Sharp AI API Documentation

Sharp AI Team

Actiontec
07*/02/2018*

TABLE OF CONTENTS

Page #

1.0 Overview

1.0 OVERVIEW

1.1 System Overview

Welcome to the Sharp AI API, an embedding-based service that provides the most advanced face algorithms. Sharp AI API has two main functions: face detection with attributes and face recognition.

1.2 Face Detection

Sharp AI API detects up to 64 human faces with high precision face location in an image. And the image can be specified by file in bytes or valid URL.

Face rectangle (left, top, width and height) indicating the face location in the image is returned along with each detected face. Optionally, face detection extracts a series of face related attributes such as pose, gender, age, head pose, facial hair and glasses.

1.3 Face Recognition

Face recognition is widely used in many scenarios including security, natural user interface, image content analysis and management, mobile apps, and robotics. Four face recognition functions are provided: face verification, finding similar faces, face grouping, and person identification.

1.3.1 Face Verification

Sharp AI API verification performs an authentication against two detected faces or authentication from one detected face to one person object.

1.3.2 Finding Similar Face

Given target detected face and a set of candidate faces to search with, our service finds a small set of faces that look most similar to the target face. Two working modes, `matchFace` and `matchPerson` are supported. `matchPerson` mode returns similar faces after applying a same-person threshold derived from Face-Verify. `matchFace` mode ignores the same-person threshold and returns top similar candidate faces. Take the following example, candidate faces are listed.

And query face is

To find 4 similar faces, `matchPerson` mode returns (a) and (b), which belong to the same person with query face. `matchFace` mode returns (a), (b), (c) and (d), exactly 4 candidates even if low similarity.

1.3.3 Face Grouping

Given one set of unknown faces, face grouping API automatically divides them into several groups based on similarity. Each group is a disjointed proper subset of the original unknown face set, and contains similar faces. And all the faces in the same group can be considered to belong to the same person object.

1.3.4 Face Identification

Face API can be used to identify people based on a detected face and people database (defined as a person group) which needs to be created in advance and can be edited over time.

The following figure is an example of a person group named "Xun Dong". Each group may contain up to 10,000 person objects. Meanwhile, each person object can have up to 248 faces registered.

After a person group has been created and trained, identification can be performed against the group and a new detected face. If the face is identified as a person object in the group, the person object will be returned.

1.3.5 Face Storage

Face Storage allows a Standard subscription to store additional persisted faces when using Person objects or Face Lists for identification or similarity matching with the Face API. The stored images are charged at $0.5 per 1000 faces and this rate is prorated on a daily basis. Free tier subscriptions are free, but limited to 1,000 total persons.

Pricing for Face Storage is prorated daily. For example, if your account used 10,000 persisted faces each day for the first half of the month and none the second half, you would be billed only for the 10,000 faces for the days stored. Alternatively, if each day during the month you persist 1,000 faces for a few hours and then delete them each night, you would still be billed for 1,000 persisted faces each day.

2.0 Sharp AI Introduction

2.0 SHARP AI INTRODUCTION

2.1 How to Detect Faces in Image

In this sample, we will demonstrate the following features:

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

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

Step 1: Authorize the API call

Every call to the Face API requires a subscription key. This key needs to be either passed through a query string parameter, or specified in the request header.

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

```
faceServiceClient = new FaceServiceClient("Your subscription key");
```

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

The most basic way to perform face detection is by uploading an image directly. This is done by sending a "POST" request with application/octet-stream content type, with the data read from a JPEG image. The maximum size of the image is 4MB.
Using the client library, face detection by means of uploading is done by passing in a Stream object. See the example below:

