \
<registry-location>/<image-name> \
Eula=accept \
Billing=<endpoint> \
ApiKey=<api-key> \
HTTP_PROXY=<proxy-url> \
HTTP_PROXY_CREDS=<proxy-user>:<proxy-password> \
```

Logging settings

The `Logging` settings manage ASP.NET Core logging support for your container. You can use the same configuration settings and values for your container that you use for an ASP.NET Core application.

The following logging providers are supported by the container:

PROVIDER	PURPOSE
Console	The ASP.NET Core `Console` logging provider. All of the ASP.NET Core configuration settings and default values for this logging provider are supported.
Debug	The ASP.NET Core `Debug` logging provider. All of the ASP.NET Core configuration settings and default values for this logging provider are supported.

PROVIDER	PURPOSE
Disk	The JSON logging provider. This logging provider writes log data to the output mount.

This container command stores logging information in the JSON format to the output mount:

```
docker run --rm -it -p 5000:5000 \
--memory 2g --cpus 1 \
--mount type=bind,src=/home/azureuser/output,target=/output \
<registry-location>/<image-name> \
Eula=accept \
Billing=<endpoint> \
ApiKey=<api-key> \
Logging:Disk:Format=json
```

This container command shows debugging information, prefixed with `dbug`, while the container is running:

```
docker run --rm -it -p 5000:5000 \
--memory 2g --cpus 1 \
<registry-location>/<image-name> \
Eula=accept \
Billing=<endpoint> \
ApiKey=<api-key> \
Logging:Console:LogLevel:Default=Debug
```

Disk logging

The `Disk` logging provider supports the following configuration settings:

NAME	DATA TYPE	DESCRIPTION
`Format`	String	The output format for log files. **Note:** This value must be set to `json` to enable the logging provider. If this value is specified without also specifying an output mount while instantiating a container, an error occurs.
`MaxFileSize`	Integer	The maximum size, in megabytes (MB), of a log file. When the size of the current log file meets or exceeds this value, a new log file is started by the logging provider. If -1 is specified, the size of the log file is limited only by the maximum file size, if any, for the output mount. The default value is 1.

For more information about configuring ASP.NET Core logging support, see Settings file configuration.

Mount settings

Use bind mounts to read and write data to and from the container. You can specify an input mount or output mount by specifying the `--mount` option in the docker run command.

The Face containers don't use input or output mounts to store training or service data.

The exact syntax of the host mount location varies depending on the host operating system. Additionally, the host computer's mount location may not be accessible due to a conflict between permissions used by the Docker service account and the host mount location permissions.

OPTIONAL	NAME	DATA TYPE	DESCRIPTION
Not allowed	`Input`	String	Face containers do not use this.
Optional	`Output`	String	The target of the output mount. The default value is `/output`. This is the location of the logs. This includes container logs. Example: `--mount type=bind,src=c:\output,target=/output`

Example docker run commands

The following examples use the configuration settings to illustrate how to write and use `docker run` commands. Once running, the container continues to run until you stop it.

- **Line-continuation character**: The Docker commands in the following sections use the back slash, `\`, as a line continuation character. Replace or remove this based on your host operating system's requirements.
- **Argument order**: Do not change the order of the arguments unless you are very familiar with Docker containers.

Replace {*argument_name*} with your own values:

PLACEHOLDER	VALUE	FORMAT OR EXAMPLE
{API_KEY}	The endpoint key of the `Face` resource on the Azure `Face` Keys page.	`xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx`
{ENDPOINT_URI}	The billing endpoint value is available on the Azure `Face` Overview page.	See gathering required parameters for explicit examples.

NOTE

New resources created after July 1, 2019, will use custom subdomain names. For more information and a complete list of regional endpoints, see Custom subdomain names for Cognitive Services.

IMPORTANT

The `Eula`, `Billing`, and `ApiKey` options must be specified to run the container; otherwise, the container won't start. For more information, see Billing. The ApiKey value is the **Key** from the Azure `Cognitive Services` Resource keys page.

Face container Docker examples

The following Docker examples are for the face container.

Basic example

```
docker run --rm -it -p 5000:5000 --memory 4g --cpus 1 \
containerpreview.azurecr.io/microsoft/cognitive-services-face \
Eula=accept \
Billing={ENDPOINT_URI} \
ApiKey={API_KEY}
```

Logging example

```
docker run --rm -it -p 5000:5000 --memory 4g --cpus 1 containerpreview.azurecr.io/microsoft/cognitive-services-face \
Eula=accept \
Billing={ENDPOINT_URI} ApiKey={API_KEY} \
Logging:Console:LogLevel:Default=Information
```

Next steps

- Review How to install and run containers

Deploy the Face container to Azure Container Instances

10/16/2019 • 5 minutes to read • Edit Online

Learn how to deploy the Cognitive Services Face container to Azure Container Instances. This procedure demonstrates the creation of an Azure Face resource. Then we discuss pulling the associated container image. Finally, we highlight the ability to exercise the orchestration of the two from a browser. Using containers can shift the developers' attention away from managing infrastructure to instead focusing on application development.

Prerequisites

- Use an Azure subscription. If you don't have an Azure subscription, create a free account before you begin.
- Install the Azure CLI (az).
- Docker engine and validate that the Docker CLI works in a console window.

Request access to the private container registry

Fill out and submit the Cognitive Services Vision Containers Request form to request access to the container. The form requests information about you, your company, and the user scenario for which you'll use the container. After you submit the form, the Azure Cognitive Services team reviews it to make sure that you meet the criteria for access to the private container registry.

> **IMPORTANT**
> You must use an email address associated with either a Microsoft Account (MSA) or an Azure Active Directory (Azure AD) account in the form.

If your request is approved, you receive an email with instructions that describe how to obtain your credentials and access the private container registry.

Log in to the private container registry

There are several ways to authenticate with the private container registry for Cognitive Services containers. We recommend that you use the command-line method by using the Docker CLI.

Use the docker login command, as shown in the following example, to log in to `containerpreview.azurecr.io`, which is the private container registry for Cognitive Services containers. Replace <*username*> with the user name and <*password*> with the password provided in the credentials you received from the Azure Cognitive Services team.