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

    foreach (var face in faces)
```

```
    {
        var rect = face.FaceRectangle;
        var landmarks = face.FaceLandmarks;
    }
}
```

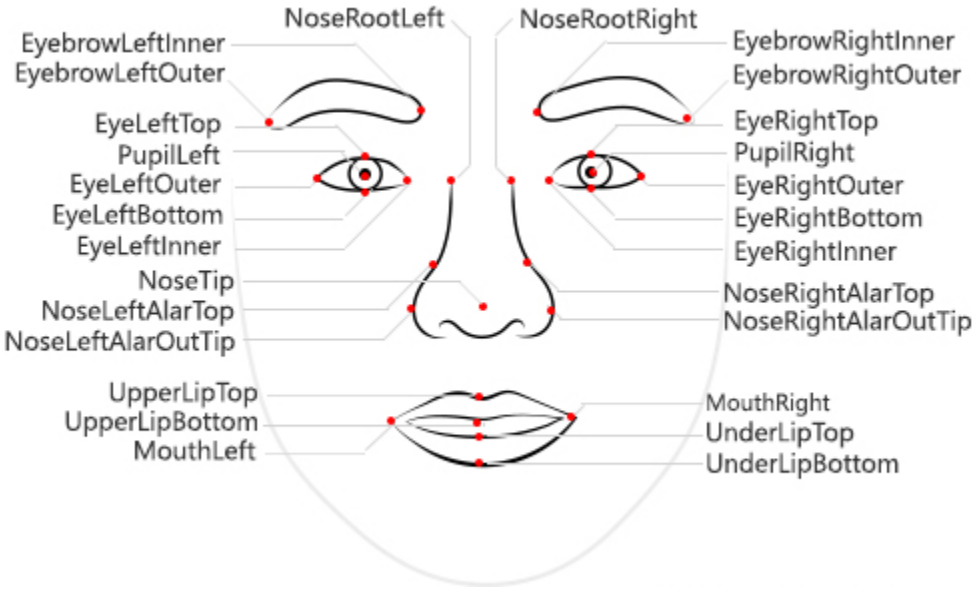
Please note that the DetectAsync method of FaceServiceClient is async. The calling method should be marked as async as well, in order to use the await clause. If the image is already on the web and has an URL, face detection can be executed by also providing the URL. In this example, the request body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

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

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

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

Step 3: Understanding and using face landmarks

Face landmarks are a series of specifically detailed points on a face; typically points of face components like the pupils, canthus or nose. Face landmarks are optional attributes that can be analyzed during face detection.
By default, there are 27 predefined landmark points. The following figure shows how all 27 points are defined:



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

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

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

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

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

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

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

```
var landmarks = face.FaceLandmarks;

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

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

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

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

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

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

Step 4: Using other face attributes

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

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

These attributes are predicted by using statistical algorithms and may not always be 100% precise. However, they are still helpful when you want to classify faces by these attributes.

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

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

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

2.2 How to Identify Faces in Image

In this sample, we will demonstrate the following:

- How to create a person group - This person group contains a list of known people.

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

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

- A few photos with the person's face. Click here to download sample photos for Anna, Bill and Clare.

- A series of test photos, which may or may not contain the faces of Anna, Bill or Clare used to test their identification. You can also select some sample images from the link above.

Step 1: Authorize the API call

Every call to the Face API requires a subscription key. This key needs to be either passed through a query string parameter, or specified in the request header.

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

```
faceServiceClient = new FaceServiceClient("Your subscription key");
```

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

Step 2: Create the person group

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

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

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

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

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

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

Detection is done by sending a "POST" web request to the Face - Detect API with the image file in the HTTP request body. When using the client library, face detection is executed through the DetectAsync method for the FaceServiceClient class.
For each face detected you can call Person – Add a Person Face to add it to the correct person.
The following code below demonstrates the process of how to detect a face from an image and add it to a person:

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

foreach (string imagePath in Directory.GetFiles(friend1ImageDir, "*.jpg"))
{
    using (Stream s = File.OpenRead(imagePath))
    {
```

```
        // Detect faces in the image and add to Anna
        await faceServiceClient.AddPersonFaceAsync(
            personGroupId, friend1.PersonId, s);
    }
}
// Do the same for Bill and Clare
```

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

Step 3: Train the person group

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

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

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

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

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

    await Task.Delay(1000);
```

```
}
```

Step 4: Identify a face against a defined person group

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

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

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

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

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

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

2.3 How to add faces

Step 1: Initialization

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

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

```
const int PersonCount = 10000;
const int CallLimitPerSecond = 10;
```

```
static Queue<DateTime> _timeStampQueue = new Queue<DateTime>(CallLimitPerSecond);

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

Step 2: Authorize the API call

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

```
FaceServiceClient faceServiceClient = new FaceServiceClient("Your subscription key");
```

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

Step 3: Create the person group

A person group named "MyPersonGroup" is created to save the persons. We also enqueue the request time to `_timeStampQueue` to ensure the overall validation.

```
const string personGroupId = "mypersongroupid";
const string personGroupName = "MyPersonGroup";
```

```
_timeStampQueue.Enqueue(DateTime.UtcNow);
await faceServiceClient.CreatePersonGroupAsync(personGroupId, personGroupName);
```

Step 4: Create the persons to the person group

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

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

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

Step 5: Add faces to the persons

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

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

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

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

2.4 How to Analyze Videos in Real-time

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

A Simple Approach

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

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

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

Parallelizing API Calls

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

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

```
        }
    }
}
```

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

A Producer-Consumer Design

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

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

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

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

```
            catch (Exception e)
            {
                output.Exception = e;
            }
            return output;
        }

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

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

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

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

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

Getting Started

The library contains the class FrameGrabber, which implements the producer-consumer system discussed above to process video frames from a webcam. The user can specify the exact form of the API call, and

the class uses events to let the calling code know when a new frame is acquired, or a new analysis result is available.

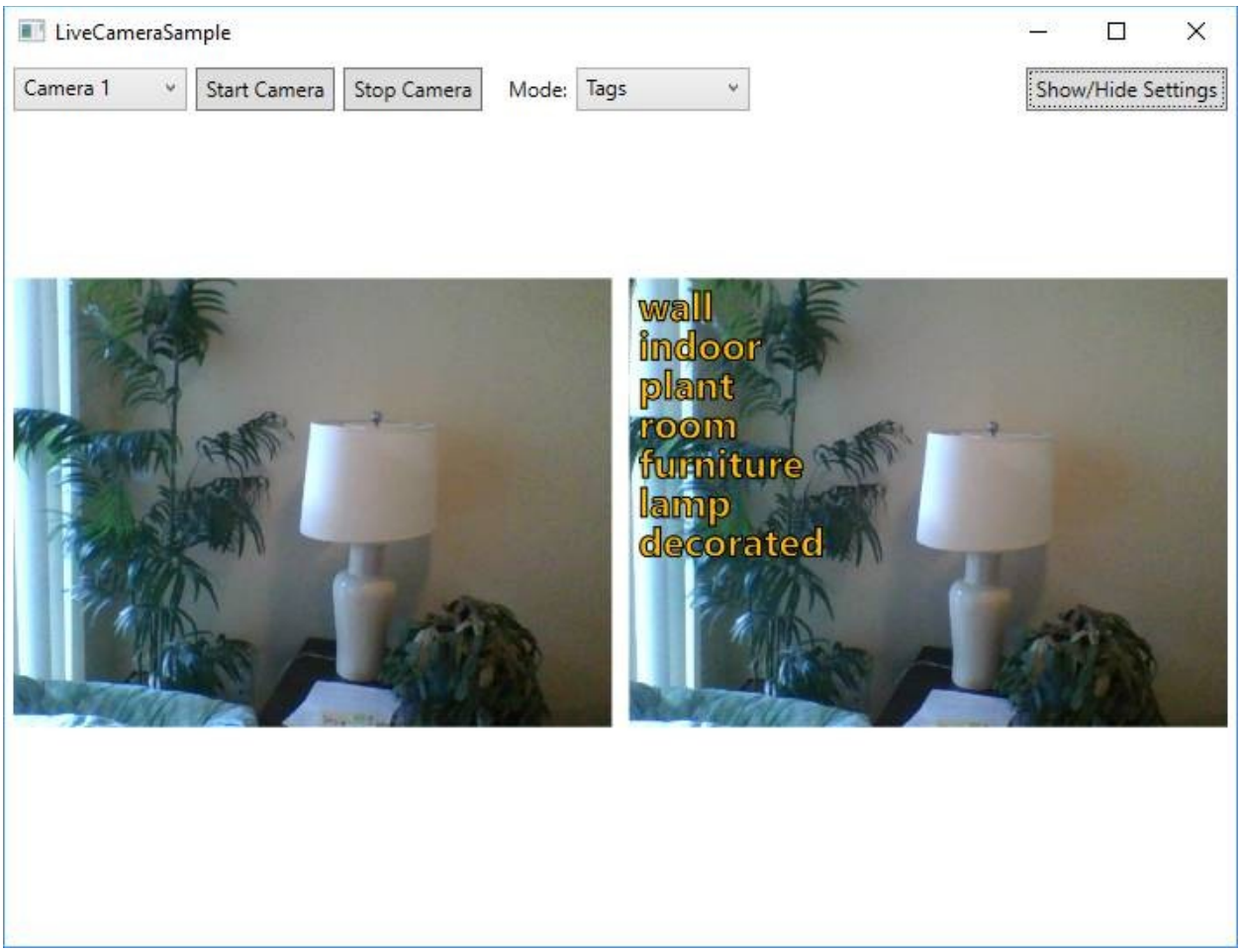
To illustrate some of the possibilities, there are two sample apps that uses the library. The first is a simple console app, and a simplified version of this is reproduced below. It grabs frames from the default webcam, and submits them to the Face API for face detection.