```
docker login containerpreview.azurecr.io -u <username> -p <password>
```

If you secured your credentials in a text file, you can concatenate the contents of that text file to the `docker login` command. Use the `cat` command, as shown in the following example. Replace <*passwordFile*> with the path and name of the text file that contains the password. Replace <*username*> with the user name provided in your credentials.

```
cat <passwordFile> | docker login containerpreview.azurecr.io -u <username> --password-stdin
```

Create an Face resource

1. Sign into the Azure portal
2. Click Create **Face** resource

3. Enter all required settings:

SETTING	VALUE
Name	Desired name (2-64 characters)
Subscription	Select appropriate subscription
Location	Select any nearby and available location
Pricing Tier	`F0` - the minimal pricing tier
Resource Group	Select an available resource group

4. Click **Create** and wait for the resource to be created. After it is created, navigate to the resource page

5. Collect configured `endpoint` and an API key:

RESOURCE TAB IN PORTAL	SETTING	VALUE
Overview	Endpoint	Copy the endpoint. It looks similar to `https://face.cognitiveservices.azure.com/face/`
Keys	API Key	Copy 1 of the two keys. It is a 32 alphanumeric-character string with no spaces or dashes, `xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx`.

Create an Azure Container Instance resource from the Azure CLI

The YAML below defines the Azure Container Instance resource. Copy and paste the contents into a new file, named `my-aci.yaml` and replace the commented values with your own. Refer to the template format for valid YAML. Refer to the container repositories and images for the available image names and their corresponding repository.

```
apiVersion: 2018-10-01
location: # < Valid location >
name: # < Container Group name >
imageRegistryCredentials:
  - server: containerpreview.azurecr.io
    username: # < The username for the preview container registry >
    password: # < The password for the preview container registry >
properties:
  containers:
  - name: # < Container name >
    properties:
      image: # < Repository/Image name >
      environmentVariables: # These env vars are required
        - name: eula
          value: accept
        - name: billing
          value: # < Service specific Endpoint URL >
        - name: apikey
          value: # < Service specific API key >
      resources:
        requests:
          cpu: 4 # Always refer to recommended minimal resources
          memoryInGb: 8 # Always refer to recommended minimal resources
      ports:
        - port: 5000
  osType: Linux
  restartPolicy: OnFailure
  ipAddress:
    type: Public
    ports:
    - protocol: tcp
      port: 5000
tags: null
type: Microsoft.ContainerInstance/containerGroups
```

NOTE

Not all locations have the same CPU and Memory availability. Refer to the location and resources table for the listing of available resources for containers per location and OS.

We'll rely on the YAML file we created for the `az container create` command. From the Azure CLI, execute the `az container create` command replacing the `<resource-group>` with your own. Additionally, for securing values within a YAML deployment refer to secure values.

```
az container create -g <resource-group> -f my-aci.yaml
```

The output of the command is `Running...` if valid, after sometime the output changes to a JSON string representing the newly created ACI resource. The container image is more than likely not be available for a while, but the resource is now deployed.

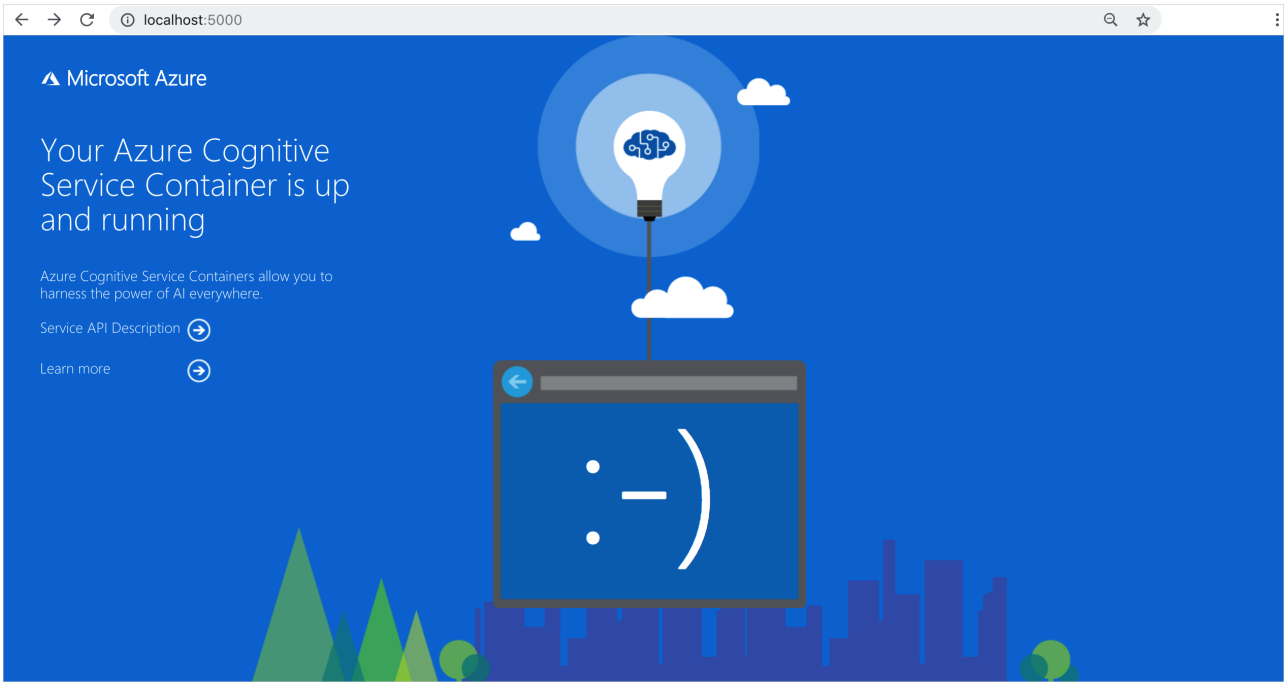
TIP

Pay close attention to the locations of public preview Azure Cognitive Service offerings, as the YAML will needed to be adjusted accordingly to match the location.

Validate that a container is running

There are several ways to validate that the container is running. Locate the *External IP* address and exposed port of the container in question, and open your favorite web browser. Use the various request URLs below to validate the container is running. The example request URLs listed below are `http://localhost:5000`, but your specific container may vary. Keep in mind that you're to rely on your container's *External IP* address and exposed port.

REQUEST URL	PURPOSE
`http://localhost:5000/`	The container provides a home page.
`http://localhost:5000/status`	Requested with an HTTP GET, to validate that the container is running without causing an endpoint query. This request can be used for Kubernetes liveness and readiness probes.
`http://localhost:5000/swagger`	The container provides a full set of documentation for the endpoints and a **Try it out** feature. With this feature, you can enter your settings into a web-based HTML form and make the query without having to write any code. After the query returns, an example CURL command is provided to demonstrate the HTTP headers and body format that's required.



Next steps

Let's continue working with Azure Cognitive Services containers.

Use more Cognitive Services Containers

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

7/5/2019 • 5 minutes to read • Edit Online

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

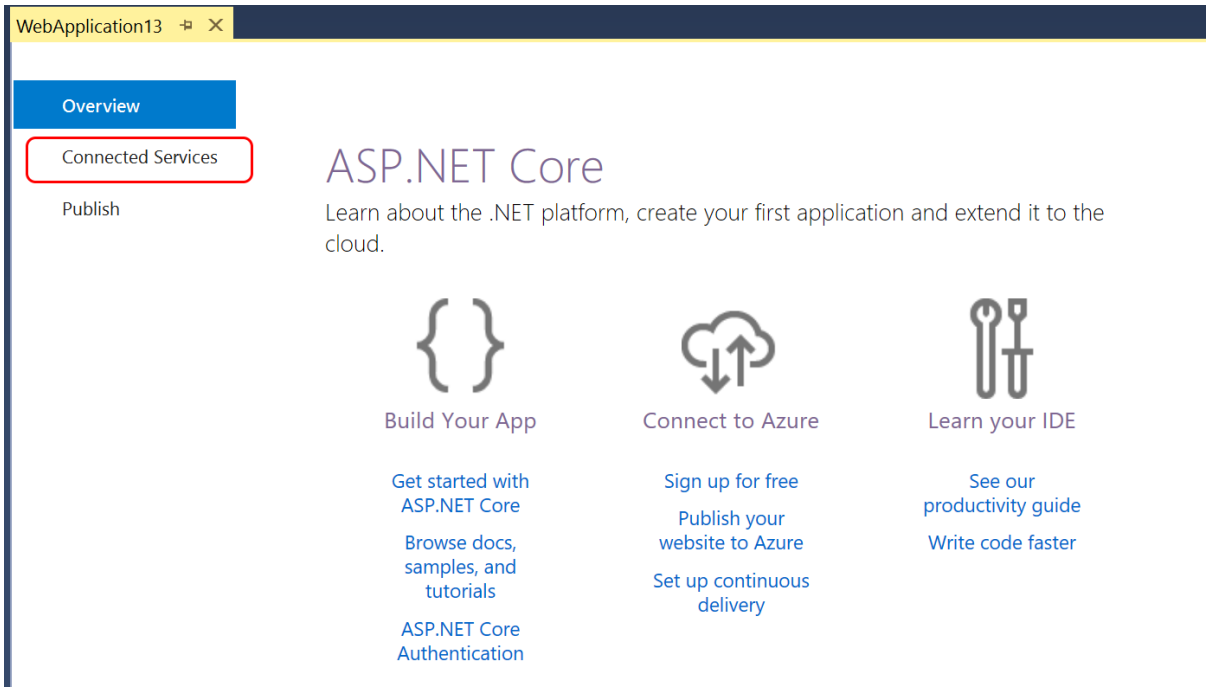
This article and its companion articles provide details for using the Visual Studio Connected Service feature for Cognitive Services Face API. The capability is available in both Visual Studio 2017 15.7 or later, with the Cognitive Services extension installed.

Prerequisites

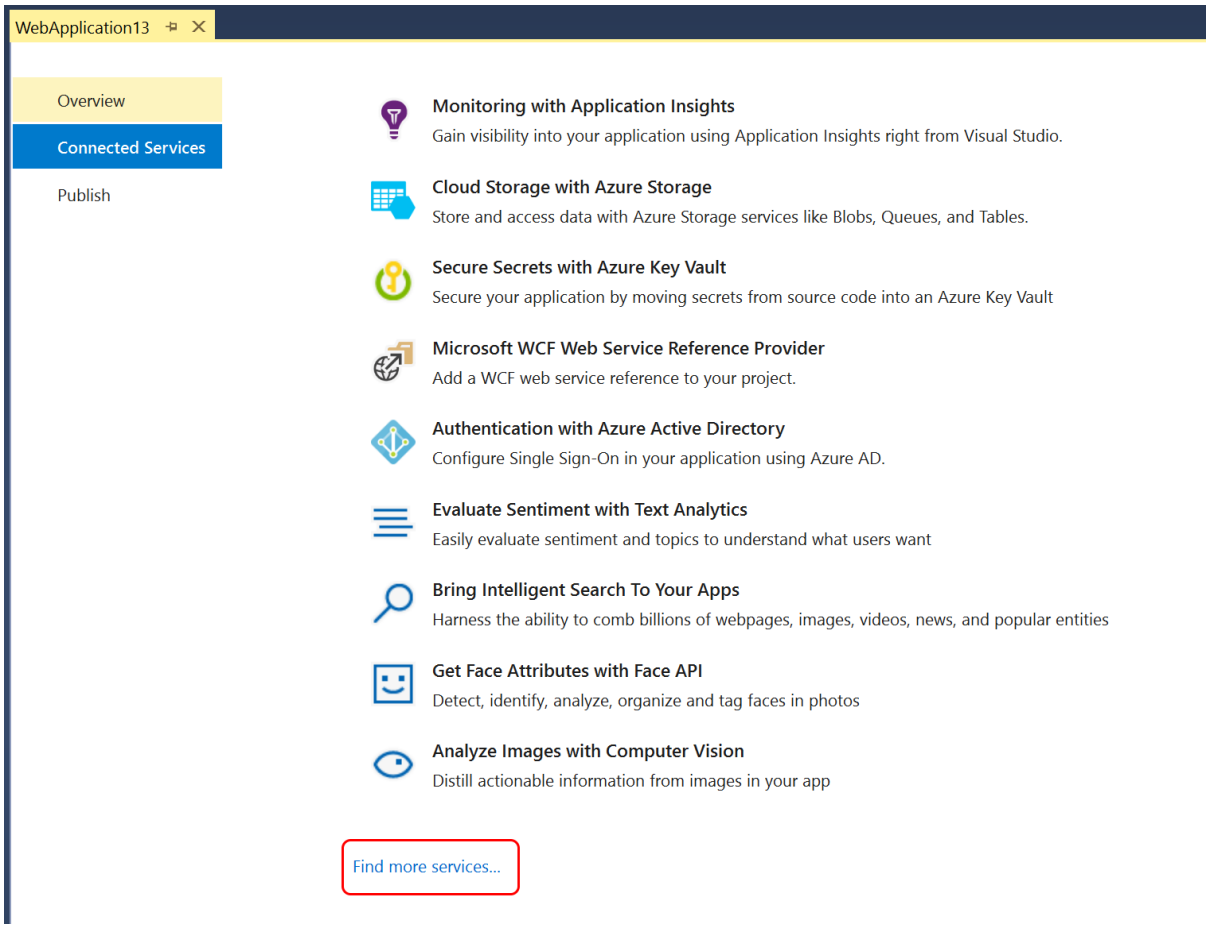
- An Azure subscription. If you do not have one, you can sign up for a free account.
- Visual Studio 2017 version 15.7 or later with the **Web Development** workload installed. Download it now.

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.

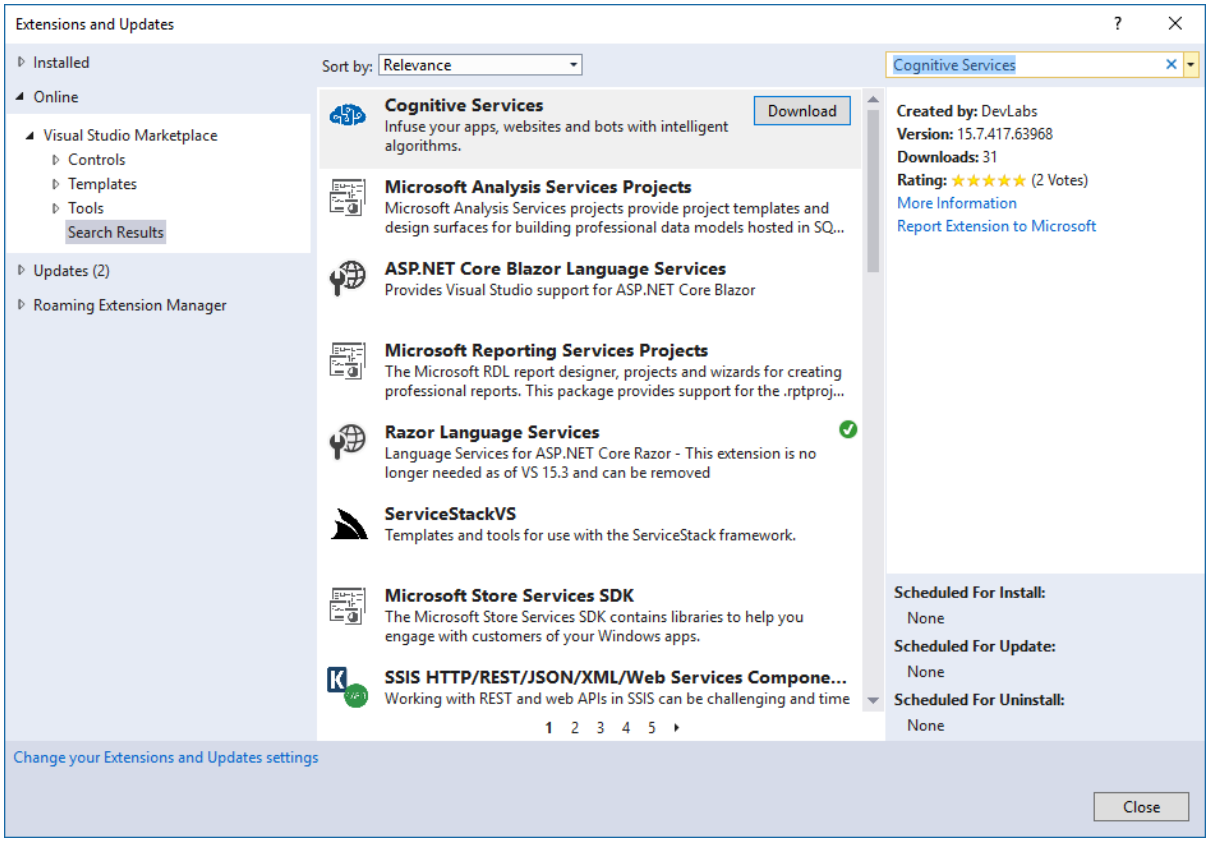


2. Scroll down to the bottom of the list of services, and select **Find more services**.



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.



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

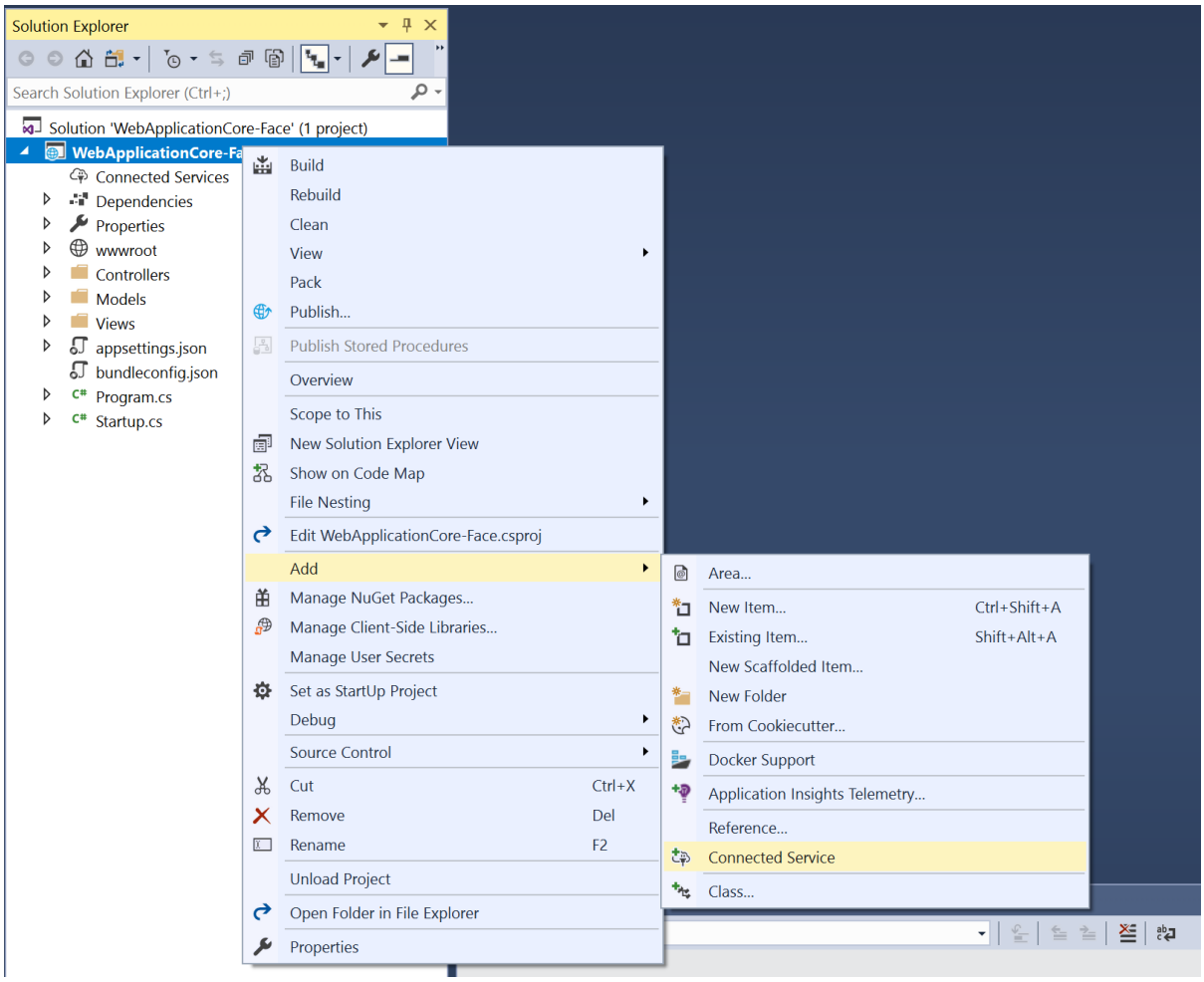
4. Restart Visual Studio. The extension installs when you close Visual Studio, and is available next time you

launch the IDE.

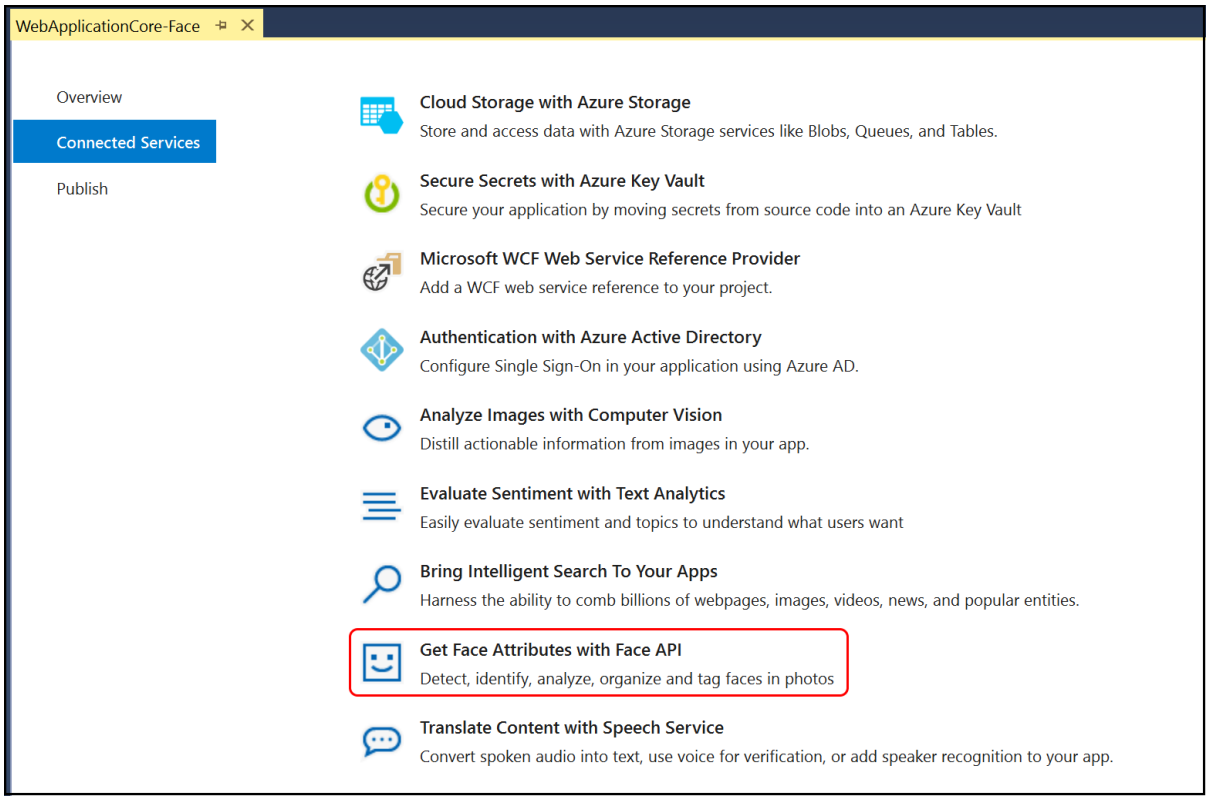
Create a project and add support for Cognitive Services Face API

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

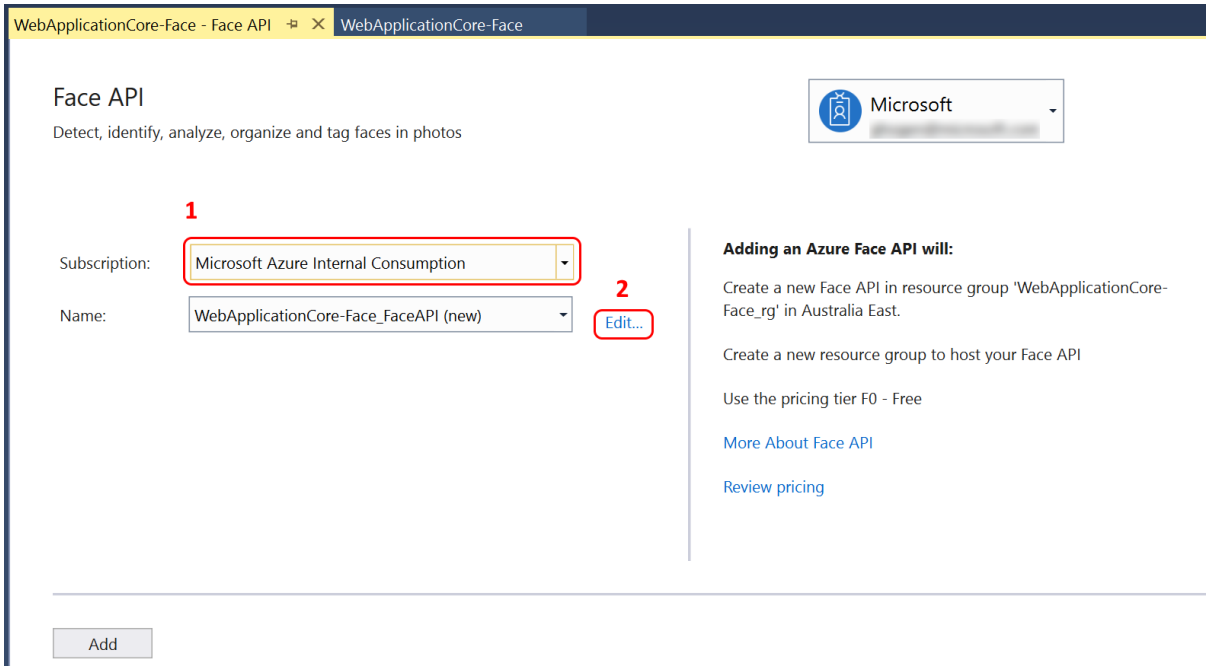
2. In **Solution Explorer**, choose **Add** > **Connected Service**. The Connected Service page appears with services you can add to your project.



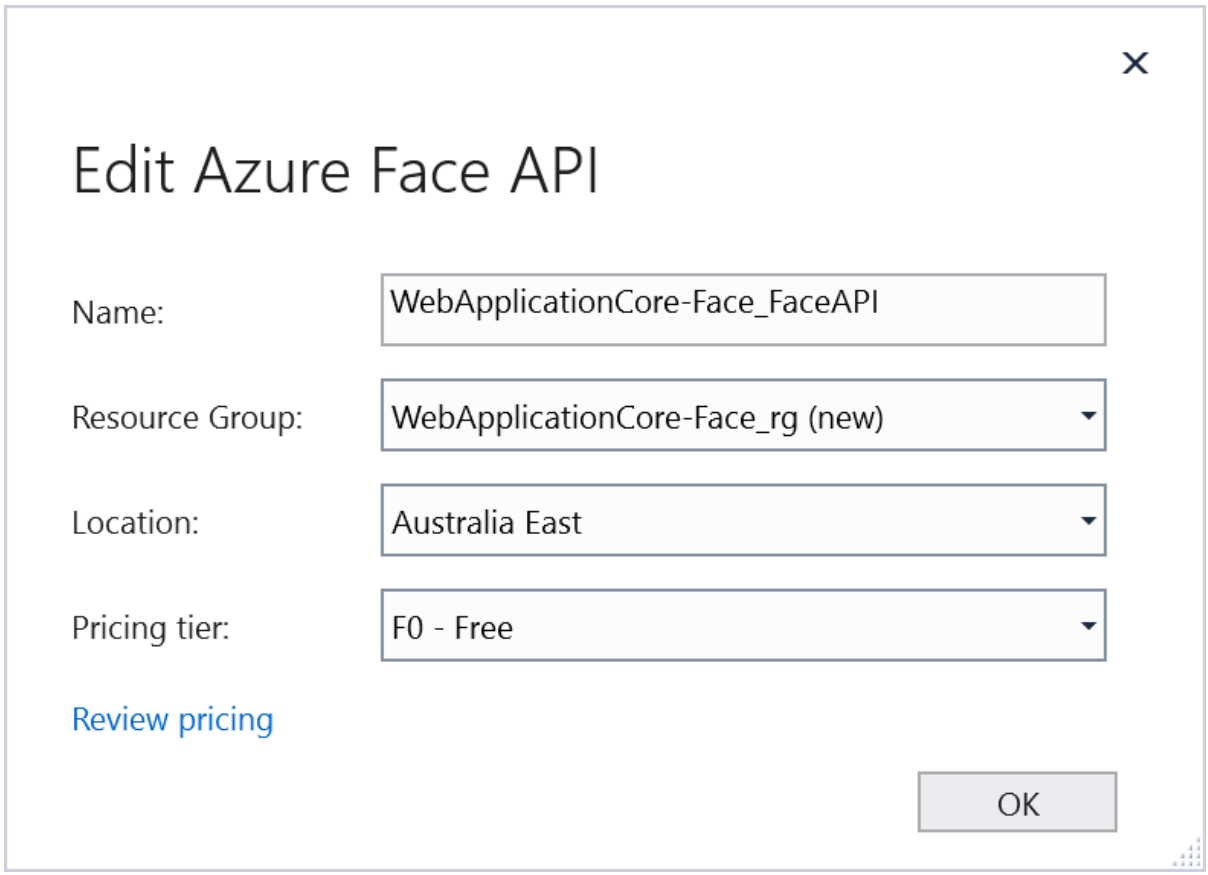
3. In the menu of available services, choose **Cognitive Services Face API**.



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.



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.



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

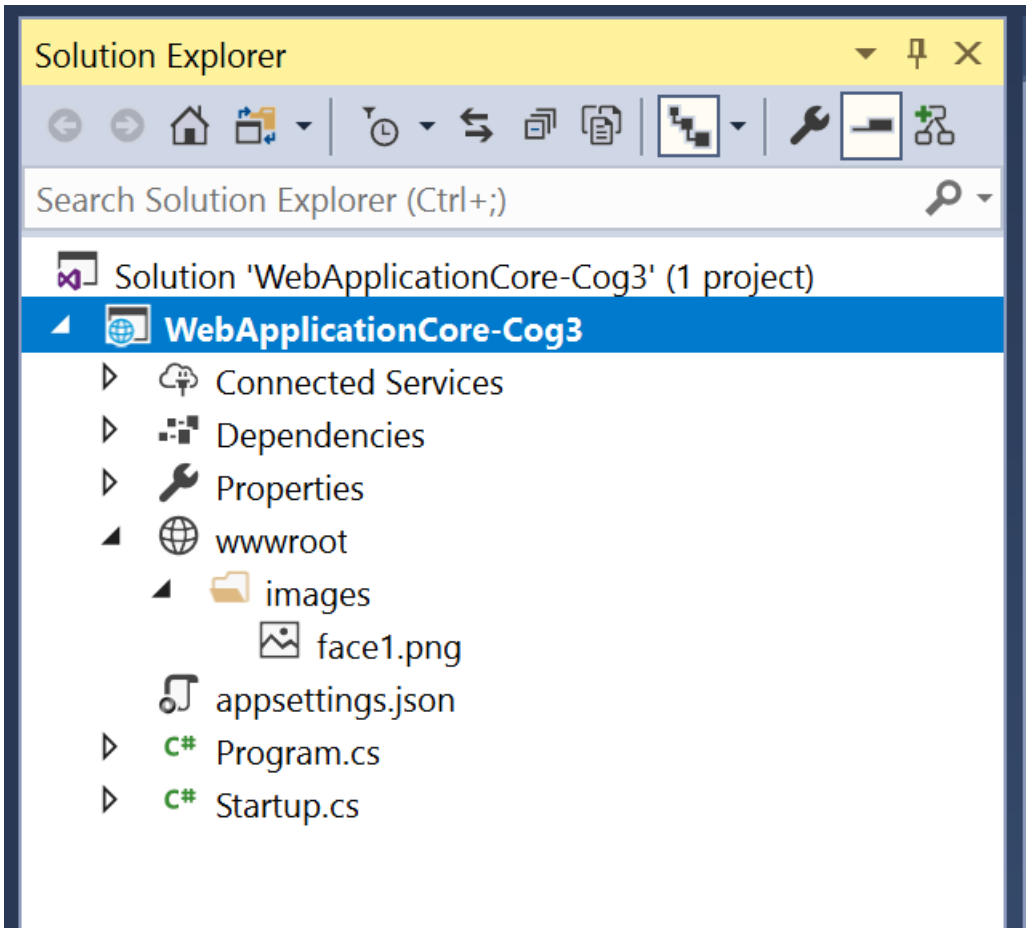
1. Add the following using statements in Startup.cs.

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

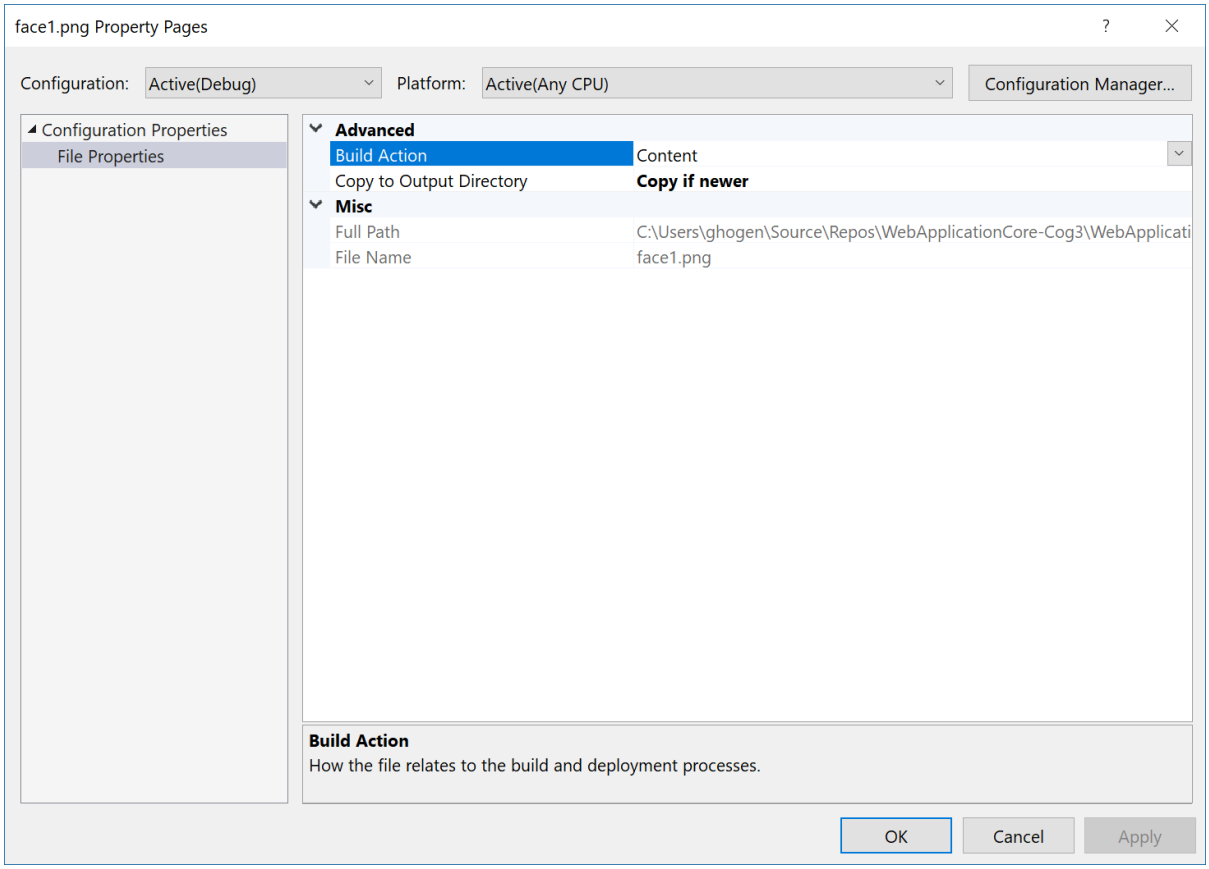
2. Add a configuration field, and add a constructor that initializes the configuration field in Startup class to enable Configuration in your program.

```
    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 choose **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**.



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 an HTTP request with a call to the Face REST API, using the key you added when you added the connected service.

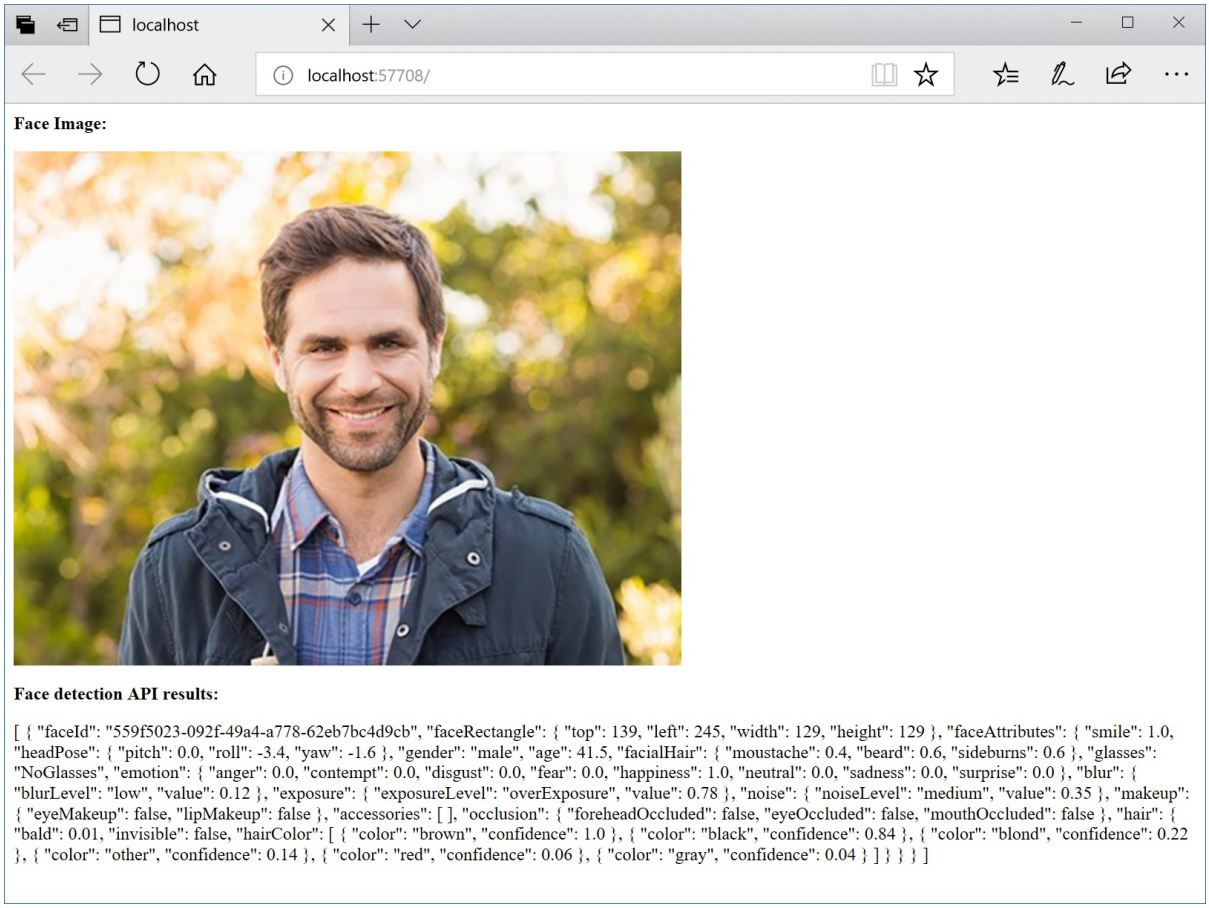
6. Add the helper functions GetImageAsByteArray and JsonPrettyPrint.

```
     /// <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.



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.

Face API Reference List

10/15/2019 • 2 minutes to read • Edit Online

The Azure Face API is a cloud-based API that provides algorithms for face detection and recognition. The Face APIs comprise the following categories:

- Face Algorithm APIs: Cover core functions such as Detection, Find Similar, Verification, Identification, and Group.
- FaceList APIs: Used to manage a FaceList for Find Similar.
- LargePersonGroup Person APIs: Used to manage LargePersonGroup Person Faces for Identification.
- LargePersonGroup APIs: Used to manage a LargePersonGroup dataset for Identification.
- LargeFaceList APIs: Used to manage a LargeFaceList for Find Similar.
- PersonGroup Person APIs: Used to manage PersonGroup Person Faces for Identification.
- PersonGroup APIs: Used to manage a PersonGroup dataset for Identification.
- Snapshot APIs: Used to manage a Snapshot for data migration across subscriptions.

Face API Release Notes

6/8/2019 • 3 minutes to read • Edit Online

This article pertains to Face API Service version 1.0.

Release changes in June 2019

- Added a new face detection model with improved accuracy on small, side-view, occluded and blurry faces. Use it through Face - Detect, FaceList - Add Face, LargeFaceList - Add Face, PersonGroup Person - Add Face and LargePersonGroup Person - Add Face by specifying the new face detection model name `detection_02` in `detectionModel` parameter. More details in How to specify a detection model.

Release changes in April 2019

- Improved overall accuracy of the `age` and `headPose` attributes. The `headPose` attribute is also updated with the `pitch` value enabled now. Use these attributes by specifying them in the `returnFaceAttributes` parameter of Face - Detect `returnFaceAttributes` parameter.
- Improved speed of Face - Detect, FaceList - Add Face, LargeFaceList - Add Face, PersonGroup Person - Add Face and LargePersonGroup Person - Add Face.

Release changes in March 2019

- Added a new face recognition model with improved accuracy. Use it through Face - Detect, FaceList - Create, LargeFaceList - Create, PersonGroup - Create and LargePersonGroup - Create by specifying the new face recognition model name `recognition_02` in `recognitionModel` parameter. More details in How to specify a recognition model.

Release changes in January 2019

- Added Snapshot feature to support data migration across subscriptions: Snapshot. More details in How to Migrate your face data to a different Face subscription.

Release changes in October 2018

- Refined description for `status`, `createdDateTime`, `lastActionDateTime`, and `lastSuccessfulTrainingDateTime` in PersonGroup - Get Training Status, LargePersonGroup - Get Training Status, and LargeFaceList - Get Training Status.

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.