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

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

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

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

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

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

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

            // Wait for keypress to stop
```

```
            Console.WriteLine("Press any key to stop...");
            Console.ReadKey();

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

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

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



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

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

3.0 REST API FOR EMBEDDED SOLUTIONS

3.0 REST API FOR EMBEDDED SOLUTIONS

This paragraph provides information and code samples to help you quickly get started using the Sharp AI API with Python to accomplish the following tasks:

•Get HTTP as:

http://172.16.10.35:5000/api/detect_face/

•Respose face recongnition results as:

[{"detected": true, "image_url": "http://workaiossqn.tiegushi.com/a03c2864-1d3f-11e8-aa8a-50e5494ddd66", "time_stamp": "1519901973241", "face_url": "http://workaiossqn.tiegushi.com/a03c2864-1d3f-11e8-aa8a-50e5494ddd66", "face_id": "9f96f45c-1d3f-11e8-aa8a-50e5494ddd660001", "accuracy": 0, "style": "lower_head", "recognized": true, "labeled_name": "todo"}]

Before using API, you need to first create an account.

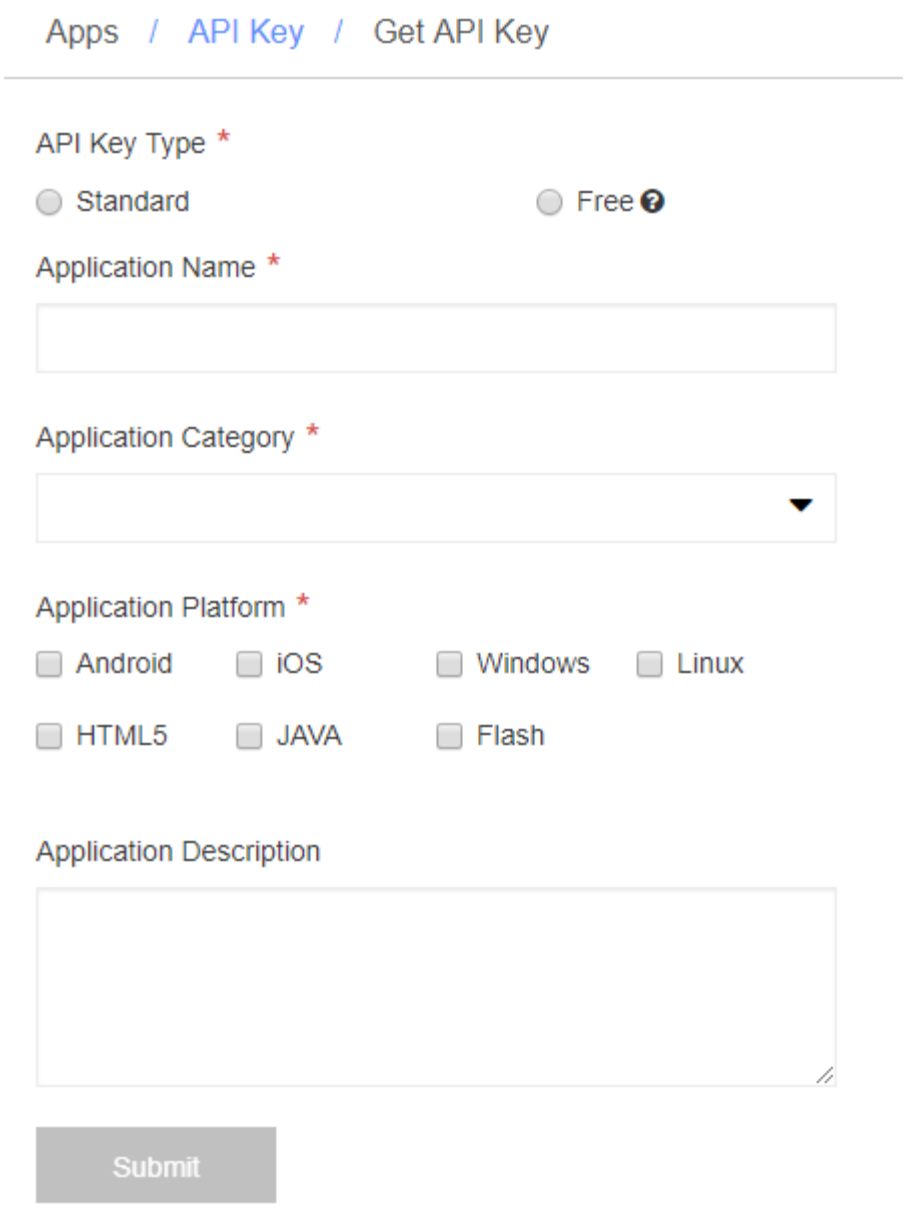
- If you already have an account, please skip this step.
- If you do not have an account, you can create one. Simply click on "Sign Up" button on top of “Dian Quan”APP, and then follow the guide to complete your registration.

After you have successfully created your account and logged into the console, you will see a page as below:

To make an API call, you need to create an API Key, this is your credentials to use API and SDK.

You can choose Free API Key or Standard API Key. If you would like to use Free API service, or to test Mobile SDK, you can use Free API key - It's totally free. If you want to use premium API key, or to

license Mobile SDK, please create a Standard API Key. You need to set up the name, category and platform of your application.



Now you've completed all the steps to get started. You can start using APIs following the API Reference now.

3.1 Return all result fields with Rest API

Use the Sharp AI API to detect faces in an image and return face attributes including:

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

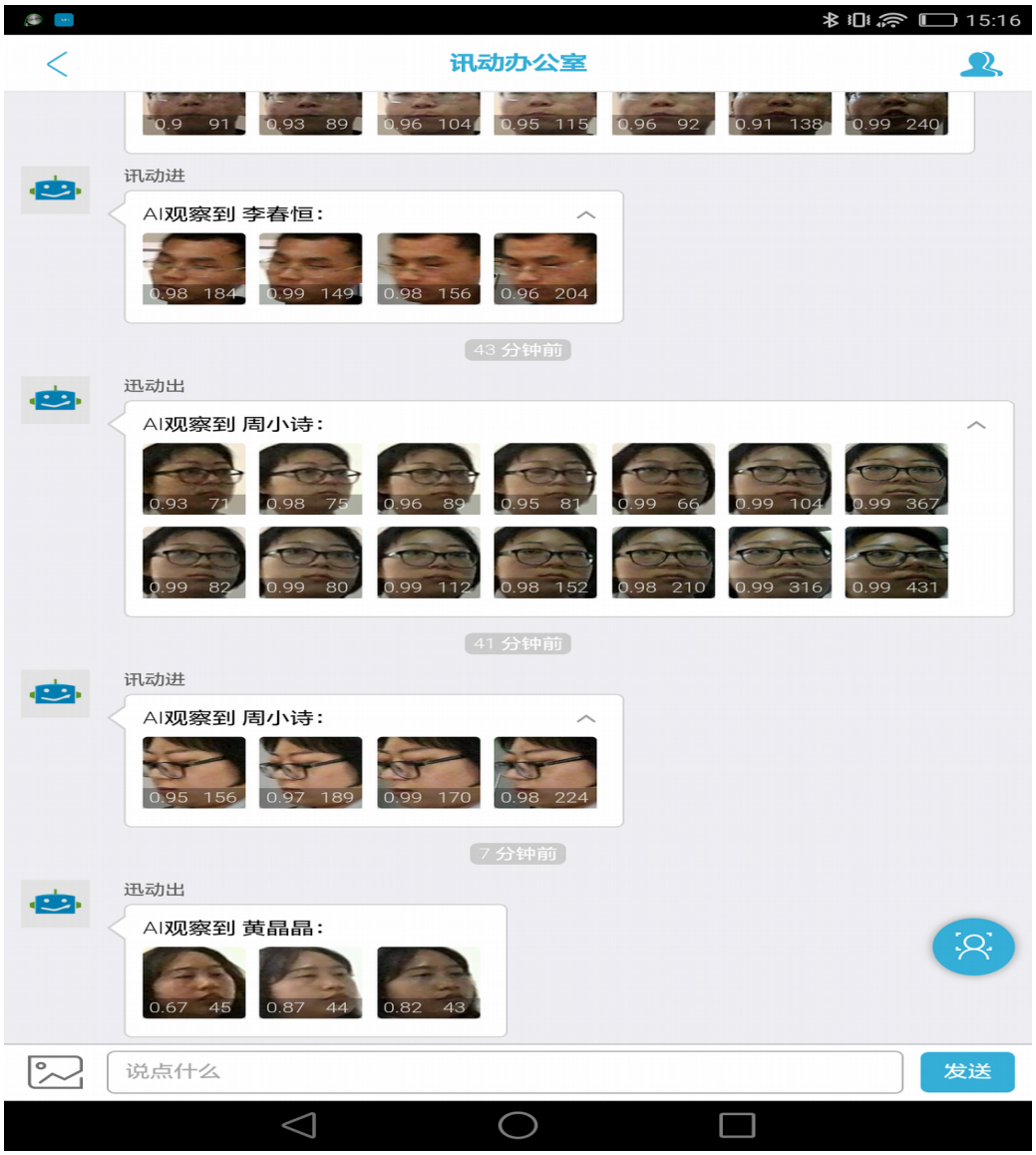
For example, we need all result fields, we give “@app.route” code as:

```
@app.route('/api/detect_face/', methods=['GET'])
def detect_face():
    json_result = json.dumps(get_resultQueue())
    return Response(json_result, status=200, mimetype='application/json')
```

We can obtain face result by get_resultQueue() function. Then, we Access the result data from http, as :

```
[{"detected": true, "image_url": "http://workaiossqn.tiegushi.com/a03c2864-1d3f-11e8-aa8a-50e5494ddd66", "time_stamp": "1519901973241", "face_url": "http://workaiossqn.tiegushi.com/a03c2864-1d3f-11e8-aa8a-50e5494ddd66", "face_id": "9f96f45c-1d3f-11e8-aa8a-50e5494ddd660001", "accuracy": 0, "style": "lower_head", "recognized": true, "labeled_name": "todo"}]
```

The face images are shown in “Dian Quan” app, for example:



3.2 Return the specified result field with Rest API

Now that we want the specific result field, and we give “@app.route” code as:

```
@app.route('/api/detect_face/', methods=['GET'])
def detect_face():
    json_result = json.dumps(get_resultQueue())
    data = json.loads(json_result)
    if data:
        data_image_url = data[0].get("face_url")
    else:
        data_image_url = json_result
    return Response(data_image_url, status=200, mimetype='application/json')
```

After all of these steps, we can obtain face_url as an example:

```
"face_url": "http://workaiossqn.tiegushi.com/a03c2864-1d3f-11e8-aa8a-50e5494ddd66"
```

And the URL result is: